

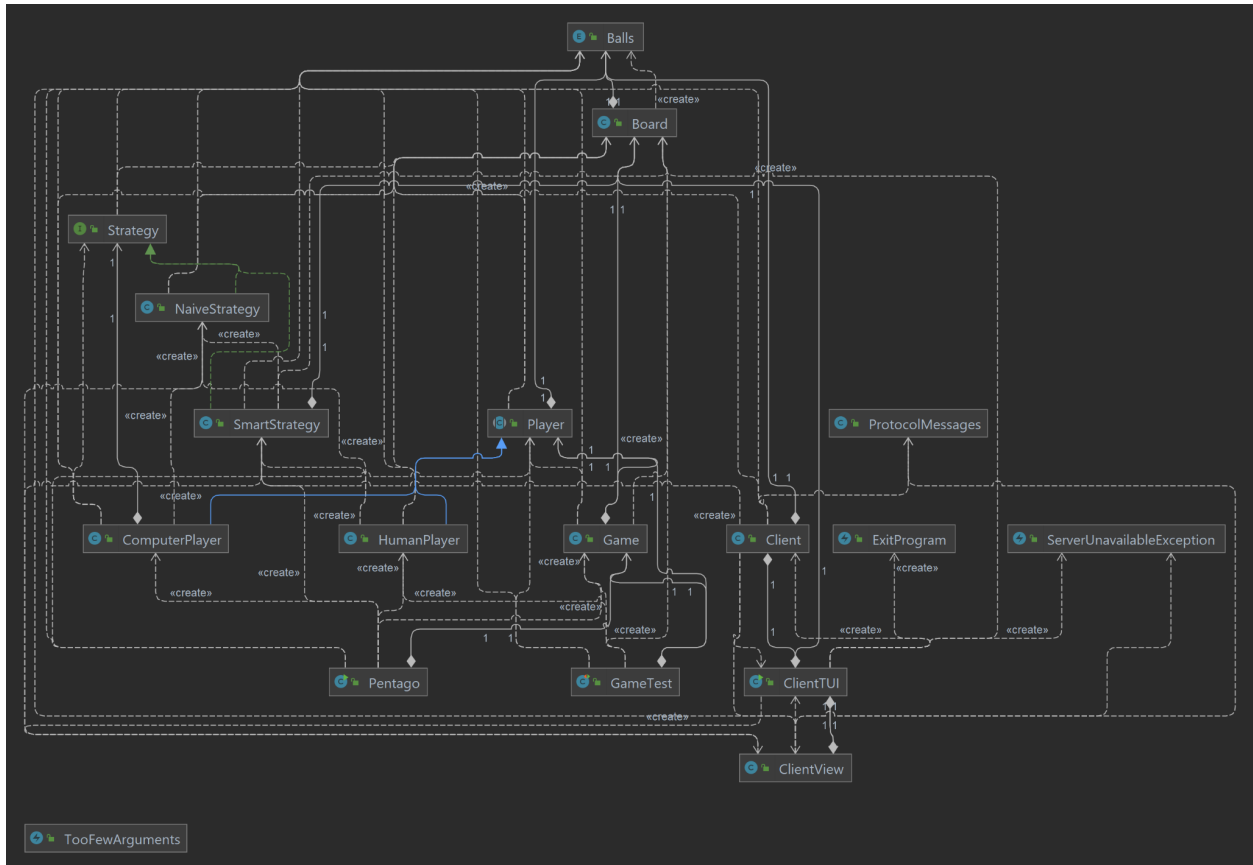
# Programming project report

Botnarenco Daniel s2386593

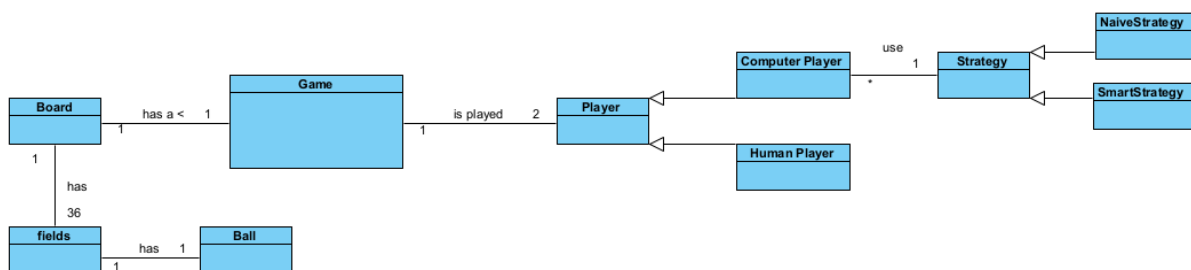
Resit-11

# Overall Design

## Class Diagram



Given the fact that I believe this diagram does not give the appropriate presentation of the intended interaction between the classes, another one has been provided, but without any more information.



More information about the class diagrams with fields and methods will be provided in the appendix.

The board of the game is represented by a two dimensional array with 36 fields ( 35 if you are a computer ). The board class also takes care of the input from the user and parses it accordingly, it also checks the validity of the commands. Moreover, the class has the functionality to determine the winner but this is being called by the Game class and can also detect when the board is full, meaning the game has ended.

