Final Report

Programming project
Group Other 6.4

1. Justification of minimal requirements

The crucial requirements :

1. A standard game can be played on both client and server in conjunction with the reference server and client, respectively.

Implemented.

2. The client can play as a human player, controlled by the user. When the game starts you can choose to play as or against an AI or a normal player.

Implemented.

3. The client can play as a computer player, controlled by AI.
When one or more players are selected as an AI the client or clients play for themselves and make their own moves.

Implemented.

4. At any point in time, multiple games can be running independently and simultaneously on the server. You can run multiple clients on the same server because they use clientHandlers so they don't necessarily store the server which makes it so that it is possible to play multiple games on the same server.

Not implemented.

The 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. There is a loop in the setup, when the port is taken it will loop again until the input port is something that is not taken.

Implemented.

2. When the client is started, it should ask the user for the IP-address and port number of the server to connect to. When you start the client it asks for both and then tries to connect to the server.

Implemented.

3. When the client is controlled by a human player, the user can request a possible legal move as a hint via the TUI.

Implemented.

4. The AI difficulty can be adjusted by the user via the TUI.

Not implemented.

5. All of the game rules are handled perfectly on both client and server in conjunction with the reference server and client, respectively.
The board and game class make sure that every move is valid and the server validates everything.

Implemented.

6. Whenever a game has finished (except when the server is disconnected), a new game can be played without needing to establish a new connection in between. When the game is over, the server sends GAMEOVER to both clients and then it is also specified who won the game or in case of a draw a draw is represented.

Implemented.

7. All communication outside of playing a game, such as handshakes and feature negotiation, works on both client and server in conjunction with the reference server and client, respectively.
All the connection handling is being done between the clienthandler and the serverhandler.

Implemented.

8. Whenever a client loses connection to a server, the client should gracefully terminate.
The collector client runs a shutdown method where it sends a goodbye message before shutting down properly.

Implemented.

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.

Implemented.

2. Explanation of realised design

2.1 Overall structure

In this part of the report we will explain our design choices, in particular which working pattern we have chosen, why we have chosen this design in general, will briefly explain how the classes and methods interact with each other and which role they play.

First to mention is that we have used MVC pattern, because this is what we took a closer look at during the module, secondly because this system is easy to understand and faster to implement, you have a good structure of dividing your classes, which you follow.

Let’s see now how our project is structured. In our project we got in the end *utils* folder with all model classes, client and server interfaces, several exception classes and threads classes used for communication between these two. We also have a *modelTest* folder with model test classes implemented. Secondly, we got an *application* folder with two classes regarding game and board business logic. All the business logic of the game, all the rules are implemented there. Then we got the *view* folder, with the TUI file there.

So on this diagram our model classes can be seen:

