Unit Test

Our group conducted thorough unit testing. The identified features that needed to be unit tested were the packages: UI, GameObjects, Mechanics and Application tests. Our approach was to ensure that all of these features would act as expected and to achieve this we strived for the highest possible line and branch coverage. GameObjects contained packages for actors, environment, and interactables. Mechanics contained controller, game and map. UI contained all of the in-game menus and playerhud. We used mock classes to isolate classes that interacted with others. This practice was prevalent in testing the controller and menu testing. For example the controller test class used a mock application in its unit testing phase. In the case of menus, a mock controller was made to ensure that the menus would correctly update the game state.

Integration Tests

In addition to unit tests, we identified key interactions between UI and controller. For example, the UI has buttons that start/resume the game or exit to menu or the application. The controller has to take these changes in states and handle them accordingly. To test if each of these functions worked and the controller correctly handled them our group implemented tests like: testReturnButtonTriggersMenuAction() in the GameOverMenuTest. This test verifies that the return to main menu button in the final screen updates the controller correctly. A similar test case was made for testStartButtonTriggersGameAction() and testExitButtonTriggersExitAction() in the MainMenu tests. These cases were made to ensure that when the start button or exit button was pressed, the controller would update correctly to either start or exit the game. Similar tests were implemented for the PauseMenu. In addition, PlayerHUD testing included the integrated test: testPauseButtonTriggersPauseActionAndStopsTime() which ensured that the controller would correctly handle the pause button to stop the timer from running. This test involved interactions between UI, Time and Controller.

Another key interaction is between the controller and the application. The controller updates the application based on inputs received by the player. Test cases in the ControllerTests class cover player movement and ensure that user input updates the application state. The GameTest class also covers key interactions between the Game, Player, Enemies, Map, Score, and Stage systems. This test class validates that the Player moving updates the position, map state and enemy positions. In addition, the MapTest class tests interactions between Player, Tile types (Wall, StartTile, ExitTile), and Interactables. The tests in this class verify the proper setup and behaviour of map tiles in coordination with game entities, such as placing a Player and ensuring ground tiles are available and valid.

In EnemyTest, we tested scenarios like testMovementTowardsPlayer, where the enemy tracks the player's position and moves one tile closer, and testLineOfSight, where walls can obstruct enemy visibility. These tests validate how the Enemy, Player, and Cell systems interact to simulate intelligent movement and spatial logic. Another key set of integration tests comes from InteractablesTest, where we verified the effects of in-game items like traps and power-ups. For example, testBearTrapInteraction simulates how a BearTrap affects a Player by trapping them and decreasing their page count, which reflects changes to both player state and scoring logic. Similarly, testPageInteraction checks that collecting a Page properly increases the player's score. These interactions confirm that environmental objects correctly trigger and apply gameplay effects through shared systems like Player, Game, and Interactable.

Coverage

We used JaCoCo to evaluate both line and branch coverage of our codebase. Our project achieved an overall 85% line coverage and 65% branch coverage, reflecting strong test completeness across most gameplay logic and UI components. Coverage was particularly high in the packages: GameObjects, Interactables, Actors, UI, and Utility. For example, the UI package had 99% line and 100% branch coverage. The deficiencies in this package occur from the Menu class, which contains 2 methods meant to be overwritten by its subclasses. Therefore, these methods were never directly tested. While GameObjects had 100% line coverage and over 90% branch coverage. The GameObject class has 90% branch coverage primarily due to untested edge cases in sprite frame handling, such as invalid frame indices or null sprite arrays. These scenarios are unlikely during normal gameplay and would require artificial conditions to trigger, so they were deprioritized during testing. However, the CMPT276.Group12 and Mechanics packages showed lower branch coverage, largely due to complex control flow and edge cases in-game coordination logic, input handling, or error states that are difficult to reach without highly specific test setups. Some lower-coverage areas include rendering and animation paths that are inherently harder to test in headless environments. Overall, our focus on meaningful assertions and integration points has resulted in a robust and maintainable test suite. We have noted what we need to refactor for assignment 4, and after refactoring, branch coverage will increase.

Quality

To ensure the quality of our test suite, we focused on writing clear, focused, and meaningful tests that target specific features or behaviours within the game. Each test was designed to include strong assertions that verify not just the occurrence of actions, but their effects on the game state, such as position updates, UI transitions, and state changes like trapping or dying. We followed consistent naming conventions for our test classes and methods to improve readability and traceability. We also wrote integration tests wherever we thought it was necessary. Edge cases were included where relevant, such as attempting to move while trapped, interacting with tiles at map boundaries, or invoking pause/resume behaviour mid-game. Additionally, we ensured that tests were isolated and repeatable by resetting the game state before each test using @BeforeEach. This eliminated flakiness due to the shared state. Overall, we prioritized correctness, maintainability, and completeness to deliver a reliable test suite that supports both development and debugging.

Findings

We have learned that writing and running tests can be challenging. Especially when dealing with highly coupled classes on a complex project like this. We discovered that edge cases were often hard to test. We also discovered that the complex decision logic in UI rendering, exception handling, and animation timing, are inherently harder to test due to their visual and time-based nature. A few changes that we made were removing null checks in conditionals that could never occur. We also refactored Game so it was changed to Application, and Map was split into new classes Game & Map. The tests helped us discover bugs that would occur like, barriers blocking the exit, enemies spawning on top of the player, and refactoring the timer out of PlayerHUD and into its own Time class.