CMPT 276 - GROUP 9 - PHASE 3 - TESTING REPORT

I. Unit and Integration Tests:

a) Main package

1. CollisionChecker class:

Approach: The CollisionCheckerTest class assesses the game's collision detection mechanics. Testing strategy consists of setting up mock game environments, mock of GamePanel, Prisoner, Objects, and simulating movements and positions of characters and objects to test different collision scenarios. By adjusting the characters' positions and directions, the tests can verify the collision detection of the game. Three main parts of the tests includes:

- **Object Collisions:** Tests verify that the game accurately detects when the prisoner collides with objects, considering all directions. These tests help ensure that interactions with game objects run smoothly without any bugs.
- **Characters Collisions:** These tests check for collisions between character classes, including the prisoner and enemies(the ghosts in this case). The tests validate that the game correctly identifies collisions between different characters (ghost and prisoners), from different movement directions.
- **Prisoner Collisions:** These tests simulate scenarios where enemies approach the prisoner from different directions, validate that the game correctly identifies these collisions, and verifies gameplay events like health reduction or item pick up.

2. EventChecker class:

Approach: The EventCheckerTest class evaluates event handling functions of the game, specifically on when the prisoner touches with game tiles that have special effects like lava, water, and exit. Through simulated scenarios within a mock game environment, it tested the game's state changes in response to the prisoner encountering these tiles:

- **Lava Pit:** In the game, lava is a punishment, you get hp deduction of 1 whenever you touch a lava tile (walked in). These tests created scenarios where the player character touches lava tiles, assessing if health points decrease appropriately, reflecting the punishment impact.
- **Water Pit:** Water pits are the opposite of lava pits, they are the bonus rewards of the game. You get hp increment of 1 whenever you walk into a water tile, however, it will only heal up to half your hp count. Tests in this category simulate the prisoner stepping on water tiles, checking if the game correctly increases prisoner’s health points.
- **Exit Tile:** Exit tiles, or the exit door to finish the game, located at the top of the map. If you collected all the skulls and chest, you will be able to finish the game by touching the exit tile, if you haven’t collected all of them, these tiles won’t have any effects. These tests validate the game's ability to recognize when the player reaches an exit tile, and reacts based on whether or not items are fully collected.

3. GamePanel class:

Approach: The GamePanelTest class evaluates the functionality of the GamePanel class in managing the game's state, rendering, updating game components, and handling user input. The testing strategy involves setting up mock game environments, including mock objects such as Prisoner, Enemy, TileManager, and UI, to simulate various game scenarios and interactions. The tests focus on different aspects of the GamePanel class:

- **Setup Game:** This test verifies that the setupGame() method initialises the game state correctly, including the creation of game objects such as the prisoner, enemies, and objects at their respective positions.
- **Retry Game:** Tests the retry() method to ensure that the game state is reset properly when the player retry after a game over, including resetting player attributes and regenerating game objects.
- **Run Game Loop:** Evaluates the main game loop functionality by running the game thread and ensuring that the game updates and renders components at the specified frame rate.
- **Start Game Thread:** Verifies that the startGameThread() method initialises and starts the game thread correctly.
- **Update Play State:** Tests the update() method to ensure that game components are updated correctly during the play state, including player and enemy movements.
- **Update Pause State:** Validates that the update() method behaves appropriately during the pause state, ensuring that no updates occur when the game is paused.
- **Paint Component:** Verifies that the paintComponent() method correctly renders game components, including tiles, objects, characters, and UI elements, on the game panel.

4. KeyChecker class:

Approach: The KeyCheckerTest test class examines the game's response to keyboard direction keys WASD input by simulating key presses and releases, along with P for pause/play feature and ENTER for retry feature. It uses a mock GamePanel to observe how the game's state or character movement changes based on keyboard input. The tests cover directional controls (up, down, left, right), and pause/play, replay controls, ensuring the game accurately runs accordingly to these inputs for movement and gameplay control. We did 3 areas of testing:

- **Direction keys:** Tested if the game recognizes when keys for moving up, down, left, or right are pressed and released. This ensures the character moves as expected and stops moving when the keys are released, essential for navigating in the map.
- **Game State Toggle:** This part tests the game's ability to pause and resume by pressing the key P, by simulating the key pressed and checking if the game state changes.
- **Game restart:** This part tests the game's ability to retry a game by pressing the key ENTER (only works if when the game’s over), by simulating the key pressed and checking if the game state changes from either gameFinished or gameOver to play.

