Software Testing

Mario

GROUP 28

Joseph Frankland

Anna Singleton

Saj Hoque

Leif Kemp

Shi (Lucy) Li

Hugo Kwok

Part A

Methods and Approach

When planning to prepare our software testing we analysed our options and came to the conclusion of using unit testing. We considered alternatives such as integration testing however we quickly deemed this as too complicated and excessive for our project bounds. From our research we found that unit testing was one of the most common methods and is relatively easy to implement which also contributes to how familiar our developers are with unit testing. Once this was established, we examined the range of unit testing frameworks, and from this we came to the conclusion of using JUnit 4. We considered JUnit 5, however we disregarded this as JUnit 4 more easily integrated with our libgdx test runner which is itself written for JUnit 4. In addition to this JUnit 4 is the industry standard for java which furthermore strengthened our decision in this sector.

Our approach was to test function by function ensuring that each one was working as intended. It was worked on by several developers to make sure all main criteria of the function were checked. We felt this method was appropriate because we are working with a smaller project and henceforth felt as though it was feasible to cover all bases this way. Before implementing this, we were aware that not all classes would be testable in an objective way via the unit test, therefore we tried to capture all of the cases that were testable and for those that we deemed were unsuitable, we closely observed these functions and carried out thorough manual testing. Although this method was tedious, we felt it was appropriate because it ensured that all our classes and functions were accounted for. Because of this, in terms of unit testing we did not have great code coverage which was lacking because there were a lot of areas of the code that weren't really unit testable such as update functions, UI assets or random generation, but of those that have been covered by unit testing have been deeply implemented which is obviously supplemented with manual testing for those without.

Also in our approach to software testing, we were heavily associated with the usage of gradle to assist in running our unit tests. Using gradle as a build system allowed us to set up projects that compiles and runs unit tests using JUnit 4 which was beneficial considering gradle has native support for JUnit 4.

Part B

Testing report

Our testing had about 35% coverage of the code. While this number may seem low, a large majority of the code was written within render functions which cannot be unit tested, or update functions which ran in real time, making them unsuitable for unit testing.

The Code Coverage Report in the deliverables section of our website highlights which classes we have unit tested and to what extent, and which we haven't, those with high coverage %'s we considered a high priority when it came to unit testing, and thus most of our efforts were dedicated to those, and those classes with a low % we considered to be mostly tested via manual testing because of either complexity or because it was tied to rendering/real-time behaviours. Some of the testable code we couldn't add to the unit test due to time constraints. This uncovered code was, however, tested manually in as many ways as possible.

We tested most of the key features such as the individual components for combat with the colleges and the points/gold system. Once the testing of the individual components of combat had been passed we did some manual testing to make sure all of these elements worked together properly. This was the method we used for integration testing of our more complex features. Other features such as entity spawning for powerups, we would test the correct amount spawning in a unit test and then we would test the random distribution manually by playing the game multiple times. For things we couldn't test such as the UI and AI elements all we could do was manually test them. So for these manual tests we did the most we could to assure ourselves that there were no errors within these parts of the code. A higher level of scrutiny for these tests was vital as they were something that couldnt definitively be proven to work all of the time, this level of detail was set aside mainly for testing the UI functions as well as testing the AI of enemy ships and colleges.

Part of the time constraints can be attributed to optimising the unit tests, as it took approximately 70 seconds to run 50 unit tests pre-optimised, which was a massive issue when we needed to rewrite tests or add new unit tests to a class. Before the optimisations were implemented, we instead shifted focus onto manual testing due to the length of time required to run tests. The main culprit was the GameScreen class, as generating the worlds took approximately 2 seconds depending on the difficulty of the game, and we generated a new GameScreen for each unit test.

This was remedied in 2 ways:

- Tests were rewritten to generate only 1 instance of GameScreen during the initialisation of the class (more when necessary) and we implemented additional functions to the GameScreen to reset the necessary states.
- A new ‘difficulty’ was added; TEST, which generates only 10 obstacles and 10 power-ups compared to the 100-500 mines and 10-50 power-ups generated with the other difficulties. This difficulty is only accessible via code and cannot be enabled in game, this massive reduction in entity count allowed us to reduce the instantiation of GameScreen objects considerably.

After implementing these optimisations, the testing was reduced from ~70 seconds to run 50 tests to ~10 seconds, which made writing new tests and rewriting old ones much easier to carry out.

Unit Tests written per class, all of these tests pass unless otherwise specified:

EntityTest:

- This testing class is used to test behaviour that all Entities have; setting positions, getting hitboxes, testing that positions and hitboxes are synced, testing collisions.

CollegeTest:

- This testing class is used to test college behaviour; taking damage, spawning ships based on conditions, and shooting behaviour.
- Shooting at entities specifically was not included in the testing, because it is too complex to unit test, this was manually tested with the conditions for shooting being tested instead.

EntityShipTest:

- This testing class is used to test behaviour that the player ship adheres to; setting health based on difficulty, setting velocity
- It also defines some behaviour that both the player and AI ships adhere to; testing college affiliation, taking damage, repairing.
- Combat was not unit tested, this is because player and AI ships use different shooting behaviour and is too complex to set up via unit test, thus this was manually tested instead.

EntityAIShipTest:

- This testing class is used to test behaviours that are specific to the EntityAIShip objects; setting health based on difficulty.

EntityCannonballTest:

- This testing class is used to test cannonball behaviour; moving based on given Vector2 directions, setting itself for removal based on distance travelled, damaging appropriate entities.

MineTest:

- This testing class is used to test mine behaviour; colliding with the player.

- We did not test collisions with other entities as they do not apply, the mines will only collide with and damage the player.

GameScreenTest:

- This testing class is used to test the game's behaviour; testing that storms slow entities, plunder is added/deducted
- Other game behaviours were deemed too complex (i.e testing that storms proc) and were instead manually tested.

CollegeManagerTest:

- This testing class is used to test the CollegeManager; testing that colleges generate, testing that college names can be fetched, and that colleges can be set as friendly.

PlayerPowerupTest:

- This testing class is used to test that the player can have powerups applied to them appropriately; that power-ups can be applied and decayed based on time passed, they can be collided with, and applied properly.

PowerupManagerTest:

- This testing class is used to test that power-ups are generated properly; testing that power-ups are generated and each type is distributed equally.

ShopTest:

- This testing class is used to test that the shop and the relevant purchases are applied properly; that the selection can be increased/decreased properly, purchases can be purchased/not purchased based on plunder, one off purchases can't be repurchased, storm is skipped properly, and that player based purchases are applied properly.

WorldTest:

- This testing class is used to test that the world and its entities can be interacted with; test that entities of generic and special types can be added and cleared properly.

LoadSaveGameTest:

- This testing class is used to test that the game can be saved and loaded appropriately.

Part C

Please refer to the website.

These may be found on the deliverables tab - in the New / Edited Deliverables section, under the name of “Unit Testing Report” and “Code Coverage Report”