Assignment 7: Pawns on the World Stage – Pawns Board, part 3

Due dates:

- **Implementation: Friday Apr 04 at 8:59pm**
- **Self-evaluation: Released Saturday, Apr 05 at 09:05pm, Due Sunday, Apr 06 11:59pm**

1 Purpose

In this portion of the project, you will be building a controller for your game, listeners for various features interfaces to connect the components, and ultimately building a playable, complete game. Read through the entirety of this assignment before diving into coding, as the suggested designs here may influence how you structure your work.

2 Preparation: cleaning up prior code

You should likely have received design feedback about your model and/or view implementations. Incorporate that feedback, as well as complete any missing functionality from the prior assignments.

In the previous assignment, you implemented selecting grid cells and cards via mouse-clicks and then confirming moves or passing with key presses. Nothing in particular needed to happen in response to those choices *yet*, but in this assignment we need to connect those choices to the controller. If you have not done so already, design an observer interface describing those *player actions* the view publishes, and enhance your view with the ability to add observers for those actions. (Again, review Lecture 13: GUI Basics and Lecture 14: The Observer Pattern for more details on this approach.)

3 Thinking about control-flow

As discussed in class, GUI controllers are *asynchronous*: they respond to events that may arrive at any time. However, the game-play of Pawns Board is *synchronous*: like Poker Triangles or TicTacToe, the rules of the game enforce turn-based play. These two facts are in conflict with each other, so we need to design a way to reconcile them.

Think about the strategies you implemented on the previous assignment, or for TicTacToe in class: ultimately, they were simple *functions* from `State of the game -> Choice of move`. This approach works well for *synchronous* games, since another player strategy cannot move while the first one is computing its result. So conceivably, the *model* could call the player strategies directly, to get their choice of move. But this has two design problems. First, it gives the model too much controller-like power, by bypassing the views and controllers and talking to players directly. Second, and equally importantly, we want to be able to have games with any mix of human and machine players: humans take a while to choose their next move, while machines compute much more quickly. In particular, there is no obvious way to implement "a human strategy" as a *function*, since while that function is executing, the GUI is unresponsive to input — but the GUI is how the human would produce the desired answer for the strategy!

Ultimately, we need a way for the *model* to tell the players "it's your turn", and then wait for the players to choose a move (by means of their respective strategies and *controllers*). We need to separate the two steps of notifying players of their turn, and waiting for their responses.

3.1 Wait, how many controllers?

In single-player games, it makes sense to have a single controller. In single-user applications, likewise it makes sense to have a single controller. But in multi-player games, where players can act independently, it makes more sense for each player to have their own controller dedicated to that player. Think of each controller as interacting with the rest of the system on behalf of that one player.

3.2 Listening and reacting to events

Making your controllers work will require thinking carefully about features interfaces. You have already considered what events the view can publish, that the controller should listen for and respond to. These likely consisted of at least four events: player choosing a card to play, player choosing a cell to play to, player confirming their move, or player passing their turn. We needed these events because they could happen at any time, so the controller had to wait for them to occur before it could respond.

Now think about what notifications the *model* can publish, that would be of interest to its players. At minimum, the notification that "it's your turn" could come at any time, since the other player could be arbitrarily slow in choosing a move. (There may be other notifications you think the model can send out as well.) Accordingly, design a features interface for your *model*, and augment it with the ability to add listeners for those events too.

Your controller should now subscribe itself as an observer for *both* types of notifications: since it needs to coordinate between the model and the view, it needs to know when the view has actively chosen something *and* when the model notifies that the current player has changed.

A *subtle note:* If the model is responsible for notifying players when it is their turn, then you need to be very careful that the *very first* player gets notified. This requires that all the players (or the players' controllers) are registered as listeners *before* the first "it's your turn!" notification is sent. This implies that you should have a `startGame` method on your model, to ensure that there is a clear distinction between "setting everything up" and "actually playing the game".

