CONNECT FOUR

documentatie, implementatie en testresultaten

SEBASTIAAN DEN BOER (S1523511) & WOUTER BOLHUIS (S1532286)

|  |  |  |
| --- | --- | --- |
|  |  |  |

Inhoud

[Discussion of Overall Design 1](#_Toc410252677)

[Discussion per Class 7](#_Toc410252678)

[Test Report 10](#_Toc410252679)

[Metrics Report 13](#_Toc410252680)

[Reflection on Planning 14](#_Toc410252681)

Discussion of Overall Design

Observer

As soon as we create a Game in our implementation of Connect Four, we create a GameTUIView as well. This class implements Observer and GameView, while GameView itself extends the Observer interface of Java.

In Game we can then add an Observer to a specific class. In our implementation we added Observers to both Board and Rules. Every time the Board class and the Rules class are created in the constructor of Game, the addObserver(view) is called on these classes.

This view is the GameTUIView where most of the important messages are displayed directly or indirectly to the user of the game. Obviously these classes should implement Observable to work properly.

After the Observer has been set on the Board and Rules classes, we can now send messages from these classes to the GameTUIView as well as notify the TUI of a change in the game. The classes call the methods setChanged(); and notifyObservers(string); to notify the game of such a change.

Both methods are necessary for the Model-View-Controller/ Observer patterns to be implemented properly. notifyObservers(string) is the interesting one here. This one makes sure that the right message is delivered to the player.

For example, when a game is a draw, the string in notifyObservers(string) will be replaced by “draw”. After that the associated Observer will notify the GameTUIView of a change. The message passed by the Observer will then be read and output will be show accordingly.

**Model-View-Controller**

In our implantation of Connect Four we’ve tried to stick as much to our previous experience with the Model-View-Controller pattern as humanly possible. For example, our Client and Server ask for input and print messages to the screen, basically the purpose of a View, even though they were designed to be Controllers.

However, because we use the Observer pattern which is in most cases directly linked to a Model-View-Controller pattern, the core functionality like checking whether or not a move is valid, but also to see if there is a winner or a draw, have been implemented according to the MVC-pattern.

Beforehand we had determined that we needed at least the classes Board, Server and Player to be Models. In the end, we almost implemented our game like that, but in some cases we needed specific input from the player straight away, so we chose to do that right there in these classes. Of course, we could have just sent a setChanged(); and notifyObservers(); to the GameTUIView and ask for input there depending on the message sent, but sometimes that didn’t make sense, like in Server where a port has to be chosen.

Of course, Game is more or less still a view, but in our implementation it also got the role of a Controller, because the GameTUIView calls events from the Game. This means that we’ve tried to split up our View and Controller elements to GameTUIView and Game respectively.

As for the other classes, most of them were designed to be Controllers, but some of them have some elements in them which can also be found in Models or Views. But in our complete implementation, we’ve tried to stick as much to the original design as much as possible.

**Class Diagram**

On the right hand of our class diagram we placed all the classes that had to do with server connection, those are the classes ClientHandler, Server, Client & the two protocol classes ProtocolControl and ProtocolConstants.   
More in the middle are the classes which are required for plenty of other classes, like Game, Player and Mark.   
Located at respectively the bottom and the top are Rules and Board, they have been placed there because they are targeted by a bunch of classes especially Game and thus would fit in at a location so close to Game.  
On the left we have the different types of Player’s, HumanPlayer and ComputerPlayer. Besides that we have the strategy classes required for ComputerPlayer, Strategy, Smart and Stupid.   
As you can see at Game it has a couple of connections with Player, these connections are for every Player in the game class. Every Game has three of those Player’s, the two Player’s which play the game ‘one’ and ‘two’ and the Player ‘current’ which is equal to either ‘one’ or ‘two’ depending on whose turn it is. Thus Game has a total of two Player’s at all times, as a Game can’t be started without players. These Players can only play one game at a time, so they can have only one Game.   
Whenever a Game is created it will generate a Board, so this game will always have one Board. It is possible for a Board to be constructed without being immediately linked to a Game as Smart will construct a new Board to find the best move.

Every Board has a Mark array stored. This Mark array consists off the Marks for every single slot of the board and thus has a length of 42, there for every Board has a total of 42 Marks.