The fields inside the board are defined by having a Balls enum class. The fields can then take values of EMPTY, BLACK and WHITE. The balls are assigned in the order they connect in. Therefore, the first player is going to be black, so the second one is going to be white.

Considering the common methods between HumanPlayer and ComputerPlayer, a new abstract class Player has been created to link these two types of players. Each player has the purpose of making moves on the board. The HumanPlayer will ask the user to manually input a move to be made and the ComputerPlayer using different strategies will return a legal move.

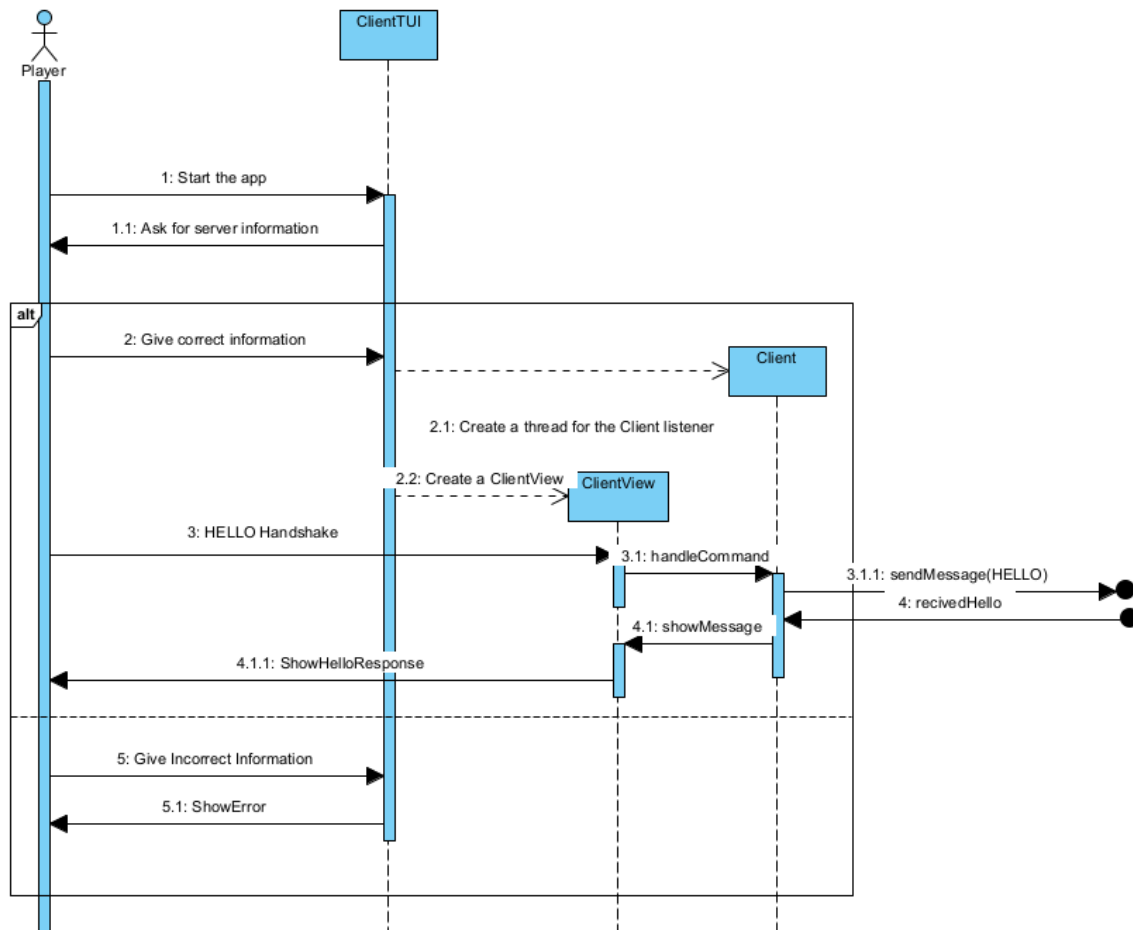
Provided this was a solo project I had the option to choose between implementing the server or the client, I chose to implement the Client.

When launching the Client, a connection to the server has to be established. In order to do this, the method createConnection() takes care of the input from the user to connect to a specific IP and port. If by any chance the client inputs wrong information, the application will ask the user again for an IP and port. After the connection has been made, a new Client thread will be created to listen to the messages sent from the server, and for specific messages it will automatically reply with the predefined protocol messages.

