EVALUATION AND TESTING REPORT

NP Studios – Team 8

Team Members

Lucy Ivatt, Jordan Spooner, Alasdair Pilmore-Bedford, Matthew Gilmore, Bruno Davies, Cassandra Lillystone

Approaches to Evaluation and Testing of the Final Product

Evaluation Approach – How we ensured our product met the final brief

From the beginning of the project, we knew to create strong product iterations we needed to produce high-quality requirements, keep them updated, and use them as the foundation for our testing. Because of this, as we inherited a new project in assessment 4, first we needed to carefully read the previous groups requirements and become familiar with them and their product as a whole. This would allow us to see what they had implemented and what still needed implementing so we would have a good understanding of how to move forward.

Therefore, we planned a meeting to go through the requirements as a group. This allowed us to compare their requirements with the brief, ensuring we added in any missing detail or missed requirements ensuring the full scope of the project was covered. For example, in SFR_FIRETRUCKS_STATS the requirement specified four Fire trucks even though the previous group had implemented six, so we updated the requirement to be accurate. Any other changes we made to the requirements were communicated with the customer to gain their approval before changes in the code were made.

This also gave us a chance to discuss the new features introduced for assessment four and the requirements we would need to cover them. Initially, we sat down and discussed how we wanted to implement these changes, for example, what specific power-ups we wanted and how the three difficulty levels would be implemented (what would make the game easier, harder etc.). Through this discussion, we made notes to be referred back to during implementation and formulated the new requirements. These included new user requirements as well as new matching functional/non-functional requirements. We then added the changes to the change tracker and assigned each one a branch and a change builder. Among the group, these changes were given priority over some of the other changes we wanted to implement. This is because missing out these new requirements would mean the product would not meet the brief, whereas some of the other changes were only minor improvements and less important.

Once our requirements list had been finalised (https://npstudios.github.io/files/FinalRequirementsA4.pdf), we then created our change tracker backlog based on what requirements were currently unfulfilled. As we progressed through the development phase, we ensured that as each requirement was met it was pull requested and highlighted as completed in our Requirements Met document (https://npstudios.github.io/files/RequirementsMetA4.pdf).

While this helped to ensure we did not miss anything from the brief, we also needed to ensure that the development did not breach the scope of the brief (other than smaller quality improvements already agreed upon by the client via email). Therefore, linking each change to its respective requirement on the change tracker (and within the implementation report) allowed us to stay on track to create a product in line with what the client originally imagined.

Finally, once the development phase was complete and all necessary requirements implemented to the best of our ability, we then needed to assess the quality of the code written. We therefore carried out extensive testing to ensure that the technical implementations of these requirements worked as expected and no edge cases ignored that could prevent requirements from being fulfilled or violate requirements such as UR_FUN and SNFR_SIMPLE.

Testing – How we ensured our code was of good quality

The testing approach that our team took for this assessment was more influenced by the time period than it was in our previous assessments. In assessment 2, the testing team tackled testing in a more traditional, linear, approach. This worked well for the first instance of the game due to it being a game that we built from the beginning and the ground up. However, in assessment 3,

while attempting to test in a similar method we indirectly fell into a more exploratory approach. This is due to the need to 'explore' the previous group's tests and code to ensure quality and test coverage. If there was code that was not tested, then we would tackle that section. This was one of the reasons we adopted exploratory testing fully for this assessment. Alongside that, this assessment we would be spending a large amount of time apart on Easter break.

The exploratory testing approach encourages individuals to tests specific sections of the code or requirements, allowing us to efficiently test sections on a person-by-person basis. We believe that this form of testing will cover more niche scenarios and more in-depth player-scenarios that a more traditional, scripted, testing approach would accomplish. Keeping these tests in order, we derived a testing table which has the following information: what requirement/s it covered, the specific approach to test said scenario, why it is useful, and who was testing it. Split into code testing and playthrough testing, it allowed the testing team to explore a specific case, whether that needed to be tested through the codebase or playing the game.

