Testing Report

Group 28

Piazza Panic

By OuseWorks

Ben Harris
Joshua Gill
Niamh Hanratty
Amy Raymond
Matthew Czyzewski
Matt Rohatynskyj

PART A - Summary of Methods
A multifaceted approach was used for software testing during the development of the Piazza Panic game to ensure our code executes as expected. This approach consists of automated unit testing using the JUnit Testing Framework wherever possible, and manual testing where unit testing could not be performed to evaluate the software.

Automated Testing - JUnit Testing with Headless LibGDX Backend and JaCoCo coverage report
The JUnit Testing Framework was chosen as our automated testing method due to its wide adoption for testing Java-based applications in the global industry. It is a relatively simple framework in terms of writing and execution and is supported by almost all popular IDEs (Integrated Development Environments). Furthermore, multiple features which JUnit provides are helpful for test writing and reading. For example, JUnit provides methods called assertions, allowing us to compare expected behaviour with actual behaviour easily. Also, JUnit provides annotations which are syntax that allows a file of tests to be structured more strongly to give better readability. An example of an annotation used in almost all of the tests we have written is the @Before annotation. This annotation can be used on methods inside the test file, to tell JUnit to execute before each test case (annotated by @Test) inside the file.

The headless backend for LibGDX was used in the testing environment, to test the expected behaviour of individual classes within our source code, without initialising the graphical aspects of the game. This is because unit testing refers to the behaviour of the backend of the game, not the front end. This is appropriate for most of our code, as we have tried to separate the game logic from the graphical components as much as possible.

Tests were annotated with @RunWith(GdxTestRunner.class) which was an attempt at using Mockito to mock GL20 in files with graphics using the headless backend tests, however, we were unable to get this working as Mockito couldn't mock the GL20.class correctly - meaning had to use manual testing for these files.

JaCoCo is a code coverage tool which was chosen to identify the structural and statement coverage of our tests during development. This tool produces an html file which provides a graphical representation of the file and instruction coverage for all classes in our Piazza Panic package. The approach we made towards code coverage was to ensure that at least 80% of the code which is able to be covered using the headless backend (no graphics code), should be covered using the automated tests. This goal was mainly set due to the infeasibility of obtaining ~100% coverage within the time constraints of the project.

Manual Testing - Playtesting
A manual approach to testing was required to cover the code which we couldn't provide coverage with automated testing utilising the headless backend of LibGDX (mainly graphical features of the game). After research into multiple other manual testing methods, playtesting seemed to be the most appropriate approach for the goal mentioned above, as it is used often in the video game development industry to identify potential design flaws/ bugs which are difficult to visualise from looking at the source code alone. For the majority of the playtests we conducted during development, we ran the full game build on a machine to inspect the graphical front end of the functionality of the game.

Playtesting was also used to test releases of the game, to check that the functional requirements given in the product brief alongside those elicited during the development were met to the correct specifications. These were tracked using the functional requirements provided in the requirements documentation (Req1).

PART B - Test Report

JUnit Test - Writing

- A 'BasicTest' interface was written before any tests were written as a simple structure from which the tests could implement. This allowed the writing of tests to happen in a quicker manner as the basic Test methods could be overridden very easily using an IDE. For example, the basic structure of a BasicTest includes a constructorTest (to test the instantiation of a class as a variable) and an @Before initialization method which does any necessary computation before the class could be instantiated.
- All attributes which were instantiated during test execution are kept as private fields, preventing any unwanted access.
- Throughout the duration of the writing period, multiple variations of the same test were created to try and create variation within the input space. For example in Figure 7, the test method keyDownTests() checks multiple inputs to see if any variation makes a difference in the rigidity of the keyDown() method.

JUnit Test - Execution Results

- On every run of the automated tests (example tests illustrated in Figures 5, 6, 7, 8), a test report is generated (Figure 1) providing a visual representation of the areas of code where tests passed, failed or were ignored. It also indicates the time duration which it took for all tests to be run.
- For in-development branches, the tests are available to be run on command at any point in time, allowing developers to preserve old functionality when working on new functionality by checking any tests which fail due to new code, allowing them to fix these bugs dynamically during production, instead of later during manual testing.
- Due to our implemented continuous integration system, all automated tests are scheduled to run on every execution of Gradle build, creation of pull requests, committing to a branch with an open pull request and merge of pull requests. For these actions to complete, all tests must pass (100% success rate). This acts as a protective guard to ensure previously implemented functionality remains undisturbed. This system means that the **main branch of the GitHub repository will always have a test success rate of 100%** unless the branch protection is removed.

The final release build of the game has an automated test success rate of 100%, meaning that no tests failed. As all tests pass, this does throw a question about the completeness and correctness of our tests.

