3 Methods for Complex Class Hierarchies; Libraries

Portfolio Problems

- **Methods for a grocery store**

  Work out the problem 14.7 in the textbook.

- **Sorting the runner's log**

  Work out the problem 15.4 (page 156) in the textbook.

- **Methods for FEDEX shipping**

  The shipping company from the previous assignment keeps a list of all packages ready to be shipped. Design the data definition for a list of packages and then design the methods to solve the following problems:

  - Find out if the total weight of all packages exceeds the truck weight limit. (Method name *withinWeight*)
  - Produce a list of all packages going to a given customer. (Method name *packagesFor*)
  - Produce a list of URLs for all customers for the packages in today's shipment. Every customer's URL should be in the list only once. (Method name *customerList*)
  - Sort the packages by their weight. (Method name *sortByWeight*)

- **Methods for a Soccer Team Phone Tree**

  Design the following methods for the soccer team phone tree classes:

  - Count the number of players a coach has on a team. (Method name *countPlayers*)
  - Produce the name of the player with the given phone number. If the phone number does not appear in the phone tree, produce an empty *String*. (Method name *whosePhone*)
  - Does one player (*the caller*) call another player (*the callee*) directly? (Method name *isCallee*)

    Note: If the caller is not in the phone tree, the answer is *false*.

Pair Programming Assignment

Binary Search Trees

3.1 Problem

Here is an HtDP data definition:

;; A Binary Search Tree (BST) is one of
;; — empty
;; — Node

;; A Node is (make-node Number BST BST)
(*define-struct node* (*value left right*))

;; we expect the value to be a whole number

The BST (the binary search tree) has the property that all values in the left sub-tree are smaller than the value of the node, and all values of the right subtree are larger than the value of the node. (We will not allow the same number to be a value of more than one node in the tree.)

1. Define the Java class hierarchy that represents a BST.
2. Design the method that counts the number of *Node*s in a *BST*. (Method name *count*)
3. Design the method that adds the values of all nodes in the BST. (Method name *totalValue*)
4. Design the method that determines whether a given number is one of the node values in the BST. (Method name *contains*)
5. Design the method that inserts a new node with the given value into the right place in the BST. If there already exists a node with the given value, the method produces the BST that looks the same as the original one. (Method name *insert*)
6. Design the method that produces the smallest number recorded in the BST. (Method name *first*)
7. Design the method that removes the node with the smallest value from the BST. The method produces a new BST. (Method name *rest*)

Morphing a Polygon

3.2 Problem

Design the following methods for the classes that represent polygon data that you have defined in the previous two assignments:

1. Count the number of points in the polygon. (Method name *count*)
2. Produce a morphed polygon from two original ones, with the given morphing factor (a number between 0.0and 1.0). Make sure you follow the *one task, one method* rule. (Method name *morphPoly*)
3. Draw the polygon on the *Canvas c*. (Method name *drawPoly*) Use a sample program that uses the *draw* library as a guide - or consult the **Help Desk**.

City Map

We continue designing classes that help us draw the city map and its attractions. For this problem you do not need any of the classes from the previous assignment, other than the class *Place* that represents a location on the map.

3.3 Problem

Develop the data definition to represent a route through the city. We are especially interested in being able to locate specific intersections of streets, or named squares, plazas. etc. and to deal with city streets.

1. Design the class *Xing* that represents an intersection on a city map. It should include a name and the location information.
2. Next we need to define the class that represents a segment of the street that connects two intersections. For each street segment we need to know the name of the street and its starting and ending intersection.

   Design the class *Road* to represent one street segment.
3. Finally, we need a list of street segments that may represent either a routing in the directions generated by a map program, or the list may represent one street in the city.

   Design the classes to represent a list of *Road*s — call it a *Route*.

4. Design the method that determines the distance each *Road* covers. (Method name *roadLength*)

5. Design the method that computes the length of a *Route*. (Method name *routeLength*)

6. A *Route* provided by a map program must have the street segments connected to each other — i.e. next segment must start where the previous one ended. Design the method *isRoute* that determines whether a *Route* represents valid map directions.

7. A street in a city not only consists of adjacent street segments, but additionally, every segment has the same street name. Design the method *isStreet* that determines whether a *Route* represents a valid city street data.

8. **Optional**

   Design the methods and classes that will draw each of the intersections as a black dot and will draw the streets as well. You do not need to add the names to the map — simple dots and black lines are fine.