# Module 2 Programming Project Report
# Abalone Board Game

Authors
Ivo Broekhof s2185970
Kevin Singpurwala s2205858

February 2, 2020

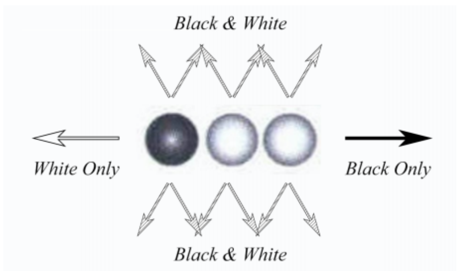
## Contents

## 1 Overview

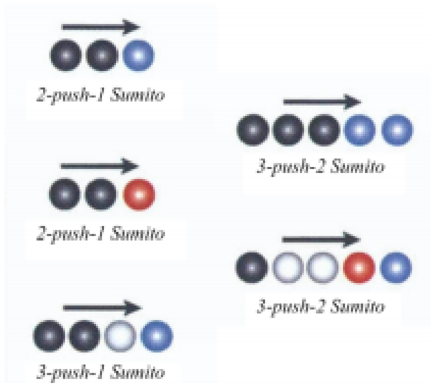
We are required to design and implement a functional Server Client board game called Abalone. We followed a model view controller architecture.

### 1.1 The board Game

The board consists of 61 circular spaces arranged in a hexagon . In a two player game each player has 14 marbles that rest within the spaces.In different game modes the number of marbles each player has changes but the logic is the same for a two and three player game modes.The rules differ slightly in a four player game as you have a partner. The players take turns with the player with the black pieces moving first. Then blacks opponent(s) move once each. We now loop in the same format until the game ends.
Eg. Black → Red → White → Blue → Black.
In this case we loop back to black and continue in the same format. Note that in the four player game one of the other players is your ally. The moving format must be TeamONE → TeamTwo → TeamONE → TeamTwo.
For each move,a player moves a straight line of one, two or three marbles of one color one space in one of six directions. A player can push their opponent's marbles back one square(a "sumito") in the same direction they moved their marbles if the marbles they moved are greater than the opponents marble(s) one pushed. As seen in the project description.



(a) In a 4 player game possible legal moves directions are slightly altered



(b) Possible sumito positions

One can only push if the pushing line has more marbles than the pushed line (three can push one or two; two can push one. Marbles must be pushed to an empty space or off the board.In the case that you push two marbles one of them moves to an empty square or off the board whilst the other one moves back one square. [1]

### 1.2 The Networking

The server first opens up. A client is now capable of making a connection via Server.java. The client then initiates a connection to the server via Client.java. The client will then we prompted to enter the hosts name to which they wish to connect to. If they enter the right host name and both are connected to an appropriate port.(in this case port 69 as agreed upon in the protocol a connection will be made. Until such time the server will be awaiting for a client to join. The client and server can now communicate to each other.

Multiple clients can connect to one server. Thus threads are required. Each client has a nickname by which they will be known to the server. Once a client has connected to the server and received a name it can choose a game mode. The range of choices include and are limited to :

- 2 player game

- 3 player game
- 4 player game

The client can be either a Human player or an AI player.The client has a Program argument that is either "r" or "h" to decide if it will be randomAI or Human. Once the correct number of Players have connected for the game mode selected by the first player that joined a game will begin.

The clients will each take turns and the server will send the updated state of the board to each client.

## 2 Description Of Each Class

### 2.1 Network Package

#### 2.1.1 Server package

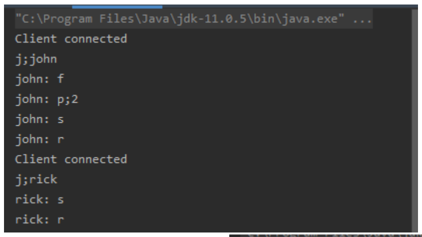
Firstly, we have a file named Server.java which implements runnable.

- Map<Player, Boolean> getPlayers() returns a map of players and their ready values.
- Set<Player> getPlayerSet() return the set of players.
- synchronized void addPlayer(Player player) requires a player object and adds them to the player map. This method is synchronized to prevent concurrency thread issues.
- int getMaxPlayers() returns the maximum players.
- setReadyPlayer(Player player) sets a player to the ready state in accordance with the Prtocol Messages. [3]
- setMaxPlayers(int maxPlayers) can set the maximum allowed number of players to a different integer value.
- synchronized void setMove(Move move) requires a move object as a parameter. It is used in the ConnectionHandler.java class. This method is synchronized as only one player can set a move at a time. It sets the move inputted as a parameter as the move we want to make.
- Game getGame() returns a game object. This method is vital as for a game to start we first need to know which game to start. A game object. Look at the description of then controller package for more info on Game.java.
- startGAme() starts a game on the server. We first create a new game object which has an arrayList of players. Then we create a map for each player to a connectionhandler. Now for each client we add a PlayerKey and a ConnectionHandler value.We now use the makeMove() method described above. Now we perform a while loop1 until the game is over. We start the game and the "current" connectionhandler object tells us which players turn it currently is. We now enter a while true loop which calls the method Game.move(Move move). Our input parameter here will be the move we want to make. If the move is legal we send broadcast the protocol.Valid message to all clients. If the move we want to make is illegal or an old move we throw an IllegalMoveException. After we will exit the whie loop1 as the game is finished. We will send each client the name of the winner or draw as appropriate.
- The main method creates a server object. The server is threaded and calls an inbuilt java class Thread.java -synchronized void start().
- broadcast(String message) broadcasts(sends) the inputted string to each client that is connected. This is done via the connectionHandler class.
- If a player disconnects the disconnect(Player player) is called. This uses the broadcast(String message) method to send the ProtocolMessages.DISCONNECT_WINNER message and the name of the winner.

