Lab 5

4 Oct 2007

Next, we'll move on to a portion of the class where we play games. Well, actually, we'll write games and then we'll write *programs* to play games.

Task 1: Comprehend the code, part 1

First, another code-reading exercise. In the `lab5` subdirectory, I've placed a few Java files that you'll be adding to. The only reason I wrote them is to save time; there's nothing in there that should be mysterious to you. So, first, look them over and see if you understand how they work. These are the classes that are in there:

- `TicTacToe` represents a game of tic-tac-toe. It doesn't actually play the game itself; it's basically just up there marking down the Xs and Os while the player calls off grid locations.
- `GameRunner` actually plays the game, although it's really, *really* dumb about how it does so. (The next few days of class discussion will involve how we could improve on this, but for today's lab we're not worrying about it.)
- `GameState` is relevant to the second part of the lab. Skip it for now.
- `Pair<X>` represents a pair of values of the same type, and is hashable. In the context of TTT it is storing Integers representing (respectively) the row and the column.
- `PlayerID` is for representing whose turn it is, whose marker is in a given board square, and who won.

As it stands, everything in there compiles, but if you run it (by running the `GameRunner` class) it will throw an exception. Once you've read through the code and have a handle on it, go on to the next section, where you'll fix this.

Task 2: Finishing the code base for TTT

There are two methods in `TicTacToe` which are incomplete: `isValidMove` and `nextState`. There are lots of values for a `Pair<Integer>` that do not represent valid TTT moves at all, and a few that might be valid but not in the current game state—the `isValidMove` method should return `false` for all of those, and `true` for the values of `move` that are actually valid right now.

For `nextState`, I've already put in the code that checks whether a move is valid (by calling `isValidMove`), makes a copy of the current state, and returns it, but that means that right now it just returns something identical to the current state! Fix it by adding code that modifies the *returned* value to represent the new state. *Do not modify `this` in any way.*

Then go ahead and run `GameRunner`. If all is well, it should spit out, in text format, a game of tic-tac-toe that runs until the board is full. It happens that this game would be a draw. You would certainly be capable of writing `isGameOver` and `winner` at this point, but that's not what I want you to do for this lab.

Task 3: Comprehend the code, part 2

You may have noticed that `GameRunner` didn't actually have to know anything at all about how `TicTacToe` worked, other than that its moves were of type `Pair<Integer>`—and even at that, it didn't need to actually look inside the `Pair`. This is a convenient way to divide the work between classes that implement the game mechanics and classes that do the work of playing the game.

So now go back and look at `GameState`. It's an interface that boils down all the information that `GameRunner` needs from a specific-game class to actually play the game. Look it over and compare with the `TicTacToe` implementation to understand what each method does and is supposed to do. Ask questions if there's anything you don't understand.

Then move on to the next task.

Task 4: Implementing a different game

Connect Four is a game that is in some ways very similar to tic-tac-toe: it's played on a grid, with each player taking turns to place tokens, trying to make a line (in any direction). The CF grid is bigger—seven columns and six rows, in the standard version marketed by Milton Bradley. To win, you need four in a row, rather than three. And the only valid moves are those that place tokens at the *top* of a non-full column. (In the face-to-face version, the board is mounted vertically and tokens dropped in, so that physical gravity enforces this rule.)

Using the `TicTacToe` class as a guide or inspiration, implement a `ConnectFour` class that also implements `GameState`. You will have to think about what the `MoveType` should be, and how to detect a valid move, and how to update the grid.

Because `GameRunner` is not itself generified, you will have to go in and update the types to make it run with your `ConnectFour` class. And again, it's not likely to play very well—but it will demo your code just fine.

As in the first part of the grid, writing `isGameOver` and `winner` is outside the scope of the lab. If you do finish, start thinking about how you could modify `GameRunner` to pick the "best" next move.