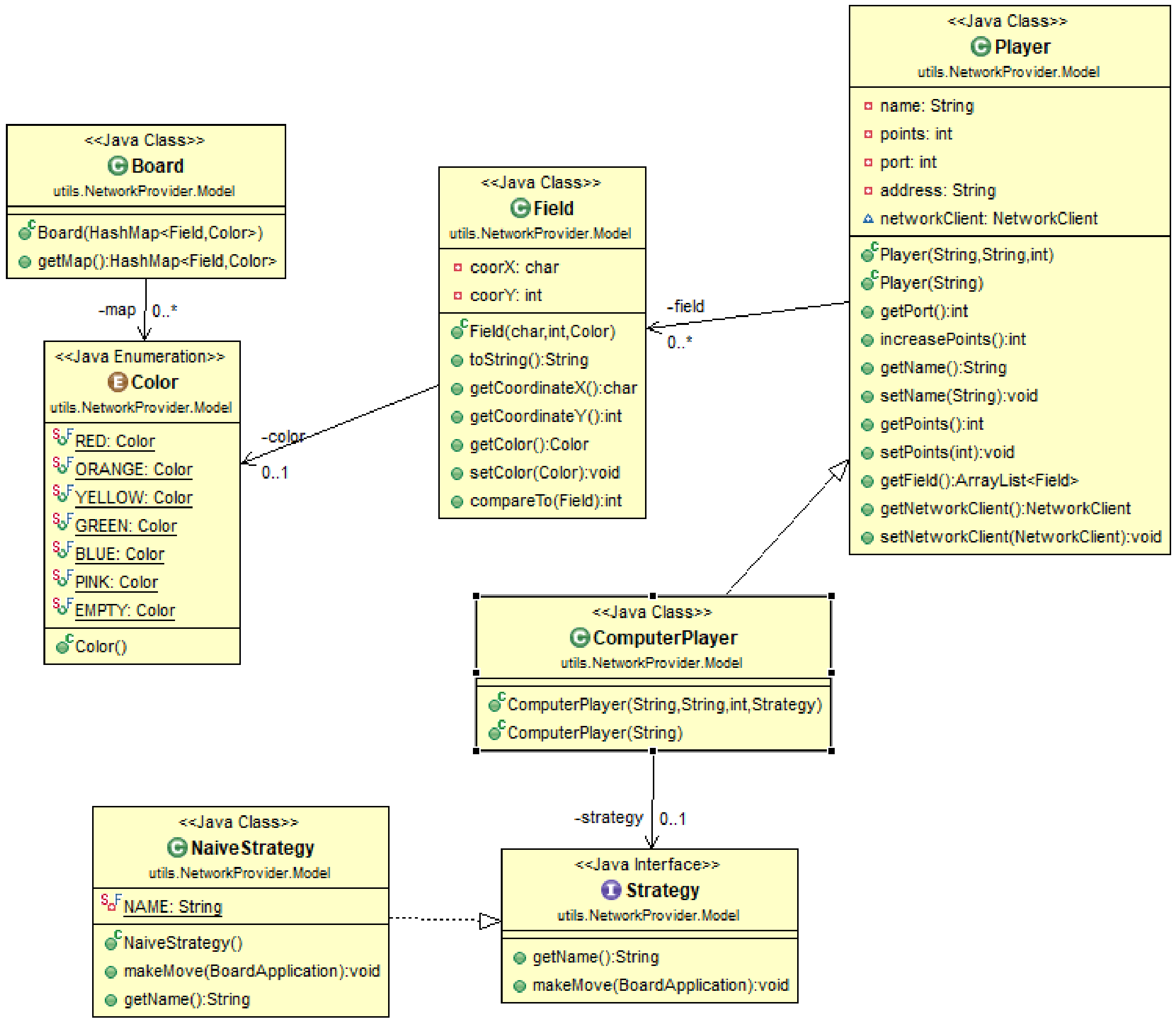

Diagram 1. *Model classes. Design phase.*

Huge part of our project is the client and server part. *Server* and *Client* folders contain all classes regarding server and client. In each of these two folders there are sub folders *Controller, Services and Services.Controller*. The last one contains classes BaseController, which is a simple abstract class, and Resolver, which is responsible for resolving messages coming from client to server and vice versa, also it can tell whether the message is supported and add other controllers. Then we have in each of the *Server* and *Client* folders the *Services* folder. This contains the class NetworkClient. On the server side it sends the messages from the corresponding client and on the client side this class creates new clients on the server, using socket and sends the message from client to output stream. Also, on the client side we have NetworkReceiver class, which basically catches messages for clients in the input stream, calls resolver to resolve the message. On the server side we have a UsersInQueue class. Regarding the *Controller* folder on each side, these contain classes to handle the requests from the client, e.h HELLO or MOVE. Finally, we have two classes CollectoClient and CollectoServer, running which you can start our server and create and connect new clients to it to run our game on one machine locally.

Here we have another diagram made during design phase of the project, so was modified a bit till the end of developing the game:

