CMPT276 Phase 3 Report: Group 4

I. Unit and Integration Testing

Unit Testing:

1. Test Player changing position, image switching and add and score
2. Test Player movement for return target position available and return reward player collect
3. Test RecordUsedPlace storing available position adding, randomly return the available position t and the position is not blocking the enter space of some are
4. Test Room environment storing reward and enemy and result relative function about enemy and reward
5. Test Room layout the storing wall, obstacle, tile and count number of each item
6. Test InitialiseGameItem the generate different object that needed in the game
7. Test KeyboardListener will return the direction to player when use pressing moving key and ignore or key
8. Test the LoadingPanel ability to check all object is correctly loading and will not aspect the invalid number of object
9. Test MainPanel change after the start button is pressed
10. Test PanelController will switch between different panels and give out the information needed for these panel
11. Test Reward generates reward, rewards appear and disappear and draw function.
12. Test Reward initialization generates the right number of candy and pumpkinHead.
13. Ghost catches player test verifies that when a ghost’s position coincides with the player’s position, the game manager’s method for handling player capture by an enemy is called correctly.
14. Spider constructor test checks whether the Spider constructor correctly instantiates a spider object and places it at a random available position, ensuring the correct type of spider is created based on the number of enemies.
15. Enemy movement towardsplayer tests check the ghost class's behaviour when the player is nearby. It verifies that the ghost moves towards the player's position when within a certain range.
16. Enemy random movement test ensures that when the player is not nearby, the ghost moves to a random new position, testing the ghost's ability to roam freely in the absence of the player.
17. Player exit and show score test validates that upon a player exiting through a door after collecting all rewards, the game correctly transitions to the end screen and shows the player's score.
18. Player caught by ghost test confirms that if a player is caught by a ghost, the game acknowledges the game end condition and transitions to the end screen, appropriate for a game-over scenario.

19. Turn on door test checks to see if the door is open after the game manager indicates that the player has collected all rewards. There is another unit test that does the same thing but makes sure the door is closed when the player doesn't have all the rewards.
20. Initialize room test checks if the initializeRoom method sets up wall and tile positions correctly.
21. Iroom Test checks to see if the method properly returns a room instance by the mock RoomFactory
22. ITombs, IWalls, and ITiles all test if the create(Tomb/Wall/Tile) is called correctly with specified positions and counts.
23. TestSetDoorPosition and TestSetObstacle test to see if the methods set the object to the correct position after being called.
24. TestGetDoorPoisition and TestGetObstacle test to see if the methods return the correct position after being called

Components for Integration Testing:

Below are the main integration tests we conducted.

1. Player Movement:
    a. Moving Player test is combine the keyboardListener, Player and PlayerMovement class to work on the key input and player moving
    b. Using the pressing or release keyboard to control the player moving position and observer its behavior when the key release, meet available position and other situation
    c.

```
/**
 * Test player will change position fellow by keypress, if target position is
 * available
 */
@Test
public void pessKeyMovePlayer(){
    Position[] positions = createAvailablePositions(number:1,-WindowConfig.tileSize,y_change:0);

    movePlayer(MovingKey.A, press:true, time:1);
    assertEquals(positions[0], player.getPosition());
}
```

```
/**
 * Test player will not change position fellow by key press, if target position
 * is not available
 */
@Test
public void pressKeyMovePlayerFail(){
    Position playInitPosition = player.getPosition();
    Position unaviablpos = new Position(WindowConfig.tileSize, WindowConfig.tileSize * 2);
    when(mockLayout.isPositionAviable(unaviablpos)).thenReturn(value:false);

    movePlayer(MovingKey.S, press:true, time:1);
    assertEquals(playInitPosition, player.getPosition());
}
```

2. Enemy's movement: Checks whether the position calculation of moving enemy's is calculated corrected whether player is around or not.

3. Ghost catch player: Check if when ghost and player position overlap, game manager should have appropriate action.
4. Player Step on Enemy: Check when score is deducted when player and fixed enemy's position overlap.
5. Showing scores when player exit: Check if the number panel is set up and called when game ends.
6. Showing scores when game is over or caught by enemy: Checks whether the number panel is being called and correct image is selected for the panel.
7. Reward Generation test at every difficulty level
8. Enemy generation combine with difficult level for testing: Checking if number of each enemy is correct based on game difficulty.

