Joseph DiPalma, Annan Miao, and Ben Xu

May 2, 2017

Professor Brian King

CSCI 205 Final Project

Design Manual

Introduction:

Our project consists of the GUI, the non GUI functionality and the networking. The GUI we designed follows the MVC design pattern. The non GUI functionality consists of all of the objects in the physical version of Battleship. The networking creates a server and client and initializes a connection between them. Our code follows good object-oriented design practices including minimizing coupling and maximizing cohesion. We tried to incorporate as many of the design patterns as possible.

User Stories:

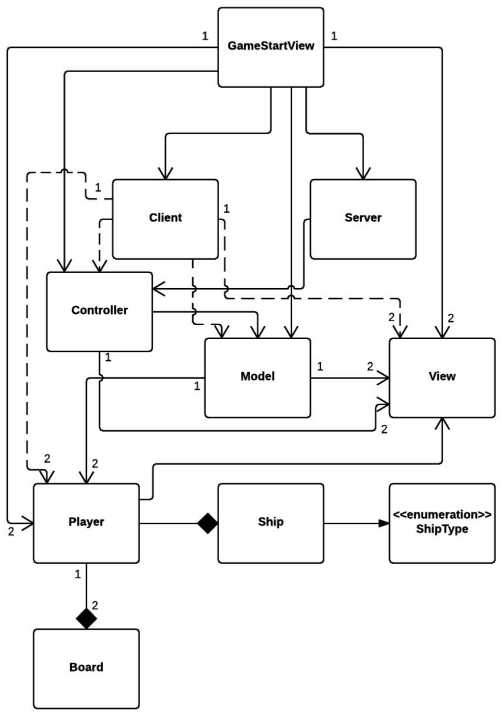
1. *As a user I want to be able to place my ships on my board so that I can start the game.*
    i. Relevant classes:
        - Player
        - Board
        - View
        - Ship

    ii. Each Player object has a Board object and an array of Ship objects. Each Ship object has an attribute of enumerated type ShipType that represents the type of the ship. The location of the Ship object on the Board is held within an ArrayList containing arrays of length 2 representing ordered pairs. The ordered pairs correspond to a coordinate on the Board. The View class displays all of this visually.

2. *As a user I want to be able to view where my opponent has attacked my ships so that I can make my next move.*
    i. Relevant classes:
        - Player
        - Board
        - View
    ii. Each Player object has a Board object that keeps track of enemy attacks. The View class displays the attacks visually.
3. *As a user I want a view of the hits and misses I have made on my opponent's board so that I can plan where to make my next move.*
    i. Relevant classes:
        - Player
        - Board
        - View
    ii. Each Player object has a Board object that keeps track of where they have attacked their enemy. Each attack is represented by a red square if it is a hit and a white square if it is a miss. The View class displays the red and white squares for the Player.
4. *As a user I want to be able to connect to the network so I can play with another person on a different computer.*
    i. Relevant classes:
        - Server
        - Client
        - GameStartView
    ii. The Server object hosts the game and displays its IP address for the other user. It waits for the Client object to connect. The Client object represents the other user. This interaction is displayed graphically by the GameStartView class.

5. *As a user I want to be able to use the game without lag so it is an enjoyable experience.*
    i. Relevant classes:
        - All classes
    ii. All of the tasks are separated into different threads. The Server, Client, and GUI objects are all separated into their own threads to ensure that no lag occurs.

Object-Oriented Design:

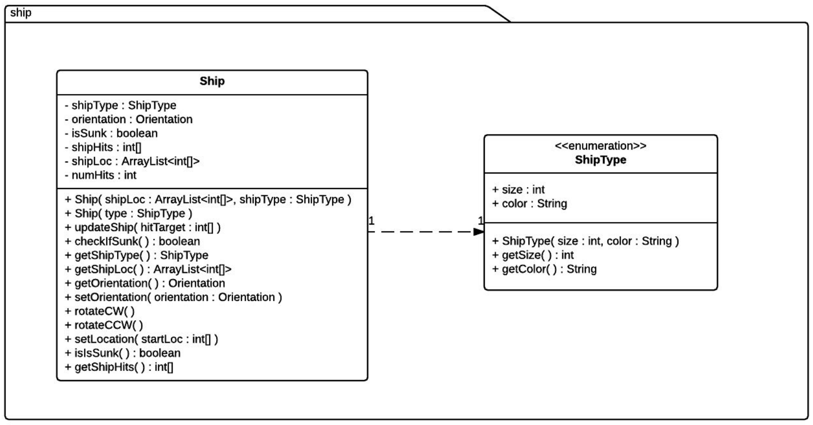


Ship	
Has location in board	Board
Has size	Player
Has hit locations	ShipType
Knows if sunk	
Knows orientation (vertical or horizontal)	

