19 Lecture: Getting a New Phone Number

Goals: - Modifying fields, modifying structures.

Introduction

A cell phone keeps lists of phone numbers for people we call often. We can assign people to groups. However, a person can be a member of several groups. So a friend can also be a co-worker, a co-worker may be a family member as well. For examples we may have the following lists:

Family: Bob 2345, Matt 1234, Anne 7897
Friends: Anne 7897, John 8866
Work: John 8866, Jane 3456

Obviously, these can be represented as three lists of *Object*s where each element will be an instance of *Person*. We have the same classes as before: *ILoObject*, *MTLoObject*, and *ConsLoObject*.

The information above translates into the following examples:

```
Person bill = new Person("Bill", 2345);
Person matt = new Person("Matt", 1234);
Person anne = new Person("Anne", 7896);
Person john = new Person("John", 8866);
Person jane = new Person("Jane", 3456);


ILoObject mtphlist =
        new MTLoObject();
ILoObject family =
        new ConsLoObject(bill,
        new ConsLoObject(matt,
        new ConsLoObject(anne,
                         mtphlist)));


ILoObject friends =
        new ConsLoObject(anne,
        new ConsLoObject(john,
                         mtphlist));
```

```
ILoObject work =
        new ConsLoObject(john,
        new ConsLoObject(jane,
                         mtphlist));
```

When Jane gets a new phone number, we can easily design the method *newPhone* that will produce a new list with Jane's new phone number.

So, we need a method *changePhone* that produces a phone list with the number for the given name changed to the given number. (We assume all the names are different.)

The purpose statement and the method header is

```
// change the phone number for the person with the given name
ILoObject changePhone(String name, int phone);
```

which will appear the same way in the **interface** *ILoObject*.

Next we make examples:

```
boolean testChangePhone1 =
  this.family.changePhone("Anne", 4444).same(
  new  ConsLoObject(bill,
          new ConsLoObject(matt,
          new ConsLoObject(new Person("Anne", 4444),
                           mtphlist))));


boolean testChangePhone2 =
  this.friends.same(
          new ConsLoObject(new Person("Anne", 4444),
          new ConsLoObject(john,
                           mtphlist)));
```

Of course, if Anne's phone number changes, we want that to be the case in every list where her phone number appears.

Designing the method is easy — this is a task we have done many times before. We get:

```
//------- In the class MTLoObject:
  ILoObject changePhone(String name, int phone){
    return this;
  }
```

```
//------ In the class ConsLoObject:
  ILoObject changePhone(String name, int phone){
    if (((Person)this.first).name.equals(name))
      return new ConsLoObject(new Person(name, phone),
                              this.rest);
    else
      return new ConsLoObject(this.first,
                              this.rest.changePhone(name, phone));
  }
}
```

We now run the tests. The first one succeeds, but the second one fails! We examine the *friends* list and see that Anne's phone number in that list is still 7896 — it did not change. The *family* phone list has no way of knowing which persons in that list are also included in some other list. Therefore, it cannot go and replace a *Person* object by a new instance of *Person*. If the change is to be seen by all lists that refer to this *Person*, the data that the object represents need to change — we need to change the value of the *phone* field of the given person.

We start by adding a method *newPhone* to the class *Person*. This method only changes the phone number and produces no new result.

The purpose statement and the header are:

```
// effect: change the phone number for this person to the given number
void newPhone(int newNumber){
  this.phone = newNumber;
}
```

We actually wrote the code — it was so simple. However, the tests are harder. We cannot test the value produced by the method - this method does not produce any value. We can only observe the *effects* of this method. The tests for such method always consists of three parts:

- **Before:** Define the data to be used in the test.
- **Run:** Run the method to be tested.
- **After:** Evaluate the effects of the method invocation.

Here are the tests for the method *newPhone*:

```
// tests for the method newPhone in the class Person
boolean testNewPhone(){
  Person rick = new Person("Rick", 3344);
  Person alie = new Person("Alie", 7777);
  rick.newPhone(5566);
  return rick.samePerson(new Person("Rick", 5566)) &&
         alie.samePerson(new Person("Alie", 7777));
}

boolean testNewPhoneResult = this.testNewPhone();
```

We can now proceed to changing the phone number of the person in one of the phone lists. It is our explicit intent that this change be reflected in all lists where this person appears - and anywhere this person's information is known.

Our examples from above will serve as the **Before** and **After** parts of the test. The method header and purpose will be:

```
// give a new phone number to the person with the given name
// effect: the person's phone number changes wherever person is referenced
void newPhone(String name, int phone);
```

In the empty case we do nothing. Alternately, we could throw an exception indicating that the person is not in our list. The methods for both classes then become:

```
//------- In the class MTLoObject:
  void newPhone(String name, int phone){}

//------- In the class ConsLoObject:
  void newPhone(String name, int phone){
    if (((Person)this.first).name.equals(name))
      this.first.newPhone(phone);
    else
      this.rest.changePhone(name, phone);
  }
```

