Assignment 4: Playing with Cards, Part 3: Changing the Game

Due dates:

- **Examplar submissions:** [illegible]9pm
- **Implementation: Wedne** [illegible]pm
- **Self-evaluation: Thursd** [illegible]n

Starter files: code.zip

Note: Homeworks 5 through 8 will begin a new project, and you will be working with a partner for them. Start thinking about who you'd like to partner with. You will sign up with your partner on the handin server, and you will not be able to submit subsequent assignments until you are part of a team on the handin server. If you do not know (or remember) how to request teams, follow these instructions. Please request teams no later than the start of homework 5, which is Oct 20 — if you have not requested a teammate by then, we will randomly assign you one. You only need to request a partner for Assignment 5; we will copy the teams to the remaining assignments.

1 Purpose

The benefits of the model-view-controller architecture shine when we need to add new features, by isolating relevant parts of our design and changing them independently. In this assignment we will see those benefits pay off by supporting other forms of Klondike. The goal of this assignment is to give you a chance to critically examine your earlier design choices, and either leverage or revise them to enable adding variations of this game with minimal change and duplication of code.

With one exception (main), all *new* classes and interfaces for this homework should be in the `cs3500.klondike.model.hw04` package. All classes written in previous assignments, even if improved upon, should remain in their respective packages.

There will be three submissions for this assignment:

- An *Examplar* submission, early in the week, where you will submit a small set of examples designed to probe *our* implementations of the controller and find several simple possible bugs or points of confusion.
- Your *actual implementation* and full test suite
- A *self-evaluation*, due *one day plus one hour 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.model.hw04` package. Additionally, your code from the previous assignment should remain in the `cs3500.klondike.model.hw02`, `cs3500.klondike.view` and `cs3500.klondike.controller` packages.**

2 Klondike with more challenging rules

The rules of Klondike, as pr[illegible] fairly straightforward. Cards can be moved between cascade piles, from piles to [illegible]he draw pile to cascade piles or to foundation piles, and can be discarded from the [illegible]y, until the pile gets recycled and the card comes around again.

This ability for cards in the draw pile to be recycled as often as needed makes the game substantially easier. Therefore in this assignment, you will design a variant of the game that allows only *limited draw attempts*: the draw pile can be recycled only a finite number of times (specified when the game is constructed). For example, suppose the draw pile contains cards A, B and C, and the game is constructed to allow only two draws (i.e. allow redrawing each card only once), and suppose the game is started with a draw pile size of 3. Then repeatedly discarding draw cards will produce the following:

Draw state	Player action
`Draw: A B C`	Player discards A.
`Draw: B C A`	Player discards B.
`Draw: C A B`	Player discards C.
`Draw: A B C`	Player discards A.
`Draw: B C`	Player discards B.
`Draw: C`	Player discards C.
`Draw:`	

Each card has a chance to be the leftmost draw card exactly twice (as constructed in this example), after which it is permanently discarded.

3 Klondike with other board rules

Another variant of the game is called *Whitehead Klondike*. In this game, four rules are changed:

- All of the cards in the cascade piles are dealt face-up.
- Instead of *alternating* colors, builds must be *single-colored* – red cards on red cards, black cards on black cards.
- When moving multiple cards from one cascade pile to another, all the moved cards must be *all the same suit*, not merely a valid build.
- When moving a card into an empty cascade pile, it can be any card value, not just a king. (When moving multiple cards into an cascade pile, it must be a single-suit run, as above, but it can start from any value.)

Everything else about the game stays unchanged: draw cards are recycled indefinitely, foundation piles each start at Ace, stay in a single suit, and count up to the highest card, scoring rules stay the same, etc.

4 Examplar

As with previous parts of this project, there will be an Examplar submission early in the week for you to develop your understanding of the project requirements. You will develop your examples in the class `cs3500.klondike.ExamplarExtendedModelTests`. That class will need to include test methods aimed at distinguishing Limited-Draw Klondike wheats from chaffs, and test methods aimed at distinguishing Whitehead Klondike wheats [illegible] o not need to distinguish *both* variants within a single test method.

