Testing

Group 12

Team 12

Daniel Agar

Charles Thompson

James Leney

Leyi Ye

Matt Smith

Vanessa Ndiangang

Introduction

Our focus in testing was to ensure that the requirements set out in req2 were met, the code is of a high quality and also to allow the project to be expanded on. Our testing approach for this project is loosely based upon Mike Cohn's testing pyramid [1]; where the majority of our testing is carried out by unit tests, followed by integration tests and a small amount of end to end GUI tests as these are the most costly to carry out. In the end our end to end tests made up around 10 percent of the tests; the rest being unit and integration tests.

Our automated tests have been implemented using JUnit and TomGrill's GdxTestRunner [2], using mockito when necessary. We have chosen to use this GdxTestRunner as a framework for our unit and integration tests to simplify creation of tests and due to its good documentation and reliability. Our automated tests focused on the backend logic of the game ensuring that aligns with our requirements and the product brief.

For most of our tests we have used white box testing however in one case this was not suitable. In testing our scoring logic it seemed preferable to use black box testing as a way to guide development. This meant we could avoid having test cases that are over-specific to our code. The scoring of the game has some particular requirements to meet: penalties for over studying, the player must eat and do recreational activities and we found it important to efficiently test that the scoring was reflective of how a player performed.

In creating our tests we have aimed to prevent having brittle tests by following the guidance given in chapter 12 of Software Engineering at Google [3]. By only using the public interface of each class and testing just the behaviour our tests should remain unaffected by pure refactorings, bug fixes and new features. In our tests we have avoided using control logic within tests where possible and move any control logic into helper methods for more confidence in the correctness of our tests [4].

Due to the small scope of the project we have opted to manually run end to end tests, including exploratory testing, which allows us to verify behaviour of the game in a way that automated testing is unable to do. A human tester is able to tell whether the system 'feels slow' or stutters and more generally comment on the user experience which is arguably the most integral aspect of a game.

Testing report

Automated Tests:

Overall out of 57 automated tests 98% are successful. Our automated tests cover the entirety of the backend portion of our code and is automatically run for each commit. Generated test and coverage reports are available on the website.

Asset Tests:

As part of our test suite we have included asset tests which check that each asset used during runtime exists. The code was refactored so that the path for each asset is now stored as a public static final string at the top of each class, this has improved code readability and has allowed us to verify that assets are correctly referenced without any typos.
100% of asset tests passed in the most recent version.

PlayerTests:

PlayerTests involved tests for the movement and collision of the player functions in the Player class, all player tests passed. This class covers main functionalities and some edge cases. It tests any keys that should move the player, moves the player in the correct direction. Verifies that diagonal movement happens and the player does not exceed their maximum speed. The player can't go through collidable tiles on the map.
During development a test in this class found an issue whereby given a large enough time delta the collision system would not activate and the player could skip over a collidable tile. This was fixed.
One test failed which tested whether the motion of the player was proportional to the time delta between calls of the update method. This failed and it was found that the player would move by a set amount each time the method was called which would mean the user experience would differ depending on the users hardware, to rectify this the player update method would have to be altered to move the player sprite according to the time elapsed.

DayManagerTests:

Tested the DayManager class which controls the scoring and streaks. These tests are crucial in the refinement of the scoring of the game, allowing us to prove whether the scoring system aligns with the brief and the requirements. All tests in class passed.
Tests involved:

Comparing the scores of mock games,
Verifying the game ends after 7 days,
Checking fail conditions, the player missing study 2 days in a row,
Verifying that penalties were given to a player if they overstudied,
A normal game should score between 40 and 100.
Triggering streaks and checking they were stored.

DayTest:

The day class is relatively simple and the tests reflect as such, this class stores and summarises information about the day for use in determining the user score, DayTest included tests for incrementing the study eat and recreational scores and for the conversion of time in hours into morning, afternoon and evening.. All of these tests have passed.

ActivityManagerTests:

The activity manager tests are a collection of integration tests for the activity manager class. The activity manager class is heavily dependent on the day manager class and the day class hence to fully test the functionality of the activity manager we used integration tests to allow us to test this class using real dependencies, thus more accurately demonstrating the class' behaviour in a real game environment.
One test failed in this class which verified that the player shouldnt be able to study twice in one day unless they haven't studied the day before (FR_STUDY_RESTRICTIONS). To amend this more logic will need to be included in the activity manager class that uses the statistics from the previous day to check whether the player can study or not.

GameDataTests:

The game data class handles the logic about scoring and whether to add a score into the highscores array or not

SaveTests:

The save class handles saving and loading the GameData class, we have used mock objects to test this class' functionality. It covers the following test cases: successful save and load, an exception thrown during a save and load, and cases where the file doesn't exist.

100% of these tests passed for the most recent build.

Score Tests:

Score tests are unit tests that check the output of the score.compareTo method. Both of these passed for the most recent build.

Manual Testing

Due to the difficulty of testing some methods and requirements automatically, we have carried out several manual tests. Roughly, they cover aspects in relation to User Interface and performance of the game. All of them have achieved the expected results and passed successfully. Test cases are provided on the website.

Bibliography

[1] C. Ashbacher, “Succeeding with agile: software development using scrum, by mike cohn.,” *JOT*, vol. 9, no. 4, 2010 [Online]. Available: 10.5381/jot.2010.9.4.r1.

[2] *TomGrill/gdx-testing: This is a skeleton for libGDX projects which require testing with JUnit and Mockito.* [Online]. Available: https://github.com/TomGrill/gdx-testing. [Accessed: May 19, 2024].

[3] *Software Engineering at Google*. [Online]. Available: https://abseil.io/resources/swe-book/html/ch12.html#unit_testing. [Accessed: May 17, 2024].

[4] *Google Testing Blog: Testing on the Toilet: Don’t Put Logic in Tests*. [Online]. Available: https://testing.googleblog.com/2014/07/testing-on-toilet-dont-put-logic-in.html. [Accessed: May 19, 2024].