Assignment 2

Formalities

This assignment may be solved in groups of up to 3 members. Each member must hand in themself. Attach a file naming the group members.

Deadline is Friday the 30th of November.

This assignment must be handed in an accepted, for you to go to the exam.

Topics for this assignment:

- RMI
- MVC
- Adapter
- Observer

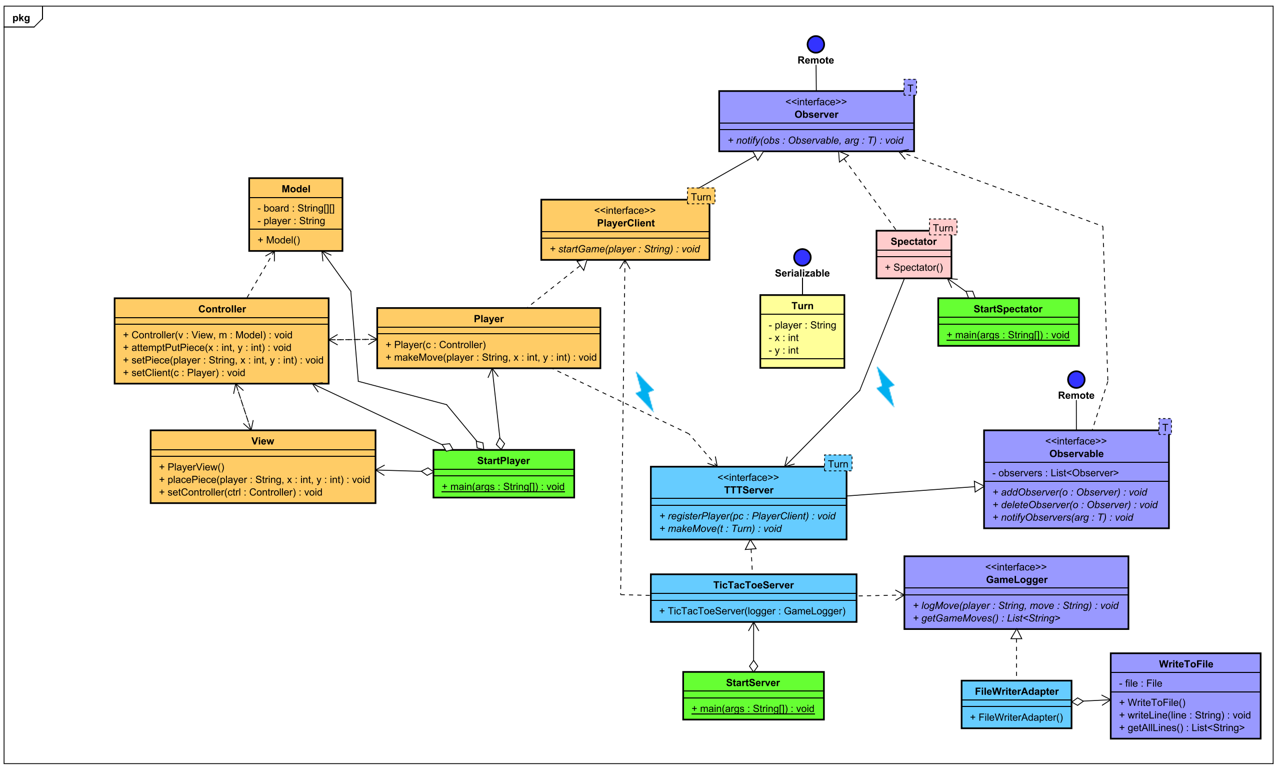
In this assignment you're going to implement a client/server system using RMI. It will be a game of TicTacToe (https://en.wikipedia.org/wiki/Tic-tac-toe) between two players, with an arbitrary number of spectators.
One type of client will act as a player's client to play the game, another type of client will act as a passive spectator. Both are observers, and the server will act as an Observable.
Whenever a player makes a move, that move is sent to the server, and the server will then notify all Observers currently listening.

The game has a logging mechanic to keep track of how the game went, as a sort of 'replay'. You're given a WriteToFile class, which you will have to adapt into your system.

Each player is represented by "X" or "O", or e.g. 0 or 1, or something else, that's up to you.
You must have the classes presented in the UML diagram below, but you're allowed to make changes, and do things differently, if it suits you better.