5. UI class:

Approach: The UITest class evaluates the functionality of the UI class, which is responsible for rendering the game's user interface. The testing strategy involves verifying that the UI elements are drawn correctly under various game states, such as playing, paused, game over, and game finished. Additionally, the initialization of fonts and images in the UI constructor is tested to ensure proper setup. The testing strategy encompasses two main aspects: constructor initialization and drawing functionality under different game states.

- **UI Constructor Initialization Test:** This test verifies that the UI constructor initialises the necessary attributes, including fonts and images, correctly. It ensures that the UI object is properly configured upon instantiation.
- **Drawing Functionality Test:** This test evaluates the drawing functionality of the UI class under different game states. It simulates the rendering of UI elements, such as player health, score, game objects, and text messages, ensuring that they are displayed accurately on the screen.

- Playing State: Verifies that the UI draws the player's health, score, game objects (e.g., chests and skulls), and time are displayed correctly during gameplay.
- Paused State: Checks that the UI renders the pause screen with the "PAUSED" message centred on the screen.
- Game Over State: Ensures that the UI displays the game over screen, including the "Game Over" message and the option to retry the game.
- Game Finished State: Validates that the UI renders the game finished screen with the appropriate message, playtime, final score, and option to play again.

6. Utility tool class:

Approach: The UtilityToolTest class evaluates the functionality of the UtilityTool class, specifically focusing on its ability to scale images accurately under various conditions. The testing strategy involves creating test images with different sizes and validating the output of the scaleImage() method against expected results. The testing strategy consists of setting up test cases that cover different scenarios related to image scaling. Each test case assesses the behaviour of the scaleImage() method under specific conditions, ensuring that it produces the expected output. Some of the tests cases are explained below:

- **Basic Functionality Test:** This test case validates the core functionality of the scaleImage() method by providing a test image and verifying that the scaled image has the correct dimensions and is not null..
- **Negative Input Sizes Test:** This test case evaluates the behaviour of the scaleImage() method when provided with negative input sizes, ensuring that it returns null as expected.
- **Null Input Image Test**: This test case checks how the scaleImage() method handles a null input image, verifying that it returns null as expected.
- **Performance Test:** This test case measures the performance of the scaleImage() method for large input images, ensuring that the scaling operation completes within a specified time limit.
- **Zero Width and Height Test:** This test case evaluates how the scaleImage() method handles zero width and height input sizes, ensuring that it returns null as expected.

b) Character package

Approach: The CharacterTest class evaluates the functionality of the game's character behaviours and interactions. The testing strategy involves setting up mock game environments, including mock instances of GamePanel, Prisoner, and Enemy, to simulate various game scenarios. The tests focus on different aspects of character behaviour, such as movement, interaction with objects, collision detection, and enemy AI. Key Aspects Tested:

- **Movement:** Tests verify that the player character responds correctly to keyboard inputs, changing direction and moving accordingly. Assertions are made to ensure that the character moves in the correct direction based on user input.
- **Interactions:** Tests assess the interactions between the player character and various game objects. This includes picking up items such as boots, hearts, chests, and skulls, and updating the player's score and health accordingly.
- **Enemy Behaviour:** Tests evaluate the behaviour of enemy characters, including their movement patterns and interactions with the player character. Assertions are made to check if enemies pursue the player when in proximity and wander randomly when not in pursuit.

- **Collision Detection:** Tests verify that collision detection between characters and objects functions as expected. This includes checking if collisions are detected accurately in different directions and scenarios, such as moving into objects or colliding with enemies.

c) EnemyAI package

Approach: The EnemyAITest class assesses the functionality of the enemy artificial intelligence (AI) responsible for controlling enemy behaviour and pathfinding. The testing strategy involves setting up mock game environments, including mocks of the GamePanel, Node, and other relevant classes, to simulate different scenarios and test various aspects of the enemy AI.

