PHASE 3 REPORT

For our overall initial approach to creating tests for our game, we decided to split our testing based on what parts we did not contribute to as much during Phase 2. This was in hopes of reducing bias and preventing the recreation of the same mistakes. Everyone responsibilities were as follows:

Claire: StaticEntity Testing
Evan: Enemy Testing
Tim: Player Testing
Moussa: Gamestate Testing

To help calculate code coverage and identify untested statements we used the library Java Code Coverage (JaCoCo) and the VSCode extension Coverage Gutters.

Enemy Testing

In our testing process for the enemy functionality, we aimed to ensure that the enemy could effectively move towards the player in all directions (up, down, left, and right), navigate around obstacles to reach the player, and handle collisions with the player.

Our unit tests specifically focused on verifying the collision behaviour between the enemy and player when both objects were positioned at the same location. This test helped us validate the fundamental interaction between these two components.

To test these features comprehensively, we set up a test environment that included initializing various game objects such as Game, Playing, TileManager, CollisionChecker, Score, Enemy, and Player. This step was crucial for simulating the actual game environment and testing the interactions between different components.

For integration testing, we divided the movement tests for the enemy into individual tests for each direction. This approach allowed us to thoroughly evaluate the enemy's movement behaviour in different scenarios, including navigating around obstacles.

Using the Java Code Coverage library, we measured our test coverage. Our tests for the Enemy class achieved 88% statement coverage and 76% branch coverage, indicating a high level of code coverage for the enemy functionality.

Despite our comprehensive testing approach, there were some code segments in the Enemy class that remained uncovered. These segments were considered low priority or involved complex interactions that required specific conditions to execute, making them less critical for testing.

Through our testing efforts, we identified and fixed bugs in the enemy's movement behaviour, particularly in cases where the enemy was getting stuck against walls. Additionally, the Java

Code Coverage library helped us identify and remove unnecessary lines of code in the enemy class, further improving our code quality and coverage percentages.

Reward Testing and Trap Testing
In our testing for rewards and traps, originally we mostly focussed on the functionality. This meant ensuring the score incremented and decremented properly based on the collisions between Player and entity.

Our bonus and regular rewards could be tested with the same tests, as they use the exact same collision method, just with different reward values and sprites. All the tests were set up using JUnits BeforeAll feature, and initializing game objects such as Game, Playing, TileManager, CollisionChecker, and Score. Our unit tests were as follows:

testCheckPlayerRewardCollision: this test checks to see if the collision between the Player and Reward is true by initializing a player and a reward with the same positions on the map. The test also checks if the Score is incremented correctly.

testCollectMultipleRewards: this test checks the collision between the Player and 3 Rewards and ensures it is true. It also checks if the score is accurately incremented as well.

To check the despawn aspect for the Bonus rewards, as well as its update method we wrote the following tests:

testBonusReward: this test waits until the Bonus reward despawns and then moves the Player to the position of the despawned reward. It checks to ensure that the score did not increase.

testBonusRewardCollide: this test checks to see if the collision between the Player and Bonus reward is true by initializing a player and the reward with the same positions on the map.

After running these tests, we got a line coverage of 84% and a branch coverage of 83%. The lines that were not covered turned out to be exceptions when the reward sprites could not be loaded or unused methods. With this information, we refactored our Reward class to remove the methods setRewardAmount and getRewardsToCollect.

For trap testing, similar to the reward testing, all the tests were set up using JUnits BeforeAll feature, and initializing game objects such as Game, Playing, TileManager, CollisionChecker, and Score. Our unit tests were as follows:

testDamage: this test checks to see if the collision between the Player and Trap is true by initializing a Player and a Trap with the same positions on the map. The test also checks if the Score is decremented correctly.

testMultipleTrapPlayerCollisions: checks to see the collision of multiple traps and Player. Also checks if the score decrements properly.

After running these tests, we got a line coverage of 90% and a branch coverage of 100%. Similar to the reward tests, the only lines that were missed were the exceptions that are thrown when the image sprites are not loaded. We did not have to adjust any of our trap code as the tests worked the way we intended it to.

Our group struggled with getting the testMultipleTrapPlayerCollisions to pass when we ran JaCoCo, however the test passed when the file was run individually. We figured out this was due to updating the Player and Playing class in the test case which was unnecessary.

Door Testing
In our testing for doors, our goal was to check if the Player met the open door conditions on collision. The door's open condition is met once the Player collects all three grad caps.

To make the end goal clearer to the user, we added a new open door animation, where when the Player collects all 3 grad caps the door "opens". To achieve this, the Door class was adjusted to contain a boolean attribute which gets updated to true when the Player's win condition is also met. The Door's render condition also contains 2 different sprites, one for the closed door and one for the open door.

