Programming project report

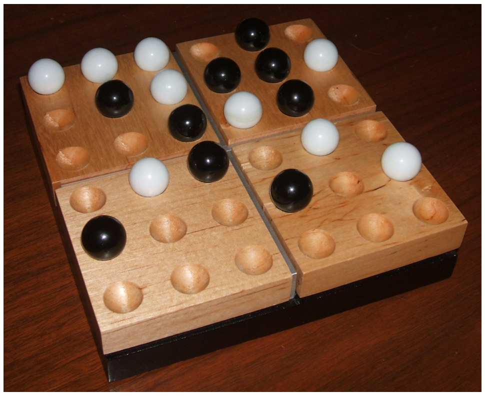
TCS Module 2

02-02-2022

Titas Lukaitis (s2749319)
Beerd van de Streek (s2840952)

Table of contents

Introduction



Pentago is a game that has to be played by two players at the same time. The rules are fairly simple. One player starts the game and places a black/white marble in one of the 36 positions. After this, you have to rotate one of the four so-called quadrants of the board. Then it's the next player's turn. The goal is to get 5 marbles of your own in a row. A Pentago board could look something like the object in the image on the right, but of course there are many alternatives looking a bit different.

Now I hear you think: "What does this have to do with computer science?". The actual project consists of two different parts, a server program and a client program. With these programs combined, two users should be able to play a game of Pentago against each other, or optionally against a bot. With this project, the goal is to bring the taught subjects from module 2 "Software Systems" into practice by using code conventions, writing tests, using git, and more. All in one project. This involves implementing the game logic, writing a client and server, making sure these all work together, etc.

Justification of minimal requirements

Below is a list of minimal requirements as listed in the project manual

For information on how to run different parts of the program please refer to the README

1. To play a game with our client on the reference server, you can run our client and then fill in the details of the reference server, or just use the `--preset` command line argument. This argument automatically chooses the reference server.

   In order to play a game with the reference client on our server, follow the instructions in the README on how to start the server, and then connect the reference clients to it. The server only asks which port it should run on, after which it does not require any user input anymore.

2. To play as a human player with the client, it should be started with the `--human` flag. Optionally, you can also start it without any flags and just answer the questions the client will ask you.

3. The client can also play automatically. This so-called AI player has two different levels of difficulty: naive and smart. The first one can be started by using the `--naive-bot` flag, the second one with the `--smart-bot` flag, as described in the project README. It is also possible to omit the flags, and enter “N” or “S” for a naive or smart bot respectively after the client has started.

4. When starting the server, it will ask for the port at which it has to listen for new connections. In the case the port bind fails, for example when it is already in use, or the user does not have permissions to use the specific port, it will ask for a new port.

5. When you start the client, it will ask if you want to use the preset values, or use custom values. When you choose for the preset values, the client will automatically fill in the details of the reference server. This makes it very easy to test quickly if needed. In the case you choose to enter custom values, the client will first ask for a server address and then for the port the server listens on.

6. When you connect as a human player and you have successfully logged, queued and started a game with another player, you can type the “hint” command at any time. This command will output something like this:

```
Place: A8
Rotate: AR
```

7. The client can also automatically play. To start this, run the command with the `--smart-ai` flag. This automatically selects the smart AI as player. After you've entered the server details and a name, the bot should automatically queue up and start playing the game against your opponent. Whenever the game is finished, it outputs the results and queue up again. To stop the bot, you can type "quit" at any time.

8. When starting the TUI without any flags, you will be able to set the AI difficulty level via the TUI itself. Once you start the TUI, you will see something like this:

   ```
   (H)uman, (B)ot or (S)mart bot
   ```

   It is now possible to enter "B" or "S" depending on what difficulty you want the AI player to be

9. Whenever a game has finished, the client will first output the results. The user can then type any command, like "list" to list all players, and "queue" to queue up again without having to restart the client or login again.

10. Since not only our client and server, but also the reference client and server respect the protocol, they will all be able to communicate with each other. The protocol describes how a command from client to server and vice versa should look like.

11. Whenever a client gets disconnected from the server for any reason, it will not crash. Instead, when it notices the disconnect, it will ask the user if they want to retry to establish a connection. When the user agrees, it will retry. Otherwise, the client will gracefully terminate.

12. When a client disconnects midway during a game, the other players will be informed by the server. The server will send a message to the client that is still connected.

    That should look something like this

```
Player 1 Won the game by disconnect!
```

After this, the player that is still connected can queue up again

Explanation realized program design

List of responsibilities

Each class has its own responsibilities. Below is a list of all classes, along with their responsibilities

Client

PentagoClient

The `PentagoClient` class is the main class of the `java.pentago.client` package. This class is responsible for running the `main()` method. This main method starts a `Scanner` for `System.in` and asks the user for some information that is needed to connect to the server, such as where the server is running, the type of player you want to connect with, or the username.

After the connection has been made. The `PentagoClient` class will listen for user commands from `System.in`. For example, if the user types “quit”, the program will exit gracefully. Also, when a user types “help”, this class will print out a help message.

This class is also responsible for instantiating a new `Networking` object, which will receive messages from the server. This class is described below.

Player

The `Player` class is an `abstract` class. This means that you can’t instantiate such a class. To make use of this class, you can create other classes that `extend` this class. Such classes are `Bot` and `Human`.

This class has fields for the name and the mark of the player, which are set at the constructor. Of course appropriate getters and setters are available as well. This class also has an abstract method, `determineMove`, which can be implemented in the different classes that `extend` this one.

Bot

The `Bot` class overrides the `determineMove` method, but inherits all other methods from `Player`. Also, this class has an additional field, which holds the `Strategy` that will be used for this bot. This class is described below.

Human

This class also `extends` `Player`, so all methods are inherited.

Networking

The `Networking` class handles the connection between the client and server. It implements the `Network` interface. This class can connect to the server, send messages and also close the connection.

Listener

This class is used to receive messages from the socket, and then parse them. For all available methods, this class starts the appropriate actions, depending on what it receives from the server.