HumanPlayer and ComputerPlayer both extend Player. As they extend Player it is not necessary to add that HumanPlayer and ComputerPlayer both have a Mark as that is already defined for Player it selves.   
As a ComputerPlayer needs a Strategy to make its move every ComputerPlayer has a Strategy and every Strategy has a ComputerPlayer. As Strategy is an Interface it needs to be implemented by other classes which do the actual work. These classes are Smart and Stupid and as you can see in the class diagram these indeed implement the Strategy.

A couple of classes make use of the Rules class, these classes are Game & Board. Smart uses the game it’s given to get to the Rules and there for does not need an association. Rules needs a Mark to perform some of its commands to check whether something the Player with that specific Mark does changes the state of the Game.

From Game there is an association to GameTUIView as that makes sure a user can play the Game in offline mode. Every GameTUIView will have one Game associated with it and that is the other way round as well.

**Formats for Data Storage & Communication Protocols**

All of the data in this project has been saved in fields of the Client and Server applications. We haven’t used any other possible method to store our data. Of course we’ve made choices what to make static or final fields, but usually everything is stored in fields, except for the class Mark, which is actually not a class, but an enumeration.

As for the communication protocol, we’ve tried to use it to the best of our ability. Sometimes the protocol was unclear about certain situations. We’ve tried to code around those issues by making our checks mostly case-independent or by using converters to transmit/ receive protocol messages in a correct manner.

We’ve tested our application with multiple teams to ensure a number of teams would at least be able to communicate with us, even though they had another implementation of the protocol provided to us by BlackBoard.

The protocol can be found online (on BlackBoard) but also in the ZIP-file of our project.

Systematic Overview of the Requirements

## Server

* Class(es): Server  
  Method(s): Server()  
  Entering the port number is covered in the Server constructor, there will be an opportunity to enter the port number. If the users decides to leave this empty the server will automatically start on port 1337 as that is the default port according to the protocol. If the input from the client isn't a number it will catch the NumberFormatException, warn the user and ask for the port again.
* Class(es): Server  
  Method(s): startServer()  
  Whenever the port number is in use or the socket can't be created for any reason whatsoever the server will catch the IOException, warn the user and ask for the port again.
* Class(es): Server  
  Method(s): analyzeString(String msg)  
  Whenever two players are waiting for a game they will be automatically linked and put in a game. As Game extends Threads the server can start multiple games and run them simultaneously.
* Class(es): Server  
  Method(s): constructor, startServer(), analyzeString(String msg), removeHandler(ClientHandler handler), shutdown(), getInput(String variable), sendMessange(ClientHandler handler, String msg)  
  Everything that happens on the Server that is worth noting will be printed. To make some of the more important things stand out they will be printed using System.err.println() as that will print the text in red instead of the standard black.
* chattenchattClass(es): Server, ClientHandler  
  Method(s): Server.analyzeString(String msg),

acceptRequest(ClientHandler handler, Mark mark),   
 removeHandler(ClientHandler handler),   
 sendBoard(ClientHandler handler)   
 doMove(ClientHandler handler, int decision)  
 gameEnded(Game game), turn(ClientHandler handler),  
We have implemented the protocol as best as we could, due to some confusions within the tutorial group and the protocol not every client can play on every server. We have made an attempt to have our server work for all the different implementations. We are not entirely certain it works with every single client, but as far as we have experienced it worked for everyone.

## Client

* Class(es): Client, Player, HumanPlayer, ComputerPlayer  
  Method(s): Client.getInput(String msg), Player.createPlayer(Mark mark),

ComputerPlayer.ComputerPlayer(Mark mark)  
 & all HumanPlayer method

When we created our TUI the aim was to make it as understandable as possible. That way it would make using the program easier.

