UNIVERSITY
OF TWENTE.

Pentago Game Report

Authors:
Krystof Mitka (s2788144)
Thai Ha Bui (s2788004)

Table of contents

Introduction

This report outlines the design and development of Pentago Game software. The program was written in Java. Note that this report was written by double degree students, who have not participated in the Design Course of Software Systems module.

Pentago is a two-player board game. The game is played on a 6x6 board divided into four 3x3 quadrants. Each player is represented by a mark of colour black or white, usually the player with black mark starts. Two players take turns and place their mark on an empty field, then choose one of the quadrants to rotate it 90 degrees clockwise or counterclockwise. A player wins by getting five of their marks in a vertical, horizontal or diagonal direction after quadrant rotation in their move. In case both players win after a quadrant rotation, the game is a draw. If the board is full and no one wins, the game is a draw.

Type:	two-player strategy board game
Mode:	with human | with computer
Victory:	have five of your marks in a row, column or diagonal next to each other
Move:	place your mark on an empty field, choose a quadrant, rotate the chosen quadrant (clockwise, counterclockwise)

System requirements

Board requirements

- The board should be able to display the game.
- The board should be able to update the game according to the given move.
- The board should be able to determine when there is a winner or when it is full.
    - The game should end when there is a player that has 5 marks in a row, column or diagonal, i.e. such player has won the game.
    - The game should end when the board is full, i.e. noone has won, it is a draw.
    - The game should end when both players have won, it is a draw.
- The board should only accept valid indexes.
    - An index is not valid when there is already a mark on that index.
    - An index is not valid when it is smaller than 0.
    - An index is not valid when it is larger than 35.

Server requirements

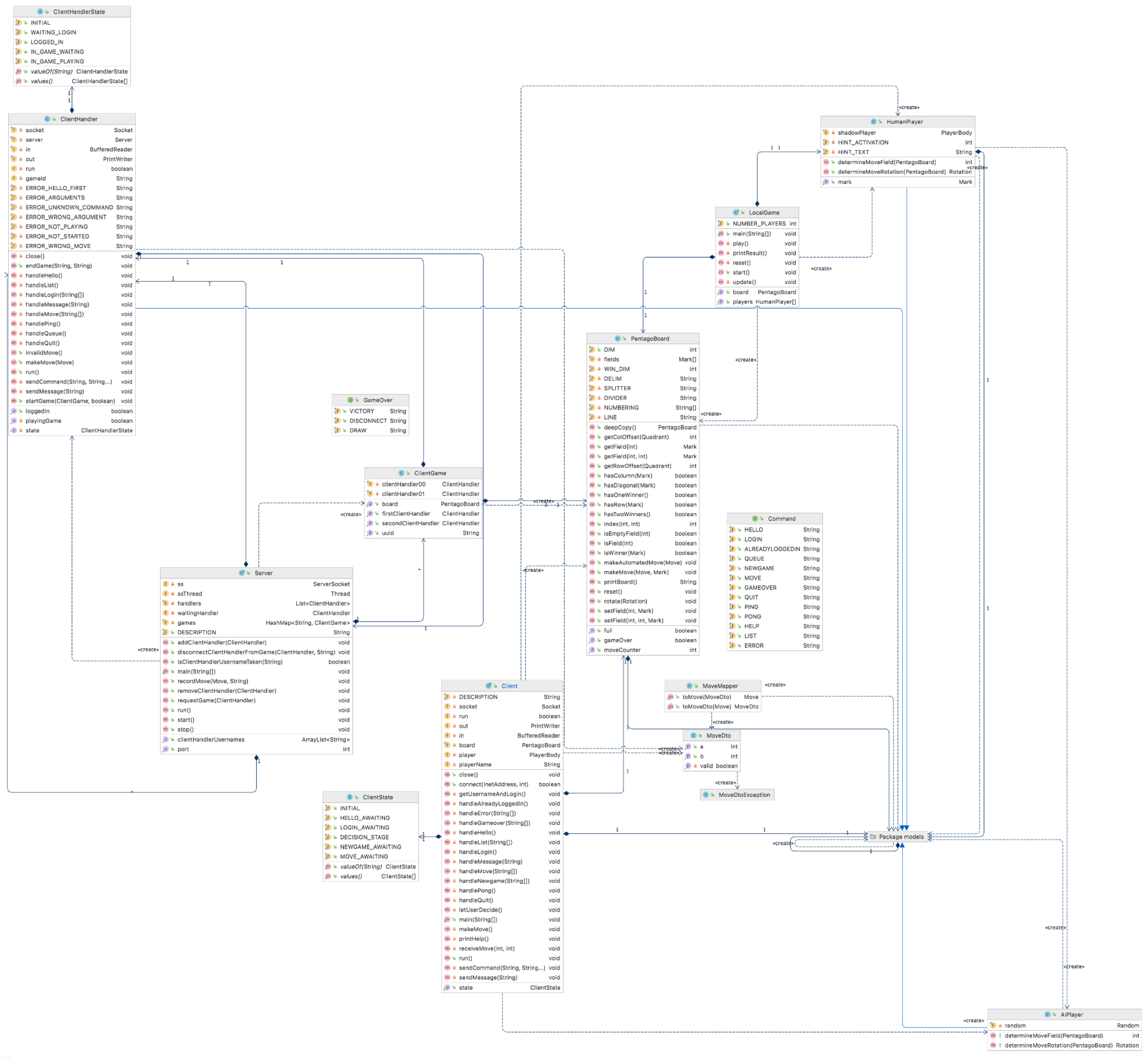
- The server should notify when the connection is established.
- The server should end the game accordingly when one of the clients disconnects.
- The server should be able to check if the client is already logged in.
    - The server should be able to check if an username exists.
- The server should check if a move is viable and act accordingly.
- The server should be thread safe and not create any race conditions.
- The server should take care of clients waiting to play the game.
    - There should not be more than two people waiting for a new game.
    - The queue should be thread safe, so overwriting does not happen.
- The server should be in control of the games
    - Each client should play at most one game simultaneously.
    - The server should send a message to the players when starting a new game as defined in the protocol.
- The server should be able to send a list of all players currently logged into the server.
- The server should be able to indicate the end of the game.
    - The server should send the appropriate result message to the clients as defined in the protocol.
    - If the game ended due to losing connection to one of the players. The winner is the player with whom the connection is still alive.
    - The server should keep both clients connected to the server, but not in the queue.

