SOFTWARE TESTING REPORT

GROUP 5 - BITCRUSHED BOB

Maryam Mathews
Joseph Hinde
Jacob Mace
Will Aston
Zathia Jacquesson-Ahmad
Bulganchimeg Munkhjargal
Evan Weston

Methods and Approaches:

For this project, testing is primarily focused on ensuring functional correctness, completeness, and appropriateness of game mechanics - in line with the functional suitability quality characteristic from ISO 25010 software quality standards. We used a combination of unit testing, component-level integration testing, and exploratory testing to balance automated verification with human-judgement-based evaluation.

Our automated testing is primarily dynamic testing implemented using JUnit, where systems are executed and their behaviour is checked against expected outcomes. We focused on testing at the system/component level, particularly for the gameplay systems we modified. Many narrow-scope tests were used to cover individual methods and edge cases within each system. Some behaviours, such as rendering and real-time user interaction, were tested manually due to their reliance on graphics, timing, and player perception.

This approach is particularly appropriate for a small-scale maze game. The game's mechanics are implemented as discrete, well-defined systems which makes them well-suited to isolated unit and component testing. At the same time, the player experience depends heavily on how these systems combine in real time, so manual and exploratory testing is necessary to confirm that the game is coherent when played.

Unit Testing:

The majority of testing (around 70%) was performed using unit and component tests. Core gameplay systems were tested in isolation by supplying them with controlled inputs and verifying their outputs or state changes. Rather than aiming for maximal line coverage, we adopted a risk-based testing strategy, focusing automated unit tests on the most critical and failure-prone gameplay systems.

Integration Testing:

We performed component-level integration testing by running real ECS entities and components together inside a system, verifying that they interact correctly. While full engine-level integration was not used in automated tests, these component integrations ensured that game logic behaved correctly across system boundaries. End-to-end integration was validated through manual playtesting, exercising the complete game loop from user input to visual and audio output.

Exploratory / Manual Testing:

Manual testing was used to validate playability, responsiveness, and game flow, which cannot be fully verified through automated tests. This involved playing the game in different scenarios, checking how it responds to user input, and observing whether behaviour matched design expectations.

Test Doubles:

To test individual components of our LibGDX maze game without the overhead and nondeterminism of rendering and physics, we replace external dependencies with test doubles. We use LibGDX's headless backend as a fake for most systems, and mocks to verify interactions (such as physics and rendering calls) in cases where game state cannot be directly observed.

Tests Report:

Overview:

Automated tests were written for 8 of the major gameplay systems in the project: PlayerSystem, TimerSystem, SurveyorSystem, LecturerSystem, InteractableSystem, GooseSystem, SlipTriggerSystem and HiddenWallSystem.

These systems were selected because they implement the core gameplay mechanics and were the systems most affected by the team's modifications and extensions. They also contain the majority of the game logic that can be reliably tested without requiring graphical or physics engines to run.

In total, 45+ automated tests were written across these systems, covering the most important behavioural logic. All written tests passed in the current implementation.

PlayerSystem:

- Methods: 16, Methods Tested: 12, Tests Written: 8, Result: All Tests Passed

These tests covered most of the different possibilities of how the method may be called. In the case of the processClockCollect, processLecture, processCoffeeCollect, footsteps and gooseBite all of the possible possibilities were covered as the method did not contain many variables.

The methods addKnockBack and movementAndAnimation were covered but had configurations that were not tested. For example the movementAndAnimation was not tested for every direction and the addKnockBack also did not test for only adding vectors in the y direction. We are assuming in these cases that since the code works for one direction and is identical when operating with the other direction that it will remain accurate.

TimerSystem:

- Methods: 2, Methods Tested: 2, Tests Written: 3, Result: All Tests Passed

The tests covered timer counting down and expiring, the timer not counting down when the isRunning flag was false and the timer not counting down when the hasExpired flag was true. These tests cover all of the cases we decided are important in the timerSystem.

