Software Systems

Java Project

Philipp Claßen & Teun Kok

Ringgz

In this exciting strategy game, players challenge each other to win the most territories using colorful rings. By creating a board, a server and a client it should be possible to play this game over a network against other players who made the game with the same protocol.

Table of contents

Introduction

In this report the game will be discussed in multiple steps. The first step is to show an overview of the game in a form of a class diagram. This will be discussed and some of the design choices will become clear.

After this , we will discuss all the classes separately. First we discuss the model classes. Right after that we discuss the view classes and finally the controller classes are discussed.

The MVC pattern will also be elaborated and there will be an overview of which classes are involved in the Model-View-Controller and Observer pattern.

The classes are tested by JUnit tests and these test are also discussed in the report and there will be a summary of the coverage of the tested classes. Moreover the JML provided classes and Metrics will be shown and explained.

Class Diagram

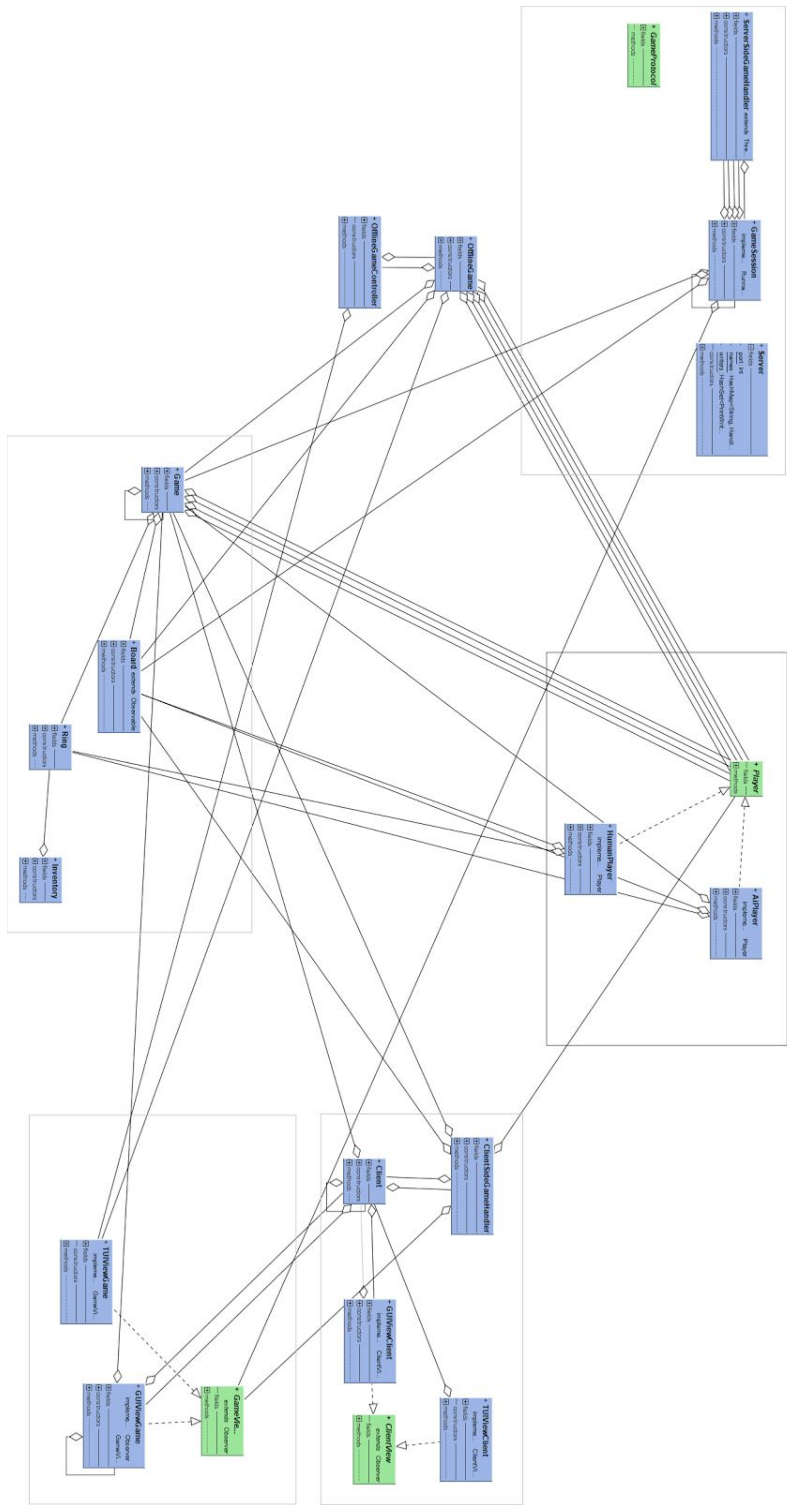
The class diagram shows the connection between all the classes that are made to play a game of Ringgz.

To start a new game, there should be a server started. The class Server contains the main method a server can be started by running that class.
A client can be started by running the class Client. The client that is started can be connected to the server. The Client creates a TUIViewClient/GUIViewClient.

The Server is the lobby where Clients can join. The Client makes the connection to the Server and tries to join the lobby.

When two or more clients are connected to the server and a player sends a gamerequest, a game is started. This is done by creating a new GameSession, the place where the clients are playing the game, on a separate thread. There can be created multiple GameSessions and multiple games can be played at the same time. When a game is started, the Client creates a ClientSideGameHandler to handle all the messages during a game from the server. The Server creates a ServerSideGameHandler for every Client in the GameSession.

