Assessment 2: Brownfield Development

Testing

Cohort 3 Group 9

Oliver Barden

Connor Burns

Sam Goff

Milap Keshwala

Lewis Mitchell

Jeevan Singh Kang

Harry Thomas

Titas Vaidila

Approach

Our testing strategy was focused mostly on automated unit testing with a small amount of manual testing where automated didn't make sense. We limited our manual testing to only part of our game which can't be tested using unit tests. Such manual testing included UI, screens, player inputs, game audio etc.

We set an internal goal to test as much of the code through unit testing as possible, using JUnit. The LibGDX headless launcher was used to allow for the testing of code that requires the LibGDX core to be running without needing the graphical application, which made these tests faster.

To measure code coverage we used JaCoCo, set up within the Gradle configuration files. JaCoCo gives an in-depth report containing the percentage of each class covered by the tests we wrote. We also used Mockito for mocking to more easily test gameplay events without having to run the whole game itself.

The first task we had was writing tests to cover the existing code we inherited from the other team. This involved some refactoring to separate any graphics-related code (which causes errors when using the headless LibGDX instance) from code we could test with JUnit. This was mostly done by creating new methods to house the rendering code. We are not able to achieve 100% test coverage because of this issue, but we have attempted to cover as much as possible. This strategy was continued across the new code added to implement the new requirements.

There is a lack of unit tests across the game screen classes, as most of their functionality is within pre-existing LibGDX UI classes which are just put together to form the game screens. It is easier to manually test these to see if they are visually correct and function correctly at runtime. We were able to conduct these manual tests as part of the user evaluation along with constant playthroughs which the team itself conducted. This allowed us to test things like making sure the buttons work as expected, the audio for certain events gets played only when the event is triggered and much more.

Test report

Coverage report on website
Test results on website

All automated tests were executed in the headless module using the Gradle test pipeline.
95 tests were run and they all passed resulting in a success rate of 100%.

Module	Description	Success rate	Instruction Coverage	Branch Coverage
io.github.yetti_eng.screens	Contains all the libGDX screen	100%	0%	0%
io.github.yetti_eng	Contains the core game logic and managers	100%	65%*	63%*
io.github.yetti_eng.entities	Contains game's interactable entities	100%	68%	50%
io.github.yetti_eng.events	Contains all the positive, negative and hidden events	100%	97%*	100%

*Coverage is not accurate because certain aspects of the module were manually tested, as a result coverage will likely be higher.

io.github.yetti_eng.screens:

- This module could not be tested via automated unit tests and so we resorted to manual testing, which we mention in the manual testing part of this report.

io.github.yetti_eng:

- This module had 21 tests which all successfully passed.
- The instruction coverage and branch coverage were 65% and 63% respectively. The classes YettiGame and InputHelper in particular had low coverage as it was more appropriate to perform manual tests. This resulted in the overall coverage being dragged down, removing these classes from the average increases the instruction coverage to 84% and branch coverage to 66%.
- The main tests for this module were LeaderboardManager, AchievementManager, MapManager, and Timer.
- LeaderboardManager tests; fetching leaderboard, adding score to leaderboard if it's in the top 5, reset leaderboard and sorting the scores in descending order. 94% instruction coverage and 71% branch coverage.
- AchievementManager tests: loading achievements, saving achievements, getting achievements, unlocking achievements, and resetting achievements. 86% instruction coverage and 75% branch coverage.
- MapManager tests; if an entity has collisions with the map. 82% instruction coverage and 90% branch coverage.
- Timer tests the behaviour of: starting timer, pausing timer, getting current or remaining time, if the timers completed. 70% instruction coverage and 45% branch coverage.

io.github.yetti_eng.entities:

- This module had 13 tests which all successfully passed.
- A key aspect to highlight was the lack of any coverage for the SocialSec entity due to time constraints. This would be a key area to focus future testing on after this project has been shown to the client.
- The main tests for this module were Player, Entity, Item, Dean.
- Player tests: if items(for events) get added to player inventory. 64% instruction coverage and 33% branch coverage.
- Entity(which all other entities inherit form) tests: entity movement, collisions, position. 94% instruction coverage and 83% branch coverage.
- Item tests: if an item is used, if another entity has interacted with it. 92% instruction coverage.
- Dean tests: if the dean moves towards the player. 100% instruction coverage.

io.github.yetti_eng.events:

- This module had 61 tests which all successfully passed.
- In-game events were crucial to the client and we had focused a large part of the testing to ensure the coverage for events near 100%. The WinEvent had two methods createNameInputScreen and createWinScreen, and as mentioned previously the UI and screen were tested manually. Removing these two methods meant we achieved a 100% coverage on both instructions and branches.
- There are 12 events which extend from the main Event class..
- Each event tests; its activation/trigger, the change in score and other other changes it is expected to perform e.g. coffee should increase speed.

All tests being run successfully suggests that the implemented tests are correct and internally consistent with the values of the expected behaviour of the system. We set out to make sure each unit test will accurately test the intended behaviour of the code, for example when implementing the leaderboards we only wanted the top 5 scores being displayed in order. In order to test this behaviour we designed unit tests that would; insert more than 5 scores in the leaderboard, then assert that the leaderboard indeed only saves the top 5 and that the scores get scored in descending order. The unit test would fail if this was not the behaviour of the code. Its success informs the team on the correctness of this specific behaviour. We applied this method to all the tests in order to ensure the correctness of our tests.

As for completeness of our tests, this can be measured through instruction coverage and/or branch coverage. However this does not give the entire picture as certain aspects of the systems were not part of the unit tests. We instead looked at the behaviours which are crucial to meet the requirements the client had set out, such behaviours are Timer, Leaderboard, Achievements and Events. Overall, we feel that these behaviours meet the requirements and therefore we meet the definition for completeness since core behaviours were all tested. That being said, there are definitely more improvements we can make for further testing such as covering more branches and edge cases that we were limited to due to time constraints.

Manual testing on website

Manual tests were carried out for menu navigation, visuals and graphical rendering, so we could confirm proper functionality. As mentioned previously in the approach section, we

conducted the manual tests as part user evaluation along with the constant tests conducted by our own team during development.
The website mentions how we conducted these manual tests. We had certain tasks the tester would perform which is mentioned in the website and we would then check off if these tasks were completed successfully. The results for manual testing showed all the UI, screens, audio, map behaviours performed as expected.