Requirements

Cohort 1, Group 6 - M6

Members:

Mir Baydemir
Adam Fraulo
Azib Hj Abu Akmar
Agata Pittarello
Esther Scanlon
Jazz Stubbins
Sonia Szetela

Testing methods and approaches

To test our game, we used automated unit testing as our main method since it's the most practical way to test game logic without manually playing through repeatedly. Unit testing is good for smaller projects as it allows us to verify individual components quickly and catch bugs early [1]. We implemented unit tests using JUnit 5 with a headless testing approach using a custom GdxTestRunner class [8]. This creates a mock graphics context so tests can run without a real display.

We used Mockito for mocking dependencies. Mock testing is ideal for our project because it allows us to test individual systems in isolation without needing the entire game engine running, keeping our tests fast and focused [3]. For example, testing if picking up a key correctly calls the rendering system to hide the key sprite without actually rendering anything.

We created thorough unit tests for core game mechanics, including boundary/edge case testing and state based testing. Boundary testing is particularly valuable in games for checking collision detection, movement limits, and inventory capacity, areas where off by one errors commonly occur [4].We stuck to the result/state test pattern for the majority of our unit tests which works well for straightforward game logic where we can easily verify the game state after an action [5].

We started with exploratory testing. Valuable at the beginning, helped us understand the existing codebase structure and identify potential problem areas, efficient for small scale projects [6]. This also helps find any bugs or pieces of code which may make development more difficult so we could refactor them. We then went through and created thorough unit tests for main functionalities, following best practices of writing clear, maintainable tests that verify one thing at a time [2]. We then kept this approach up as new code was added.

As we developed the game, regression testing became increasingly important to ensure our changes didn't introduce new bugs [1]. Since this project involves visuals difficult to unit test in a meaningful way, we chose to test the rendering system mostly manually, and mocked it for other test classes. We also tested manually by playing the game through the whole testing process. This hybrid approach of automated testing for logic and manual testing for visuals is a practical strategy for small game projects where visual bugs are best caught through human observation, while logic errors benefit from automated verification [3].

We used integration testing to test the functionality of the game, ensuring individual components work together correctly. Even in small games systems like collision detection, player movement, and item collection need to interact properly [7]. This uses the same headless testing approach, whilst using actual map data to ensure the tests work the same way as the game. We waited to do these until implementation was done with the majority of their components.

To measure how well our tests cover the codebase, we integrated Jacoco coverage reporting. This showed us which lines of code are tested and which aren't, helping us identify gaps in our testing. We also ran tests with coverage locally (through our IDE) while developing our testing to check coverage easily as we went along.

Test execution statistics summary[10][11]

We have 83 automated tests across 15 classes. All of them pass. According to jacoco we have 44% instruction coverage and 38% branch coverage. These are lowish since we focused our testing on general functionality and critical systems, and filling in harder to auto test gaps with manual testing (e.g specifics of rendering). We also didn't test files from libGDX that we didn't edit.

By package:

Main class: 67% instruction coverage, 38% branch coverage; Entities package: 50% instruction coverage, 27% branch coverage; Systems package: 32% instruction coverage, 46% branch coverage. The systems package has the lowest code coverage since we chose to test most aspects of the rendering system manually (and its the largest chunk of the systems package).

CollisionSystem (88% instruction, 70% branch): Every collision detection method is tested using actual map data from testMap.tmx. This is important because if collision detection breaks, the game is unplayable, and messed up collisions can be hard to find through manual gameplay.

TimerSystem (100% coverage): All timer logic is tested including countdown, gradual time addition, and display formatting. We even test edge cases like what happens when time goes negative, which could crash the game if handled badly. Easy to unit test but hard to spot in manual testing if time is calculated wrong.

ToastSystem (85% instruction, 64% branch): Toast creation, expiration, colours, and the new bubble toast feature are all tested. This is less critical than collision but still important for player feedback, especially since its very relevant to a few main requirements.

Entity (96% instruction, 75% branch): The base entity class is thoroughly tested for positioning, collision detection, and scaling since all game objects inherit from it.