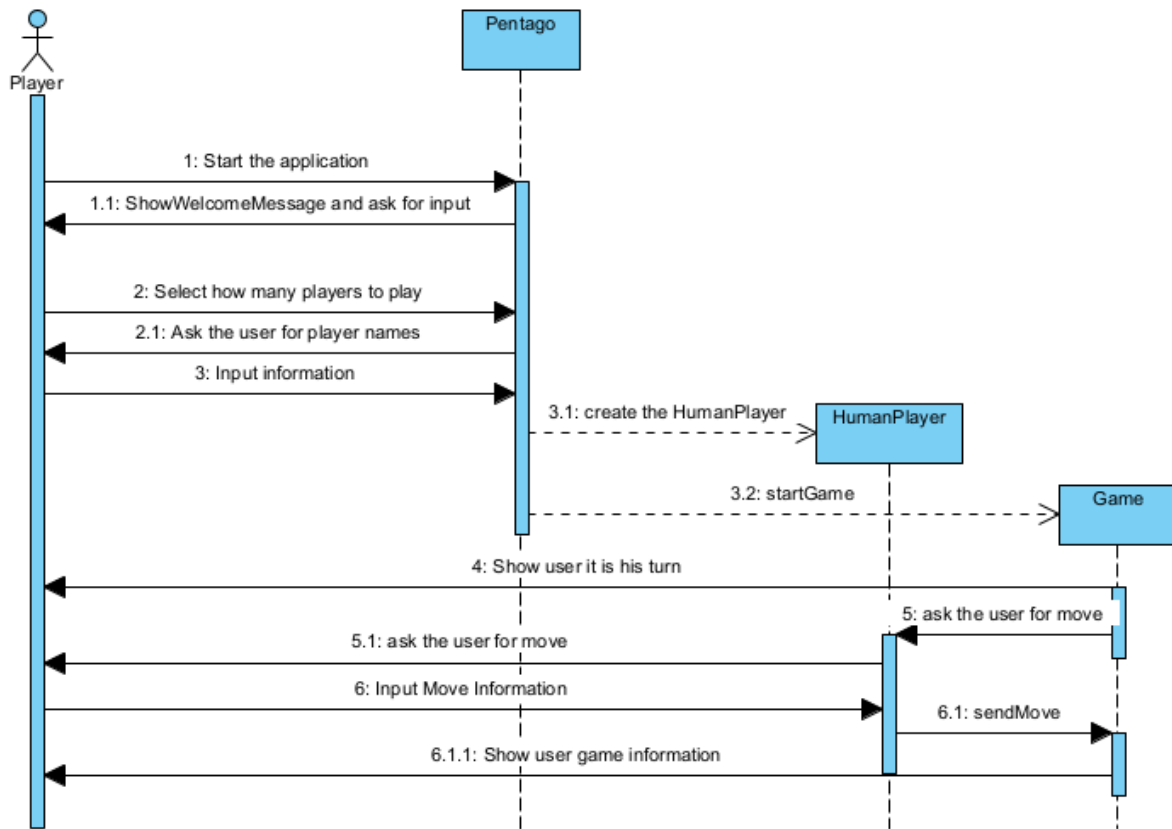
When launching the Client, a ClientView for that client will be created, where the input of the user will be validated in the handleuserinput() method against the predefined protocol. Then it will send the messages to the server.

Sequence diagrams will be provided to show a more detailed description of the interactions between components.

## Sequence Diagrams



This diagram shows the interaction between the Player that would like to play a multiplayer game and the application itself. We can see that if the player gives correct information about the server, a new thread will be created to listen on that socket. Lastly, a clientView has been created to show output from the server to the player.



This sequence diagram presents the way in which running the Pentago client you input moves to a HumanPlayer that sends the move to the game. There it will get treated correctly and will update the game with regards to the move that was made.

## MVC Pattern

We observe that the project implements a Model-View-Controller pattern from the package names. These package names divide the main tasks of the project into three responsibilities: model, view and controller.

The model package contains the information of the application. This package doesn't know how to present the data and does not handle the user input. This is defined by the most important class in this package, the Board class.

The package also contains the strategies, the players, the Ball enumeration and the protocol messages.

The View package displays the data from the model to the client in an appealing manner. This being a Textual User Interface, it will show the data on the console of the program.

The controller package takes care of the input from the user and triggers different components in the program. The package is represented by the Client and the ClientTUI, Game and the Pentago class. All these classes require user input in order to actually make changes to the state of the game or even clients.

## Functional requirements

Crucial requirements:

1. A standard game can be played on both client and server in conjunction with the reference server and client, respectively.
  - This was implemented, but, considering that I had implemented the application on my own, I had the choice to pick the client or the server. Therefore, I only implemented the client, which is able to connect and play on the reference server.
2. The client can play as a human player, controlled by the user.
  - This has been implemented by giving the user the ability to input his own moves.
3. The client can play as a computer player, controlled by AI.
  - This has been implemented on an offline game in the setup stage. The user is asked if he would like to play against or be a computer player. On the client side, the user can play a game by using the AI.

## Important requirements:

1. When the server is started, it will ask the user to input a port number where it will listen to. If this number is already in use, the server will ask again.
  - This has been implemented by having the `createConnection()` method inside the `ClientTUI` class.
2. When the client is started, it should ask the user for the IP-address and port number of the server to connect to.
  - This can take place because of using the implementation of the `createConnection` method in the `ClientTUI` class.
3. When a client is controlled by a human player, the user can request a possible legal move as a hint via the TUI.
  - This is done by typing "HINT" inside the textual user interface. The AI will then return a legal move.
4. The client can play a full game automatically as the AI without any intervention from the user.
  - This is done by creating an offline game with at least one AI player, or by activating the AI on the client by typing "AI" and then queuing up for a game. The AI will then play the game for the player.
5. The AI difficulty can be adjusted by the user via the TUI.
  - This has been implemented by typing "CHANGE" in the console. It will then be shown that the AI will return a move.
6. Whenever the game has finished ( except when the server is disconnected ), a new game can be played without having to establish a new connection in between.
  - After a game is finished, the client is still connected to the server. Therefore, he/she can join the queue again to play another game.
7. All communication outside the recurring game, such as handshakes and feature recognition, works on both the client and the server in conjunction with the reference server and client, respectively.