    - The clients should be able to join the queue and play a new game.

Client requirements

- The client should send all moves to the server according to the defined protocol.
- The client should be able to receive moves from different players and update the local board accordingly.

Software Design



Final UML Diagram

Responsibilities of classes

Models

Classes used to represent data.

Mark

This is an enumerated type with values EMPTY, BLACK, WHITE used to represent a value of a field or wmark of a player.

Quadrant

This is an enumerated type with values TOP_LEFT, TOP_RIGHT, BOTTOM_LEFT, BOTTEM_RIGHT used to represent the quadrant of the board.

Direction

This is an enumerated type with values CLOCKWISE, COUNTERCLOCKWISE used to represent the direction of rotation.

Rotation

This is a class used to store a Quadrant type and Direction type.

- This class is responsible for storing a quadrant and a direction of rotation.

Move

This class represents a move which is an integer as a valid field index, an object of type Quadrant and an object of type Rotation.

- This class is responsible for storing a move.

Player

This abstract class that represents a player. It has all default methods that a Player should have.

- This class is the base for default implementation of a player.

Main classes for game

PentagoBoard

This class represents the board on which the game will be played. The board consists of a field from 0 to 35 representing 36 fields. Each field is of type Mark with values either EMPTY, BLACK or WHITE

- This class can return the value of any field on the board.
- This class is responsible for checking if a field is empty.
- This class is responsible for displaying the game given moves.
- This class is responsible for setting a field given a move.
- This class is responsible for checking if any player has 5 of their marks in a row, column or diagonal.
- This class is responsible for checking if any player has won.
- This class is responsible for checking if the board is full.
- This class is responsible for checking if the game is over.

HumanPlayer

This class implements the abstract class Player and it represents a human player. It has a name (username of the player) and a mark of type Mark.

- This class is responsible for getting the move (index field, quadrant and direction of rotation) by taking values from the player.
- This class is responsible for checking if the moves are valid.

Explanation of the game logic

- How does the application move given the field index, quadrant and direction for rotation?

  The method `makeAutomatedMove(Move move)` is called. This method then triggers the method `makeMove(Move move, Mark mark)` and passes the argument mark to it based on the order of the current turn (even turn is BLACK, odd is WHITE).

  `makeMove(Move move, Mark mark)` then set the mark to the field based on the given move by calling method `setField(move.getField(), mark)` then rotate the given quadrant based on the given move by calling method `rotate(move.getRotation())`. In the end, it increases the `moveCounter` by one (so that the next move has the other Mark).

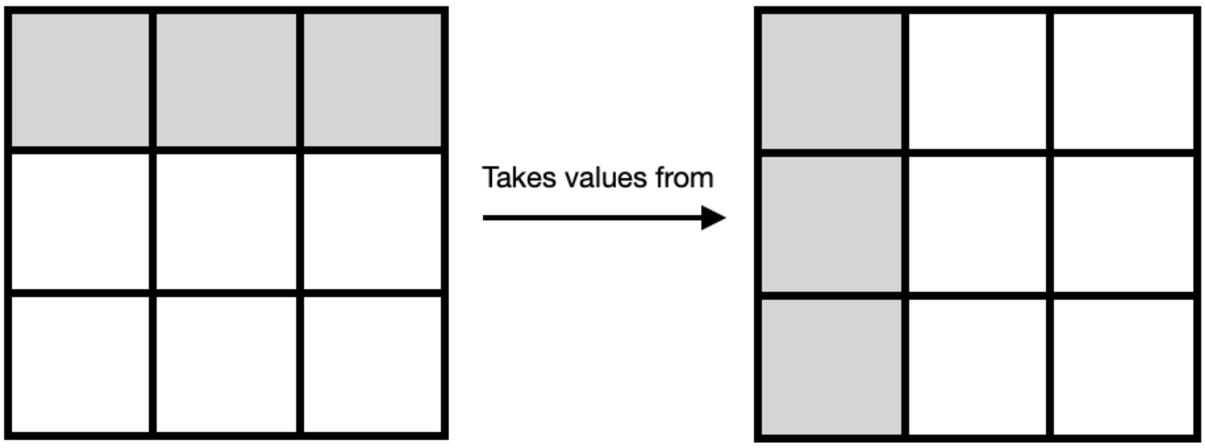
  Note: having this `makeAutomatedMove(Move move)` method is very effective, we do not need to give each player a Mark and store it. The first player always has the BLACK mark.

- How does the board rotate (how does `rotate(Rotation rotation)` work?

  This method loops through every field in that quadrant (starting from the most top left field to the right) and calculates the coordinates (row index, column index) of the field that the current field will take value of and set the value to the current field.

  It calculates the `rowOffset` and `colOffset` for the indexes based on the given quadrant (we have four quadrants in Pentago so the offsets are calculated with dimension divided by two). The `startingPointToRotate` is the index of the initial row / column position that we are going to put to the first field of that quadrant.

