Phase 3 Report: Testing

In this phase, we focused on testing the functionality of our game and making necessary modifications to ensure it worked smoothly and correctly. Our team worked together to carefully identify different parts of the game that needed testing, including the core features like player movements, enemy behavior, and game mechanics, as well as the interactions between different components such as graphics, sounds, and input handling. Each team member focused on specific areas of the game, testing methods, and fixing any issues they found.

To do this, we used unit testing to look at individual pieces of the game in isolation, making sure each part worked properly on its own. Integration testing was used to check how different parts of the game worked together, like how the player's movements interacted with enemies or how the game ended when certain conditions were met. As we wrote the tests, we discovered areas of the game that needed improvement. For instance, some methods needed optimization to handle edge cases, and certain features, like stopping background music, had to be modified to handle null cases gracefully.

Through this process, we learned the importance of testing every small part of the game and how focusing on specific methods helped us catch bugs we might have missed otherwise. By breaking the game into smaller parts and testing them one by one, we were able to pinpoint exactly where problems occurred and fix them. Writing tests not only helped us find and fix bugs but also made us think more deeply about how each part of the game should work. This made the overall game more robust and ensured a better experience for the players.

Unit Tests:

In this part we focused on testing individual features of the game in isolation to ensure they work as expected. Here are the main features we tested:

One of the first features tested was player movement, which is critical to the gameplay. We needed to ensure that the player could move up, down, left, and right when the appropriate keys are pressed. The tests checked whether the player's position on the X and Y axes updated correctly for each direction. Additionally, we tested diagonal movement to confirm the player could move simultaneously in two directions when required.

Another key feature was collision detection, which prevents the player from moving through obstacles like walls. This test ensured that the player’s position did not update when a collision occurred. We also verified that the game responded appropriately to collisions, such as triggering a "game over" state when the player collided with an enemy or harmful object.

Next, we tested the game scoring system, which increases the player's score when they collect items or defeat enemies. These tests ensured that scores updated correctly under the right conditions and remained unaffected by invalid actions, such as attempting to collect an item that was already taken.

We also focused on enemy behavior, ensuring that enemies moved and acted according to the game logic. For example, tests checked whether enemies moved toward the player when within range and followed predefined movement patterns when the player was far away. This feature was tested to confirm that enemy behavior remained consistent under different game scenarios.

We tested the game timer, which is used for time-based events like level countdowns or spawning new enemies. These tests validated that the timer counted down accurately and that the events triggered by the timer occurred as expected. This was crucial to ensure the game's pacing felt balanced and fair.

We also tested the functionality of our EndMenu. We checked that all key components, such as labels displaying messages like "Game Over" and "Score," were created correctly. This included verifying the display of game statistics, such as the score and time, and ensuring that interactive elements, like the "Play Again" and "Quit" buttons, were not null. Another test ensured that the appropriate message either "You Win!" or "You Lost!" was displayed based on the game outcome. These tests validated that the dialog provided accurate feedback to players under different conditions. We tested the functionality of the buttons in the EndMenu. The "Play Again" button was clicked to ensure it started a new game, and we verified this by checking whether the content pane of the parent frame switched to the game view. Similarly, the "Quit" button was tested to confirm that clicking it returned the player to the main menu.

The MenuTest class tests the main menu of the game to ensure all features work properly. One of the key tests checks if the menu follows the Singleton pattern, meaning only one instance of the menu can exist at a time. Other tests focus on the visual elements, such as ensuring the menu size matches the screen size and that the background image loads correctly. Additionally, the tests check the functionality of the buttons, like making sure the "Play" button only works when the player has selected a difficulty and verifying that the correct difficulty is saved when selected by the player. These tests make sure the menu is user-friendly and behaves as expected.

Integration Tests:

In Integration testing we focused on verifying interactions between different components of the game to ensure they worked together smoothly.

One of the most critical interactions tested was between the player and game logic. For example, when the player collided with an obstacle, the tests checked whether the collision correctly triggered the game logic, such as stopping the player's movement or reducing their health. Similarly, we tested whether player actions, like collecting an item, updated the score and removed the item from the game world.