The Board class, the Game class, the Inventory class ,the Human/Ai Player class and the Ring class are used by these classes to create a board, create a game and play the game on the board. These three classes make sure that the board can be filled and that a win can be checked. The TUIView is used to show the board. The class ClientSideGameHandler creates a new TUIGameView/GUIGameView to show the board.


+ GameProtocol
+ ServerSideGameHandler extends Thre...
+ GameSession impleme... Runna...
+ Server
port : int
names : HashMap<String, Handl...
writers : HashSet<PrintWrit...
+ OfflineGameController
+ OfflineGame
+ Game
+ Board extends Observable
+ Ring
+ Inventory
+ Player
+ HumanPlayer impleme... Player
+ AIPlayer impleme... Player
+ ClientSideGameHandler
+ Client
+ TUIViewGame impleme... GameVi...
+ GUIViewGame impleme... Observer GameVi...
+ GameVie... extends Observer
+ GUIViewClient impleme... ClientVi...
+ ClientView extends Observer
+ TUIViewClient impleme... ClientVi...
fields
constructors
methods

Systematic Overview

In the reader of Software Systems, there is a list of requirements which show what the application should do. In this part we show an overview of which classes are responsible for which requirements.

Server

1. When the server is started, a port number should be entered that the server will listen to.	Server
2. If the port number already is in use, an appropriate error message is returned, and a new port number can be entered.	Server
3. A server should be able to support multiple instances of the game that are played simultaneously by different clients.	GameSession, Server
4. The server has a TUI that ensures that all communication messages are written to System. out.	ServerSideGameHandler, Server
5. The server should respect the protocol	GameProtocol, ServerSideGameHandler, Server

Client

1. The client should have a user-friendly TUI, which provides options to the user to request a game at the server	ClientView, GUIViewClient, TUIViewClient
2. The client should support human players and computer players with some artificial intelligent behavior.	HumanPlayer, AiPlayer, PLayer
3. The thinking time of the computer player (and thus the power of the artificial intelligence) should be a parameter that can be changed via the client TUI.	Game, AiPlayer
4. The client provides a hint functionality. This shows a human player a possible move, as indicated by the computer player. The move may only be proposed, the human player should have the possibility to decide whether to play this move, or make a different one.	Game
5. After a game is finished, the player should be able to start a new game.	Client, ClientSideGameHandler
6. If a player quits the game before it has finished, or the client crashes, the other player(s) should be informed, and the game should end cleanly. In this case, the other player(s) should be allowed to register again with the server for a new game.	ServerSideGameHandler, Server
7. A server might at all times disconnect. The clients should react to this in a clean way, closing all open connections etc.	ClientSideGameHandler, Client
8. The client should respect the protocol	GameProtocol, Client, ClientSideGameHandler

Protocol

The communication between the server and the client should be clear. Therefore there is a protocol that describes which messages are sent from the server to the client and vice versa. The next protocol commands are important:

CLIENT_JOINREQUEST →
the client sends a message to server to join the lobby

SERVER_ACCEPTJOIN / SERVER_DENYJOIN→
the server checks if the entered name has not yet occurred in the list of names. If the name is not in the list, the server accepts the client by sending an accept request. If the name is already in the list, the client is denied.

CLIENT_GAMEREQUEST →
If a client wants to start a game, a gamerequest, with the amount of players is sent to the server.

SERVER_STARTGAME→
The server checks if there is another player available to play against. If there are enough opponents available, the server will start the game.

SERVER_MOVEREQUEST →
If the game is started, there should be one client who should do a move. The server sends a move request to that client.

CLIENT_SETMOVE →
The client can do a move by sending a setmove message with an x-, y-, and z-coordinate and a Colour to the server.

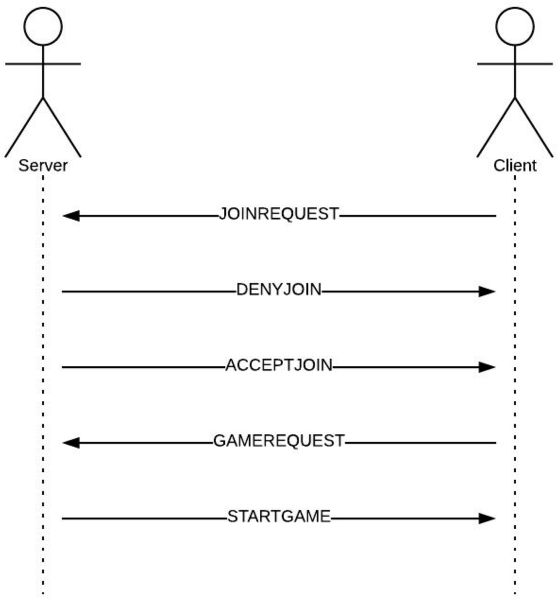
SERVER_DENYMOVE / SERVER_NOTIFYMOVE→
The server receives the move and check if the move is valid. If the move is valid, the move will be made and a notifymove message will be sent. If the move is invalid the server will send a denymove message.

SERVER_GAMEOVER →
If the game has come to an end the server sends a gameover message with the name of the person that lost the game. If there is a draw, the message is sent without a name.

