UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE

Software Testing Report

Group 14

Tecch Titans:

Bradley Mitchell

Daniz Hajizada

Ellie Gent

Joel Crann

Keela Ta

Leo Crawford

Lukas Angelidis

Testing Methods and Approaches:

Beginning the testing of our project, we designed a test plan aiming to ensure that a good coverage of the project was tested ensuring functionality across the project and the best possible user experience. This was intended to have a good amount of unit tests, with less integration tests than unit tests and even less end to end tests to test the overall system functionality. This is as the smaller isolated unit tests' functionality are often incorporated into the larger tests (and of course user evaluation with further tests were carried out in conjunction to this that can catch other unexpected results).

Given the architecture of the project uptaken, we decided to begin with the implementation of frameworks for each individual class. Some of the tests were removed as deprecated tests since the highly interlinked coupled architecture would have required a large level of refactoring to be able to test well which was not ideally acceptable given the time scope of the project.

We ended up with an unintended, almost antipattern, almost uniform testing spread as the tight coupling architecture makes it very hard to test dependencies in isolation. The coupled architecture also made us need to utilise mocking for a lot of the project to replace the external interface with uninstantiated attributes. This allows assurance that the code runs with a correct call, the correct number of times and with the correct parameters alongside other functionalities such as where an internal interface cannot be defined. We also had to add some getters, unnecessary to the game and core code, solely for testing purposes.

End to end system testing was then performed manually on the overarching system aspects that are not easily tested; this was done to ensure a quality user interface implementation and interactions functioned as intended in many areas.

We added the JaCoCo library as a Gradle dependency and a feature in our CI workflow to auto generate local coverage and test reports. We also set this up to store these as artifacts online to check how well we and how valid our tests functioned and also how much of the code we have tested vs the amount of code that we could have tested. This was generated for every GitHub workflow action as a commit or pull request on any branch. We found that after adding JaCoCo, our original testing plans became more clear and it was easier to consider where we needed to add more testing content.

IntelliJ also notably has a Run Tests with Coverage feature that is well built and was very useful to suggest what areas of code that we needed to test without any CI implementations. We also chose to implement the skeleton class GdxTestRunner requiring JUnit and Mockito that allows you to run a headless (runs without a user interface) LibGDX environment.

The automated code tests package aimed to test the major methods and mutators of each class with their single functionalities or their grouped interactions to ensure the functionality and readability of various game components. Generally, given the small scope and amount of rapid changes the project undertakes, automated tests are very suitable due to their ability to quickly and repeatedly validate functionality, catch regressions, and give ample detail on test coverage as code develops.

Test Report:

This report provides a comprehensive overview of the tests conducted during and after implementation of Heslington Hustle to evaluate the game's performance and correctness. The tables below show both automated and manual testing performed throughout this process and the project requirements that they meet.

Automated Tests

Test ID	Requirement	Tests ran	Checks	Pass/ Fail
1	FR-MENU FR-MENU3	CreditScreenTest	Ensures credit screen is initialised and renders correctly	Pass
2	FR-SCORE	DailyAcitvityTest	Ensures the many mutators for the different daily activities function as intended	Pass
3	FR-INTERACT2	DialogueBoxTest	Ensures dialogue can be set to specific text	Pass
4	FR-GAME-PLAY1 FR-GAME-PLAY2 FR-GAME-PLAY3 FR-GAME-PLAY4	EventManagerTest	Checks events run and return the correct text	Pass
5	FR-INTERACT2	GameObjectTest	Checks the interaction between objects and getting the correct keys	Pass
6	FR-WEEK	GameOverScreenTest	Checks if the player will pass their exams if they get the requirements to do so. Also checks if they fail if they fail to meet the requirements	Pass
7	FR-NAVIGATE	HustleGameTest	Checks if maps work correctly when switching from and to town	Pass
8	FR-MENU1 FR-MENU3	LeaderboardScreenTest	Ensures leaderboard screen is initialised and renders correctly	Pass
9	FR-MENU1 FR-MENU3	SettingsScreenTest	Checks that the music and sfx volumes are updated properly across the program	Pass
10	FR-MENU FR-MENU3	SoundManagerTest	Ensures changing music and sfx volume works appropriately	Pass
11	UR-INTERACT	PlayerTest	Ensures the player position is properly updated and object distance calculations work properly to interact with	Pass

			objects/buildings	

Manual Tests

Test ID	Requirement	Why	How	Pass/ Fail
12	UR-SETTINGS	All game settings should be able to be accessed by all players.	Main menu loaded correctly, with all links to other interfaces checked.	Pass
13	UR-INTERACT	Ensures that all buildings on the map have the correct functionality.	Interactions were made with all buildings on both maps, as well as trees on both maps.	Pass
14	UR-MEMORY	To ensure the leaderboard functionality is working correctly	Game played until the end and checked the ‘leaderboard’ interface.	Pass
15	UR-STREAK	Ensure that the streak functionality is working correctly.	Completed a particular activity every day during play-through.	Pass
16	UR-CUSTOMISE	Ensures customised characters are visible on the screen.	Selected both avatars and played through the game.	Pass
17	UR-DESIGN	Possible improvements and bug-fixes	User evaluation performed by other module students	Pass*
18	UR-WORLD	Ensure the maps load correctly and the player can move around the maps.	Map loaded and WASD keys tested on both maps.	Pass
19	UR-SOUND	Ensures all sound and music plays as expected.	Adjusted sound/music settings and game played until the end.	Pass
20	UR-SLEEP	Ensures sleep functionality works, and time of sleep is correct.	Game played with sleeping and checked sleep time on screen matched hours slept.	Pass

Given the coupled architecture and the time scope of the project, we were happy with the coverage we achieved testing our project. All of the tests that remained to the end of the project passed or passed after some thorough considerations to the project and the architecture. While some of these tests possibly should have remained to the end, they were removed early on as deprecated tests as the scope would have made it very hard to implement useful workings.

We only managed to reach a 22% code coverage but would have liked to have achieved at least a covered range of 30% upwards. However, we did then for every test that we could implement and utilise, manage to reach 100% success rate in a very lightweight body of code, taking a very short time to run and complete. This can all be seen on either GitHub repository or directly under the testing section of the website which you can find the link to below at the bottom of the document.

This does tell us that while the testing was not wholistically complete, the tests implemented were seen to be very correct. Another success point of our testing implementations was that we used a DAMP testing framework that was almost exclusively followed throughout the entire project to make sure that the testing code promoted and strived to achieve Descriptive and Meaningful Phrases to make up for the lack of documentation and still keep them lightweight but easy to follow.

Failed Test Analysis
Some tests in the automated and manual testing tables appear with an asterisk in the ‘Pass/Fail’ column. This refers to tests that originally failed but after a different method of implementation, they passed. For automated tests such as “MenuScreen” and “GameScreen” were initially implemented using a specific framework but did not avail to success given the coupled architecture and it’s highly interlinked design that would have needed huge amounts of code refactoring to be able to be suitably and correctly tested.

The manual test 17, due to its usability testing nature, did not originally pass as after these sessions were conducted, all feedback for improvements then needed to be implemented in order for this test to be considered successful.

To view all testing report statistics, the link to this can be seen through [this link].