UML diagram

Here's a rough UML diagram:


pkg
Remote
<<interface>>
Observer
T
+ notify(obs : Observable, arg : T) : void
Model
- board : String[][]
- player : String
+ Model()
<<interface>>
PlayerClient
Turn
+ startGame(player : String) : void
Spectator
Turn
+ Spectator()
Serializable
Turn
- player : String
- x : int
- y : int
Controller
+ Controller(v : View, m : Model) : void
+ attemptPutPiece(x : int, y : int) : void
+ setPiece(player : String, x : int, y : int) : void
+ setClient(c : Player) : void
Player
+ Player(c : Controller)
+ makeMove(player : String, x : int, y : int) : void
StartSpectator
+ main(args : String[]) : void
Remote
<<interface>>
Observable
T
- observers : List<Observer>
+ addObserver(o : Observer) : void
+ deleteObserver(o : Observer) : void
+ notifyObservers(arg : T) : void
View
+ PlayerView()
+ placePiece(player : String, x : int, y : int) : void
+ setController(ctrl : Controller) : void
StartPlayer
+ main(args : String[]) : void
<<interface>>
TTTServer
Turn
+ registerPlayer(pc : PlayerClient) : void
+ makeMove(t : Turn) : void
TicTacToeServer
+ TicTacToeServer(logger : GameLogger)
<<interface>>
GameLogger
+ logMove(player : String, move : String) : void
+ getGameMoves() : List<String>
StartServer
+ main(args : String[]) : void
FileWriterAdapter
+ FileWriterAdapter()
WriteToFile
- file : File
+ WriteToFile()
+ writeLine(line : String) : void
+ getAllLines() : List<String>

The diagram is colour coded, with the following:

🟦 Server classes, you implement.

🟩 Classes with a main method to start either a server, or one of the clients.

🟪 Classes given to you by me.

● Interface from Java.

🟧 Classes belonging to the Player client

🟥 Classes belonging to the Spectator client

🟨 Container object to send information about a move

⚡ This icon marks a connection between a client and a server

You are allowed to add methods and functionality, which are not shown in the UML diagram, if you need to.

Class information

Turn

This class just contains basic information about where a player put his game piece, i.e. either “X” or “O”. I suggest representing the game board by a String[][] of size 2x2. This is optional, and you can do something else, if you like. But the Turn class is used to send information to the Server, which then notifies observers.

This class implements the Serializable interface from Java, so that it can be sent across the network.

Given classes

Observer

This is the interface given to you in the Observer session. It has a small modification, because it now extends Remote, the Java interface. That means all methods in the Observer interface must be declared to “throws RemoteException”.

Both clients are observers, so that they can observe the Server.

Observable

This is the interface given to you in the Observer session. It has a small modification, because it now extends Remote, the Java interface. That means all methods in the Observer interface must be declared to “throws RemoteException”.

It has the functionality to add, remove, and notify Observers.

GameLogger

This is the interface the Server uses to log the game information. Each move a player makes is logged through this interface, so that we can ‘replay’ the game later on. This interface is given to you.

WriteToFile

This is the class, you must use to write the game information to a file. It is a standardized class, that can just add a line of text to a file and retrieve all lines from the file. This class must be adapted to fit into your system, so we can use its functionality.

Player client

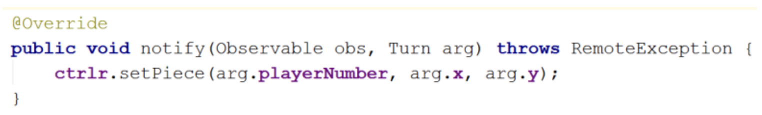
StartPlayer

This is a class with just a main method to start the Player client. How you instantiate all the classes on this client side is up to you. This class could instantiate a view, controller, model, and client, and give references to the relevant classes, or other classes can instantiate what they need. But the client program is started from this class.

Player

This is the client class. It is responsible for connecting to the Server using RMI. All communication to the server happens through this class.

It implements the PlayerClient interface, which enables the server to call methods on this client. We have a call-back system here. Because the PlayerClient extends Observer, the notify method must be implemented in the Player class:



```
@Override
public void notify(Observable obs, Turn arg) throws RemoteException {
    ctrlr.setPiece(arg.playerNumber, arg.x, arg.y);
}
```

The notify methods receives information about moves being made, and must react accordingly. It receives a Turn object, and must forward this information to the Controller. The controller then updates the Model, and tells the View to also update.

The Player class has a method which can send a player-move (I.e. a Turn object) to the Server.

The constructor is suggested to take as argument a Controller. This is optional, but the Player needs a reference to the Controller somehow.

The Player class is an Observer, and it will add itself as Observer to the Server (which is an Observable).

PlayerClient

This is the interface of the player client. This enables the server to call methods on the client.

It has the method startGame(player : String). The idea is that when two player clients have connected to the server, this method will be called on both servers. The parameter is the player piece, i.e. “X” or “O”. When this method is called, the game can start. You are allowed to come up with your own way of starting the game.

This interface extends Observer, meaning that the Player class will be an Observer, so that the Server can notify the Player about updates.

Model

The model contains the game board, and the player representation, i.e. “X” or “O”. The model keeps track of the current game state. Suitable methods to interact with the board should be implemented.

