Assignment 3: Playing with Cards, Part 2: The Controller

Due dates:

- **Examplar submissions: Tuesday, Oct 3 at 8:59pm**
- **Implementation: Sunday, Oct 8 at 8:59pm**
- **Self-evaluation: Monday, Oct 9 at [illegible]:59pm**

Starter files: code.zip

1 Purpose

Th[illegible] writing a controller. While the model in a program represents the ga[illegible]ram, effectively facilitating it through a sequence of operations. Th[illegible]revious assignment, in that you are provided an interface that you m[illegible] class you implement will act as the controller and work with the m[illegible]signment. This controller will "run" a game of Klondike, asking for in[illegible]e the game will still be text-based and the input/output will be lin[illegible]otion of a "view" will be minimal in this assignment.

Th[illegible]signment:

- [illegible]eek, where you will submit a small set of examples designed to [illegible]roller and find several simple possible bugs or points of [illegible]
- Your *actual implementation* and full test suite
- A *self-evaluation*, due *one day plus three hours later* than the actual implementation, where we will ask you to reflect on your implementation and testing choices.

The same late-day policy as on the previous homework applies: each of these three submissions independently may use up to one late day, and you are still able to submit your self-evaluation on time even if you submit your implementation late.

You are expected to use your code from the previous assignment as the starting point for this assignment. **However, please ensure all of your new code is in the `cs3500.klondike.controller` package. Additionally, your code from the previous assignment should remain in the `cs3500.klondike.model.hw02` and `cs3500.klondike.view` packages.**

2 Preliminaries: Improving your view

In the previous assignment, you implemented a `KlondikeTextualView` class, such that its `toString` method provided the desired output. This was not ideal design, but it was convenient at the time. For this assignment, you will refine that view to be slightly more flexible. Enhance your class such that it now implements the following interface, which should replace the one given in the previous assignment:


```
public interface TextualView {
  /**
   * Renders a model in some manner (e.g. as text, or as graphics, etc.).
   * @throws IOException if the rendering fails for some reason
   */
  void render() throws IOException;
}
```


This view interface is tiny, but it abstracts away the idea that views are intrinsically `String`-based.

You should then add a second constructor to your `KlondikeTextualView` class, that takes in both a model and an `Appendable` (see below), and implement `render` such that it appends the current textual output to that `Appendable`. *You should preserve* your `toString` method, since it is useful, but you should *also* implement this `render` method — it should be very short!

3 The Controller interface

The interface for the Klondike controller must support the following functionality (as an interface `KlondikeController` that you should place in the `cs3500.klondike.controller` package):

1. A method `void playGame(KlondikeModel model, List<Card> deck, boolean shuffle, int numRows, int numDraw)`. This method should play a new game of Klondike using the provided model, using the `startGame` method on the model. It should throw an `IllegalArgumentException` if the provided model is `null`. It should throw an `IllegalStateException` **only** if the controller is unable to successfully receive input or transmit output, or if the game cannot be started. The nature of input/output will be an implementation detail (see below).

Before you implement your controller (see The Controller implementation below), you should first work through examples for it.

4 Examplar

In order for you to build examples for the controller, you should implement a class `ExamplarControllerTests` in the `cs3500.klondike` package, analogous to how you implemented one in the model package last time.

Again, your test methods should either have descriptive names or descriptive Javadoc comments explaining the intent of the scenario in your example, and what it's trying to confirm.

Hints:

- If you instantiate a `BasicKlondike` model object, it will use *our* reference implementation: there are no (known or deliberate) bugs in it. You *may* choose to use a mock implementation of the model instead, if that makes your examples easier to write (and this mock will be useful for your own tests, below).
- All the wheat and chaff implementations of the controller implement `playGame` to call the provided model's `startGame` with the arguments exactly as you provide them.
- Your challenge is to find a way to probe the rest of the behavior of the controller, despite there not being a direct way to do so. Again, take advantage of all the things you have available to configure your models and controllers, in order to "drive" the controller to test certain behavior.
- None of the chaffs need lengthy examples to find the problems. (All of our tests are short, mostly consisting of boilerplate to set things up as needed.)

As with part 1 of this project, we will reopen Examplar after the due-date, for experimentation purposes only. The experimentation-only "assignment" will only appear after the actual Examplar submissions are due, to prevent confusion in submitting to the wrong assignment.