SurveyorSystem:

- Methods: 20, Methods Tested: 16, Tests Written: 10, Result: All Tests Passed

The tests covered the enterWander, enterChase, enterRetreat, processWander, processChase, processRetreat, randomiseWaypoint and resolveAnimation. The enterWander, enterChase and enterRetreat methods covered all of the possible configurations in the game where the surveyor would enter a different state. The processWander, processChase and processRetreat all covered the reasonable cases in the game where they would be called. The randomiseWaypoint also covers the reasonable

configurations of the game when it could be called. The resolveAnimation test covers all the cases in which a surveyor would enter and idle state but not all of the possible ways in which it could enter a walk state. This is because there would be 4 vector direction * 4 idle states or 16 different cases to be written which was unnecessary for the scale of the project. The covered cases and manual evaluation provide enough extra information to assume the code is working correctly.
The untested methods involve a separate window being opened and were decided to be tested with a manual evaluation.

<u>LecturerSystem:</u>

- Methods: 21, Methods Tested: 16, Tests Written: 10, Result: All Tests Passed

The tests covered enterWander, enterChase, enterRetreat, processRetreat, processChase, processWander, randomisedWaypoint and resolveAnimation. The enterWander, enterChase and enterRetreat are identical and cover all the cases where they will be used in the game. The processWander, processChase and processRetreat all cover the reasonable cases when they would be called. The randomised waypoint covers all the configurations where it would be called. The resolve animation test is also identical and covers all possible ways the Lecturer can enter the idle state but only covers a section of the ways the lecturer can enter the walk state due to the larger number of cases that would have to be tested.

<u>InteractableSystem:</u>

- Methods: 3, Methods Tested: 3, Tests Written: 1, Result: All Tests Passed

The InteractableSystem uses 1 method and 1 test was written. The test that was written passed. The test checked that the Interactable would properly send messages to the player, include any additional messages and then remove the entity and disable its interactions. This covered all of the features that the interactable system provided.

<u>GooseSystem:</u>

- Methods: 15, Methods Tested: 15, Tests Written: 10, Result: All Tests Passed

The tests covered enterWander, enterChase, enterRetreat, processRetreat, processChase, processWander, randomisedWaypoint and resolveAnimation. The enterWander, enterChase and enterRetreat are identical and cover all the cases where they will be used in the game. The processWander, processChase and processRetreat all cover the reasonable cases when they would be called. The randomised waypoint covers all the configurations where it would be called. The resolve animation test is also identical and covers all possible ways the goose can enter the idle state but only covers a section of the ways the goose can enter the walk state due to the larger number of cases that would have to be tested.

<u>HiddenWallSystem:</u>

- Methods: 3, Methods Tested: 2, Tests Written: 2, Result: All Tests Passed

The tests check that when a wall has the hidden wall component and receives a correct trigger message the wall will remove its own wall entity. The second test checks that if the wall receives a trigger message with the wrong information it will not remove the wall entity. This covers all of the interactions.

SlipTriggerSystem:

- Methods: 4, Methods Tested: 3, Tests Written: 1, Result: All Tests Passed

The tests covered fixedUpdate. This covers the interaction between the player and the wet floor, displaying the wet floor message and the wet floor achievement. The tests cover the cases of the player entering the wet floor, leaving the wet floor and returning to the wet floor to check that the message is not displayed twice.

Coverage and Completion:
The core gameplay systems achieved approximately 63% line coverage. The remaining uncovered code falls into three categories:

1. UI-based or window-based methods:
- (eg. survey questions and combination locks). These require graphical input and cannot be reliably unit-tested, so they were manually tested

2. Library-provided functionality:
- Many systems depend on LibGDX and Ashely (com.badlogicgames.ashley:1.7.3) for physics, rendering, audio, and entity management. These libraries are assumed to be correct and were not re-tested

3. Symmetric or repetitive logic:
- Some logic (such as movement and animation by direction) was partially tested as the same code is reused for all directions.

Testing Material:

Automated Testing Coverage Report:
https://escape-from-uni.github.io/web/Jacoco-report/index.html

Manual Testing:
https://escape-from-uni.github.io/web/assets/Manual%20Test-Cases.pdf