Luleå University of Technology
Computer Science
Håkan Jonsson

February 20, 2012

Global Gomoku

– Lab 4 in D0010E –

1 Introduction

Modern forms of communication are more and more carried out over the Internet, everything from streams of sound from radio stations to video-on-demand and on-line games. In this lab you will get some insight into how the Internet can be used for communication in the form of an old board game: Gomoku.

2 Gomoku

Gomoku is an abstract strategy board game for two players. It is traditionally played with go pieces (black and white stones) on a go board (19x19 intersections); however, once placed, pieces are not moved or removed from the board. Gomoku may also be played as a paper-and-pencil game on a sheet of paper.

Black plays first, and the players then alternate in placing a stone of their color on an empty intersection. Such placements are referred to as *moves*. The winner is the first player to get an unbroken row of five stones horizontally, vertically, or diagonally.

Gomoku is also known as *luffarschack* in Sweden and is played by putting crosses and circles on a squared piece of paper.

3 Assignment: Implementing Gomoku

In the traditional setting, both players sit on each side of a common board. In this lab you will write a program that makes it possible to play Gomoku over the Internet. Your program should be written according to these instructions, a UML diagram (Figure 2), and Javadoc comments.

The program should contain a graphical user interface (GUI) that displays a board, the stones placed, and a text describing the current state. The GUI should also provide interaction with the player so that it is possible to place stones by mouse clicks on the board. Part of the program handles communication over the Internet; this part is mainly given in advance in the form of JAR files. The GUI should contain buttons for connecting to another host, restarting the present game, and disconnecting from the other host.

Note that an important distinction is made between the two players involved in a game. In each running program, "me" refers to the player that affects the game by mouse clicks in the GUI. The other player, the one whose actions are received over the Internet and who sits at another computer, is referred to as the "other" player. (In fact, "me" at one computer is the "other" player at the other computer, and vice versa!)

Since this is not a computer communication course, you will be provided with the classes that take care of communication between players. They are stored in a JAR file `Gomoku.jar` that is available from the course page. You will also be provided with "class stubs" that will be a starting point for you when writing some classes. They can be found at the course page as well.

4 Main Program

Start by creating a new Java project `lab4` in Eclipse, download the file `Gomoku.jar` from the course home page to the directory of the project, and then add it to the build path of your project.

Like all stand-alone programs written in Java, our program has to include a class with a static public method `main`. This class should have the name `GomokuMain` and lie in the package `lab4`.

The argument of `main` should be a *port number*. Ports are used in Computer Communication contexts and specify a kind of address at an Internet host. One computer can read from and write to "ports". We will use it to hook up computers to each other. A port number is a positive integer.

In `main`, three objects are created: a `GomokuClient`, to which the port number is given as an argument, a `GomokuGameState` that takes the client as its argument, and a `GomokuGUI` that takes the game state and the client as its arguments. In addition, `main` should also contain code that checks that there is exactly one argument. If there is no argument, the port number should be set to a default value of your choice but at least 4000.

5 A Given Client

Apart from the class that contains `main` that you have to write yourself, there is also code that is given. The class `GomokuClient` in the package `lab4.client` is one of the classes included in the JAR file. It handles communication and calls methods in the game state class based on the messages it receives from the other player. In practice, other parts of the program go via `GomokuClient` to reach the other player's computer.

6 Model – View – Controller

The design of the rest of the program can be found in a UML diagram (Figure 2). It is influenced by the *model – view* pattern (where the model part includes the controller) that was also used in Lab 2. The idea behind this way of programming is to separate the model from views of the model, thereby increasing the flexibility of further programming (for example, it should be possible to change the view without changing the model).

In this lab, the model is an object that holds all relevant data that describes a game in progress. The view is a window showing the board, previous moves, an indication of whose turn it is to move, a message that tells what happens, and three buttons. In addition to these, there is also a part handling communication between the host computers.

6.1 The Model

The model is an object of the class `GomokuGameState`. Such an object contains a reference to an object of the class `GameGrid`, which is also a part of the model.

GameGrid

An object of this class represents a board on which players can make moves. *In addition* to what the UML diagram (Figure 2) and the Javadoc (see the course home page) stipulate, the following should be observed:

- The class belongs to the package `lab4.data`.
- The board is 2-dimensional and has as many rows as it has columns. Declare a 2-dimensional array of integers for representing boards. The actual size is given as a parameter to the constructor. If n is the size of a board, the board contains n^2 squares[1].
- Each square is either
  1. empty,
  2. occupied by the player "me", or
  3. occupied by the player "other".