* Class(es): Client, Player, HumanPlayer, ComputerPlayer  
  Method(s): Client.initializeClient(), HumanPlayer(Mark mark),  
   ComputerPlayer(Mark mark)  
  Whenever a client starts initializing it will ask several questions to the user. These questions are to determine if the player should be a humanplayer or a computerplayer, what the name of the humanplayer should be, what difficulty the computerplayer should be & how long the computerplayer should wait before making a move.
* Class(es): ComputerPlayer  
  Method(s): ComputerPlayer(Mark mark)  
  When a new ComputerPlayer is initialize it will ask what difficulty the ai should be and how long it should wait before making a move.
* Class(es): HumanPlayer  
  Method(s): DetermineMove(Game game),  
   DetermineMoveInt(String[] board, Mark mark)  
  The user gets a hint after 10 seconds of not filling in a valid move. The hint will be generated using the algorithm of the Smart ComputerPlayer and printed to the user. It will not make the move and let the player decide what to do.
* Class(es): GameTUIView  
  Method(s): checkInput(String input)  
  Whenever a singleplayer game ends the user can decide to restart it, whenever a multiplayer game ends the user can send a request to rematch.
* Class(es): Server, ClientHandler  
  Method(s): Server.removeHandler(ClientHandler handler)  
  Whenever a client terminates in any way whatsoever the server will contact the other player and tell them the other player left the game. The client will open the menu and allow both players to reconnect to the server.
* Class(es): Client  
  Method(s): readResponse();  
  Whenever the client loses connections it catches the IOException thrown, informs the player and opens the menu.
* Class(es): Client, ClientHandler  
  Method(s): Client.readResponse(),

acceptRequest(ClientHandler handler, Mark mark),   
 removeHandler(ClientHandler handler),   
 sendBoard(ClientHandler handler)  
 doMove(ClientHandler handler, int decision)  
 gameEnded(Game game), turn(ClientHandler handler),

We have implemented the protocol as best as we could, due to some confusions within the tutorial group and the protocol not every client can play on every server. We have made an attempt to have our client work for all the different implementations. We are not entirely certain it works with every single server, but as far as we have experienced it worked for everyone.

The client will make sure that whenever a change occurs the client's state is synced with the server's state.

Discussion per Class

## BOARD

The Board class is used in Game to store what the Mark value is of a slot. With Board you can alter those values, request the values. Board has to make sure every single move gets stored correctly, if that goes wrong people might be able to overwrite a Mark. Both Game and Smart use Board. Smart uses it to look forward and determine the best move. Game uses it to store all correctly inputted moves and to check whether a move is correct.

## CLIENT

The Client class is the class which is initially launched, if the user wants to play offline the Client class will be closed and the GameTUIView will be used to handle user input. Client is essential for communication with any ClientHandler. Client will handle all the input it gets from ClientHandler and take actions according to that input. The main responsibility of Client is to play online games without problems. It isn’t used by any other class as all the input Client gets and needs come from the user directly.

## GAME

Game is the main class which keeps everything together. Game is the link between Player, Board, Rules, GameView & GameTUIView. It basically sends commands through to the right class. All the classes named above occasionally request information from Game to continue working correctly. Game is the thing actually run to play a game, hence the name. Whenever a offline game starts a new Game will be started and whenever there are enough people waiting a Server will also start a new Game.

## RULES

Rules define whether a move is valid and whether a Player won or not. Every Game has Rules. These Rules will be requested by Board and Smart to do their work properly. These Rules can make or break a game. If the Rules are not defined correctly a Player could do an incorrect move and completely screw up a Game

## COMPUTERPLAYER

ComputerPlayer is a form of Player. A ComputerPlayer is a Player which automatically comes up with a correct move. This allows the user to play against a non-human Player. The ComputerPlayer determineMoveInt method is also used to give a HumanPlayer a hint occasionally. These ComputerPlayers can be chosen to connect on a server if the user is too lazy to play the game by themselves.

## HUMANPLAYER

HumanPlayer is the other form of Player. HumanPlayer is a Player completely controlled by a human user. This HumanPlayer will wait for user input before doing anything.

## MARK

Mark is the class used by Board to determine what value a certain slot on the playfield has. A Mark is assigned to each Player so that the Board can figure out which Mark to put on the right slot.

## PLAYER

A Player is someone who partakes in a Game. Every Game has two Player’s. A Player can determine the move someone should be made whenever it’s the Player’s turn. So Player has to make sure a move is made at the right point in time.

## CLIENTHANDLER

ClientHandler is the link between a Client and Server. All messages on the outputstream of the Client will be send to the Server to be analyzed. All messages the ClientHandler then receives from the Server will be send to the inputstream of the Client. Every ClientHandler is linked to one Server and one Client. Without the ClientHandler Client Server communication would be (almost) impossible.

## SERVER

The Server is launched when someone wants to make a server on which they want to play connect four. Server will connect to Clients and assign them a ClientHandler to handle communication from that point onwards. Server will make sure Clients get put into a Game and Clients can interact with that Game from all over the world. Server makes use off Game and Rules to handle most things.

