TESTING REPORT

NP Studios – Team 8

Team Members

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

Testing Methods and Approaches

In this section we will cover the methods and approaches the team has taken to tackle testing our game thoroughly. In general, as mentioned in the other reports, we split our team into speciality sub-groups. These sub-groups were to focus on specific parts of the game and the project. Thus, our testing strategy applies to those in the sub-group, with frequent discussion with the development team. However, this all starts with a strong test strategy, which is essential for a well-structured testing process.

For our strategy the whole team worked together to decide which roles go to whom; what platforms and frameworks are to be used; the risks and mitigation; and the schedule that testing will run on. The schedule was complex due to the ideal tests practice - which is to write tests before implementation. This allows for the development team to test their code to ensure it follows the overall game requirements. However, we did not want the development team to have to wait for platforms and frameworks to be integrated and then tests written. We decided that the tests will be created alongside the code developed. The platforms and frameworks that we used for the testing were JUnit and Mockito. JUnit is a very popular framework that is robust and well tested. This gave the group assurance that the passing and failing unit tests were not flukes. Then for mocking we decided to use Mockito, it is widely and professionally used, which gave the team confidence. For the risks, the largest was the time to learn testing to an effective level. Not all members of the testing team had done Unit testing, and none had done mocking. The only mitigation to this was for the testing team to take a week to focus on honing their testing skills. After, we came back and started working on the testing the game. Another risk was new, unexpected, code that could be implemented by the developer team. This would be managed via two methods; regularly meetings between the two teams, and for every pull the testing team member did - to read the changes made through IntelliJ's merge system. Overall the testing strategy worked well with the small team that we had for testing.

The overall goal of the testing team was to ensure all requirements of the game worked as intended. Thus, the functional testing approach was appropriate, since it allowed us to directly check these requirements. This is done through creating input data based on the functions purpose, entering it, and checking if the output is as intended by comparing to the expected outcome. This is possible by using JUnit and Mockito which allows the team to test outcomes in an environment that is solely for a specific function. The key is to isolate each requirement, ensure it is testing only one thing, make it repeatable, fast, timely and self-validating (this is where JUnit is especially useful). Then to ensure all possible combinations and types of data is accounted for, we used boundary and parameterised testing. This is where we see the boundary of input data for a function, for example, and test around said boundary. We test right outside the boundaries, on the limit, just inside and an average value. This is not needed for all functions, this would take too much time, but it is crucial for functions that are vital to the game.

This testing, in general, took a structural (white box) testing approach to all its testing. Since there was overlap with the testing team and the development team, this was a given. However, that was not the only reason, as there are many advantages to this testing method compared to 'black box' testing. First and foremost, it allows for very concise and thorough testing, due to the team being able to analyse the code. We often find ourselves trying to break the code with our testing, finding cases that the development team did not cover for. This testing method also allowed the team to start the testing process for some functions that the development team had not written code for yet. While this testing can be more resource intensive, in terms of time and people required to cover the needed tests, it does allow for a strong tested code base.

Brief Report on Actual Tests

In this section we will be covering some concrete examples of our testing, explaining what they do, cover and why it is useful. We will also go over the results produced by said tests, the overall statistic of testing in the project. Then, discussing what in the testing part of the project could be improved in the next iteration, why they would be useful, and what we think went well and should continue.

First, we will cover the class/sprint testing that was completed on the game. This was most of the testing done by the team and is the most comprehensive. This was decided since this part was where libGDX had the least control, thus had the biggest chance for errors and bugs. The most challenging aspect of this set of testing, was the ever-changing code base. This caused the test team to need to rewrite tests, move tests, change constructor orders, etc. While the code base changes did result in a cleaner product, it did limit the amount of testing that was created before said changes to stay after.

We started testing from the top of the UML Class Diagram, with the Entity class, a simple class with no complex functions. It is, however, the class that all other in-game objects are extended from, so it was imperative to test it thoroughly and correctly. One of the biggest areas of concern was the *setPosition()* function, as it is called extensively through the game, from setting objects in the game, positioning graphics, and moving the Characters. Moving Characters being so vital to the came further emphasised how critical the Entity class is.

The tests we ran on *setPosition()* where to check both basic functionality and niche cases. For example, **test_1.1** tests the standard change of position with an increment of 100 by 100. This test has a standard input data to ensure the function does as described for regular data inputs. We also check if *getTopRight()* of the test entity was in the correct position, since this will be important when implementing the collision detection function and attack range calculations. Then we also tested a boundary data input for the *setPosition()* function, with testing the bottom of the map, (0,0). This test, **test_1.2**, was to ensure that there were no unexpected outputs with a 'non-regular' input. We tested out of boundary values as well, such as negative numbers, since we cannot allow Entities outside of the allotted game board space. To check we used a more primitive method of a try catch since most team members understood this better than Lambdas, which are supported in JUnit5. Most of these tests passed by the end of the project, however the negative number check was a failure. This is because the implementation of catching negative numbers would cause the game to crash. Due to later implementation, Projectiles would end up going to negative position before we called *dispose()* on the object. In the next iteration the test should not expect an Exception but test if the objects are disposed when in any range outside of the game.