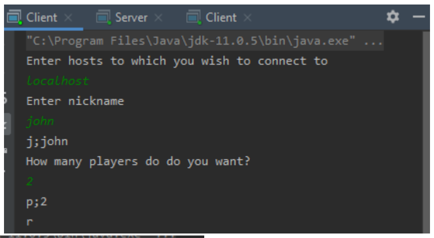
- The run() method is overwritten. A while true loop then begins. Within this loop the server creates a serverSocket to which a client can later connect to. Then it timesout which gives a chance for the client to connect via serverSocket.accept(). When a client connects we create a new ConnectionHandler object which is threaded and is state via thread.start().

  Now we move on to the ConnectionHandler.java class

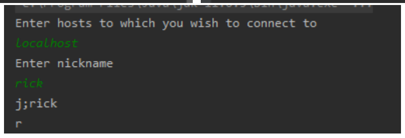
  - This class handles server to client communication. It implements runnable and has BufferedReader which reads incoming messages from client $\rightarrow$ server and PrintWriter which sends outgoing messages from server $\rightarrow$ client.
  - We have a constructor in Connectionhandler with parameters ConnectionHandler(BufferedReader in, PrintWriter out, Server server)
  - getPlayers() returns the player of that connectionhandler object.
  - The handleMessage(String message) throws an ArgumentException for malformed arguments and a CommandException for malformed commands. This method processes messages reveived from the client and performs the appropriate actions in accordance with the message protocol.[3].It splits the message according to the Protocol.Delimiter which is used to separate the message.
  - j(String[] args) handles the protocol.Join messages recieved. If the message starts with for example. j;john then the player with nickname john wishes to join a game. If he is the first to join john will be asked to select how many players he wants and thus the game mode and we send a ProtocolMessages.FIRST_PLAYER. If you are not the first player to join the server will send a ProtocolMessages.SUCCESS message in accordance with the protocol. [3] The server prints an f only for john as he is the first to join.



```
"C:\Program Files\Java\jdk-11.0.5\bin\java.exe" ...
Client connected
j;john
john: f
john: p;2
john: s
john: r
Client connected
j;rick
rick: s
rick: r
```



```
"C:\Program Files\Java\jdk-11.0.5\bin\java.exe" ...
Enter hosts to which you wish to connect to
localhost
Enter nickname
john
j;john
How many players do do you want?
2
p;2
r
```



```
Enter hosts to which you wish to connect to
localhost
Enter nickname
rick
j;rick
r
```

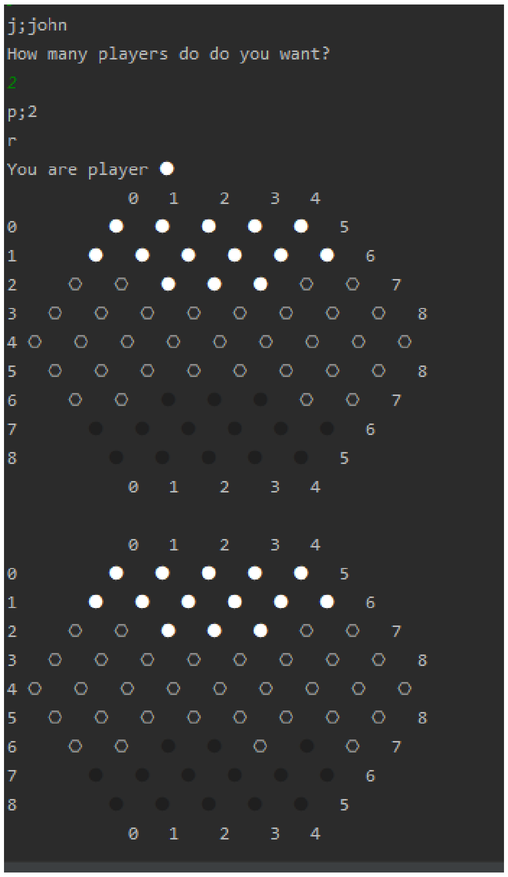
### 2.1.2 Client package

As for the Client side of the connection, it is handled by Client.java.

- Client(InetAddress serverAddress) creates a new client. The parameter serverAddress specifies the address of the server(host) to connect to.
- The main method of the Client, starts a Client object and handles the client-side networking.We initiate the InetAddress ip as null as currently. The String host we want to connect to is also initialised as "". While the ip is null we get the InetAddress via ip= InetAddress.getByName(nameHost); We ask the client to input the host name using a buffered reader and print writer. Then we create a new client object(cl). Now we start a try catch block. cl = new Client(InetAddress.getLocalHost());
  This client gets its own view via cl.ui = new Tui(); This client obect then selects a nickname if one has yet to be assigned. If there is an incoming message sent from the server to the client the client handles that message via the method handleMessage(String message). This is the end of the try and we catch exceptions. It is also the end of the while.
- handleMessage(String message) handles incoming messages sent from the server by splitting them up with the delimiter. We do this via splitMessage and take the first position. Then

decrement the length of splitMessage. parameter args the arguments that come with the message.

