# Escape Game Project

## Final Report

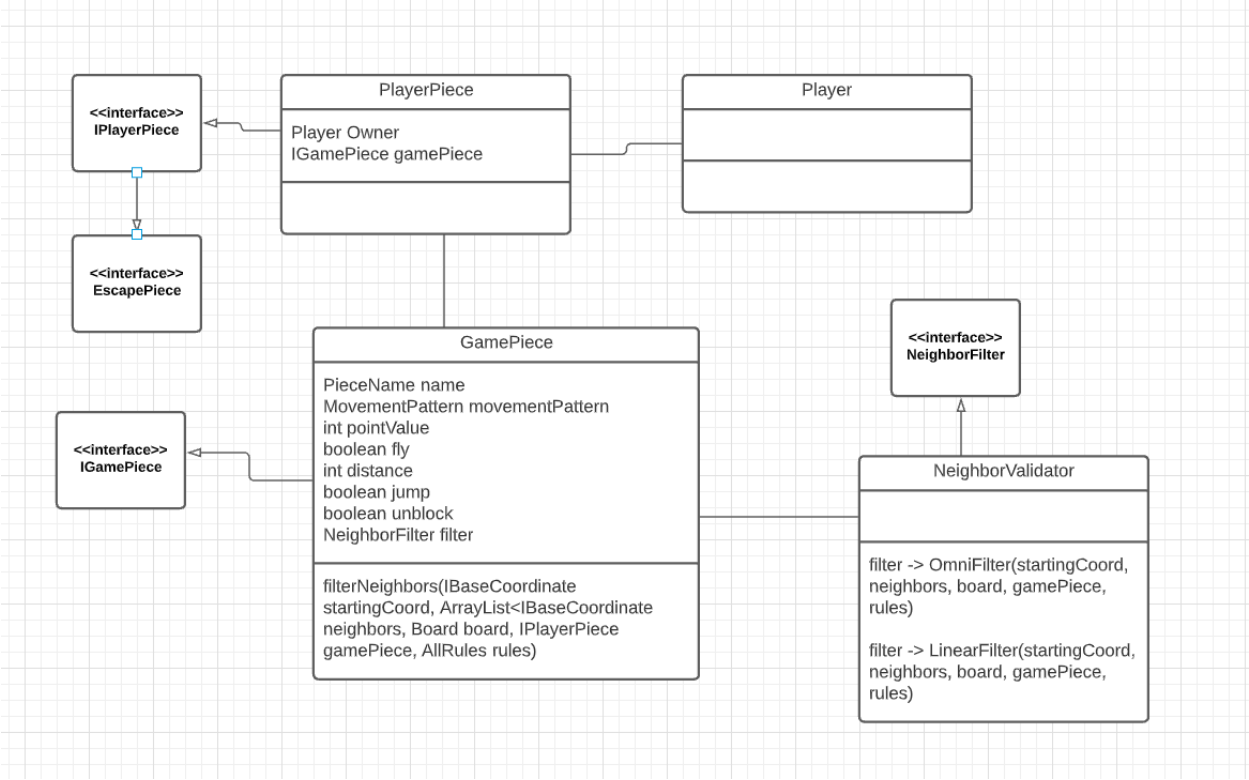
Yash Patel
12/15/2021

## Introduction

The escape project I designed follows the different designs and patterns throughout the project. The project needed to implement these designs so that the project is readable, intuitive, and maintainable. I also followed the TDD process throughout the entire project using the TODO.txt file. Throughout the project, I always tested with coverage on all the tests to ensure coverage and previous tests are still passing. Please note that the UML diagrams don't show all the variables and methods for simplicity for the reader especially the getters and setters.

## Escape Piece

To start understanding the project, I want to show the smallest components. The first of these is how the game pieces are implemented.



