Northeastern University
CS5500 – Managing Software Development
Fall 2018 Professors Jump, Katz, and Weintraub

Homework 2: Creational Design Patterns

The purpose of this homework is to give you experience with creational design patterns.

Your work will be assessed for:

1. Whether it works for the standard test and your tests.
2. Whether it works for our tests.
3. Design efficiency, efficacy, and robustness.
4. Code style (documentation and code style). You should create files in such a way so that someone else can use, test, or extend your files without you being present.

In all work, you should run `sonarlint` and eliminate all major issues. Instructions on installing `sonarlint` are included at the end of the assignment.

REQUIREMENT: You should build your project using Maven so we can use Maven as part of grading your project. Maven will create the directory structures for your project. Maven comes built into Eclipse (Photon version – the latest), Netbeans, and Intellij. If you are unfamiliar with Maven, there's a nice introduction on youTube: (`https://youtu.be/sNEcpw8LPpo`, or a more comprehensive lesson at lynda.com (Java: Build Automation with Maven with Peggy Fisher). Or, you can try the Apache tutorials (e.g. `https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html` or `https://maven.apache.org/guides/getting-started/index.html`).

Turn in your answers (packages) by pushing them to your course repository into the `homework 2` directory.

REQUIREMENT: You must work by yourself; your work must be original which means you may not use any internet resources other than Java language/library documents, junit documents, and the like. All violations of these rules are academic misconduct.

1 Playing Cards

Games based on decks of cards are popular. The "base" card deck is typically 52 cards built from four categories called suits. In the USA at least, the suits are clubs, diamonds, hearts, and spades and are usually presented by symbols: ♣, ♦, ♥, and, ♠, for clubs, diamonds, hearts, and spades respectively.[1] Suits may have colors, either red or black. Each suit contains cards, and each card within a suit has a unique rank, typically the numbers two through ten and face cards: *jack, queen, king and ace.*[2] Thus, the card 2♣ is a *club* whose rank is two. Often, a card has a name, which is simply its rank and suit, *e.g.* 2♣, as a convenience. In most games, the rank is used to order cards. For example, a three is better than a two, a ten is better than a five; but this depends on the game. Face cards are a confusing bunch. Sometimes they simply extend the ordered set (as in {2, 3..., 10, jack, queen, king, ace}). Sometimes they are simply worth a constant amount (e.g. 10). Sometimes there are just odd rules such as aces may sometimes be worth one or 11 (blackjack players will be familiar with this). Except for magic tricks, cards are immutable.

A deck of cards is an ordered set of cards. A standard deck is built from the cards {2, 3..., 10, jack, queen, king, ace} from each suit (clubs, diamonds, hearts, and spades), which means a standard deck has 52 cards. However, there are other types of decks as well:

1. A Euchre deck, which is composed of 9, 10, J, Q, K, and A of the four suits – 24 cards in total.
2. A pinochle deck, which is composed 9, 10, J, Q, K, and A of the four suits, but there are two copies of each suit. There are 48 cards in total.
3. A Vegas blackjack deck. It's composed of two or more standard decks. Typically it's six to eight decks.

Decks may be shuffled, meaning the cards within the deck are randomly ordered, or decks may be sorted. Sometimes, a deck may be cut, meaning a point in the deck is selected and first half of the deck {$\text{card}_1 \ldots \text{card}_{\text{selected-1}}$} is placed at the end of the deck in order. That is, the first card in the deck is now $\text{card}_{\text{selected}}$; the first card after the original end card is now card_1. Card_2 follows card_1 and so forth up to $\text{card}_{\text{selected-1}}$.