A suggestion about workload management: You should spend a *reasonable* amount of effort trying to test your controller. If you find yourself getting stuck, switch gears and start working on the implementation, and maybe new testing ideas will occur to you as you work through that implementation. For your own learning, keep notes of which ideas occurred to you before implementing anything, vs which ideas only occured as a result of trying to implement the controller: is there a pattern of "things you only noticed later" that you might try to deliberately look for sooner, on future projects?

5 The Controller implementation

Design a class `KlondikeTextualController` that implements the `KlondikeController` interface above (also in the `cs3500.klondike.controller` package). You will need to:

1. Think about which additional fields and types it needs to implement the promised functionality.

2. Design a constructor `KlondikeTextualController(Readable rd, Appendable ap) throws IllegalArgumentException`. Recall from Lecture 8: Controllers and Mock Objects that `Readable` and `Appendable` are two existing interfaces in Java that abstract input and output respectively. The constructor should throw the `IllegalArgumentException` if and only if either of its arguments are `null`. Your controller should accept and store these objects for doing input and output. Any input coming from the user will be received via the `Readable` object, and any output sent to the user should be written to the `Appendable` object by way of a `KlondikeTextualView`. *Hint:* Look at the `Readable` and `Appendable` interfaces to see how to read from and write to them. Ultimately you must figure out a way to transmit a `String` to an `Appendable` and read suitable data from a `Readable` object. The `Scanner` class will likely be useful, as will the lecture notes on controllers.

3. The `void playGame(KlondikeModel<Card> model, List<Card> deck, boolean shuffle, int numRows, int numDraw)` method should play a game. It should "run" the game in the following sequence until the game is over. **Note:** Each *transmission* described below should end with a newline.

   a. *Transmit* game state to the `Appendable` object exactly as the view of the model provides it.

   b. *Transmit* `"Score: N"`, replacing N with the actual score.

   c. If the game is ongoing (i.e. there is more user input and the user hasn't quit yet), obtain the next user input from the `Readable` object. A user input consists of a "move" specified by a move type followed by a sequence of values (separated by any type of whitespace):

      - `mpp` followed by three natural numbers. The first is the index of the cascade pile to move cards *from*; the next is the *count* of how many cards to move; and the last is the index of the cascade pile to move cards *to*. For example, an input of `mpp 4 2 5` should cause the controller to call the `movePile` method on your model with apprrpriate inputs to move two cards (see note below).
      - `md` followed by a natural number. For example, `md 4` should cause your controller to call the `moveDraw` method on your model to move the topmost draw to the desired cascade pile.
      - `mpf` followed by two natural numbers. The first is the index of the cascade pile to move a card *from*, while the second number is the index of the foundation pile to move the card *to*. For example, `mpf 6 3` should cause your controller to call the `moveToFoundation` method with appropriate inputs.
      - `mdf` followed by a natural number. For example, `mdf 2` should cause your controller to call the `moveDrawToFoundation` method to move the topmost draw card to the desired foundation pile.
      - `dd` should cause the controller to call the `discardDraw` method on your model to discard the topmost draw card.

   d. If the game is over, the method should *transmit* the final game state one last time (which will either be the message `"You win!"` or the message `"Game over. Score: N"`) The method should then end.

Key points:

- **User input numbering:** To make the inputs more user-friendly, all indices in the user provided input begin from 1. This will affect the inputs that your controller passes along to your model.
- **Quitting:** If at *any* point, the next value is either the letter `'q'` or the letter `'Q'`, the controller should *transmit* the following in order: the message "Game quit!", the message "State of game when quit:", the current game state, and the message "Score: N" with N replaced by the final score. The method should then end. For example:

  ```
  Game quit!
  State of game when quit:
  Draw: 3♢, 2♢, 8♡
  Foundation: <none>, <none>, <none>, <none>
   2♠  ?  ?  ?  ?  ?  ?
      6♣  ?  ?  ?  ?  ?
         A♢  ?  ?  ?  ?
            K♣  ?  ?  ?
               7♠  ?  ?
                  7♣  ?
                     5♡
  Score: 0
  ```