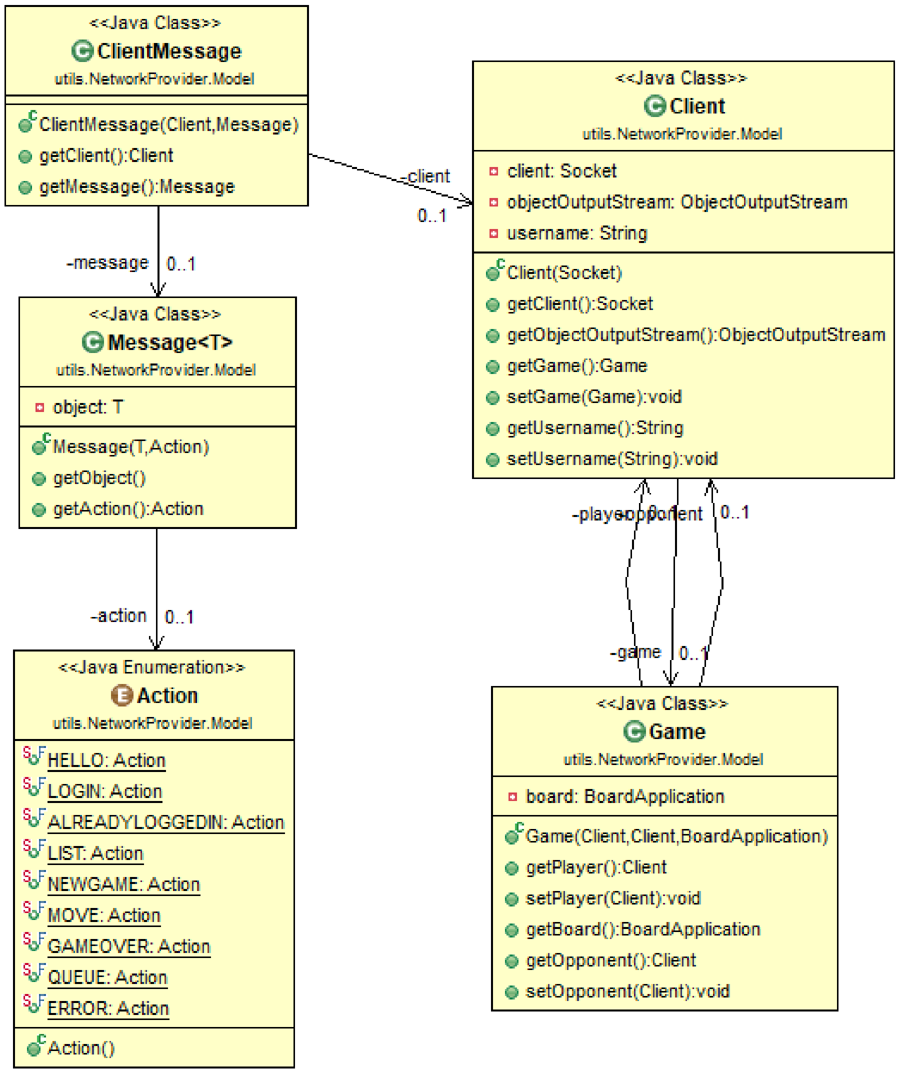

Diagram 1. *Classes interacting with the client. Design phase.*

2.2 Problems and solutions

We have made a couple of mistakes in our design choices which resulted in in restructuring our code multiple times, which of course is something you want to avoid at all costs because it essentially is a big waste of time and could have been avoided if we had taken more time to coincider design wisely before implementing it.We have tried to stick to our design we had made prior to when we actually started implementation of the project.

Let’s firstly discuss model classes and then move on to controllersIn our first design we wanted to have multiple classes, namely: game, board, field, player, ball, move, and multiple classes regarding the server and client. We later realised that this way was too complicated and brought us a lot of problems. Some of the classes had variables that we didn’t use but still should have been connected in the system, which caused lots of problems when we, for instance, tried to do the deep copy of the board instance. We needed to deep copy the board to see whether there are the available moves on a board which result in adjacent balls, but here we need to have a copy of the original board and since we could not do that with the old structure, we needed to change something. So we asked for help and the TA’s advised us to restructure our whole program which took us a lot of time as can be imagined.

After a lot of restructuring we removed the ball and move class and added a computer player class that extended the player class, also added color and direction classes which are basically enums, so nothing much to say about them. After simplifying our code, the deep copy function we created finally worked and we could move on to our next goal. What we had first was that every field has a ball object stored with a color on it and that ball was also connected to the field. Which accidentally resulted in a lot of problems so we chose to remove the ball class entirely and change it so that a field only stores a color and could easily get changed if we wanted to. We also noticed that we could make the moves on the board class instead of the move class because it all happens in the board and we only needed 4 methods to implement all the movements. So we didn’t need the move class as well and deleted it to make it all more simplistic.

