Testing was quite a unique experience. Traditionally in the past, to test our programs we would simply run our programs with different inputs and try to deal with edge cases. With all our previous assignments in previous classes being run through the command line, this meant plugging in different values every run to test edge cases. I specifically remember one case in CMPT 125 where we made a memory card game, and would pipe a .txt file in as the input to test all possible combinations of inputs. With JUnit, we were able to practice testing specific functions themselves and find issues instead of assessing the final product and reverse engineering where the issue propagated up from.

Unit and Integration Tests

We started from the top package, that being Entities, and worked our way down. Within the Entities package, the Entity class is a superclass, in which all other entities implement the supers method by simply calling super.methodName(). Thus, we tested all the methods within the Entity class. We then tested all the unique methods found within Player, Reward, and Trap classes. As entities only store a point and their respective entity type, we only had to test the getters and setters. In the Player class, we test the alive boolean attribute with its getters and setters, score getter and setter, and score decrement and increment methods. In the Reward class and Trap class, we test the status method alongside their respective methods that interact with the entity, and return the new score.

We didn’t test EventHandler in an independent sense like we did with Entities. We didn't implement any unit tests for this class as it only includes 2 methods, both essentially running the whole game, and not letting us test in JUnit by asserting anything. This was one of the Classes where we had to test simply by trial and error, by playing the game and seeing how inputs and events, like controlling the character using WASD, or mouse clicks for menus panned out.

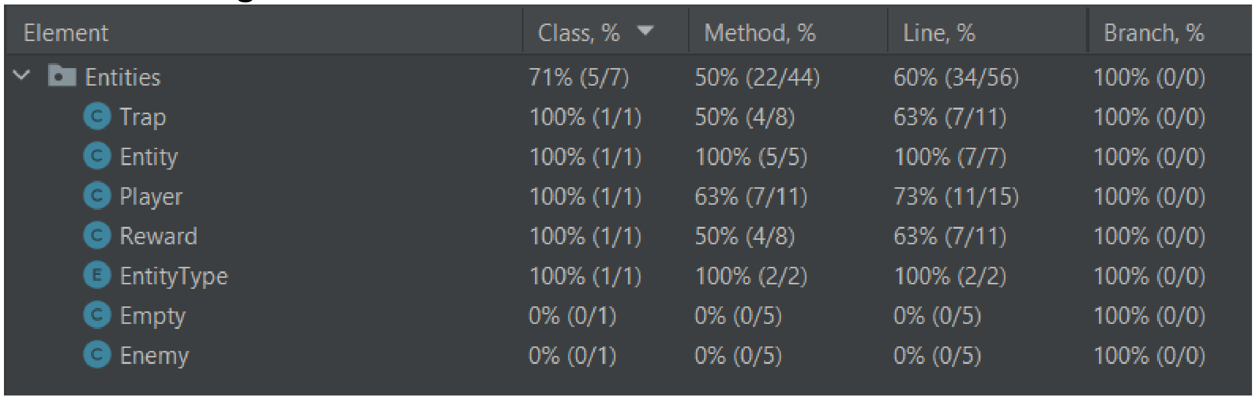
Map Package had several classes we tested. Starting at the most primitive class in the package is the Point Class. We tested the getters and setters for the height and width attributes, and then had 2 functions left to test. The first was an equals function to check if two points were the same. The implementation of this method checks if the height and width match between both points. The second is a move method. To test this we instantiated a point, and then passed in a move direction. We then asserted that the new height equals what we expected it to be, and it did. We then move up a level to our Cell class. Our Cell class stores a Point, which we previously covered, alongside an enum for the cell type, and an entity. We simply followed the same structure we had in the previous tests, testing getters and setters. We then had a helper function to test if the cell is of type wall or barricade, so the player wouldn't be able to move to that cell. So we instantiated a cell, asserted that it is false as the cell is initialized as a path, we then set the cell type to wall and asserted true. The highest level for the Map package is the Map class. This class creates a 2D array of cells for the game to operate upon. We tested if the height and width of the map are true to what they should be. As the maze is randomly generated, it proved difficult to test as each time the test was run it would potentially have different results. This meant that when testing, we simply asserted not null for if the maze was generated. A concrete factor of all randomized mazes is that the starting position is always 1,1 in the array, therefore we were able to test if the starting cell was correctly set as a path, and not a barricade or wall. The exit cell doesn't exist until the game reaches a win condition, so to test we assert null for getExitCell(), then we set the exit cell to open so it's created, and then assert not null. Last key function was a swap entity method. To test this we create a maze, and two entities, swap their entity type, and then check if they successfully changed. Game Class is the main driver of the logic of the game. Once again we found it difficult to test as the maze was randomly generated on each creation of a game object. To generate the map 3 parameters are passed in, Enemy, Reward, and Trap counts, so to test map generation we tested if the values passed in were properly stored in the game object. For the reset method it followed the same principle but we check if the exit cell is closed after resetting. We then have several getters and setters that we test, alongside boolean methods that checks the game's state and returns true whether the game is running, idle, won, or lost. We have a method to remove all rewards, so we test the size of the rewards list before and after removing them. Move player was a tricky method to test as the game is randomized, and we didn't know which cell would be open to move to. This problem was remedied by an if else statement in the test itself, checking if the player could possibly move right or down, and then whichever way the player can move, they do and we test their new position in the game grid. We then test methods for getting the lists of entities and the size of each list.

