

Phase 3 Report

CMPT 276

Group 9

Features Unit Tested:

endingCell

- The ending cell should initialize as an invisible and non-interactive. Correctly displays and makes itself interactable upon being notified that all regular rewards have been collected, and that the game has not yet ended.

endScreen

- The end screen should initialize itself as invisible and remain that way until the outcome of the game has been determined. Correctly receives and stores the result of the game.

Enemy

- The enemy should initialize itself at the correct position and be visible throughout the entire game. Images should be updated according to the direction the enemy moves. Correctly determines if a visual overlap with the player has occurred.

gameMap

- The game map should set all of the positions of the regular rewards, stationary punishments, enemy, and player without overlapping. Correctly stores and updates the position of the player and enemy as well.

gameWindow

- The game window should initialize and store all of the classes properly for future use. The game loop is working correctly at the set frames per second. Timers for events such as random event spawning, and showing the ending screen are created.

handleInput

- The input handler contains four boolean values to check if a direction arrow key is pressed. Correctly updates the boolean values when a key is pressed or released.

Player

- The player should initialize the position, score, and the direction facing. Correctly updates its images and position according to the user input. Also, the class checks if the player score needs to change and whether the player is touching the ending cell. Lastly, it keeps track of the time spent in the round.

Punishment

- The Punishment should store the image of stationary punishment. It also returns the value of damage to the player.

PunishmentList: The punishmentList should parse the map from the gameMap and store punishments into an arraylist.

regularReward

- The regular reward should place regular rewards according to its parameter coordinates.

rewardManager

- The reward manager should parse the map from the gameMap and store all the regular rewards into an arraylist. Also, the class checks if a regular reward is collected by the player by comparing their coordinates and removes it from the map once collected.

specialReward

- The special reward should initialize itself as hidden and non-interactable. The special reward should disappear and remove itself from the screen after the player has touched it.

wallManger

- The wall manager should correctly store all of the image sprites to be used in the game. Additionally, it should set the barriers sprite's collision status to true while the background sprite's collision status should be false.

Important Interactions between Different Components of the System:

Special reward spawning:

- The special reward uses the game map to help locate a safe position to place itself through avoiding any punishments, regular rewards, players, and enemies on the map.

Player movement with user input:

- The game places the player at the starting position and loads the player image. Also, according to the user input, which changes the movement of the player, the system determines if the player is facing one of the following possible scenarios: collecting a regular reward or the special reward, and touching a stationary punishment. The player also uses the collision variable of wallManager to determine if there is a wall blocking the path and making sure the player can not pass through the wall.

End screen getting time spent in game and points earned:

- Upon the game ending, the end screen interacts with the player to store how much time was spent in the game and how many points were earned in order to display these metrics in the victory screen.

Enemy tracking player:

- The enemy uses a breadth first search with its own position in conjunction with the player position and barrier positions on the game map to help determine the shortest path to take to reach the player.

Features/Interactions Covered by Test Case/Class:

gameMap

- setSpecialReward(): Checks if the represented value of special reward is assigned to the desired position in the TheMap according to the given x and y coordinate.
- setPlayerPosition(): Checks if the represented value of player is assigned to the desired position in the TheMap and the EnemyPlayerMap according to the given x and y coordinate.
- setEnemyPosition(): Checks if the represented value of enemy is assigned to the desired position in the EnemyPlayerMap according to the given x and y coordinate.
- update(): Checks if the map coordinate values are successfully updated for the position of moving entities.

Punishment

- constructorTest(): Checks if the variables of the class are initialized successfully.
- getDamage(): Checks if the correct value of damage is returned.
- testDraw(): Checks if the image is successfully stored.

wallManager

- getWallImages(): Checks if the images of water and rock are successfully stored and the collision is set true for rocks.

Enemy

- constructorTest(): Checks the initialization of the moving enemy's starting position, eight images facing different directions, and initial facing direction.
- isTouchedPlayer(): A getter for the boolean value indicating whether the player has touched an enemy or not.
- update(): Manually position the player and the enemy next to each other to check if the boolean value gets toggled.
- BFS(): Manually input a coordinate for the player to see if the method correctly calculates the shortest distance between the enemy and the player.
- pathExists(): Manually places the player and enemy side by side to check if the correct path is taken by the enemy to reach the player in the shortest steps.

gameWindow

- startGameThread(): Checks to see if a thread is initiated.
- startGameStage(): Checks if the initial conditions (game ending variables set to false) are set.
- run(): Checks if the method successfully creates objects of other classes.

handleInput

- keyPressed(): Checks the initial values of the boolean variables to indicate if a key is pressed. Also, checks if the respective boolean is toggled when a key is pressed.
- keyReleased(): Checks if the respective boolean is toggled off when a key is pressed and then released.
- getterTest(): Checks if the getter returns the correct boolean value.

Player

- changeScore(): Checks if the method updates the player's score according to its parameter.

