Lecture 11, Object Oriented Programming, Part II

December 4, 2001

Reminders

Object oriented programming is a style of programming that relies on five main things

- Classes
- Objects (these are instances of the class)
- Methods/Generic functions. Generic functions are dispatcher while methods are the *real* functions. Making methods available only through the generic functions is considered to be a good design feature.
- Inheritance – we will consider that next.
- Polymorphism – more on that later.

A major reason for using OO programming is that it supports (and in fact encourages) code reuse. That is, the same code (functions, procedures, data structures) can be used in many different places. This is a good thing because it means that you can do not have to always reimplement something.

Code reuse is mainly encouraged through inheritance. Inheritance is slightly confusing because the terminology is a new (and sort of goes in the wrong direction).

Inheritance

Suppose that I define a class shape (or geometric-shape, but we'll use shape). Now one of the interesting features of our shapes is area. So we will define shapes to have areas.

```
setClass("shape", representation( area = "numeric") )
```

Now, we might want to define a number of shapes such as `circle`, `triangle`, `rectangle`, `polygon`, `square`. If we define a new class for each one of these shapes and we say that they extend the shape class then we get two things,

- We have an `area` slot which is inherited from `shape`.
- Any one of these objects can be used where a `shape` object is required.

```
setClass("triangle", representation("shape", a = "numeric",
b="numeric", c="numeric") )
tr1 <- new("triangle", a=3,b=4,c=5)

setClass("rectangle", representation("shape", a = "numeric",
b="numeric") )
rtgl1 <- new("rectangle", a=3,b=4)

setClass("square", representation("shape", a = "numeric") )
sq1 <- new("square", a=3)
```

So that even though squares and rectangles have a lot on common it is not clear how to use that to advantage.

Some terminology. The class `square` is said to *inherit* from the class `shape`. In Splus you should always use the function *inherits* to determine inheritance. Another common description is that `shape` is a *superclass* of `square` and hence that `square` is a *subclass* of `shape`. Now some people find this confusing since `square` has more slots than `shape` since it gets all of `shape`'s and any of its own. So `square` is in some sense bigger – but it is still the subclass. The reason for that is that we calling something a subclass or a superclass we generally are thinking of the inheritance tree.

In a good class system the classes form a tree–like structure. The top of the tree is the base element (often called *object*) and inherited by all classes (unfortunately S does not quite work this way). Then you can think of a new class as a node and

it has an edge to every class that it *extends* (or that is a superclass of that class). This gives us a tree structure. Most good implementations prevent there from being cycles in that graph (or tree). We don't want to say that A inherits from B and that B inherits from A. That doesn't really make any sense and causes some programmatic as well as logical difficulties.

Virtual Classes

It is sometimes convenient to link together classes into a hierarchy underneath a *virtual* class. This class does nothing more than provide a common ancestor for all of the derived classes. Consider the `shape` class described above. We never want to have an instance of a shape. It would make no sense really. Shape is an abstract concept that ties together our notions of rectangle, circle, and so on. Generally, a virtual class cannot be instantiated.

In the S methods system there is a virtual class called `vector`. This class is extended by all the different types of vectors. Integer vectors, character vectors etc. These different flavors all share a common structure and will respond to some common questions. Examples of common functions:

[The subset operator (and of course, `[<-`, the subset assignment operator).

length All vectors should be able to tell us how long they are.

The list is much longer of course, but you should see the idea. I can implement these methods once for all vectors. Other methods, such as printing will be handled differently for the different vector–types.

This is an example of code reuse, or perhaps of good design which prevented code explosion.

Notice that inheritance and abstraction are closely related. One of the most powerful tools we can bring to bear on any problem is abstraction (not just programming problems). Inheritance helps me out because it allows me to think of shape–type solutions for shapes and about circle–type solutions for circles.

1 Polymorphism

One of the real strengths of generic functions is that they support polymorphism. If you have done any plotting in SPlus or R you have used polymorphism. The

code `plot(x)` does not know until run time what `x` will be and hence it does not know which `plot` method will be called.

In the S system polymorphism is used a lot by `plot`, `summary` and `print` (or `show` in Splus6).

There is another set of polymorphic operators (functions in some sense but not quite).

```
> get("+")
function(e1, e2)
if(missing(e2)) e1 else .Internal(e1 + e2, "do_op", T, 5)
```

Notice that `+` is simply a function. Now, many of these operators can be grouped together. Then, methods can be defined for the group.

```
> getGroupMembers("Arith")
[1] "+"   "-"   "*"   "^"   "%%"  "%/%" "/"
> getGroupMembers("Ops")
[1] "Arith"   "Compare" "Logic"
```

So you can define methods specifically for any operator, eg a `+` method, or a method for the `Arith` group or for the `Ops` group.