3. Concurrency mechanism

It is clear that creating client-server communication requires implementing multiple threads to run to not have problems with communication and receiving requests when multiple clients want to access the server. To ensure everything works in good order, we have made multiple classes in our client server package runnable. Basically for every new client that is connecting to the server, a new thread is being created to handle that. So when we start our server by running class CollectoServer, we start *message listener*, *message handler* threads and clientPullListener thread. On the client side, when we run CollectoClient, we start only one thread for each created client, that allows the client to receive messages for him from the input stream. This did not create any problems for us. Using this we made it work and no real problems regarding concurrency came up. So for this project it wasn't a big problem luckily.

4. Reflection on design

As we have mentioned before, we initially created classes way bigger than necessary which resulted in a lot of problems down the line like deep copies not working and the whole system not working as efficiently as it could have. We have restructured a lot of our code after this which cost us a lot of precious time, but we did not change everything. We think this problem was caused because we rushed the design of our program. If we could do our project again we would do a lot of things differently like creating really simple classes that are really easy to work with. Our game now doesn't use balls but stores the colors on the field so we have less objects to work with and thus easier to deep copy which we needed for the available moves method. We could not do the random moves on the board we use for the game because that would break the game. We needed to deep copy our whole board because if we did not do it that way we would just create a reference to the board we use in the game which will make the same changes we apply to the so called "copy".

If we would do the project again we would definitely make the starting classes like field and board as simple as can be to save a lot of time in the long run and avoid problems like we had in this project. What we would also change is where we make moves on the board which is currently on the game class. This made it hard for us to make an AI player because we had to work around the fact that the moves didn't happen in the player but in the game class. To change this next time we need to make it in the player interface with the move method and after that let both player and AI player class implement the move method but in another way, which is way simpler than what we have now. We could not do this in the project because when we came to the AI part we already made the move methods in board and game respectively and we didn't have much time to change it around in time because we still had to implement the server correctly.

The parts that really worked well were how we decided to check for adjacent colors and how to make moves on the board. We did this by first getting a row or column, sorting it so that the field with the highest number or character was the latest and the field with lowest number or character would be first. After that it basically checks every row and column if there are fields next to each other with the same color if that is the case it puts them in a list for the player. We got this in the first try and never got more problems with it down the line.

5. System tests

During the implementation phase we always have to test everything that we are doing in order to make sure everything works as expected. We do have some test classes, but only for the model classes, they have passed all the tests successfully. In addition, we have tested a lot of methods regarding the rules of the game and the board manually, using the output from the console to understand if it is correct or not.

We were testing the server and client locally first and after that we tried to connect to the reference server as well. The server and client could connect to each other, but we had a problem with the getMap() method that we could not fix resulting in an empty board every time we called it on the server or client.

6. Overall testing strategy

We mainly tested the offline implementation in the main method and with junit testing. What we did for example in the main method was creating a board and making moves on it to check if first of all the moves were actually being made and if the isAdjacent method actually works. We also manually made sure if the start board was actually valid by printing the board on main. When those things all worked we moved to testing if the adjacent fields actually got changed to empty and correctly stored to the player. When all those things worked after testing them we moved to the junit testing, we tested the biggest methods we used in the board and game classes. With the junit tests we could test the game in a better way because not everything is easily detectable using the main method. With those junit tests we found some dumb mistakes we made and fixed them. By testing the 2 biggest classes of our offline implementation we know that our game is working properly and is stable.

7. Reflection on process

Abdel: I believe that the communication was good between us and that we have worked quite hard on this project to make it work. However there were some problems we have encountered. Firstly I believe that we have underestimated the initial planning we needed to do before we started programming. This is also why I think we encountered the restructuring problem and some rushed design choices. I would definitely recommend myself to put a lot more effort in the design phase than I did before. Also making a clear plan is something that I noticed is more important than I thought. Because our plan about the project wasn’t really clear and so we could not really work efficiently on the project a lot of the time. One more thing i also noticed was that we didn’t really use the TA’s as much as other groups which is a shame. Because a lot of the time they can really help you progress further on the project if you are stuck. For the project resit we still had some problems to get the client server working. We had to rewrite a lot of code and combining this with the other modules was a hard task.

Albina: The communication between me and my partner was good. We did have some troubles implementing because we both did not see on time requirements for the project and started implementing the things that differed from protocol. In this way we lost some time which would have been useful to use on implementation of some additional things. While planning we did not consider the client and server would take that much time. So we had to hurry up in the end with debugging and changing things. I think if we could do the project one more time we would consider all of these aspects to never let it happen again.