Game

The `Game` class keeps track of the status of the game locally. It uses the `game_logic` package to create a board of the game, and it is used by the `Listener` class, which sets the board values based on what moves it receives from the server.

NaiveStrategy

This class overrides all methods from the `Strategy` interface. The `determineMove` method is used to get a random move that the client can send to the server.

SmartStrategy

This class is essentially the same as the one above, but of course the `determineMove` method is a bit more complicated. This one doesn’t just select a random move, but also checks if there is a move with which the client can instantly win. It can also prevent the other player from winning sometimes.

Mark

Technically, this is not a class, but it is natural to also name this enumeration. This `Mark` enumeration gives the option to choose `EMPTY`, `BLACK` or `WHITE`. It also provides a way of representing a mark in String format, by providing a `toString` method.

Network

This too is technically not a class, but an interface instead. This interface is implemented by the `Networking` class.

Game logic

The game logic does not have a main method. The reason that the game logic is placed in a different method is because its functionality is used by both the client and the server.

Board

This class is used to create, edit and print out a board. This helps the client and server to keep track of the state of the board.

CommandParser

This class is used to translate the coordinates that are used in the `Board` class to the coordinates used in the protocol. More about this can be read in the reflection on the realized design.

Server

Server

The main method of the `java.pentago.server` package is included in the `Server` class. This main method is responsible for asking the user what port the server should run on upon starting the program. After this, it will start a new instance of the `SimplePentagoServer` class. This class is described below.

SimplePentagoServer

This class is responsible for starting a new `ServerSocket` and calling the `accept` method. This will make a server that runs on the earlier specified port.

As soon as a client connects to the server. This class will put its information into a `List` containing all connected clients.

When 2 or more connections have indicated that they want to play a game by entering the queue, this class will create a new game with the corresponding players.

AcceptConnection

The `AcceptConnection` class will make a new `ClientHandler` instance on a new thread every time a new client is connected.

ClientHandler

This class will handle a single client connection. It implements `Runnable`, so it can be run in a separate thread. When a client sends commands, this class receives and parses them and then updates the local game state accordingly.

Game

The game class in the server package is used to keep track of the game locally. This class uses the `game_logic` package to instantiate a new `Board` with two `Player` objects. It sends the players messages containing the other player's move, or if a game has ended for example.

PentagoServer

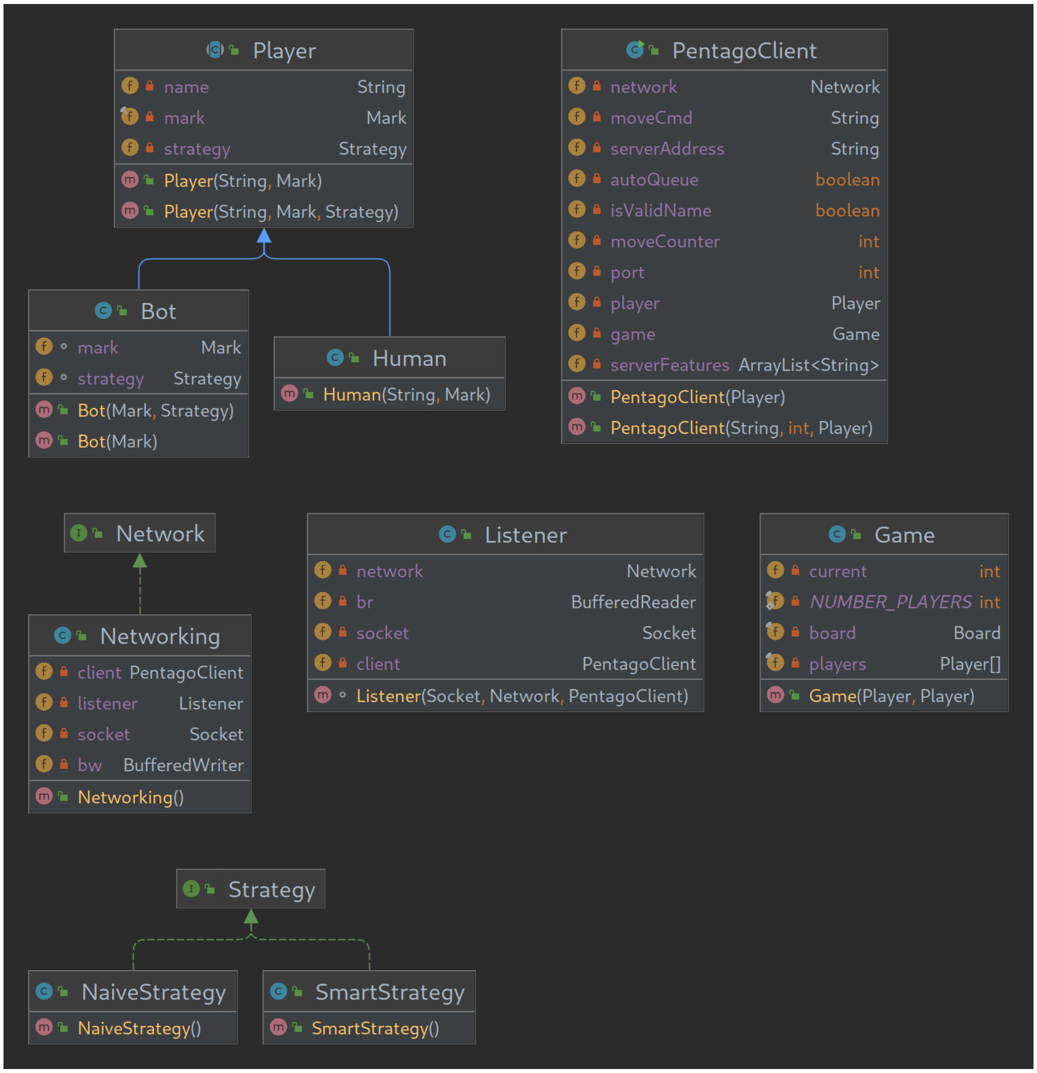
This is not a class, but an interface. This interface is implemented by SimplePentagoServer. It makes sense to use an interface here, since this can then be easily implemented in case a new type of server will be added (for example one that supports Pentago games with four players).

