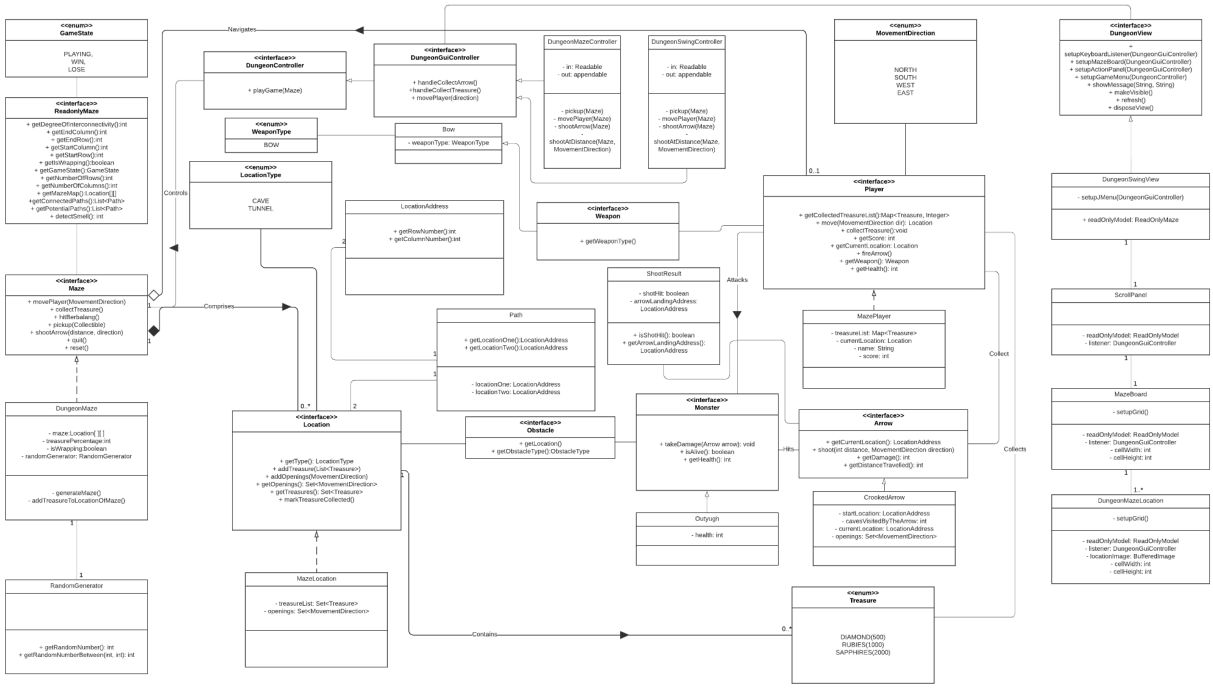
Project 5 - Graphical Adventure Game

Anuj Ashok Potdar - 001098687



Design Implementation -

1. My design consists of a ***Maze*** interface which consists of all the methods of a maze. The maze interface is now segregated into two interfaces, namely, the ***ReadonlyMaze*** interface that contains only the read methods of the Maze and the existing Maze interface now only has the writing methods but as it extends the ReadonlyMaze interface it has both the read and write methods. Methods that are required for the maze are as follows -
    a. **getIsWrapping(): boolean** - This represents whether the dungeon's maze is wrapping or not.
    b. **getDegreeOfInterconnectivity(): int** - Degree of Interconnectivity represents the number of pathways from one given location to another.
    c. **getEndColumn(): int** - The column number of the end of the maze where the player is expected to reach.
    d. **getEndRow(): int** - The row number of the end of the maze where the player is expected to reach.
    e. **getStartColumn(): int** - The column number of the start of the maze where the player is going to start from.
    f. **getStartRow(): int** - The row number of the start of the maze where the player is going to start from.