  Declare three public static integer constants `EMPTY`, `ME`, and `OTHER` that stand for these square contents. (If you know about *enumerations* in Java 5.0, you are welcome to use them instead of all integer constants in this lab, where appropriate. However, this is not a requirement.) Initially all squares are empty.

[1] Instead of placing stones at the *intersections* that form corners of squares we place them *inside* the squares.

- The number of filled squares in a row (horizontally, vertically, or diagonally) that is required to win should be defined as a constant `INROW`. This means that if we like to play a game where the goal is to obtain a number of filled squares in a row that is different than 5, we should only need to change this constant.
- The method *isWinner()* should use `INROW` when it checks if the board contains a winning row of filled squares.
- The method *move()* makes a move but only if the square to which the move should be made is empty. It then returns *true*. Otherwise it returns *false* to signal to the caller that the move could not be made.
- The methods *move()* and *clearGrid()* should notify potential observers if they succeed in changing the board. Since the class extends Observable, it inherits the methods suitable for doing this: *setChanged()* and *notifyObservers()*.

GomokuGameState

An object of this class represents the game state during play. *In addition* to what the UML diagram (Figure 2) and the Javadoc (on the course home page) require, the following should be observed:

- The class belongs to the package `lab4.data`.
- The state of an object of this class should contain:
  1. A variable that refers to an object of the class `GameGrid`. This is the board of the game.
  2. A constant `DEFAULT_SIZE` that stands for the size of the board (the number of rows). 15 is a suitable value for this constant. If we like to play on a board of a different size, all we should have to do is to change the value of this constant.
  3. Four integer constants representing the different game states:
     - `NOT_STARTED` when no game is going on,
     - `MY_TURN` when the game is being played and it is "my" turn,
     - `OTHERS_TURN` when the game is being played and it is the "other" player's turn,
     - `FINISHED` when a game has been played and one player has won.

     (Once again, if you know about *enumerations* in Java 5.0, are welcome to use them instead of all integer constants, but this is not a requirement.)
  4. A variable `currentState` that at any given moment in time is equal to one of the four possible game states above. This variable is used and changed by methods in the class only.
  5. A string variable `message` that holds a text that should be displayed below the board. Whenever something essential happens in the game (a move is made, a new game is started, the other player disconnected etc), `message` should be set to a suitable text that the GUI can then show for the player "me". The class should notify any observers whenever the message is changed.
  6. A variable `client` that refers to the `GomokuClient` of this game. The client is an object that is responsible for communicating with the "other" player.
- The method *move()* carries out a move the player "me" makes, if possible.

  This method is called whenever the player "me" clicks on the board. It has to make sure that there is a game in progress, that it is "my" turn, and that the square clicked on is empty. If this is not the case, `message` should be set to a suitable error message, observers should be notified so they can display `message`, and the method returns.

  If, on the other hand, the move is legal, this must be communicated to the "other" player by a call to *sendMoveMessage()* of the client (and `message` set accordingly). It must also be checked whether the player "me" has won the game by the latest move, and the game state should be changed accordingly. Finally, observers should be notified.

  The method *move()* in `GameGrid` should be used to make a move. Note that it signals back to the caller whether the move was possible to make or not. If it was possible, the board is updated to reflect the move made.

- The method *receivedMove()* is called when the "other" player has made a move. The board should be updated and we must check whether the "other" player has won the game by the latest move. The game state and `message` should be changed accordingly. Finally, observers should be notified.
- The method *disconnect()* is called when the player "me" clicks the button "Disconnect". The board should be cleared, the game halted so that no game is going on, `message` set, observers notified, and the method *disconnect()* in the client called.
- The method *otherGuyLeft()* is called when the connection to the "other" player is lost. The board should be cleared, the game halted so that no game is going on, `message` set, and observers notified.
- The method *receivedNewGame()* is called when the "other" player has clicked on the button "New Game". The board should be cleared, the game state changed so that it is "my" turn, `message` set, and observers notified.
- The method *newGame()* is called when the player "me" clicks the button "New Game". The board should be cleared, the game changed so that it is the "other" players turn, `message` set, the method *sendNewGameMessage()* in the client called, and observers notified.
- The method *update()* is called to determine the initial game state when a game starts. It must not be altered in any way.

Note that the class extends the class Observable, which means that it inherits the methods *setChanged()*, *notifyObservers()*, and *addObserver()*.