Explanation of class structure

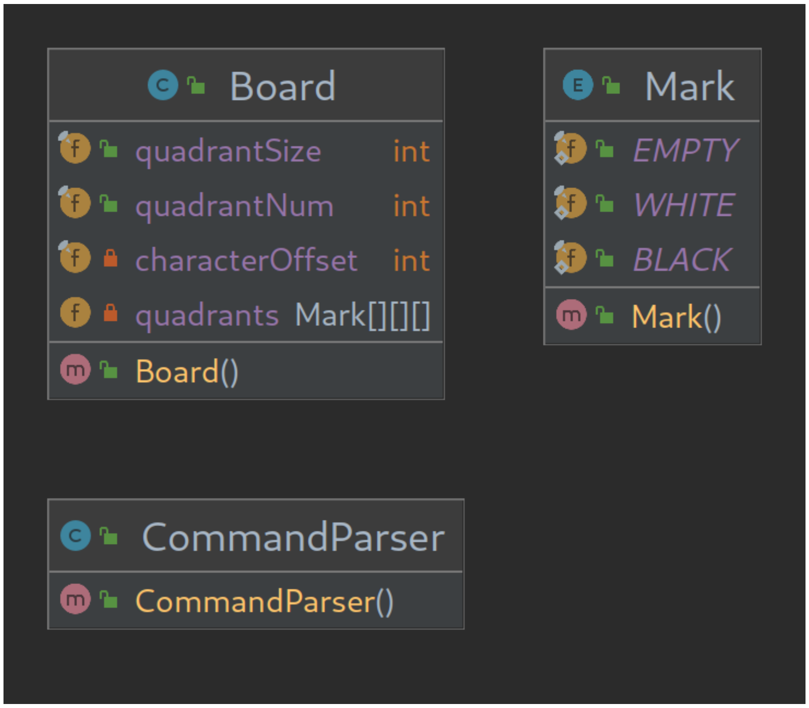
Below you find three different class diagrams. These diagrams show the three different packages that are included in our project. After these diagrams, there is an explanation of why they are made like they are.

Class diagrams

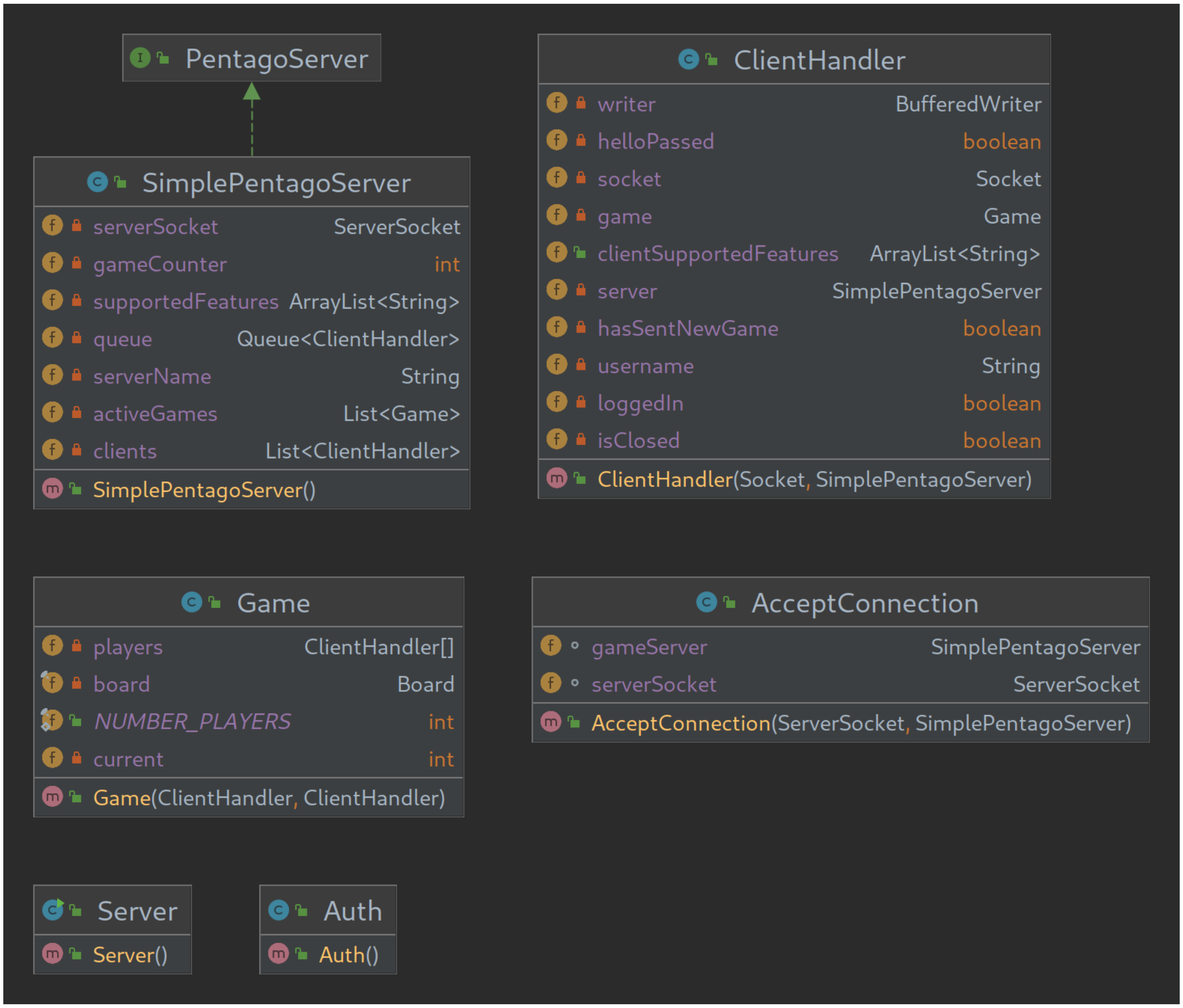
The following diagrams have been made as big as possible to make them better visible. An explanation of why the structure has been realized like this, can be found on the pages following the diagrams.