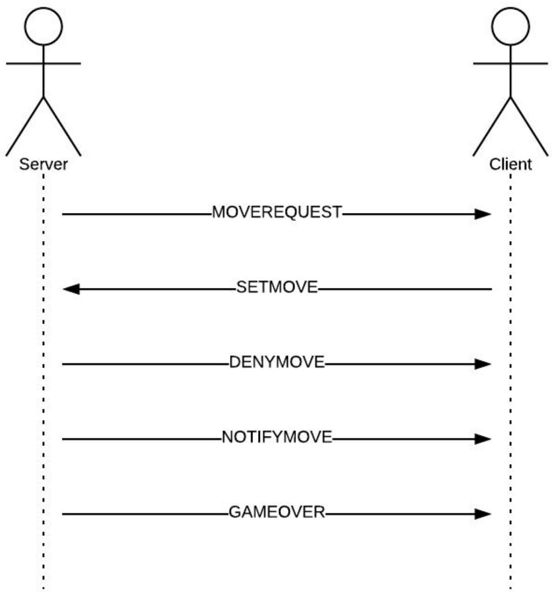
SERVER_CONNECTIONLOST →
It is possible that one of the clients lost connection and the other client should be informed. This client gets a message from the server that the other client disconnected.

SERVER_INVALID COMMAND →
Clients can enter a lot of wrong commands and therefore there is also a command to send a message to the client if he/she entered an invalid command.

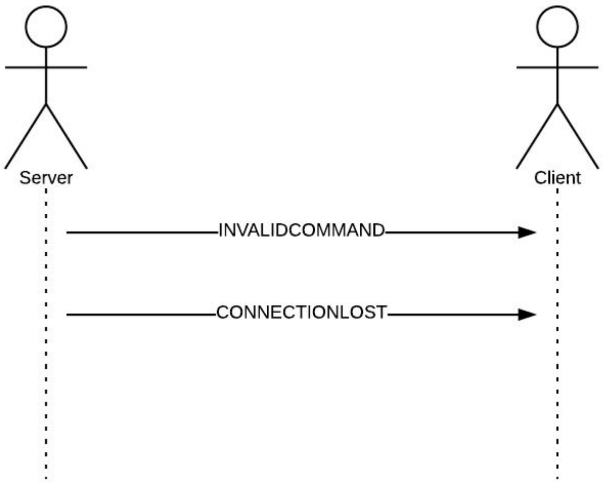
Starting up the Game:



Playing the Game:



Additional Messages:



Discussions of classes

Game

The game class is used to create the game session itself. It also manages the players and performs all the necessary checks before a move can be made.

Explanation
The **constructor** of Game asks only for an Integer describing the amount of players in the game. It also creates the board. To be used by this specific game session.

playerCreate receives up to 4 names represented by Strings. It uses these Strings to create a player for every name String given to it. If -AI is present in the String **playerCreate** will create an AI player instead of a human one.

Next up is the **getCurrentPlayer** method. When the game is created it creates a Integer currentP keeping track of whose turn it is. Every turn the currentP integer will be raised by 1 with the **changeCurrentPlayer** method. When **getCurrentPlayer** is called the correct player will be given back using a modulo and a case switch.

Then we get to the following checks:

- **checkInv**, receives a Ring and checks if this Ring is in the inventory.
- **checkIfSkipPlayer**, checks if the current player can make a move. If not it will check if the next player can make a move. It does this until there are no players left (ends the game) or untill it found a player that can still make a move.

makeMove, this method gets three integers describing a board position and a colour. Next it checks if the move to be made is the first of the game (by checking if the board is empty). If this is not the case **makeMove** checks if the given move is valid and if the ring is in the inventory, if so it returns a String in the correct format to be sent to the server.

getWinner, does what it's name suggests. The method sets integers describing the amount of points per colour by using the board method **hasPoints**. It then return the player with the most points, taking the playerCnt into account.

reset, empties the board and sets the currentP to 1.

Hint, goes through the board and the players inventory searching a possible move for the player to make. When found the move is printed for the player to use.

stringToColour, finds the correct Colour object for the given String

The remaining get/set methods:

- **getBoard**, returns the board.
- **getGameStatus**, returns the current game Status (being boolean gameRunning true or false).
- **getPlayCnt**, returns the amount of players in the game.
- **getPlayerByName**, returns the player object with the name given to the method.
- **setGameRunning**, sets the boolean gameRunning to either true or false.
- **setPlayer**, ascribes the given Player object to the games variables.

Responsibilities

The Game class is responsible for keeping track of the current player and of the player creation. Next to that it is also responsible for calling the checks validating the player's moves.

Usage of other classes

The only class used by game is the board class.

Board

The board class represents the board of Ringgz. The class contains only methods that either create or remove pieces on the board itself. It also has a few checks used by the game class for move validation.

Explanation

constructor, “constructs” the board itself. It creates a hashmap with an ID and an array of Rings with the colour EMPTY. When the value of a position on the board are x = 2 and y = 4, then the ID will be 24. The ID system was chosen to make the HashMap key more logical.

hasWinner, checks if the board has a winner. The board has a winner if the following requirements are met:

1. No player has any rings left in their inventory, checked with the **emptyInv** method.
2. No player can place a move on the board, checked with the **checkMovePossible** method.

If the board has a winner the method will return true.

checkMovePossible, checks if the given player can make a move. It does this by trying to validate every remaining ring in the player’s inventory on every position of the board.

validateMove, checks if the requested Ring placement is possible on the board. If the move is a starting move on the allowed spots the move is immediately validated. Else it is checked if the position is empty and the **checkValid** method is called.

checkValid, checks if the given Ring may be placed on the Given position. The method checks if the correct colour is present in the array itself or in one of the neighbouring arrays.

emptyInv, checks if the inventories in the given InventoryList are empty.

hasPoints, counts the amount of territories won by the given colour.

reset, fills all the arrays of the board with rings with the colour EMPTY.

isEmpty, checks if all the Rings in all of the arrays are of the colour EMPTY.