The Ui package received zero testing as well. This doesn't mean it's an unfinished section riddled with bugs, but rather we could not write unit tests for the GUI. Testing for this came from just playing the game, and recording any issues we encountered

Test Quality and Coverage

Initially I thought our code coverage would be quite low, as most of the methods we were testing were getters and setters. However, Intellij has a code coverage system that lets us view line, method, and branch coverage for each class, and each package. Below we will cover all the packages and discuss why coverage might be low or non-existent in certain places.

Entities Package



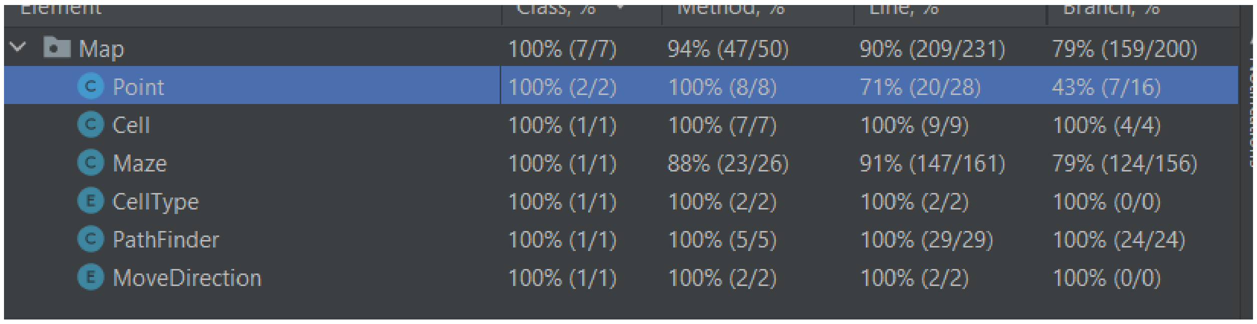
Element	Class, %	Method, %	Line, %	Branch, %
Entities	71% (5/7)	50% (22/44)	60% (34/56)	100% (0/0)
Trap	100% (1/1)	50% (4/8)	63% (7/11)	100% (0/0)
Entity	100% (1/1)	100% (5/5)	100% (7/7)	100% (0/0)
Player	100% (1/1)	63% (7/11)	73% (11/15)	100% (0/0)
Reward	100% (1/1)	50% (4/8)	63% (7/11)	100% (0/0)
EntityType	100% (1/1)	100% (2/2)	100% (2/2)	100% (0/0)
Empty	0% (0/1)	0% (0/5)	0% (0/5)	100% (0/0)
Enemy	0% (0/1)	0% (0/5)	0% (0/5)	100% (0/0)

The Entity class itself has 100% coverage. The Trap, Player, and Reward class has lower coverage due to it inheriting methods from the superclass. Because those functions call super.exampleMethodName(), we didn't write tests for them, as due to the fact that the tests ran in the super class, calling super in a subclass inherently would work. Without those duplicate methods, we would have 100% coverage in them too. Empty and Enemy are essentially duplicates of the Entity class with no unique methods, thus we didn't write any tests.

EventHandler Package

Like stated above when discussing unit tests for each package, we weren't able to test the 2 methods in EventHandler class. The events were tested not in code, but rather in practice.