- I emphasize that this was a solo project, since handshakes and feature recognition work on the client, following the protocol that was given to us.
8. Whenever a client loses connection to a server, they should gracefully terminate.
    - When the client gets thrown a IO Exception, it will close the connection and reset it, terminating gracefully.
  9. Whenever a client disconnects during a game, the server should inform the other clients and end the game, allowing the other player to start a new game.
    - After receiving a "GAMEOVER" message, the client will show the users the commands he/she can call.



## Concurrency mechanism

One of the requirements was “multithreading in networking code”. In order to abide this requirement, the use of threads was necessary. As it was previously discussed in the document, when the client manages to establish a connection to the server, a new thread of a Client will be created.

```
view.showMessage(s: "Attempting to connect to " + address + ":" + port + "...");
serverSock = new Socket(address, port);
in = new BufferedReader(new InputStreamReader(serverSock.getInputStream()));
out = new BufferedWriter(new OutputStreamWriter(serverSock.getOutputStream()));
view.showMessage(s: "Connection successful.");

// Initialize the listener, that will take all the messages coming from the
// server and forward them to the view
Client listener = new Client(serverSock, client: this);
(new Thread(listener)).start();
```

The purpose of this Thread is to read messages from the server and react to them if necessary. Otherwise, it will show them to the client through the use of the ClientView class. In addition, because this was a solo project, no other threads were needed.

Moreover, since each Client believes that they are using the black ball and the opponent has the white ball, each of the clients has their own board to make moves etc.

One important thread that we had to implement was the thread that handles the PING and PONG messages. This thread reads from the same socket and will only write to the server the pong message when it gets notified about that message. To have the application thread safe as two threads are reading the same message from the server, in the PingPong class a wait command will have the thread wait until it's interrupted by changing the condition in the if statement.

```
public void run() {
    while (!ping) {
        try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    client.sendMessage(ProtocolMessages.PONG);
}
```

The program is deemed thread safe, as no other threads use the same objects to determine moves etc.

## Metrics

After a lot of browsing, I have managed to find the CodeMR plugin for IntelliJ, which gives interesting information about the metrics of the project.

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	Client					82	low-medium	low-medium	low	low-medium
2	Pentago					40	low	low-medium	low	low
3	Board					145	medium-high	low	low	low-medium
4	ClientView					134	low-medium	low	low	low-medium
5	ClientTUI					79	low-medium	low	low-medium	low-medium
6	HumanPlayer					67	low-medium	low	low	low-medium
7	Game					65	low-medium	low	low	low-medium
8	Balls					28	low-medium	low	low	low
9	ComputerPlayer					11	low-medium	low	low	low
10	ServerUnavailable...					4	low-medium	low	low	low
11	TooFewArguments					4	low-medium	low	low	low
12	ExitProgram					4	low-medium	low	low	low
13	SmartStrategy					35	low	low	low	low
14	NaiveStrategy					27	low	low	low	low
15	ProtocolMessages					13	low	low	low	low
16	Player					11	low	low	low	low
17	Strategy					3	low	low	low	low

## LOC

### Analysis of resit-11

#### General Information

**Total lines of code: 752**

**Number of classes: 17**

**Number of packages: 7**

**Number of external packages: 6**

**Number of external classes: 44**

**Number of problematic classes: 0**

**Number of highly problematic classes: 0**

The project consists of 752 lines of code.

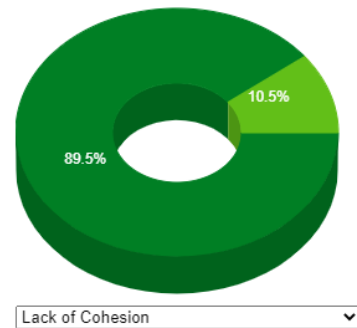
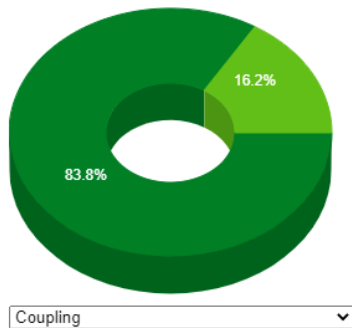
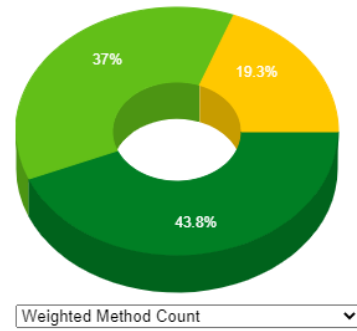
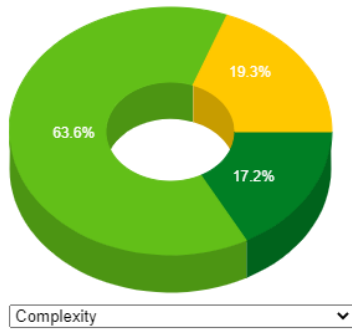
## Cyclomatic Complexity