II. Test Quality and Coverage

Measures taken for ensuring test quality

Mocking and Dependency Isolation:
We extensively used Mockito to mock dependencies, ensuring that our unit tests were isolated and focused only on the component under test. This was particularly useful in testing classes like EnemyInitialization, where external dependencies like EnemyFactory were mocked to focus solely on the initialization logic.

Continuous Integration and Regular Testing:
We integrated a continuous integration system to run our tests automatically with each commit. This helped in catching bugs early and ensuring that new changes didn't break existing functionality. After each refactoring, we ran all tests again to ensure behavior not changed.

Refactoring Tests:
Similar to production code, we refactored our tests for clarity and maintainability. This involved renaming tests for better readability, breaking down complex tests into smaller units, breaking dowb larger classes, pull out similar functions, and removing redundant functions.

Line and Branch Coverage of Each Package

Package	Line Coverage (%)	Branch Coverage (%)
cmpt.group4	98	90
cmpt.group4.Enemy	97	92
cmpt.group4.GameMap	98	90
cmpt.group4.Logic	99	100

cmpt.group4.Player	96	93
cmpt.group4.Reward	83	50
cmpt.group4.Room	81	56
cmpt.group4.Time	100	100
cmpt.group4.UI	97	96
cmpt.group4.WindowAndInpupt	93	84

Reasons for uncovered feature and code segments

Testing graphical components like drawing methods (draw(Graphics2D g2)) in the Enemy classes proved challenging. Automated tests for graphical output are complex and often require visual verification, which falls outside the scope of typical unit testing frameworks.
Certain exception handling blocks, especially those catching generic exceptions (e.g., catch (IOException e)) in getEnemyImage() method in Spider class, were difficult to trigger. Simulating these exceptions often required contrived scenarios that were not realistic for the game's operation.

Certain classes had complex logical conditions that were hard to satisfy in a test environment. For instance, the canPlaceEnemyAndObstacle() method in RecordUsedPlace class involved multiple conditional statements and randomness that required specific setups, making it challenging to achieve full coverage.

The singleton pattern used in RoomLayout makes it difficult to test RoomFactory due to its global state. The shared instance makes it hard to isolate dependencies. Therefore couldn't cover RoomFactory.

III. Analyzing Findings

What we learned from writing the test

We realized that excessive use of singletons can lead to difficulties in testing, as they often carry state across different parts of the application. This can lead to unexpected behaviors during tests. We learned the importance of limiting singleton usage and exploring alternative design patterns that allow more testable and maintainable code.

We observed that large classes with multiple responsibilities were harder to test and maintain. This led us to focus on making our classes more cohesive with single responsibilities, thereby enhancing

testability and readability. High cohesion and low coupling became guiding principles in our design choices.

What we learned from running the test

A recurring issue we encountered was that the Position object was frequently null in our tests. We realized this was due to not properly initializing these objects or not storing parameters correctly in constructors. We incorporated Mockito in our code to ensure proper values and objects are returned.

Some tests succeeded in certain environments but failed in others. This inconsistency prompted us to examine our test environments and configurations more closely, leading to a better understanding of how different settings can impact test outcomes.

Bugs we found

We uncovered a bug in our score panel where negative scores were not being displayed correctly. This was an important finding as it directly impacts the user experience and the game's feedback system. Our testing revealed that obstacles could occasionally block critical paths for the player, leading to gameplay issues. This discovery was significant as it affected the game's balance and playability.

What we did to Improve Quality of the code

We undertook an extensive refactoring process, renaming variables and functions for clarity, and removing dead or redundant code. This made the codebase more understandable and maintainable. We refocused on high cohesion and low coupling in our classes and modules. This structural refactoring meant breaking down complex classes, redefining responsibilities, and ensuring that each module had a single, well-defined purpose.

What we changed and refactor in phase 3

Based on our findings from the tests, we adjusted some of our design patterns, moving away from heavy singleton usage to more flexible and testable designs. We overhauled the code structure to better align with the principles of clean code. This involved reorganizing classes, improving method structures, and ensuring that our code adhered to SOLID principles. We improved our testing framework, incorporating more rigorous unit and integration tests. This not only helped us find bugs but also gave us confidence in the robustness and reliability of our code.