4 Abstracting over Data Definitions; Complex Class Hierarchies

4.1 Abstracting over Data Definitions.

A bank customer can have three different accounts: a checking account, a savings account, and a line of credit account.

- The customer can withdraw from a checking account any amount that will still leave the minimum balance in the account. The customer can withdraw all money from a savings account. The balance of the credit line represents the amount that the customer already borrowed against the credit line. The customer can withdraw any amount that does not make the balance exceed the credit limit.

- The customer can deposit money to the account in any amount. If the customer deposits more to the credit line than the current balance, the balance will become negative — indicating overpayment.

Review of Designing Methods for Unions of Classes.

The code in *banking.java* defines the classes that represent this information.

1. Make examples of data for these classes, then make sure you understand the rules for withdrawals.

   Now design the methods that will manage the banking records:

2. Design the method `canWithdraw` that determines whether the customer can withdraw some desired amount.

3. Design the method `makeDeposit` that allows the customer to deposit a given amount of money into the account.

4. Design the method `maxWithdrawal` that computes the maximum that the customer can withdraw from an account.

5. Design the method `moreAvailable` that produces the account that has more money available for withdrawal.

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

Abstracting over Data Definitions: Lifting Fields

Save your work and change the language level to Intermediate ProfessorJ.

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

Lift the common fields to an abstract class `ABanking`. 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:

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

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

3. The methods look very similar, but each produces a different variant of the union — therefore it cannot be lifted to the `super` class.

4. 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.

4.2 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:

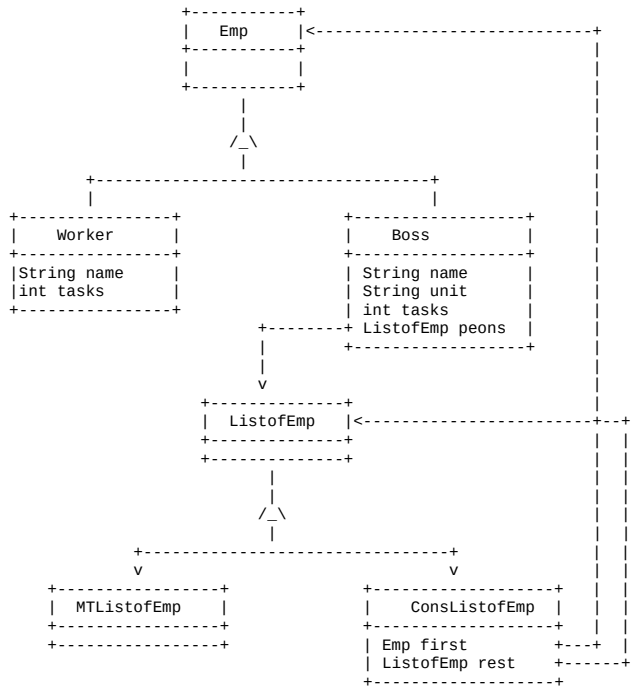
```
/*
;; A Boss is (make-sup String String Number [Listof Employee])
(define-struct sup (name unit tasks peons))

;; 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:



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);
```

1. 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.

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

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

4. 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.