Player (65% instruction, 56% branch): Most player mechanics are tested including inventory (keys, potions), the 15 second invisibility timer, and state changes. This is important because the player is the main character.

RenderingSystem (0% coverage): Mocking LibGDX's shader system isn't practical, and visual bugs (like misaligned sprites or incorrect layers) wouldn't crash the system but would be immediately obvious during manual gameplay.

Dean character (29% instruction, 8% branch): We test basic reach detection (the Dean can reach the player at 47 pixels but not 48 pixels) and path restarting, but the complex pathfinding algorithm isn't covered. The behaviour is difficult to test in isolation without running the full game loop, and honestly we focused testing efforts on player facing mechanics that would break gameplay more obviously.

MovingEntity (24% instruction, 0% branch): The base move() method involves complex collision checking that's better tested through integration tests of Player and Dean movement rather than testing the abstract parent class. Testing it in isolation wouldn't prove anything useful.

InputSystem (80% instruction, 58% branch): We test name input (including special characters, backspace, and enter) and the pause key functionality, but we don't have comprehensive tests for movement keys, interaction keys, or dialogue choices. These get verified during manual gameplay testing instead.

LeaderboardSystem (33% instruction, 40% branch): Main functionality works, we test score sorting, top 5 selection, and file I/O. We even have a regression test for the bug where reading the leaderboard multiple times would duplicate entries.

TriggerSystem (33% instruction, 32% branch): We test the trigger infrastructure (adding, removing, detecting collisions with triggers) and verify that trigger 1 (chest room key) works correctly (overall trigger testing rather than test per trigger).

AnimatedEntity (75% instruction, 66% branch): We test changing animations, playing and pausing animations, and retrieving current frames) and verify that switching between different animations does not cause crashes, including various edge cases (repeating the same animation, first and last animation). The focus is on overall animation-behaviour testing rather than testing individual animations in isolation.

Integration tests

SystemIntegrationTest: Tests coordination between systems including Dean catching with timer penalties, invisibility protection, pause freezing entities, achievement counter updates, score calculation with bonuses, and multi-door mechanics.

GameFlowIntegrationTest: Tests realistic complete playthroughs including collecting keys and winning, chest room sequence, longboi event chain, invisibility scroll usage, all game state transitions, and perfect run with all collectibles.

Tests in relation to requirements (some up to interpretation requirements omitted)

ID	**Automated Tests**	**Manual Tests**
UR_VICTORY_CONDITION	TimerSystemTest tests timer logic; full victory condition is manual, GameFlowIntegrationTest.testKeyAndWin() tests complete victory flow	Play through and make sure game is winnable, try exit without key
UR_SCORE	MainTest.calculate() MainTest.longboiScore(), SystemIntegrationTest.testScoreIntegration() and testScoreNoBon()	Calculate expected score, play through game, check if expected score achieved (repeated with multiple achievement combinations)
UR_EVENTS	TriggerSystemTest tests infrastructure; specific event counts verified manually, SystemIntegrationTest.testCountersUpdate() verifies event counting	Trigger all possible events
UR_ACHIEVEMENTS	MainTest.longboi() tests longboi achievement bonus	Achieve all achievements through play through
UR_LEADERBOARD	LeaderboardSystemTest.topFiveEntries() formatLeaderboard() correctInsertion() multiReadBoard()	Check leaderboard at end of a run and rerun
UR_TIMER	TimerSystemTest (all 5 tests) for logic; visibility tested manually	Play game and look at timer
UR_PAUSE	MainTest.pauseTest(), SystemIntegrationTest.testPauseSystem() (in addition to existing MainTest)	Play and pause the game repeatedly

