Design Manual

Introduction:

We used a client-server protocol and the MVC structure in order to create the board game Monopoly. In the game, each player who joins the game is a client and has their own model, view, and controller. Each view and controller have their respective sub-views and sub-controllers that perform specified functions. The model contains a game object and the list of Character objects in the game. The game object contains all of the artifacts of the game and compiles them to together to allow the game to run.

User stories:

Finished:

Do research to on what we want the board to look like
Create 40 squares in a larger square representing the board with colors and names
Create a UML diagram
Create a character on the board that can move around the board
Create all the properties with their attributes (color, rent, name, etc.)
Create the Railroad spaces
Create the Utility spaces
Create the Community Chest and Chance spaces
Create the tax spaces
Create the corner spaces
I want to be able to roll and move my character the correct number of spaces
I want to be able to buy properties that are unowned when I land on them
I want to collect $200 when I pass Go
I want to pay rent if I land on a property that is owned by someone else
I want to draw a Community Chest/Chance card when I land on the space
I want to perform the action described on the Community Chest/Chance card
I want to choose my player's name and character
I want to go to Jail when I am supposed to
I want to be able to roll again if I roll doubles
I want to go to Jail if I roll 3 doubles in one turn
I want to have to roll doubles or wait 3 turns and pay $50 to leave Jail
I want to be able to see my character move on the board while I play

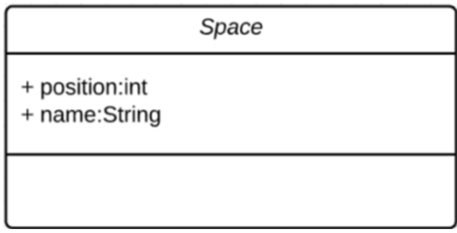
Unfinished

I want to be able to build on my Monopolies
I want to be able to mortgage if I run out of money
I want the game to end once all players except 1 go bankrupt
I want to be able to choose an image to represent my player
I want to be able to play with CPU's with different difficulties
I want to be able to play a game with simple computers
I want to be able to use Get Out of Jail Free Cards if I choose
I want to be able to see my money
I want to be able to see all the properties I own
I want to be able to choose a max number of turns in a game

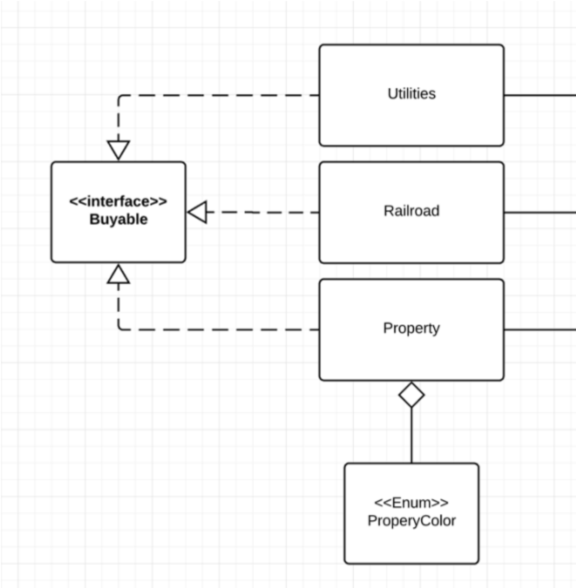
Object Oriented Design:

For this project, we stuck to strict object-oriented design in order to create classes that had a high degree of cohesion and low degree of coupling. To start, we chose to focus on implementing and designing the basic artifacts that make up a Monopoly game.

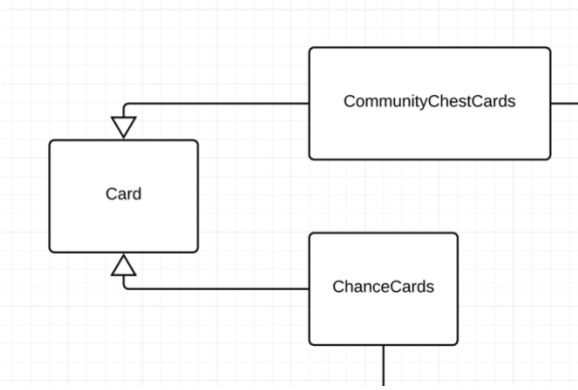
First, we will start with the spaces on the board which are all contained within csci205FinalProject/csci205FinalProject/src/Game/Spaces. We determined that the best approach to handle the board was to break each square on the board into its simplest form. Thus, we created an abstract class called Space that contained the most basic info possible for each square on a Monopoly board.