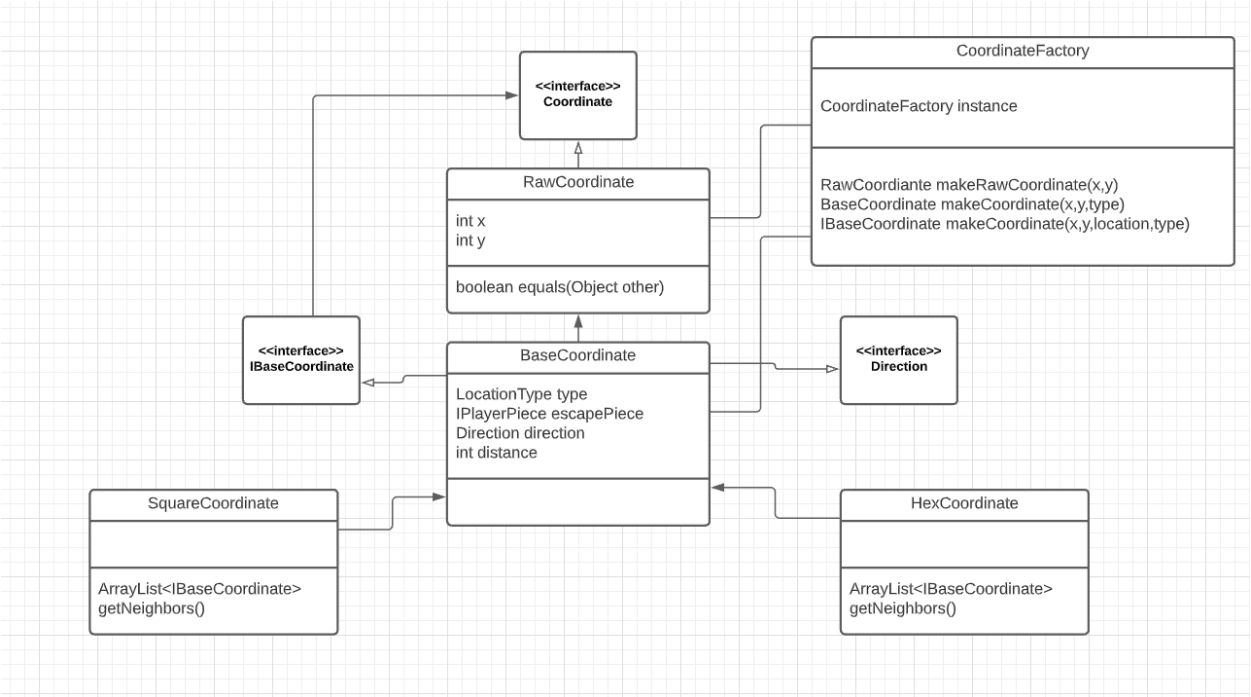

Here is the UML diagram for how the game pieces are done. I split the idea of an escape piece to a PlayerPiece and GamePiece. A GamePiece is the description of an escape piece. It will hold its attributes, name, movement pattern, and filter. You can think of a GamePiece like the Rook or Bishop in Chess. It is idea of a game piece and it doesn't have an owner or show how many pieces there are. The GamePiece implements the IGamePiece which holds all the public functions of the piece. The point to note is NeighborFilter. This is a functional interface to reduce the class explosion and doesn't have a separate function for each GamePiece. The filter is used

for PathFinding where it will filter the neighbors of the GamePiece to correspond to the movement pattern and attributes of the piece.

The other class is PlayerPiece. This holds an individual piece on the board. So it holds a game piece's owner. The purpose of separating PlayerPiece and GamePiece is so that if there is a new GamePiece on the board for example a cow, PlayerPiece is abstracted enough so it doesn't matter what type of GamePiece is on the board. This follows the Liskov substitution principle.

## Coordinate

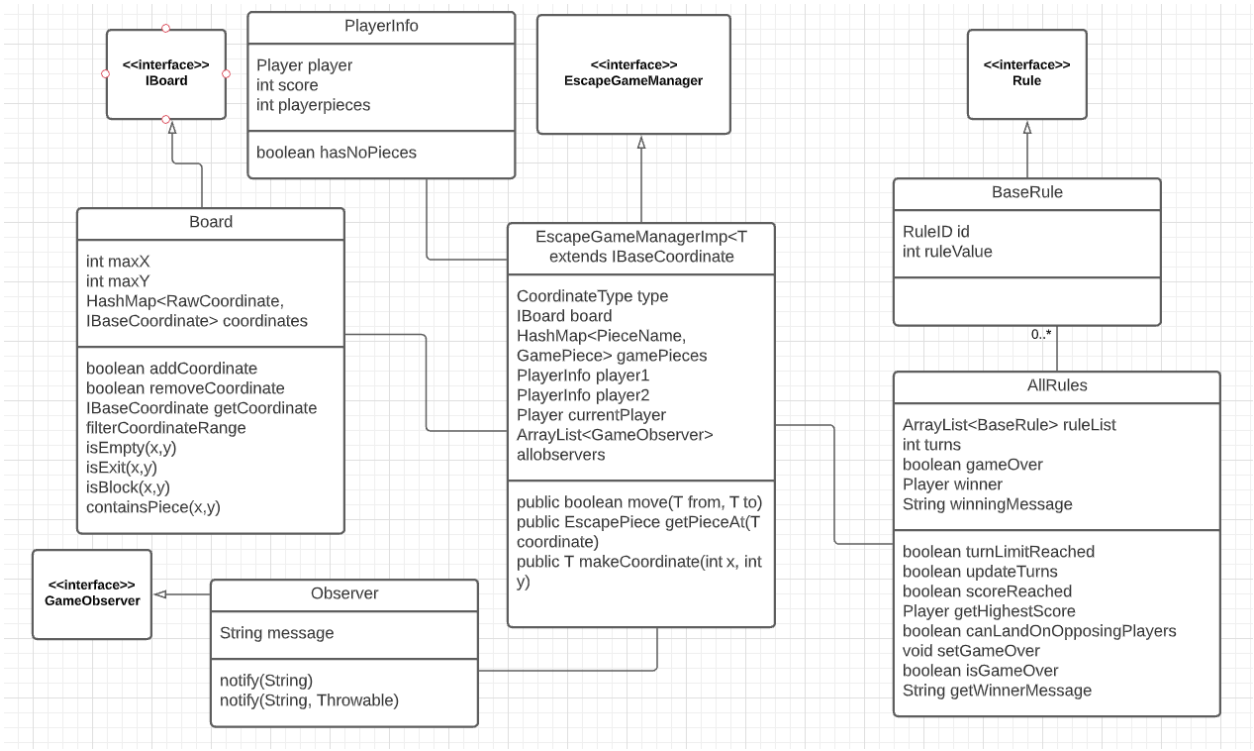
The next component of our game is Coordinate. The way I implemented Coordinate was that it stores the x,y location, the type of location, and whatever game piece is on it.