UR_TRACKER	N/A	Play game and confirm collected items are accurate with expected
UR_INSTRUCTIONS	N/A	Read instructions on main screen
UR_GAMEPLAY	MainTest.gameStates() tests state transitions; invalid input handling mostly manual	Play through the game, spam keys,
FR_EVENT	TriggerSystemTest.checkTouchTriggers() checkInteractTriggers() trigger()	Fully automated
FR_EVENTS_LOCATION	Dean movement tested, SystemIntegrationTest.testDeanCapture() tests dean-player interaction	Confirm dean movement changes in run throughs
FR_TIMER	TimerSystemTest.add() timeCheck() getTimeLeft() getClockDisplay() addGradually()	Play game and look at timer
FR_MAP	CollisionSystemTest.testInit() loads map; visibility tested manually	Walk around the whole map to check for glitches/movement issues
FR_MOVEMENT	InputSystemTest.pressingPForPause() tests pause key; WASD movement tested manually, SystemIntegrationTest.testAtticCrawlspace() and testGoDownAttic() test layer transitions	Check you can walk using WASD
FR_SCORE_CALC	MainTest.calculate() longboiScore(), SystemIntegrationTest.testScoreIntegration() tests score with timer	Play runthrough with multiple routes
FR_RESET	N/A	Play the game 3 times with all exit routes (force quit, win, lose) and check everything resets correctly each time.
FR_INSTRUCTIONS	N/A	Read on screen instructions
FR_PAUSE_MENU	N/A	Make sure pause menu works throughout
FR_START_SCREEN	InputSystemTest.nameInput() tests name entry; full start screen UI is manual	Check you can input your name and navigate start screen
FR_ENDING_SCREEN	MainTest.gameStates() tests win/lose state transitions; UI display tested manually	Check last screen after you win and lose the game
NFR_RESILIENCE	CollisionSystemTest.removeCollision() tests removing nonexistent collision; PlayerTest tests duplicate key giving,	Try breaking the game with unexpected choices (exploratory testing approach)

Exact manual testing approach and results[12]

We did some manual testing since aspects of the game (graphics, UI, gameplay feel) can't really be tested effectively with unit tests. We split into two approaches: targeted requirement testing and full playthroughs. For targeted testing, we went through each requirement that needed manual verification (like UR_VISIBILITY, NFR_UI, FR_MOVEMENT) and specifically tested those aspects. We also did scenario based testing for specific mechanics. Then we did multiple full completion runs of the game from start to finish, playing through both win and lose scenarios to ensure everything works together as a cohesive experience. This caught issues our unit tests didn't find, like the leaderboard bug where entries were duplicating after multiple playthroughs (which we then wrote a regression test for), as well as general aesthetic issues like text box sizing. [9]

References:

[1] GeeksforGeeks, "Software Testing Techniques," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/software-testing/software-testing-techniques/. [Accessed: 28-Nov-2024].

[2] Thoughtworks, "Write Better Tests in 5 Steps," Thoughtworks Insights. [Online]. Available: https://www.thoughtworks.com/en-gb/insights/blog/write-better-tests-5-steps. [Accessed: 28-Nov-2024].

[3] Tutorials Point, "Software Testing Methods," Tutorials Point. [Online]. Available: https://www.tutorialspoint.com/software_testing/software_testing_methods.htm. [Accessed: 28-Nov-2024].

[4] BrowserStack, "Software Testing Strategies and Approaches," BrowserStack Guide. [Online]. Available: https://www.browserstack.com/guide/software-testing-strategies-and-approaches. [Accessed: 28-Nov-2024].

[5] Raygun, "Unit Testing Patterns," Raygun Blog. [Online]. Available: https://raygun.com/blog/unit-testing-patterns/. [Accessed: 12-Dec-2024].

[6] BrowserStack, "Exploratory Testing," BrowserStack Guide. [Online]. Available: https://www.browserstack.com/guide/exploratory-testing. [Accessed: 15-Dec-2024].

[7] BrowserStack, "What is Integration Testing," BrowserStack Guide. [Online]. Available: https://www.browserstack.com/guide/integration-testing. [Accessed: 03-Dec-2024].

[8] T. Grill, "gdx-testing," GitHub repository. [Online]. Available: https://github.com/TomGrill/gdx-testing. [Accessed: 24-Nov-2024].

[9] "How to practically playtest your game," Game Design Skills. [Online]. Available: https://gamedesignskills.com/game-design/playtest/. [Accessed: 03-Jan-2025].

[10] JUnit results on our website

[11] Jacoco results on our website

[12] Manual test cases on our website