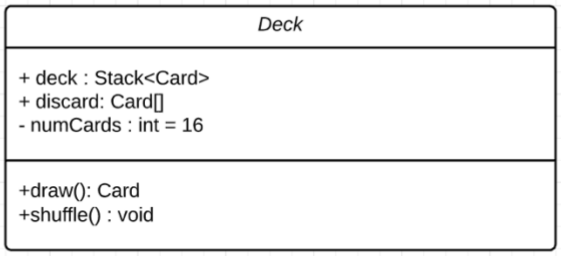


We then wanted to abstract further as there is a clear difference between spaces you can purchase, properties, railroads, and utilities, and spaces that you cannot, jail, free parking, community chest, etc. Thus, we decided to create an interface called *Buyable*. We also decided to break down each unique type of space into its own class and had them implement Buyable if they were purchasable. Therefore, we created the Utilities, Railroad, and Property classes that all implemented Buyable and extended Space. In addition, each Property contained a PropertyColor which was an enum representing the color of the property. We chose to make this an enum because the colors of the properties are set for every Monopoly game and therefore are constants.



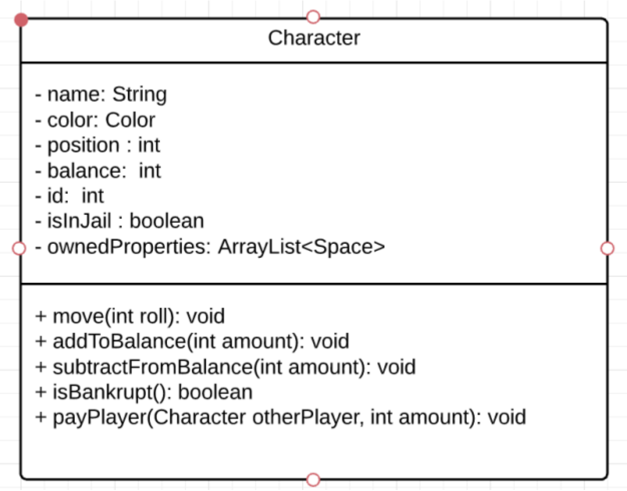
The next item we created was the Community Chest and Chance cards. This is all contained within csci205FinalProject/csci205FinalProject/src/Game/Cards. After some discussion, we came to the conclusion that there was no real difference between Community Chest and Chance cards. Thus, we implemented, and class called Card which both CommunityChestCard and ChanceCard extended. We then had card handle the action described by the card as well by creating an enum CardType. CardType contained constants that represented all the different actions that could be performed by a card such as a bank transaction and player transaction. Next, we wanted to create decks for each of the types of cards in



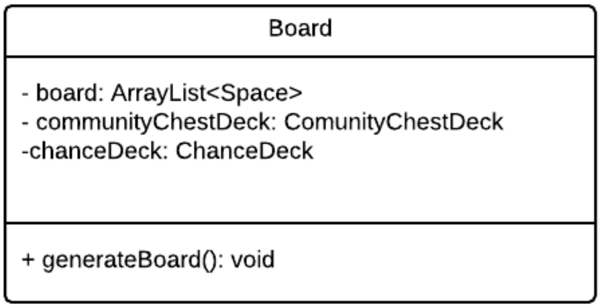


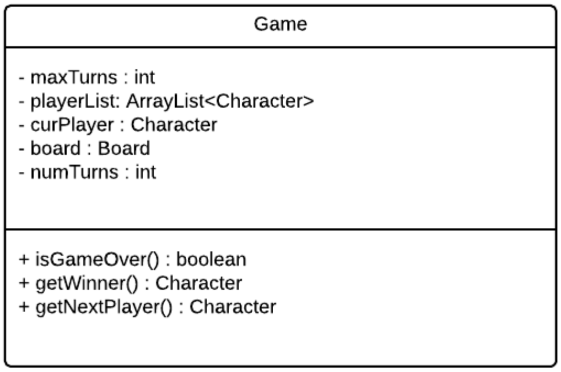
order to contain them within the game. Thus, we created an abstract class called Deck that created the basic outline for a deck of cards of any type. Deck contained two data types, a stack and an array, that maintained it and allowed the ability to draw and shuffle the deck. CommunityChestDeck and ChanceDeck both extend Deck and fills the deck with cards of their respective types.

The last main components that were needed to create the structure of the game was the Character class. The Character class is a representation of a player that contains all of the necessary components a player needs such as balance, name, etc. Each player also has a unique id which is used to determine which properties they own later in the Board class.

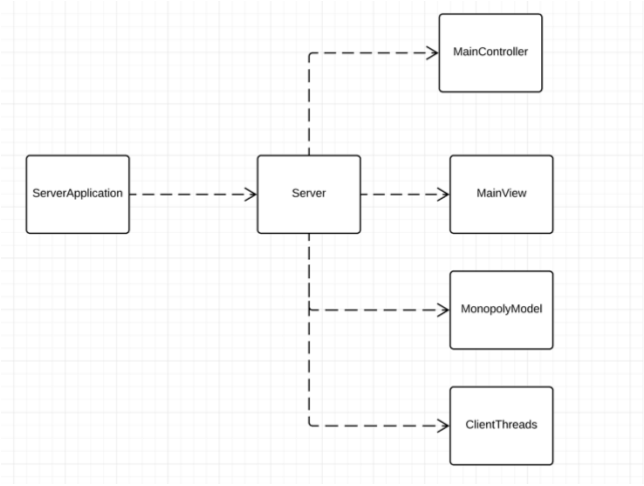

Character
- name: String
- color: Color
- position : int
- balance: int
- id: int
- isInJail : boolean
- ownedProperties: ArrayList<Space>
+ move(int roll): void
+ addToBalance(int amount): void
+ subtractFromBalance(int amount): void
+ isBankrupt(): boolean
+ payPlayer(Character otherPlayer, int amount): void

