Software Testing

Cohort 2 Team 4

Abdul Fofanah
Matthew Holleran
Toby Watchorn
Daneena Roydean
Arwen Minton
Jess McKerill
Henry Bambrough

1.0 Introduction

The testing for this project was conducted using a combination of manual testing for gameplay events and automated unit testing with JUnit. This approach was chosen to make sure there was full game coverage for the core game logic and the interactive, visual scenarios that could not be verified to function with automated tests alone.

Manual testing was used for the features that involve textures, animations, collisions and movement within the game. These tests were conducted by playing the game and taking screenshots before, during and after specific events. This provided clear evidence showing the changes that occur and ensuring they work as intended and in a repeatable manner.

Automated testing was carried out using JUnit with JaCoCo to test parts of the game that do not depend on graphics or input. These were to test logic, such as Score updates and Achievements. These tests work at a small scope, targeting individual classes and methods, making failures easier to identify and reducing effects from outside sources. JaCoCo was used to help guide us on which parts of the game had not been sufficiently covered and needed additional manual tests. Overall, the testing approach was driven by the project requirements, ensuring that each one is tested either manually or automatically, with clear traceability and expected outputs.

2.0 Automated Tests Report

The frameworks used are JUnit 5.9.3, Mockito 5.3.1, JaCoCo 0.8.10, and HeadlessLauncher (LibGDX). Local test execution results show a total of 228 tests, passed tests of 228 (100%), failed tests of 0 (0%), ignored tests of 0, duration 8.600 seconds, and a success rate of 100%. CI test execution results show a total test of 228, passed tests of 222 (100%), failed tests of 0 (0%), disabled tests of 6 (DrownTest cases due to CI environment incompatibility, and a success rate of 100% active tests. Diagram 1 shows the test distribution by 17 test classes and the test coverage produced by JaCoCo Report. All 228 tests passed consistently across multiple local runs, with 222 tests passing in CI environments. Since no tests failed, the following pages analyse test completeness and correctness, covering areas where tests are testable and verifying tests' behaviour.

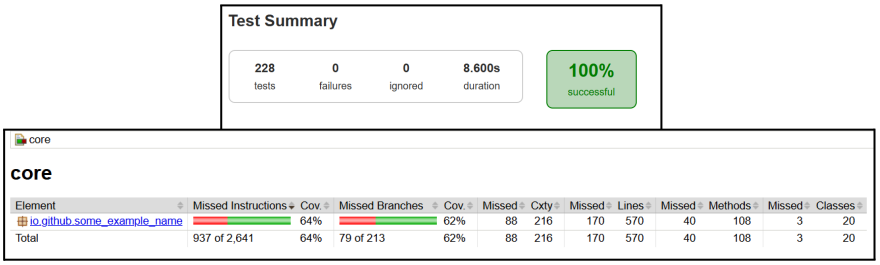

Diagram 1: Test distribution and coverage of 17 test classes on local

3.0 Test Completeness Analysis

Since all tests passed, we assess whether the test suite adequately covers the codebase and requirements. Well-covered classes that have a range of 80%-100% coverage have comprehensive test coverage with all testable logic verified. Classes that are appropriately covered from 60%-79% have good coverage of testable logic that's left untestable for rendering methods. Partially covered classes are somewhere along the lines of 50% coverage. Low coverage classes cover a percentage of 32% or below. Table 1 shows the test completeness of each class.

Drown (91% lines, 83% branches) ● All logic is fully tested, including the constructor, collision, detection, respawn,	**Achievement (100% lines)** ● Consists simple data class without conditional logic