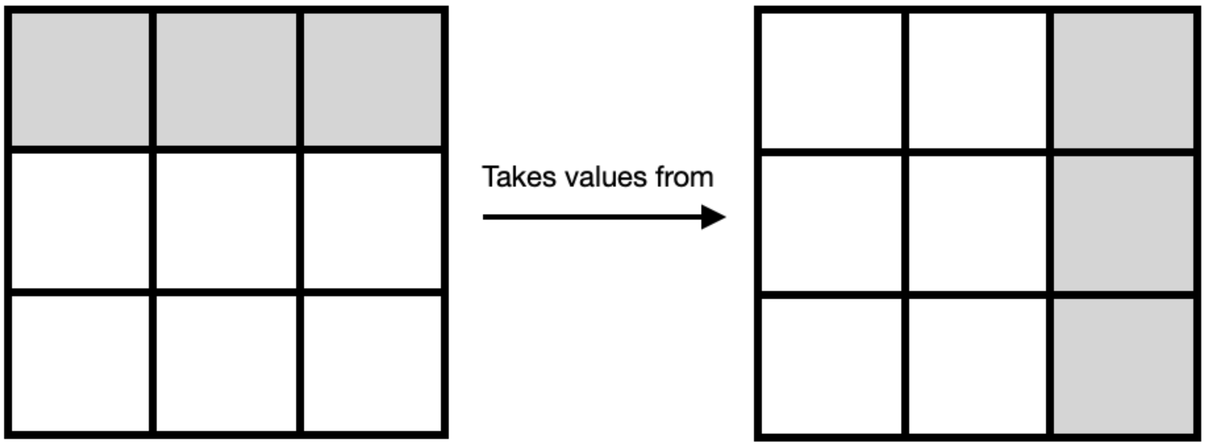
  For rotation clockwise:
  - *Set the row index of the current field to the field with index equals to `startingPointToRotate` minus the column's index*
  - *Set the column index of the current field to the field with index equals to the difference between `colOffset` and `rowOffset` added to the current row's index*



(illustration rotation clockwise)

For rotation counter-clockwise:

- *Set the row index of the current field to the field with index equals to the difference between* `rowOffset` *and* `colOffset` *added to the current column's index*
- *Set the column index of the current field to the field with index equals to* `startingPointToRotate` *minus the row's index*



(illustration rotation counter-clockwise)

Note: The calculations took us a while to come up with, we initially started with transforming the quadrant to a matrix, rotate the matrix and afterwards put it back to the quadrant. But later on, we realized that we could improve the process.

- How does the application know if a player has won?

PentagoBoard checks if there are any five same marks next to each other in a row (`hasRow(Mark m)`*), in a column (*`hasColumn(Mark m)`*) or in a diagonal (*`hasDiagonal(Mark m)`*).*

`hasRow(Mark m)` *- iterate through all fields and count the number of consecutive same marks next to each other in a row (if the number is five, the mark has a winning row).*

`hasColumn(Mark m)` *- iterate through all fields and count the number of consecutive same marks next to each other in a column (if the number is five, the mark has a winning column).*

`hasDiagonal(Mark m)` *- iterate through all diagonals (there are 6 in total) and count the number of consecutive same marks next to each other in a diagonal (if the number is five, the mark has a winning diagonal).*

`isWinner(Mark m)` *checks if a mark has any winning position using the three mentioned methods above.*

`hasOneWinner()` *checks if there is explicitly one winner (explicitly* `isWinner(Mark.BLACK)` *or* `isWinner(Mark.WHITE)`*).*

If any player has either one of these winning positions and the game is not drawn, then the mark wins.

- How does the application know if there is a draw?

  `hasTwoWinners()` *checks if there are two winners at the same time (*`isWinner(Mark.BLACK)` *and* `isWinner(Mark.WHITE)`*).*

  `isFull()` *that iterates through all fields and checks if any field is empty using another method* `isEmptyField(int i)` *that checks if the field of a given index has an enum value EMPTY. If no field is empty then it returns true (meaning the board is full).*

  If the board is full or it has two winners, then the game is drawn.

Explanation of networking

Server

Server

- *Server* waits for socket to connect, once connected server creates a new *ClientHandler* and passes on the socket number, the server itself and starts the *ClientHandler* on a new *Thread*. The *Server* also adds the *ClientHandler* to the list of client handlers.