- **Bad inputs:** If any individual value is unexpected (i.e. something other than `'q'`, `'Q'` or a number) it should ask the user to re-enter that value again. For example, if the user is trying to move a card from one cascade pile to another, and has entered the source pile number correctly, but entered the number of cards to move incorrectly, the controller should continue attempting to read a value for the number of cards to move before moving on to read the value for the destination pile. You should behave similarly for the other commands. Once all the numbers are successfully read, if the model indicates the move is invalid, the controller should *transmit* a message to the `Appendable` object saying `"Invalid move. Play again. X"` where X is any informative message about why the move was invalid (all on one line), and resume waiting for valid input. *Hint:* You should probably design a helper method to retry reading inputs until you get a number or a `'q'`/`'Q'`. Using that helper consistently will make it much easier to implement the desired retrying behavior described here. That helper probably should *not* be responsible for determining if a number is a valid coordinate — that's the model's job — but that helper does need to return either the user's number *or* their desired to quit the game. Think carefully about the signature of this method before you start implementing it...
- **Error handling:** The `playGame` method should throw an `IllegalArgumentException` if a `null` model is passed to it. If the `Appendable` object is unable to transmit output or the `Readable` object is unable to provide inputs (for whatever reason), the `playGame` method should throw an `IllegalStateException` to its caller. The `playGame` method **must not** throw any other exceptions, nor should it propagate *any* exceptions thrown by the model.
- **Write sufficient tests** to be confident that your code is correct. **Note:** once the model has been tested thoroughly (which you hopefully did in the previous assignment), all that remains to be tested is whether the controller works correctly in all cases. Lecture 8: Controllers and Mock Objects will prove to be helpful.

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.

If you had to change your implementation from the previous assignment, please document your changes in a README file (a plain text file) that explains what you changed and why. This doesn't have to be long; a simple bullet-point list will suffice. But having this documentation will make your TAs' grading job a lot easier!

6 Testing

After you've submitted your Examplar examples, you will need to add additional tests to assess whether your controller works *regardless of whether your model works*. (Again, if you've sufficiently tested your model in the previous assignment, then you can rely on your model working here.) You will likely need to use the techniques in Lecture 8: Controllers and Mock Objects.

You should create your primary test class in the `cs3500.klondike` package, alongside the `ExamplarControllerTests` class. This test is outside your controller package, and so can only test the public-facing behaviors of your controller. If you want to test internal implementation details as well, you should create one more test class in the `cs.klondike.controller` package, so that you can check package-private implementation details if needed.

Be mindful of which test cases you place in which test class! *Technically*, you could run all the tests from a single class. But using multiple classes like this helps convey to the reader of your code some of your thought processes behind each test: the reader should understand the examples first, then look at the tests of public behavior, and finally look at implementation-specific fiddly details.

Note: When you submit your full implementation, you will see automated tests that *we* wrote and run against *your* code. We give some of our test methods mnemonic names, so that you can try to deduce what our tests are checking for. Just because *we* have a test for a given scenario, though, does not mean that you shouldn't write your own test case to confirm your understanding!

7 What to submit

- For Examplar: submit a properly-structured zip containing
  - *only* your `ExamplarControllerTests.java` file
  - and any stub model implementations you added, if appropriate.
- For your impelmentation: submit a properly-structured zip containing at minimum
  - The model interface (`KlondikeModel.java`)
  - Your implementation of the model (`BasicKlondike.java`)
  - The view interface (`TextualView.java`)
  - Your implementation of the view (`KlondikeTextualView.java`)
  - The controller interface (`KlondikeController.java`)
  - Your implementation of the controller (`KlondikeTextualController.java`)
  - Any additional classes necessary to compile your code
  - All your tests (including Examplar) for all your implementations in one or more JUnit test classes. You should include at least all your tests from the previou assignment, and add to them...
  - A brief README file (a plain text file) explaining what changes from the previous assignment you made, and why.

As with the previous assignment, please submit a zip containing *only* the `src/` and `test/` directories with *no surrounding directories*, so that the autograder recognizes your package structure. **Please do not include your `output/` or `.idea/` directories — they're not useful!**

8 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,
- How well your code follows the design principles discussed so far,
- The comprehensiveness of your test coverage, and
- How well you follow the style guide.

Please submit your homework by the above deadline. Then be sure to complete your self evaluation by the second deadline.