In addition to these standard testing methods, the team decided the need to have actual end-users to 'test' features. To accomplish this we conducted a small, focus group, play-through of the game with all said features. The demographic we focused on were those who would play the game when visiting York on open days. These range from current students, prospective students, and also parents. Their fresh approach gave the testing team invaluable insight that would influence changes within the final product. The feedback received made us aware of both bugs in parts of the game we were unaware of and also general player feel only a new player could provide. For example, we spent a long time ensuring that difficulty altered the correct aspect of the Firetruck and UFOs. However, we did not assess if, for example, 'easy' difficulty was actually easy for our end-user. While this would not be necessary for a traditional testing of applications, it is essential that we ensure this for our target audience. This is where our focus group play-through was key to informing our team with the necessary information. For example, player_ID #1 gave feedback that the completion of the game in medium 'mode' (difficulty) was a bit too hard. With their experience in video games, supported by a couple of other players with similar feedback, we scaled back the difficultness.

Overall, our approach to testing in this assessment was more holistic to ensure both the granular level of the game works but also the feel of the game is tuned to our users. From specific unit tests of methods, to specific game scenarios, and finally whole gameplay, we fully tested aspects of the game that matter to the final product. All of our testing documentation can be found here: (testing link).

Testing Quality

Quality testing is covering a unique state or requirement of the game, whether that be a specific use of a method or a specific situation a player might find themselves in. Accumulating these tests in sections of the code and game that are critical to the user will yield a strong, quality, test suite. This criteria for quality heavily influenced our thinking when devising tests. When assessing how to accomplish this we had two main factors: requirements and scenarios. Requirements being the statements that must be implemented and fully functioning within the final product. This was the base of our quality measurements as this fully described the product and all of its capabilities. Through thorough testing of all requirements, we would be able to have confidence that the core of the product is in order. To support the testing of the requirements we used scenario testing. This is where the testing team would look at a specific requirement and its pertaining code, and craft scenarios where they would be used. These could be either the ideal scenario of the requirement or strange niche scenarios to try to cover all possible cases a user might run into. This is where exploratory testing is key since it allows individual testers to explore a requirement with different scenarios.

For the inherited codebase, we ensured that the pre-existing tests met our own standards and adapted them if they did not, while also testing new additions or critical code the previous group missed. For example, the Entity package needed to have great coverage since it contains classes the player interacts with frequently. The quality criterion of playtesting is similar to code-testing, where we focused on specific and unique scenarios for each test. For instance, this could ensure all textures render correctly, that the Firetruck cannot drive over houses, the level difficulty gives the correct adjustments, etc. This form of testing could test either functionality or user-experience (or sometimes both). This is where we focused most of our effort, and to ensure quality we needed to ensure 'uniqueness' in our tests. That being the tests must evaluate the game in a way that no other test has. Through our approach of having pre-planned scenarios that testers can 'register' to test, it allowed for more effort into devising unique scenarios.

Modifications

The testing completed by the previous group for assessment 3 was heavily cluttered with extra features that were not used appropriately. As mentioned in the Implementation Report there was some needed refactoring of the codebase at the beginning of the assessment. For example, the testing technology utilized by the team, a LibGDX Junit testing skeleton, was very particular with the use of static calls. These were especially prevalent in constructors thus not being able to instantiate a class to test. To overcome this the previous group created additional constructors , only used for testing, which took out the line which caused an error. We found this bad practice especially for future developers; thus, we refactored the code to remove as many of these calls as possible. Even with this, a few remain which make those classes impossible to test without completely restructuring the game or implementing some form of bad practice.

We also modified some of the tests that the previous group had done. There were various naming violations, incorrect use of assert, and multiple methods being tested in one test **(Figure 1: Inherited Test Example).** However, the actual scenarios they tested were very sound which is why we elected to rework their tests as opposed to rewriting them. They used a white-box testing approach for their JUnit tests which we found perfect when paired with requirements.

