|  |
| --- |
|  |
| Four |
| A Connect Four implementation |

|  |
| --- |
| Bouke Bommerson ( s1243497) & Etienne Khan (s1008439) |

Contents

[Overall design 3](#_Toc410596575)

[MVC usage 3](#_Toc410596576)

[Server requirements 3](#_Toc410596577)

[Classes 4](#_Toc410596578)

[Package: ai 4](#_Toc410596579)

[Class: MonteCarloStrategy 4](#_Toc410596580)

[Class: NaiveStrategy 4](#_Toc410596581)

[Class: Strategy 4](#_Toc410596582)

[Package: client 4](#_Toc410596583)

[Class: Client 4](#_Toc410596584)

[Class: ClientInputHandler 4](#_Toc410596585)

[Package: constants 5](#_Toc410596586)

[Class: Constants 5](#_Toc410596587)

[Package: gamelogic 5](#_Toc410596588)

[Class: Board 5](#_Toc410596589)

[Class: ClientAIPlayer 5](#_Toc410596590)

[Class: ClientHumanPlayer 5](#_Toc410596591)

[Class: ClientOtherPlayer 5](#_Toc410596592)

[Class: Game 5](#_Toc410596593)

[Class: Mark 5](#_Toc410596594)

[Class: Player 6](#_Toc410596595)

[Class: ServerPlayer 6](#_Toc410596596)

[Package: mvc 6](#_Toc410596597)

[Class: ViewInterface 6](#_Toc410596598)

[Package: networkcode 6](#_Toc410596599)

[Class: ClientHandler 6](#_Toc410596600)

[Class: Lobby 6](#_Toc410596601)

[Class: NetworkMessage 6](#_Toc410596602)

[Package: server 7](#_Toc410596603)

[Class: Server 7](#_Toc410596604)

[Package: utils 7](#_Toc410596605)

[Class: States 7](#_Toc410596606)

[Class: Tools 7](#_Toc410596607)

[Metrics 8](#_Toc410596608)

[Reflection on Planning 9](#_Toc410596609)

[Countermeasures 9](#_Toc410596610)

[What did we learn? 9](#_Toc410596611)

[Being next year’s student assistant 9](#_Toc410596612)

# Overall design

We have made use of two different patterns for this project, the first being Model-View-Controller. The second pattern, Observer, was used to rewrite significant parts of the server implementation.

In the first implementation of the server we tried to handle input from multiple threads. Which quickly turned out to be more difficult than expected. We spawned one thread for each player connected to the Server and one thread for a game being run on the server. Inter-thread communication quickly became a mess.

By asking a more experienced coder, unaffiliated to the Softwaresystemen class, we learned that the Observer-pattern may be more useful for us in this situation as it only sends updates on client input. We have implemented a messaging system that exchanges data amongst all observing classes and each class only extracts relevant information from a notification, called NetworkMessage, based on certain states and rules which are to be explained later in this document.

There are two high resolution class diagrams included, one hierarchically designed, the other one navigational. They were constructed with the help of Visual Paradigm.

## MVC usage

In our implementation our Board class represents the Model, the View is represented by the class ViewTUI and the controller is the class Game.

## Server requirements

1. We do this directly in the server class. Because the server has to be started with the proper command line arguments there is no other way for it to work except with a proper, free, port.
2. Because our server gets its parameters via command line arguments a new port can easily be chosen.
3. The server supports multiple clients easily due to our lobby and ClientHandler system.
4. Server debug messages only use System.out
5. Preliminary experiments with clients from our group have shown that we were able to communicate with them.

# Classes

What follows is a summary of classes we made and their discussion regarding how they fit into the framework.

# Package: ai

## Class: MonteCarloStrategy

We implemented a pure Monte Carlo strategy that leaves lots of room for improvement. We mainly use very slow self-written functions, such as the *deepCopy* for the board. Some points for improvement are: multithreading, a bitboard and using a better random function than java.util.rand (for example xorshift1024). Because Monte Carlo is based on randomness we actually call our NaiveStrategy to play random games.

## Class: NaiveStrategy

The NaiveStrategy places a random move on the board according to the rules.

## Class: Strategy

Strategy is an interface which is implemented by Monte Carlo and Naïve.

# Package: client

## Class: Client

Client launches the client implementation based on launch parameters. It needs the IP and Port of a host, the player’s name and either the keyword *Human* to be able to play yourself, or the keyword Monte for the Monte Carlo strategy. The client also keeps track of its own board so one can only play valid moves.

## Class: ClientInputHandler

When a human player wishes to play ClientInputHandler will wait for moves via System.in and then send these moves to the server.

# Package: constants

## Class: Constants

This class mainly exists for the purpose of being able to use IDE features such as “find usage”. If this function is used on one of our error constants we can quickly see in which files it is implemented. It is also keeps the code neat by not having Strings everywhere.

# Package: gamelogic

## Class: Board

The board class is the Model part of the MVC model. It contains all rules needed to play Connect Four. Board is observable so status updates can be sent to the proper ClientHandler via our NetworkMessage class and to the view.

## Class: ClientAIPlayer

The ClientAIPlayer is the stand-in for the computer during a game. This means either the NaiveStrategy or the MonteCarloStrategy can be loaded.

## Class: ClientHumanPlayer

This class is used for human players, it will receive input from System.in. If the player requests a hint from the game, a move from the NaiveStrategy will be offered. This of course can be extended to implement the MonteCarloStrategy instead.

## Class: ClientOtherPlayer

This class acts as a dummy handling input from the opponent if this game is played via the network.

## Class: Game

This is the controller part of the MVC model. All requests regarding moves originate from here.