Class diagram for `java.pentago.client`



Class diagram for `java.pentago.game_logic`


PentagoServer
SimplePentagoServer
serverSocket ServerSocket
gameCounter int
supportedFeatures ArrayList<String>
queue Queue<ClientHandler>
serverName String
activeGames List<Game>
clients List<ClientHandler>
SimplePentagoServer()
ClientHandler
writer BufferedWriter
helloPassed boolean
socket Socket
game Game
clientSupportedFeatures ArrayList<String>
server SimplePentagoServer
hasSentNewGame boolean
username String
loggedIn boolean
isClosed boolean
ClientHandler(Socket, SimplePentagoServer)
Game
players ClientHandler[]
board Board
NUMBER_PLAYERS int
current int
Game(ClientHandler, ClientHandler)
AcceptConnection
gameServer SimplePentagoServer
serverSocket ServerSocket
AcceptConnection(ServerSocket, SimplePentagoServer)
Server
Server()
Auth
Auth()

Class diagram for `java.pentago.server`

Explanation

Client

The project is divided into three different packages. The first package is called `java.pentago.client`. This package should be able to run without any code from the server, so it's logical to separate them from each other.

The client package is divided into multiple classes. As described in the list of responsibilities, the `PentagoClient` class contains the code that parses the user commands. The `Listener` class is used to receive messages back. It makes sense that these are divided into two different classes, since they run asynchronously. Lastly, we have the Networking class. This class is used to send messages to the server and manage the connection.

Next, since a player can be a human, or a bot with different strategies, this class is abstract. Human and Bot both extend this class. Since a bot can have multiple strategies, this is included as a field in the class.

Server

Next, there is the `java.pentago.server` package. This is used to spin up a server that listens on a specific port. The reason that this is extracted is because of the fact that it should be able to run standalone. It makes sense to keep it separate from the client code in this case.

The class `Server` has the main method with which you can run the program. This main method creates a new `SimplePentagoServer` when it knows the port. This class is responsible for keeping track of the clients, matching clients, etc. When a new client connects, this class creates a new `AcceptConnection` instance, which in turn will run a `ClientHandler` thread. The reason this is done is because of the ability of the server to handle multiple clients. This requires the clients to all be connected to a different thread. The `ClientHandler` keeps track of the game for the specific client.

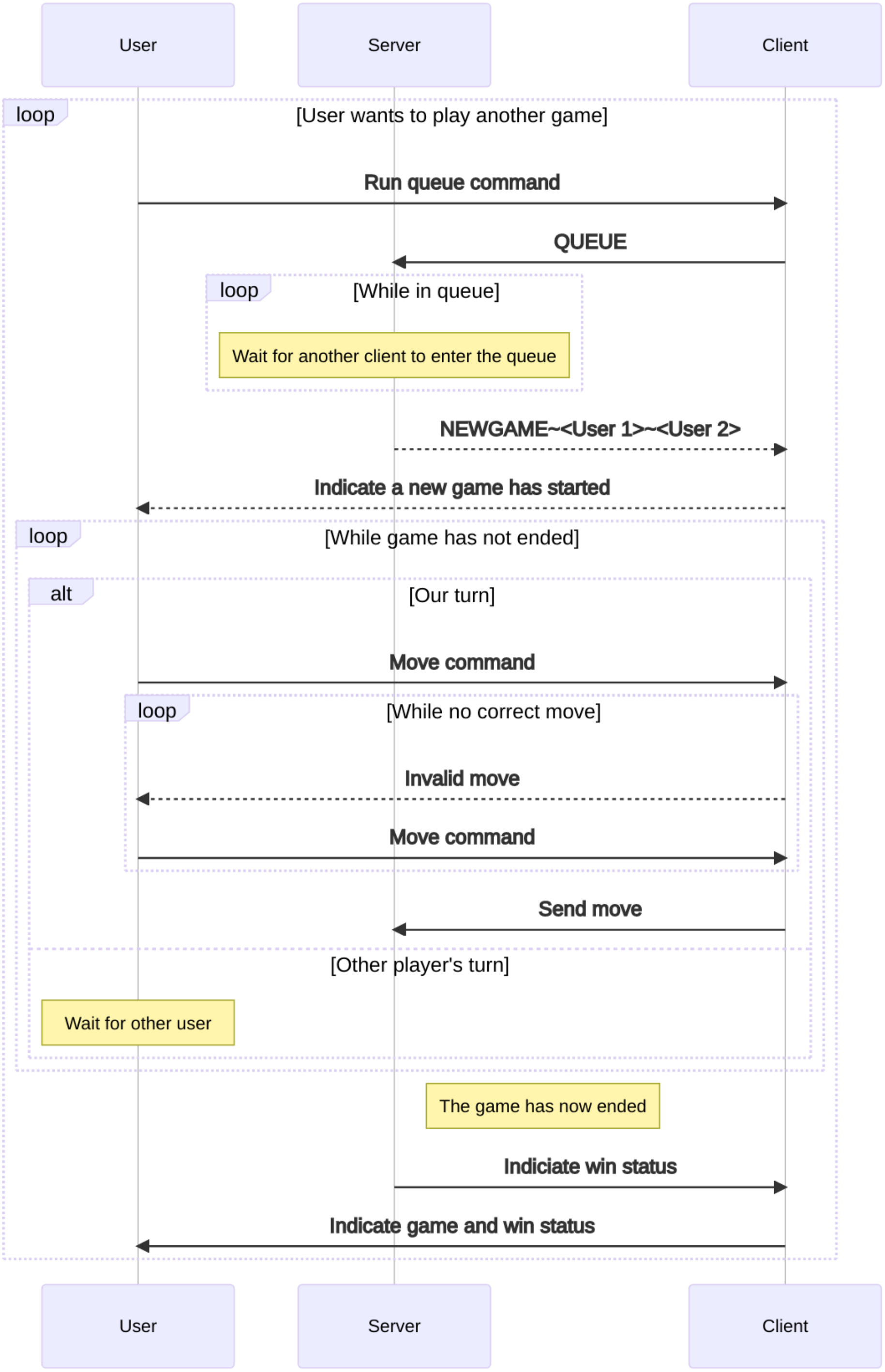
Game logic

Lastly, we have the `java.pentago.game_logic` package. This package contains the code to generate a board and apply the placing of marks, rotations, etc. The reason that this package is kept separate from the client and server code is because it is used by both the client and the server. Otherwise, a lot of the code that you find here would’ve been duplicated.