We added many play-testing cases to ensure all of our new features and changes work as intended within the game, not just the code. We re-ran all of their previous playtests, altering or taking out those that needed that. We also increased the granularity of the tests, for example: Play Test ID 24, FlappyMiniGame, was a very general test that ensures it is 'fully functional', where we broke this up into smaller tests based on score. This way we ensure that no matter what score the player gets, the minigame is functioning and the developers and client know that we ensured that.

All together these testing approaches, quality assurances, and modifications resulted in a testing suite that strongly ensures the quality of the product and its code. This can be visualised with the traceability matrix, found here (https://npstudios.github.io/files/TraceabilityMatrixAllTestA4.pdf), which has each requirement supported multiple times with different tests. Alongside that, our play-though test feedback gave us a deeper understanding of what our audience are thinking, and you can see the changes we made here (https://npstudios.github.io/files/UserFeedbackAndResponseA4.pdf). Overall, we believe that our testing has thoroughly tested our requirements and all scenarios that they could be used in.

Meeting Requirements

Our final product meets the vast majority of the requirements, with exceptions being a small number of the requirements with a 'may' priority. We have gone through the final requirements thoroughly and are able to provide evidence of meeting 90% of them.

One of the requirements which we have unfortunately only partially met is **UR_GAME_TIMER**. This requirement states that the game's length shall be decided keeping in mind the target audience i.e. open days attenders and is based on the timer that is triggered following the first attack to an ET. In our game, the timer begins as soon as the game starts, rather than at the time of the first attack on a fortress. The reason for this is because when we inherited the game it was implemented this way, and sadly this specific detail in the requirement slipped through the net. We could have quickly implemented it differently to fully suit the requirement however we felt as though it was not a huge issue; the player is likely to initialise the first attack very soon to start the game anyway.

The other requirements, making up the 10% we missed, were related to the game being ported to mobile, being playable with a controller, and featuring a colour accessibility toggle and an open day toggle. Each of these were not hard requirements, they were other features to improve the overall quality of the game which is why we did not prioritise them.

We did not attempt to port the game to mobile or use a controller to play. We do know that libGDX is an engine which makes it fairly easy to port games to mobile, therefore it could be possible with more time to implement. However, as the focus for the game is primarily open days, the game would likely only be opened on the lab computers, so we did not feel like this was a priority.

As for the colour accessibility toggle, we feel as a group we do not know enough about the colour schemes that would be provided under an accessible nature. This means to implement this feature would require lots of research beforehand. As it is not a core requirement, just an extension, we believed there were better uses of our time to ensure the final product hit the brief fully, first and foremost.

Finally, the open day toggle was an idea we all had that, we agreed, was always only going to materialise if we had sufficient time after feeling highly confident that the game fully met the brief. Due to the current climate and having to adapt to working 100% remotely, it has been much more of a struggle to get the core functionality of the game finished as it is, meaning this idea failed to come to fruition.

You can see our final requirements, which show changes and additions we have made this time around, here https://npstudios.github.io/files/FinalRequirementsA4.pdf. You can also view a table which shows which requirements we have met and which ones we haven't here https://npstudios.github.io/files/RequirementsMetA4.pdf.

Appendix

Figure 1: Inherited Test Example

```
public void testRefill() {
    testTruck.addHealth( x: 2);
    testTruck.replenish();
    assertTrue( condition: testTruck.getHealthPoints() !=  102);
    assertFalse( condition: testTruck.getCurrentWater() ==  302);

    testTruck.applyDamage(10);

    assertTrue( condition: testTruck.getHealthPoints() ==  90);

    testTruck.replenish();
    assertTrue( condition: testTruck.getHealthPoints() ==  92);
    assertFalse( condition: testTruck.getCurrentWater() ==  304);

    testTruck.addWater( x: 96);

    assertFalse( condition: testTruck.getCurrentWater() ==  400);

    testTruck.replenish();

    assertTrue( condition: testTruck.getHealthPoints() ==  94);
    assertFalse( condition: testTruck.getCurrentWater() ==  400);

}
```