Remaining get/set methods:

- **getBoardMap**, returns the board as a hashMap.
- **getID**, turns the given X and Y values into a usable ID and returns this ID.
- **getBoardArrayXY**, returns the Ring array on the given X, Y position.
- **getBoardRingXYZ**, returns the Ring with size Z on the given X Y position.
- **getBoardRingID**, returns the Ring with size Z on the given ID position.
- **setRingXYZ**, replaces the Ring with size Z in the Array on the given X Y position with a Ring of the same size and a given colour.

Responsibilities

The board is responsible for the Ring placements and keeping track of the positions of placed rings. Next to that it also contains most checks performed on the board regarding placements and winning conditions.

Usage of other classes

Board uses the Ring and Board classes.

Ring

The class Ring creates the Ring object that is placed on board positions.

Explanation

constructor, the Ring constructor creates a new Ring object with the given size and colour. The colour can be RED, BLUE, GREEN, YELLOW, SPECIAL or EMPTY, as defined in the enumerator class Colour.

Remaining get/set methods:

- **getSize**, returns the size of the Ring.
- **getColour**, returns the colour of the Ring.
- **toStringColour**, returns the colour of the Ring as a String.
- **toStringRing**, returns the Ring as a String (Red 1 = R 1).

Responsibilities
The Ring class is responsible for the creation of the Ring objects.

Usage of other classes
Ring only uses the enumeration class Colour.

Interface → Player

The Interface Player is implemented by the classes AiPlayer and HumanPlayer. We will discuss the methods and responsibilities only in this Interface discussion since they are so similar.

Explanation

constructor, the constructor creates a new Ai- or HumanPlayer with a name, board, InventoryList and (for the AIPlayer) a game.

Get/set methods:

- **getName**, returns the name of the Player.
- **getColourList**, returns the colourList of the player.
- **getInv**, returns the Colours of the player's Inventories in a List.
- **getBoard**, returns the Board the Player is playing on.
- **getColourByInt**, returns the primary or secondary colour of the player dependant on the given Integer. Can also return special.
- **getMove**, only implemented by AiPlayer. The AiPlayer returns a String with a calculated move.
- **getStartMove**, only implemented by AiPlayer. The AiPlayer returns a String with a calculated starting move.

Responsibilities

The Player implementing classes are responsible for keeping track of the Inventories of players and the player names. Additionally the AiPlayer also calculates the moves made by the Ai. The HumanPlayer uses view to determine the move, this is done in the clientController.

Usage of other classes

Player makes use of:

- Board
- Inventory
- (Ai) Game

Inventory

The inventory contains an array of all the Rings of a certain colour that can be in a game. This means 3 bases, 3 tiny rings, 3 small rings, 3 medium rings and 3 large rings.

Explanation
constructor, creates a new Ring array and fills it with rings of the given colour. Like said before the rings are of size 0 to 4.

changeToSpecialInv, changes the Inventory to an Inventory to be used as secondary colour in a 3 player game. This means the Inventory now contains only one ring per size and they are all of colour yellow.

removeRing, searches for the given Ring in the Inventory and replaces it with one of colour EMPTY.

Remaining get/set methods:

- **getArray**, returns the Inventory as an Array.
- **getInvColour**, returns the colour of the Inventory.

Responsibilities
The Inventory keeps track of the types of Rings in the Inventory of a player.

Usage of other classes
Inventory uses the classes:

- Ring
- Enumeration class Colour

OfflineGame

OfflineGame is the offline implementation of our Game. It also makes use of our TUI instead of our GUI.

Explanation

main, creates a new OfflineGameController and runs it.

constructor, creates a new OfflineGame with a playercount, View and a OfflineGameController. After that it calls the **offlinePlayers** and start **functions**.

start, starts a game and keeps it running while the game is not yet finished.

createOfflinePlayers, creates new human/ai Players using the names given by the OfflineGameController method setPlayerName.

play, starts the loop that enables players to make moves.

Responsibilities

OfflineGame starts and stops a new offline game of Ringgz. It also manages when the player is done with his/her turn and calls the methods needed for the game to work properly.

Usage of other classes

Offline game uses the classes:

- Game
- OfflineGameController
- Board
- Player

Interface → GameView

The GameView interface is used by both our TUI and GUI gameViews. They both contain the same useful methods. The GameView is used to represent the game to the player. Since the method names speak for themselves we made a list with a small description.

Explanation

The methods:

- **printMessage**, prints the given message.
- **getXValue**, returns the x value given by the player.
- **getYValue**, returns the y value given by the player.
- **getZValue**, returns the z value given by the player.
- **getColour**, returns the colour given by the player.
- **showBoard**, shows the board.
- **setSecondaryColour**, sets the secondary Player colour.
- **setPrimaryColour**, sets the Primary player Colour.
- **getValuesSet**, returns the ValueSet made with the x y z and colour given by the player.
- **colourRing**,
- **setHintEnabled**, turns the Hint function on or off.

Responsibilities

Responsible for showing the player what the game needs to show. It also has the task to gather input from the user.

Usage of other classes

GameView uses the classes:

- Board
- Game
- Player
- Client

Client

The class Client is responsible for starting up the Client. It creates a client that will connect to the server with the given variables. The port number and ip address should be given for this.

