3 Methods for Simple Classes

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 10.3 on page 97
2. Problem 10.4 on page 97
3. Problem 11.2 on page 116
4. Problem 12.1 on page 129
5. Problem 12.4 on page 131

Pair Programming Assignment

3.1 Problem

Start with the file **City.java** from Problem 2.1 from the previous assignment.

Design the following methods for the class that represents one city:

A. the method `sameState` that determines whether a city is in the given state.

B. the method `isSouthOf` that determines whether one city is located South of another city.

C. Design the method `distanceTo` that computes the distance from one city to another. (See the problem 1.1 C) for help with figuring out how to compute the distance.)

D. Design the method `toPosn` that produces a `Posn` that corresponds to the location of this city in a 100 x 100 Canvas. (Add *import geometry.*;* statement to the beginning of your program.)

E. Design the method `draw` that shows this city as a small disk in the given `Canvas`. Assume the `Canvas` has the size 100 x 100. You may want to also show the name of the city.

Note: These is no way one can test this method. However, include the code that will display at least three cities in a `Canvas`.

3.2 Problem

The file *Banking.java* contains the definitions of classes the represent bank accounts.

A. Make examples of the following accounts:

- A checking account for Adam Smith with id 123, a minimum balance of $50 and current balance of $150.
- A savings account for Betty Jones with id 456, a balance of $120 and interest rate of 2.5%.
- A certificate of deposit account for Pat Malloy with id 334, a balance of $300 that has not yet matured.

B. Design the method `amtAvailable` for the classes that represent bank accounts that produces the amount that the customer can withdraw from the account.

C. Design the method `moreAvailable` that determines whether one account has more available for withdrawal than another account.

D. Design the method `withdraw` that produces a new account with the given amount withdrawn. If the amount the customer wants to withdraw exceeds the available amount, no money will be withdrawn.

Note: Later we will learn how we can signal that the transaction is not valid.

3.3 Problem

Creative Project

Select the simplest version of your game. Your data definitions should not contain self-reference (no lists, binary trees, combo shapes). If you wish, your world may contain a fixed number of objects of the same kind (e.d. five fish, four invaders, etc.).

Design the following methods for this simple version of your game:

A. the method `draw` that will display the world state in the given `Canvas`.

   Include in the `Examples` class a *visual test* that shows the initial *world* and a *world* at some point during the game. The lab sample program *DrawFace.java* shows you how to make this happen.

B. Add to your class the following method:

```
// signal the end of the world and display the final message
MyWorld endOfWorld(String message){
return this;
}
```

   (do not change anything here, other than the name of your *world* class. Invoke it in the two methods you define below, when the conditions for the ending of the game are satisfied.

C. the method `onKeyEvent` that consumes a `String` and produces a new instance of your *world* in response to the given key. The arrow keys are defined as `"left"`, `"right"`, `"up"`, and `"down"`, the space bar is defined as `"space"`.

   Note: If some key event leads to the end of the game, that case should return `this.endOfWorld("end of world message")`.

D. the method `onTick` that produces a new instance of your *world* after one clock tick elapsed.

   Note: If on tick we determine that the game ends, that case should return `this.endOfWorld("end of world message)"`.

Design one method at a time, make sure you follow the *Design Recipe*, and once all the parts are there, you are almost ready to run the game.

Note: I will be more impressed with a well designed simple game than with a game that has all kinds of fancy options, but the code is not readable, methods are jumbled together, there are no tests, and there are no purpose statements.

When you are ready to run the game do the following:

- Change the line that defines you `GameWorld` class to be:

```
class GameWorld extends World{
```

Of course, you will use whatever is the name of your class that defines your world. If you have named it just `World`, you need to change its name to something different.

- Comment out your method the method `endOfWorld`.
- Make the return type for your methods `onKeyEvent` and `onTick` be just `World`.
- Include on your `Examples` class

```
boolean go = this.myInitialWorld.bigBang(200, 300, 0.1);
```

— assuming you have defined `myInitialWorld` in the `Examples` class, want your `Canvas` to be 200 pixels wide and 300 pixels tall, and want the clock tick at every 0.1 second. Of course, you choose your own names, sizes, and the speed.

- Run the program as usual.
- Have fun.