6.2 The View

The view of the model consists of a window that shows a board, a message, and three buttons. Buttons and the board are clickable. The implementation of the view can be found in the class `GomokuGUI` and the class `GamePanel`.

GamePanel

A game panel is the view of a board. An object of the class *observes* a `GameGrid` object and displays a graphical representation of the board.

- This class belongs to the package `lab4.gui`.
- `UNIT_SIZE` is a constant that defines the side length of a square on the board as measured in pixels. Write the class in such a way that you only have to change this constant if you want to change the size of the squares.
- The method *getGridPosition()* maps between pixel coordinates (where the player "me" clicks) and a square on the board.

  Hint: Use `UNIT_SIZE` for the conversion.
- The method *paintComponent()* draws a picture of the board and all moves made. You are free to come up with a design of your own (as long as it works).

GomokuGUI

An object of this class presents a view of a game and three control buttons together with a text field with a message to the player "me". It also defines listeners that react to mouse clicks on the board and the buttons. Note that mouse clicks outside the board and the buttons should be ignored!

- This class belongs to the package `lab4.gui`.

- The constructor of the class should contain code that creates a JFrame, components, and a layout of the components. The components are a `GamePanel gameGridPanel`, a JLabel `messageLabel`, and three JButtons: `connectButton`, `newGameButton`, and `disconnectButton`.

  The layout should be as follows. The status field `messageLabel` should lie at the bottom. On top of it three buttons should lie ordered next to each other: `connectButton`, `newGameButton`, and `disconnectButton` from left to right. Finally, the `gameGridPanel` should lie on top of the buttons.

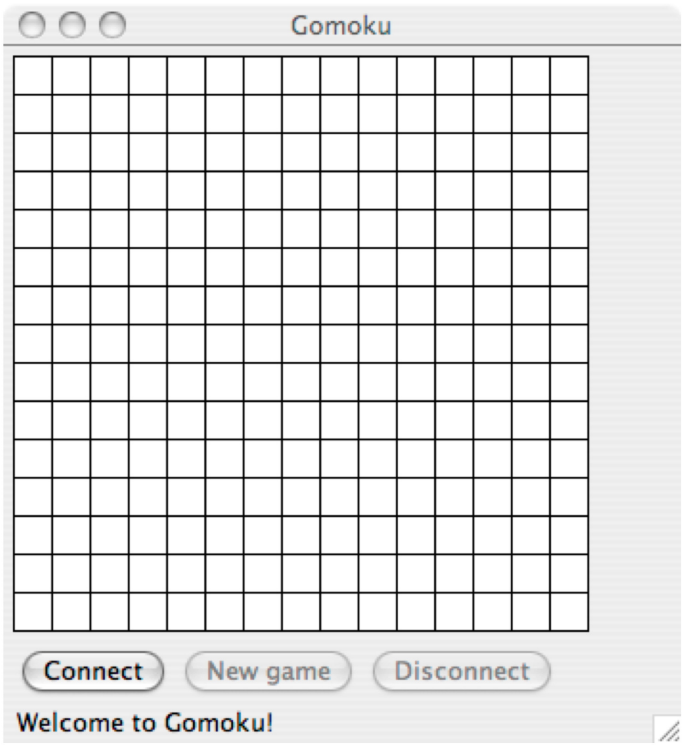

Figure 1: How the GUI can look like.

Hint: You can use Boxes created with static methods *createVerticalBox()* and *createHorizontalBox()*. Alternatively, you can use SpringLayout. Se Fig. 1 for an example where the buttons have been given proper labels.

There should be one or more listeners connected to the game panel and the three buttons in order to have the program respond when the user clicks on these components.

gameGridPanel Use an *anonymous* class that extends MouseAdapter as a listener. Program its method *mouseClicked()* to first map the pixel coordinates where the player "me" clicked to a square on the board, and then call *move()* in the game state. The net effect of this is that whenever the player "me" makes a move by clicking a square, the square on the board will be assigned the value `ME`, if the square is not already occupied.

connectButton For buttons, an ActionListener is the appropriate choice of listener. Add an *anonymous* listener of this type to the button and make its method *actionPerformed()* create an object of the class `ConnectionWindow`. That class has already been written. When playing the game, you should specify the IP address to the player you want to connect to as well as the port number that player is listening to. Instead of an IP address you can use `localhost` to connect to a game running on the same computer.

newGameButton Add an *anonymous* listener of type ActionListener and make its method *actionPerformed()* call the method *newGame()* that `gamestate` has.

disconnectButton Add an *anonymous* listener of type ActionListener and make its method *actionPerformed()* call the method *disconnect()* that `gamestate` has.