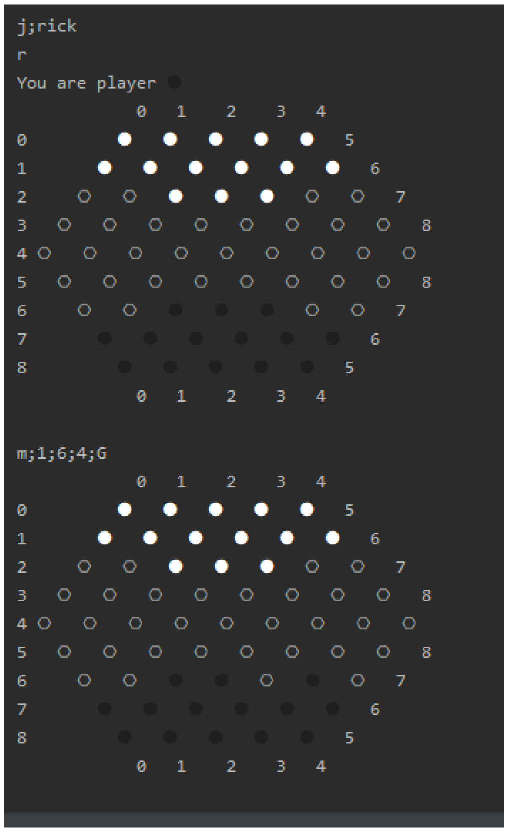
* The f(String[] args) method. If the client receives an f they are the first to join and they will be prompted via a scanner object to input the number of players they want(player amount). We send ProtocolMessages.PLAYER_AMOUNT + ProtocolMessages.DELIMITER + playerAmount to the server.
* If the client receives an s the s(String[] args) method is called. We know that a successful connection is made. We also send a ProtocolMessages.READY alongside this informing that the player is ready to start the game.
* g(String[] args) handles ProtocolMessages.NUM_OF_PLAYERS. Gets the number of players and starts the game when teh number of players required is reached. It gets whether the player is human or ai, prints the tui and tells the client what color marble they are and whos turn it is.



```
j;john
How many players do do you want?
2
p;2
r
You are player ⚪
            0  1  2  3  4
0           ⚪ ⚪ ⚪ ⚪ ⚪  5
1         ⚪ ⚪ ⚪ ⚪ ⚪ ⚪  6
2       ○ ○ ⚪ ⚪ ⚪ ○ ○  7
3     ○ ○ ○ ○ ○ ○ ○ ○  8
4   ○ ○ ○ ○ ○ ○ ○ ○ ○
5     ○ ○ ○ ○ ○ ○ ○ ○  8
6       ○ ○ ⚫ ⚫ ⚫ ○ ○  7
7         ⚫ ⚫ ⚫ ⚫ ⚫ ⚫  6
8           ⚫ ⚫ ⚫ ⚫ ⚫  5
            0  1  2  3  4

            0  1  2  3  4
0           ⚪ ⚪ ⚪ ⚪ ⚪  5
1         ⚪ ⚪ ⚪ ⚪ ⚪ ⚪  6
2       ○ ○ ⚪ ⚪ ⚪ ○ ○  7
3     ○ ○ ○ ○ ○ ○ ○ ○  8
4   ○ ○ ○ ○ ○ ○ ○ ○ ○
5     ○ ○ ○ ○ ○ ○ ○ ○  8
6       ○ ○ ⚫ ⚫ ○ ⚫ ○  7
7         ⚫ ⚫ ⚫ ⚫ ⚫ ⚫  6
8           ⚫ ⚫ ⚫ ⚫ ⚫  5
            0  1  2  3  4
```

(a) John is the white pieces

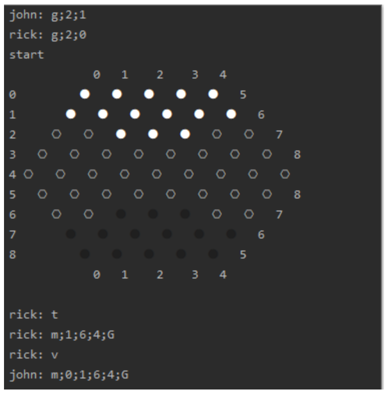


```
j;rick
r
You are player ⚫
            0  1  2  3  4
0           ⚪ ⚪ ⚪ ⚪ ⚪  5
1         ⚪ ⚪ ⚪ ⚪ ⚪ ⚪  6
2       ○ ○ ⚪ ⚪ ⚪ ○ ○  7
3     ○ ○ ○ ○ ○ ○ ○ ○  8
4   ○ ○ ○ ○ ○ ○ ○ ○ ○
5     ○ ○ ○ ○ ○ ○ ○ ○  8
6       ○ ○ ⚫ ⚫ ⚫ ○ ○  7
7         ⚫ ⚫ ⚫ ⚫ ⚫ ⚫  6
8           ⚫ ⚫ ⚫ ⚫ ⚫  5
            0  1  2  3  4

m;1;6;4;G
            0  1  2  3  4
0           ⚪ ⚪ ⚪ ⚪ ⚪  5
1         ⚪ ⚪ ⚪ ⚪ ⚪ ⚪  6
2       ○ ○ ⚪ ⚪ ⚪ ○ ○  7
3     ○ ○ ○ ○ ○ ○ ○ ○  8
4   ○ ○ ○ ○ ○ ○ ○ ○ ○
5     ○ ○ ○ ○ ○ ○ ○ ○  8
6       ○ ○ ⚫ ⚫ ○ ⚫ ○  7
7         ⚫ ⚫ ⚫ ⚫ ⚫ ⚫  6
8           ⚫ ⚫ ⚫ ⚫ ⚫  5
            0  1  2  3  4
```

(b) Rick is the black pieces and has just made his move

* t(String[] args) Handles ProtocolMessages.TURN and tells whos turn it is. This can be seen in the picture of the server below. When it is ricks turn the server sends t and also prints it to its terminal. The server received the move and ricks move was vaid so v was sent and printed to terminal. Below we see that rick moved 1 marble from position 6,4 in direction vector G. Thus the marble ends up at location 6,5. A move object is (number of marbles you want to move ; Row_from, Column_from(repeated x times for x marbles where $0 < x < 4$) , direction(G,V,C,D,R or T) relative to f key.



```
john: g;2;1
rick: g;2;0
start
          0   1   2   3   4
0        ●   ●   ●   ●   ●   5
1      ●   ●   ●   ●   ●   ●   6
2    ○   ○   ●   ●   ●   ○   ○   7
3  ○   ○   ○   ○   ○   ○   ○   ○   8
4 ○   ○   ○   ○   ○   ○   ○   ○   ○
5  ○   ○   ○   ○   ○   ○   ○   ○   8
6    ○   ○   •   •   •   ○   ○   7
7      •   •   •   •   •   •   6
8        •   •   •   •   •   5
          0   1   2   3   4


rick: t
rick: m;1;6;4;G
rick: v
john: m;0;1;6;4;G
```

∗ m(String[] args) Handles ProtocolMessages.MOVE. It does so by first taking the first agrument and making sure it is between 1 and 3. ie. The number of marbles selected is legal. We then check for a valid vector(direction in which to move). Theses are relative to the F key and are G,V,C,D,R or T. We check if a field is on the board. If it is and the direction is valid we continue. We get the string from which is the current location of the marbles we want to move and we get the "move" which is where we want to move the marbles to. The method game.move(move)If this is valid the move will be made .