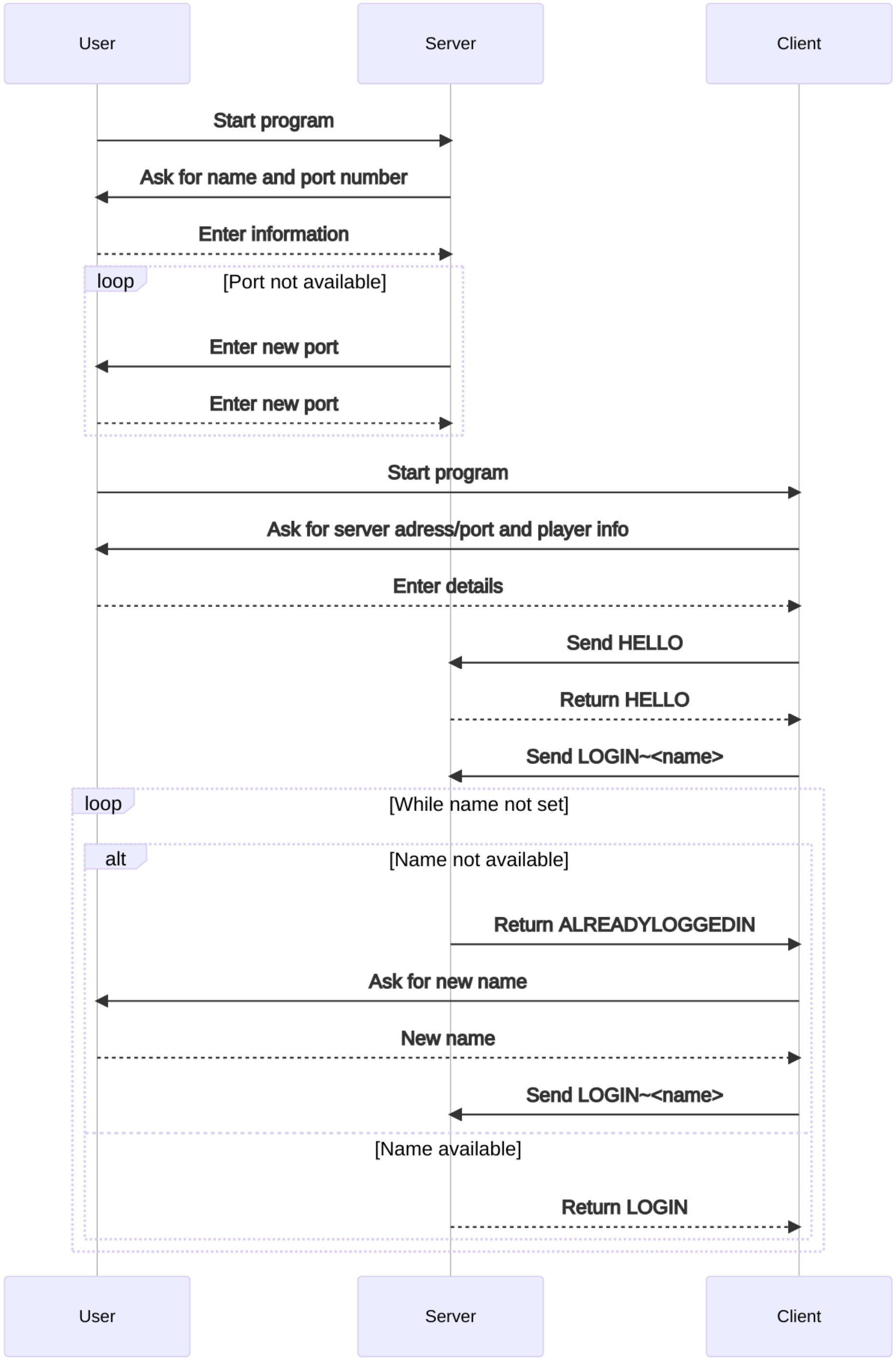
The game logic really only has one very important class. This class is used to create, update, print out boards, etc.

Sequence diagrams

Below you find two different sequence diagrams. These have been enlarged to make them easier to view. On the pages following the diagrams, you'll find an explanation.



Sequence diagram 1: Playing a game



Sequence diagram 2: Establishing a connection between client and server

Sequence diagram 1

The first diagram shows the communication between the different components when a game is being played.

The whole process starts once a user sends the QUEUE command. This makes the server wait for another user that uses this command, and then starts the game by sending NEWGAME back. After this, the client indicates to the user that a new game has been started.

Now the users can send moves in turns. Hence the loop. When it is not our turn, we naturally have to wait. In the case it is our turn, we can type moves. These moves will be validated by the client before sending to the server. When they are not valid, the user should try again. When a move is valid, the command is sent to the server, and now it’s the other player’s turn. This repeats until the game has ended.

When the game has finished, the users are informed about the game status. The client outputs the board status and the user that has won this round. The outer loop repeats, so a user can send the QUEUE command again now and start a new game.

Sequence diagram 2

The second sequence diagram describes the process of establishing a connection between a server and client.

First of all, the user starts a server process. For example, this can be done in a terminal window. The program then asks what the server name should be and at what port it should listen. In the case a user does not have permission to open this port, or the port is already in use, the server asks again. This repeats until the port can be successfully opened.