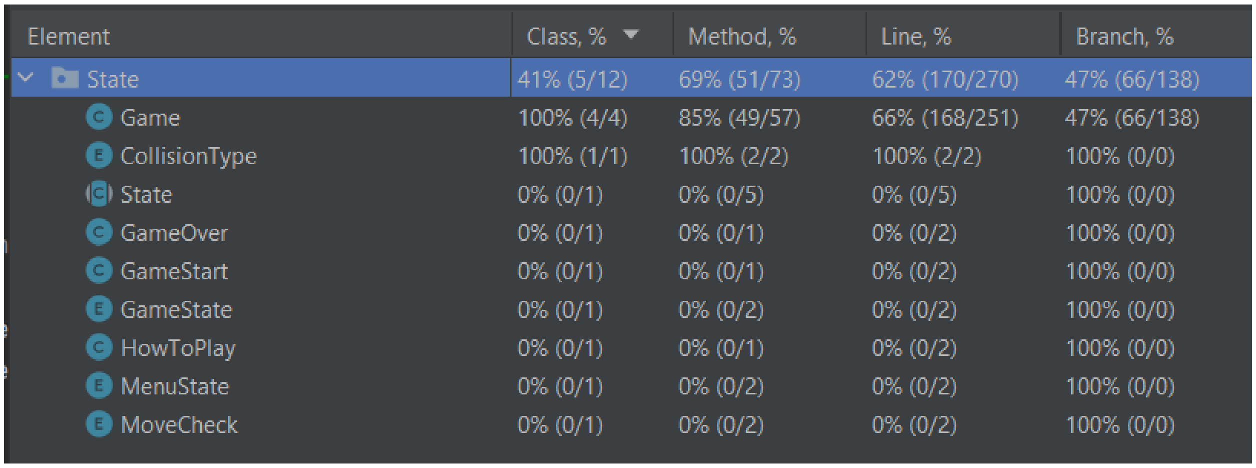
Map Package



Element	Class, %	Method, %	Line, %	Branch, %
Map	100% (7/7)	94% (47/50)	90% (209/231)	79% (159/200)
Point	100% (2/2)	100% (8/8)	71% (20/28)	43% (7/16)
Cell	100% (1/1)	100% (7/7)	100% (9/9)	100% (4/4)
Maze	100% (1/1)	88% (23/26)	91% (147/161)	79% (124/156)
CellType	100% (1/1)	100% (2/2)	100% (2/2)	100% (0/0)
PathFinder	100% (1/1)	100% (5/5)	100% (29/29)	100% (24/24)
MoveDirection	100% (1/1)	100% (2/2)	100% (2/2)	100% (0/0)

Point, Cell, and Maze were the classes we tested. Branch coverage on Point is pretty disappointing to see, but relatively speaking we had good coverage for those 3 classes. I'm not sure as to why Intellij is reporting 100% coverage for PathFinder and Enums, but we didn't test the PathFinder class and you don't test Enums. The reason we didn't test PathFinder is simply due to time. With other commitments and PathFinder being implemented relatively late, we didn't realize and remember to implement testing for it, so although the image says 100% coverage, it's 0% in reality.

State



Element	Class, %	Method, %	Line, %	Branch, %
State	41% (5/12)	69% (51/73)	62% (170/270)	47% (66/138)
Game	100% (4/4)	85% (49/57)	66% (168/251)	47% (66/138)
CollisionType	100% (1/1)	100% (2/2)	100% (2/2)	100% (0/0)
State	0% (0/1)	0% (0/5)	0% (0/5)	100% (0/0)
GameOver	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
GameStart	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
GameState	0% (0/1)	0% (0/2)	0% (0/2)	100% (0/0)
HowToPlay	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
MenuState	0% (0/1)	0% (0/2)	0% (0/2)	100% (0/0)
MoveCheck	0% (0/1)	0% (0/2)	0% (0/2)	100% (0/0)

The Game class is what runs the logic of the game, due to it being a randomized maze, it was incredibly difficult to test all facets and branches. This resulted in a rather low coverage % for Method, Line, and Branch. All the other Classes that have 0% coverage are dead classes we aren't using anymore, however as I'm typing this report I don't necessarily want to delete them just in case it breaks the game. Please ignore and keep in mind we will refactor before the final submission.

UI

As with the EventHandler package, we could not implement any unit tests for GUI elements and rendering the game. To test this we once again had to simply play the game. This resulted in us finding any issues and adding them to a thread in our discord to fix.

Findings

Due to a lot of the methods in each class being getters and setters, we didn't find any issues, as they are incredibly simple in nature. We actually made a lot of changes to our production code in this phase as we weren't finished implementing everything we wanted to in phase 2. For testing before the UI was implemented, we ran the game in console in a text format to output the games grid and map. Because of this, and the fact that this was in phase 2, we were solving any issues and bugs we ran into as soon as they happened. By the time we actually got around to implementing unit tests, they all passed the first try. This isn't really helpful in terms of developing this game as the tests didn't reveal any bugs, but rather served more as a learning experience for JUnit and testing in general. Now that we have learned the basics of testing, we can implement it simultaneously with development of future projects, rather than waiting till the end and not having any bugs reported with testing. Bugs we found in testing by playing the game were few in nature but quite drastic. One issue we had was height and width weren't synced between the game and the UI, thus resulting in controls being flipped 90 degrees. Other issues we had that couldn't be found through testing but rather playing the game was that enemies would simply stop moving after a certain period of time. It was strange as it wasn't after a set amount of time, but would occur rather randomly. This was remedied by implementing a new moving algorithm for the enemies.