5   Abstracting over Data Definitions; Accumulators

5.1   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 *lab5-banking.bjava* 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 open it again with the file type 'ijava'. 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.

5.2   Designing Recursive Methods with Accumulators

Our goal is to understand how we can use accumulators to keep track of the knowledge we have acquired as we progress towards the program solution.

We are given a list of `String`s and want to combine all of them into one `String`. We also want to find the length of the longest `String` in this list.

Here is an example:

```
Initial list:
"Mary "
"had "
"a "
"little "
"lamb."

Expected result:
"Mary had a little lamb."
```

Our solution that follows the DESIGN RECIPE is easy:

```
// to represent a list of Strings
interface ILoS{
  // combine all Strings in this list into one
  String combine();

  // find the length of the longest word in this list
  int maxLength();
}

// to represent an empty list of Strings
class MtLoS implements ILoS{
  MtLoS(){}

  // combine all Strings in this list into one
  String combine(){
    return "";
  }

  // find the length of the longest word in this list
  int maxLength(){
    return 0;
  }
}

// to represent a nonempty list of Strings
class ConsLoS implements ILoS{
  String first;
  ILoS rest;

  ConsLoS(String first, ILoS rest){
    this.first = first;
    this.rest = rest;
  }
```

```
  // combine all Strings in this list into one
  String combine(){
    return this.first.concat(this.rest.combine());
  }

  // find the length of the longest word in this list
  int maxLength(){
    return Math.max(this.first.length(),
                    this.rest.maxLength());
  }
}

// to represent examples for lists of strings
class Examples{
  Examples(){}

  ILoS mary = new ConsLoS("Mary ",
               new ConsLoS("had ",
                new ConsLoS("a ",
                 new ConsLoS("little ",
                  new ConsLoS("lamb.", new MtLoS()))))));

  boolean testCombine =
    check this.mary.combine()
    expect "Mary had a little lamb.";

  boolean testMaxLength =
    check this.mary.maxLength() expect 7;
}
```

Designing the loops with accumulators

We now convert this methods to use the accumulator style. As we did in Scheme, we will design a helper method that takes one more argument, the accumulator. The accumulator at any given point in the computation will hold the "answer" to our question thus far.

The first question to ask is:

Step 1: What is the type of the expected result?

This determines not only the return type for the method, but also the type of the value that will be accumulated. If the type is `AccType`, we add an

argument `AccType acc` to the method signature and add `Acc` to its name.

That means we add the following methods to the interface:

```
// combine all Strings in this list into one
String combine();

// combine all Strings in this list into one
// with the given String
String combineAcc(String acc);

// find the length of the longest word in this list
int maxLength();

// find the length of the longest word in this list
// given the max length so far
int maxLengthAcc(int acc);
```

The next step is to change the purpose statements to explain the meaning of the accumulated value.

Step 2: Change the purpose statements for all new methods.

For example, we say:

```
  // combine all Strings in this list into one
  // adding the Strings from the list to the String
  // that contains all the Strings we have found so far
  String combineAcc(String acc);

  // find the length of the longest word in this list
  // if it is longer that the known longest length so far
  // otherwise, produce the known longest length
  int maxLengthAcc(int acc);
```

Next we need to figure out what are the values each method produces when it is invoked with the empty list. Think of where can you find out. We call this the **base value**.

Step 3: Make a list of the base values for all new methods.

Of course, the base value for the `maxLength` method is `0` . What is the base value for the method `combine`?

We now deal with the following problem. The original question did not mention the accumulator. That means that the method that uses the accumulator is only our helper method. The original method needs to invoke this helper so that it would produce the original result.

There are two steps that make this possible.

Step 4: Invoke the method that uses the accumulator using the base value as the initial accumulator value.

For example we get:

```
// find the length of the longest word in this list
int maxLength(){
  return this.maxLengthAcc(0);
}
```

As the things stand now, we need to add this method body to the method definition in both the empty class (`MtLoS`) and the class that represents a nonempty list (`ConsLoS`). In the next section we will learn how to do this only once.

Revise the definition of the method `combine` in a similar way.

The next step deals with the class that represents the empty list.

Step 5: Implement the method body in the class MtLoS.

The purpose statement makes it clear that the method just produces the value of the accumulator. This makes sense. If the original method is invoked by an instance of the empty class, it produces the base value, as expected.

Implement all method bodies for the class `MtLoS`.

Now comes the hardest part of the whole process.

Step 6: Design the update method.

Let us examine what happens inside of the method body in the class `ConsLoS`. We list again all original method definitions here:

```
// combine all Strings in this list into one
String combine(){
  return this.first.concat(this.rest.combine());
```

```
}

// find the length of the longest word in this list
int maxLength(){
  return Math.max(this.first.length(),
                  this.rest.maxLength());
}
```

In each case, there is some computation that involves `this.first` and the recursive invocation of the original method by the list `this.rest`.

The `update` method has the following common structure:

```
// update the value of the accumulator using this.first value
AccType updateMethodName(Song first, AccType acc){ ... }
```

For the method `maxLength` the `updateMaxLength` method is defined as:

```
// update the value of the longest word so far
int updateMaxLength(String first, int acc){
  return Math.max(this.first.length(), acc);
}
```

Design the `update` method for the `combineAcc` case. And, yes, *do follow the design recipe*.

We are just about done.

Step 7: Complete the body of the method using the update method.

The method that uses the accumulator is the same for all cases. The instance of the `rest` of the list invokes the method self-referentially, using the updated value of the accumulator.

For the `count` method this will be:

```
// find the length of the longest word in this list
// if it is longer that the known longest length so far
// otherwise, produce the known longest length
int maxLengthAcc(int acc){
 return
   this.rest.maxLengthAcc(this.updateMaxLength(first, acc));
}
```

Finish designing the body of the `combineAcc` method. When done, run the original tests for the main methods as well as all of your other test cases.

Save the work you have done.