Software Testing Report

Cohort 2 Team 1 (Assessment 1)
Ahmet Abdulhamit
Zoey Ahmed
Tomisin Bankole
Alanah Bell
Sasha Heer
Oscar Meadowcroft
Alric Thilak

Cohort 2 Team 2 (Assessment 2)
Bader Albeadeeni
Dan Hemsley
Jennifer Bryant
Mathilde Couturier-Dale
Oliver Elliott
Rosie-Mae Connolly
William Mutch

Software Testing Report

Summary of Testing Methods and Approaches

We used a hybrid testing approach for our game, combining automated and manual tests to test both unit-level code and user-facing features. Automated testing focuses on logic and game-flow components that can be reliably exercised without a rendering context, allowing many tests to be run quickly and repeatedly throughout development. Manual testing is used to validate user-facing features such as UI behaviour and rendering, which require human judgement and a graphical environment, for example usability and visual correctness.

We followed the **Test Pyramid** model[1], having many unit tests forming the base of pyramid, fewer integration tests in the middle, and even fewer manual end-to-end tests at the top:

- **Unit Testing**
    - **Purpose:** Testing isolated code units to ensure individual functions, methods, and classes work properly.
    - **Scope: Narrow**, focused only on pure logic and not involving external elements such as the UI and rendering.
    - **Example:** Testing score calculation logic, timer logic, leaderboard sorting algorithm, etc.
    - **Tools:** JUnit 5 for writing and executing unit tests.
    - **Approach:** Using Equivalence Partitioning to select inputs and Boundary Value Analysis to ensure edge cases are handled correctly.
- **Integration Testing**
    - **Purpose:** Verifying interactions and data flow between core game components, particularly between game logic and game-flow control.
    - **Scope: Medium**, integrating multiple real logic components while excluding rendering and asset-loading code.
    - **Example:** Testing interactions between menu logic, settings logic, and leaderboard logic to ensure correct game-flow decisions and data handling.
    - **Tools:** JUnit 5 for structuring and executing tests.
    - **Approach:** Integration tests focus on real interactions between logic components rather than mocked dependencies.
- **Manual Testing**
    - **Purpose:**Testing scenarios which are unreliable and impractical to automate.
    - **Scope: Large**, using the full system.running in a graphical environment.
    - **Example:** Testing user experience, UI responsiveness and stability, and Exploratory Testing to identify bugs.
    - **Approach:** A focus on user interactions, like key presses, inputs (especially edge cases), and game flow, to ensure an enjoyable user experience.

Tests were written using the DAMP (Descriptive And Meaningful Phrases) principle, so that they are self-evident and easy to understand. For instance, each test case has a clear, meaningful name, like deanMovesTowardsPlayer in the DeanTest. We also used Equivalence Partitioning and Boundary Value Analysis to efficiently select valid, invalid and extreme inputs so that tests cover a wide range of possible values.

JaCoCo was used for generating coverage reports, with coverage focused on logic classes instead of UI and rendering code. UI components depend on the LibGDX runtime, so are

therefore tested manually. Mocking was largely unnecessary, as most logic classes were self-contained and didn't have external dependencies.

Report on Actual Tests

Automated Test Summary

Automated tests are organised into high-level suites corresponding to the game's main logic components. Each suite is implemented as a test class targeting a specific area of functionality, e.g. timer logic, leaderboard logic, or goose logic. In total, 145 automated tests were written and executed, and all tests currently pass.

Statistics of Automated Tests

- Total Automated Tests: 145
- Pass Rate: 100%
- Line Coverage: 91%
- Branch Coverage: 83%

145	0	0	0.413s
tests	failures	ignored	duration

100% successful

Packages | Classes

Package	Tests	Failures	Ignored	Duration	Success rate
io.github.some_example_name	145	0	0	0.413s	100%

Code coverage was measured using JaCoCo, with results showing strong coverage across the core logic classes. UI-heavy classes were excluded from JaCoCo coverage reporting via the build.gradle configuration, as they primarily consist of rendering logic that is not suitable for unit testing. These components were instead tested using integration and manual tests.

core > io.github.some_example_name

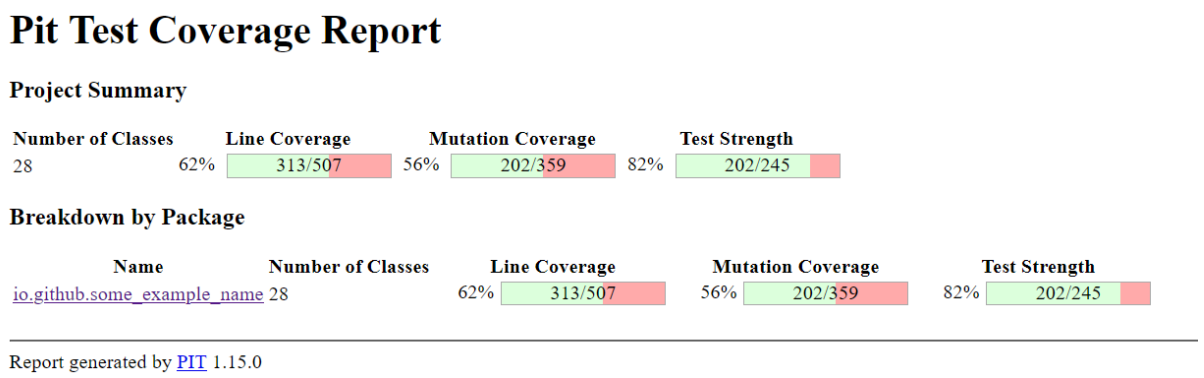
io.github.some_example_name

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
MyGame		21%		n/a	3	4	12	15	3	4	0	1
SettingsScreen.new ClickListener() {...}		0%		n/a	2	2	3	3	2	2	1	1
SettingsScreen.new ChangeListener() {...}		0%		n/a	2	2	4	4	2	2	1	1
WinScreen.new InputAdapter() {...}		0%		n/a	2	2	3	3	2	2	1	1
PlayerLogic		85%		88%	2	10	2	11	1	2	0	1
WinScreenLogic		92%		81%	5	18	3	30	1	7	0	1
JSONHandler		97%		80%	2	11	1	24	0	6	0	1
FinalScoreLogic		92%		100%	1	5	1	14	1	2	0	1
NPCLogic		88%		100%	1	7	1	8	1	2	0	1
MenuLogic		85%		100%	1	6	1	6	1	2	0	1
TutorialScreenLogic		76%		100%	1	4	1	4	1	2	0	1
Main		0%		n/a	1	1	1	1	1	1	1	1
SafeLogic		99%		53%	12	24	1	37	0	11	0	1
AchievementLogic		98%		75%	2	11	0	22	0	7	0	1
CodePageLogic		96%		87%	1	7	0	9	0	3	0	1
DeanLogic		100%		100%	0	8	0	21	0	6	0	1
GooseLogic		100%		87%	1	11	0	22	0	7	0	1
LockerLogic		100%		80%	2	11	0	22	0	6	0	1
QuizLogic		100%		100%	0	11	0	19	0	7	0	1
DeanRepellentLogic		100%		75%	3	13	0	17	0	7	0	1
GameTimerLogic		100%		100%	0	9	0	15	0	5	0	1
BookLogic		100%		83%	1	8	0	18	0	5	0	1
Player.Direction		100%		n/a	0	1	0	9	0	1	0	1
MenuLogic.Action		100%		n/a	0	1	0	6	0	1	0	1
SettingsLogic		100%		100%	0	5	0	9	0	4	0	1
TutorialScreenLogic.Action		100%		n/a	0	1	0	4	0	1	0	1
QuizLogic.AnswerResult		100%		n/a	0	1	0	4	0	1	0	1
GameOverInputLogic.Action		100%		n/a	0	1	0	4	0	1	0	1
PlayerLogic.Frame		100%		n/a	0	1	0	4	0	1	0	1
SlimeLogic		100%		n/a	0	3	0	7	0	3	0	1
LeaderboardEntry		100%		n/a	0	2	0	6	0	2	0	1
GameOverInputLogic		100%		100%	0	4	0	6	0	2	0	1
PlayerLogic.FrameState		100%		n/a	0	1	0	4	0	1	0	1
Total	129 of 1,538	91%	29 of 173	83%	45	206	34	388	16	116	4	33

Software Testing Report

Mutation testing was performed using PIT, to assess the effectiveness of the tests. The mutation report shows:

- Mutation Coverage: 56%
- Test Strength: 82%



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
28	62%	313/507	56%	202/359	82%	202/245

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
io.github.some_example_name	28	62%	313/507	56%	202/359	82%	202/245

Report generated by PIT 1.15.0

The high test strength indicates that most meaningful mutants were detected by the tests. Some surviving mutants are equivalent or exist in unimportant or UI-focused code paths, where mutant testing is not informative.

Completeness and Traceability
Test completeness was assessed using a traceability matrix, mapping test cases to functional areas of the system. Core gameplay logic is covered by automated tests, and UI and rendering behaviour is covered through manual testing. This division ensures high confidence in correctness while avoiding brittle or low-value automated UI tests. As no automated tests currently fail, the test suite is both complete and correct with respect to the game's requirements.

The testing traceability matrix can be found in the Testing Material section below.

Manual Test Summary

Manual testing focused on full game-flow scenarios and UI-heavy components that could not be automated practically or reliably. These tests were carried out in a graphical environment using the built game executable.

Manual tests were focused on:

- UI feedback and screen transitions
- Player controls and collision
- Audio and timing
- Full gameplay flow from start to win/lose screen
- Stability over long runtimes
- Accessibility and usability
- Offline execution

Each test case was documented with a unique ID, category, instructions, expected and actual outcome, and pass/fail status, ensuring traceability and clarity of results.

Software Testing Report

Out of the documented manual text cases, 8 out of 9 tests passed successfully. The single failing test relates to unlocking an achievement, where one achievement pop-up does not trigger because of incomplete implementation, rather than an issue in the testing process. Overall, the tests provide confidence that the game is stable and works correctly in real-world conditions.

Testing Material

The following testing materials are provided:

- Test Source Code: https://github.com/TheDebugThugs/Assessment_Part_2/tree/main/Game/core/src/test/java/io/github/some_example_name
- Automated Test Results: https://thedebugthugs.github.io/Assessment-2/assets/reports/test/index.html
- JaCoCo Coverage Report: https://thedebugthugs.github.io/Assessment-2/assets/reports/jacoco/html/index.html
- Mutation Testing (PIT) Report: https://thedebugthugs.github.io/Assessment-2/assets/reports/pit/pitest/index.html
- Manual Test Cases: https://thedebugthugs.github.io/Assessment-2/assets/Manual-Test-Cases.pdf
- Traceability Matrix: https://docs.google.com/spreadsheets/d/1glTDR1mG-X9hMZCe9fLkYBJO096aJf1iZztUK3g3luE/edit?usp=sharing

Overall, the current testing approach provides strong confidence in correctness, stability, and usability of the game. Future improvements to the testing process could include expanding integration tests to cover more game-flow scenarios, and refining mutation testing by excluding additional equivalent mutants.

References

[1] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*. Boston, MA, USA: Addison-Wesley, 2009.