Assessment 2: Software Testing Report

Group Number: **Team 20**

Group Name: **Gourdo Ramsay**

Group Members:

Lauren Waine
Megan Bishop
Tommy George
Bartek Grudzinski
Davron Imamov
Nathan Sweeney

a) In our testing approach, we aimed to take a requirements-focused approach to ensure that the game could deliver upon them. In order to document this, I created a spreadsheet that listed our requirements, and whether they had been covered or not. We found this to be a useful benchmark for testing coverage in the initial stages of the project as there were significant amounts of code that could not be tested through automated testing and this enabled us to see where our testing coverage was lacking. It also meant that we could work towards a more concrete metric rather than an arbitrary testing coverage percentage. However, as the scale of the project increased, it became unfeasible to update the sheet with the level of detail required. Therefore, the Javadoc comments were more appropriate to document which requirements were being tested.

There was also additional testing conducted that did not concretely link to requirements. For example, the AssetTests package in our project contains tests for the presence of the assets needed for game functionalities (for example images, json files and txt files). This was more related to risk mitigation than requirements as it prevented accidental deletion or renaming of assets in the game. The ToolsUnitTests package does not directly test our game requirements, but tests the architecture that underpins our game functionality.

Our aim was to conduct the vast majority of our testing through automated tests. For this, we used the Junit framework, with the LibGDX headless backend. The headless backend provides a good “fake” as a lightweight implementation. However, it is primarily intended for use in servers, meaning that it is not always perfectly suited for software testing, and relevant documentation was difficult to find. Unfortunately, there were no available appropriate, purpose-designed alternatives for use with LibGDX.

When writing unit tests, our test cases aimed to cover as many boundary cases and scenarios as possible to ensure that the code works correctly in all situations. We ensured that we considered both valid and invalid inputs to the unit tests wherever possible.

Our continuous integration implementation allowed us to track our automated testing coverage throughout the project’s lifespan, giving us a full coverage report with each GitHub commit.

We aimed to only conduct manual testing when automated testing was not possible or feasible. One area that required a lot of manual testing was the graphical interface of the game. This was because the LibGDX headless backend provides a level of abstraction that decouples the game logic from the graphical rendering, making it impossible to test through JUnit tests in this way. The button listeners that make the main menu functional also needed to be tested manually as the implementation of the buttons means that the graphical rendering and the functionality of the buttons are coupled together using LibGDX graphics tables. The buttons themselves have no testable logic.

Our manual testing was recorded onto a spreadsheet. Because the manual testing was heavily focused on testing the graphics of the game, we split the tables into screens to test that all graphical assets rendered and functioned correctly on each screen. The spreadsheet also listed which requirements were tested, if applicable.

B)

Automated Testing

In total, 147 automated JUnit tests were run. This represented 54.4% (68/125) of classes, 58.4% (324/555) of methods and 42.5% (1406/3312) of lines in the codebase.

AssetTests

61 AssetTests were run, testing for the presence of required game assets. This represented 100% coverage of game assets. Note that temporary assets created during runtime are not tested as it is not possible for them to pass consistently on all machines.

All of the AssetTests passed, meaning that all required game assets were present in the build. We are satisfied with the completeness and correctness of the AssetTests package, as all game assets were tested, and the usage of assertions is correct and appropriate throughout.

RequirementsUnitTests

50 unit tests that directly test our requirements were run. The breakdown for this is as follows:

Package	Class, %	Method, %	Line, %
Sprites	61.8% (21/34)	73.6% (67/91)	69.7% (288/413)
Recipe	85.7% (6/7)	70% (7/10)	57.9% (33/57)
Ingredients	100% (11/11)	95.5% (21/22)	92.7% (76/82)

100% of our requirements unit tests passed. However, this is the testing package with the most scope for improvement in terms of completeness. There are many classes which are lacking in test coverage, for example the Powerup classes, and the bin class. The most effective way to resolve this is to simply write more test cases to cover these classes.

