5 Abstracting over Data Definitions, Methods

Portfolio Problems

Work out as complete programs the following exercises from the textbook. You need not work out all the methods, but make sure you stop only when you see that you really understand the design process.

Problems:

1. Problem 19.4 on page 271.

2. Problem 19.15 on page 285.

Pair Programming Assignment

Note: Please, download the new version of *tester*. It has been posted on February 5th at 12:30 pm.

5.1 Problem

Complete Part 5.3 of the Lab 5, dealing with bank accounts.

Hand in the solution after you have designed all abstractions.

5.2 Problem

Complete Part 5.4 of the Lab 5, dealing with equality.

5.3 Problem

Rewrite the `same` methods in the previous problem using the `instanceof` operator and *casting* as shown on the lab instructions after the item 5.3-7.

Define a new class `PremiumChecking` that extends `Checking` as follows. The Premium checking account includes a bonus points field. A customer gets one point for each $20 deposited. Define the `same` method for the `PremiumChecking` class, following the same technique as has been used in the *TestSame.java* example. Add test cases that shows how this definition fails.

5.4 Problem

We would like to consider the typical problems we encounter in designing interactive games. The *geometry* library provides us with a simple class `Posn`, but this comes with no methods for manipulating the data.

We know that there are some typical questions that involve manipulating `Posn`s, and so we would like to build a framework that can encapsulate the typical behaviors.

To do so, we decide to build a class `CartPt` that `extend`s the class `Posn` with the following functionality:

1. We want to make sure the location is always within bounds of our `Canvas`. To enforce this, **design the method `adjust`** that consumes the *width* and the *height* of the `Canvas` and produces a `CartPt` with the coordinates *moved* to fit within the `Canvas`. If the `x` coordinate is negative, it changes to zero, if it is greater than the given `width` it is set to the `width`. The `y` coordinate is handled similarly.

2. We often need to move the `CartPt` by some distance. **Design the method `move`** that produces a new `CartPt` with the coordinates at a location moved by the given `dx` and `dy`.

3. At times we need to move the `CartPt` by a random distance within some range. **Design the method `moveRandom`** that produces a new `CartPt` moved by the given random range in both the horizontal and vertical direction.

   So, for example, for our falling star that moves down 5 pixels on each tick, but moves randomly in the range from -2 to +2 horizontally, the method call would be

   ```
   this.loc.moveRandom(-2, 2, 5, 5)
   ```

   indicating that the `x` coordinate changes by any value in the range [-2, 2] and the `y` coordinate changes by a value in the range [5, 5] (so it must be 5).

4. Finally, **design the method `closeTo`** that determines whether `this CartPt` is close to the given `CartPt` within the given distance.

Having a library class that provides all this functionality should make the design of our games much easier to write —- and much easier to read as well.

Note 1: To define a random number we use methods similar to those used in ProfessorJ languages. Unfortunately, the implementation of the `nextInt` method for the class `Random` in the `java.util` library is different from that provided in ProfessorJ. In Java the method `nextInt` has two options:

```
// produce a new random integer in the range [0, n)
// throws exception if n <= 0
int nextInt(int n);

// produce a new random integer
// any valid value is equally likely
// including all negative values and zero
int nextInt()
```

Be sure to make the necessary changes in your program.

Note 2: Please, download the new version of *tester*. It has been posted on February 5th at 12:30 pm.