Cyclomatic complexity is a metric to determine the complexity of a program.

class	▼	Cyclic
controller.Client		2
controller.ClientTUI		2
view.ClientView		2
Pentago		0
controller.Game		0
exceptions.ExitProgram		0
exceptions.ServerUnavailableException		0
exceptions.TooFewArguments		0
model.Balls		0
model.Board		0
model.ProtocolMessages		0

class	▼	Cyclic
model.Balls		0
model.Board		0
model.ProtocolMessages		0
model.player.ComputerPlayer		0
model.player.HumanPlayer		0
model.player.Player		0
model.player.computer.NaiveStrategy		0
model.player.computer.SmartStrategy		0
tests.GameTest		0
<b>Total</b>		
<b>Average</b>		0.35

Looking at the values the IDE has returned, we can observe that most of the classes have a cyclomatic complexity under 2, for an average of the total project to 0.35. This means that the code is easy to understand and to modify.







As we can see, almost all the colors of the program are at maximum yellow, which means medium/medium high values for the metric we have set it underneath.

These metrics and the graphs show in an appealing manner how complex and easy to understand the code is. This also gives interesting information on each of the important aspects of the metrics that are available.

## Testing

When testing a software, we have to keep in mind what parts of the program are more prone to errors. Taking this into account and having the prior knowledge of the coverage of the application from the initial report, we can see what parts of the system have to be tested more.

Element	Class, %	Method, %	Line, %
 Game	100% (1/1)	71% (10/14)	80% (79/98)
 player	75% (3/4)	63% (7/11)	51% (25/49)
 Balls	100% (2/2)	71% (5/7)	66% (12/18)
 Board	100% (1/1)	88% (8/9)	82% (63/76)

We can see that the Game class and also the Board class are extensively used. Given that, these are the main two foci that we would have to keep in mind when creating the tests.

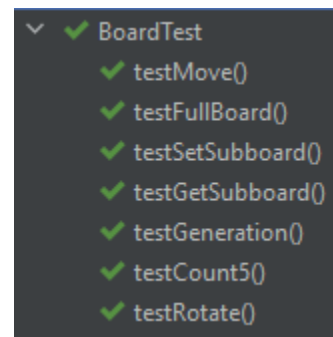
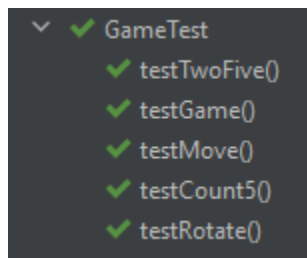
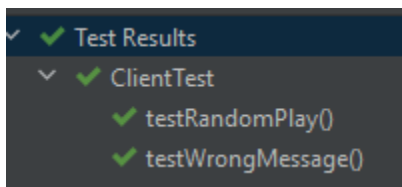
Starting with the Board class which has methods such as `move()`, `boardIsFull()`, `count()`, `isWinner()` and `rotate()`, tests for these methods had the purpose to check if the manipulation of the data is being done correctly. In order to do this, the following tests were created: `testFullBoard`, `testMove`, `testCount5`, `testRotate` and tests for setting and getting off sub boards (smaller 3 by 3 arrays that are divided in four quadrants). At first, the tests were failing, meaning that I had to look in deeper to see what the problem was. Some errors that were encountered were: not selecting the right quadrant, not rotating the sub board as intended, or even some out of bounds errors that were necessary to fix. Therefore, by using the tests I was able to make my code run as intended.

Secondly, we have to take into account the `GameTest` class, which tests if the game logic is implemented correctly. Methods inside the Game class are: `endGame()`, `determineWinner()`. Tests for these methods were done even in edge cases, where both the players win at the same time and there is no winner. More tests were done too in order to check if players can actually make moves and then be registered. A test was made to see a full course of a game, two AI's, Smart and Naive play each other. This test checks if the game is being played correctly and is finished and that the Smart AI is indeed smart.

In addition to these two test classes, a third class, `ClientTest` has been added. Considering the fact that it was a solo project, it seemed necessary to test how a client would react to the messages received from the server. The Client class takes care of reading input from the server and writing to the server. In this way, the tests were made to see if the data is being transmitted between these two components over a socket.

In addition to this, a test has been added that connects an AI to the server and allows him to play a full game. The tests in this class are highly dependent on the sequence of running the tests and what the other client on the server is doing (the player who I have to play against).

It is highly suggested that each method should be run separately from each other rather than running the full class. I also needed to run them a couple of times to see if the intended outcome was accomplished.



Unit testing is really important for a project this size, but sometimes it might not be enough to check the functionality. This is why System testing has also been done for this project.