Explanation

To give an answer to some of the server side protocol messages, there is a method called **handleMessage** which will handle the received messages from the server regarding everything that is not related to the game. Messages about a game will be forwarded to ClientSideGameHandler.

Responsibilities

Gives the player the possibility to play the game.

Server

The class Sever is responsible for starting up the server. It creates a server that is listening on a given port.

Explanation

To give an answer to some of the client side protocol messages, there is a method called **handleMessage** which will handle the received messages from the client regarding everything that is not related to the game. Messages about a game will be forwarded to ServerSideGameHandler.

Responsibilities

The Server gives the player the possibility to play the game over the network against other players with the same protocol.

ClientSideGameHandler

The class ClientSideGameHandler handles all messages sent by the Server/ServerSideGameHandler during a game.

Explanation

constructor creates a clientSideGameHandler with a player, board, game and a client and a view of the client.

To give an answer to most of the server side protocol messages, there is a method called **handleMessage** which will handle the received messages from the server. All the received are splitted and the argument at position zero is read. A client can be accepted, denied, a game can be started, a move can be made and a game can end. All these commands can be handled by this method handleMessage.

Responsibilities

The ClientSideGameHandler is responsible for handling all the messages from the server side and is therefore a very important class. If the messages aren't handled right, there can no game be started and moves cannot be notified.

ServerSideGamehandler

The class ServerSideGameHandler handles all messages sent by the Client/ClientSideGameHandler during a game.

Explanation
constructor creates a serverSideGameHandler with a Handler and a GameSession

The **handleMessage** method in this class gives an answer to all the received messages from the client. The method can start a new game, notify a move or give a game over. The gamerequest is the biggest message to handle. There will be checked if there are enough players available in the list of connected players. If this is the case, the names are received from the map and a new GameSession is created. It is also possible that there is only one player in the list. Then the server will not start a game.

Responsibilities

The ServerSideGameHandler is just like the ClientSideGameHandler an important class, because it handles all the messages regarding a game.

GameSession

The class GameSession is responsible for running the game on the server in a separate thread so that the server can support multiple games at the same time.

Explanation
constructor creates a GameSession with 2 - 4 names. Creates the amount of serverSideGamehandlers for each socket, sets up a game on the server in a new thread and sends out the first moverequest to the player.

Responsibilities

GameSession is responsible for running the game in a separate thread.

OfflineGameController

The class OfflineGameController is the controller for the offline game.

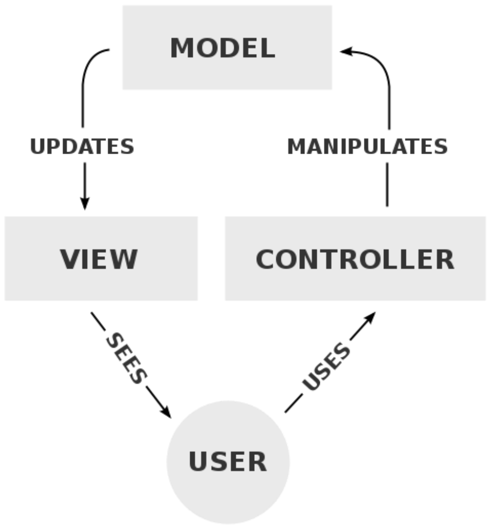
Explanation
The class creates the game, view, in, and playercnt for running a game.
The only methods in that class are getPlayers and setPlayerName to use the players

Responsibilities

The class is responsible to set-up and create an offline game.

The MVC pattern

The MVC pattern stands for Model-View-Controller Pattern and it's a software design pattern for implementing user interfaces in your application. The application is composed of:



- Model components:
  These components manage the behavior and the data of the application
- View components:
  These components determine the way in which the model is displayed
- Controller components:
  These components take input from the user and convert these to commands for the model or the view

In our design the classes are divided this way:

- Model components:
  Board, Game, Colour, Inventory, AiPlayer, HumanPlayer, Ring, Player
- View components:
  ClientView, TUIClientView, GUIClientView, GameView, TUIGameView, GUIGameView
- Controller components:
  Client, ClientSideGameHandler, Server, ServerSideGameHandler, GameSession, OfflineGame, OfflineGameController

Model components

The Board, Game, Ring and Colour are added to the model, because they all fulfill the role of managing the data. These classes are responsible for creating a board, filling a board with marks and checking the board for wins. The image above is also showing that the model is manipulated by the controller and that the model updates the View.

For updating the board, there is a class Observable and an interface Observer needed. The Observable informs the Observer when the state of the board is changed. Therefore Observable needs to be extended in the class Board.

Board contains the method setRingXYZ, that will change the state of the board. After setField is used, the board should be updated.

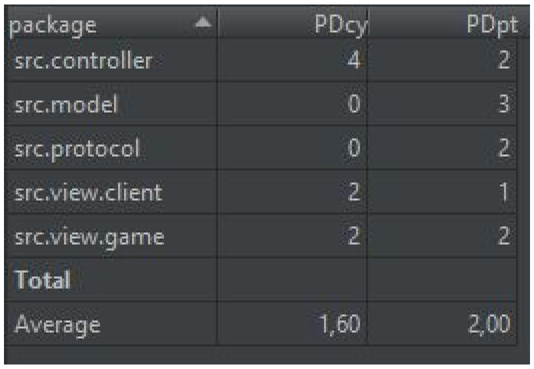
SetChanged sets the Observable's hasChanged attribute to true. After that the notifyObservers is called. If the hasChanged is true when notifyObservers is called, all the registered Observers will be notified and hasChanged is set to false again. This way the board will be updated every time

View components
The TUIGameView is the method that implements the interface Observer. The interface Observer contains a specification of the method update. When the Observer is notified of a change, the update method will be executed. In our case, we call the method showboard, that will return the representation of the board.

