Assignment 6: Game Server

Submission date: May 20th, 2016 at 16:00

Objective: Client-Sever Application using the Sockets. Serializing objects. Multithreaded programming.

Before you start, create a package assignment6 and copy Main.java from assignment5 to assignment6. All your code for this assignment should be defined under assignment6. We remind you that it is strictly forbidden to modify any file in the package basecode. Moreover, it is highly recommended that you do not modify anything under assignment4 and assignment5, unless it is for fixing BUGs.

1. Game Server

In this assignment you are required to extend the board games application with a new functionality to provide the possibility of playing over the network, i.e., several players from different machines can play a given game against each other. In particular, you are required to develop a game server that can be started in one machine to provide a service "*to play a specific game*", and develop clients that can connect to the server from other machines to play that specific game (in window mode) against each other. The new functionality should be completely transparent to the views developed in Assignment 5, namely, the code of Assignment 5 should not be changed at all.

The application is started in server or client mode depending on the command-line options. For example, to start the server a that allows playing *Ataxx* with two players we could use the following command-line:

```
java -jar pr6.jar -g ataax -p X,O --app-mode server --server-port 4000
```

This command starts a game server on port 4000 for playing *Ataxx* with two players `X` and `O` (if the list of players the default one is used, as in previous assignments). Once the server is up, clients could connect to the server to play the corresponding game using the command-line:

```
java -jar pr6.jar --app-mode client --server-host hostname --server-port 4000
```

where `hostname` is the name of the machine on which the server is running. Each client is assigned a unique piece from the list of pieces (that is provide in the `-p` option). Once the number of connected client is the same as the number of expected players (i.e., as the number of pieces), the server starts the game. Then clients can play against each other as follows: (1) they forward their actions (e.g, moves) the server which in turn forwards them to the corresponding game; and (2) the GameObserver notifications that are issued by the game should be forwarded by the server to the clients. Once the game started, if more clients try to connect to the server they should be rejected, and once the game is over, the server should allow clients to connect to start a new game. The server should open a simple Swing window in which it prints informative messages when clients connect, when the game starts, etc. This window should also have a button to allow stopping the server (i.e., quitting the application).

You should use the following command-line options for the new functionality:

- --app-mode or -am: it indicates in which mode we want to start the board games application. It takes one of the values `normal`, `client` or `server`. The default value of this option, i.e., if not provided, should be `normal`.
- --server-port or -sp: it indicates on which port the server should be started (when used together with -am server) or on which port the server is listening (when used together with -am client). The default value of this option, i.e., if not provided, should be 2000.
- --server-host or -sh: it takes one parameter that corresponds to the name (or IP address) of the machine on which the server is running (i.e., it is used together with -am client). The default value of this option, i.e., if not provided, should be "localhost".

2. Automatic Players in Window View (OPTIONAL)

2.1. Part I

In the previous basecode, automatic players were programmed to wait for sometime and then return a random move, let us call them "dummy" players. The new basecode has a new functionality that allows using the `MiniMax` algorithm to make an automatic move. This algorithm explores the state space and decides what is the best move for the current player – it relies on the method evaluate of the corresponding GameRules to evaluate how good a state is. See MinMax.java for more details. Note that the basecode has already has an evaluate implemented of *Connect-N* (which we also use for *Tic-Tac-Toe* and *Advanced Tic-Tac-Toe*). The following command-line options can be used to activate this functionality:

- --ai-algorithm or -aialg: it indicates which algorithm we want to use for the automatic player. It can be minmax for the MinMax algorithm, minmaxab for the MinMax algorithm with $\alpha\beta$-pruning (has better performance), or none for the "dummy" automatic player. If this option is not provided the dummy player is used.
- --minmax-depth or -md: it indicates the maximum depth of the search tree constructed by the MinMax algorithms. Larger numbers provide "smarter" algorithms but they are less efficient (since the tree is larger). The default value is 3.

In this part you are requested to copy this new functionality from Main.java of basecode to Main.java of your assignment (you only need to copy the methods that process the above command-line options). You are also requested to implement method evaluate for the game *Ataxx*. For example, you could use the following naive strategy:

```
public double evaluate(Board board, ..., Piece p) {

   double m = 0;
   double n = 0;

   // 1. let 'n' be the number of pieces of
   //    kind 'p' on 'board'
   // 2. let 'm' be the number of pieces not
   //    of kind 'p' in 'board' (excluding obstacles)

   double total = n+m;
   return (n / total) - (m / total);
}
```

2.2. Part II

Currently, in the window view, when you make an automatic move you must have noticed that the GUI becomes unresponsive until the move is completed. This is because the automatic player "sleeps" for some time (to simulate a calculation that takes some time) and then returns a random move. Therefore, while it is sleeping, the event-dispatch thread of Swing is blocked and thus the GUI becomes unresponsive.

In this part you are requested to improve this, making the GUI responsive, by changing the window view such that calls to ctrl.makeMove(p) are done in a thread that is different from the event-dispatch thread of Swing. This can be done either by creating a new thread that executes the call to ctrl.makeMove(p), or better by using a SwingWorker (to avoid creating a new thread for every call).

2.3. Part III

In this third part you are requested to change the behavior of automatic moves (only in window view) to the have the following: if the automatic player does not return a move in X seconds, an error is issued and the mode of the current player is changed from *automatic* to *manual*. The value of X can be predefined, e.g., 5 seconds, or you can add a swing component to the view where the user can control this number.

In your implementation you want to rely on that fact that *automatic players*, both the "dummy" one and the one that uses the MinMax algorithm, immediately return null if the thread in which they run is interrupted –see DummyAIPlayer.java and AIPlayer.java.

3. Submission of the Assignment

The assignment must be submitted as a single compressed archive via the Campus Virtual submission mechanism not later than the date and time indicated at the start of this document. This archive must contain the following elements:

- A directory called src, containing the Java source code of your solution to the assignment.
- A file called students.txt, containing the names of the members of your student group.
- A directory called doc, containing the API documentation in HTML format generated automatically from the Java source code of your solution (using the *Javadoc* tool).