System testing is conducting a test on a complete integrated system to evaluate the system's compliance with its specified requirements.

These have been conducted by connecting the client to the server and then, manually inputting malformed information to see how the program will handle that. At first, there were a lot of errors of the protocol not being implemented correctly.

System testing had to be done for the ClientView as well, because it first interacted with the user and then with the server. As discussed in the requirements the AI functionality had to be incorporated into the client, but the server should not know about the existence of the AI.

It took some time until we made the AI react to messages from the server and be activated/deactivated at specific points, but it was all thanks to the system tests.

Some other types of systems tests were the sudden disconnects from the server to see if the Client gracefully terminates, or to see if the AI has been implemented correctly so that it sends the server legal moves.

As seen from the Metrics, I have incorporated in the report and the explanation of which classes are being most run ( Game and Board ). We have seen that the classes are not that complex, knowing this we can assume that the other classes work as intended everytime. For example: HumanPlayer only asks the player for an input, this will always work as this is how you are supposed to play. In addition, trivial classes such as Balls ( enum ), Strategy ( Interface ) were not tested.

## **Reflection on Design**

Looking back on the initial Design of the project, I came to the conclusion that I was not that far off from the actual implementation of the project. For example, how the board is defined in the application has stayed the same, a 6 by 6 array. In addition to this, because the board was a 2D array, it was straightforward to have a rotation of a 3 by 3 sub board but also to select each of these sub boards.

In the Initial Design, I also talk about having the project divided into model, view and controller. This has stayed the same because it is a good practice to divide the tasks of the program. Als, in the present document I had discussed the Game class that it will get input from the user. This Is not right, the input from the user was received in an offline game through the HumanPlayer and on the server game through the ClientView class. Also the initial design did not contain the new class PingPong and I have not discussed the threads that will be created in the application. This has been addressed in this document in the Concurrency Mechanism part.

Also, a way to improve the overall design of the application was to have an interface for the clients (Game/Client) but as previously mentioned only one of them had to be made. Therefore, there was no implementation of an interface in this regard.

## **Reflection on process**

The process of creating this application started 5 weeks ago. At first, when I saw the game we had, I thought this was going to be quite easy. As a matter of fact, it was not as easy as i thought, but, fortunately, it was not that difficult either. Sadly, because I have worked on my own on this project, there was no other person to provide feedback on the design of the application other than the Teaching Assistants.

This turned out to be a problem before the Initial Project Design deadline was, because I had to discuss with TA's about the design that I had in mind. After I had received good,useful feedback on it, I started implementing the actual application.

This problem could have been solved by starting a bit earlier on the designing of the project and making a plan to follow during it, or by having a teammate to discuss these topics with him.

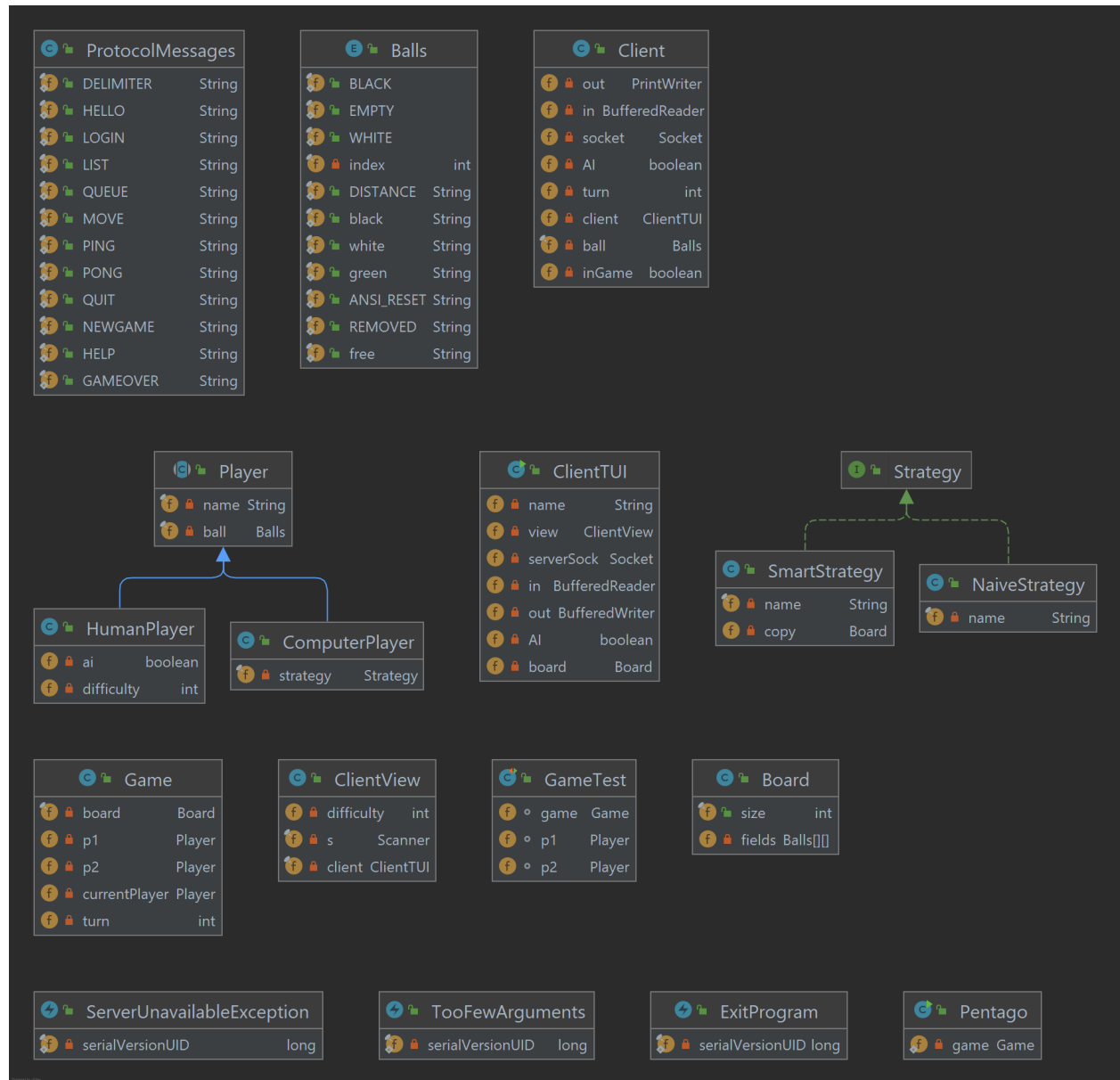
I believe that the most demanding part of this project was the creation of the Client to be able to follow all the Protocol messages. While developing this part of the application, I would still find bugs in other parts of the code. This fact, combined with the integration tests and system tests, have helped me solve those problems.