Another important interaction was between the player and enemies. These tests checked whether enemies reacted appropriately to the player's presence. For instance, if the player entered an enemy's detection range, the tests verified that the enemy started chasing the player. We also tested how the game handled situations where the player and enemy collided, ensuring the correct response, such as the player losing health or the game ending.

Also, we tested the interaction between the game timer and the rest of the system. For example, the tests verified that new enemies spawned at the correct intervals and those time-based challenges triggered appropriately. This ensured that time-sensitive events in the game functioned as intended.

Test Quality and Coverage:

We tried our best to write clear and specific assertions to ensure each test case reliably validated the expected behavior. For example, instead of just checking if a value changed, we confirmed it changed to the correct expected value. We also used descriptive test names to make it easy to understand what each test case was verifying. To ensure high test quality and coverage, we used JaCoCo to examine our test coverage in order to perform White Box Testing. Examining the test coverage helped us identify the missing branches and lines in our tests. Adding tests for the missing branches ensured thorough testing of the internal code of the game allowing us to identify bugs in the code.

Our testing efforts demonstrated coverage across the main components of the game. The com.sfu package achieved 85% line coverage and 76% branch coverage, reflecting comprehensive testing of core game logic. However, the branch coverage indicates room for improvement, particularly in conditional paths and edge cases.

The com.sfu.Entities package excelled with 91% line coverage and 83% branch coverage, showcasing thorough testing of critical elements like player and enemy behaviors. While these numbers are high, additional branch testing could target rare scenarios or complex interactions between entities.

The com.sfu.Tiles package achieved impressive results with 94% line coverage and 84% branch coverage, indicating strong testing for tile-related functionality, such as collision detection and

environmental interactions. Efforts to increase branch coverage could focus on additional edge cases involving unique tile setups or boundary conditions.

The com.sfu.Items package achieved 92% line coverage, emphasizing effective testing of item collection and scoring logic. Although branch coverage data for this package is unavailable, the current results highlight successful validation of primary item-related features.

Improvements:

During the testing phase, several improvements were made to enhance the game's functionality, performance, and user experience. One of the things we improved was the player the collision detection to address overlapping boundaries and reduce false positives or negatives by altering the sizing of the collision boxes. Timer accuracy was improved by refining countdown intervals and adding safeguards to prevent premature or missed triggers. Enemy behavior algorithms were also optimized for responsiveness while minimizing computational load. We also improved the quality of our production code based on the tests. Writing tests for enemy behavior helped us identify inconsistencies in their movement patterns, leading us to refactor the code for better clarity and performance. Error handling was enhanced across the game, particularly for null cases in features like background music and menu components, to prevent crashes and ensure seamless functionality. The menu was further improved to address unresponsive buttons and ensure proper synchronization during navigation.

Test coverage was significantly strengthened by increasing branch testing in critical areas, such as player actions and item interactions, and expanding integration tests to cover complex scenarios involving multiple simultaneous interactions. Game balancing adjustments were made to scale difficulty appropriately, refine enemy spawn rates, and improve pace for a more engaging experience. Additionally, menu accessibility was improved by enhancing layout responsiveness and ensuring proper rendering on different screen resolutions. To address feedback from JaCoCo analysis, additional tests were written to cover uncovered branches and lines in key packages, ensuring comprehensive validation of game logic and environmental interactions. Visual and audio elements were also enhanced, with optimized sprite rendering to maintain consistent frame rates and added fade effects for smoother audio transitions. Finally, the codebase was refined through better documentation, detailed comments, and the removal of redundant methods, ensuring improved readability and maintainability for future updates.

A common bug that was found throughout the Game was due to not handling null values. This caused many null pointer exceptions which were enough to break the game. Through structural testing, we were able to identify these bugs and add null handlers to ensure the consistency of the code among all input values. Additionally, a bug was found that caused 2 enemy entities to freeze if they collided. This bug was discovered through structural testing of the Enemy class, specifically the nonPlayerEntitityCollision() method. To fix this bug, a priority value was used to

resolve the conflict between the entities which made sure the enemy with the higher priority value moved first to avoid collision conflicts.