Controller components
The rest of all the classes belong to the controller components. All these classes handle messages from the user. The ClientConnection and the ClientHandler are handling the protocol messages and the Client, Server and GameSession are handling messages for setting up the client and the server. The ServerSideGameHandler is handling the messages for the moverequests.

Metrics Report

Afferent & Efferent coupling



package	PDcy	PDpt
src.controller	4	2
src.model	0	3
src.protocol	0	2
src.view.client	2	1
src.view.game	2	2
Total		
Average	1,60	2,00

PDcy: number of package dependencies

PDpt: number of dependent packages

The afferent coupling of the model package is the highest. This can be easily explained. The model package contains the class Board and the Game class. These classes together create the board and check the wins in a board. Other classes like the ServerSideGameHandler and the ClientSideGameHandler uses these classes often. Moves should be made and wins should be checked.

The efferent coupling is exactly the opposite of the afferent coupling. The efferent coupling of the controller package has the highest value. This is because the classes ClientConnection, ClientSideGameHandler and GameSession uses the Board and the Game class a lot.

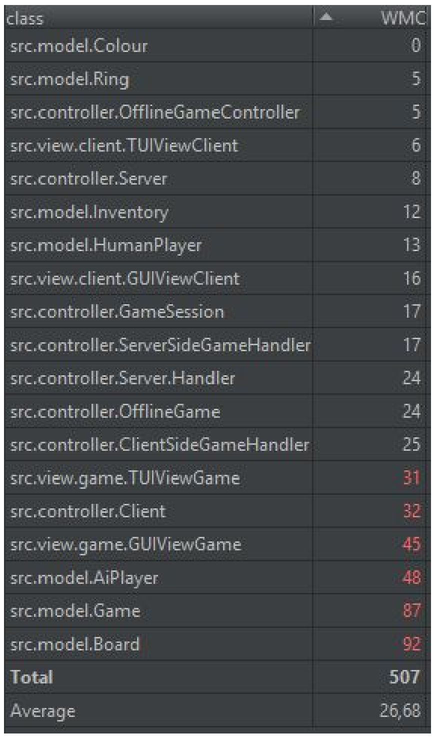
Consequences

The consequences of the dependency of the classes is that the controller classes stop working if there is are mistakes in the model classes. This is not really a big problem, because the Board and the Game class are well-ordered.

Weighted methods per Class

The weighted methods per class give an indication of how complex the code is. Every branch increase the weighted methods by one. The branches that will increase the weighted methods per class are: if statements, for statements , while statements, do statements, ternary operators, and the && and || expressions.

The weighted methods of the game, the board and the AiPlayer are the biggest.
This is because those classes use a lot of for and if statements. That will increase the
weighted methods per class. On the other hand, the GameSession, Inventory and
Server don't use a lot of these statements and therefore the weighted methods per
class is a lot lower



class	WMC
src.model.Colour	0
src.model.Ring	5
src.controller.OfflineGameController	5
src.view.client.TUIViewClient	6
src.controller.Server	8
src.model.Inventory	12
src.model.HumanPlayer	13
src.view.client.GUIViewClient	16
src.controller.GameSession	17
src.controller.ServerSideGameHandler	17
src.controller.Server.Handler	24
src.controller.OfflineGame	24
src.controller.ClientSideGameHandler	25
src.view.game.TUIViewGame	31
src.controller.Client	32
src.view.game.GUIViewGame	45
src.model.AiPlayer	48
src.model.Game	87
src.model.Board	92
Total	**507**
Average	26,68

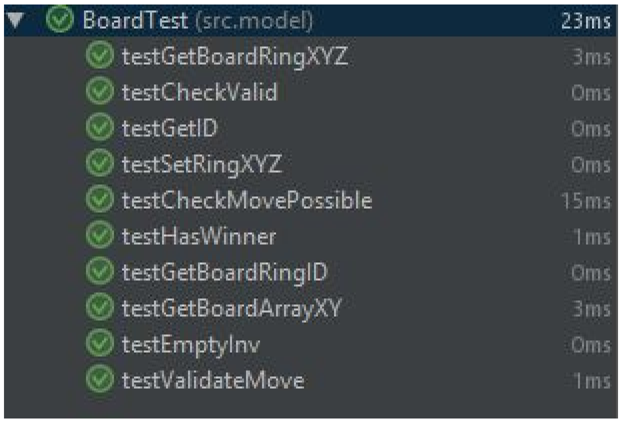
Consequences
You want to keep the weighted methods per class as low as possible. The complexity decrease if the weighted methods per class are lower. However it is not a big problem that the class Game for example has a high value of weighted methods. The class will be a bit more complex, but it was not really an option to split the class in two different classes.

Unit testing

The program is also tested. We tested Board, HumanPlayer and a test for game in general.

