University of York Department of Computer Science
Engineering 1

Software Testing Report

Cohort 3 Team 5 - alltheeb5t

Aaron Heald
Alex Gu
Arun Hill
Jade Stokes
Maksim Soshchin
Meg Tierney
Will Hall

This report outlines the testing methods and results for the game developed by our group. We aimed to test all system requirements, using 70 automated tests with JUnit wherever possible. Some features, such as those requiring user interaction or visual feedback, could not be tested automatically, so we conducted 18 manual in-person tests, detailed in a separate document titled Manual In-Person Testing. Each test is linked to a specific requirement. This ensures comprehensive coverage and clear traceability for all requirements.

Testing Methods and Approaches

Code Refactoring for Testability -
To facilitate thorough testing, we reorganized the code by increasing modularisation and moving key components into the GameGlobals file, enabling easier access and manipulation within unit tests. This restructuring improved test coverage and maintainability since we reduced dependencies on UI-related components. By utilizing GameGlobals, key variables are stored independently of UI-related instances, simplifying testing and ensuring the separation of concerns.

Automated Testing with JUnit -
We used JUnit to create automated tests, allowing us to quickly verify the game logic against functional requirements. With their quick run time, automated tests enabled rapid debugging and supported test-driven development. To assess coverage, we used JaCoCo to analyze which parts of the code were tested. JaCoCo generated detailed reports showing which lines and if-statements were executed during tests, helping us identify areas of the game that were entirely untested. This ensured thorough testing of the game's logic, decision-making processes, and edge cases, making the development process more robust and reliable.

Manual In-Person Testing -
In addition to automated testing, we conducted manual testing by actively playing the game. This approach was essential for evaluating aspects that are challenging to automate, such as usability, user experience, and ensuring the accuracy and appearance of UI elements. The coverage report reflects poor automated test coverage on files containing LibGDX UI-related code due to these manual testing requirements. Additionally, the game incorporates randomness, which we excluded from automated tests to maintain repeatability. Manual in-person testing ensured comprehensive coverage of all system requirements.

Functional Requirements-Based Testing-
All tests were designed to validate compliance with functional requirements, which were themselves derived from user requirements. This ensured that the game met its intended purpose from the perspective of end users.

Automated Test Result Table

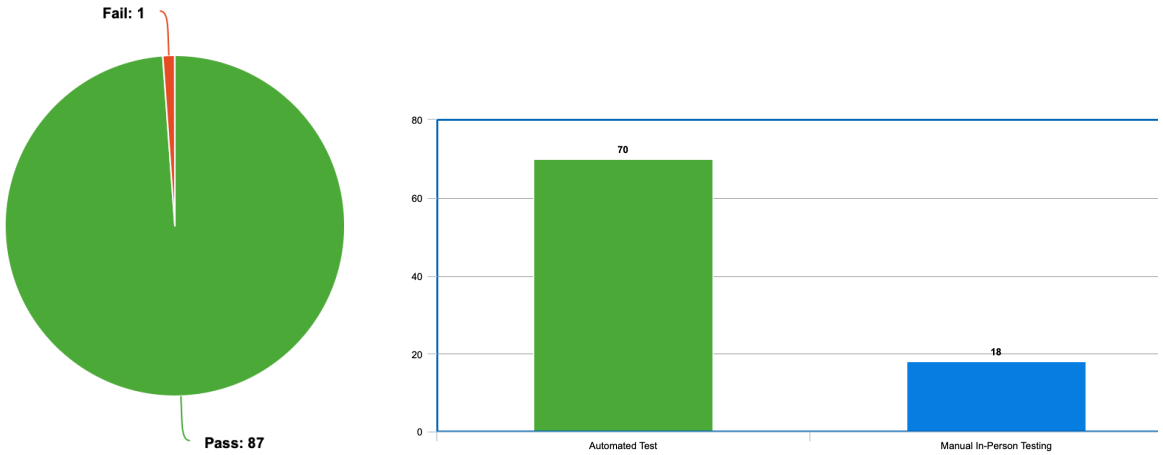
Test Name	No. of Tests	Pass / Fail	Purpose
Achievements Test.java	20	Pass	Ensured achievements triggered correctly under gameplay conditions, handled edge cases robustly,and displayed accurate notifications for unlocked achievements.
Building Test. java	7	Pass	Verified that buildings could be placed correctly, collisions were handled properly, deleted spaces could be reused, and buildings stayed within the map.
CounterTest.java	1	Pass	Ensure that building counters are appropriately incremented when a new building is placed and decremented when a building is deleted
EventsTest.java	18	Pass	Checked that events were scheduled and triggered correctly, updated game parameters like balance and satisfaction, handled positive and negative events properly, and worked in edge cases like no buildings being present.
FileHandlerTest.java	2	Pass	Checked that building data files loaded correctly with all details and textures, and that map files loaded properly for valid filenames while handling invalid filenames safely.
GameConfig Test.java	2	Pass	Make sure that game settings are saved and loaded properly, so changes are kept, and old values are replaced with the correct ones when loaded.
GameTimeTest.java	2	Pass	Checked that the game timer counts down correctly to zero and never goes below zero, even if decremented too much.
LeaderboardTest.java	5	Pass	Checked that leaderboards load correctly, new scores are ranked properly with a limit on entries, leaderboards save and reload accurately, and edge cases like empty or full leaderboards are handled well.
MoneyTest.java	5	Pass	Checked that placing buildings deducts the right cost, deposits and withdrawals work correctly, buildings earn money as expected, multipliers affect earnings properly, and earnings adjust when the player is in debt.
SatisfactionTest.java	6	Pass	Checked that satisfaction starts at the right value, stays between 0 and 100, adjusts correctly with