ClientHandler

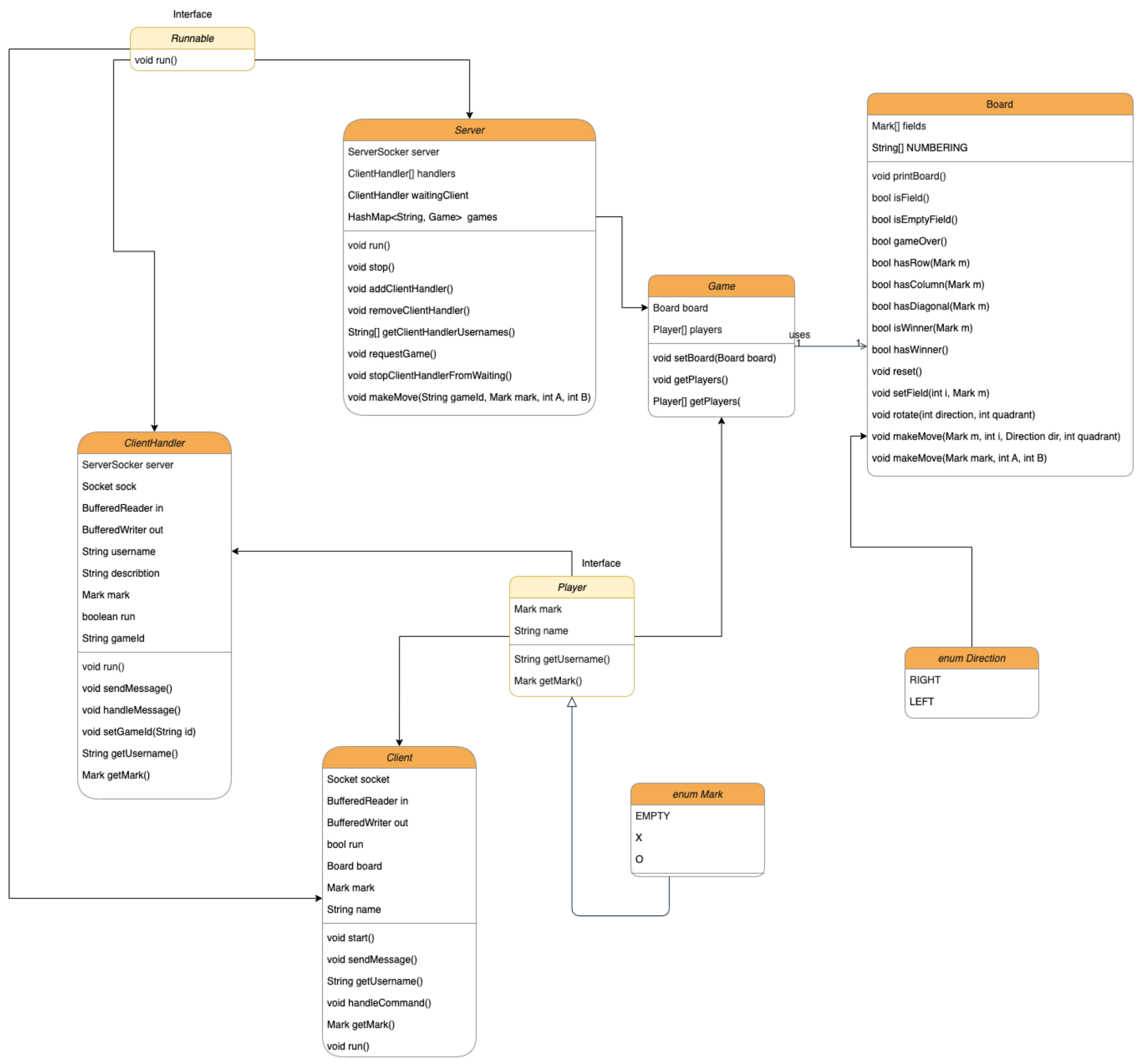
- Once client handler is initialized, it calls the

Concurrency mechanism

Our application is thread-safe and we were able to achieve that by running *Server*, *ClientHandler*, and *Client* on different threads.

- *Server*, *ClientHandler*, and *Client* implement the interface *Runnable*.
- All methods in *Server* are synchronized - there are fields that cannot be accessed at the same time (for instance HashMap games) and they are used in many methods. It is more convenient to synchronize the whole method instead of the fields themselves.
- All methods in *ClientHandler* are synchronized with the same reason - ClientState needs to be synchronized and is used many times in different methods.
- Each of our client, and games have their own board that they update independently on their side based on the moves they receive, so the board is not shared at all.

Reflection on initial design



Initial UML Diagram

The diagram we created initially was much simpler than the one we ended up with in the end. The initial design was not bad, the core functionality of our system is still the same, but it misses some classes and methods. We made a lot of adjustments and improvements to make the code cleaner and the whole system more efficient.

Pros of initial design

- Good networking mechanism
    - *ClientHandler*, *Server* and *Client* are implementations of *Runnable*, which plays an important role for the program to be thread-safe.

    - Each *Client* has their own *Board*, this prevents concurrency problems (because Board is not shared, but everyone just updates their own board).
    - Server has a list of games, this makes it easier for the server to manage the games.
- *Board* class has necessary methods such as getting the field's value, setting the field's value, checking if the field is empty, checking winning positions in a row, column or diagonal.

Cons of initial design

- Class ClientHandler does not have to have a field to store username.
- Class Board lacks a method to make a deep copy of the board.
- Class Board lacks a method to check if the board has one winner or two winners.
- Class Board does not store the dimension of the board anywhere, this could be useful if we ever wanted to expand the board.
- There was a lot of data that was used many times => it could be good to store them somewhere.
- Letting *Client* implements *PlayerSkeleton*

Final design (Why did we choose this design?)

- We added models to store data that are used multiple time (Direction, Move, Quadrant, Rotation).
- We created an abstract class PlayerSkeleton and PlayerBody to have the general fields and methods needed for a player.
- We added classes HumanPlayer and AIPlayer to extend the abstract class PlayerBody
    - HumanPlayer ask user for move
    - AIPlayer decides the moves by itself
- We added DTO (data transfer object)
    - Dto stores the indexes of the move
- We added mappers
    - MoveMapper class translates (maps) the DTO object containing indexes of the move to the components needed to move (index, rotation) and vice versa
- We added models in networking to store data that are being used multiple time
    - Class ClientGame is used to store informations about the two players, creates a board and new game id for each new game being created
        - This class is used in ClientHandler and is stored in a hashmap
        - When the game is over, the ClientGame object is removed from the hashmap

        - This is similar to the Game object in initial design
    - Interface GameOver to store states for different game over situations
        - VICTORY - someone wins
        - DISCONNECT - someone was disconnected from the server
        - DRAW - the game is draw

Test plan

Test explanation

Pentago Board Test

Before each test we initialize a Board object.

