4 Methods for Self-Referential Data; Abstracting over Data Definitions

4.1 Methods for Self-Referential Data.

4.1.1 Problem: Mobiles

This problem continues the work on mobiles we have started during one of the earlier lectures. The file **MobileMethods.java** contains the data definitions, examples of data, and the method `countWeights`.

A. Make an additional example of mobile data that represents the following mobile (The number of dashes in the struts and lines

B. represents their length):

```
               |
    - - - - - - - - - - - - -
  |                         |
 60                         |
blue            - - - - - - - - - - -
               |                     |
               |                    40
            - - - - - -             red
           |            |
          10            |
         green          5
                       red
```

C. Design the method `totalWeight` that computes the total weight of a mobile. The weight of the lines and struts is given by their lengths (a strut of length *n* has weight *n*).

D. Design the method `height` that computes the height of the mobile. We would like to hang the mobile in a room and want to make sure it will fit in.

Make sure you keep updating the *TEMPLATE* as you go along. (We have already started you on your way.)

4.2 Abstracting over Data Definitions.

Review of Designing Methods for Unions of Classes.

A file in a computer can contain either a text, or an image, or an audio recording. Every file has a name and the owner of the file. There is additional information for each kind of file as shown in the program **Files.java**.

Download the file and work out the following problems:

A. Make one more example of data for each of the three classes and add the tests for the method `size` that is already defined.

   Now design the methods that will deal with the files:

B. Design the method `downloadTime` that determines how many seconds does it take to download the file at the given download rate.

   The rate is given in bytes per second.

C. Design the method `sameOwner` that determines whether the owner of this file is the same as the owner of the given file.

Save the work you have done. Copy the file and continue.

Abstracting over Data Definitions: Lifting Fields

Save your work. Possibly start a new project and import the file into it. Alternatively, save the a copy of the file you have been working on in another folder.

Look at the code and identify all places where the code repeats — the opportunity for abstraction.

Lift the common fields to an abstract class `AFile`. Make sure you include a constructor in the abstract class, and change the constructors in the derived classes accordingly. Run the program and make sure all test cases work as before.

Abstracting over Data Definitions: Lifting Methods

For each method that is defined in all three classes decide to which category it belongs:

A. The method bodies in the different classes are all different, and so the method has to be declared as `abstract` in the `abstract` class.

B. The method bodies are the same in all classes and it can be implemented concretely in the `abstract` class.

C. The method bodies are the same for two of the classes, but are different in one class — therefore we can define the common body in the `abstract` class and override it in only one derived class.

Now, lift the methods that can be lifted and run all tests again.

Note: You can lift the method `sameOwner` only if you change its contract. Do so — make sure you adjust the test cases accordingly.

4.3 Complex class hierarchies

Goals

We will focus on understanding the connection between information and data. The lab handout contains a class diagram and definitions of objects in this class hierarchy. Your goal is to read the data and explain what information it represents. In the second part of this lab task you will then add methods to these classes that allow us to ask questions about the given data.

Class hierarchy and its data

For this problem you will work with the classes define by the following *Scheme-like* data definitions:

```
/*
;; A Boss is (make-sup String String Number [Listof Employee])
(define-struct sup (name unit tasks peons))

;; A Worker is (make-worker String Number))
(define-struct worker (name tasks))

;; An Employee is one of
;; -- Boss
;; -- Worker

;; A [Listof Employee] is one of
;; -- empty
;; -- (cons Employee [Listof Employee])
```

and represented by the following class diagram:

```
                             +-----------+
                             |   Emp     |<-----------------------------+
                             +-----------+                              |
                             |           |                              |
                             +-----------+                              |
                                   |                                    |
                                   |                                    |
                                  /_\                                   |
                                   |                                    |
                   +-----------------------------------+                |
                   |                                   |                |
           +----------------+                 +------------------+      |
           |    Worker      |                 |    Boss          |      |
           +----------------+                 +------------------+      |
           |String name     |                 | String name      |      |
           |int tasks       |                 | String unit      |      |
           +----------------+                 | int tasks        |      |
                                     +--------+ ListofEmp peons  |      |
                                     |        +------------------+      |
                                     |                                  |
                                     v                                  |
                               +--------------+                         |
                               |  ListofEmp   |<------------------------+--+
                               +--------------+                         |  |
                               +--------------+                         |  |
                                      |                                 |  |
                                      |                                 |  |
                                     /_\                                |  |
                                      |                                 |  |
                        +--------------------------------+              |  |
                        v                                v              |  |
               +-----------------+              +-------------------+   |  |
               |  MTListofEmp    |              |    ConsListofEmp  |   |  |
               +-----------------+              +-------------------+   |  |
               +-----------------+              | Emp first         +---+  |
                                                | ListofEmp rest    +------+
                                                +-------------------+
```

The following collection of data examples represent some information this class hierarchy can represent:

```
// examples/tests for the classes to represent a company employee hierarchy
class Examples {
  Examples() {}
  Emp wkA = new Worker("A",3);
  Emp wkB = new Worker("B",5);
  Emp wkC = new Worker("C",6);
  Emp wkD = new Worker("D",4);
  Emp wkE = new Worker("E",5);
  Emp wkF = new Worker("F",2);
  Emp wkG = new Worker("G",8);
  Emp wkH = new Worker("H",6);

  ListofEmp mtlist = new MTListofEmp();

  ListofEmp grpAlist = new ConsListofEmp(this.wkC,this.mtlist);
  Emp mike = new Boss("Mike", "Group A", 10, this.grpAlist);
  ListofEmp secAlist =
  new ConsListofEmp(this.mike,
                  new ConsListofEmp(this.wkD,
                  new ConsListofEmp(this.wkE,this.mtlist)));
  Emp jack = new Boss("Jack", "Section A", 25, this.secAlist);

  ListofEmp secBlist =
    new ConsListofEmp(this.wkF,
    new ConsListofEmp(this.wkG, this.mtlist));

  Emp jenn = new Boss("Jenn", "Section B", 15, this.secBlist);

  ListofEmp secClist = new ConsListofEmp(this.wkH,this.mtlist);
  Emp pat = new Boss("Pat", "Section C", 20, this.secClist);

  ListofEmp secDlist = new ConsListofEmp(this.wkB, this.mtlist);
```

```
Emp pete = new Boss("Pete", "Section D", 10, this.secDlist);

ListofEmp operList =
  new ConsListofEmp(this.jack,
  new ConsListofEmp(this.jenn,
  new ConsListofEmp(this.pat, this.mtlist)));

Emp dave = new Boss("Dave","Operations", 70, this.operList);

ListofEmp financeList =
  new ConsListofEmp(this.wkA,
  new ConsListofEmp(this.pete, this.mtlist));

Emp anne = new Boss("Anne", "Finance", 20, this.financeList);

ListofEmp ceoList =
  new ConsListofEmp(this.dave,
    new ConsListofEmp(this.anne,this.mtlist));

Emp meg = new Boss("Meg","CEO", 100, this.ceoList);
```

A. Think about the information this data represents and describe the company hierarchy that the given data represents. Draw a chart so you can easily tell who works for which group, who are the bosses for a given employee, etc.

   Can you tell how many subordinates does `Dave` have?, who are the bosses of worker `wkA`, etc.

B. Design the method `countAll` that will count all people the given employee oversees. Include self in the count.

C. Design the method `allUnit` that produces all subordinates of this worker. Include self in the list.

D. Design the method `isBoss` that consumes a name and determines whether the employee or one of its subordinates is a boss with the given name.