∗ v(String[] args) handles the ProtocolMessages.VALID. Here we do nothing if valid.

∗ n(String[] args) handles ProtocolMessages.NOT_VALID. If the move is not valid we call ui.invalidMove().

∗ w(String[] args) handles ProtocolMessages.WINNER. If the turn limit is reached we call ui.draw(); Else we get the winning player.

∗ u(String[] args) Handles ProtocolMessages.UNEXPECTED_MESSAGE. Here we try to print args[0] and catch if index is out of bounds.

∗ synchronized void send(String message) sends a message to the other side of the socket. Client → Server

## 2.2 Model package

### 2.2.1 Objects package

This package entails all game objects, these being Board and Marble. As for the Board:

- Marble[][] state; the current state of the board.
- Marble getField(int, int) or Marble getField(String); returns the content of the board on the specified field.
- moveMarble(int, int, int[]) or moveMarble(String, int[]); moves a marble from the specified position in the direction specified with a vector.
- static boolean isField(int, int) or static boolean isField(String); checks if the specified coordinates refer to a valid field on the board.
- static boolean neighbors(int, int, int, int) or static boolean neighbors(String, String); checks if two fields specified are neighbors (reachable with one move)
- String toString(); Generates a String representation of the board for the TUI of the server or client to print.

And the Marble:

- static Marble[][] startState(int); returns the initial state of a board for a game with the specified amount of players.

- boolean isFriendly(Marble); checks if the specified Marble is on the same team (BlackWhite, BlueRed) in a 4-player game.
- String toString(); returns the ANSI color code associated with its color.

#### 2.2.2 Player package

This package entails the Move and Player classes, parts of the model that are related to the players of the game. As for the Move class:
All fields have a getter but no setter.

- int players; the amount of players in the game.
- Player player; the Player making the move.
- List<String> from; the position(s) the marble(s) get moved from.
- int[] vector; the direction of the move, in the form of a [row, column] vector.
- make(Board); performs the move on the specified Board.

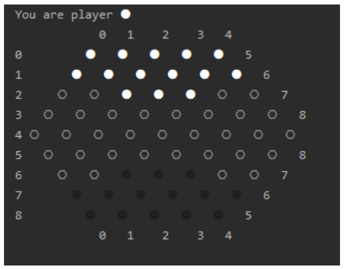
And the Player class:
All fields have a getter, but only color can also be set.

- int score; the current score of the player.
- Marble color; the color of Marble this player plays with.
- String name; the name of the player.
- boolean winner; whether this player is the winner of the game.
- incScore(); increments the score of the player.

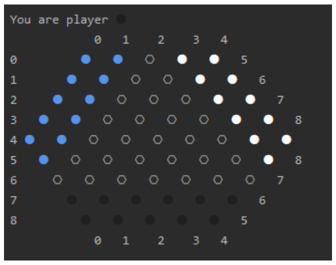
## 2.3 View package

### 2.3.1 TUI

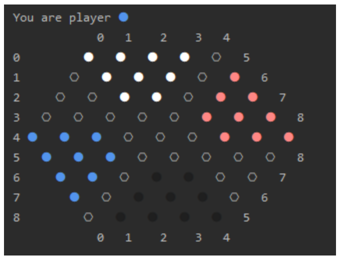
The Text User Interface(TUI) is fully functional on all game modes and is the current view of our client-server board game. This is as the GUI has yet to be integrated in to the client server and can only be run locally.The TUI view looks as such below.



2 Player



3 Player



```
You are player ●
```

4 Player

Once a game starts the client will be informed what color they are.
The Tui is an extension of the abstract class UI.java. The Tui implements each method in this subclass.

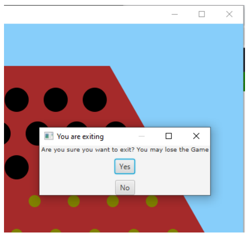
*Description of the methods in The Tui.java:*

- invalidMove() tells you if the move is valid or not. When the move is not valid a message is printed to the terminal/console.
- draw() tells you if the game has ended in a draw. This is due to an occurrence where 96 moves have been made and a winner has not been reached.
- winner() tells you if a winner has been made. This can occur if a player disconnects. In this case the player with the highest score that is not the player that disconnected wins. In a 4 player game the team of players who did not disconnect win.
- makeMove(Game game,Player player) This method has two input parameters. It requires knowledge of which game it is on and the player who is making the move. The amount of marbles which one moves must be greater than 0 and this is what the int marbleCount guarantees. Now we create an array list "from" which gives us knowledge of where the marble is currently located (row , column) format as shown by the notation beside the Tui board. This method returns a move object. A move object contains 1)the number of players, 2)a player object, 3) A List<> of where each marble one wants to move is coming from ,4)An int array vector which tells us in which direction one wants to move. This is done relative to the F key with the letters G,V,C,D,R or T.

### 2.3.2 GUI

The graphical user interface(GUI) was created with the addition of JavaFx which is a software platform that helps in the development of for instance GUI applications. [2]

To run the GUI run the View.java class. Here at first we created a simple "hello world" like GUI screen. Then we made a class called MyCircle.java which has a constructor which creates a Circle object with attributes X position , Y position , radius and color. This circle object represents the marble. Each marble has an X,Y coordinate of where it is located on the board , a size and a color. JavaFx works with stages and scenes. Think of stage which can have multiple scenes. The scene which is currently on stage is the one we see , similar to a play the scenes can be changed by Stage.setScene(sceneWeWant). Next we have Class called ConfirmBox.java. The purpose of this class would be to close the application when the client presses the X in the top right hand corner. This class is here to confirm whether the client truly wishes to exit the program. Thus we show a pop up window with a warning message. If we select no we are brought back to the scene we are currently on which is this case is the 2PlayerGameScene.