- Test moves
    - Test for at least three correct indexes (within 0 to 35).
    - Test for index smaller than 0.
    - Test for index larger than 35.
    - Test method setField(int index, Mark mark) and getField(int index).
        - Set a mark to a field and check that the mark is indeed in the field.
        - Test that other fields are empty.
    - Test for rotations for each quadrant (top left, top right, bottom left, bottom right)
        - Set three marks to three specific fields of a quadrant (one in the middle, one on the corner and one on the edge) and then rotate the quadrant clockwise.
        - Check that the marks are not in the old fields.
        - Check that the marks are in the correct new fields.
        - Do the same for rotating counterclockwise.
- Test board setup
    - Test that all fields are empty.
- Test board reset
    - Set two marks on the board (anywhere).
    - Reset the board and test that all fields are empty.
- Test deep copy
    - Set two marks on the board (anywhere).
    - Make a deep copy of the board and check that all the fields of the copied board are the same as the fields of the original board.
    - Set a mark on the copied board (for example: change the mark of one of the fields that were marked before).
    - Check that the field of the original board still has the same mark.
    - Check that the field of the copied board has the new mark.
- Test full board

    - Set all the fields with a mark except the last one and check that the board is not full.
    - Set the last field with a mark and check that the board is now full.
- Test game over
    - Test game over and has no winner (i.e. board is full).
        - Set the fields so that the board has only one field left with no winner.
        - Check that the game is not over and the board is not full.
        - Set the last field to a mark that does result in a full board and no winner.
        - Check that the game is over and the board is full and there is no winner.
    - Test game over and has two winners (i.e. both players win after a rotation).
        - Set the fields so that after rotating one quadrant, both players win.
        - Check that the board does not have one player.
        - Check that the game is over and there are two players.
    - Test game has one winner in a row / column / diagonal
        - Set one type of mark to the fields in the board to have an almost winning position (missing one mark).
        - Test that there is no winner with a row.
        - Set one same mark to the field in the row to have a winning position.
        - Test that there is a winner with a row.
        - Do the same test for column and diagonal (for diagonal check all diagonals)
    - Test a random game
        - Use a while loop to play random moves until the game is over (full board, draw or there is a winner).
        - After each move, generate a new arraylist of playable moves, so we do not play on a non-empty field.
        - Check if the game is over.
        - Have another test to test the random game for 30000 times.

Server Test

Use a PrintWriter, a BufferedReader and connect a socket to communicate with the server. Create multiple writers, readers and sockets to test multiple connections. Players here mean a connection to the server.

- Test command PING
    - When the server receives “PING” it should send the client a “PONG” back.
    - Passes if after a client sends a “PING” client reads a “PONG”.
- Test wrong input
    - Send a message using a different separation symbol than the protocol (“HELLO-Server’s test” instead of “HELLO~Server’s test”).
    - Send a message that lacks an argument (“LOGIN~”).
    - Send a message that contains “~” in the argument (“LOGIN~Ha~n”).
    - Passes if after each wrong message server responds with an error message starting with “ERROR”.
- Test used username
    - Login with the same username (using two different connections).

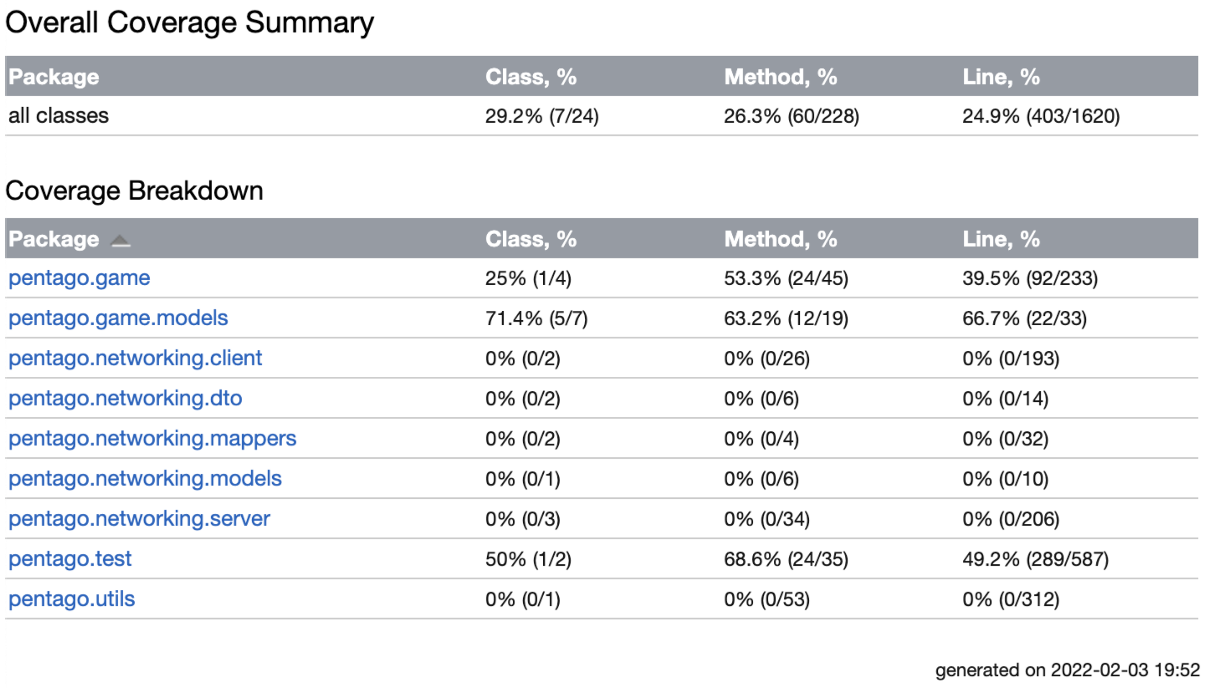
    - Passes if server sends a “ALREADYLOGGEDIN” to the second reader.
- Test wrong moves
    - Start the game and send invalid moves to the server.
    - Passes if the server responds with an error message starting with “ERROR”.
- Test wrong turn
    - Send two moves from one writer to the server.
    - Passes if the server responds with an error message starting with “ERROR”.
- Test commands
    - Log two players in (call them Player1 and Player2).
    - Test command “LIST” => test if server sends back the list containing names of both players.
    - Queue both players (Player1 first, Player2 second) => test if server sends a “NEWGAME” message to both players.
    - Send a move from the second player => test if the server sends back an error message starting with “ERROR”.
    - Start playing by sending valid moves so that Player1 wins => test if server sends a victory message to both players (“GAMEOVER~VICTORY~<name of the first player>”).
    - Passes if every test is correct (green).
- Test queue
    - Log in and queue one player, send a move from that player => test if the server sends back an error message starting with “ERROR”.
    - Log in and queue another player => test if the server sends back a message “NEWGAME~<first player’s name>~<second player’s name>”.
    - Log in and queue another player => no error occurs.
    - Let the third player send a move => test if the server sends back an error message starting with “ERROR”.
    - Passes if every test is correct (green).
- Test client lost connection
    - Log in two players and let them start a new game.
    - After playing a few (4) moves, disconnect the second player (close socket2).
    - Passes if the server sends back “GAMEOVER~DISCONNECT~<first player’s name>” (game is over due to connection lost).
- Test quit command
    - Log in two players and send a “LIST” command => test that the server sends back a list containing both players.
    - Let the first player quit and send a “LIST” command from the second other player => test that the server sends back a list containing only the second player.
    - Passes if every test is correct (green).