Most people think of a hand as something a player holds, and a game has one or more players. Our way of thinking about a hand is a game has one or more hands.[3] A hand contains some number of cards, usually fixed by the game. For example in five card poker, a hand has five cards. In bridge, the deck is split evenly among four players; each player's hand has 13 cards; and the same goes for hearts. The cards within the hand may be ordered, and in some cases, shuffled. Sometimes cards are pulled out of a hand, such as in hearts when players pass cards or when you play a card in bridge or euchre. Sometimes cards are added to a hand, such when you receive cards from another player in hearts, win cards off a pile in war, or when you replace cards in five card draw in poker. You can also test if a card is in a hand, like when you ask for aces in *Go Fish.*

A game involves a deck and some number of hands. Cards are usually dealt, which means cards end up in hands. After that, the game proceeds according to whatever rules apply. For example, War, perhaps the simplest card game after the little kids' favorite *52 card pickup*, usually is a two player game that uses a standard deck where the all the cards are dealt to two hands. Bridge and hearts, as already mentioned, use a standard deck and all the cards are dealt to four hands. Blackjack, casino style, usually involves multiple decks, and each hand (player) gets two cards. Poker has hands typically of five or seven cards. But, sometimes each hand gets all n cards; sometimes some of the cards are shared. You get the picture.

[1] See `https://en.wikipedia.org/wiki/Playing_cards_in_Unicode` for the unicode values for the suits.
[2] Let's not quibble if an ace is a face card. For this assignment, we'll consider it as one.
[3] We're not going to worry about players for the moment. Maybe in the next assignment.

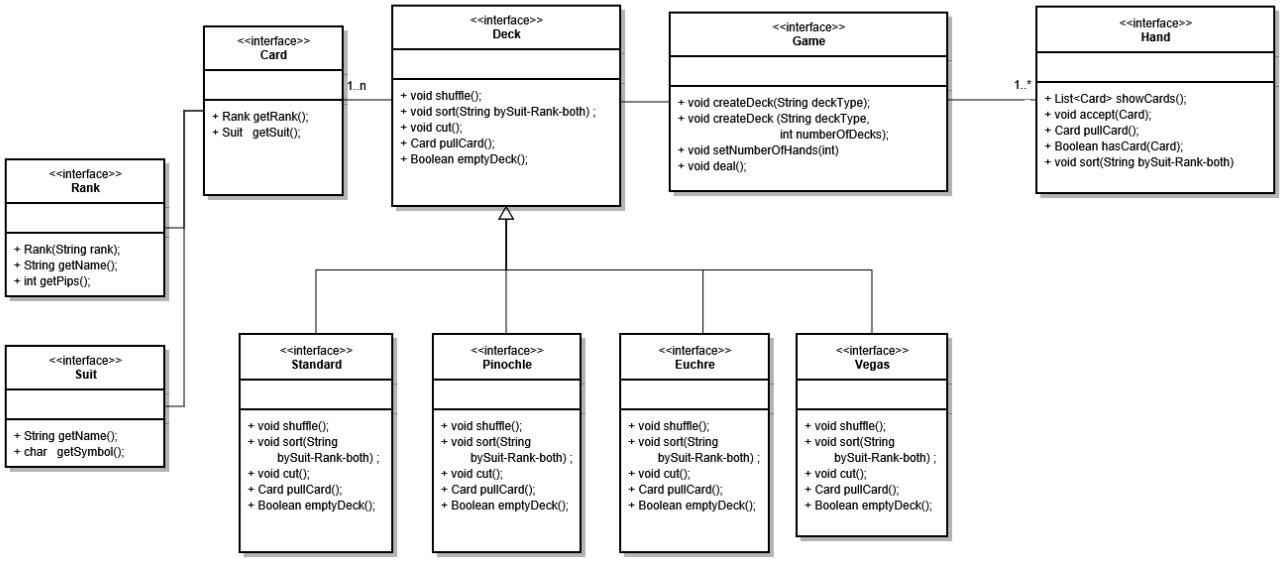


Figure 1: UML for Playing Cards

Figure 1 depicts one way to model this problem.

OK, enough with the introductions. Let's get busy.

