Object-Oriented Analysis and Design

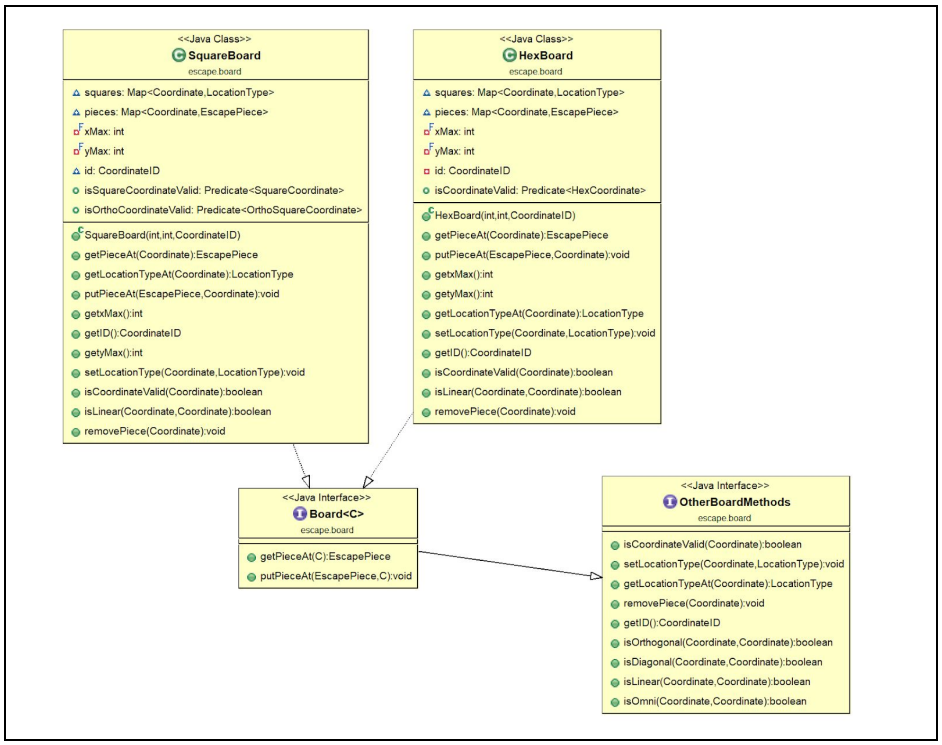
D20

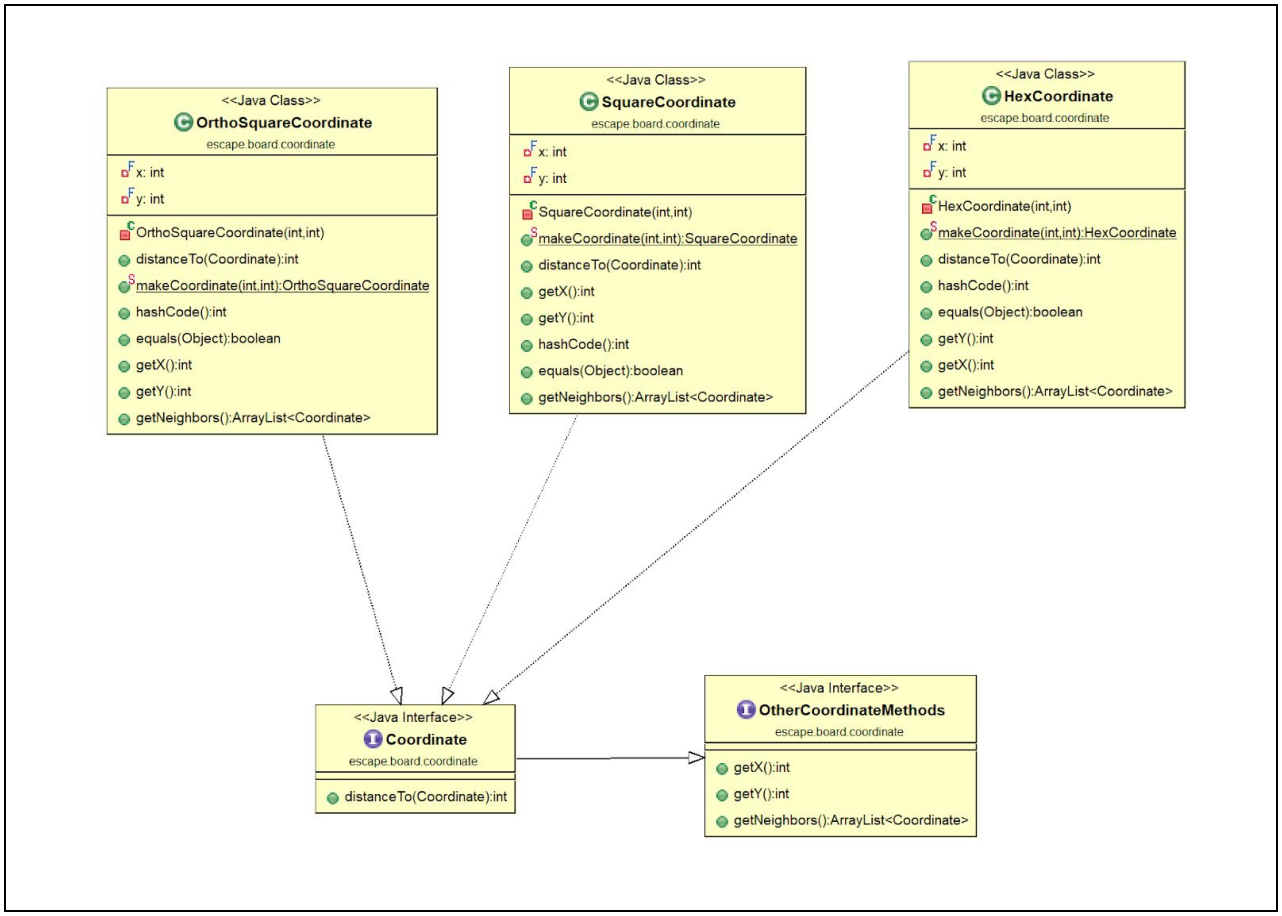
Escape

Kenneth Desrosiers

Alpha

For the Alpha release of Escape, I decided to keep the layout of the example SquareBoard file and use it for my HexBoard as well. I chose not to create a separate OrthoSquare board because it was not necessary. In these classes, I used functional interfaces to determine whether a coordinate was valid to the board because they are cleaner and need less lines of code. When starting this project, I also decided to keep the SquareCoordinate file; it uses hash maps to organize its pieces and coordinates, which is the most efficient way to quickly store and access that information. My OrthoSquareCoordinate and HexCoordinate classes are based on this file. There were a lot of methods that the boards shared, as well as methods that the coordinates shared, so I created two interfaces for additional board methods as well as additional coordinate methods. Below are UML diagrams for the structure of Boards and Coordinates.





For creating boards, I use a factory class. I realized that later on in the development of the game, the only objects the client should be able to create are observers, the game manager, and the game builder. Even within the code, factory methods make it easier to create new objects without having direct access to the desired class for creation.

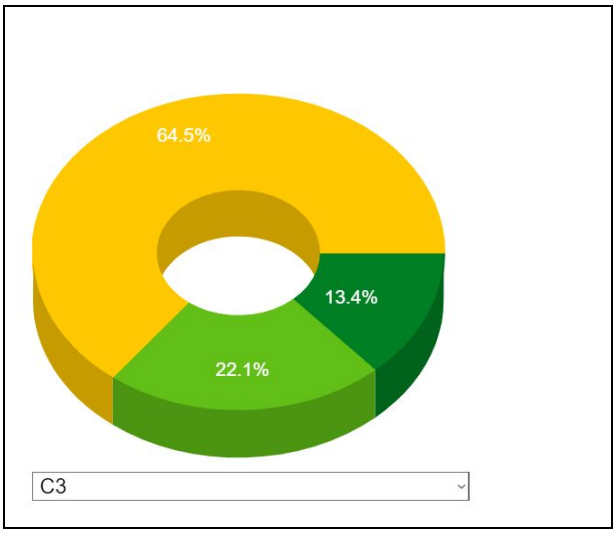
Beta

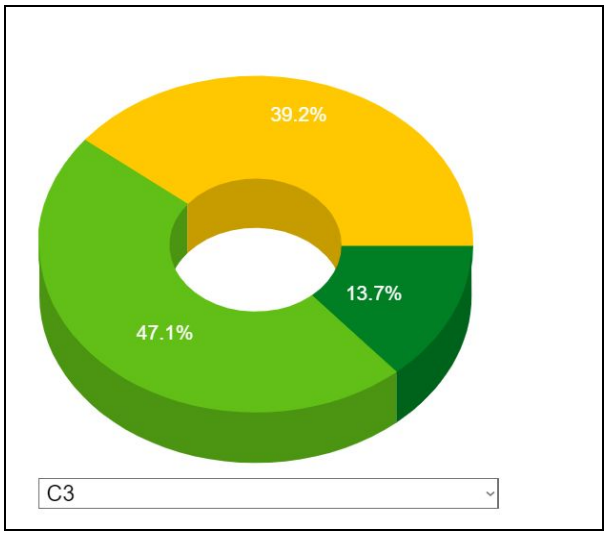
For the Beta release, I deleted the BoardBuilder and BoardInitializer classes because they were no longer needed with the introduction of the EscapeGameIntializer and EscapeGameBuilder classes. Similar to the board factory, I created a game manager factory for cleaner object

