CMPT 276 - D100
GROUP FOUR

Phase 3 Report

Group Four Members:

- Ana Premovic
- Regan Li
- Matin Keivanloo
- Zhaoyue Yuan

Topics Discussed:

1. Tests
    a. Unit Tests
    b. Integration Tests
2. Test Quality & Coverage
3. Findings
    a. What we Learned from Writing and Running Tests
    b. Changes made to the Production Code during the Testing Phase:
    c.

Tests

Unit Tests:

Feature	**Test**
Display menu	MenuTest testDisplayMenu()
Display winning menu	WinningMenuTest testDisplayWinningMenu()
Display game over menu	GameOverMenuTest testDisplayGameOverMenu()
Game starting after start game button is pressed	MenuLogicTest testStartGameButtonPressed(), testStartGame()
Exiting the game	MenuLogicTest testExitProgram()
Store tile data	TileManagerTest testGetBoard()
Store types of tiles on the screen	TileManagerTest getTileTypes()
Display all rewards and punishments to screen	ImmovableObjectsDisplayTest

	testDisplayObjects()
Generate random position for a new reward or punishment to be displayed to the screen	ImmovableObjectsDisplayTest testGenerateRandomPosition()
Allow enemy movement	EnemyTest testPassable()
Enemy movement	EnemyTest testRight(), testLeft(), testUp(), testDown()
Transpose the correct enemy matrix	EnemyTest testTransposeMatrix()
Stop main character movement on game restart	KeyHandlerTest testNomovement()

Integration Tests:

Interaction	**Test**
Interaction where enemy catches the player	EnemyCatchPlayerIntegrationTest testEnemiesCatchPlayer()
Find shortest path between player and enemy	PathFinderTest testShortestPath()
More complicated case of shortest path with a big maze	PathFinderTest testBigMaze()
Case where path length between player and enemy is 0	PathFinderTest testNoPath()
Player touching rewards/punishments	MainCharacterTest testTouchObject()
Player touching an ocean tile	MainCharacterTest testExitCave()
Player colliding with tiles	CollisionDetectorTest testDetectTile()
Player colliding with rewards and punishments	CollisionDetectorTest testDetectImmovableObject()
Store tile data (type of tiles) - interaction between file system and display	TileManagerTest testLoadBoardData()
Store tile images - interaction between file system and display	TileManagerTest testGetTileImages()

Test Quality & Coverage

In order to view our test code coverage, we used some features built into the IntelliJ IDE to our advantage. This feature enabled us to view line as well as branch coverage for each test class, independent and/or dependent from each other, while also displaying coverage for integration tests.

All of our classes pass all the tests that we have written, and we found that certain classes were more difficult to test than others, because they require most of the other components/classes to be running concurrently alongside them. So, some of our tests do not test methods in isolation, but rather with some of the required components. All tests correlated to the menus of our game yielded an average of 85% line coverage with an 87.5% branch coverage. The only class that did not achieve 100% branch coverage is the WinningMenu class with 50% (2/4) branch coverage.

A couple of classes that were difficult to fully test and achieve 100% line coverage were the MainCharacter and CollisionDetector classes. These two classes achieved 62% and 67% line coverage respectively.

In regards to branch coverage in general, we strove for tests that cover as many potential scenarios as possible. For instance, testing the KeyHandler class meant that we would assert that the key being pressed would produce the expected result in moving the main character a certain direction, and not an unexpected direction. Another instance of branch coverage testing is in CollisionDetector, where we would test the main character coming into contact with rewards, obstacles, and enemies, both static and dynamic.

Findings

What We Learned from Writing and Running Tests:

Since our group members did not have any prior experience in game development or larger scale Java projects in general, we were not very familiar with writing rigorous tests or working with JUnit in general, which is a reason that IntelliJ's testing coverage feature came in very handy. One of the most significant and applicable lessons that we learned during the testing phase of our project was that code that is all over the place and ambiguous can be detrimental as they lead to a multitude of issues/bad code smells that pose difficulties when writing competent, thorough tests.

Changes made to the Production Code during the Testing Phase:

Our production code from phase 2 often followed some poor design choices, resulting in bad code smells that led to troubles writing tests. For instance, in our MainCharacter class, whenever our character touched lava, points would be lost, which meant that more carrots needed to be generated in order to provide points for the player to escape. In order to generate additional carrots, we copy pasted code. So instead, we created a new method called generateRandomPosition() to achieve the generation of extra carrots, and invoked this method in both the ImmovableObjectDisplay and MainCharacter classes. Another common issue we had was many instances of unused/unnecessary variables, which affects coverage when testing as the method(s) are not used in the first place. For example, we made certain changes to our path finding algorithm that rendered the directionEntered() method redundant which led to its removal.

Revealing and Fixing Bugs and/or Improving Quality of Code:

In terms of revealing and fixing bugs, we ran into an issue where there would be multiple “Congratulations!” menus upon our main character exiting the cave. In order to fix this, we split up our displayMenu() method into multiple methods — winningMenu() displaying the winning menu, GameOverMenu() displaying the game over menu, and Menu() which displays the start menu. These separate methods are each solely responsible for displaying their own menus so that displayMenu() is not responsible for displaying all possible menus. Another bug that we ran into was with the restart button, where everytime our character restarted and spawned at the initial spawn location, both the enemies and the main character would be sped up, doubling each time restart was clicked. We ended up finding that there was an issue with multiple startGameThread methods of the same instance running upon respawn, so removing the additional thread fixed the issue at hand.