## Class: Mark

This enumerator keeps track of all values that a field can have during play, namely XX, OO or empty (EM).

## Class: Player

This abstract class can be implemented by the other player classes, it offers some basic functionality such as return the name and mark of a player.

## Class: ServerPlayer

The ServerPlayer is initialized whenever a network game is played, it enforces and checks the rules to which the clients have to adhere.

# Package: mvc

## Class: ViewInterface

This class is the view interface of the mvc.

Class: ViewTUI

Whenever a game state has to be redrawn the methods from this class are being called.

# Package: networkcode

## Class: ClientHandler

The ClientHandler class is the link between our server and a connected client. It constantly waits for input from the BufferedReader. Once input has been read, the input will be sent to the NetworkMessage class which then will be distributed to all the observers. Based on set flags and the input’s contents the proper observer knows how to handle said input.

## Class: Lobby

We have introduced our own lobby able to observe. It contains two lists, ClientHandlers that have successfully sent the *join* command and ClientHandlers which have sent the *ready\_for\_game* command.

As soon as two players are ready the lobby will create a new game session for the two players. The lobby also randomizes the order of play.

## Class: NetworkMessage

The NetworkMessage offers all observers the necessary tools to see where a message is from, what its contents are and if it has been sent by the server or by a client. The constructor can either get a String and ClientHandler as argument, denoting that this is a message from a client or just a message without a ClientHandler meaning that it originates from the server.

# Package: server

## Class: Server

The Server class can be constructed by passing a port number as argument. This will construct a new Server which will listen on said port number for new connections. Furthermore it also creates a “lobby” (not to be confused with the lobby extension from the protocol) in which ClientHandlers are put that have successfully “authenticated” by providing the *join username group* command.

The server is an observer so it can analyze the content of a NetworkMessage should it be relevant to the server (e.g. the *join* command).

# Package: utils

## Class: States

We chose to create an enumerator for all the states a client can be in so that there is no confusion later on when working with client states.

## Class: Tools

Tools exists for a simple purpose, namely to provide an easy way of sending data through a BufferedWriter and to send Error messages to the proper client.

# Metrics

By far our most complex class is the Board class. Its Weighted Method complexity is 59 whereas the second complex class just reaches 25. To understand why the complexity is so high we have to take a look at its functions, because the WMC is calculated by summing the complexity of each function within the class. In this metrics analysis we are ignoring the high complexity of the Client class as we think of it as an anomaly based on how the score is calculated.

When calculating the Cyclomatic Complexity for all classes and sorting them by complexity, five functions out of the top ten belong to the Board class. The function *hasDiagonal(Mark)* being the most complex function in our project with a complexity score of 9 (and an extended cyclomatic complexity of 11).

A closer look quickly reveals that the *hasDiagonal(Mark)* function essentially consists out of two parts working through the same input, namely finding out if a diagonal win is present or not. It checks for a diagonal win with left leaning marks and right leaning marks. This could be split in two different functions drastically reducing the complexity of the function, but not of the class.

We are happy with these metric results as, apart from one class which arguably has to be this complex, we achieved a very non-complex project.

We also took a look at the LCOM, Lack of Cohesion in Methods. A large LCOM value means that a class has many different responsibilities and is prone to faults. In our case the NetworkMessage class had the biggest LCOM value of 8 whereas the second highest class had a value of 5.

This is understandable considering the NetworkMessage class is used to exchange update information between all observing classes. Even though it has the highest score out of all our classes, we are confident that this class is no more prone to faults than any other class, considering it only provides parsing mechanisms for other classes.

# Reflection on Planning

Up until now both group members have worked together on all projects of this academic year, bar the design project, so there was no time required to “get to know each other”. On the one hand each of us knew what to expect from the other, on the other hand this sometimes lead to some communication issues, as both group members fall out of the definition spectrum of a normal student.

A planning for this project had to be drafted and consecutively verified by a student assistant before January 8. At this point in time it was not clear yet that the illness of one of the group members would persist for about ten more days. The planning was made with a lot of free time in mind before the deadline of the project to be able to either flesh out certain elements some more or use it as a fallback buffer in case one element took longer to implement.

Tension eased after receiving the message that this project was not due on Friday, but Sunday and that the grade was not limited to a 6.0, which lead to a more productive workflow. For example the whole server implementation has been rewritten in one day whereas it took several days to build it the first time.

## Countermeasures

In the winter vacation both group members read the paper of Victor Allis “A Knowledge-based Approach of Connect-Four” in hopes of being able to implement the *perfect strategy.* As time went on it became more and more obvious that the current schedule could not be adhered to. After the illness was over all plans to implement additional features were dropped in favor of only implementing the basic requirements. Only if time was left would the group intend to implement additional features. On the last day a pure Monte Carlo Strategy was implemented based on the earlier implemented NaiveStrategy.

For Etienne the biggest obstacles in this project was to construct a big and complex software after only having worked with the rather self-contained assignments from the prior weeks. As for Bouke: Not having a clear design for something to be implemented.

## What did we learn?

A project is more work than just the sum of its parts.

## Being next year’s student assistant

We would recommend the following “Do’s and Don’ts”

Do:

Start really early with your design for the end project and treat class-diagrams as one of the most valuable assets you will have.

If possible look at more complex projects than what you currently work with. Even if you don’t understand all of it. It is important though that you get a feeling for larger projects.

When writing code, always add documentation to anything you do, if at a later moment it seems superfluous you can always remove it, but sometimes its handy when you look back and wonder what your thoughts were.

Don’t:

(Don’t get sick!)

Don’t be afraid to rewrite code if that means your solution will then be easier to maintain/extend, but be sure that this doesn’t happen at the last minute.

Don’t underestimate the project as a whole.