and positioning ● 8 tests to verify water hitbox, respawn position, and -10 penalty score, where 6 were disabled in CI due to mock incompatibility ● Pure game logic ● Completeness: Edge case testing (exact boundaries and multiple drownings) passes and works locally	● 12 tests for positive/penalty/zero scores, null handling, and field access ● Completeness: All edge cases tested, including values with boundary conditions
Patrol_Dean (89% lines, 100% branches) ● 13 tests to verify patrol position, boundary reverse, and blocked cell handling ● Only *render()* and *dispose()* untested (LibGDX class) ● Completeness: All branching logic tested, movements verified	**MyGame (100% lines)** ● 8 tests that cover player name management, first name, last name, and null handling ● Completeness: Complete coverage for name logic
Player (77%), Dean (78%),Locker (75%), Slow_Down (75%), Extra_Time (73%), Freeze_Dean (70%), Decrease_Time (69%), NPC (63%) ● The average coverage is 71% for game mechanics classes with testable logic for constructors, collision detection, timers, state management, and use of E key ● Untestable parts are *render()*, *dispose()*, texture, animation frames ● Completeness: Only the visual rendering needs manual testing	**Questionnaire (57% lines, 70% branches)** ● 24 tests cover complex logic like E key activation, quiz answer validation, spawn patrol dean, and freeze dean ● The high branch coverage of 70% confirms with thorough logic testing ● Untestable parts are text display, result message rendering, and quiz UI rendering ● Completeness: All answer validation logic tested, whereas UI rendering needs manual testing
BusTicket (32% lines, 0% branches) ● The isDiscovered field is only set during *render(),* so it can't be tested without visuals ● Tested the constructor, ticket position, and collection states ● Discovery state change and visual rendering remain untested ● Completeness: All testable logic covered, but the discovery state needs visual	**GameScreen(4% lines, 0% branches)** ● Completeness: Only the testable method is fully covered; the rest is UI **GameTimer (0% lines, 0% branches)** ● Completeness: Architecturally limited for TimerLogic and TimerUI; this was tested and verified manually **Save_Leaderboard(0% lines, 0% branches)** ● Completeness: Data model only due to the primary UI/game loop class; this was tested and verified manually

Table 1: Test Completeness

4.0 Assessing Test Correctness

To verify tests, we validate correct behaviour rather than false positives. That includes validation techniques such as *assertEquals()*: matches the exact value of positions, scores, and timers. To verify a boolean state, *assertTrue()* or *assertFalse()* is implemented and to verify interaction, we use Mockito: *verify(mock, times(N))*. Reflection is used to verify private fields. For instance, to prove correctness can be seen in the Questionnaire Test, where answer C triggers *freezeAllDeans()* (correct answer). Answers A/B/D trigger *spawnSecondDean()* (penalty). To ensure the correct method is called exactly once, and wrong methods are never called, this is achieved through mock verification. Preventing false positives can be seen in the tests, such as mock verification strictness (*times(1)* vs *never()*),

exact value assertions where it fails when the errors encountered are off by one, negative testing to verify that wrong behaviour shouldn't occur, and state consistency, where some tests consist of multiple related assertions.

17 functional requirements verified through a total of 228 automated tests, with 222 active in CI. Positive events with 44 tests are Freeze_Dean, Extra_Time, and Locker. Negative events with 73 tests are Dean, Patrol_Dean, Slow_Down, Drown, and Decrease_Time. Hidden events with 47 tests are Questionnaire, BusTicket, and Teleport. Core mechanics with 64 tests are Player, Achievement, GameScreen penalty calculation, NPC, and MyGame. 100% requirement coverage through combined automated and manual testing.

5.0 Manual Testing Approach

Manual testing was used throughout the development of our project to ensure that it verifies UI components, gameplay mechanics, visual elements, and bugs that are untestable in automated testing. The methods *render()* and *dispose()* were tested manually.

5.1 Gameplay Events Testing

All events were tested manually to make sure there were no visual glitches that occurred when triggering them. That includes Bus Ticket, Dean, Decrease Time, Extra Time, Drowning, Freeze Dean, Slow Down, Teleport, Questionnaire, and Patrol Dean.

5.2 Game Screens and UI Testing

The game consists of five screens: the start screen, main gameplay screen, Name character screen, and final end game screen. Each was manually tested to ensure a smooth transition from one screen to another and a consistent UI display. There is also a pause game feature that was tested manually as well.

5.3 Interface and Progression Features

- **Tutorial screen**: Tested for clarity, accessibility, and readability so users can easily understand the game.
- **Start screen:** Tested that the game started correctly, allowing the user to start the game.
- **Name screen**: Verified correct input handling and made sure the name displayed correctly on the end screen.
- **Leaderboard/End screen**: Tested to see if the name was correct and if the score displayed was accurate to what should have been displayed.
- **Game Screen/Pause:** Tested that when the user moved around and interacted with events that no bugs were present, as well as pressing the 'P' key to pause, game elements that include the whole gameplay were frozen and can be resumed by pressing the 'P' key again.

6.0 Testing Deliverables and Evidence

In accordance with the Assessment 2 brief, the precise URLs for the automated test results, coverage reports, and manual test case descriptions are provided below:

- Test Execution Results: https://mattholl26.github.io/assets/test-reports/index.html
- JaCoCo Coverage Report: https://mattholl26.github.io/assets/coverage-reports/index.html
- Test Source Code (GitHub): https://github.com/MattHoll26/Engineering-1/tree/main/Game/core/src/test/java/io/github/some_example_name
- Manual Test Case Descriptions: https://mattholl26.github.io/assets/PDFs/ManualTest.pdf