Some shortcomings that I encountered while working on this project was that I did not keep in mind that the programs might have bugs and I would have to fix them. Hopefully, this did not go way over the schedule, but a great way to combat this would be to test the application while developing it. Additionally, before finishing this project I started having problems with the IntelliJML plugin inside the IDE, therefore I could not commit or push my work through to GIT. I tried contacting different people about this problem but was unsuccessful.

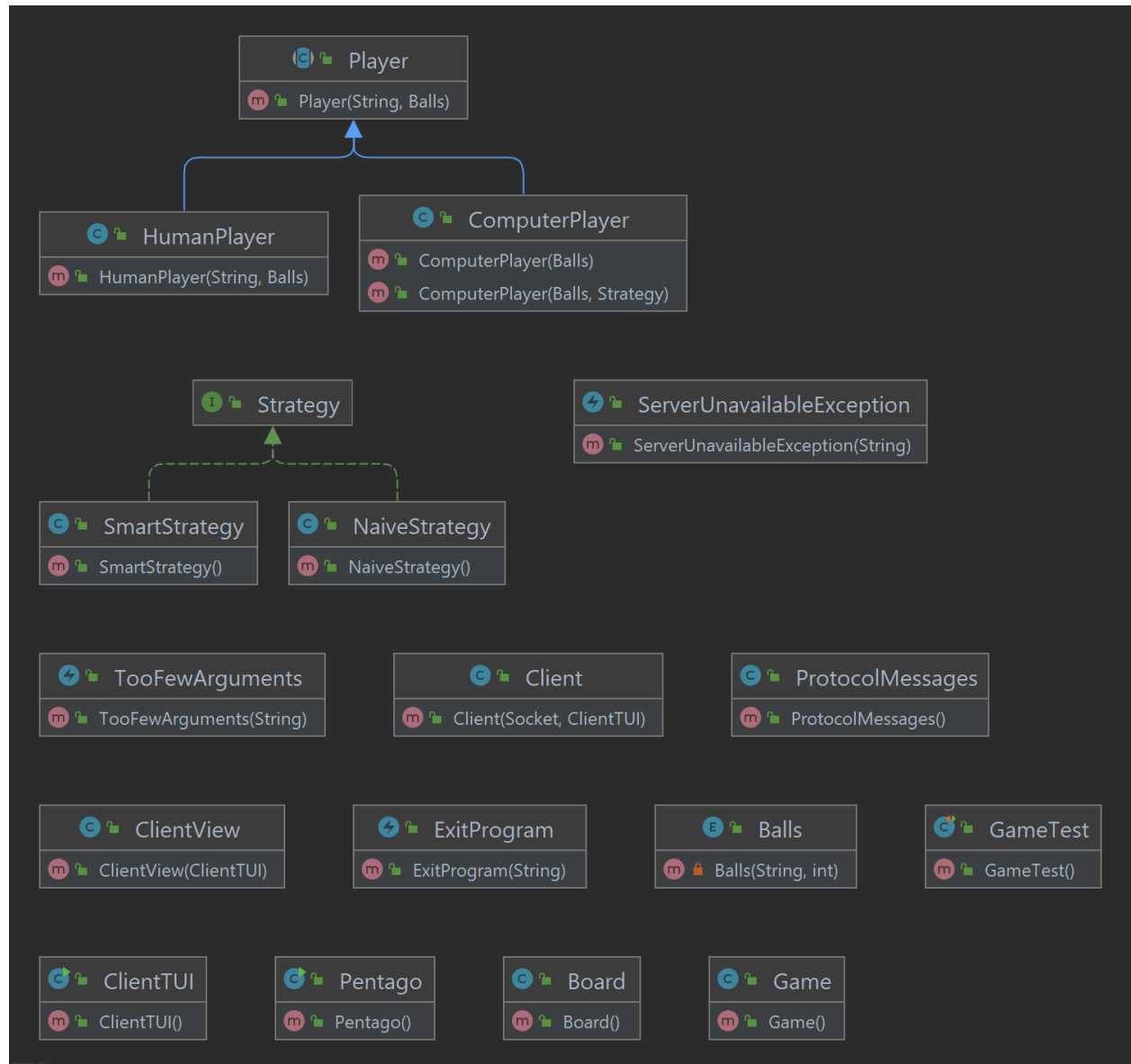
In conclusion, in order to create a project like this, rules have to be set from the beginning, such as time frames to when each part of the project has to be completed. If having a teammate is not a problem, I can now fully argue that having daily Scrum meetings with him would be beneficial to the entire project.



