Module	SEPR
Year	2019/20
Assessment	3
Team	Mozzarella Bytes
Members	Kathryn Dale, Emilien Bevierre, Ravinder Dosanjh, Elizabeth Hodges, Daniel Benison, Callum Marsden
Deliverable	Testing Report

Software Testing Report

Testing software prior to release is essential to ensuring users avoid negative experiences as a result of errors or defects when using the software, and although it is difficult to verify the software can handle *all* possible eventualities, the employment of various testing models can enable developers to minimise and mitigate these faults before they impact user experiences, and give feedback to assist in making decisions around further development around quality.

The ISO/IEC/IEEE 29119 standard[1] proposes that testing is a process; it should be preplanned, but continue throughout and beyond the development of software. For this project, this model is adapted such to make it easier to follow for the smaller number of developers involved, and simpler given the relatively minimal complexity involved with the scale of the project when compared to that which the standard was designed.

A key principle to the testing model used is that it is *agile*, and so can be used in conjunction with the SCRUM-based development cycle. One recommendation for agile testing is that as much of it should be automated as possible[2]. By using Java as the primary development language many potential faults are removed through the use of *static typing*. One particular difficulty with using the LibGDX framework and libraries is that it is not trivially compliant[3], [4] with JUnit5[5], a white-box style unit testing framework for Java projects. It instead requires significant additional configuration and a *headless* implementation due to LibGDX's dependency on OpenGL for rendering components.

It can additionally be difficult to ensure code reliability when developing in an agile way, particularly without unit tests, as an increased number of smaller updates can lead to a higher number of opportunities for the overall software code to fail (such as through concurrently modifying a feature and its dependency). Regular regression testing must therefore be performed throughout the project to catch errors or bugs arising from these code updates.

Not all errors can be caught through regression testing, and therefore it is important to have additional points of software verification and testing throughout the project. By performing a *code review* it is possible to catch significant issues in code prior to their implementation in the main program - done practically in this project by requiring a different person to approve the merging of branches than the developer committing the changes.

Furthermore, at the point of developing each feature within the project, a new series of tests should be written that exercise the feature as a replacement for unit testing which could otherwise have been performed.

Finally, black-box validation and defect testing can be performed at key points in the project, such as at submission points. Much of this can be done internally, with final smoke testing and acceptance testing performed with the client and users representative of the target audience. As part of this, there was a meeting held with the client on Tuesday 7th January.

Detailed documentation, such as Code Standards, the tests performed and documentation can be found on the project website.

Testing *Kroy*

It is not practical, or in many cases possible, to exhaust every possible scenario when testing a developed project. Non-trivial incompatibilities between the two frameworks introduce effective and thorough testing by visual inspection – one method adopted by the project. There was a learning curve, but we did manage to get JUnit tests working for core features of the game; *Firetruck*, *ETFortress*, *Firestation*, ResourceBars, Arrow, Patrols, PatrolMovementSprite, *SimpleSprite and MinigameScreen*.
Testing must be done using Java 8 as LibGDX is not supported on newer versions of the JDK.

End-to-end testing was ran frequently on the project, often by immediately implementing methods and testing their functionality. All of our 69 unit tests have passed and can be accessed at **https://sepr-docs.firebaseapp.com/testing** . Code Coverage shows how much of each class is tested, our Traceability Matrix shows which tests satisfy which requirement (Critical requirements are tested multiple times, non-critical are tested at least once), Test Cases shows a breakdown of each unit & end-to-end test, explaining what the tests cover, which requirement they satisfy, their expected outcome and their pass/fail status, and Testing Results shows the test methods and their pass status.

While writing unit tests, we encountered issues with initializing objects with heavy dependencies on others. We therefore used the Mockito framework to replicate crucial methods from said classes in order for the unit tests to function properly.
Many of our classes cannot be tested, such as LibGDX's *Screens*, we therefore mainly tested entities except for Minigame screen which was small enough to test (although the test's code coverage is low). While the percentages can be used to identify the strengths or weaknesses of our Unit tests, a low percentage on certain classes does not imply weak testing, as some methods do not need to be tested (getters & setters). Each method that remained untested is listed on our website to clarify why we deemed it unnecessary or simply impossible to do so.

