Darian Tavana

12/16/21

CS4233: Police

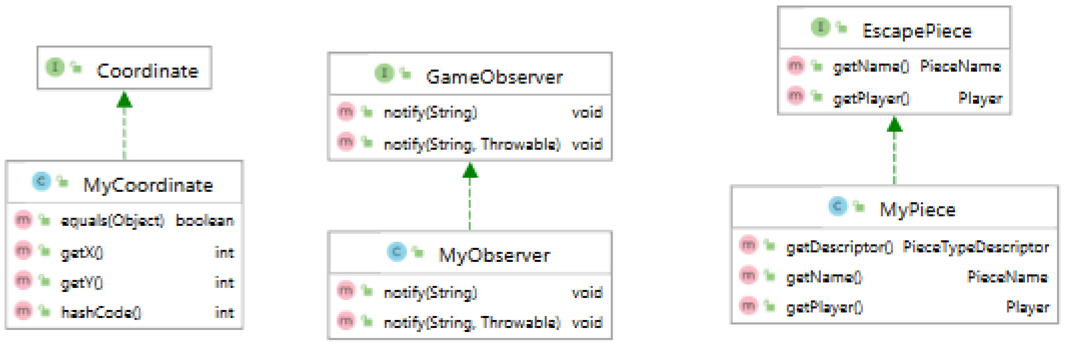
Escape Final Report

Directory Structure

Overall, the project is split into five packages. The packages include **component, exception, gamedef, manager, and util**. The **component** and **manager** packages will be discussed in detail later in this document. However, I will provide some insight as to what exactly lives in each of the other three packages. First, the **gamedef** package contains all the initial interfaces and enumerations given to students as part of the starting codebase. The **exception** package contains a single class which is the blanket **EscapeException** class. And finally, the **util** package contains classes that are mainly used in parsing the config file. I opted to go for this overall project structure as it allowed me to easily separate what classes were provided to me from the classes I needed to create myself.

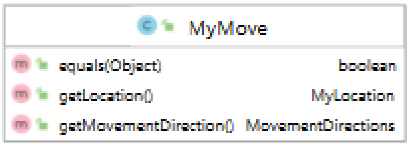
Component Package

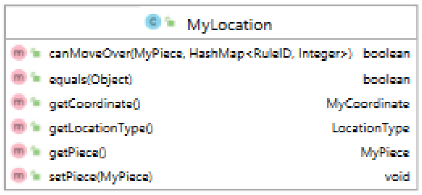
In the **component** package, all my implementations of definitions found in the **gamedef** package are held. In other words, all the interfaces that were provided with the initial codebase for the project such as **Coordinate, GameObserver, and EscapePiece** have classes accompanied with them located in this package.

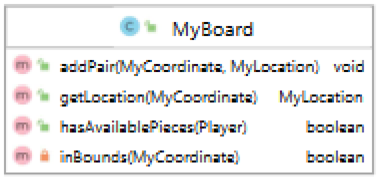


In specific, the **MyCoordinate and MyObserver** classes contain implementations for just the necessities of the upper-level interfaces. The **MyCoordinate** class contains *equals* and *hashCode* methods for ease of use comparing objects as well as storing objects in map data structures. Similarly, the **MyPiece** class contains implementations for the methods defined in the **EscapePiece** interface. However, along with these methods, this class contains a reference to the **PieceTypeDescriptor** that accompanies the given piece. By doing so, instances of **MyPiece** have information on their movement pattern as well as their attributes. This allows **MyPiece** to virtually be a one-to-one representation of what a piece in a realistic simulation of the game would look like.

