MIT
6.031: Software Construction
Prof. Rob Miller and Max Goldman

6.031 Spring 2020 Quiz 2

Your Kerberos username: _____

You have 50 minutes to complete this quiz. There are 14 problems. The quiz is closed-book and closed-notes, but you are allowed one two-sided page of notes.

This page automatically saves your answers as you work. Saved answers are marked with a green check. If you see a red exclamation mark, or a red notification that you are disconnected, your answers are not being saved: try reloading the page right away, before continuing to work on the quiz.

If you want to ask a clarification question, please put yourself on the online help queue and a staff member will talk to you in a text chat.

Good luck!

Before you begin, please sign this honor statement.

I affirm that this is a closed-book quiz, which means:

- I will not reference any materials aside from my 1-page 2-sided cribsheet (including my class notes, Eclipse, the course website, any files stored on my laptop, and any resources on the internet except this quiz itself and the online help queue).
- I will not communicate with classmates or anyone else (other than 6.031 staff members) about anything related to this quiz until the solutions are officially released.

By entering your full name below (first name and last name), you agree to this honor statement.

The code for this quiz has two ADTs.

`Card` represents an immutable standard playing card. A card has a *rank* (2-10, jack, queen,

king, ace) and a *suit* (clubs, diamonds, hearts, or spades).

`Deck` represents a mutable ordered set of at most 52 different playing cards.

The code is provided at the bottom of this page, and you can open all the code in a separate tab.

---

1. (9 points) Write a threadsafe rep for `Deck`, and write the `Deck()` constructor. Don't implement any other operations for `Deck`. You may use the `Card.ALL_52_CARDS` constant.

```
public class Deck {
```

```
    private final List<Card> = Collections.synchronizedList(new ArrayList
    <>(Card.ALL_52_CARDS));

    public Deck() {
    }
```

```
}
```

---

2. (8 points) Write the abstraction function for your `Deck` rep:

> AF(cards) = the deck of playing cards consisting of the cards in `cards` in the same order, where cards.get(0) is the top of the deck and cards.get(cards.size()-1) is the bottom of the deck.

---

3. (8 points) Write the rep invariant for your `Deck` rep:

> cards.size() <= 52
> all elements of cards are pairwise distinct

---

4. (8 points) Observe that the return type of `Deck.draw()` is `Optional<Card>`, an abstract datatype that either has a reference to a Card, or has no value.

Suppose you are going to implement the generic `Optional<E>` yourself. Write a datatype definition for `Optional<E>`. Your datatype definition should have two variants.

> `Optional<E> = Empty + Present(e:E)`

5. (9 points) Write the Java interface and class definitions for your datatype definition. Include the reps, but **no methods or constructors**. Your answer should consist of Java code from which all methods and constructors (and their signatures and bodies and associated comments) have been deleted. Don't write more than the 10 lines this box allows.

```
interface Optional<E> { }

class Empty<E> implements Optional<E> { }

class Present<E> implements Optional<E> {
    private final E e;
}
```

6. (9 points) `Optional<E>` has an operation `orElse`, such that (for example) `deck.draw().orElse(aceOfSpades)` returns the top card on the deck if the deck is nonempty and `aceOfSpades` if the deck is empty.

Define `orElse` as a function using mathematical notation, with one case for each variant of your datatype definition. Your answer should be two lines long, and should not be written in Java code. (An example of a function written in mathematical notation is f(x) = x+1.)

> orElse(Empty, defaultValue) = defaultValue)
> orElse(Present(e), defaultValue) = e

7. (8 points) A playing card can be described by a 2- or 3-character string such as "10C" for the ten of clubs, or "JH" for the jack of hearts. Write a grammar for these string representations of playing cards. Your grammar should have three nonterminals: `card`, `suit`, and `rank`.

> card := rank suit;
> rank ::= [AJQK2-9] | '10';
> suit ::= [CDHS];

8. (9 points) Suppose `Deck` has a `filter` operation that keeps cards matching a predicate, and discards cards that don't. Write a mathematical type signature for a `filter` **producer** operation of `Deck`. (An example of a mathematical type signature is `factorial` : `int` → `int`.)

> filter: Deck x (Card -> Boolean) -> Deck

---

9. (8 points) Write a Java instance method signature for `filter` as a **mutator** operation of `Deck`. (An example of a Java method signature is `public static int factorial(int n)`. Java has an interface `Function<T,U>` to represent a one-argument function.)

> public void filter(Function predicate);

---

10. (8 points) Assume there are two threads, T1 and T2, running the `run` method below and sharing the same `Deck` object. Fill in `run` with at most 3 lines of Java, using the `deck` variable, that would have a race condition when run by both T1 and T2 concurrently.

```
Deck deck = new Deck(); // this deck is shared by the threads
Card aceOfSpades = ...;  // these are the cards corresponding to their va
riable names
Card tenOfClubs = ...;
...
public void run() {
```

> ```
> deck.move(aceOfSpades, 0);
> deck.draw();
> // other answers are possible
> ```

`}`

---

11. (8 points) Explain your race condition from the previous question. Don't use more space than the box allows.

> The behavior of this code depends on which thread does the draw() first. If T1 draws first, then it will see the ace of spades, and T2 will see some other card. If T2 draws first, then it will see the ace of spades and T1 will see some other card.

---

12. (8 points) In response to your race condition, Louis Reasoner says "Use the monitor pattern!" Explain very briefly what "use the monitor pattern" means here, and why your race condition cannot be solved by applying the monitor pattern to `Deck`. Don't use more space than the box allows.

> "Use the monitor pattern" means synchronize the public methods (move() and draw()) on

> the Deck object's own lock. It doesn't help here, because the race condition is caused by interleaving *between* methods, and synchronizing the individual methods

Deck

```
/** Represents a mutable deck of distinct playing cards. */
public class Deck {

    // rep: ...

    // creates a deck starting with 52 unique playing cards in unspecified
    public Deck() {
        ...
    }

    // removes and returns top card from deck, unless deck is empty
    public Optional<Card> draw() {
        ...
    }

    // moves card so that it is nth from the top of this deck
    // (i.e. move(card, 0) makes card the top card of the deck).
    // No effect if card is not found in this deck.
    public void move(Card card, int position) {
        ...
    }


}
```

Card

```
/** Represents an immutable threadsafe standard playing card. **/
public class Card {
    // immutable set containing all the standard playing cards
    public static final Set<Card> ALL_52_CARDS = ...;

    // operations: ...
}
```