4 Designing players

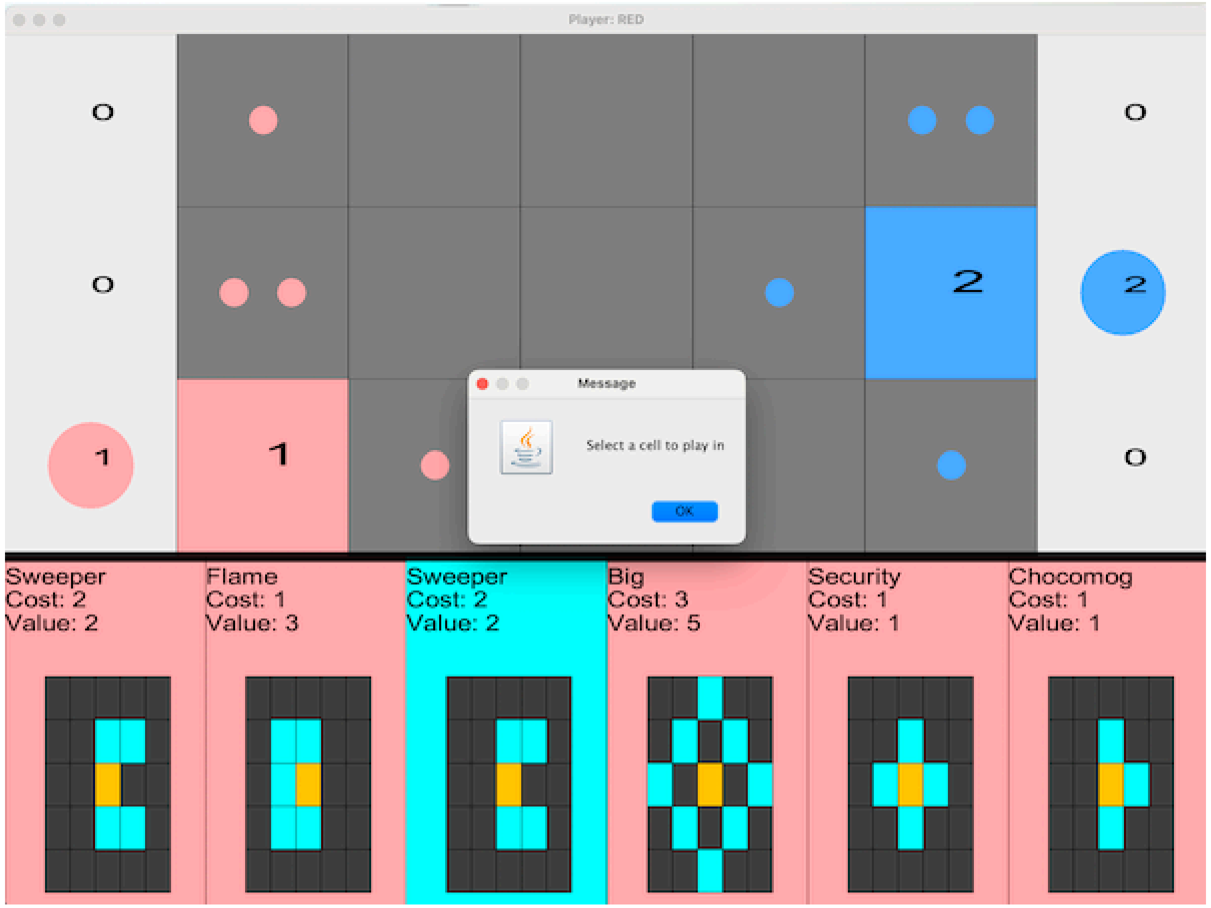
There are clearly two implementations of players needed: human players and machine players. Machine players need a strategy in order to compute their next move; human players wait for the human to interact with the view to choose their next move. By abstracting out "what can a player do?" as a player-actions interface, we have a very clean opportunity to reconcile human and machine players:

- In the design so far, your views are capable of publishing player-action events, and your controller registers itself as an observer for those events.
- A machine player can *also* publish those player-action events! If the controller registers itself as an observer for those events, then it can control machine players as well as human ones.
- This implies that the human-player implementation also must be *capable* of publishing such events...but in practice it never will, since the relevant events for that player will be published by the *view* for that player.

5 Designing the controller

- The controller implementation will need to take in a player implementation (either machine or human), and a view showing the game from that player's perspective.
- The controller will need to register itself as a listener of player actions coming from both the view and the player that it's controlling — and again, in practice *either* the view *or* the player will emit those actions, not both.
- The controller must ensure that its player can only take action when its that player's turn. This means the other player waiting should not be able to place cards on the grid or select cards at all, or even pass their turn. Accordingly, the controller must register itself as an observer for events coming from the model, so that it can be informed when the active player changes.
- The controller must prevent a player from selecting cards in their opponent's hand at all times. (Not a problem if your view only shows one player's hand.)
- The controller must ensure that the player has selected a card from their hand and a cell to place that card in *before* confirming their move. Once both have occurred and both are valid, then the player can make the move and pass over/switch over to the other player. After confirmation of a valid move, those selected choices should be forgotten.
- The controller must ensure the player can pass even if they selected a card or cell. right before passing, any card or cells selected must be forgotten by the controller.