Once the server is listening on the specified port. The client program can be started. The client asks the user for information such as if the client should play as a bot, what port the server is running on, etc. After this, the client will connect to the server. Now a HELLO message will be sent to the server along with the client name. When the server has sent a HELLO back, the client can send the LOGIN command along with the username. The server will check if the user already exists on the server. If this is the case, the client receives ALREADYLOGGEDIN back, and has to send LOGIN again with a different name. If the name does not yet exist, the server sends LOGIN back to the client.

After this, the client can send commands to the server, such as QUEUE, as described above.

Concurrency mechanism

Server

The server is synchronized in many ways, in the “SimplePentagoServer. Java” . Anytime a call is made to the `queue` object a `synchronized (queue) { }` is wrapped around any of the code that interacts with the queue.

Similarly any code that interacts with the `clients` list has a synchronized statement around it.

```
List<ClientHandler> clients = new ArrayList<>();
```

These are done because each time a client connects to the server they are given their own thread, meaning that we have to make sure that no client gets accidentally lost while trying to add itself into the list of connected clients. Similarly to make sure that no client that is connecting to the queue at the same time as another is lost.

One other thing that we synchronize on is in the “Game.Java” class where anytime a call is made to the `board`, this is to make sure that no functions that interact with the board are able to return accurate results that don’t overwrite other functions.

Client

In the board logic, which is used in the client, the `synchronized` keyword is used several times. When you would use the methods in the board class to set a field for example, the `this.quadrants` field is locked for other threads. This ensures that it is impossible to have to thread accidentally setting the same field at the exact same moment. This is used for `getField()`, `setField()` and `rotateQuadrant()`.

Also, a new thread is made for the `Listener` class. This is used to output lines, while also waiting for user input in the main thread at the same time. One issue with this is, that it is possible for the program to output text to the terminal while the user is typing a command. This is very annoying, since you will have to start typing the command again. If there would’ve been more time available, this could for example be fixed by a locking mechanism. This would prevent the program from outputting text while the user is typing. However, this would result in delayed chat messages for example. Another solution could be a different terminal window for the output. For this timespan, this feature was classified as *won’t have*. This class also has parts of it synchronized around a `Game` object whenever a `move` or `newgame` command is received from the server.

Reflection on design

Reflection on the initial design

The end result generally complies to the class diagrams that were made for the initial design looking at the structure. Of course there were some extra methods/classes introduced, or methods left out, but this was expected. An example of this could be the `CommandParser` class. From this, we can conclude that the general structure of the initial design has been very helpful, but by no means did it describe all the details that were needed to actually write the program.

Reflection and improvements on the realized design

In the current design, a Pentago board is realized as four different quadrants. Each with nine spots for marks. Initially, this seemed like the smartest way to implement this, because of the rotations. It was thought that the rotation would have been way easier if the board was separated into quadrants. However, since the rotation method eventually ended up being mostly hardcoded for every position, this did not end up making it easier. Unfortunately, it only ended up making things harder to implement. Every time a message will be sent to or received from the server, the client has to translate it to the custom coordinates that are used. While this might make it a bit easier for the user, it did make the development process a lot more complicated.

Improvements to the design process

As described above at the reflection on the initial design, the initial design did not include all details that were needed to actually write the program. An improvement to the design process could have been spending more time on the actual design instead of writing the program. Spending more time designing could have prevented a lot of design mistakes that only showed up after they had mostly been implemented. Meaning that a lot of time has been lost at implementing things that would end up never working.

Another thing that could have improved the design process, is not splitting up the design into three different components: the client, server and game logic. With this, you would be able to focus on the program as a whole better, rather than focussing on one thing at a time. This will evidently result in a better design, since we then probably would’ve realized that a different coordination system for the board is not as practical as we thought for example.

System tests

Not all tests can easily be automated. With the system tests, the program can be tested as a whole. Any developer without previous knowledge of this specific software should be able to run the system tests.

Before going through the system tests, it is important to make sure that you have a local copy of the latest version of the system. The latest version can be built from source (see the README). Make sure that you're on the main branch and you've checked out the latest commit. To make things easier, you can also download the precompiled .jar file from the GitLab CI pipelines here. This link always contains the latest build that passed all tests.

Steps to go through a system test:

Overall testing

1. Follow the instructions in the *README* to build the project, or just download the jar from the latest GitLab pipeline.
2. Follow the instructions in the *README* to run the client part of the test. For example, you can run `java -cp <output-file>.jar pentago.client.PentagoClient` in a command prompt to run the client.

   Expected output:

   ```
   (H)uman, (B)ot or (S)mart bot
   ```

3. Here you can choose what kind of player the client should represent. When you select a *Human* player, you will have to make the moves yourself. You also have the option to select one of the different *Bots*. These will automatically make moves when needed. As the names imply, they each have their own level of difficulty.

4. For testing purposes, it is easy to use one of the bots, as you don't have to manually play a whole game. So press `B` and then press `enter`.

   Expected output:

   ```
   B
   Player: Naive bot
   (P)reset or (C)ustom?
   ```

5. For now, you can press `P` and then `enter`. This will connect the client to the reference server automatically. Then enter a username.

   Expected output:

```
(P)reset or (C)ustom?
P
Username:
Player 1
Connected
Server Name:
        PenTomGo
Supported Features:
        CRYPT
        AUTH
        RANK
        CHAT
Logged In
WHISPER
        FROM: PenTomGo
        MESSAGE: Welcome to the PenTomGo server!
WHISPER
        FROM: PenTomGo
        MESSAGE: Say "!status" or "!players" or "!player someName"!
```

6. This indicates that we are connected to the server. Now you can type `help` in order to see the list of supported commands. This should look something like this:

```
help
Commands:
list
        Lists all users currently connected to the server
queue
        Queues up for a new game
place [A-D][0-8]
        Places a piece down a piece on the position
rotate [A-D][L|R]
        Rotates a quadrant in a specific direction
```

```
ping
        Pings the server to see if its still alive
chat [message]
        sends a message to everyone on the server
whisper [user] [message]
        sends a message to a specific person
help
        Displays this help message
hint
        Displays a possible move
show
        Shows the current state of the board
autoqueue
        Toggles the autoqueue
quit
        quits out of the program
```

7. Now, to make sure we are properly connected to the server, type `ping`. This command will send a ping command to the server and will then, if the server is still alive, receive `pong` back.