At the highest level, we have a RawCoordinate which is just the x and y of a coordinate. This is used inside of Board. Then we have IBaseCoordinate which holds the public functions for a coordinate for example, getting the location or getting game piece. A BaseCoordinate is an abstract class to ensure nothing can be created without defining the type but it also holds all of the similar functions that extend BaseCoordinate. Those two functions are Square and HexCoordinate. From the coder's perspective, I wanted the game to not functionally be different depending on Square and Hex. So I noticed the only difference is the Neighbors. This follows good design practice because if I wanted a new coordinate, I can make a new class and defining just the neighbors. So if I were to implement Triangle, it won't affect Square or Hex coordinates. Also, the IBaseCoordinate holds direction and distance. This is for pathfinding. The direction is where a gamepiece is coming from and distance is how far the game piece has traveled.

Another design pattern we have is the CoordinateFactory. The coordinate factory is a Singleton so the user only needs to use the CoordinateFactory to make the different coordinates. The coordinate factory only needs to the type to give the correct coordinate. It also is used to make RawCoordinate. The reason it is a singleton is because we don't want to have multiple factories to waste memory and avoid confusion.

## Game Manager




The GameManager is the basis for the whole Escape game. It is the connection between the board, observers, pathfinding, and rules. The GameManager first holds functions that create the different objects from the game builder. The game manager holds these variables globally so that it can be used for move, makeCoordinate, and getPieceAt.

The Rules component deals with the rules of the game and holds functions on deciding when the game is over. A BaseRule holds one rule in the game such as SCORE, TURN_LIMIT, or REMOVE. AllRules function holds a list of BaseRules. The reasoning for this is so that more rules can be added such as point_conflict. AllRules have the functions to determine to update the turns, check if game is over depending on the different rules. AllRules also hold the winner of the game and the winning message. This is so that the GameManager doesn't need to worry about the specifications of the rules and it only needs to check if the game is over.