View

The view presents the game. It should be some kind of simple GUI. I suggest creating a JPanel, using a GridLayout and then just adding 9 buttons to the panel:

```
JPanel panel = new JPanel(new GridLayout(3, 3));
```

```
for (int i = 0; i < 9; i++) {
    JButton btn = new JButton();

    panel.add(btn);
```

When a button is clicked, a method on the Controller is called, e.g.:

```
ctrlr.attemptSetPiece(a, b);
```

Where *a* is the x coordinate, and *b* is the y coordinate.

The view has a method, called by the constructor, to update the presented board: placePiece(player : String, x : int, y : int). It will place the player, either “X” or “O”, at coordinates x, y, to update what the player sees.

You’re welcome to structure your GUI in any way you like.

Controller

This class is responsible for directing the flow of the client program, as well as the game logic.

When a button is pressed the controller must check

- Has the game started?
- Is it the players turn?
- Is it a valid move? I.e. is the cell already filled?
- Is the game over?

If all is okay, it will send the ‘move’ to the client (Player), which sends the ‘move’ to the server Server. The Server will then notify all Observers, which can then add this ‘move’ to the game board in the Model, and present the new state in the View.

The constructor is suggested to take parameters of types Model and View. This is optional, but the Controller must somehow get references to these two classes.

In order to keep track of whose turn it is, you can e.g. say that “X” always start. When the Controller receives a Turn update from the Client, it can check if the player (“X” or “O”) in that Turn is different from its player stored in the Model.

At the end of this document is a method, which can check if a game is over.

Spectator client

StartSpectator

Just a class with a main method to start the Spectator client.

Spectator

This is a passive class to spectate the game. When started, it will connect to the Server, add itself as an Observer, and get updates from the Server.

I suggest creating a JPanel using a GridLayout again, and then create 9 labels, which you can update, when the game is being played. Again, you’re welcome to do something else.

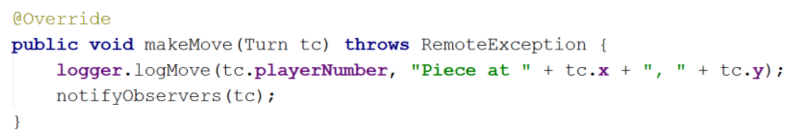
Server

TTTServer

This is the Server interface, making this a Remote Server. It extends Observable, so that the Server is Observable, and it can notify all interested clients about changes in the game.

It has a method so that a player can register itself at the beginning. When two Players have registered, it calls the startGame method on both.

It has a makeMove method. Whenever a Player makes a move, that information is sent to the Server through this method. The Server uses the GameLogger to log the move. The Server then notifies all clients, something like this:



```
@Override
public void makeMove(Turn tc) throws RemoteException {
    logger.logMove(tc.playerNumber, "Piece at " + tc.x + ", " + tc.y);
    notifyObservers(tc);
}
```

TicTacToeServer

This is the Server class. It is an Observable (by indirectly implementing the interface Observable).

The server is able to register players, and when two players have registered, the server must let both players know that the game can begin.

The Server receives Turns from the Players, and notifies all Observers about this Turn.

StartServer
A class with a main method to start the Server. Create the registry, put the Server instance into the registry, as usual.

FileWriterAdapter
This is the adapter, which adapts the WriteToFile to fit to the GameLogger interface.

Implementation
Again, try to break the system down into smaller tasks. Some parts can be created a tested individually.

You can create a GUI before anything else, and just check it looks right, and that the buttons print out the right information.

Start out by creating a simple client/server connection.

Do the Spectator and Adapter parts at the end. They're just 'attachments' to the system, and the core system doesn't rely on either.

Appendix

hasWon method to use in the Controller. Arguments are player, i.e. “X” or “O”, or whatever you use, and then the board, represented as a String[3][3] array. It will return true, if the argument player has won the game.

```java
public boolean hasWon(String player, String[][] board) {
    // check columns
    out : for(int x = 0; x < 3; x++) {
        for(int y = 0; y < 3; y++) {
            if(board[x][y] == null || !board[x][y].equalsIgnoreCase(player))
                continue out;
            if(y == 2) return true;
        }
    }
    // check rows
    out : for(int x = 0; x < 3; x++) {
        for(int y = 0; y < 3; y++) {
            if(board[x][y] == null || !board[x][y].equalsIgnoreCase(player))
                continue out;
            if(y == 2) return true;
        }
    }
    // check diagonal
    for(int i = 0; i < 3; i++) {
        if(board[i][i] == null || !board[i][i].equalsIgnoreCase(player))
            break;
        if(i == 2) return true;
    }
    // check other diagonal
    for(int i = 0; i < 3; i++) {
        if(board[2-i][i] == null || !board[2-i][i].equalsIgnoreCase(player))
            break;
        if(i == 2) return true;
    }
    return false;
}
```