Task: Implement the Class Diagram

Your task is to design and implement a set of classes that implement the interfaces defined in Figure 1. You must test it thoroughly. For example, if there are four suits, is one test to see if the card reports the right suit (meaning just one suit is reported) good enough?[4] Make sure that all your code is properly documented and well styled. The following JUnit test should be included in your test suite. All tests must pass. We will run our own test suite as well, and your code will be expected to pass those tests as well.

At a minimum, you must test:

1. The creation of individual cards (meaning test that a card has the right suit and rank)
2. A deck has the right cards in it and the right number of cards.
3. A deck after being shuffled has a different order than before.
4. A deck after being sorted by rank, suit, or both rank and suit, is in the expected order.
5. A deck after being cut is in the right order.
6. A deck after a card is pulled has one less card than before that card was pulled (and that card is no longer in the deck). Beware of the pinochle deck for this test case!
7. An empty deck is indeed empty. A deck with cards in it is indeed not empty.
8. A hand is properly constructed.

[4] Obviously since we're making a deal of this, the answer is no.

9. A hand properly accepts a card.
10. When a card is pulled from the hand, the hand now has one less card and that card is no longer in the deck. *Op cit*, beware the pinochle deck...
11. The hand tells accurately whether a particular card, rank, or suit is present.
12. The hand properly sorts the card by suit, rank, or both suit and rank.
13. The hand properly shuffles the cards in it.
14. A similar list pertains to the game as well.

Now, you may think this is a lot of tests. Well, it is. Writing good code involves writing a judicious set of tests. Your mantra should be, if there's a feature, it has to be tested.

Important: please ensure that all your code is properly documented. A nice list of do's and don'ts for commenting is at `https://javarevisited.blogspot.com/2011/08/code-comments-java-best-practices.html`.

You should use javadoc to organize and style your documentation. See how Oracle does it at `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`.

For this course, the key tags are:

- @author
- @param
- @return
- @throws
- @version (optional - more important later on)

Figure 2 gives an example of a javadoc'ed class. We used the `JAutodoc` plug-in in eclipse to create the framework text. You can set up text templates for the repeated entries like licensing and the like. JAutodoc has a nice feature by marking `// TO-DO Auto-generated comments` in your file, which means that your environment will warn you if you forget to take care of this task.

You will be using `Maven` to build your system. To facilitate grading, when you create the project, please use the following conventions:

1. Under groupID, use `edu.northeastern.ccs.cs5500`.
2. For artifact id, use `homework2a`.

When you are done, please push your project to your personal class repo. All work must be checked in by the due date. We will pull your code from the repo using the last check-in before the due date.

```java
/*
 * This is where you put any licensing, copyright, or disclaimer notes.
 *
 * The Foo class is a simple example to illustrate documentation
 * comments.
 *
 * @author Mike Weintraub
 */
public class MadeUp{

    /**
    * Keeps count of the number of good answers provided in a class.
    *
    * @param numberOfGoodAnswers the amount the value should be incremented by
    * @return the new count
    */
   int increment(int numberOfGoodAnswers){
       ...
   }

   /**
    * An internal counter to keep track of good answers.
    */
    private int count;
    ...
}
```

Figure 2: Simple code snippet with javadoc

2 Flexible Playing Cards

So, you've implemented the interfaces and are ready to start playing some games. But, you may have realized that creating games is a bit messy if sometimes we want to create the same game with different kinds of decks or different kinds of hands.

Your next assignment is to create a new version of the application you created in Part 1 (Playing Cards). Your task is to create a version where all games are created so that it is easy to create specialized types of games (decks or hands) and to perform additional operations when either is being created. Your solution should make it easy to vary between different specialized creation options. For example, you should be able to build a five-card hand from a euchre deck, a pinochle deck, or a standard deck easily and directly.

Please test your code thoroughly by updating your tests from the previous question to reflect the new design and adding additional tests as needed.