Hint: We give the same gua[illegible]his assignment as on part 1 of the project. All the observation methods work [illegible]e chaffs, as does `startGame`, and you should focus your attention on the mutator m[illegible]

5 Assignment Requirements and Design Constraints

1. Design classes implementing the Limited-Draw and Whitehead variants of Klondike. Both classes should clearly implement `KlondikeModel`, and clearly share some commonalities with the existing `BasicKlondike`. In your implementation, strive to avoid as much code-duplication as possible among the three models, while making sure that all three fully work properly. If done correctly, none of your code from before should break or be affected. You may need to refactor your earlier code, though, to make it more flexible and enable better code reuse for these new classes.

2. Design a *factory* class, named `KlondikeCreator`, as described below.

3. Implement a `main` method to allow you to choose different game variants from the command line, when running your program. (This is described below.)

4. If you had to change any part of your design from prior assignments, document those changes in a README file. (This must be a plain-text file.)

5. Test everything thoroughly: make sure the new models work properly, and that the controller can control them as well as it could the original model. You do not need to test your `main` method.

You must complete these requirements while respecting the following constraints:

- You are not allowed to change the interface of the model (`KlondikeModel`) at all from before.
- You are not allowed to change the controller interface (`KlondikeController`) at all from before.
- As before, you are not allowed to add any additional public methods in your classes, with the exception of any expected constructors.
- You must create separate model implementations, without eliminating `BasicKlondike` from before. That is, models that represent all variations of the game must co-exist.

In this assignment it is important not only to have a correctly working model, but also a design that uses interfaces and classes appropriately. Make sure you minimize replication of code. You *may* refactor your earlier designs to do this. You may also have to change earlier implementations to remove bugs. This is OK, but *must* be properly documented and justified.

6 Hints

- You should always plan before you code. But this assignment is **designed** to be significantly more difficult if you code prematurely and try to figure out things as you code!
- For each variant, go through each operation of the model interface, and think about how that operation works for this variant. Planning out every operation before coding will save you a lot of time!
- Recall that your view needs to be able to work with any model implementation, without knowing which one!
- You may be tempted to discover all possible abstractions beforehand. Make sure you minimize code repetition, but not over-abstract so much that some methods lose meaning. Even private helper methods should have a specific purpose! While it is good to anticipate abstraction, there is nothing wrong with thinking about abstraction after implementing all the code. It's the end result that counts!

7 The `KlondikeCreator` class

Design a class with the above name. The class should define a `public enum GameType` with three possible values: `BASIC`, `LIMITED` and `WHITEHEAD`. It should offer a `static` factory method `create(GameType type)` that returns an instance of (an appropriate subclass of) `KlondikeModel`, depending on the value of the parameter. If this is `LIMITED` redraw game, then the number of redraw attempts should be 2 (i.e. each card can be drawn three times before being permanently discarded).

8 The `main()` method

Add the following class to your project:

```
package cs3500.klondike;

public final class Klondike {
  public static void main(String[] args) {
    // FILL IN HERE
  }
}
```

This `main()` method will be the entry point for your program. Your program needs to take inputs as command-line arguments (available in your program through the argument `args` above). Review the documentation for command-line arguments in a Java program.

Specifically:

- The first command-line argument must be a string, specifically one of `basic`, `limited`, or `whitehead`. This argument will decide which game variant (and hence which model) you should use. If the argument is `limited`, it must be followed by an integer `R` indicating the number of times the draw pile can be used.
- You may optionally pass one or two more arguments `P` and `D`, both of which should be parsed as integers, where P specifies the number of cascade piles, and D specifies the number of draw cards. If unspecified, you should use default values for these parameters.

The following are some example command lines, and their meanings:

- `basic` produces a basic game with default number of cascade piles and visible draw cards
- `basic 7 3` achieves the same thing, but explicitly sets the size of the game.
- `limited 3 6 2` produces a game of limited-draw solitaire, whose draw pile can be used 3 times, with six cascade piles and 2 visible draw cards.
- `whitehead 8` produces a Whitehead Klondike game with 8 cascade piles and the default number of visible draw cards
- `whitehead 7 8` produces a Whitehead Klondike game with 7 cascade piles and 8 visible draw cards

You may add additional methods to your `KlondikeCreator` class, but you must maintain the `create` methods specified above, for our tests to compile against your code.