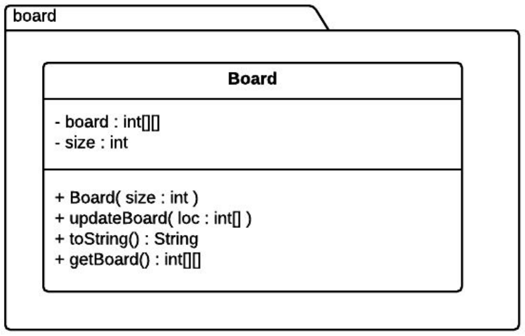
Player	
Contains ships (knows types and sizes)	Board
Knows the total number of ships	Ship
Has my board	
Has opponent board	
Knows how many ships remain	
Places ships in locations	

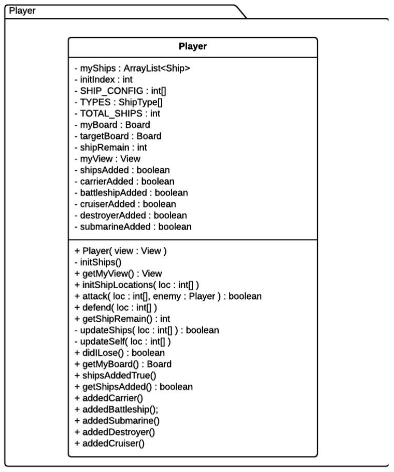
Board	
Has size	Ship
Updates after attack	Player

For our design we broke down everything into individual objects that represent the physical objects in the game Battleship. We started off by listing the physical "objects" in the game in order for the game to function as a general concept. The "objects" needed were Board, Ship, and Player.


ship
Ship
- shipType : ShipType
- orientation : Orientation
- isSunk : boolean
- shipHits : int[]
- shipLoc : ArrayList<int[]>
- numHits : int
+ Ship(shipLoc : ArrayList<int[]>, shipType : ShipType)
+ Ship(type : ShipType)
+ updateShip(hitTarget : int[])
+ checkIfSunk( ) : boolean
+ getShipType( ) : ShipType
+ getShipLoc( ) : ArrayList<int[]>
+ getOrientation( ) : Orientation
+ setOrientation(orientation : Orientation)
+ rotateCW( )
+ rotateCCW( )
+ setLocation(startLoc : int[])
+ isIsSunk( ) : boolean
+ getShipHits( ) : int[]
1
1
<<enumeration>>
ShipType
+ size : int
+ color : String
+ ShipType(size : int, color : String)
+ getSize( ) : int
+ getColor( ) : String

Next we considered how each of these objects should interact using the User Stories we created. Since we wanted to be able to keep track of each player's attacks, we needed each Player to have two Board objects. We also needed to keep track of the ships so each Player contains an array of Ship objects. Each Ship object has an attribute that is of the enumerated type ShipType that indicates the type of the Ship object (carrier, cruiser, submarine, destroyer, battleship).





All of these objects were enough to build the basic Battleship game. We then used the MVC design pattern to create the graphical representation of the game. In order to ensure proper synchronization between the players, we decided to have one Model object that connected both players. This made sure each player had a consistent experience. Hence a Model object takes in two View objects and two Player objects. Then the Model object is passed into a Controller object which handles all of the functionality of the game. The Controller object is then passes to the Server and Client objects so the game can be run over the network.

We followed the Server/Client model using the Socket API for the networking portion of the project. None of us had any prior networking experience. We decided on the Socket API because it was the only one that made sense when reading through the documentation. It may not have been the best option but we were able to get it mostly working. The ideas for networking are as follows:

1. User 1 hosts the game and displays the IP address of their computer (Server).
2. User 2 obtains the Server's IP address and then enters it into a text box to connect to User 1's Server.
3. Once the connection is successful, both users see their respective views and the game begins.

In order to ensure a smooth gameplay experience, we incorporated multithreading into the design. Since Battleship is a turn based game we made the game interactions run within the JavaFX thread. This meant that the game ran alongside the GUI while the Server and Client ran within their own thread.