- Due to time constraints, the majority of the automated tests only check singular expected and unexpected values. This means that the functionality of particular methods in classes isn't tested with a wide assortment of input values, which is a limitation of our tests. This means that our implementation may not be as strong as it needs to be to deal with unusual parameters.
- If more time was available for test writing, tests would be written more robustly using multiple cases would be used including but not limited to using low and high boundary values, alongside extreme values. Furthermore, more test cases should test for completely invalid inputs, as this is still part of the input space.

JaCoCo Test Coverage Results

- Our previously defined objective was to achieve at least 80% instruction coverage for the code which doesn't have function calls to graphical elements which are mainly contained in the sub-packages 'utility', 'components' and 'input'.
- The latest test coverage report generated on these sub-packages (illustrated on website) shows instruction coverage of 82%, 95% and 100% respectively meaning we have met our targets in these areas. The 'utility' sub-package only just met the 80% target coverage, therefore if we were less limited by time it would be sensible to improve this as much as possible.

- The ‘utility’ sub-package fails to meet the branch coverage target of 80% with only 69%. While this is still quite a large proportion of the code, ~30% of branches were not tested. This can lead to previous functionality being broken without notifying the developer during development which will lead to a lengthy debugging process. Provided with more time, more test cases that fit the missed branches would be produced within the tests that have low branch coverage, to prevent these types of issues.
- For the other sub-packages, the instruction coverage was limited by the testing environment not being able to mock the graphics correctly using Mockito leading to low coverage scores. This was expected and manual tests were developed to deal with the lack of automated tests in these areas.

Manual Playtesting Results

As mentioned previously in this document, playtesting was the method used to simulate the functionality unit tests would test on code which couldn’t be tested using the automated system in place, as well as general checking of the functionalities required by the product brief as listed inside the requirements documentation.

- Testing the HUD was not possible with automated testing because of the high amount of graphic calls, leaving us to manually test functionality that utilises the HUD. Throughout the testing process, we were able to see how adjustments to the code appeared to the user (i.e. the spacing of different icons). We were able to identify the fulfilment of 96% of functional requirements.
- The only functional requirement that was not met was FR_PREP_FAIL.

Testing Traceability Table

Requirement ID (referencing Req1 document)	**Testing Method**	**PASS/FAIL**
FR_TIMING	Manual (Playtest)	PASS
FR_TOGGLE_CUSTOMERS	Manual (Playtest)	PASS
FR_COMPLETION_TIME	Manual (Playtest)	PASS
FR_COMPLETION_TIME_LIMIT	Manual (Playtest)	PASS
FR_HELP	Automatic (Unit Test for the existence of assets) + Manual (Playtest for drawing on screen)	PASS
FR_COMPLETION	Manual (Playtest)	PASS
FR_DIFFICULTY	Manual (Playtest)	PASS
FR_CUSTOMERS	Manual (Playtest)	PASS
FR_RECIPES	Automatic (Unit	PASS
FR_COOKING_STATIONS	Automatic (Unit Test for entity creation) + Manual (Playtest for graphical	PASS

	representation)	
FR_INGREDIENT_STATIONS	Automatic (Unit Test for entity creation) + Manual (Playtest for graphical representation)	PASS
FR_CUSTOMER_FLOW	Manual (Playtest)	PASS
FR_CONTROLS	Automatic (Unit Test for functionality) + Manual (Playtest)	PASS
FR_STATION_NUMBERS	Manual (Playtest)	PASS
FR_CUSTOMER_TIPS	Manual (Playtest)	PASS
FR_MONEY	Manual (Playtest)	PASS
FR_DISH_PRICES	Manual (Playtest)	PASS
FR_COOKS	Automatic (Unit Test for cook creation functionality) + Manual (Playtest for game specific objective)	PASS
FR_MORE_COOKS	Manual (Playtest for graphical confirmation)	PASS
FR_MENU	Manual (Playtest for graphical representation)	PASS
FR_POWER_UP	Manual (Playtest for graphical powerup menu)	PASS
FR_UNLOCK_STATIONS	Manual (Playtest for graphical confirmation)	PASS
FR_COLLISIONS	Manual (Playtest for graphical confirmation)	PASS
FR_RECIPE_BOOK	Manual (Playtest for graphical confirmation)	PASS
FR_COUNTER	Manual (Playtest for graphical confirmation)	PASS
FR_SAVE_GAME	Manual (Playtest for graphical confirmation)	PASS
FR_PREP_FAIL	Manual (Playtest for graphical confirmation)	FAIL

https://ouseworks.github.io/WEBSITE-ASSESSMENT-2/diagrams/testingdiagrams.html
https://ouseworks.github.io/WEBSITE-ASSESSMENT-2/jacoco-report/test/html/index.html