   Expected output:

   ```
   Server up!
   ```

8. To queue up for a game, enter the `queue` command. This command will look for other players, and start a new game once found. Note: if the matchmaking takes a long time, no one else is in the queue. You might want to spin up another client instance to play against.

   Expected output:

   ```
   New Game:
           Player 1: player1
           Player 2: player2
   ```

9. At any time during the game, you can use the `show` command. This will output the board. This could, depending on the status of the game, look something like this:

```
                                     A           B
-------------------------     -------------------------
| ● | ● | ● | ○ |   | ○ |     | 0 | 1 | 2 | 0 | 1 | 2 |
-------------------------     -------------------------
| ● | ● | ○ | ○ | ● |   |     | 3 | 4 | 5 | 3 | 4 | 5 |
-------------------------     -------------------------
| ● | ● | ● | ○ |   |   |     | 6 | 7 | 8 | 6 | 7 | 8 |
-------------------------     -------------------------
|   | ○ | ○ | ○ | ● | ○ |     | 0 | 1 | 2 | 0 | 1 | 2 |
-------------------------     -------------------------
|   | ○ |   | ● |   |   |     | 3 | 4 | 5 | 3 | 4 | 5 |
-------------------------     -------------------------
|   |   |   |   | ○ |   |     | 6 | 7 | 8 | 6 | 7 | 8 |
-------------------------     -------------------------
                                     C           D
```

As you can see on the right, the show command will also output some help to the user by showing the possible board positions and quadrant names.

10. To quit the program, enter `quit`.

    Expected output:

    ```
    Quitting!
    ```

11. The steps 1-10 can be repeated with the different bots, a human player, a different server, etc.

Testing a client disconnect

1. Start a server and two clients

   Follow the instructions in the *README* to build the project, or just download the jar from the latest GitLab pipeline.

   First, follow the instructions in the README to run the server part. Enter port 8080 when asked.

   After this, follow the instructions in the *README* to run the client part. Do this in two different terminal windows. Run this with the `--smart-bot` flag. Enter “C” to connect to a custom server and enter `0.0.0.0` as server address and `8080` as port. Now choose a different name for both.

You should now have 1 server and 2 clients running

Expected output:

Server	Client 1
What port should the server listen on (0 for random)? 8080 Server: 0.0.0.0/0.0.0.0:8080 8080	Player: Smart bot (P)reset or (C)ustom? C Server Address: 0.0.0.0 Server Port: 8080 Username: Name 1
	Client 2
	Player: Smart bot (P)reset or (C)ustom? C Server Address: 0.0.0.0 Server Port: 8080 Username: Name 2

2. The clients should start playing games against each other. To test the behavior when one of them crashes/disconnects, we can kill one of them. First, type “quit” at one of the clients. This should disconnect the client.

   Expected output:

Server	Client 1
What port should the server	Name 1 Won the game by

listen on (0 for random)? 8080 Server: 0.0.0.0/0.0.0.0:8080 8080	disconnect!
	Client 2
	Quit Quitting!

As you can see, the server should send a win message to the first client whenever the second one disconnects.

3. To test the same thing, but instead of gracefully quitting you kill the client, just repeat steps 1 and 2. Instead of typing “quit”, you can now kill one of the clients by pressing `ctrl-c` in the terminal. The output should be the same.

Testing a server disconnect

1. Connect a client to our own server, by following the first step of [Testing a client disconnect](#) again. This should result in a server with two connected clients.

2. To kill the server, you can simply press stop in IntelliJ, or press `ctrl-c` in your command line.

   Expected output on the client:

```
It looks like the pipe to the server is closed...
Do you want to reconnect? Y/n
```

3. You can now make the client try to reconnect by pressing “Y”. This should of course result in an error, since the server does not exist anymore.
   Expected output:

```
Reconnecting...
ERR: bad connection
```

The program should then exit

You can also just type “n”. The client will then quit without trying to reconnect.

Testing the server

To test the server, we use telnet.

1. Start the server as described in the README and enter port 8080.

   Expected output:

   ```
   What port should the server listen on (0 for random)?
   8080
   Server: 0.0.0.0/0.0.0.0:8080
   8080
   ```

2. In a new terminal window, type `telnet`. This starts the telnet command.

   Now type `open 0.0.0.0 8080`

   (`0.0.0.0` stands for your local ip address. The server will output a different address if this is different for you)

   Expected output:

   ```
   Trying 0.0.0.0...
   Connected to 0.0.0.0.
   ```

3. Send the following to the server:

   ```
   HELLO~<Client name>~CHAT
   ```

   Expected output:

   ```
   HELLO~TestServer~CHAT
   ```

4. Send the following to the server:

   ```
   LOGIN~<Username>~CHAT
   ```

Expected output:

```
LOGIN
```

5. Now you can test several different commands

   Type: `PING`

   Expected output:

   ```
   PONG
   ```

6. Also, we should test unsupported commands, like `THISCOMMANDDOESNOTEXIST`

   Expected output:

   ```
   ERROR~ERROR~Unrecognized command: THISCOMMANDDOESNOTEXIST
   Connection closed by foreign host.
   ```

   The first line in the expected output is a message we receive back from the server, after which it will close the connection. The last line is an indication from telnet that the program has dropped the connection with the server.

Overall testing strategy

1. Overview and scope

The project is a Pentago game that consists of a client and server. It allows users to find and play against other players.

The main scope of these tests is the game logic

2. Test approach

Unit tests are written before starting the development on a function. Once a function is completed, it should pass all unit tests. It should then be handed over to another team member for them to test.

Once a section of code is complete integration tests will occur. The integration tests will be handled by 1 team member which will then report any errors that were given and the team member that wrote the function in which the errors occurred will then be tasked with solving the problem. This cycle repeats until there are no errors found.

Once the whole system is assembled then the whole system will be tested by the team leader to make sure that there are no errors.