The example above took a **boundary** testing approach, in this approach we used a **structural** testing. As mentioned above, most, if not all, of our testing was done in a **structural** testing approach. For example, in the testing for the Unit class, in **test_2.5.2** , we test the capping of the amount of damage a Unit takes. For this we knew the code behind *takeDamage(),* which in result used *setCurrentHealth().* Through this, we knew that *setCurrentHealth()* allowed for a negative health value. This in turn would break the *isDead()* function which initially checked if *currentHealth* was equal to 0 and if it was return true. Since it only checks for '0', we know having it go negative would cause the Unit to be alive when *isDead()* is called. With this theory made through analysing the code directly, we devised a test to ensure this would cause an error. Then once this failure was fixed this test can ensure that the fix was implemented correctly.

There were cases where the testing team ran into problems, where we did not know how to test certain requirements with the previous method, including **UR_start_screen**, **UR_select_level**, **UR_pause**, and **UR_music.** To test these User Requirements, we conducted manual tests in which we ran through the game, and manually checked that these User Requirements were working. For example, with UR_music, when manually testing the music, the music started when opening the game and continued while moving states as required. The music also speeds up at 15 seconds left in the game, to encourage a sense of urgency in the user. However, when trying to mute the game we found and fixed a bug that caused the music not be muted until exiting the settings page. This would clearly confuse the user who is trying to mute the music, but it is not stopping.

Another key example where manual testing ensured User Requirements were met is UR_level_select. We needed to test that level select would 'unlock' the new level when the previous level was completed, and not when failed. We ran through the game several times, all with different cases to check all variance. We had a test that ran through the whole game, passing each level, and here the level-select worked perfectly. Then, we passed the first level but then failed the second to ensure the last level would not unlock, which it did not. Then we failed the levels in different ways (e.g. time ran out or fire trucks destroyed) to ensure in all cases of level failure, only the currently unlocked levels were available. We then quit the level partway through which is a niche test to make sure no levels were unlocked in that case. After testing these cases, level selected worked exactly as needed and described in the User Requirements. The team felt confident in these testing methods, however for next testing iteration we wanted to find more concrete testing states, in which we could use mocking.

A vital aspect to unit testing is isolating the test to only the unit that is under interrogation. As the results of said tests need to indicate a failure, error, or bug in the unit and not its dependencies. Our team understood the principles but had a tough time using mocking and identifying correct units to mock over standard JUnit tests. However, near the end, a very key and new function had a large dependency. The *truckInRange()* in the Alien class will check if a Firetruck instance is in range and sets the aliens target accordingly. The method takes an ArrayList<Firetruck>, which it then loops through and checks if any are in range and sets the Aliens target to the instance with the lowest health.

This method has a large dependency on the Firetruck class, since it calls Firetruck's inherited methods such as *getPosition(), getCurrentHealth(),* and *getTopRight().* We did not want a possible failure in any of these to cause a failure in the tests we needed to write. Thus, we mocked the Firetruck class using Mockito. Using *when()* and *thenReturn()* allowed the team to force a response from Firetruck method calls. For example, for test_7.1.1 (Figure 1: test_7.1.X), we mock the responses for *getPosition(), getCurrentHealth(),* and *getTopRight().* The mocked values were to ensure the Firetruck instance is in range of an Alien instance, to allow it to be set as the new target. Then for the other test of *truckInRange()*, we mocked Firetruck again to ensure it was out of range of an Alien instance test_7.1.2, and to ensure another Firetruck instance was the current target and had no health (Test ID: 7.1.3). All these mocked tests passed without the dependency of the Firetruck class, ensuring it works as the game needs it too. If these tests were to fail in another iteration of development, the testing and development team can be sure it is because of a fault if *truckInRange()* and not in the Firetruck class.