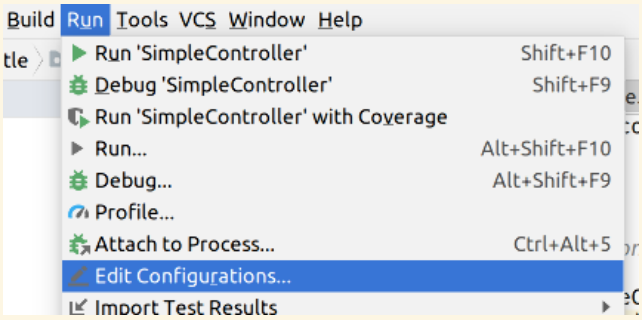
This is *not* an exhaustive list; other command lines are possible.

When you specify command-line arguments, they are always treated as strings, even if they are not within quotes. However, quotes are necessary if you want to pass a string that contains spaces in it.

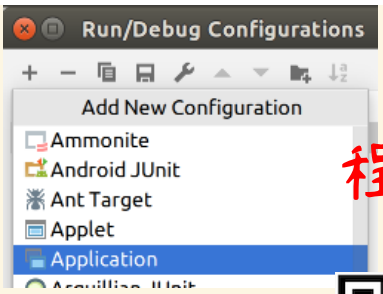
These arguments will appear in the `String[] args` parameter to your `main` method; you can use them however you need to, to configure your models. For this assignment, you do *not* need to explicitly handle invalid command lines (e.g. by producing an informative error message). However, your code should not *crash* (e.g. by specifying -1 as the number of rows, and causing an `IndexOutOfBounds` exception).

8.1 To actually run your program with command line arguments in IntelliJ IDEA:

- Go to `Run > Edit configurations`

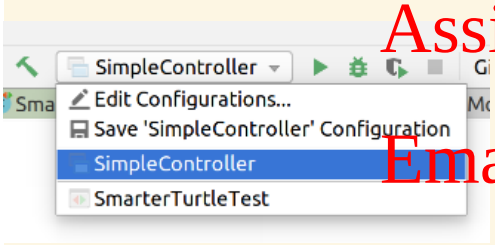


- Click the + button in the upper left, and select `Application` from the dropdown that appears.



- Give the new configurat…ll remember (e.g. "Basic 7/3").
- In the `Main class` textb…`ondike.Klondike` – the fully-qualified name of the class with your `main` method.
- In the `Program argumen…`e command line arguments you want to use for this configuration, e.g. `basic 7 3`
- Leave everything else at their defaults, and click Ok.

You can repeat this process as many times as you want, to make as many run configurations as you need. Then to choose among them, use the dropdown menu next to the run icon in the toolbar:



and press Run.

9 What to submit

- For Examplar: submit a properly-structured zip containing
  - *only* your `ExamplarExtendedModelTests.java` file
- For your implementation: submit a properly-structured zip containing
  - All your code from before (with as few changes to the classes as possible, and any changes fully explained in comments)
  - Your implementations of Limited-Draw Klondike and Whitehead Klondike, and any support classes needed
  - Your `Klondike` class with the `main()` method.
  - Tests for all models in one or more JUnit test classes. It is a good idea to include your earlier tests as well, for regression testing. We certainly will...
  - Your README file documenting your changes.

Your main class should be in the `cs3500.klondike` package, as should your `ExamplarExtendedModelTests` class, while all other new classes and interfaces for this homework should be in the `cs3500.klondike.model.hw04` package. All classes written in previous assignments, even if improved upon, should remain in their respective packages.

As with Assignment 3, 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!**

10 Grading standards

For this assignment, you wi

- Whether you had to mo written interfaces,
- whether your code impl ations (functional correctness),
- how well your code is st uplication, improve readability, etc.
- the clarity of your code,
- the comprehensiveness of your test coverage, and
- how well you follow the style guide.

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