3 Designing Methods — Part 2

We continue with the the theme of the photo images. Our goal is to design methods that answer questions about list of images and manipulate these lists.

We will also work with the geometric shapes and learn to draw their images on the Canvas. At the end we will design a simple interactive program.

3.1 Methods for Self-Referential Data

In the previous lab you designed a list of photo images. We first design the method that counts the images in our list.

Note: Of course, you will quickly realize that this method will look the same regardless of what are the pieces of data contained in the list. We will address that issue later on, once we are comfortable with dealing with lists that contain specific items.

Below is an example of the design of a method that counts the number of pictures in a list of photo image.

The method deals with *ListOfPhotos*. We have an interface *ListOfPhotos* and two classes that implement the interface, *MTListOfPhotos* and *ConsListOfPhotos*. When the DESIGN RECIPE calls for the method purpose statement and the header, we include the purpose statement and the header in the interface *ListOfPhotos* and in all the classes that implement the interface.

Including the method header in the interface serves as a contract that requires that all classes that implement the interface define the method with this header. As the result, the method can be invoked by any instance of a class that implement the interface - without the need for us to distinguish what is the defined type of the object.

We can now proceed with the DESIGN RECIPE.

- Step 1: Problem analysis and data definition.

  The only piece of data needed to count the number of elements in a list is the list itself. The result is an integer.

  We will use the following data in our examples. For your work add at least one more instance of each class.

```
// Examples for the class ClockTime
ClockTime ct1 = new ClockTime(9, 50, "pm");
ClockTime ct2 = new ClockTime(11, 30, "am");
ClockTime ct3 = new ClockTime(9, 50, "am");


// Examples for the class Date
Date d1 = new Date(2006, 9, 23);
Date d2 = new Date(2006, 11, 7);
Date d3 = new Date(2006, 9, 25);


// Examples for the class Photo
Photo river = new Photo("River", "jpeg", 3456, 2304, 3614571,
                        this.d1, this.ct3);
Photo mountain = new Photo("Mountain", "jpeg", 2448, 3264, 1276114,
                           this.d2, this.ct2);
Photo people = new Photo("People", "gif", 545, 641, 13760,
                         this.d2, this.ct1);
Photo icon = new Photo("PLTicon", "bmp", 16, 16, 1334,
                       this.d1, this.ct2);


ListOfPhotos mtlist = new MTListOfPhotos();
ListOfPhotos list1 = new ConsListOfPhotos(this.river, this.mtlist);
ListOfPhotos list2 = new ConsListOfPhotos(this.mountain,
                     new ConsListOfPhotos(this.people,
                     new ConsListOfPhotos(this.icon, this.mtlist)));
```

- Step 2: The purpose statement and the header.

```
// to count the number of pictures in this list of photos
int count(){...}
```

In the interface *ListOfPhotos* we write:

```
// to count the number of pictures in this list of photos
int count();
```

indicating there is no definition for this method.

We now have to design the method separately for each of the two classes.

- Step 3: Examples.

  We make an examples for the empty list, a list with one element and a longer list:

  mtlist.count() −−−> 0
  list1.count() −−−> 1
  list2.count() −−−> 3

- Step 4: The template.

  We need to look separately at the two classes that implement the method.

  class MTListOfPhotos: The class has no member data and there is no other data available. It is clear that the method will always produce the same result, the value 0.

  We can finish the steps 4. and 5. right away — the method body becomes:

```
// to count the number of pictures in this list of photos
int count() {
  return 0;
}
```

  The template for the class *ConsListOfPhotos* includes the two fields, *this.first* and *this.rest*. However, just as in HtDP, we recognize that *this.rest* is a data of the type *ListOfPhotos* and so it can invoke the method *count* that is now under development. The template then becomes:

  class ConsListOfPhotos

```
int count(){
 ... this.first ...        −−− Photo
 ... this.rest ...         −−− ListOfPhotos

 ... this.rest.count() ... −−− int
```

  Recall the purpose statement for the method *count*:

```
// to count the number of pictures in this list of photos
```

That means the purpose of the method invocation *this.rest.count*() is

```
// to count the number of pictures in the rest of this list of photos
```

When designing methods for self-referential data, make sure you say out loud (or at least understand clearly) the purpose statement as applied to the self-referential method invocation.

- Step 5: The method body.

  We have already finished the method body for the class *MTListOfPhotos*. In the class *ConsListOfPhotos* the method body is:

```
// to count the number of pictures in this list of photos
int count(){
  return 1 + this.rest.count();
}
```

- Step 6: Tests.

  We can now convert our examples into tests:

```
// Tests for the method count:
boolean testPixels = (check this.mtlist.count() expect 0) &&
                     (check this.list1.count() expect 1) &&
                     (check this.list2.count() expect 3);
```

Design the methods that will help you in dealing with your photo collection:

1. Before burning a CD of your photos, you want to know what is the total size in bytes of all photos in the list of photos.

2. When organizing your photos you want to know whether all photos in the list are from the given date.

   Remember to use helper methods and to delegate the tasks to the classes that are best suited to provide the desired answer.

3. You now want to go over the list of photos and select only those that were taken on the given date.

3.2 Quiz

You have 10 minutes.

3.3 Methods for Self-Referential Data — Part 2 Graphics and Key Events

In the previous lab you defined classes that represent different geometric shapes - a circle, a square, and a shape that is a combination of two shapes, the top and the bottom one.

1. Design the method that computes the total area of a shape. For the shape that consists of two components add the areas - as if you were measuring how much paint is needed to paint all the components.

   You will need to use math functions, such as square root. The following example show how you can use the math function, and how to test *double*s for equality. (You can only make sure they are different only within some given tolerance.)

```
class Foo{
  double x;

  Foo(double x){
    this.x = x;
  }

  double squareRoot(){
    return Math.sqrt(this.x);
  }
}

class Examples {
  Examples () {}

  Foo f = new Foo(16.0);

  boolean testSquared =
    check this.f.squareRoot() expect 4.0 within 0.01;
}
```

2. Design the method that produces a new shape moved by the given distance in the vertical and horizontal direction.

3. Design the method that determines whether the given point is within this shape.

4. Of course, we would like to draw the shapes on a canvas. The following code (that can be written within the *Examples* class shows how you can draw one circle:

```
import draw.*;
import colors.*;
import geometry.*;

class Examples{
  Examples() {}

  Canvas c = new Canvas(200, 200);

  boolean makeDrawing =
    this.c.show() &&
    this.c.drawDisk(new Posn(100, 150), 50, new Red());
}
```

   The three *import* statements on the top indicate that we are using the code programmed by someone else and available in the libraries named *draw*, *colors*, and *geometry*. Open the *Help Desk* and look under the *Teachpacks* for the teachpacks for *How to Design Classes* to find out more about the drawing and the *Canvas*.

5. Finally, add some interaction to your program, by letting the shape move in response to the key events. Design the method *onKeyEvent* that consumes a *String* and moves the shape up, down, left, or right by 5 pixels every time the user hits one of the corresponding arrow keys.

   Use the code in the program *WorldDemo.ijava* to figure out how to respond to the key events and to see how to run the program. To get the program working, you need to use *ProfessorJ Intermediate Language* — but do so only for this part of the lab.

Save all your work — the next lab will build on the work you have done here!

If you have some time left, work on the *Etudes* part of the homework, especially on the **Etude 3.2** that deals with *Binary Search Trees*.