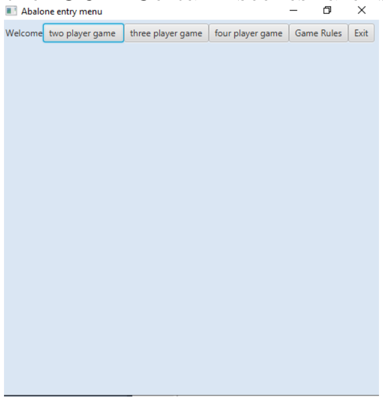


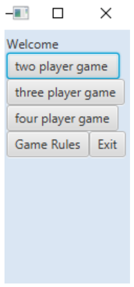
You are exiting

Are you sure you want to exit? You may lose the Game

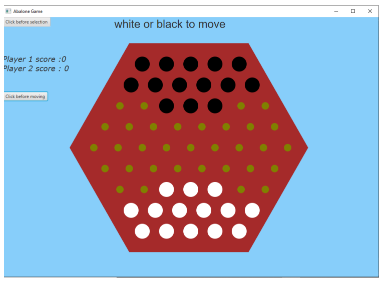
The purpose of this class would be to close the application when the client presses the X in the top right hand corner. This class is here to confirm whether the client truly wishes to exit the program. Thus we show a pop up window with a warning message. If we select no we are brought back to the scene we are currently on which is this case is the 2PlayerGameScene.

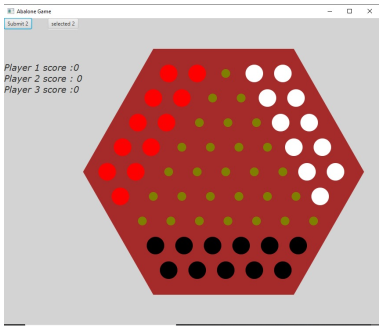
Currently the welcome screen,instructions and board game mode views have been created for the GUI. Certain scenes are scalable such as the menu whist others are not such as the boards.

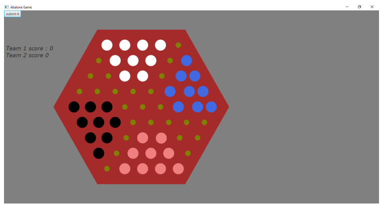




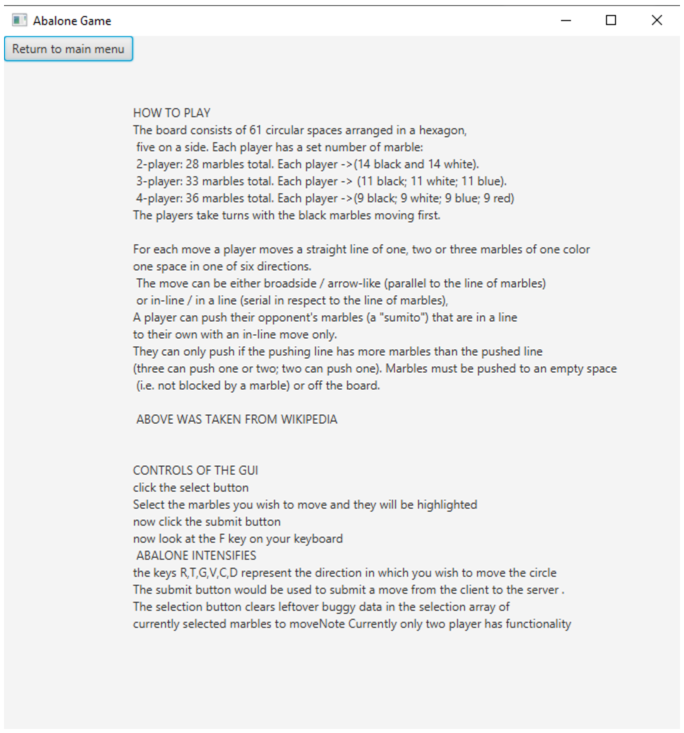
The current board layouts look like so below.







Currently only the 2 player game mode has some functionality. For the client-server we use the TUI(text user interface). The GUI is only for local games at the moment. One can enter the 2 player scene via the main menu or can enter the Game Rules scene if they are unfamiliar with the layout or controls. The controls of the GUI are stated here.



HOW TO PLAY
The board consists of 61 circular spaces arranged in a hexagon,
five on a side. Each player has a set number of marble:
2-player: 28 marbles total. Each player ->(14 black and 14 white).
3-player: 33 marbles total. Each player -> (11 black; 11 white; 11 blue).
4-player: 36 marbles total. Each player ->(9 black; 9 white; 9 blue; 9 red)
The players take turns with the black marbles moving first.

For each move a player moves a straight line of one, two or three marbles of one color
one space in one of six directions.
The move can be either broadside / arrow-like (parallel to the line of marbles)
or in-line / in a line (serial in respect to the line of marbles),
A player can push their opponent's marbles (a "sumito") that are in a line
to their own with an in-line move only.
They can only push if the pushing line has more marbles than the pushed line
(three can push one or two; two can push one). Marbles must be pushed to an empty space
(i.e. not blocked by a marble) or off the board.

ABOVE WAS TAKEN FROM WIKIPEDIA

CONTROLS OF THE GUI
click the select button
Select the marbles you wish to move and they will be highlighted
now click the submit button
now look at the F key on your keyboard
ABALONE INTENSIFIES
the keys R,T,G,V,C,D represent the direction in which you wish to move the circle
The submit button would be used to submit a move from the client to the server .
The selection button clears leftover buggy data in the selection array of
currently selected marbles to moveNote Currently only two player has functionality

In its current state the GUI can detect a legal selection of marbles to move however has not fulfilled the game logic. It is yet to implement sumito and may move to a location already occupied which would lie on top of another marble which is an illegal move. However, if the selected marbles move to a location where there is not another marbles,then the move would be lega as once one of the keys R,T,G,V,C or D are pressed a setOnkeyEventHandler will move the selected marbles the correct distance in that location. In short the GUI can detect a valid neighbouring selection of marbles and guarantees that they only move one square via the setOnKeyPressed event in JavaFx.