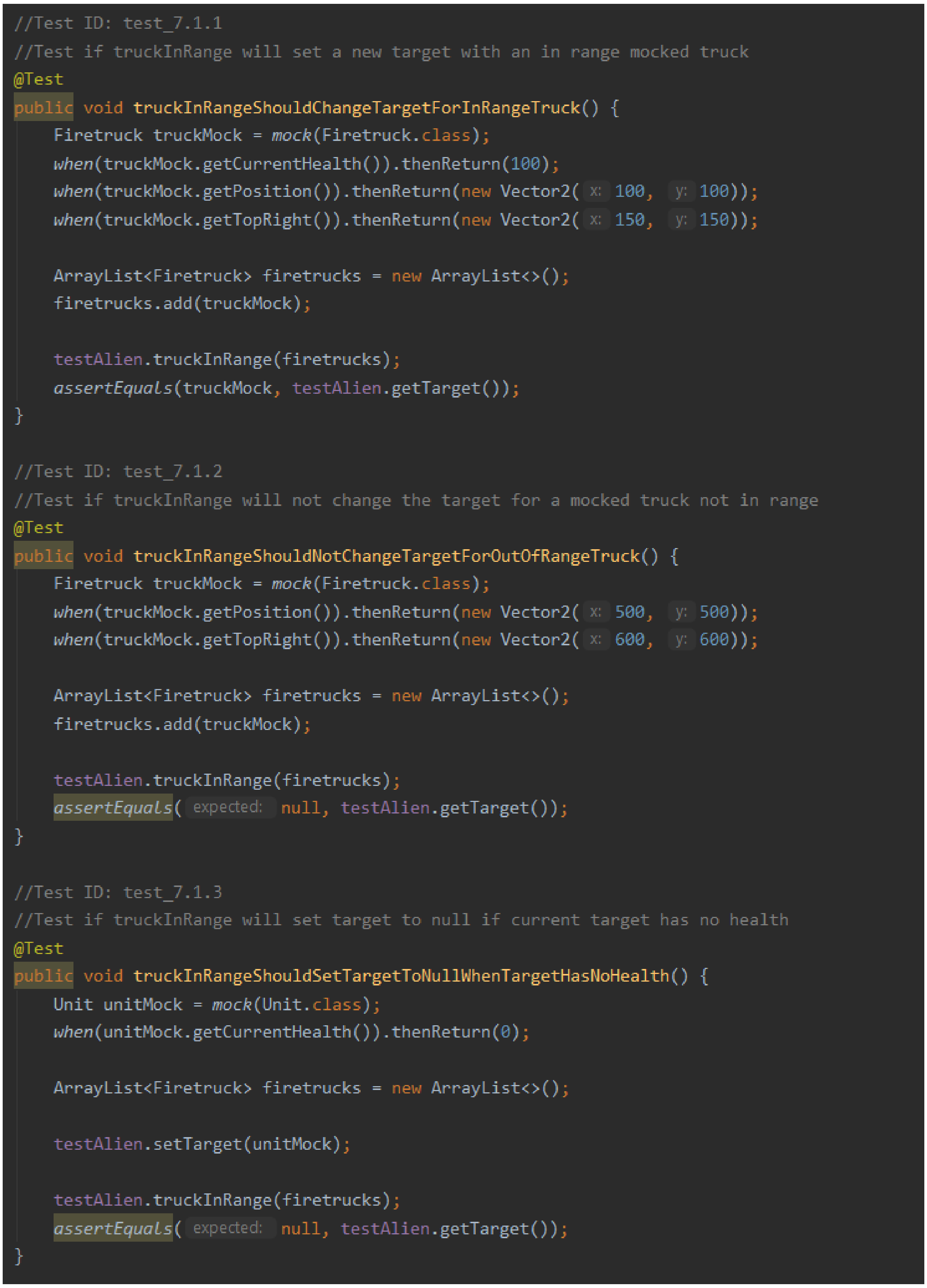
Overall, the testing was fairly complete, with most line coverage over 60% and method coverage as well. With some classes being lower due to mostly being getter and setter type functions. We were lacking in tests for the State classes (other than manual) which we would have wanted to fix in the next iteration, with more mocking driven testing. The testing done by the team gave the game a more concrete code base that will satisfy hopefully satisfy the client.

Testing Design and Evidence of Testing

View all of our testing material: https://npstudios.github.io/testing/

Appendix

Figure 1: test_7.1.X



```java
//Test ID: test_7.1.1
//Test if truckInRange will set a new target with an in range mocked truck
@Test
public void truckInRangeShouldChangeTargetForInRangeTruck() {
    Firetruck truckMock = mock(Firetruck.class);
    when(truckMock.getCurrentHealth()).thenReturn(100);
    when(truckMock.getPosition()).thenReturn(new Vector2( x: 100,  y: 100));
    when(truckMock.getTopRight()).thenReturn(new Vector2( x: 150,  y: 150));

    ArrayList<Firetruck> firetrucks = new ArrayList<>();
    firetrucks.add(truckMock);

    testAlien.truckInRange(firetrucks);
    assertEquals(truckMock, testAlien.getTarget());
}

//Test ID: test_7.1.2
//Test if truckInRange will not change the target for a mocked truck not in range
@Test
public void truckInRangeShouldNotChangeTargetForOutOfRangeTruck() {
    Firetruck truckMock = mock(Firetruck.class);
    when(truckMock.getPosition()).thenReturn(new Vector2( x: 500,  y: 500));
    when(truckMock.getTopRight()).thenReturn(new Vector2( x: 600,  y: 600));

    ArrayList<Firetruck> firetrucks = new ArrayList<>();
    firetrucks.add(truckMock);

    testAlien.truckInRange(firetrucks);
    assertEquals( expected: null, testAlien.getTarget());
}

//Test ID: test_7.1.3
//Test if truckInRange will set target to null if current target has no health
@Test
public void truckInRangeShouldSetTargetToNullWhenTargetHasNoHealth() {
    Unit unitMock = mock(Unit.class);
    when(unitMock.getCurrentHealth()).thenReturn(0);

    ArrayList<Firetruck> firetrucks = new ArrayList<>();

    testAlien.setTarget(unitMock);

    testAlien.truckInRange(firetrucks);
    assertEquals( expected: null, testAlien.getTarget());
}
```