g. **getPotentialPaths(): List<Path>** - This provides a list of all the paths that are possible in the maze. Each point connects to its potential 4 neighbors both in wrapping and non-wrapping maze. This method returns a defensive copy of the list of potential paths to the calling function.
h. **getConnectedPaths(): List<Path> -** This provides a list of all the paths connected such that there exists one and only one path between 2 points when the degree of interconnectivity is 0.
i. **getMazeMap(): Location[][]** - Returns a defensive deep copy of the maze map that is generated in the dungeon, this is useful to print in the driver class or UI.
j. **detectSmell(): int** - This method returns the level of smell in integer, 1 being bad and anything more than 1 being terrible. The smell level is calculated at runtime.
k. **shootArrow():** Shoots an arrow at the specified distance and direction in order to kill the Otyugh monster.

2. The ***Maze*** interface encapsulates the methods only of a maze and the implementation of this is the DungeonMaze class which provides specifics of the maze as per the dungeon problem statement.

3. The ***DugeonMaze*** class uses RandomGenerator as a separate entity to generate the random maze as per Kruskal's Algorithm in a private function. The following are the helper methods in the class-
    a. **generateMaze()** - This method generates the whole maze by taking in the parameters that are provided by the user while constructing the object of the DungeonMaze class. These parameters are whether the maze is allowing wrapping or not, the degree of interconnectivity of the maze, the number of rows and columns in the maze.
    b. **addTreasureToLocationOfMaze()** - The generateMaze function utilizes this method to add a random treasure to the specified percentage of caves as per the user's input.

4. Each cell in the maze is represented by a ***Location***, this is further categorized into two sub-categories (enum ***LocationType***)-
    a. **Caves** - These can have treasures in them and have either 1, 3, or 4 openings (exits and entrances combined) to them. There can be one or more treasures in one cave hence a list of treasures is used.
    b. **Tunnels** - A tunnel can only have 2 openings, and they cannot hold any treasure in them but just like a cave they are occupying one cell.

5. The following are the functions available in the ***Location*** Interface -

a. **getType(): LocationType** - denotes whether the Location is a cave or a tunnel from the number of openings, that is, if there are 2 openings then it is a tunnel else it is categorized as a Cave.
b. **addTreasure(List<Treasure>) -** This method is called to add a treasure or treasures to the selected location. Here we first validate whether the location is a Cave and only then do we add a treasure to the location.
c. **getOpenings(): Set<MovementDirection> -** Provides the openings of the location, this represents whether there is a path from a given location to which of its side.
d. **markTreasureCollected() -** Marks the treasure as collected by clearing the treasure set in the location.
e. **getTreasures(): Set<Treasure> -** Returns the set of treasure to the calling block of code. This is useful for the player to collect treasure.

6. The ***MazeLocation*** class is an implementation of the LocationInterface that houses the fields like the LocationType, TreasureList, and the number of openings of the location.

7. Each MazeLocation is connected to other through a ***Path*** and the Path is comprised of exactly 2 ***LocationAddresses*** that uniquely identifies each Location in the grid.

8. Each location has a treasure list. As per the problem statement, there are three types of ***Treasure*** -
    a. **Diamond -** I have assigned 500 points to this treasure i.e. when a player collects it they are awarded 500 points.
    b. **Rubies** - 1000 points.
    c. **Sapphires** - 2000 points.

9. The ***Player*** interface represents methods of the player that are used while navigating through the maze. They are as follows -
    a. **getCollectedTreasureList(): Map<Treasure, Integer> -** Get a map of all the treasures and the count of each of the treasures.
    b. **getCurrentLocation(): Location -** Get the current location of the player in the maze.
    c. **move(MovementDirection dir): Location -** Move the player either North, South, West, or East.
    d. **collectTreasure(List<Treasure>): void -** Collect the treasure in the location and it to the player's score and treasure Map

10. The ***MazePlayer*** class implements the Player interface and it has the current location, score, name, and treasure list fields and the **computeNextStep()** method which

calculates the next step required to be taken by the player in order to reach the destination location.