- `setDefaultValues()`: Checks if the initial values of the player are set correctly. The method is expected to position the player at a certain coordinate with its start score and facing direction.
- `getPlayerImage()`: Checks if the images of the player are properly loaded.
- `update()`: Checks if the player direction is updated when the user presses a direction key and the cell it is trying to move is empty. The test covers if the method is updating the player score when it is either on top of any rewards, or punishment. Also, it covers if the method correctly changes its position and `spriteCounter`, which is a variable for animation effect.
- `getScore()`: A getter method for the score.
- `touchingEndCell()`: Checks if the method toggles the boolean value when a game ending condition is met, which is when the player is touching the end cell. The test covers both when the player is or is not on the end cell.

regularReward

- `constructorTest()`: Checks if the class correctly initializes values and an image for regular rewards.

rewardManager

- `constructorTest()`: Checks if the class correctly stores the regular rewards into a list. Since the map contains six in total, the test checks if the list contains six elements as well.
- `claimReward()`: The test manually inputs a coordinate to see if the method returns the score of a regular reward when the player position matches one of the regular rewards' positions.

endingCell

- `testStart()`: Checks to make sure that the class does not immediately show itself when initialized.
- `testAppearance()`: Checks to make sure that the class shows itself when called
- `testDraw()`: Verifies that the image has loaded properly.
- `valuesTest()`: Checks to make sure the class is loaded in the correct position.

endScreen

- `firstTimeTest()`: Checks to make sure that the first time is always true in the beginning.
- `gameStateTest()`: Verifies that the game state is either a victory or defeat.
- `playerEndScreenTest()`: Checks for a positive time spent in the game which is received from the `Player` class.

punishmentListTest

- `testConstructor()`: Checks to make sure that the punishment list is populated with all of the punishments contained inside the map class.
- `checkImage()`: Checks to make sure the image exists for all of the punishments inside the punishment list.

specialRewardTest







































- `constructorTest()`: Checks for the initializations of the image, the score points, and the boolean value for spawning of the special reward.
- `initiateSpawn()`: `Random` is a class provided, so the test just checks for the condition inside the while loop and whether the method correctly places the special reward at the random coordinate generated by the loop.
- `removeReward()`: The test manually places the special reward at a certain coordinate and checks if it is removed.

Measures Taken to Improve the Quality of Test Cases:

Unlike phase 2 which required lots of collaboration to ensure the various parts of our program worked in unison, we unanimously agreed that writing test cases would require less teamwork.

Hence, we decided on an online meeting where we went through what features would be essential to cover in our tests, and divided up our tests equally. Additionally, we made use of the JaCoCo dependency while polishing up our test cases to help identify some of the lines and branches that we had previously missed during the development of our initial unit and integration tests.

Line and Branch Coverages:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
App.java		0%		n/a	2	2	4	4	2	2	1	1
mainWindow.java		0%		n/a	21	21	126	126	21	21	8	8
endScreen.java		29%		50%	4	11	17	34	1	7	0	1
wallManager.java		33%		0%	4	6	9	21	1	3	0	1
regularReward.java		45%		n/a	1	2	4	12	1	2	0	1
Player.java		48%		30%	46	64	65	128	2	10	0	1
Punishment.java		53%		n/a	1	4	4	15	1	4	0	1
endingCell.java		54%		n/a	1	5	4	16	1	5	0	1
gameWindow.java		59%		23%	21	31	57	122	4	12	0	4
reward.java		60%		n/a	2	3	2	3	2	3	0	1
PunishmentList.java		77%		75%	2	7	3	13	1	3	0	1
Enemy.java		78%		63%	27	56	33	121	1	7	0	2
specialReward.java		81%		75%	3	9	4	25	1	5	0	1
rewardManager.java		83%		85%	2	10	3	18	1	3	0	1
handleInput.java		98%		100%	1	16	1	26	1	8	0	1
wallImage.java		100%		n/a	0	1	0	2	0	1	0	1
Moving.java		100%		n/a	0	1	0	3	0	1	0	1
Entity.java		100%		n/a	0	4	0	5	0	4	0	1
gameMap.java		100%		93%	2	28	0	40	0	13	0	1
Total	1,900 of 4,889	61%	154 of 325	52%	140	281	336	734	41	114	9	30

Following the line and branch coverage report generated by JaCoCo, it is evident that there were a variety of different classes that were not fully covered. However, looking deeper into the content of the code helps demonstrate why this is the case.

Since App.java only contains a call to the mainWindow class, we found no reason in testing that file. Moreover, we also did not test any of the draw methods since it relies on the Graphics and Graphics2d classes that are bundled with the Java JDK. This helps account for quite a few of the missed line coverage which comes from classes such as but not limited to endScreen, regularReward, and wallManager. Also, it was impossible to test the startGameStage method of the gameWindow class because of the while loop. There was no way to escape out of it because its termination is controlled.

Another significant portion of lines and branches that were not covered were located inside the mainWindow class. The features inside this class included making the window, and creating buttons to start, restart and exit the game. The reason behind the decision to not compose any tests regarding these features were because all of the window and buttons themselves were all created through the graphical library Swing, which is prepackaged with the Java JDK. This bundling with the JDK made us confident in the robustness of the library such that we did not need to write any tests for portions of code utilizing this library. Additionally, the exit game button does call the quitGame method. However, since the quitGame method only calls System.exit(0) there was no point in testing as System.exit(0) terminates the JVM entirely.

Overall, disregarding the lines from the draw methods, we ended up with sufficient coverage for both our lines and branches tests to properly ensure a proper system which functions according to the specifications.

Findings:

Throughout this phase, we have learned about the challenges and difficulties that can arise when writing and running tests for programs. Specifically, one of the major challenges we encountered was regarding the user interface. We initially had all of our Swing components for our user interface declared and used in our main class. However, we soon found out that testing a main function written like that would be difficult and hence we rewrote it as a separate class. Moreover, since many of our classes worked in conjunction with one another, we soon found out how the existence of getters and setters would make our tests more simple through the ability to create mock scenarios.