Assignment 3: Freecell (Part 2)

Due Friday by 8:59pm **Points** 0

Interface Design & Representation Design

Starter files: **code**

1 Purpose

The goal of this homework is to practice writing a controller. While the model in a program that represents the game state, the controller "runs" the program, effectively facilitating it through a sequence of operations. You will follow the same "interface-implementation" method of creating the controller, and will use the model that you created in **Assignment 2** . Specifically you will write a controller that will "run" a game of Freecell, asking the user for input and printing the game state to output.

The only starter file is a type-checking file. You are expected to use your code from **Assignment 2** as the starting point for this homework. **Your code from Assignment 2 should remain in the** `cs3500.freecell.hw02` **package.**

2 The (modified) view

Assignment 2 defined a rather meaningless view interface: it had only one method (`toString`) that was mandated in every implementation anyway. A true view would not only produce the output that would be shown, but would also transmit it to a suitable destination.

A modified `FreecellView` interface has been provided to you. It contains two additional methods: `renderBoard` and `renderMessage` . The `renderBoard` method transmits the state of the freecell board to a specified destination (that depends on the implementation of this interface). The `renderMessage` method can be used to show an arbitrary message, allowing this view to show messages determined by whoever uses it.

The existing `FreecellTextView` implementation of this interface should now define a new constructor. This constructor takes an `Appendable` object that this view uses as its destination.

You must implement the `renderBoard` method in the `FreecellTextView` class so that it transmits the state of the board produced by this view to the `Appendable` object provided through its constructor. Similarly you must implement the `renderMessage` method to transmit the given message to the `Appendable` object. If no valid `Appendable` object was provided, then these methods should cause their respective outputs to be transmitted to the console. If the transmission failed, then these methods should throw an `IOException` exception to its caller.

For the purpose of maintaining backward compatability, its `toString` method should continue to work as before.

3 The Controller interface

The controller interface called `FreecellController<K>` has been provided to you. It is parameterized over the same card type as your model. This interface contains one method. Please read the documentation provided in this interface. This controller performs I/O in an implementation-specific way (see below).

4 The Controller implementation

Write a class `SimpleFreecellController` that implements the `FreecellController<K>` interface above. This implementation is not generic, but rather tied to your specific card type. You will need to:

> *To do:*
>
> - Think about which additional fields and types it needs to implement the promised functionality.
> - Write a constructor `SimpleFreecellController(FreecellModel<YourCardType> model, Readable rd, Appendable ap) throws IllegalArgumentException`. `Readable` and `Appendable` are two existing interfaces in Java that abstract input and output respectively. It should throw an `IllegalArgumentException` only if either of its arguments are null.
>
> Your controller should accept the model and these objects, create a view object that is connected to the `Appendable` and store them. In the game, whenever it needs to take user input, it should use the `Readable` object(you must determine ways to read integers and strings from a `Readable` object). Whenever it needs to transmit output, it should use the view.
>
> - The `void playGame(List<K> deck, int numCascades, int numOpens, boolean shuffle)` method should play a game from start to finish (or if the user quits). To do so, it should ask the provided model to start a game with the provided parameters, and then "run" the game in the following sequence until the game is over:
>
> 1. Transmit game state exactly as the view produces it, plus a newline (`\n`) character.
> 2. If the game is ongoing, wait for user input from the `Readable` object. A valid user input for a move is a sequence of three inputs (separated by spaces or newlines):
>
> - The source pile (e.g., `"C1"`, as a single word). The pile number begins at 1, so that it is more human-friendly.
> - The card index, again with the index beginning at 1.
> - The destination pile (e.g., `"F2"`, as a single word). The pile number is again counted from 1. **** The controller will parse these inputs and pass the information on to the model to

make the move. After a successful move, the controller will use the view to transmit the game state, plus a newline, to the output. See below for more detail.

3. If the game has been won, the method should transmit the final game state, and a message "Game over." on a new line and return.

- Key points:
  - **The deck:** The deck to be used is the one provided as input, and not necessarily the one returned from the model. Think about the possible advantages of doing this vs. summoning a deck from the provided model.
  - **Quitting:** If at any point, the input is either the letter `'q'` or the letter `'Q'`, the controller should transmit the message `"Game quit prematurely."` on a separate line through the view, and return.
  - **Invalid game parameters:** If the game parameters are invalid (e.g., invalid number of cascade/open piles) and the model throws an exception as a result, the method should transmit a message "Could not start game." and return
  - **Bad inputs:** If an input is unexpected (i.e. something other than `'q'` or `'Q'` to quit the game; a letter other than `'C'`, `'F'`, `'O'` to name a pile; anything that cannot be parsed to a valid number after the pile letter; anything that is not a number for the card index) it should ask the user to input it again. If the user entered the source pile correctly but the card index incorrectly, the controller should ask for only the card index again, not the source pile, and likewise for the destination pile. If the move was invalid as signaled by the model, the controller should transmit a message "Invalid move. Try again." plus any informative message about why the move was invalid (all on one line), and resume waiting for valid input.
  - **Error handling:** The controller's constructor should throw an `IllegalArgumentException` if either of its arguments are `null`. It should throw an `IllegalArgumentException` if a null deck is passed to `playGame()`. **The controller should not propagate any exceptions thrown by the model to its caller** .
- Write sufficient tests to be confident that your code is correct. Think carefully about what exactly you should test in this assignment.
- Consider writing a main method to run your controller interactively through the console. This is not required for the assignment, but may prove useful for your debugging. For the `Readable`, pass `new InputStreamReader(System.in)`, and for the `Appendable`, pass `System.out`. See the lecture notes for more details on writing a main method of this form.

Be sure to properly document your code with Javadoc as appropriate. Method implementations that inherit Javadoc need not provide their own unless their contract differs from the inherited

> documentation.

5 Deliverables

At a minimum, we need the following:

- The FreecellModel interface and its SimpleFreecellModel implementation
- The FreecellController interface and its SimpleFreecellController implementation
- The FreecellView interface and its FreecellTextView implementation
- Your `Card` implementation (and interface if you had one)
- any additional classes you saw fit to write
- Tests for all your implementations in one or more JUnit test classes. You should include at least all your tests from 2, and add to them...

As with **Assignment 2** , please submit a zip containing the src/ and test/ directories with *no surrounding directories* , so that the autograder recognizes your package structure.

6 Grading standards

For this assignment, you will be graded on

- Whether your interfaces specify all necessary operations with appropriate method signatures,
- whether your code implements the specifications (functional correctness),
- the clarity of your code,
- the comprehensiveness of your test coverage, and
- how well you follow the **style guide (https://google-styleguide.googlecode.com/svn/trunk/javaguide.html)** .