- **Open Node Testing:** This test verifies that the openNode() method correctly marks a node as open when it meets certain criteria. It ensures that nodes are appropriately marked based on their properties.
- **Set Nodes Testing:** The setNodes() method is tested to ensure that it correctly sets the start and goal nodes for pathfinding. This involves setting up a mock grid of nodes and validating that the start and goal nodes are assigned correctly.
- **Get Cost Testing:** This test evaluates the getCost() method to verify that it accurately calculates the cost values (G, H, and F) for a given node. It checks if the cost values are calculated correctly based on the node's position and the start and goal nodes.
- **Track the Path Testing:** The trackThePath() method is tested to confirm that it correctly traces the path from the goal node back to the start node and stores it in the path list. It ensures that the path is accurately recorded for future reference.
- **Search Failure Testing:** This test examines the search() method to validate its behaviour when a path to the goal node cannot be found. It simulates a scenario where no path exists and verifies that the method returns false as expected.

d) Object package

Approach: The ObjectTest class assesses the functionality of the Object superclass, which represents in-game objects containing images, names, collision types, and positions on the game map. The testing strategy involves setting up mock game environments, including mocks of the GamePanel, Graphics2D, and other relevant classes, to simulate different scenarios and test various aspects of the Object class.

- **Draw Testing**:This test verifies that the draw() method correctly draws the object on the screen. It ensures that the object's image is drawn within the specified boundaries of the game panel and relative to the position of the player character (prisoner).
- **Draw With Prisoner Within Bounds Testing:** This test evaluates the draw() method when the prisoner character is within the bounds of the game panel. It checks if the object's screen coordinates are calculated correctly based on the prisoner's position and the game panel's monitor dimensions.
- **Draw With Prisoner Outside Bounds Testing:** This test examines the draw() method when the prisoner character is outside the bounds of the game panel. It verifies that the object's screen coordinates are calculated correctly even when the prisoner's position exceeds the game panel's boundaries.
- **Draw Partially Within Bounds Horizontally Testing:** In this test, the scenario is set up where the object is partially within the bounds horizontally. It ensures that the object is not drawn when it intersects with the game panel's boundaries horizontally.

- **Draw Partially Within Bounds Vertically Testing:** Similarly, this test sets up a scenario where the object is partially within the bounds vertically. It validates that the object is not drawn when it intersects with the game panel's boundaries vertically.

e) Tile package

1. Tile class:

Since tile is just a simple object, with a simple constructor, we tested on when a new Tile object is created, assert if the default behaviour is correct, which is the image is null and Collision boolean is false.

2. TileManager class:

Approach: The TileManagerTest class focuses on testing the functionality related to tile management within the game. It specifically examines how tiles are initialised, loaded, and rendered. Mocking is used to simulate the GamePanel and Graphics2D objects, allowing the tests to assess the TileManager's interactions with these components without needing to run the entire game environment. We tested on these 3 areas:

- **Initialization and Setup:** Verifies the correct initialization of the tile array, ensuring tiles are set up with their respective images and collision properties.
- **Map Loading:** Tests the loading of map data, ensuring that the tile numbers and types are correctly assigned based on the map file.
- **Tile Rendering:** Checks that tiles within the player's view are rendered correctly, simulating different prisoner positions and verifying that the rendering process considers the player's location.

II. Test quality and coverage

In maintaining the quality of our test cases, we prioritised practices such as using mocks and stubs to isolate dependencies, adhering to the Arrange-Act-Assert pattern for clear test structure, and ensuring each test case focuses on testing a single behaviour. Moreover, we emphasised the importance of meaningful assertions to verify expected outcomes and regularly reviewed and refactored test code for readability and clarity. As for coverage analysis, the overall project achieved a commendable 96.4% line coverage and 78% branch coverage. While specific classes like ObjectTest attained 100% line and branch coverage, GamePanelTest, with its 95% line coverage and 78% branch coverage, contributed significantly to the project's overall coverage. Despite not covering certain error handling and thread management branches due to their complexity to simulate in unit tests, the tests collectively provided comprehensive coverage of the project's functionality.

III. Findings and conclusion

In conclusion, after over 100 test cases implemented throughout the test classes, our game passes all the cases, with no apparent functional or structural errors. However, throughout the process of writing the tests, it reveals that our original code in classes and components were not having high testability, making some methods, objects, or functionality of the game much more difficult to assess and write test cases. Hence, while writing the test code, we also continuously refactors the code smells, or anti-patterns, found in the program, to increase testability and make them more open for modifications. In terms of test coverage and quality, the line coverage was achieved to the highest possible, 96.4%, and the branch coverage was only moderately high, 78%. The issue that prevented the coverage from reaching higher percentages is due to lines inside exception cases, which we were

not able to simulate or trigger the exceptions in the test cases, one of the areas that we will need to improve. Apart from that, overall, the program is bug-free, and the results of the test cases reflected that successfully.