The team member that wrote the function should be the first one to do unit testing. If a function has an error in it, then the team member that wrote the function is assigned to fix it.
In integration testing the team member with the least amount of work in the subset of function is assigned to test it. If the tests pass then the team member will sign off on it passing and leave it be.

All these steps ensure that each method is very well tested separately. To make sure that the methods combined also make sense, we run a mock game 100 times. These tests look for an empty spot on the board and places a black mark first, then a white mark, etc. Then it rotates a random quadrant in a random direction. This will most likely result in 100 different games. Once they all run without errors, we can conclude that the methods work nicely together.

3. Test environment

The program(s) should be tested on machines with specifically Java 11 installed. This is because it’s also developed with this version of Java. The testing module that is used is Jupiter JUnit version 5.8.2.

4. Testing tools

First of all, JUnit is used as a testing tool. We run these JUnit tests with a Maven plugin inside of our IDE, but also on a GitLab pipeline. This allows us to prevent merging before all the tests pass.

Also, CheckStyle is used, so we are sure that we conform to common coding conventions. This is also forced by a GitLab pipeline.

5. Release control

New builds of the server and client should always be rigorously tested before deploying. For this, the reference server and client can be used. Our client should work with both the reference server and our own server, and our server should work with the reference client and our own client. If this is all true, we are sure that we conform to the protocol. To make sure no developer can

6. Risk analysis

List of foreseen risks:

1. Server down

   In the case the server is not responding to any requests anymore, the client should not fail. The client should be able to handle this, and exit gracefully, outputting an error message to the user so they know what's going on.

2. Server rejects a move

   Of course, the client should try to prevent sending illegal moves to the server in the first case, but when a server rejects a move, the client should be able to handle this. It should ask the user again to pick a valid move.

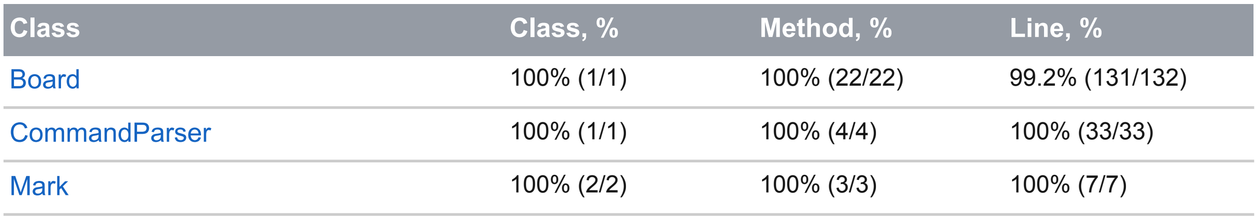
7. Review and approvals

Team members are responsible for Unit tests. Other team members make sure that those unit tests run and are signed off.

Team members then conduct integration tests and sign off on the results.

The whole team will then do system testing and sign off on the results.

Coverage metrics



Class	Class, %	Method, %	Line, %
Board	100% (1/1)	100% (22/22)	99.2% (131/132)
CommandParser	100% (1/1)	100% (4/4)	100% (33/33)
Mark	100% (2/2)	100% (3/3)	100% (7/7)

Table 1. *Coverage metrics as of commit `c4d2b4c5` on branch main.*

These coverage metrics can be generated using IntelliJ. First, right click on the test folder (game_logic), then click *More Run/Debug → Run 'tests in 'pentago.game_logic'' with Coverage*. This will run all the tests and then open a dialog on the right with the results.

As described above, for every method that will be written, the developer should first write test functions for it. From this table, one can conclude that the game logic code is very well covered by the tests. The client and server code are both covered by the system tests.

Reflection on process

Titas

I believe that me and Beerd worked very well together and there were no problems collaborating with each other.

Collaboration

I think that our work process was very smooth and we were able to efficiently assign all the tasks so that we could complete the project. Although we did not do too much pair programming, we only did it when one of us was having issues that we could not solve ourselves.

Planning

We initially started a trello to keep track of what we were supposed to do, but starting one and actively keeping it up are not the same thing. For example when we found an issue in the code we found it easier to instead put it into gitlab issues rather than on trello, since we were already on gitlab. Rather we used trello to keep track of major deadlines e.g. when submissions are going to happen. I believe some of our deadlines were on the optimistic side, because we didn’t really account for things like unexpected issues in the code or features taking longer to make than expected.

I think that we weren’t really sure how exactly we wanted to keep track of issues and deadlines and all that when we started on the project, therefore in the future I'd like to immediately set the workflow and how things should work in the future.

Another workflow that we added was CI/CD to our project. We were initially not thinking about adding this to our project but we wanted to try these new technologies out and they were really worth the effort. We had 2 pipelines, 1 that would check our project's checkstyle to make sure that our code was written correctly and another one that used maven to automate unit testing. I will definitely bring this technology to my next projects.

Beerd

Since this is the first software development project I have done from start to finish with someone else, it is very natural that not all things went entirely as predicted. The teamwork itself went very smoothly.

Collaboration

Regarding the collaboration, we really did not have any problems in my opinion. Titas and I were both actively involved in working on the project every day we were expected to do so. There were no difficulties in the communication, so if either one of us had any questions, we could easily help each other out.

Also, in our GitLab repository, we configured that we could not just merge a branch into the main branch. This means that the other person always has to approve the merge request before you're allowed to merge. This ensures that we're both always up to date on the other person's progress, and also gives both of us the chance to correct mistakes before placing the commits on the main branch.

Planning

To make sure that we both had up to date information regarding our planning, we decided to use Trello. With Trello you can easily track the status of a process (e.g. todo, doing, done). Unfortunately, having good timing management does not imply that everyone will definitely stick to the planning, or that everything goes well.

For example, the report for this project took more time than we had anticipated. Since we naturally planned this to work on at the very end, this gave some problems. This could have been avoided if we had started working on the report a bit earlier, for example, in parallel with the actual programming of the system.

Another thing we have overlooked, was the midway submission. When planning the project, we planned on trying to finish the client before the midway submission. When the deadline was coming closer however, we realized that there was a lot more work to it than we expected.