Lets consider the following example. Suppose that I have some longitudinal data. Say, some time points and some values. For example blood pressure measured at specific times for a bunch of people. A class for this might be

```
setClass("lgtdl", representation(times="numeric", values="numeric"))

lgtdl1 <- new("lgtdl", times=1:10, values=rnorm(10))
lgtdl2 <- new("lgtdl", times=1:10, values=runif(10))
```

Now, suppose I want to write a method for adding two together. What might that mean? Well, I guess if the `times` are the same then we could add the `values`. So something like:

```
setMethod("+", signature(e1="lgtdl", e2="lgtdl"),
   function(e1, e2) {
       if( !identical(e1@times, e2@times) )
          stop("can only add lgtdl's with identical times")
       e1@values = callGeneric(e1@values, e2@values)
       e1
    })
```

If I wanted to have code for subtraction, multiplication etc. I could do that. However, I can simply define a method for the `Arith` class and by inheritance it will apply to all the operators.

```
\begin{verbatim}
setMethod("Arith", signature(e1="lgtdl", e2="lgtdl"),
   function(e1, e2) {
       if( !identical(e1@times, e2@times) )
          stop("can only add lgtdl's with identical times")
       e1@values = callGeneric(e1@values, e2@values)
       e1
    })
#now you can use
 lgtdl1/lgtdl2
```

You can always define a specific method for any one of these operators. That specific method will have precedence over any `Arith` method, which will have precedence over any `Ops` method. Again, inheritance plays a role. When S decides which method to use it takes the most specific one it can find. That makes sense because you have written that specific method and want it to be used.

Here again, we have real code reuse (or viewed differently a lack of code propagation). For a more complete version of `lgtdl` see `www.cran.r-project.org`

The generic function, + and friends, can be thought of as *polymorphic*. They determine the types of their arguments and dispatch on them. This allows us to think of a natural set of generic functions such as `plot`, `print`, without having to think of how the name has been mangled to get a different plot for each type of object that we want to plot.

Some important steps in any object oriented design process:

1. identify the objects and their attributes.
2. determine what sorts of things can be done to each object.
3. determine what the objects do to each other (how do they interact?)
4. determine which parts of an object should be visible
5. decide on a public interface to the object

Some of these are not available in the current S methods system. But hopefully over time they will become available (this system is now under active development and is evolving).

A problem

I want to store the results of a simulation that I have done into a list. I also want to be sure that that list contains only the results of a `lm` fit. If that is true then it is much simpler to write other functions (methods) that operate on the list.

We define a container to be any list whose elements are all of a specific type (we will change this slightly a little bit later).

If you look in the file called `container.R` in

```
~rgentlem/SplusExamples
```

you will see some code to implement this class.

Basically a container is simply a list and I have done two things to it. I have controlled what can go into the list and I have controlled whether what is in the list can be changed (`locked` prevents changes).

In both cases I have implemented that behaviour by controlling access to the replacement method for lists. The replacement method is actually the function `[[<-` which is very special. It is hard to get this to work correctly so don't be too discouraged if you try and it doesn't work.

The problems are caused by the fact that S is basically a pass–by–value language and that this is a function. So the function gets a copy of the list and we want to change the list, not its copy. These calls are rearranged in a special way (details provided to the curious) and then evaluated. There are two important things when writing one of these functions. First, the last argument must be named `value` and second the last line of the function must return the copy of the list (it is `x` in the current case).

Containers are interesting and useful because they allow you to keep objects of a certain type together. They can indicate, by error messages when you try to put the wrong type object into the container.

Some old code

Some triangle functions from last day, they are repeated here for completeness.

```
tr1 <- new("triangle", a=3, b=4, c=5)

##Law of cosines: aˆ2 = bˆ2+cˆ2-2*b*c*cos(A)
threesidestoangle <- function(a=a, b=b, c=c)
```

```
   acos((b^2+c^2-a^2)/(2*b*c))

angleA <- function(object)
   threesidestoangle(a=object@a,b=object@b, c=object@c)

angleB <- function(object)
   threesidestoangle(object@b, object@c, object@a)

angleC <- function(object)
   threesidestoangle(object@c, object@a, object@b)

if( ! isGeneric("area") )
   setGeneric("area", function(object) standardGeneric("area"))

setMethod("area", signature(object="triangle"),
          function(object)
              object@a*object@b*sin(angleC(object))/2 )

area(tr1)
```