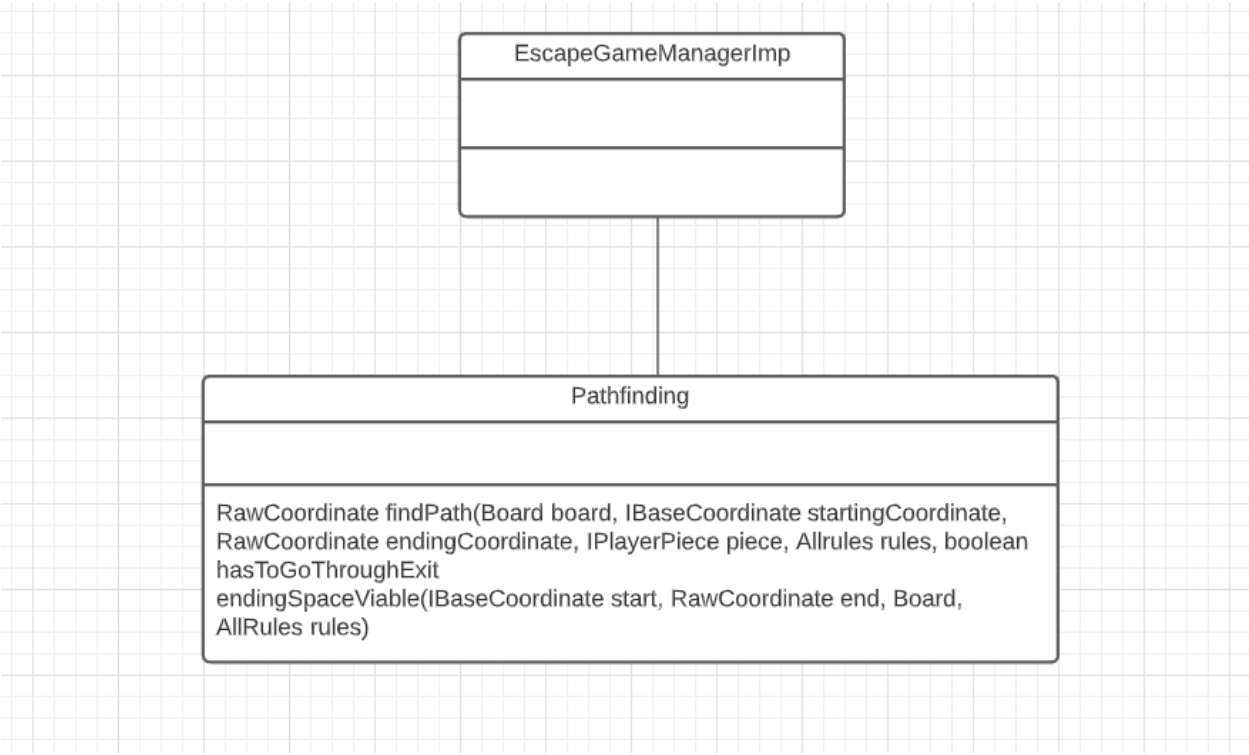
There is the observer component. The BoardGameManagerImp has a list of GameObservers. The manager has a method called notifyAll which iterates through all the observers and notifies each of them with the same method. This follows good design practice

since manager doesn't need to worry about every observer but instead calls just one function which handles all of the observers.

The PlayerInfo component holds the information for each player in the game. The GameManager is designed so that only 2 PlayerInfo's can be stored, player1 and player2. PlayerInfo holds the score and number of pieces left for that player. Each of these attributes are pulled by the GameManager and given to the Rules component. This ensures no interaction between two classes.

The largest of the components is the IBoard. The board needed to be infinite or finite and holds any type of configuration. So I used a HashMap<RawCoordinate, IBaseCoordinate>. This hash map is used to map an x and y to the actual location on the board. The reasoning for this instead of a separate location class is for pathfinding where I felt it was weird to me to think that a location would know its neighbors and the type of tile. The hash map allowed for infinite size and it stored only the special locations on the map (GamePieces, Block, Exit). The board also holds helper functions so that we don't need to access the hashmap from other classes. This helps protect the manipulation of the data.

## Pathfinding




The Pathfinding class isn't an object itself but a utility for the game. It takes in the starting location, ending location, board, game piece, and rules to determine if there is a path to the exit. I used a BFS to easily keep track of distance and direction. The pathfinding is done is steps. It first checks on whatever coordinate we are on, if it is the destination and if it is a viable

destination. If it is then return that coordinate, otherwise we need to queue that coordinate's neighbors. What I do is first get all of the neighbors. Then I filter them depending on board restrictions, game piece movement pattern and attributes, and whether we should be avoiding exit or not. Eventually this returns null if there is no path or a coordinate to the final destination. I returned a coordinate in case we passed an exit then we go through the exit.

## One Turn

A turn is done inside the GameManagerImp class. A turn first checks if the game is over. If it is then notify the observers. Otherwise check if we are able to pathfind (Coord has a game piece, not an exit, not going to the same coordinate, we don't end on a block) Then we pathfind without tracking exits. This is so we can ensure a path without exits. If there is no path then we have exits turned back on. If there still is no path then return null otherwise return the exit. We then have to update the real game. We move the piece to the new locations and adjust points if needed. After that, we check if the game is over using the AllRules class. Then we update the turn count if Player2 and switch players.

## What I learned

I have learned better practices for my code such as Singletons and FunctionalInterface. With each of these, I learned to not duplicate my code and not cause too many classes in my project. I also learned about the names of the different Designs. I was already writing decent code when starting the class but I didn't realize I was implementing Design patterns. I really enjoyed learning about Streams which helped my code reduce the number of looping I did.

## Improvements

I wish I implemented the Board to be a singleton. So that I only have one board object for the entire Manager. However, I didn't do that because I wrote tests that create different boards and each boards have different configurations. I didn't have the time to restructure that. I also wish I used a functional interface for the rules. So each rule has a check function that will determine if that rule has been reached or violated.

## Extra Credit

I had a bit of extra time and I wanted to implement the other movement patterns because I did a lot of set up for them. I implemented Linear, Orthogonal, and Skew patterns. The way I designed movement patterns was to easily add more patterns. It simply takes in the neighbors

and filters to the correct directions. It will still use the same filterAttributes function. All of these will be in the NeighborValidator class.