## SMART

Smart is our smart ComputerPlayer strategy. It will determine the right move for the right moment to enlarge the changes of winning. Smart uses the Rules and Board to find out whether or not a move would be a good idea.

## STRATEGY

Strategy is an Interface to make using custom computer controlled Players work well. Without it neither Smart or Stupid nor ComputerPlayer would work correctly.

## STUPID

Stupid is our random ComputerPlayer strategy. It will choose a random spot to place its Mark. As long as that spot is still empty.

## GAMETUIVIEW

GameTUIView is what makes sure users can play offline games without problems. It makes it easy and clear for the user what to do next. GameTUIView will send the users input back to Game so Game can handle it from there.

## GAMEVIEW

GameView is an Interface originally created to give us the opportunity to create both a TUI and a GUI. Sadly there wasn’t enough time to also create a GUI. This Interface was kept in because the GameTUIView is based on it.

None of our classes have special cases in their contract on which precautions should be taken.

Test Report

Unit Tests

To determine if our implementation of Connect Four worked properly, we made a Junit test class. This class checks if all methods return what they’re supposed to return in case when a correct or invalid value is passed to the method. Usually these methods have built-in corrections in case a wrong value has been passed.

However, the only way invalid input could reach a method is by the hands of a client. This is why we usually check the input of the user before it is being passed on to the rest of our implementation. Just in case though, we’ve made a Junit test class to test most methods.

Sadly, the amount of classes we can cover with this test isn’t as much as we hoped. This is because some methods rely on user input. User input could not be tested as we didn’t implement a System.in override in our implementation of Connect Four.

The JUnit test can be found in the “tests” package of the project. It’s called OfflineTest. To check whether all methods are behaving correctly, simply open the JUnit class and Run it under a Coverage Test. This way the coverage is shown after the completion of each test.

Some methods are not implemented in the JUnit because of a lack of return type or because it requires user input. Also some methods are not interesting for a JUnit class.

Of course, we’ve ran our own tests and this is the end result:

JUNIT TEST COMPLETED & SYSTEM TESTS EXECUTED

55% COVERAGE ON AVERAGE OF THE ENTIRE PROJECT

System Tests

To compensate for the loss of coverage in our Unit Tests because of the limitation with respect to user input, we’ve performed the System Tests manually. As a result, we had to play a few games of Connect Four and test our implementation accordingly.

We even tested our Server class with other Clients of other project groups during and after the peer feedback hours. During those tests we discovered some fatal mistakes like index overwriting and not calling our special gravity function when we return a result to the client.

We’ve tried a couple of tests, but all of them can be done after one another, so please follow the following tutorial step-by-step to ensure a proper and complete system test.

STEP 1

* 1. Run under coverage: Client
  2. Press the Enter key
  3. Type “offline” in the Eclipse Console followed by the Enter key
  4. Press the Enter key
  5. Type “no” in the Eclipse Console followed by the Enter key
  6. Press the Enter key
  7. Type “stupid” in the Eclipse Console followed by the Enter key
  8. Press the Enter key
  9. Type “stupid” in the Eclipse Console followed by the Enter key
  10. Type “1” in the Eclipse Console
  11. Type “y” in the Eclipse Console
  12. Type “TestName” in the Eclipse Console followed by the Enter key
  13. Wait for the suggestion to appear in the Eclipse Console
  14. Wait 10 more seconds
  15. Type “8” in the Eclipse Console followed by the Enter key
  16. Type “0” in the Eclipse Console followed by the Enter key
  17. Play a game of Connect Four\*
  18. Press the Enter key
  19. Type “restart” in the Eclipse Console followed by the Enter key
  20. Play a game of Connect Four\*
  21. Type “exit” in the Eclipse Console followed by the Enter key

From this test we can now see that the coverage of the GameTUIView has increased to about 86.3%. That is a lot better than our previous Unit Test. Also, we’ve now implemented a lot of the code that requires input from the user in the Player classes.