7 Presenting the lab

Before presenting the lab, make sure that

- the program works as expected (try to place the winning sequence at different places on the board);
- you have written clear *Javadoc* for all classes and public methods; you should also write short package descriptions (plain text inside HTML "body" tags) and put them in files `package.html` in the corresponding source directory;
- you understand all the code (even the part written by your lab partner).

Study the UML diagram on the next page.

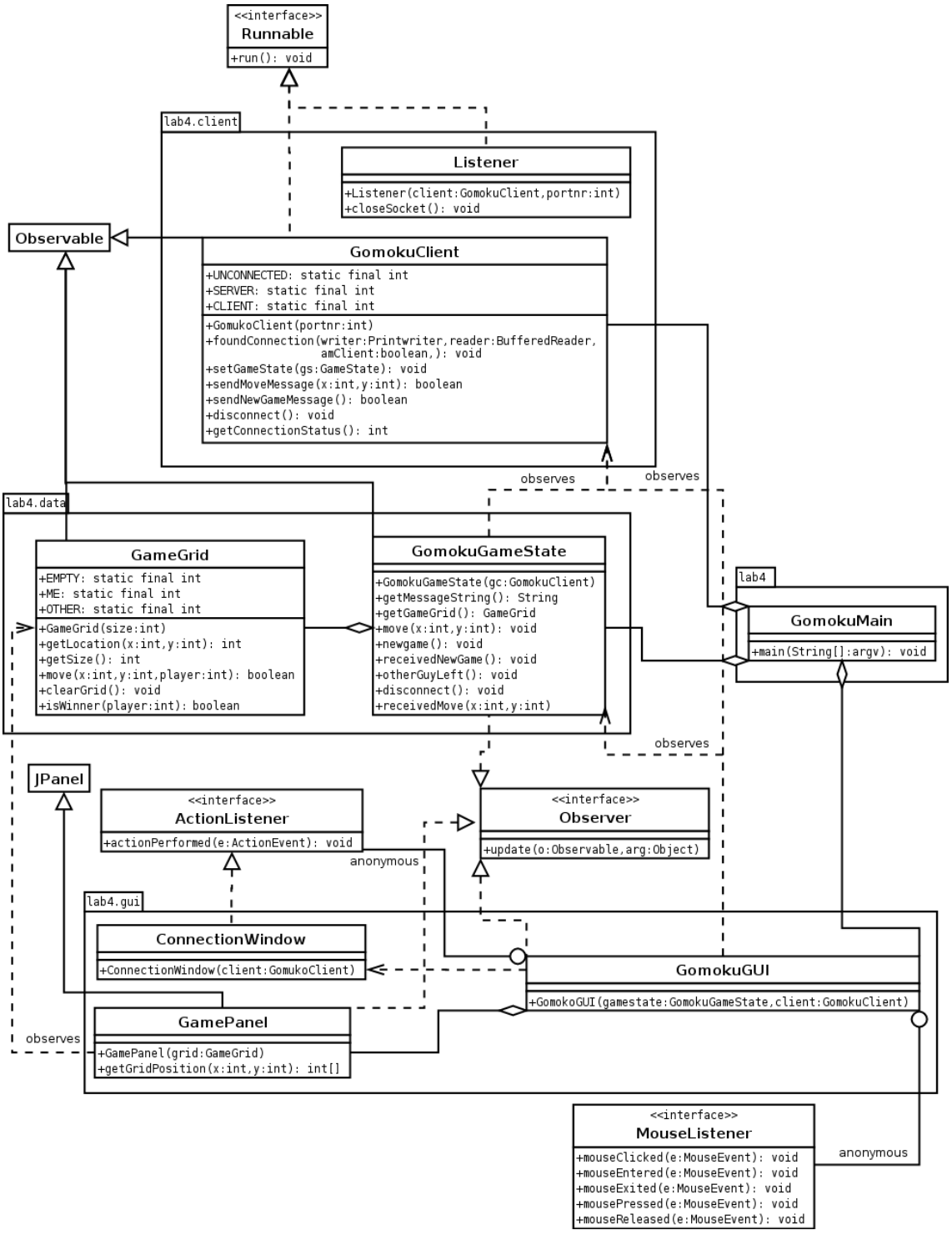

Figure 2: UML for lab 4.