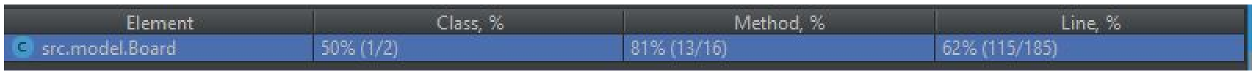
Board

The first test was a JUnit test to test the class Board. All the tested methods passed the test that was done. This was also expected.



```
BoardTest (src.model)          23ms
    testGetBoardRingXYZ         3ms
    testCheckValid              0ms
    testGetID                   0ms
    testSetRingXYZ              0ms
    testCheckMovePossible      15ms
    testHasWinner               1ms
    testGetBoardRingID          0ms
    testGetBoardArrayXY         3ms
    testEmptyInv                0ms
    testValidateMove            1ms
```

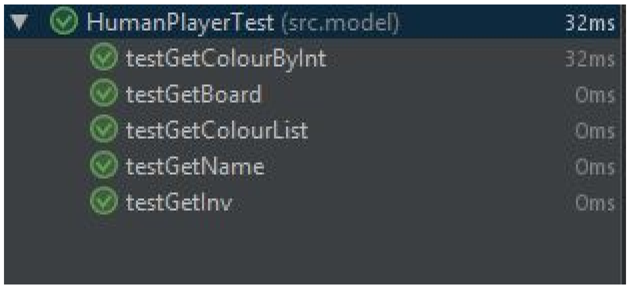
The total coverage of the test was 81%. Not all methods were tested like emptyBoard or reset.



Element	Class, %	Method, %	Line, %
src.model.Board	50% (1/2)	81% (13/16)	62% (115/185)

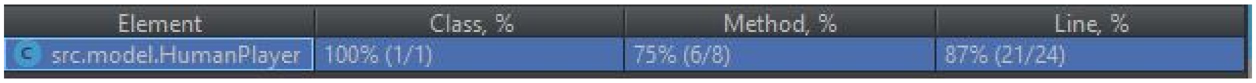
HumanPlayer

The next test was HumanPlayer. We tested human player to ensure that the specific player has only his colours in the inventory, etc...



```
HumanPlayerTest (src.model)    32ms
    testGetColourByInt         32ms
    testGetBoard                0ms
    testGetColourList           0ms
    testGetName                 0ms
    testGetInv                  0ms
```

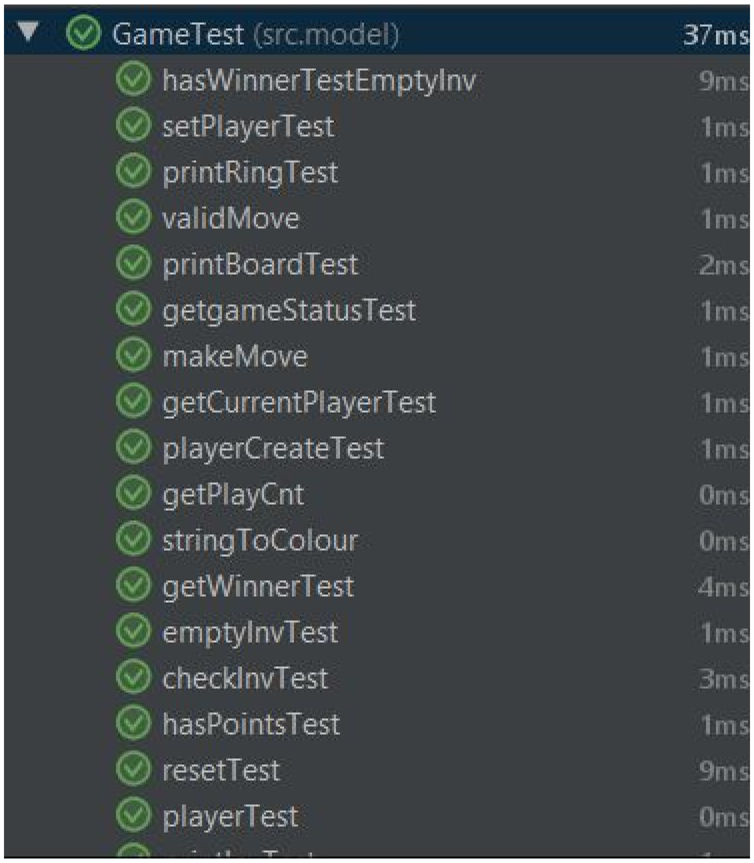
The total coverage of the test was 75%. Not all methods were tested like before.



Element	Class, %	Method, %	Line, %
src.model.HumanPlayer	100% (1/1)	75% (6/8)	87% (21/24)

Game Test

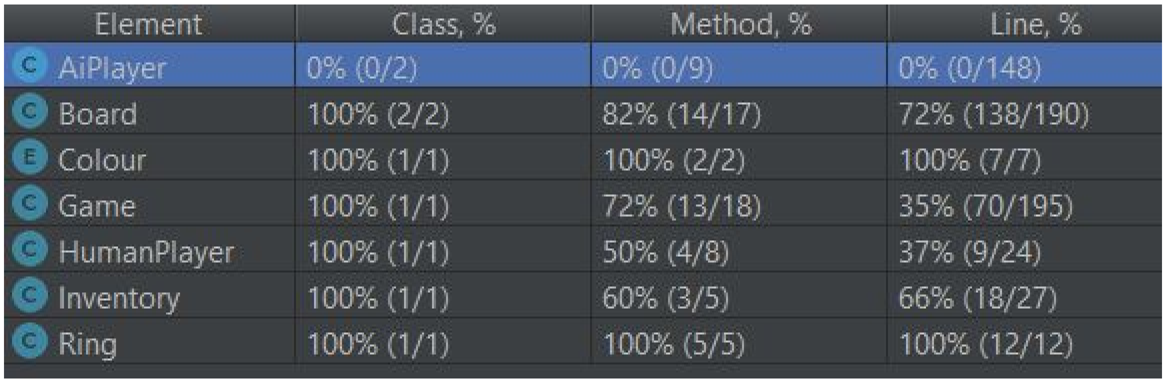
At last we tested our game class.



```
▼ GameTest (src.model)          37ms
    hasWinnerTestEmptyInv        9ms
    setPlayerTest                1ms
    printRingTest                1ms
    validMove                    1ms
    printBoardTest               2ms
    getgameStatusTest            1ms
    makeMove                     1ms
    getCurrentPlayerTest         1ms
    playerCreateTest             1ms
    getPlayCnt                   0ms
    stringToColour               0ms
    getWinnerTest                4ms
    emptyInvTest                 1ms
    checkInvTest                 3ms
    hasPointsTest                1ms
    resetTest                    9ms
    playerTest                   0ms
```

All tests methods passed and the coverage shows many classes that were tested with it.