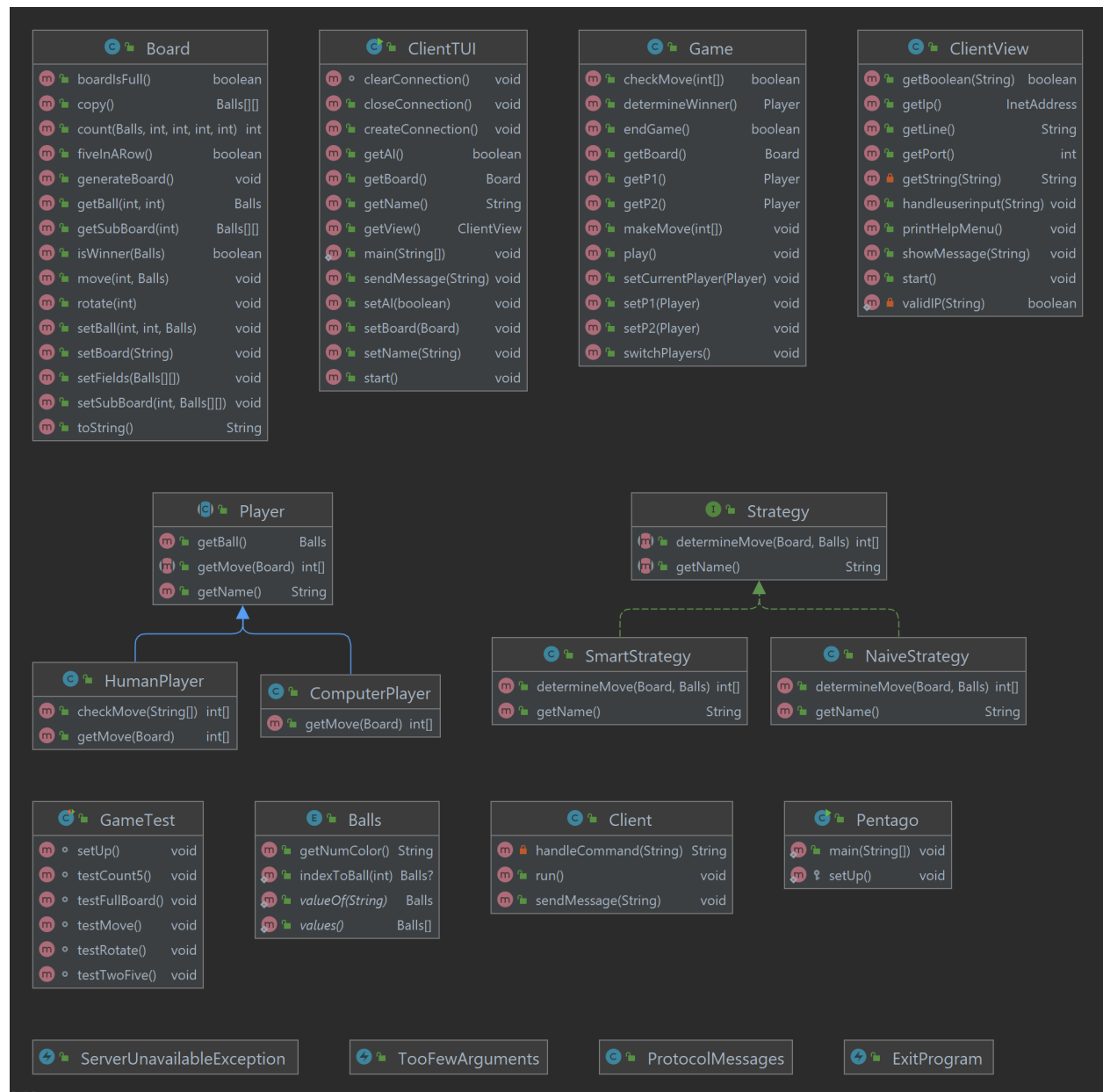
## Appendix



Class diagram with fields.



Class diagram with constructors.



Class diagram with methods.