Test metrics

Pentago Board Test (metrics)

These images below show how many percent we have covered with our PentagoBoardTest.

Image 1. Test coverage on the whole project of PentagoBoardTest



Overall Coverage Summary

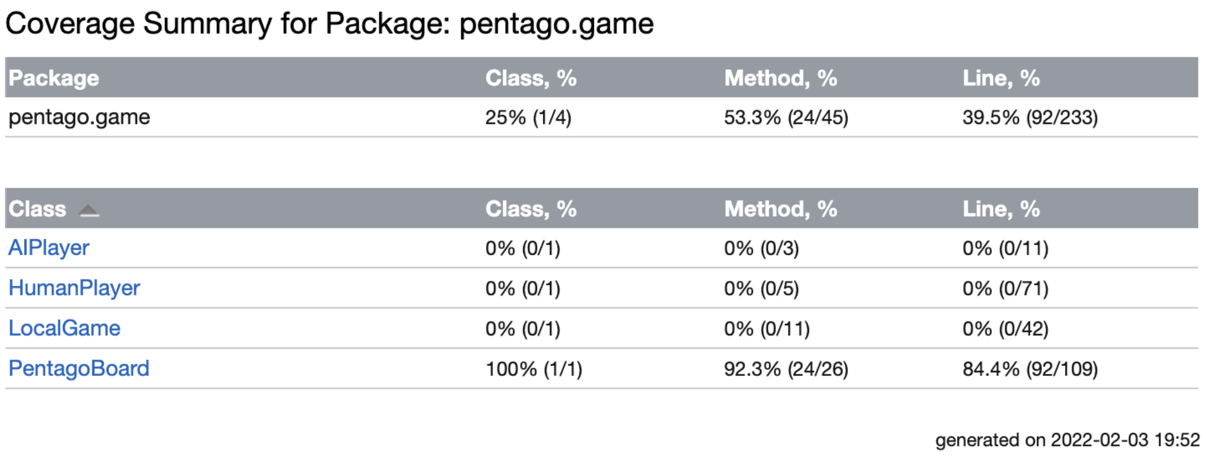
Package	Class, %	Method, %	Line, %
all classes	29.2% (7/24)	26.3% (60/228)	24.9% (403/1620)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
pentago.game	25% (1/4)	53.3% (24/45)	39.5% (92/233)
pentago.game.models	71.4% (5/7)	63.2% (12/19)	66.7% (22/33)
pentago.networking.client	0% (0/2)	0% (0/26)	0% (0/193)
pentago.networking.dto	0% (0/2)	0% (0/6)	0% (0/14)
pentago.networking.mappers	0% (0/2)	0% (0/4)	0% (0/32)
pentago.networking.models	0% (0/1)	0% (0/6)	0% (0/10)
pentago.networking.server	0% (0/3)	0% (0/34)	0% (0/206)
pentago.test	50% (1/2)	68.6% (24/35)	49.2% (289/587)
pentago.utils	0% (0/1)	0% (0/53)	0% (0/312)