In order to aggregate all of these components into the game, the Board class was created. The Board class contains an ArrayList of all the spaces on the board and the two different card decks. In essence, the board serves as a data structure that exist solely to hold all of the parts of the game for the Game class. The Game class is the actual engine that runs a given game as it contains a single Board and a list of all the players in the game. The Game controls who the current player is and rotates through the players when they end their turn.

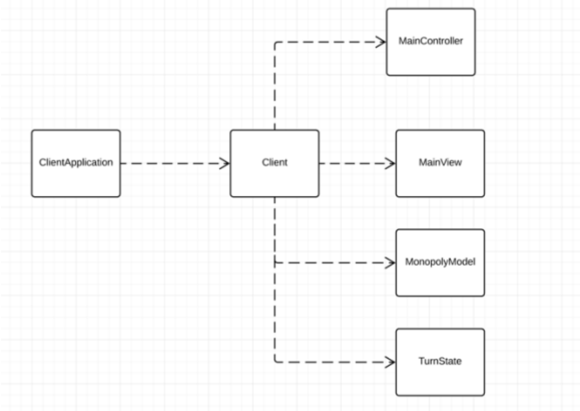

Board
- board: ArrayList<Space>
- communityChestDeck: ComunityChestDeck
-chanceDeck: ChanceDeck
+ generateBoard(): void



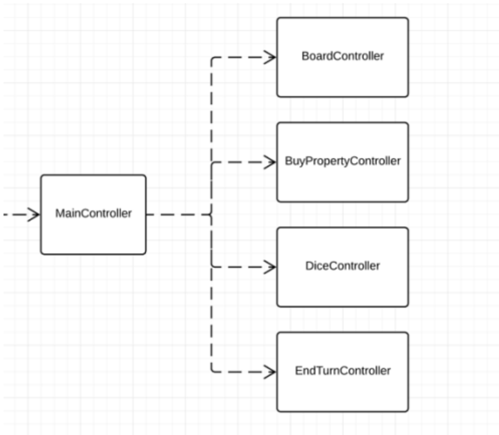
With all of the structures necessary to create a game, we can now look at the structure for the networking. We used a client-server set up where a server is opened, and each client can join the server socket that is created. The host runs ServerApplication which creates an instance of the Server class. As each player connects, the Server creates a ClientThread object that “links” the client to the server. This is used as the means of communication between the two during the entire protocol of the game. Once the game is ready to start, the server creates an instance of a model, view, and controller. Later on, we will go into the inner workings of the orientation of these three classes.

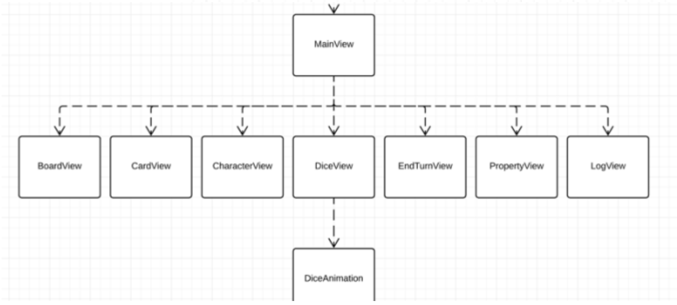


On the other end, the players attempting to play the game run the main method in the ClientApplication which in turn creates a Client object that connects to the Server via a ClientThread. Again, once the game is ready to begin, the client creates their own instances of a model, view, and controller. The model is the object that is sent back and forth over the InputStreams and OutputStreams to allow the game to update for all the clients and the server. In addition to the model, view, and controller, the client also has an instance of TurnState object that is an enum that controls whether it is that given player turn or not.



The most unique object-oriented design we used within the entire project was with the MVC component. We decided to delegate the tasks of the view and controller classes in numerous subclasses that each handled one specific function. We set this up in the constructor of the view and controller which created and maintained the sub-views and sub-controllers respectfully. As shown in the UML diagram to the right, we believed this was better because it made the classes more specific to a single task rather than having one class handle multiple functions. In the end this definitely helped us when debugging because we knew that a specific issue had to be occurring in the view or controller that was assigned to that task.


MainController
BoardController
BuyPropertyController
DiceController
EndTurnController


MainView
BoardView
CardView
CharacterView
DiceView
EndTurnView
PropertyView
LogView
DiceAnimation