## 2.4 Controller package

### 2.4.1 Agent package

In this package, the Agents are defined. Agents can be seen as the 'brain' of a Player.
The superclass of all Agents is Agent:
Both fields can be set, but only the player has a getter by default.

- abstract Move makeMove(); determines a move based on the current state of the Game for its player.
- Player player; the Player the Agent controls.
- Game game; the Game the Agent is involved in.

Agent is extended by HumanAgent, asking the user for a decision through a UI:

- private UI ui; the UI that is used to get a decision from the user.
- Move makeMove(); makes a move according to user input, implementing from its superclass.

Another extension of Agent is NaiveAgent:

- Move makeMove(); implements from its superclass, selecting a random 1-marble move.

### 2.4.2 Game

- Game(List<Player> players) method Creates a new Game to be played. with parameter players. ie. The players participating in the Game. We use a switch case for each game mode and an enum of colors. WHITE,BLACK,BLUE,RED.
- getPlayers() method return players
- finished() method returns a boolean telling you if the game is over either by 0 turns left or winner reached.
- getBoard returns a board object.
- getTurn() Gets the number of the player whose turn it is. Returns the number of the player whose turn it is.
- move(Move move) throws illegalMoveException if the move is not valid. Allows player to make a move in the game, assuming it is that player's turn.The parameter is the move one wants to make and requires a move object.

### 2.4.3 GameRules

- isLegal(Move move, Board board) method tells you if a move is legal or not. It returns a boolean.

  Checks if a move is legal using proof by (lack of) counterexample.The following rule violations are checked:It is not allowed to commit suicide, i.e., it is not allowed to push or move either your marbles or your teammate's marbles off the board.Unless in a Sumito position, all marbles must move into a free space(4-players) In-line moves are only allowed if the first marble in the pushing line belongs to the current player(4-players) The column must contain at least one marble that belongs to you.You may only move your own (4-players: or your teammate's) marbles. parameter1) move the move to be checked parameter2) board the board played on return true if none of the rules are violated
- getWinner(Game game) returns the winning marble, or null in case of draw. Gives the winner of the game (or winning team), assuming the game has finished.The parameter is the game to be evaluated. This was implemented with a hashmap with key being marble and value being Integer. Using a for each loop we calculate the score of each player and find the winner.
- boolean isFinished(Game game) method returns true if the game is finished. The parameter is the game to be checked.This was implemented by first checking if turnLeft $< 0$. This would result in the game being a draw. If the game is a 4 player game we find if a team has 6 points or more. You get a point by knocking an opponents marble off the board via sumito. If the game is not a 4 player one we can check if any player has an individual score of 6.If they do return true. else false.