Because you are using `Maven`, when you create the project please use the following conventions:

1. Under groupID, use `edu.northeastern.ccs.cs5500`.
2. For artifact id, use `homework2b`.

Appendix - How to Install Sonarlint on Eclipse and Intellij

`sonarlint` helps you in finding bugs and quality issues as you code. Simply open a file, start coding, and you will start seeing issues detected by `sonarlint` . An example is shown below.

Prerequisite

1. You should have Java 8 (JRE) installed.

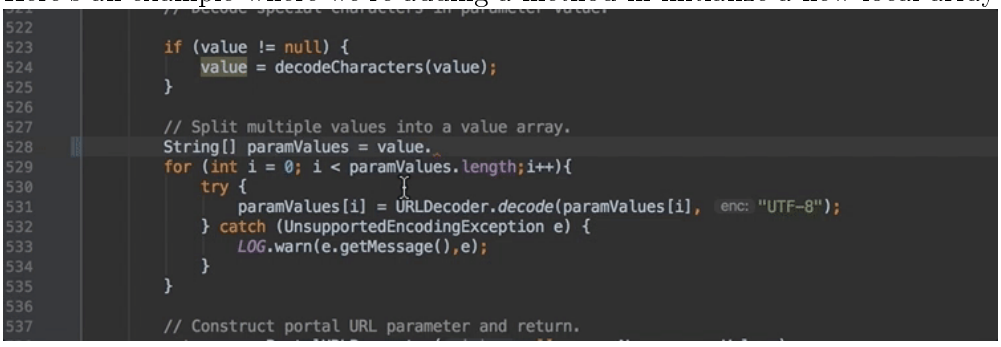
Eclipse

1. Go to Eclipse MarketPlace under Help Menu.
2. Search for 'sonarlint'.
3. In search results, you should see 'SonarLint 3.x', click on install.
4. Once installed, restart eclipse and `sonarlint` should be enabled by default.
5. For more details, `https://www.sonarlint.org/eclipse/howto.html`

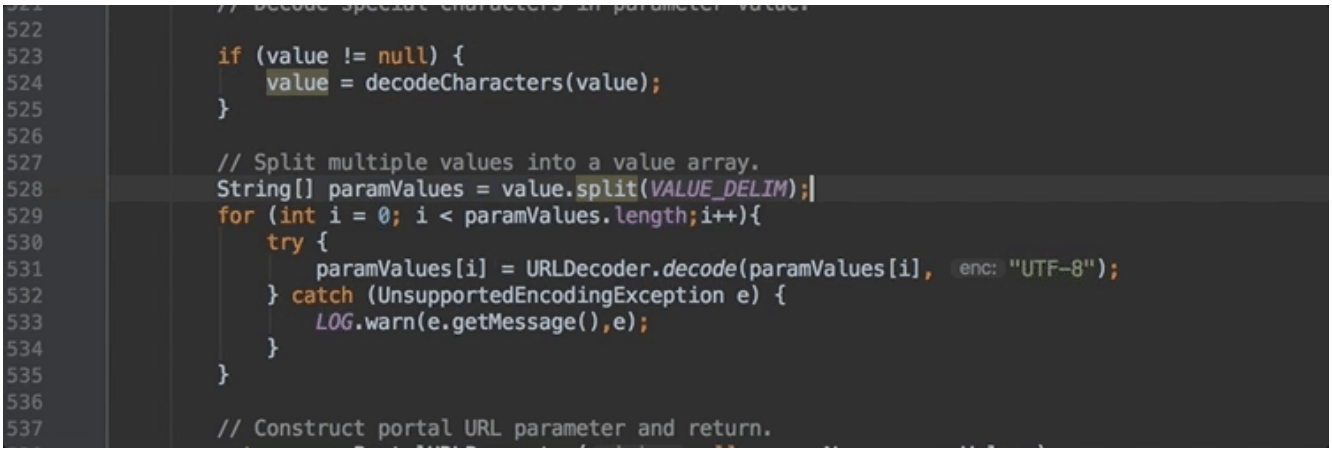
IntelliJ

1. Go to Settings under File (Windows) or Preferences (Mac) on main menu.
2. Then, go to plugins.
3. Search for 'sonarlint'.
4. In search results, you should see 'SonarLint', click on Download and Install.
5. After installation, click OK in setting dialog and restart IntelliJ.
6. `sonarlint` should be enabled by default after restart.
7. For more details, `https://www.sonarlint.org/intellij/howto.html`