Coverage Table for Individual Test classes

Class Tested	Method Coverage	Line Coverage
ETFortess	85%	88%
MinigameScreen	43%	57%
Firetruck	58%	46%
Firestation	73%	70%
Patrol	70%	71%
PatrolMovementSprite	84%	85%
SimpleSprite	90%	93%

In the previous testing, many of the tests that failed was because the feature has not yet been implemented, as tests for the initial and developing project requirements had already been written such that it is possible to gauge to some extent progress through implementation of the project. These include tests around *alien patrols* - a user requirement (with accompanying functional and non-functional elements) which have now been implemented, therefore we re-assessed the end-to-end tests and achieved *pass* outcome for all of them, including *T.U.008*, *T.U.011* and *T.U.012*. Similarly, ending the game once six *ET Fortresses* have been destroyed by the player has now been implemented, as per the requirements for the *Assessment 3* submission, and so *T.N.003* now *passes*. All of our tests, both end-to-end and unit tests achieve a pass outcome.

A test of the options available on the initial *Menu* screen displayed now passes (*T.U.013*) as it displays the options "Play Game, How to Play, Quit", in line with our requirement change for *MENU*. We also introduced *T.U.018* which proves the new How to Play button works as intended.

For our Junit 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 *Firestation*, as when playing the game, *Firestation* needs to create popup messages which are done in a method in *GameScreen*. As popup messages was not a critical feature, we did not unit test them, therefore we could mock *GameScreen* as we never call it when testing *Firestation*.

For our Unit tests that compared 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 refilling a fire truck (T.F.006), we had three JUnit tests, labelled as (T.F.006a, T.F.006b, T.F.006c). 'a' was to check that the truck refilled when empty, 'b' was to check the truck refilled when a single unit of water was taken, and finally 'c' was to check that it did not refill when the truck was already full of water. We think these are important to test as they are all common scenarios that we witnessed when we had people play a demo version of the game, and which must therefore work correctly.

Testing can never be complete[6]. If a point was reached where all tests for the project pass, it would likely be a sign of insufficient testing as 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. It is therefore the case that although through further development and implementation of the project requirements the number *Fail* test results may reduce, there should be no single action that produces a *Pass* status for all tests. Should this ever be the case, it is an indicator more tests should be written.

In an effort to ensure tests for the project are effective at ensuring it meets the initial requirements, there exists a *Traceability Matrix* https://sepr-docs.firebaseapp.com/testing 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, such as testing the way parts of the project have been implemented, but instead only that the initial and extended requirements have been met.

Bibliography

[1] ‘ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:Concepts and definitions’, *ISOIECIEEE 29119-12013E*, pp. 1–64, Sep. 2013 [Online]. Available: 10.1109/IEEESTD.2013.6588537.

[2] M. Costick, (Nov. 26, 2013), ‘Agile testing at the Home Office’, Government Digital Service Blog. [Online]. Available https://gds.blog.gov.uk/2013/11/26/agile-testing-at-the-homeoffice/ [Accessed: 02 January. 2020].

[3] thisisrahuld, ‘Unit Testing Libgdx applications’, *BadLogicGames Forum*. Jul. 27, 2015 [Online]. Available https://www.badlogicgames.com/forum/viewtopic.php?f=11&t=20103 [Accessed: 01 January. 2020].

[4] T. Pronold, *TomGrill/gdx-testing*. 2019 [Online]. Available https://github.com/TomGrill/gdx-testing [Accessed: 01 January. 2020].

[5] *JUnit 5*, JUnit. [Online]. Available: https://junit.org/junit5/ [Accessed: 02 January. 2020].

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