- The controller should ensure that its view stays up to date, even if the player is a machine and doesn't interact with the view directly. Reread the discussion specifically about `toggleColor` in Lecture 13: GUI Basics, as the design concerns there might be similarly applicable here.
- Your model is likely capable of throwing exceptions when a given move is invalid. Your controller and/or your view must somehow handle these exceptions and make them visible to the user; it is not acceptable for them to silently fail and leave a warning in the console. For example, you can use the `showMessageDialog` method to display the error message to the player:



- Finally, when the game is over, the game must display who won and the winning score. This must be visible to the user much like an error message is.

You may find it helpful to set the title of the views to their respective players and whether they are playing out their turn or waiting for their turn.

6 Running your game

6.1 Showing two views

In the previous assignment, we built a placeholder `main()` method that simply created a view for the model. Now, we need to create *two* views, one for each player — and we need to create two players and two controllers, as well.

```
package cs3500.pawnsboard;

public final class PawnsBoardGame {
  public static void main(String[] args) {
    YourModel model = ...create an example model...
    YourView viewPlayer1 = new YourView(model);
    YourView viewPlayer2 = new YourView(model);
    YourPlayer player1 = new YourHumanPlayer(model);
    YourPlayer player2 = new YourHumanPlayer(model);
    YourController controller1 = new YourController(model, player1, viewPlayer1);
    YourController controller2 = new YourController(model, player2, viewPlayer2);
    model.startGame();
  }
}
```

(It is entirely possible that when your program starts, you only see one window appear. That may be an illusion because the two `JFrame`s overlap each other completely; try moving the windows around to ensure they're not covering each other, before assuming that there truly only is one window appearing...)

6.2 Configuring the game

As with Poker Triangles, we would like to be able to configure our Pawns Board game with different combinations of players and with the paths for the decks.

Your command-line here does not have to be particularly elaborate: at minimum, it should expect four string arguments

- the first is the path to the file for Red's deck
- the second is the path to the file for Blue's deck
- third and fourth describe each of the players and their strategies, for example, `"human"`, `"strategy1"`, `"strategy2"`, and `"strategy3"`.

So an example command line on the terminal could be as follows

```
java -jar pawnsboard.jar docs/red.config docs/blue.config human strategy1
```

Document whatever command-line arguments you choose. Augment the scaffold code above for `main()` to take into account the command-line and cleanly configure the decks and players accordingly.

Review instructions from Poker Triangles on configuring IntelliJ to run your program with command-line arguments.

7 What to do

1. Attempt an initial design for the player-action and model-status interfaces. (They should be fairly small interfaces, with very few methods.) Make sure you have clearly documented the purposes of those interfaces and their methods, to guide you in the next implementation stages. Implement the ability for your views to add player-action listeners, and for your model to add model-status listeners.

2. Design a controller that takes in a model, which player they are working for, and a view for that player. It should register itself as a listener for both features interfaces, and should mediate between the view and the model on behalf of that player.

3. Update your `main()` method as described above.

4. Update your README file to include explanations of all the new classes you've designed. Be sure to include a "Changes for part 3" section documenting what you changed from your initial design or from the previous assignment.

8 What to submit

- Submit all your source and test files so far
- Submit your updated README file
- Submit the four screenshots from HW6 demonstrating your view works as intended. They should be updated for HW7 if your view changed.
- Submit a JAR file (with extension `.jar`) that can run your program. (See the instructions in Assignment 6 for how to create a JAR.)

9 Grading standards

For this assignment, you will be graded on

- the design of your view and strategy interfaces, in terms of clarity, flexibility, and how plausibly they will support needed functionality;
- how well you justify any changes made to your model;
- the forward thinking in your design, in terms of its flexibility, use of abstraction, etc.;

- the correctness and style of your implementation, and;
- the comprehensiveness and correctness of your test coverage.

10 Submission

Please submit your homework to https://handins.ccs.neu.edu/ by the above deadline. Then be sure to complete your self evaluation by its due date.