For door testing, similar to the reward and trap testing, all the tests were set up using JUnits BeforeAll feature, and initializing game objects such as Game, Playing, TileManager, and CollisionChecker. Our unit tests were as follows:

testDoorNoGrapCaps: this test checks to see if the collision between the Player and Door is true by initializing a player and the door with the same positions on the map. It also checks to make sure the Door open condition stays false.

testDoorOneGrapCap: this test checks to see if the collision between the Player and Reward is true by initializing a player and the reward with the same positions on the map. It then checks to see if the collision between the Player and Door is true by moving the player and the door to have the same positions on the map. It also checks to make sure the Door open condition stays false.

testDoorTwoGrapCaps: this test checks to see if the collision between the Player and two Rewards is true by initializing a player and the rewards with the same positions on the map. It then checks to see if the collision between the Player and Door is true by moving the player and the door to have the same positions on the map. It also checks to make sure the Door open condition stays false.

testDoorThreeGrapCaps: this test checks to see if the collision between the Player and three Rewards is true by initializing a player and the rewards with the same positions on the map. It then checks to see if the collision between the Player and Door is true by moving the player and the door to have the same positions on the map. It also checks if the Door open condition turns true.

After implementing these tests, we were left with 90% line coverage and 100% branch coverage. Similar to the reward and trap tests, the only lines that were missed were the exceptions that are thrown when the image sprites are not loaded. Since our test cases worked as intended for the Door class, the only changes we made was the door opening as stated above.

Player Testing
In our test suite for the player entity, we focused on checking the player's movement in all four directions. We specifically checked regular movement, movement into walls, and invalid movement, such as when simultaneously pressing multiple direction keys.

For our unit tests: we checked normal movement in all four directions. For example, we simulated the player going in a particular direction and checked if the player's position was updated correctly. To improve our branch coverage and test quality, we tested invalid movement by systematically setting different combinations of the direction boolean values to true. This enabled us to verify how pressing multiple movement keys won't move the character.

For integration testing: we checked if moving towards a wall correctly blocks movement. We repeated this test to check wall collision in every direction and verify that the behaviour was consistent. The notable interactions in this integration test were between the Player entity, the map, and the collisionChecker.

These thorough testing strategies yielded 88% line coverage and 93% branch coverage, which suggests the Player class was comprehensively tested. The lines and branches that were missed by the test cases were related to the animations and display of the character, which were not considered a priority that needed testing.

Findings: during our testing, we realized that one of the if statements in the updatePos() method was supposed to be an else if statement. Although this error didn't cause any errors, it still resulted in the program needlessly checking the conditions. Furthermore, we also initially tested the increment and decrement score methods from the Player class and realized that calling it did not impact the actual score object. This was because the score object we were modifying was not the exact score object that the Playing class uses. After this, we removed the methods as they were never used and faulty.

Gamestate Testing

In our Gamestate testing endeavor, our main objective was to ensure seamless transitions between different game states in response to player actions and game events. We examined how the game progresses through its various states, including MENU, PLAYING, GAMEOVER, and WIN, ensuring that it reacts appropriately to player inputs and in-game circumstances.

Our unit tests for Gamestate were carefully crafted to set up the game environment comprehensively. This involved initializing crucial objects such as Game, Playing, Menu, CollisionChecker, and other necessary components. Each test method was tailored to target a specific game state transition or behavior, including collision functionalities.

testMenuState: This test meticulously checks whether the game state seamlessly transitions to the MENU state and guarantees that the Menu object is correctly initialized and accessible.

testPlayingState: This test focuses on validating the transition to the PLAYING state and ensures that the game remains in the PLAYING state after initiating gameplay.

testGameOverState: This test scrutinizes whether the game transitions to the GAMEOVER state under specific conditions, such as player failure, and verifies that the state persists accurately.

testWinState: This test confirms the transition to the WIN state when the player achieves victory conditions and validates that the state remains in WIN after meeting the win condition.

testGameOverCollisionState: This test evaluates the collision between the player and traps, verifying that the game transitions to the GAMEOVER state when appropriate collision conditions are met.

testWinCollisionState: This test assesses the collision between the player and rewards, ensuring that the game transitions to the WIN state when the victory conditions are satisfied.

During our testing endeavors, we encountered challenges in achieving a high branch coverage, primarily due to the automation of key inputs rather than manual input. This automation resulted in certain branches in the code not being exercised fully. However, despite this limitation, our tests effectively covered essential functionalities and state transitions within the GameState class.

Through rigorous testing practices, we successfully identified and addressed issues related to incorrect state transitions and inconsistencies in game behavior. Overall, our Gamestate testing significantly contributed to enhancing the reliability and stability of the game's state management system. It's worth noting that our test suite achieved a code **coverage** of **72%** for lines and **40%** for **branches**, acknowledging the room for improvement in branch coverage due to the automation nature of key inputs.