creation. I kept the board factory class so the game manager factory would not have more responsibility than it needs. Game managers have a board and a hashmap for storing each piece type's rules (map<PieceName, PieceType>). PieceType is a class I created that has a MovementPatternID field and a map<PieceAttributeID, PieceAttribute> field. When organizing data, I use hashmaps a lot because of their efficiency. They are the most efficient (timewise) data structure for organizing classes in this project. For the movement of the pieces, I created a MovementRule class to encapsulate the responsibility of checking if neighbor coordinates are valid according to the piece's specific movement pattern. This class was used in the Dijkstra class I created for pathfinding. The Dijkstra pathfinding worked, but I quickly realized that the obstacles made the algorithm stray far from the goal sometimes. When the graph was large, the runtime was very slow. I changed my pathfinding to use A* instead. This pathfinding algorithm uses heuristics to guide the search closer to the goal without getting lead away.

Final

For the final release, I spent the most time refactoring my code to make it cleaner. For example, in my AStar class, I changed the responsibility of finding viable neighbors to be the responsibility of the Coordinate classes. Some of my classes had issues with high coupling because of relying on other classes such as the BoardFactory and GameFactory. In order to combat this, I created an interface for the factory classes to decrease the coupling. The following image shows graphs that represent the level of cohesion, coupling, and complexity before and after I refactored my code. The color range goes from green to red (good to bad).


64.5%
13.4%
22.1%
C3


39.2%
13.7%
47.1%
C3