11. The ***Monster*** interface represents the monster that resides in the cave who eats the player when they are at their full health or when their health is half the fight with the player and has a 50-50 chance of either eating the player or the player escape. It has 3 methods, namely -
    a. **takeDamage(Arrow arrow): void** - Recording the damage done to the monster by the arrow that is fired to it.
    b. **isAlive(): boolean** - Signifies if the monster is alive or not by checking its health. If the health of the monster is 0 then this method returns a false.
    c. **getHealth(): boolean** - Returns the health out of 100 of the monster.

12. The **Otyugh** class implements the Monster Interface and defines the methods of a monster that include showing the alive status of the monster, the current of the monster, and record damage from an arrow.

13. The ***Arrow*** Interface represents the weapon arrow used by the player to slay the monster, it houses the methods to move around the maze in both wrapping and non-wrapping fashion, the distance traveled by the arrow, and the damage done on each hit.

14. The **CrookedArrow** class implements the Arrow interface and defines the movement strategy of the arrow in caves and tunnels. A CrookedArrow is expected to go through tunnels and bend through the openings of the tunnels endlessly until it comes to a stop be either exhausting the fire distance or when it reaches a cave that does not have an opening in the direction in which the arrow is going.

15. The **ShootResult** enum shows whether the outcome of the shoot was a HIT or a MISS depending on whether it hits the monster or not.

16. **DungeonGuiController** is a new controller that houses the methods of the interaction from the view to the model, this is required to perform write operations in the model from the view as the model that the view has is the read-only variant of the model.
    a. The DungeonGuiController does not have any methods that are specific to the library that is used to implement the GUI, they are generic in order to be open for extension and closed for modification.
    b. There is only one instance of the controller in the whole game.
    c. It is implemented by a class **DungeonSwingView** which has the Java Swing specific implementations of the view.

17. The **DungeonView** interface houses the methods of the view that is used to perform the interaction with the model. The DungeonView is then extended by the **DungeonSwingView** which performs the Java Swing Specific implementation of the view. This way the view still stays isolated from library-specific logic and the interface is still generic.
18. The DungeonView is comprised of a **ScrollPanel** that houses the **MazeBoard** which has a GridLayout and the elements of the GridLayout are wrapped in **DungeonMazeLocation** class objects which represents each Location in the Dungeon being either a cave or a tunnel.

Testing Plan -

1. **Unit Testing -**
    a. Validating the constructors of DungeonMaze, MazeLocation, and MazePlayer for illegal arguments.
    b. Test the points added when the treasure is collected.
    c. Test the maze's degree of interconnectivity by writing an appropriate test case.
    d. Test all the methods of the Player interface, Maze Interface, and Location interface.
    e. Test the movement of the player through the maze.
    f. Test illegal movement of the player. For example, moving the player to the North when there is no opening available in the North direction.
    g. Test that there is always a valid path between the start and the end location for every randomly generated path.
    h. Test that at least the percentage specified by the user number of caves has the treasure.
    i. Test that arrows are found in both tunnels and caves.
    j. Test that Otyughs are found only in caves and not in tunnels.
    k. Test that the number of Otyughs is equal to the difficulty parameter.

2. **Integration Testing -**
    a. For testing the whole system in integration testing I have created the random generator class whose reference will be passed throughout the program run.
    b. There will be 2 constructors in the DungeonMaze class, one which accepts a randomGenerator object as the parameter and one which does not.
    c. The constructor that accepts the randomGenerator as a parameter can be used to test the Object by mocking the random numbers that are to be generated as we will have the reference of the randomGenerator that we pass throughout the DungeonMaze and we can have a specific file for testing module of the ***RandomGenerator*** class that has the logic to mock the random numbers.
    d. Once we are aware of the sequence of random numbers that are generated we will be able to know the output of the maze, the steps required and the final output in terms of score and the treasures collected.
    e. This way we can perform the integration testing of the module even after refactoring the code.