Moreover, higher-level components such as **MyMove, MyLocation, and MyBoard** are found in this package. These higher-level components are built upon the lower-level components. The **MyLocation** class brings together several aspects of the game. In specific, an instance of **MyLocation** not only contains information about that location's rules, such as *BLOCK* and *EXIT* locations but also contains information about the coordinate of the location along with the current piece on the location at any given time. This means that **MyLocation** contains all the information needed to determine whether another piece can pass over itself along with acting as a reference for a real location on the board. **MyBoard** is essentially a hash-map with **MyCoordinate** keys accompanied by **MyLocation** values along with information on the bounds of the board. Using the map as well as the bounds of the board, instances of this class can dictate whether a given coordinate is in the bounds of the board. I incorporated this functionality into the main *getLocation* method of **MyBoard**. Essentially, if an existing location is in the map for a given coordinate, that location will be returned. However, if a matching location does not exist in the map, the method will then check if the queried coordinate should exist in the bounds of the game. If it should, the method will return a new location. Otherwise, *null* will be returned meaning the coordinate given to query was invalid. **MyMove** is simply a class that has a reference to a location in the board as well as the given movement direction of other moves that would have come before. This class is used for pathfinding specifically for pieces with linear patterns. Essentially, pathfinding is done via an increasing list of **MyMove** instances. With linear patterns, the linear direction (vertical, horizontal, right diagonal, left diagonal) would be stored in the movement direction field. This allows pathfinding to generate a list of valid neighbors for a given location based not only on coordinate but also on the directions of moves that preceded.



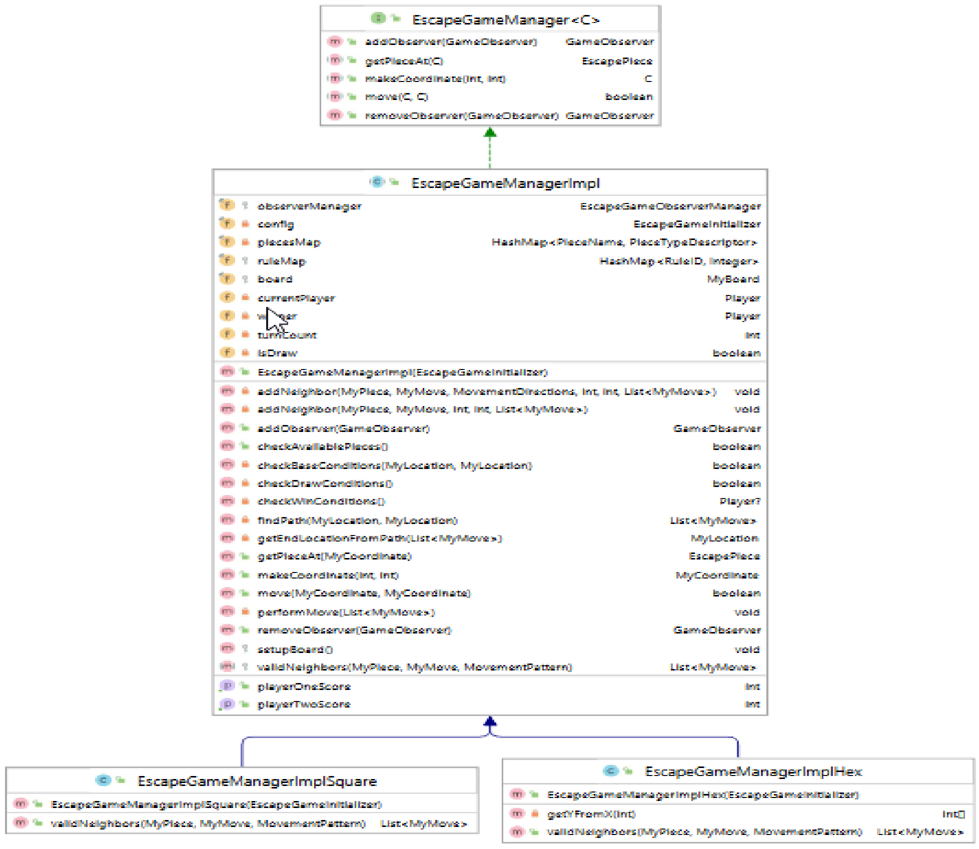

MyLocation
canMoveOver(MyPiece, HashMap<RuleID, Integer>) boolean
equals(Object) boolean
getCoordinate() MyCoordinate
getLocationType() LocationType
getPiece() MyPiece
setPiece(MyPiece) void


MyBoard
addPair(MyCoordinate, MyLocation) void
getLocation(MyCoordinate) MyLocation
hasAvailablePieces(Player) boolean
inBounds(MyCoordinate) boolean

This package contains most of the design decisions I made with this project. By having these six different classes, I feel as though I successfully abstracted different aspects of operating the game to each of the classes logically.