STEP 2

* 1. Run under coverage: Server
  2. Press the Enter key
  3. Run under coverage: Client
  4. Type “online” in the Eclipse Console followed by the Enter key
  5. Type “computer” in the Eclipse Console followed by the Enter key
  6. Type “smart” in the Eclipse Console followed by the Enter key
  7. Type “1” in the Eclipse Console followed by the Enter key
  8. Press the Enter key
  9. Press the Enter key
  10. Run under coverage: Client
  11. Type “online” in the Eclipse Console followed by the Enter key
  12. Type “human” in the Eclipse Console followed by the Enter key
  13. Type “TestName” in the Eclipse Console followed by the Enter key
  14. Press the Enter key
  15. Press the Enter key
  16. Wait for the game to start
  17. Play a game of Connect Four\*
  18. Type “yes” in the first Client Eclipse Console followed by the Enter key
  19. Type “yes” in the second Client Eclipse Console followed by the Enter key
  20. Play another game of Connect Four\*
  21. Type “no” in the human Client Eclipse Console followed by the Enter key
  22. Type “menu” in the human Client Eclipse Console followed by the Enter key
  23. Type “exit” in the human Client Eclipse Console followed by the Enter key
  24. Terminate Server using the built-in Eclipse controls
  25. Type “no” in the AI Client Eclipse Console followed by the Enter key
  26. Type “exit” in the AI Client Eclipse Console followed by the Enter key

We should now have tested most of our Server scenario’s. Users can play against an AI online, input for Client has now also been tested properly and even input for Server has been tested. Functions that haven’t been tested properly yet are a Client disconnect or whether or not the Protocol is respected or not.

To test the Protocol a Client from another implementation of Connect Four is required. We don’t have access to that. To test a Client disconnect. Simply repeat steps 2.1 to 2.16 and then disconnect one of the Clients.

To test whether or not multiple games are possible on one server, please repeat steps 2.1 to 2.16 twice with each time a different name for the human Client.

Our coverage measured by Emma is usually over 50% for most classes in our Server and main package. Strategy has a high coverage depending on the difficulty of the AI.

Of course, we’ve tried all of these steps and so far we haven’t seen our program crash with no unexpected exceptions. Some exceptions may still occur though, but they rarely do appear in our implementation.

Metrics Report

The most complex class according to metrics is the Client class with on second place Server.  
This doesn’t really come as a surprise as both Client and Server have a lot of code aimed at reading input and acting on that input. Client has the hardest job though as it shouldn’t only do things but also wait for user input on how to do those things. This greatly increases both the total lines of code, but also the method lines of code. The readResponse() method of Client has a total of 130 method lines of code whilst Server’s analyzeString(String msg) has a total of 81. The reason both these numbers are so high is because of all the different inputs and output there can be which should all be covered.

If we however look at McCabe’s cyclomatic complexity our Strategy Smart is chosen with its determineMoveInt(String[] board, Mark mark) method. This method does indeed have a bunch of complex code to check what move gives it the best chances of winning the game.

The only way to make the code less complex is to create a whole bunch of extra methods to cover the more complex bits. That could decrease the complexity, but would at other points make the code more complex again and less readable.

Reflection on Planning

The planning of Week 4 of the module didn’t influence the planning for the Programming Project at all. OK, I mean, we did some things the same, but the actual time allocation was much more different than in week 4.

What made this planning more difficult was the insane amount of other stuff that needed to be done during these weeks as well. For example all of the programming assignments (non-project). They eventually caused us to fall a week and a half behind on our project planning.

Besides that, the tests during those weeks didn’t help moving the project forward either. We had to focus on that as well. Unfortunately, a family member of one of the project group members passed away during the week before last, resulting in a lack of time to actually finish the project. Also Server-Client communication is difficult, resulting in a postponed deadline for that part of the project.

Unforeseen circumstances usually cause these problems, but the face is they weren’t problems until we realized that we got behind because of the sheer amount of normal programming exercises.

Countermeasures to solve the lack of time were simple: allocate more time to the programming project. Of course, nothing is as simple as that, but this time it was. We just HAD to complete it with not much time left. This meant that we had to shrink the scope of our project drastically. No more extensions. That was the new goal, because it would mean that we would’ve completed the project properly, but without any extras.

This would ensure quality, but not a high grade. Just a passing one. This was the philosophy behind our re-allocation of time.

Hopefully next time the pressure from the other exercises is less, but just in case we’re going to start doing a lot more normal exercises “early” so that we have a week “buffer” in case we need it.

If we were student assistants next year, we would tell new students the following do’s and don’ts:

DO’S

* Start early with your programming exercises
* Do some self-study Java development during the evening hours

DON’TS

* Focus too much on Math and Design, leave some time for Programming
* Wait with testing till the last week. Try do it a week before last.