generated on 2022-02-03 19:52

Image 2. Test coverage on the package game



Coverage Summary for Package: pentago.game

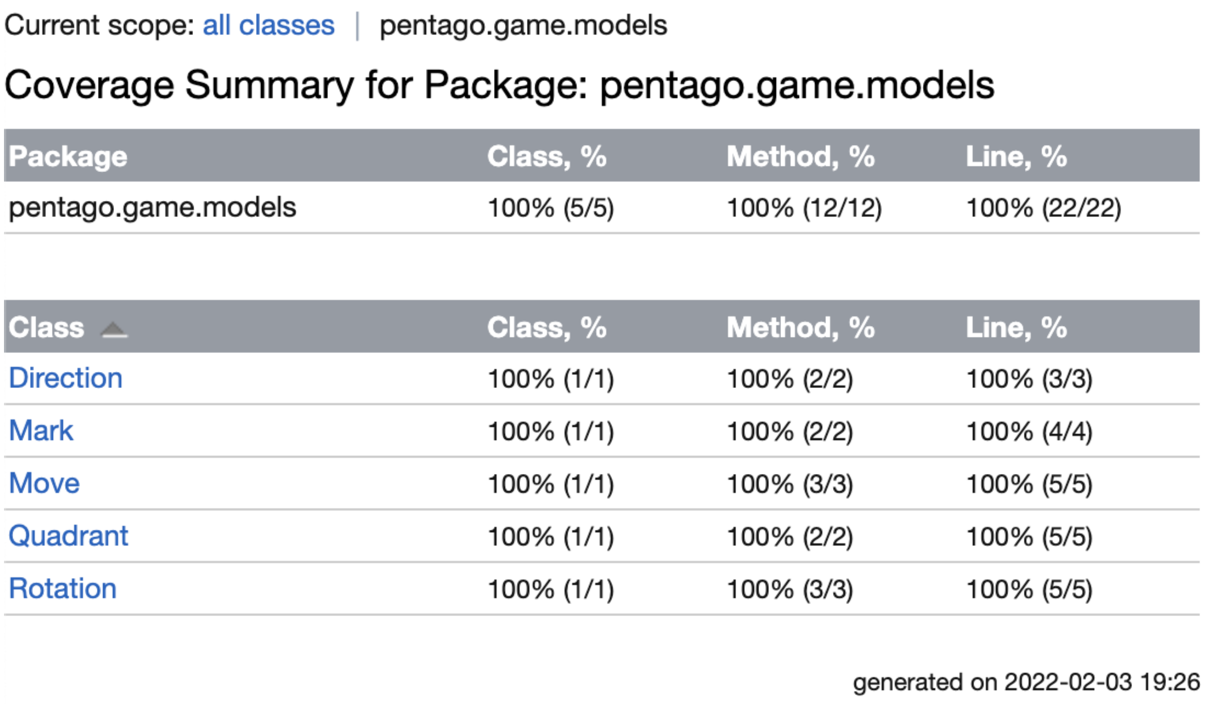
Package	Class, %	Method, %	Line, %
pentago.game	25% (1/4)	53.3% (24/45)	39.5% (92/233)

Class	Class, %	Method, %	Line, %
AIPlayer	0% (0/1)	0% (0/3)	0% (0/11)
HumanPlayer	0% (0/1)	0% (0/5)	0% (0/71)
LocalGame	0% (0/1)	0% (0/11)	0% (0/42)
PentagoBoard	100% (1/1)	92.3% (24/26)	84.4% (92/109)

generated on 2022-02-03 19:52

Test coverage over the models used in the Pentago board

Image 3. Test coverage on the package game.models


Current scope: all classes | pentago.game.models

Coverage Summary for Package: pentago.game.models

Package	Class, %	Method, %	Line, %
pentago.game.models	100% (5/5)	100% (12/12)	100% (22/22)

Class ▲	Class, %	Method, %	Line, %
Direction	100% (1/1)	100% (2/2)	100% (3/3)
Mark	100% (1/1)	100% (2/2)	100% (4/4)
Move	100% (1/1)	100% (3/3)	100% (5/5)
Quadrant	100% (1/1)	100% (2/2)	100% (5/5)
Rotation	100% (1/1)	100% (3/3)	100% (5/5)

generated on 2022-02-03 19:26

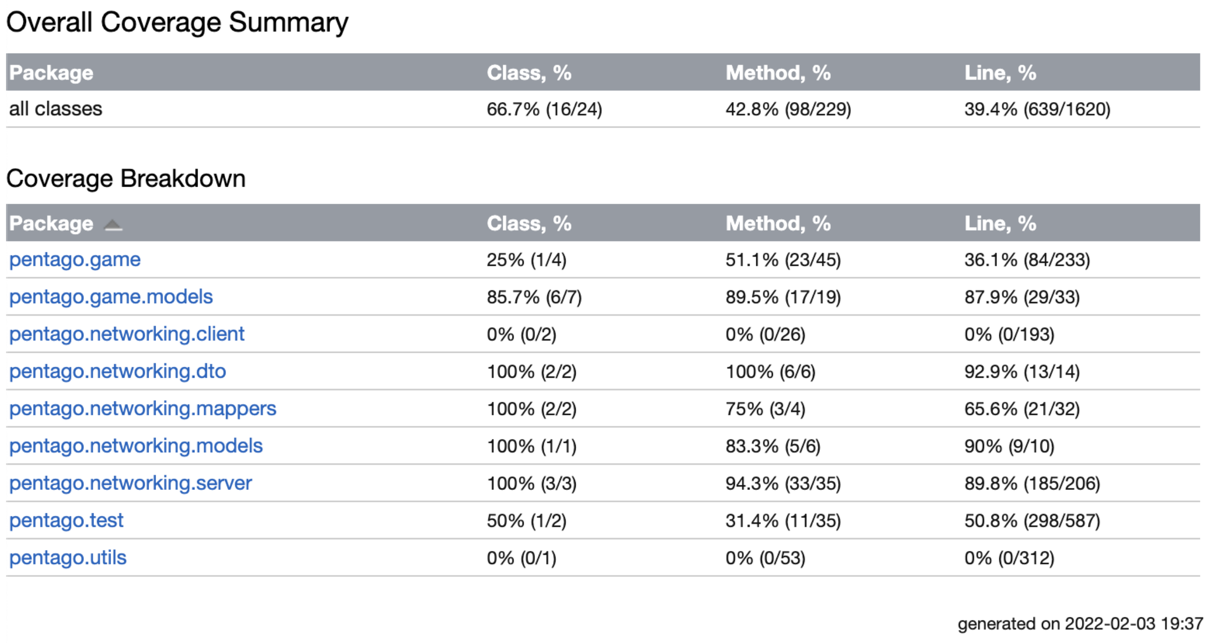
The coverage report shows that the PentagoBoardTest has been exercised very well, we also tested a lot of edged situations (see more *Pentago Board Test*).
This test has high coverage on the class PentagoBoard and
This test has lowest coverage on packages *networking* and *utils*

- Package *networking* has low coverage because we just tested the board and the logic of the game.
- Package *utils* has low coverage because it only contains a class TextIO, which is used to read from system input, which was provided by Twente, and we did not need that for testing the board.