Here's an example where we're adding a method in initialize a new local array at line 528.

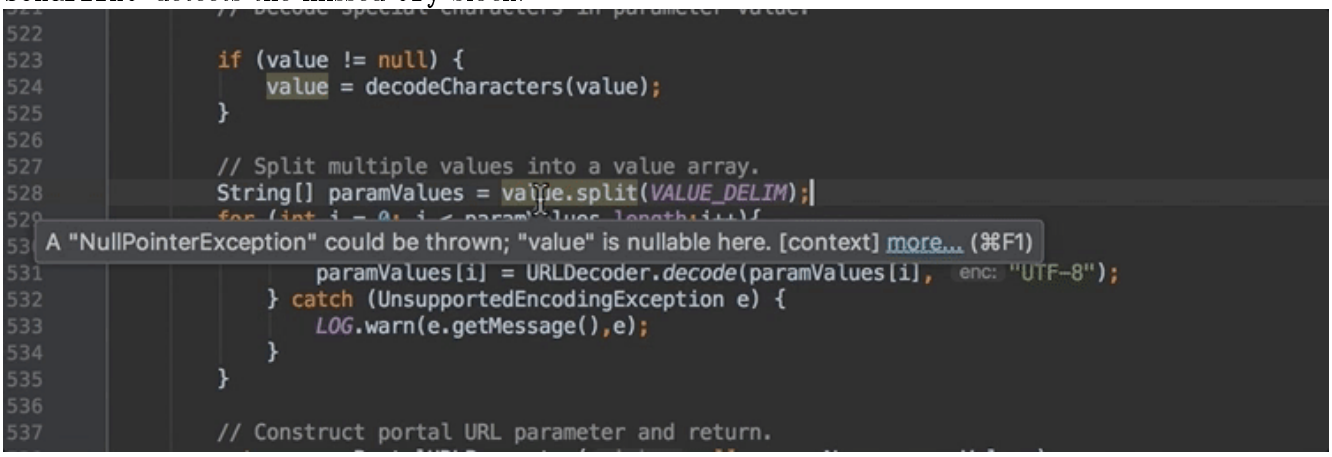


typing. . . and eventually finishing.



```
522
523         if (value != null) {
524             value = decodeCharacters(value);
525         }
526
527         // Split multiple values into a value array.
528         String[] paramValues = value.split(VALUE_DELIM);
529         for (int i = 0; i < paramValues.length;i++){
530             try {
531                 paramValues[i] = URLDecoder.decode(paramValues[i],  enc: "UTF-8");
532             } catch (UnsupportedEncodingException e) {
533                 LOG.warn(e.getMessage(),e);
534             }
535         }
536
537         // Construct portal URL parameter and return.
```

`sonarlint` detects the missed `try` block.



```
522
523         if (value != null) {
524             value = decodeCharacters(value);
525         }
526
527         // Split multiple values into a value array.
528         String[] paramValues = value.split(VALUE_DELIM);
529         for (int i = 0; i < paramValues.length;i++){
530             try {
531                 paramValues[i] = URLDecoder.decode(paramValues[i],  enc: "UTF-8");
532             } catch (UnsupportedEncodingException e) {
533                 LOG.warn(e.getMessage(),e);
534             }
535         }
536
537         // Construct portal URL parameter and return.
```

A "NullPointerException" could be thrown; "value" is nullable here. [context] more... (⌘F1)