12 Model View Controller

In this problem set you will learn about the Model-View-Control design pattern for separating the concerns when designing programs with GUI views. You will implement a model for a game (Connect Four) and for a simple calculator. Optionally, you will design a user interface for your model.

When designing programs with user interfaces, the main concern is in making sure that the computer performs the desired tasks correctly. This is the *model* for the program, the brains behind the observed behavior.

It is possible to design several different representations of the model's operations, the *views*. For example, a result of a sorting algorithm may be printed out as a list of values, saved to a file, shown as a bar chart, or, even be played as a musical scale. User selecting a state when entering an address into a GUI form may be given a scrolling list, a drop-down menu, a text field into which the information is typed, ar a choice of radio buttons, one for each state.

The connection between the model and one or more views of the values represented by the model and model's behavior is the *controller* - the glue between the user and the program.

While your assignment involves all three parts, the key focus is on the model part of the program. The interactions with the user are not the required part of this homework. You are encouraged to do at least some of the extra credit parts to gain some experience with the user interface design.

Part1: Connect Four

Overview

The game Connect Four is played with a vertical board that has seven columns of slots, each designed to hold up to six pegs. Two players take turns inserting their pegs (red or black) into the columns, one peg per turn. The player who manages to connect four of her pegs in a line (vertical, horizontal, or diagonal) wins the game. The game ends in a tie if the whole board gets filled without either player connecting four.

Here is a diagram of possible winning positions:

```
/*
 *  o o o o o o o
 *  o o b o o b o
 *  r o r b r b o
 *  r r r r b b o
 *  r b r b b b b
 *  r r b r b r r
 *
 * wins: black row (1, 3) (1, 4) (1, 5) (1, 6)
 *       black column (4, 5)
 *       black nw-se (4 , 2) (3, 3) (2, 4) (1, 5)
 *       black ne-sw (3, 5) (2, 4) (1, 3) (0 2)
 *       red   row (2, 0) (2, 1) (2, 2) (2, 3)
 *       red   column (3, 0)
 *       red   ne-sw (3, 0) (2, 1) (1, 2) (0, 3)
 *       red   nw-se (3, 4) (2, 3) (1, 2) (0, 1)
 */
```

Of course, the game will never progress to this point, as the first player that connects four wins and the game ends.

Download the files in the ConnectFour.zip bundle and create a project. Add jpt.jar library as you did before. Run the program starting with the Interactions. It allows you to play several games in a row, however, the computer does not play very smart game. Your task is to make computer play a smarter game, and optionally, design a graphical user interface that would display the board as graphics, allow the user to select the column through buttons or mouse clicks, or numeric input into a GUI field, and possibly record the tally of several games played in succession.

The Existing Files

The program you downloaded consists of six files as follows:

- **class** *Board* that represents the current state of the game and allows the player/computer to ask whether a given position is occupied by a peg of given color, add a peg to the board, and determine whether adding a peg of the given color at a specified position results in a win.
- **class** *ConnectFour* is the administrator of the game. It contains the code to start the game and loops between user input and computer generated moves until the game is finished. The computer generated move is just a random choice among the available columns.

- **interface** *Testable* that contains just one method *test* that consumes a *Tester* object.
- **class** *Tester* that is the test harness.
- **class** *Examples* that implements Testable and contains the tests that are run by the test harness.
- **class** *Interactions* that contains a method that just creates a new instance of *ConnectFour* and thus plays the game.

There are two ways to make the computer smarter. The first one is to design some way of measuring the value of different moves, based both on the position and on the state of the board at the given moment, and have the computer select the move with the best value. The other method is to learn from experience, by remembering which of the moves were good and which were bad - with some level of goodness or badness, and use that experience in later games. The first method works for playing a single game at a time, the second method requires that the computer plays several games in succession before it learns to be smart.

12.1 Problem

Currently, the *Board* has no way of determining that the game ended in a tie. Add a method *checkTie* that determines whether the result is a tie. Modify the appropriate places in the **class** *ConnectFour*, so that the program correctly reports a tie.

12.2 Problem

Your main task is to design a way for computer to select a good move, based only on the current state of the board. You may collect all this wisdom in a separate class, possibly extending the **class** *Board*.

Include a brief description of your strategy along with your documentation, so that the reader can understand what decisions your program made along the way. Make sure you inlcude tests for all the methods you design.

12.3 Problem: Extra Credit

Design a computer player that learns from its experiences. To do so, you must also modify the game controller, so that the computer can play several games in a row. It should also keep the statistics about the number of wins, losses, and ties.

12.4 Problem: Extra Credit

Desing a graphics user interface for the game, using either the *World* or the **JPT** library, or plain Java.

Part 2: Calculator

Pick up a calculator. Enter some numbers, press some operator buttons, observe what is going on. Each button press corresponds to some action — a method invocation in the program that controls the calculator. The display represents a part of the information that the calculator engine remembers, some other information is hidden and only comes to play several button presses later.

12.5 Problem

Design the class Calculator that contains a method for each of the following buttons: digits, operators (plus and minus), the clear button, and (optionally) the equal sign button. The calculator must contain a field display that represents the current value shown in the calculator register.

You are not allowed to use any if statements in your program. You may, however, design additional classes as needed.

Follow the design recipe. Make sure you include extensive tests cases.∎

12.6 Problem

Add the multiply and divide buttons to your calculator.∎

12.7 Problem: Extra Credit

Desing a GUI for your calculator. It is OK to 'steal' code written in some books (with proper citation and reference to the original), but make sure you modify it to follow true object oriented style, without excessive if statements, with clear separation of responsibilities among the classes and methods. ∎