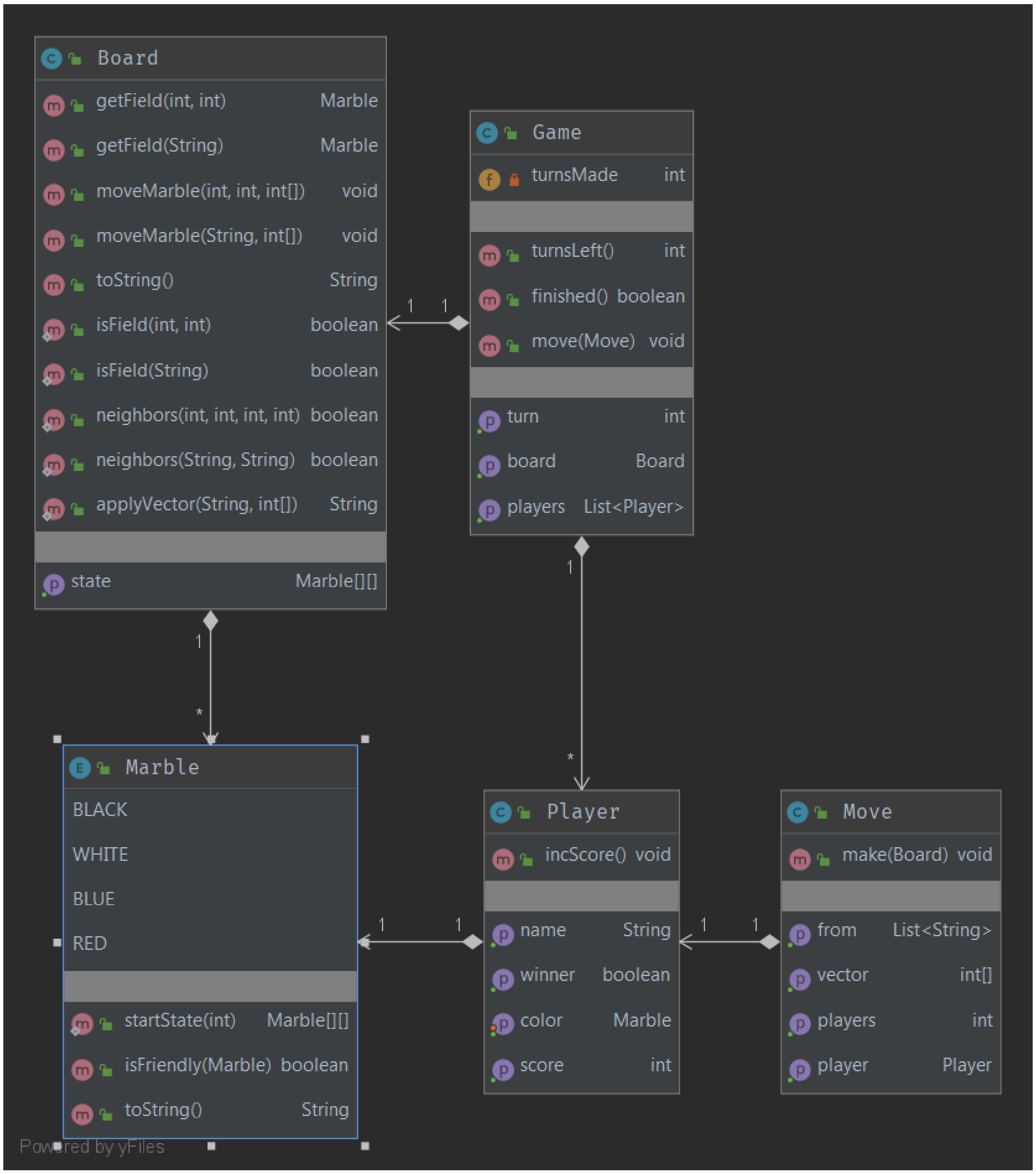
### 2.4.4 Sumito interface

- boolean pushOff(Move move, Board board). Checks if a marble gets pushed off, given the move is legal and in Sumito position. parameter1) move the move to be checked, given it is legal and in Sumito position parameter2) board the board played on returns <code>true</code> if any marble is pushed off. This is done with a String array to get the front most marble and checking if it is not located in a legal field(position square on board).

- sumito(Move move, Board board) method checks if a move has a sumito. parameter1) move the move to be checked parameter2) board the board played on return <code>true</code> if the move has any sumito. For this task we have submethods of sumito21() which is when two marbles push one , sumito31() which is when three marbles push one and sumito32() which is when three marbles push two. This is done by getting the move object and pushing the additional enemy marbles back in that same vector direction. To do this we first find if an enemy marble is located at the square to which we are moving to. For four player we made extra checks as we need to see if the marble is a teammate or an enemy. This is done with move.getPlayer.getColor.There is also a switch case for game mode 4 or game2_and_game3.