Server Test (metrics)

These images below show how many percent we have covered with our ServerTest.

Image 5. Test coverage on the whole project of ServerTest



Overall Coverage Summary

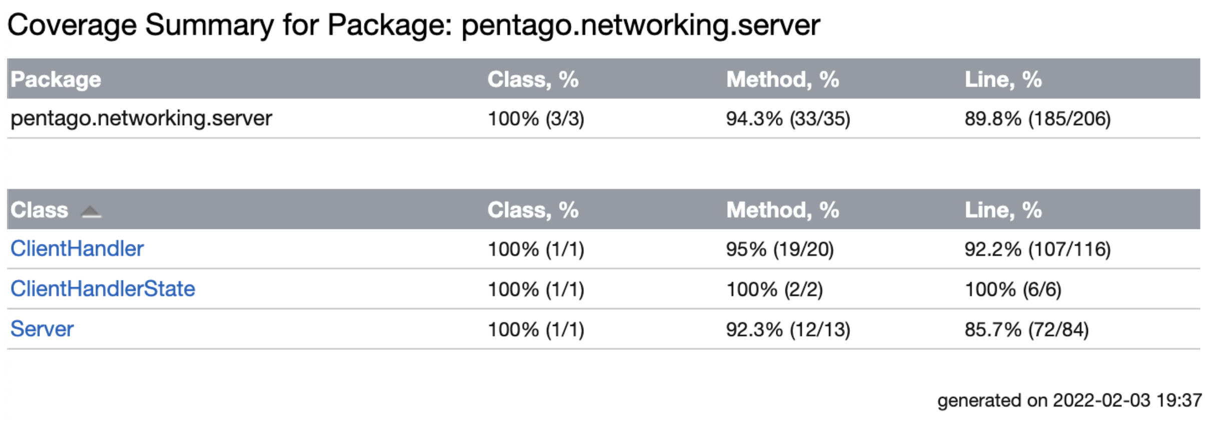
Package	Class, %	Method, %	Line, %
all classes	66.7% (16/24)	42.8% (98/229)	39.4% (639/1620)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
pentago.game	25% (1/4)	51.1% (23/45)	36.1% (84/233)
pentago.game.models	85.7% (6/7)	89.5% (17/19)	87.9% (29/33)
pentago.networking.client	0% (0/2)	0% (0/26)	0% (0/193)
pentago.networking.dto	100% (2/2)	100% (6/6)	92.9% (13/14)
pentago.networking.mappers	100% (2/2)	75% (3/4)	65.6% (21/32)
pentago.networking.models	100% (1/1)	83.3% (5/6)	90% (9/10)
pentago.networking.server	100% (3/3)	94.3% (33/35)	89.8% (185/206)
pentago.test	50% (1/2)	31.4% (11/35)	50.8% (298/587)
pentago.utils	0% (0/1)	0% (0/53)	0% (0/312)

generated on 2022-02-03 19:37

Image 4. Test coverage on package server



Coverage Summary for Package: pentago.networking.server

Package	Class, %	Method, %	Line, %
pentago.networking.server	100% (3/3)	94.3% (33/35)	89.8% (185/206)

Class	Class, %	Method, %	Line, %
ClientHandler	100% (1/1)	95% (19/20)	92.2% (107/116)
ClientHandlerState	100% (1/1)	100% (2/2)	100% (6/6)
Server	100% (1/1)	92.3% (12/13)	85.7% (72/84)

generated on 2022-02-03 19:37

The coverage report shows that the ServerTest has been exercised very well, almost all methods in the package *server* were tested and we believe we have included very risky situations for testing (more in *Server Test*).
This test has lowest coverage on packages *client* and *utils* and *game*

- Package *client* has low coverage simply because we created the test for the server only.

- Package *utils* has low coverage because it only contains a class TextIO, which is used to read from system input, which was provided by Twente, and we did not need that for testing the server.
- It also has low coverage on the package *game* which makes sense since we were mainly focused on the functionalities of the server. More edged cases for the game itself is covered in the test for Pentago board (more in *Pentago Board Test*)

Individual Reflection

Reflection of Thai Ha Bui

I think I learned a lot during this project. Working together on some parts was an obstacle, because it is hard to merge the flow of thinking of two people in general, especially when you are working on a complex system. It was until we started implementing our plan that we realised the problems and shortcomings of our initial design. The final product is functioning, but I think if we were given more time (we are double degrees), we would have been able to make the application better and add more cool functionalities.

Reflection of Krystof Mitka

In this project I have learned a great amount about creating a networking game from scratch on local computers. I think it gave a great view into how Java really works and I have tried out a bunch of different tricks in the code itself. This project also showed me it is important to be consistent with the commits and overall work otherwise there is way too much to be done at the last minute always. This module overall was crazy and I guess we could make it way better if we had more time for different stuff.