Testing Report

Our team decided to do system testing using black-box tests, unit testing (white-box), and peer reviews (static tests). We felt this was sufficient since we were developing a small system which was not safety critical. Testing was still very important to us as it improved the quality of our code, increased the confidence of the rest of the team, and likely the stakeholders too.

We used test driven development [1] when suitable which ensured our implementation satisfied the requirements. This was done by coming up with tests before implementing the software that was to be tested, giving faster feedback and helping developers understand what they need to code. Most of these tests were black-box as they were easily written up compared to unit tests. After implementation other tests, both black-box and unit tests, were added to try for as much coverage as possible. We ran these tests as we went and again when we were happy with our implementation to give this testing report. Our requirements testing [2] document creates links between tests and requirements to allow us to easily show what requirements have and haven't been met with easy traceability. A lot of the tests cannot be linked directly to the requirements as a lot of the features we are testing are implicit.

The white-box testing was done using unit tests, more specifically JUnit 4. We noticed quickly that our unit tests were useful for regression testing (checking the software still functions as expected after changes are made). Our implementation included game logic in some classes that implement Screen, notably our Level class. This made it very difficult to test with JUnit because it had to be rendered to function correctly. For this reason, mostly black-box tests cover these classes. We used IntelliJ which had built in the option to "run with coverage" which allowed us to measure the coverage of our tests. This gives both the team and stakeholders an idea of how thorough our testing is. However, this value cannot be taken at face value because our black-tests will not contribute to this metric.

The black-box testing was done by running the game and making sure it acted as expected for that test. White-box tests are preferred over black-box since errors encountered during black-box testing can be difficult to reproduce and the amount of coverage cannot be easily measured. Luckily since LibGDX itself has been thoroughly tested [3] we were able to focus on testing what we added on top of what LibGDX offers to developers.

We created a GitHub project board to assign and manage tasks, one column in the project board was "To Review". When a task that involved coding was completed it was added to this section and then another member of the team had to verify the code. If they agreed with the implementation they moved it to the "Done" column, otherwise they moved it back to "In Progress" column and told the person who originally moved it why. This made sure all code was double checked and up to standard. All code added to the git repository was reviewed this way, greatly reducing the amount of poor code in the final software system. This is a form of static testing since the code does not have to be run. This allowed us to test code before it is fully implemented which was very useful for more difficult problems that couldn't be solved straight away.

Test Statistics

There are separate files for testing evidence for white-box and block-box tests but both use a tabular format. Each test has an ID to make it easily traceable to requirements, a description to give the reader a better understanding of the test, a section to say whether it passed or failed, and another column for any additional comments. With the exception of tests 7.2 and 7.3, tests that cover unimplemented requirements have not been included. See the references section of this document for links to all the testing material.

24 of 27 of our black-box tests passed [4]. The three tests that failed are noted below with reasons why.

- 7.2 - Save button creates a text file containing the current game state.
    - Despite this feature having a test, we have not yet implemented it. However, we still felt that it was important to include this tests as a non-functioning save button still exist in our game.
- 7.3 - Load button gives you the option to load any of the save files.
    - Again we had not implemented this feature but a load button still existed in our game.
- 10.1 - The player faces in the direction of the mouse pointer at all times.
    - This test failed as when the player is attacking it will not change direction. We didn't attempt to pass this test because we liked that it stopped the player from holding down the mouse button to attack as fast as possible.

We think that our black-box tests are suitable as they cover all parts of the game that we couldn't test in a more controlled environment. Black-box tests by nature can be difficult to reproduce but we think that due to the detail in our Black-Box Testing Evidence document [$] other people will be able to reproduce our tests and get the same results.

33 of 33 tests passed of our white-box tests passed [5]. We also ran the tests using IntelliJ's built is code coverage runner. We were happy with our unit testing coverage because what was not tested in this way was covered by black-box tests which was able to system and integration tests also.

We struggled to test classes that used LibGDX's Screen class with unit testing, these included Level, TownLevel, Halifax Level, CourtyardLevel, Loading Screen, Menu Screen, SelectLevelScreen, and TextScreen. We aimed to test these classes more thoroughly with black-box tests as an alternative.

We felt ZeprInputProcessor didn't need testing because it was, for the most part, just a standard LibGDX InputProcessor which has been tested by the LibGDX team. What we did add was a mouse pointer position that is used for the player direction and player attacks, which was tested (tests 10.1 and 10.2) and passed. We have attempted to get the best test coverage possible for all other classes.

Coverage: GdxTestRunner

35% classes, 43% lines covered in package 'com.geeselightning.zepr'

Element	Class, %	Method, %	Line, %
desktop	0% (0/1)	0% (0/1)	0% (0/6)
tests	83% (5/6)	90% (39/43)	97% (247/254)
Character	100% (1/1)	72% (8/11)	55% (30/54)
Constant	100% (1/1)	100% (1/1)	100% (1/1)
CourtyardLevel	0% (0/1)	0% (0/3)	0% (0/9)
HalifaxLevel	0% (0/1)	0% (0/3)	0% (0/10)
Level	0% (0/3)	0% (0/19)	0% (0/169)
LoadingScreen	0% (0/1)	0% (0/8)	0% (0/12)
MenuScreen	0% (0/3)	0% (0/13)	0% (0/39)
Player	100% (1/1)	77% (7/9)	75% (40/53)
PowerUp	100% (1/1)	80% (4/5)	82% (14/17)
PowerUpHeal	100% (1/1)	100% (3/3)	90% (10/11)
PowerUpImmunity	100% (1/1)	100% (4/4)	100% (15/15)
PowerUpSpeed	100% (1/1)	100% (4/4)	100% (15/15)
SelectLevelScreen	0% (0/8)	0% (0/23)	0% (0/121)
TextScreen	0% (0/2)	0% (0/11)	0% (0/33)
TownLevel	0% (0/1)	0% (0/3)	0% (0/10)
Zepr	0% (0/1)	0% (0/2)	0% (0/19)
ZeprInputProcessor	0% (0/1)	0% (0/8)	0% (0/25)
Zombie	100% (1/1)	66% (2/3)	61% (13/21)

References

[1] Agiledata.org. *Introduction to Test Driven Development*. [online] Available at: http://agiledata.org/essays/tdd.html [Accessed 16 Jan. 2019].
[2] Requirement Testing. [online] Available at: https://drive.google.com/a/york.ac.uk/file/d/1KlEe5ZuO5Uau9hi8U8sXQfrEY0Y8ByP-/view?usp=sharing [Accessed 20 Jan. 2019]
[3] GitHub. (2019). *LibGDX Testing*. [online] Available at: https://github.com/libgdx/libgdx/tree/master/tests [Accessed 16 Jan. 2019].
[4] Black-Box Tests. [online] Available at: https://drive.google.com/file/d/1CoiAPB8tGS1ezoLpUOaC0VSnClbNrzlD/view [Accessed 19 Jan. 2019]
[5] White-Box Tests. [online] Available at: https://drive.google.com/file/d/1y_Ktkox5_NS9L44yocPB24uHCiNZp9_U/view [Accessed 19 Jan. 2019]