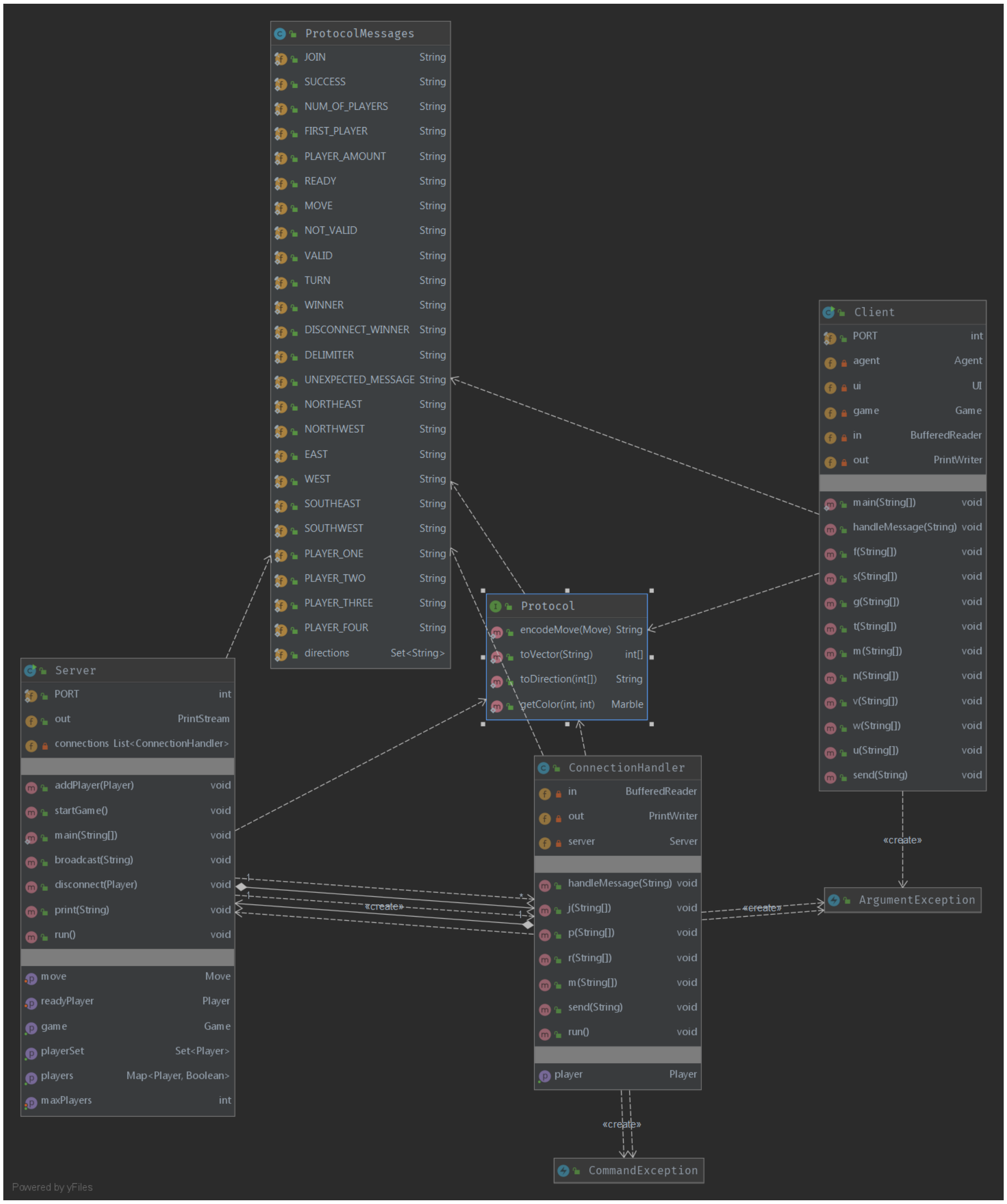
# 3 Design

## 3.1 Model



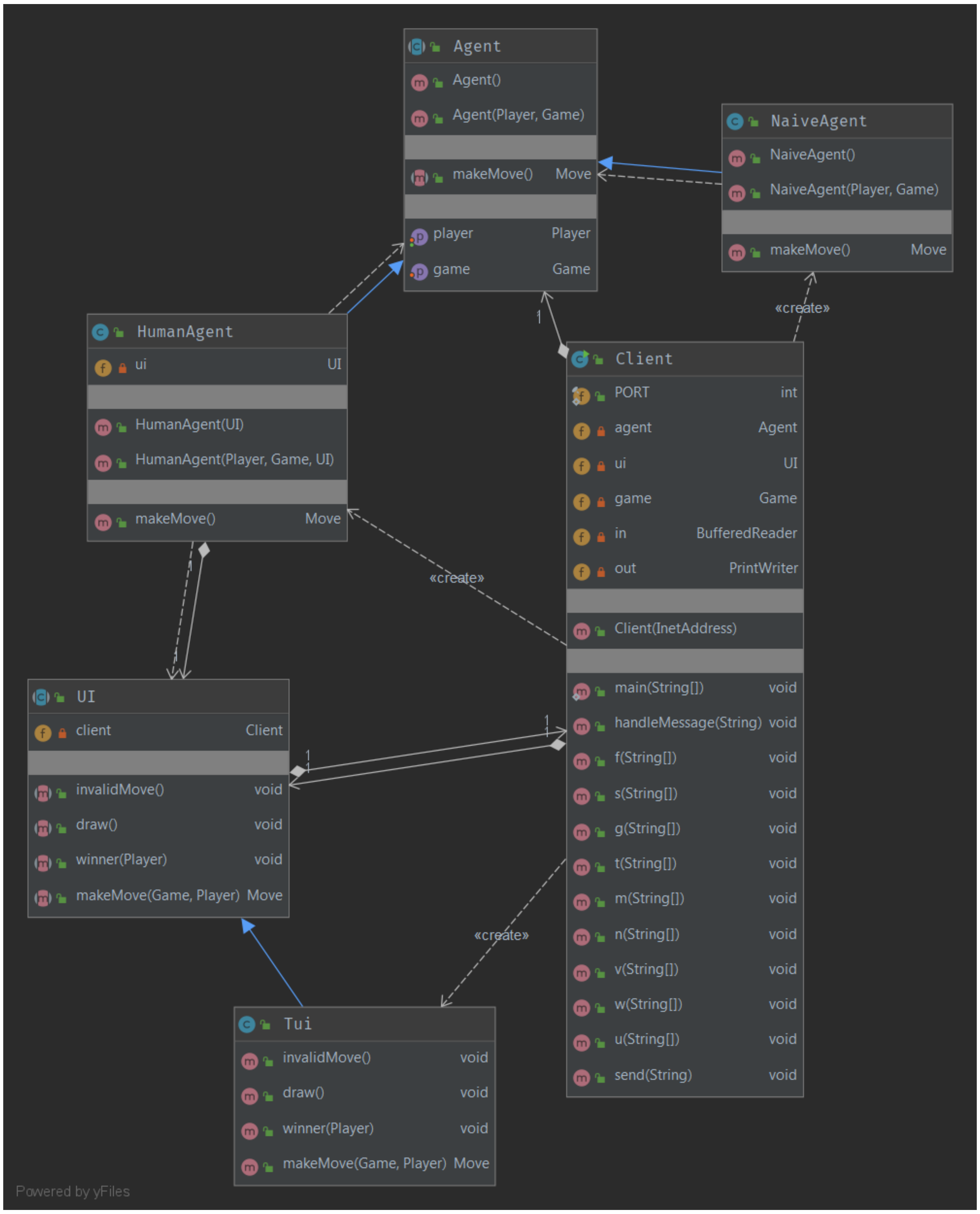
The Game is built up of a Board, containing Marbles, and 2 to 4 Players, assigned a Marble color, making Moves on the Board. A Move can be made directly on a Board, or alternatively via the Game. The inner structure of the Board is a 2D-array of Marbles (an empty field is a null entry in the array), of which the entries are named with an int[] in the form [row, column] or a String with these numbers concatenated. Whereas the rows are horizontal and straight, the columns are bent (see the documentation of the field Board.state for a map). A Move has its direction recorded as a vector.

## 3.2 Networking



The server, as well as the client, have a main method to initiate contact, and the Server will start a ConnectionHandler for each client. The ConnectionHandler and the Client handle incoming messages using reflection, which causes them to have methods with a name of one letter such that they match the Protocol command. If applicable, they will auto-respond, otherwise they only process the message. It is also possible for a side to send a message without a message to respond to, for example the announcement of the winner.

## 3.3 Player



When the client is started and a game is started, an Agent as well as a UI (especially in the case of a HumanAgent, where the Move is prompted through the UI) is created.

# 4 Testing

## 4.1 Network Testing

Here we have a Mock Server package which creates a mock connection between the client and server. There are three classes here. Namely, 1)TestClient, 2)TestServer 3)EchoTester.

The TestServer class initiates a server socket. The TestClient then initiates a socket which connects to the server socket. When a socket is accepted to the Server socket a Client connected message is printed to console of TestServer. In a live version the user would enter the Port they wish to connect to and the nickname they would like. Here we hard code the port as localhost to test on the same machine and give an arbitrary string as nickname. The Server will send a welcome plus nickname of client to the client so they too know that a connection has been successfully achieved. The TestClient will now send a string and the TestServer should receive it. This will print Input received: "input"; to console. This shows that the Client can send to the server. Now we send an "exit" string which will terminate the socket and thus cease the connection betwixt the client and socket. If this test is successful a "Server Client Test passed message will be seen in TestServer and TestClient will have an exit code and will no longer be running.

## 4.2 Model Testing

## 4.3 Controller Testing

## 4.4 View Testing

//TO DO test the tui and gui for valid init and ti

## References

[1] *Abalone game rules*
`https://en.wikipedia.org/wiki/Abalone_(board_game)`

[2] *JavaFx download and information page*
`https://gluonhq.com/products/javafx/`

[3] *Protocol_ messages*
`https://docs.google.com/document/d/1w4ei3_JhhaLxqc4mHS8kA_3mnAHtZVKL7p3reB-c1NQ/edit`