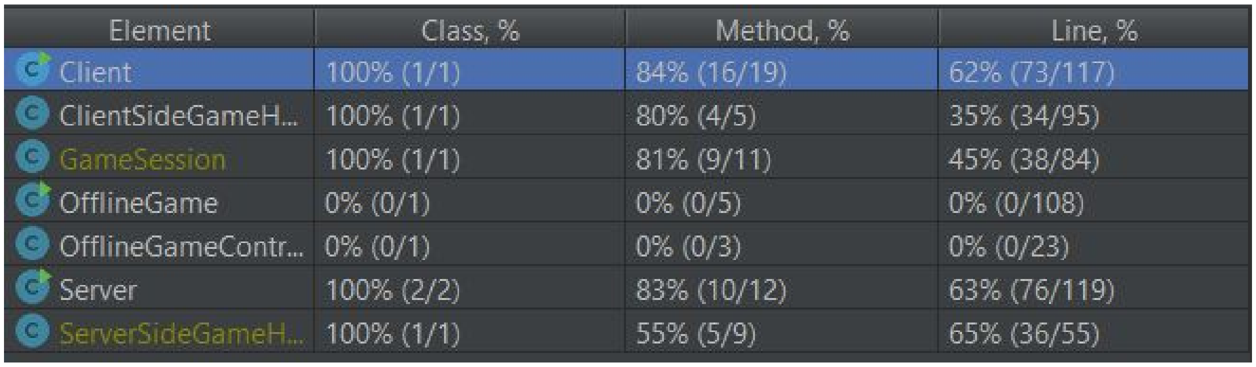
It tests game as well as Ring, Inventory, Colour.



Element	Class, %	Method, %	Line, %
AiPlayer	0% (0/2)	0% (0/9)	0% (0/148)
Board	100% (2/2)	82% (14/17)	72% (138/190)
Colour	100% (1/1)	100% (2/2)	100% (7/7)
Game	100% (1/1)	72% (13/18)	35% (70/195)
HumanPlayer	100% (1/1)	50% (4/8)	37% (9/24)
Inventory	100% (1/1)	60% (3/5)	66% (18/27)
Ring	100% (1/1)	100% (5/5)	100% (12/12)

System Test

It was not possible to test the client- and serverSideGameHandler with JUnit test so we decided to make a system check. We start a game and make all the different options. After that we check how much is covered. We started a game and made several checks. We typed in the same nickname for the second client, we made a move out of bounds and we made a move that was already full. This way we could test everything and check how much was covered.



Element	Class, %	Method, %	Line, %
Client	100% (1/1)	84% (16/19)	62% (73/117)
ClientSideGameH...	100% (1/1)	80% (4/5)	35% (34/95)
GameSession	100% (1/1)	81% (9/11)	45% (38/84)
OfflineGame	0% (0/1)	0% (0/5)	0% (0/108)
OfflineGameContr...	0% (0/1)	0% (0/3)	0% (0/23)
Server	100% (2/2)	83% (10/12)	63% (76/119)
ServerSideGameH...	100% (1/1)	55% (5/9)	65% (36/55)

The image above shows the coverage of the client and the server. The offline controller classes have 0% coverage because they are never used in the network game.

Reflection

Influence design project

Before we made this programming project, we already made a design project and we finished that well on time. The planning we made at that time was perfect and there was no reason to change the way of making a planning for the programming project.

Actual progress

The first week we planned to make all the specifications and approaches. This worked out pretty well. We started working with the implementation earlier than expected. The problem was that the beginning was a bit of a chaos, because we did not take enough time to make a good overview of all the classes we wanted to make. After a couple of days, everything became more clear and working on the implementation went better.
The documentation of the JavaDoc was done before we reached our own deadline. However we did not finish the JML specification on the deadline. This was something we did on one of the last days, but that wasn't a very big problem.

We planned to make the JUnit test classes during the programming, but we didn't do that. The JUnit tests were also made in one of the last days before handing in the project. Furthermore we made the report in one of the last days, but that was planned. It made it easier to make it, because we already finished most of the classes.
However the implementation wasn't totally done. The computer AI took a bit more time to make and the last client-server problems needed to be changed. Luckily we also finished that on time

Countermeasures

The JUnit test classes, the JML and the implementation of the computer AI took some days longer than expected. For that reason we needed to adjust the planning a bit and work a bit harder. The only way to compensate the deviation from the original planning, was to invest more hours in the programming project. Inter-Actief organised a project-evening to work on the project and in those hours we made a lot of progress.

Experience for the next planning

The planning we made was good and we did follow the planning as good as possible. We wouldn’t change a lot to the planning for the next time. It can be helpful to follow the planning a bit better, but that is only a small change for the next time.