Our next problem is that we want to remove a person from one of the phone lists. However, our cell phone has a master list of all lists — and so, again, we cannot produce a new list — we can only change the existing one. So, we design the method header, the purpose statement and the effect statement:

```
// remove the person with the given name from this list
// effect: the person will no longer be in this list
void removePerson(String name);
```

and follow up with examples/tests:

```
// tests for removal of a person from a list
boolean testRemovePerson1(){
  ILoObject list1 =
            new ConsLoObject(bill,
            new ConsLoObject(anne,
            new ConsLoObject(matt,
                             mtphlist)));


  ILoObject list2 =
            new ConsLoObject(anne,
            new ConsLoObject(john,
                             mtphlist));
  list1.removePerson("Anne");
  return list1.contains(anne) == false &&
         list2.contains(anne) == true;
}

boolean testRemovePerson2(){
  ILoObject list1 =
            new ConsLoObject(bill,
            new ConsLoObject(anne,
            new ConsLoObject(matt,
                             mtphlist)));


  ILoObject list2 =
            new ConsLoObject(bill,
            new ConsLoObject(john,
                             mtphlist));
  list1.removePerson("Bill");
  return list1.contains(bill) == false &&
         list2.contains(bill) == true;
}
```

```
boolean testRemovePerson3(){
  ILoObject list1 =
            new ConsLoObject(bill,
            new ConsLoObject(anne,
            new ConsLoObject(matt,
                             mtphlist)));

  ILoObject list2 =
            new ConsLoObject(anne,
            new ConsLoObject(matt,
                             mtphlist));
  list1.removePerson("Matt");
  return list1.contains(matt) == false &&
         list2.contains(matt) == true;
}

boolean testRemovePersonResult1 = testRemovePerson1();
boolean testRemovePersonResult2 = testRemovePerson2();
boolean testRemovePersonResult3 = testRemovePerson3();
```

The eaxmples are not complete. We do not have the case where the list contains only one person and that person is removed from the list. A method that is invoked by an instance of *ConsLoObject* class cannot change that instance to become an instance of another class (in our case that would be *MTLoObject* class. We can only change the values of first and rest of any instance of a *ConsLoObject* that invokes the method. Here is a first attempt at solving the problem. If the first item is the person we wish to remove, then replace the first with the next person in the list, i.e. *this.rest.first — the first person in the rest of this list* and make the rest of the new list be the rest of the rest of this list:

```
void removePerson(String name){
  if (((Person)this.first).name.equals(name))
    if (this.rest instanceof ConsLoObject){
      this.first = ((ConsLoObject)this.rest).first;
      this.rest = ((ConsLoObject)this.rest).rest;
    }
    else {... error - should not happen ...}
  else
    if (this.rest instanceof MTLoObject)
      {... error - we do not know what to do here ...}
    else
          ((ConsLoObject)this.rest).remove(name);
}
```

We see a problem here. This only works when the rest of the list is not an empty list. There is no systematic way to remove the last person from the list. We could try to design a helper method that only removes the last element form the list — but we do not know that the person to be removed is the last in the list until we get to the end. At that point, we no longer know where we started.

The loss of knowledge suggest that we use accumulator to remember information needed later. The problem can now be divided into two cases: either the person to remove is the first one in the list, or we can remember the list with the previous element as we look at the next element. Essentially, the new method *removeAfter* will remove the given person after the current one.

Here is the purpose statement, the effect statement, and the header:

```
// remove the given from this rest of the list, knowing original list
// effect: the given person will no longer be in the original list
void removeAfter(String name, ConsLoObject acc){...}
```

It is a strange statement — the method has effect on the accumulator passed to it as the argument. Our three original tests should work here too. The rest of the work follows the design recipe. The problem statement divides the method body into two cases - either *this.first* is the person to be removed, or we need to look further. Looking further is done by recursion — *this.rest.removeAfter*(*name*, **this**). To remove the current item, i.e. *this.first* we need to change *acc* by setting its first to be the first of the rest of the list that invoked the method, and setting its rest to be the rest of the rest of **this** list.

Now, write down the template for each case and describe in words what is the meaning of each part. Make examples of how the method is invoked and label all parts with the items in the template. Then read aloud the purpose statements for the method invocations. After you have done so, try to finish the body of the method by yourself, without looking at the solution.

This is not a nice way to solve the problem. There is no nice way. The problem is really hard. The only way to get around the problem is to design a level of abstraction between the list and its user. This is called a *wrapper*. We design a new class *MutableList* that represents a mutable list, and contains as its only field *ILoObj* list. All changes to the field use the immutable methods we have seen before, but after new list is produced, the result is assigned to the list that represents the current list. For example, the remove method would be:

```
/*---------- in the classes ILoObject:
// produce a list from this list with the given object removed
ILoObj removeILoObj(ISame obj)...


/*---------- in the class MutableList
// effect: this list will have the given object removed
void removeMutableList(ISame obj){
  this.list = list.removeILoObj(obj);
}
```

The second strategy is much preferred for clarity, though there are times when the first strategy is unavoidable.

Warning — Danger

Consider the following example;

```
Person bill = new Person("Bill", 2345);
Person matt = new Person("Matt", 1234);
Person anne = new Person("Anne", 7896);
Person john = new Person("John", 8866);
Person jane = new Person("Jane", 3456);
```

```
ILoObject mtphlist =
        new MTLoObject();
ILoObject work =
        new ConsLoObject(bill,
        new ConsLoObject(matt,
        new ConsLoObject(anne,
                         mtphlist)));

ILoObject friends =
        new ConsLoObject(john,
                         work));
```

We wish to remove Matt from the work list, but he still remains a friend. We expect the following test to be true, but it fails!

```
boolean testRemovePerson4(){
  ILoObject mtphlist =
          new MTLoObject();
  ILoObject work =
          new ConsLoObject(bill,
          new ConsLoObject(matt,
          new ConsLoObject(anne,
                           mtphlist)));

  ILoObject friends =
          new ConsLoObject(john,
                           work);

  work.removePerson("Matt");
  return work.same(new ConsLoObject(bill
                   new ConsLoObject(anne,
                   mtphlist))) &&
         friends.same(new ConsLoObject(john,
                      new ConsLoObject(bill,
                      new ConsLoObject(matt,
                      new ConsLoObject(anne,
                      mtphlist)))));
}

boolean testRemovePersonResult4 = testRemovePerson4();
```

Even though we never referred to the *friends* list, the two lists shared the structure and changing one changed the other as well. Here is a diagram illustrating the problem: