PHASE 3 REPORT

CMPT276 Group 18

Group members: Haruka Mibuchi, Gurinder Bhogal, Asifiwe Julio Patrick

What was tested:

Moveable Entities Package:
Celebrity

- **update() (line coverage:100%, Branch coverage: 100%)**
  Checks the direction the player is heading and moves them in that direction if it is permissible. In order to test it, we checked each move such as left, right, up, and down and then checked if the position of the celebrity had changed correctly.

- **setDirection(Direction direction) (line coverage:100%, Branch coverage: 100%)**
  Overwrites the inputhandler with a given direction. Tested by checking the input handler's direction after the method call

Paparazzi

- **move(Direction direction) (line coverage:100%, Branch coverage: 100%)**
  Used to move the paparazzi in the given direction. In order to test it, since the enemy/paparazzi can move four directions (up, down, left, and right), we checked each case and checked if the position of the enemy changed correctly.

- **update(int x, int y) (line coverage:100%, Branch coverage: 100%)**
  Finds the next direction to move the paparazzi closer to the celebrity. In order to test this method, we checked each scenario and checked if the direction that the paparazzi will head to was changed depending on where the celebrity/the main character is.

- **setDirection(Direction direction) (line coverage:100%, Branch coverage: 100%)**
  sets the direction of the paparazzi. Tested to see if the method was inherited correctly and made the appropriate changes.

MoveableEntites

- **getPosition()**
- **getPositionX()**
- **getPositionY()**

Return the position of an entity by accessing the Point class. Tested with a concrete implementation of the Celebrity class.

Map Package:
Map

- **LoadBonusRewards(int num) (line coverage:100%, Branch coverage: 100%)**
  loads the specified reward number. Tested to check if the right value was passed in the mapArray

- **checkCollidable(cell type)(line coverage:100%, Branch coverage: 100%)**
  checks if the cell type given is collidable or not. Tested to check if cell is correctly identified as collideable

Cell

- **getCollidable()(line coverage:100%, Branch coverage: 100%)**
  returns a boolean representing whether the cell is collidable or not. Tested to check if the cell is correctly identified as collideable

- **setCollidable(boolean)(line coverage:100%, Branch coverage: 100%)**
  sets the colidable status of the cell. Tested to check if cell is correctly set to collideable

Game Package:
CollisionFinder

- **checkMapCollision(Celebrity character) (line coverage:100%, Branch coverage: 100%)**

Used to find out where the character is about to go and passes that information onto the “checkCellType” method to take the appropriate action depending on the cell type. In order to test it we checked each way the character can move (up, down, left, right), and checked that the correct information was passed to the “checkCellType” method.

- **checkEnemyMapCollision(Paparazzi enemy) (line coverage:100%, Branch coverage: 100%)**

Used to check where enemies are about to go and stop them if they collide in the map. In order to test this method, we checked each way the enemy can move (up, down, left, right), and checked if they were stopped when they collided with the map.

- **checkEnemyCollision(Paparazzi enemy)(line coverage:100%, Branch coverage: 100%)**
  Used to check if enemies are about to collide into each other and stop them. We tested this method by having an enemy approach another enemy from the top, bottom, left, and right, and check if it was stopped.

- **checkCellTyp**e**(int cellType, int rowIndex, int colIndex, Celebrity character) (line coverage:100%, Branch coverage: 100%)**
  Given the type of cell and location the player is about to walk into, this method takes the appropriate action for the game and updates the map. In order to test this method, we created scenarios where a player is walking into each type of cell and checked if the correct actions were taken.

- **resetDisguises() (line coverage:100%, Branch coverage: 100%)**
  Used to reset the number of disguises collected. To test, we collected disguises and checked to see if the number was zero after a reset.

EnemyMovement

- **getTotalCost(Node node) (line coverage:100%, Branch coverage: 100%)**
  Takes a node and gets its different costs used for A* path finding algorithm. Tested by passing in a node and checking if it we calculate the correct costs

- **search() (line coverage:100%, Branch coverage: 100%)**
  Goes through the nodes in the nodes array until the current node is the goal node. Tested by checking if the current node is the goal nod at the end of the method call.

- **openNode(Node node) (line coverage:100%, Branch coverage: 100%)**
  Marks a node as opened, sets its parent to the current "current node" and adds the node to the ArrayList which tracks opened nodes. Tested by checking if a node passed to this method is marked as opened, its parent is the current node, and checking if it's in the opened ArrayList and by passing a node already marked as opened to see if it was still added to the ArrayList.

- **trackPath() (line coverage:100%, Branch coverage: 100%)**
  Walks back from the goal node to the start node while adding each node to the path ArrayList. Tested by checking if the path Arraylist had the correct values after the method was called.

- **resetNodes() (line coverage:100%, Branch coverage: 100%)**
  Resets the properties of all the nodes, clears the path and opened Arraylist, and sets succeeded as false. Tested by checking if each of these elements were back to their default state.

- **getNextColumn() (line coverage:100%, Branch coverage: 100%)**
  Returns the column of the first node in the path. Tested to check if the first node and its column are correct.

- **getNextRow() (line coverage:100%, Branch coverage: 100%)**
  Returns the row of the first node in the path. Tested to check if the first node and its row are correct.

Util Package:

Point

- **getX()(line coverage:100%, Branch coverage: 100%)**
  returns the current value of the x value of the point, checked by creating a new point and testing the value of x.
- **getY()(line coverage:100%, Branch coverage: 100%)**
  returns the current value of the y value of the point, checked by creating a new point and testing the value of y.

- **substractY(int val)(line coverage:100%, Branch coverage: 100%)**
  subtract the value to the current value of the Y. Tested to check if the Y value was correctly changed

- **addY(int val)(line coverage:100%, Branch coverage: 100%)**
  add the value to the current value of the Y. Tested to check if the Y value was correctly changed

- **substarctX(int val)(line coverage:100%, Branch coverage: 100%)**
  subtract the value to the current value of the X. Tested to check if the X value was correctly changed

- **addX(int val)(line coverage:100%, Branch coverage: 100%)**
  add the value to the current value of the X. Tested to check if the X value was correctly changed

Score

- **getScore()(line coverage:100%, Branch coverage: 100%)**
  returns the current score that is used incremented during rewards accumulation. Tested to check if the correct value is returned.

- **restartScore(int val)(line coverage:100%, Branch coverage: 100%)**
  sets the score to zero. Tested to check if score is correctly reset

- **addScore(int val)(line coverage:100%, Branch coverage: 100%)**
  adds the value to the score. Tested to check if score changed correctly

- **substractScore(int val)(line coverage:100%, Branch coverage: 100%)**
  subtracts the value from the score. Tested to check if score changed correctly

Line and Branch Coverage:

The line and branch coverage for each method tested can be seen above. In order to ensure that we had very good quality tests, we created multiple test cases for methods that needed them in order to cover as much code as possible and help find as many bugs as possible. We made control flow graphs for many of the methods we tested to see exactly how the method behaves when it needs to make a decision and what are the outcomes of making each decision. Also, we looked over each other's test that we created to ensure that the tests we were making made sense and were as efficient as possible.

Bugs Found:

We had a problem where sometimes our project would fail to build and would not import modules correctly. It was because we had two different JRE systems library versions on our project which resulted in us not being able to run or test our game. We also found a bug on the winning screen of our game which made the restart test yellow. Another bug we found was in the timer. When we placed the timer on the outside of the window size we set, the timer was working incorrectly. There was also a bug in the enemy movement when we created four rooms in the maze. Two enemies stopped moving toward the main character when they hit the wall or barrier.

Code changes:

In the EnemyMovement class, we realized that the clearAll() method was not necessary. Originally it was put in as a complete reset to everything in the class in order to get ready for a potential game reset. But we realized that the resetNodes() method does essentially the same thing but it keeps the original nodes array and sets the values of nodes in the array back to the default values. Originally, we had thought that the nodes array should be made again from scratch in order to ensure the reset was done correctly. But as we were testing, we realized that the clearAll() method is unnecessary.

We removed our entity superclass as the only thing inheriting from it were the moveable entities. Originally, we had planned to have movable and stationary entity classes in our game. But after implementation, we made the decision to not have classes for our stationary entities. Since only the movable entities were inheriting from the entities super class. We decided to move the methods from the entities superclass to the moveable entities superclass in order to make our code simpler and easier to understand. Now the two moveable entities “Celebrity” and “Paparazzi” inherit their properties directly through the “MoveableEntities” superclass instead of going up another step to the “Entity” superclass.

We added many getter and setter methods to aid our ability to test our code and to also improve how elements were passed between classes.

What we learnt:

None of us have ever done unit testing and we have never used JUnit. We had to do a lot of research to ensure we had good quality for our tests, and that our tests were testing the right things. We also had to keep in mind that we had as much line and branch coverage as possible. Which meant we need to improve our knowledge of how to create control flow graphs. We learnt that unit testing allows us to refactor our code and make sure the module still works correctly. Also, we realized how important and useful it is to have Javadoc comments. Since they were provided for each method, we needed less time to understand the code that other group members wrote and get started writing unit tests. At the same time, we learnt that updating those comments is also important when we make some changes to the functionality of the methods.