			bonuses and penalties, recalculates properly based on building placement, and handles duplicate buildings appropriately.
TestSuper.java	0	N/A	Ensures a consistent test setup, reusable methods for tasks like adding buildings, and predefined coordinates to simplify and streamline testing.
TimerTest.java	2	Pass	Checked that seconds convert correctly to “minutes:seconds” and that in-game time progression (semesters and summers) follows the right order.

Manual In-Person Test Report

Please see Manual In-Person Testing document for the report on the website.

Total test pass/fail pie chart & Bar chart of different method of testing:



[1]

Failed Test

Only Test 16 in Manual In-Person Test has failed.

Description - The system shall use a colour scheme that wouldn’t be confusing to colour blind people.

Result: The use of blue and green in the colour scheme might be problematic for some colour-blind users, potentially leading to confusion.

This test failed because the current colour scheme doesn’t work well for colour-blind users. While usability was tested manually, the colours need to be improved to be more accessible.

To fix this, we suggest adding a “colour-blind mode” with better colour options. This can be done by:

- Changing the colours to be high-contrast and easier to see.
- Adding a setting so players can turn on a colour-blind-friendly mode.

These changes would make the game more accessible and help pass the test.

Completeness and Correctness of Tests

Our testing approach focused on ensuring both completeness and correctness. We explicitly marked tests with their associated requirements to improve traceability and grouped them based on the part of the code they related to. For example, handlers, which manage most of the game's processing, were matched with a set of unit tests that explicitly validated each of that handler's functions. We considered a wide range of scenarios, including non-standard ones, such as events running without any buildings placed. This helped us verify the game's behavior across different and less typical situations.

When we encountered problems during playthroughs that weren't covered by existing tests, we created new ones to address them. Although automated test coverage for the UI was limited, extensive playthroughs provided confidence in those areas. Especially true as high code reuse also meant that errors in shared components would impact multiple areas, increasing the potential for error detection and increasing overall system confidence.

To ensure correctness, our tests were developed based on the system's requirements and intended behavior rather than its implementation. Having different people write the tests and implementation emphasized this distinction. Reusable methods were created for common actions within tests, reducing the risk of errors on each re-implementation. Complex tests often relied on simpler methods that had already been verified and used, ensuring a strong foundation for more complicated tests.

We used direct module instantiation wherever possible to limit the scope of tests and avoid external dependencies. However, in some cases, components relied on GameGlobals, which required a broader scope. Manual testing was often conducted in a group setting, providing opportunities for second opinions. Occasionally, tests did prove to be incorrect, which mandated a deeper analysis of both the tests and the code. Often this revealed issues in the code itself too, leading to a better understanding of the system's behavior and improving both the tests and application.

Link to Website Testing Material

https://alltheeb5t.github.io/assessment-2.github.io/assessment2/testing.html

References

Meta-Chart. "Pie Chart/Bar Chart." Accessed [date]. https://www.meta-chart.com/.[1]