Some classes, such as the Hud class, appear to be severely lacking in testing coverage in the automated tests. However, the Hud class is a class that is almost entirely responsible for graphical rendering, which cannot be tested through the LibGDX headless backend. Therefore, this class, and other similar classes and methods responsible for graphical rendering are tested manually.

We are satisfied with the correctness of our testing methods in this class. We ensured that our JUnit assertions were specific and correct, in order to effectively capture any software faults. We also ensured the tests were well-updated throughout the project in order to avoid false test failures, or new bugs introduced during the development process. Potential improvements could be made by using parameterised tests, which allow for tests to be run with different input values to more effectively test a broad range of input cases.

<u>ToolsUnitTests</u>
36 tools unit tests were run. These were tests that did not directly test one of our game's requirements, but tested the architecture that underpins our game functionality. Overall, 55.6% of classes (5/9), 60% of methods (48/80) and 24.7% (118/478) of lines were covered in our game tools package. The breakdown for this is as follows:

Class	Class, %	Method, %	Line, %
WorldContactListener	0% (0/1)	0% (0/5)	0% (0/23)
SaveDataStore	100% (1/1)	50% (12/24)	49% (24/49)
OrderDataStore	100% (1/1)	62.5% (5/8)	81.2% (13/16)
IngredientDataStore	100% (1/1)	100% (15/15)	100% (37/37)
DemoScript	0% (0/1)	0% (0/5)	0% (0/123)
Constants	0% (0/1)	0% (0/1)	0% (0/1)
CircularList	100% (1/1)	100% (8/8)	100% (28/28)
ChefDataStore	100% (1/1)	100% (8/8)	100% (16/16)
B2WorldCreator	0% (0/1)	0% (0/6)	0% (0/185)

100% of our tools unit tests passed. However, there is some scope for improvements in completeness of the testing. For example, the DemoScript class, which is responsible for making the game play automatically in a "demo mode" when the user is idle, was missed from testing and should have been tested through the unit tests.

The tools package also contains some classes, such as B2WorldCreator, which cannot be tested manually due to being responsible for graphical rendering. This was therefore tested manually.

Interestingly, although the constants class was thoroughly tested with all constants tested against their expected values, this is not reflected in the test coverage report. The reason why this occurred is currently unknown to us.

Similarly to the requirements unit tests above, we are satisfied with the correctness of the tests in this package.

Manual Testing

In total, we conducted 37 manual tests. The breakdown for this is as as follows:
- Main menu - 8 tests conducted. 8/8 passed.
- Play screen - 8 tests conducted. 8/8 passed.
- Shop screen - 9 tests conducted. 9/9 passed.
- Demo screen - 6 tests conducted. 6/6 passed.
- Instructions screen - 6 tests conducted. 6/6 passed.

This gives an overall pass rate of 100%.

We are satisfied that we tested thoroughly all aspects of the game that needed to be tested manually, and strictly did not test anything that could have been tested using automated unit testing. This is reflected in how the game runs, as there are no notable faults in the tested areas.

In order to improve our manual testing methodology, it would be beneficial to be more specific on exact requirements to pass the test, perhaps breaking each test into several smaller tests to narrow the scope of each test. The current manual tests are too broad, which could lead to false passes and missed faults. Narrowing each test's scope would make it easier to prove correctness of the manual testing.

c) All testing files can be found at: https://gourdoramsay.github.io/

- Test2.pdf (this document) - https://github.com/GourdoRamsay/GourdoRamsay.github.io/blob/main/files/team20/Test2.pdf
- Automated testing coverage report - https://github.com/GourdoRamsay/GourdoRamsay.github.io/blob/main/files/team20/TestingCoverageReportFINAL.zip
- Manual testing report - https://github.com/GourdoRamsay/GourdoRamsay.github.io/blob/main/files/team20/Manual_Testing_Records.pdf