Manager Package

Several different managers live in this package. For one, **EscapeGameManagerImpl** lives here which is an extension of the provided **EscapeGameManager** interface. This class provides definitions for all the methods of the upper-level interfaces' methods. Specifically, *addObserver* and *removeObserver* are forwarded to the same methods defined in the **EscapeGameObserverManager**. This class is quite simple in nature. It contains a list of active observers for the game. When *notify* should be invoked, it is the job of the manager to iterate through this list and notify all active observers. I decided to abstract this observer functionality into a separate class to reduce clutter in the **EscapeGameManagerImpl** class which is already quite large. The implementation of the manager contains methods to check if any draw/win conditions have been met, check if the basic requirements for a move from one location to another have been met, along with other logic that is not specific based on-board type (square or hex). However, this class has an abstract *validNeighbors* method that is defined in both **EscapeGameManagerImplSquare and EscapeGameManagerImplHex**. This method is responsible for generating a list of **MyMove** instances based on a given location. This design decision allows all the shared logic between boards to live in one class while abstracting the only different logic between different boards to separate classes. While I believe this decision makes it quite easy to add new board variations very clear, I would have liked to move a lot of the logic of the big **EscapeGameManagerImpl** class to separate classes. For example, the movement logic could live in a **MovementController**, and the scoring logic could live in a **ScoreController**. Had I introduced these two classes along with other possible classes, the implementation of the bigger manager class would have been much cleaner and easier to read.


EscapeGameManager<C>
addObserver(GameObserver) GameObserver
getPieceAt(C) EscapePiece
makeCoordinate(int, int) C
move(C, C) boolean
removeObserver(GameObserver) GameObserver
EscapeGameManagerImpl
observerManager EscapeGameObserverManager
config EscapeGameInitializer
piecesMap HashMap<PieceName, PieceTypeDescriptor>
ruleMap HashMap<RuleID, Integer>
board MyBoard
currentPlayer Player
winner Player
turnCount int
isDraw boolean
EscapeGameManagerImpl(EscapeGameInitializer)
addNeighbor(MyPiece, MyMove, MovementDirections, int, int, List<MyMove>) void
addNeighbor(MyPiece, MyMove, int, int, List<MyMove>) void
addObserver(GameObserver) GameObserver
checkAvailablePieces() boolean
checkBaseConditions(MyLocation, MyLocation) boolean
checkDrawConditions() boolean
checkWinConditions() Player?
findPath(MyLocation, MyLocation) List<MyMove>
getEndLocationFromPath(List<MyMove>) MyLocation
getPieceAt(MyCoordinate) EscapePiece
makeCoordinate(int, int) MyCoordinate
move(MyCoordinate, MyCoordinate) boolean
performMove(List<MyMove>) void
removeObserver(GameObserver) GameObserver
setupBoard() void
validNeighbors(MyPiece, MyMove, MovementPattern) List<MyMove>
playerOneScore int
playerTwoScore int
EscapeGameManagerImplSquare
EscapeGameManagerImplSquare(EscapeGameInitializer)
validNeighbors(MyPiece, MyMove, MovementPattern) List<MyMove>
EscapeGameManagerImplHex
EscapeGameManagerImplHex(EscapeGameInitializer)
getYFromX(int) int[]
validNeighbors(MyPiece, MyMove, MovementPattern) List<MyMove>

Reflection on the course

Overall, in this course, I do not feel as though I learned a whole bunch of new material but rather, reinforced good practices that I have previously been taught. However, going through the pattern module provided me with good insight on how to structure my projects in the future. Specifically, I enjoyed the MVC controller pattern section as it is very relevant to web design, which is an aspect of Computer Science, I would like to further familiarize myself with.

I do feel as though I gained a lot from learning how you structure your tests. I especially enjoyed learning how to utilize parametrized tests and how they can clean up code quite nicely. I also enjoyed strengthening my skills in TDD by acquiring new tactics such as keeping a *TODO* file that can nicely illustrate the current tests I have developed along with what tests I still need to develop.