Object Oriented Programming in S-PLUS

Robert Gentleman

Short Course

Auckland, New Zealand

February 2003

Outline

- Abstract Data Types
- Classes
- Methods
- S3 Classes and Methods
- S4 Classes and Methods
- Examples

Introduction

This module deals with advanced programming concepts. The tools we will discuss can help you to write better programs and functions and can help you to understand how and why good programmers function.

Writing good programs is very much an art of applied abstraction. Once the *correct* abstraction is found the programming often is quite simple.

Wirth: Algorithms plus data structures = programs.

One thing to remember about object oriented programming is that working in this style transfers a great deal of the programming effort from actual programming to design.

Introduction

Over the next few hours we will explore some of the basic concepts involved in *object oriented programming*.

I will first introduce the necessary ideas in an abstract setting and then show how they relate to the **two** systems available in the S language.

I have a prejudice, the old S3 class system (it isn't an object system) is very nice for interactive programming but is not sufficiently robust to support real software engineering.

Introduction

Some reasons to perform object oriented programming:

1. productivity increase
2. easier to maintain code
3. reusable code
4. the design tends to follow the objects being modeled

Object Oriented Design

One identifies real objects and the operations on them that are interesting.
These operations can then by systematically implemented.

For example: we might have pdfs and cdfs as different objects. methods might be means, median, maxima and so on.

A basic principle (or perhaps hope) is that by faithfully representing the objects we get easier to implement functions.

A cdf object should know how to answer all the cdf questions.

Abstract Data Type

This is a simple idea that can make your programs much more modular and can make it relatively easy to change them.

Using abstract data types is little more than ensuring that your programs do not depend on the representation (or implementation) of a particular data structure, but rather on the functionality that it is intended to provide.

Suppose that we have fit the following model, `lm1 <- lm(y~x)` and you want to get the residuals.

Abstract Data Type

You can do one of:

- `lm1$residuals`
- `resid(lm1)`

Which is better?

Good arguments can be made to suggest that the second approach is better. The first relies on the fact that the returned value from the function `lm` is a list with a component named `residuals`. This is not likely to always be true.

Abstract Data Type

A program that relies on the functional approach to retrieving the residuals will be more robust.

When `lm` is modified to come in line with the new modeling code the output will be an instance of an S4 class, not a list.

All code that relies on it being a list will be broken.

All code that relies on the function `resid` will still work. Because a new version will be written that does work.

Abstract Data Types

We want to think of the object in terms of what we would do with it, not in terms of how we have implemented it.
Think of a probability density – what sorts of things do you want to do?
plot, integrate, find the max, find the mode

Do you care how it is implemented?

Programming Tip One

Avoid any dependence on implementation.

If need be write your own *accessor* functions.

Suppose that you are about to embark on a project that uses needs to access components of a data structure then you should think about writing the accessors as functions.

This simple device will let you completely change the data structure at will with very little impact on your program.

Classes

A class is an abstract definition of a concrete real world object.

Suppose you are writing software to help a company manage its human resources. Then the objects are people and any program that implements those objects in format that reflects that will be both easier to write and easier to maintain.

A class system is software infrastructure that is designed to help construct classes and to provide programmatic support for dealing with classes.

Classes: HR Example

To implement a class structure for our HR example we do not need a lot of support (and in fact, what we need more than anything is adherence to an abstract data type).

We could implement the class using lists in the S language.

Each employee will be represented by a list. The list will have named components, `name`, `salary`, `capabilities`.

We call the named components `slots`.

Classes: HR Example

An employee (virtual) is said to be an instance of the employee class.

It is important to distinguish between the class definition and the objects that represent that class.

Classes: HR Example

Now, clearly there could be some advantage to making capabilities a class of its own (and the same could hold true for the other two).

If we are careful never to assume much about the actual format of the data in the capabilites slot we may be able to exchange its representation quite painlessly.

In S3 there is no way to formally define a class. An object is determined to be an instance of a particular class by examining its `class` attribute.

This attribute is a character string and it has one entry for each class that the object represents.

Classes

A second major aspect of classes is the use of *inheritance* or of extension.

Suppose on our HR example we have employees and bosses. Now a boss is simply an employee with some extra attributes.

A good way to design this is to have the `boss` class extend the employee class.

In most object systems (but not S3) there are mechanisms for having one class extend another. In S3 the programmer is largely left to manage the details.

Classes

The basics of a class system are:

- some way to define the class and to create instances of the class
- some way to specify that a class extends one or more other classes.
- some way to access and alter the values in the slots of a class
- some way to identify the class that an object belongs to

Notice that you need remarkably few constructs to have a working class system.

Classes vs Objects

An object is a specific entity that exists at run time.

A class is a static entity that exists in the program code.

A class describes how to create an object (and in some languages how to interact with that object)

We instantiate an object with a call to a constructor. Usually it has a nice name like new.

```
x <- new("circle", r=3.3)
```

Now x is an instance of the circle class.

Methods

A method is very much like an ordinary function. About the only difference is how it is invoked.

Both S3 and S4 rely on the notion of a generic function and method dispatch.

Generic functions are the way in which the S language provides *polymorphism*. Polymorphism is the property that the same function provides different outputs depending on the *types* of its arguments.

Methods

The argument list of a method is often called its *signature*.

The signature of the generic controls the signatures of the methods defined for the generic.

It is common (especially in S3) to include `...` as an argument in all generics and methods. This ensures that others can extend the definitions without too many restrictions.

Method Dispatch

The basic idea is that the generic function contains (or controls access to a set of methods).

When call is made to the generic function three things are done:

1. the list of available methods is obtained
2. the methods are arranged in order from most specific to least specific
3. the most specific method is called

Method Dispatch

The determination of which method is most specific is made based on the classes of the arguments to the methods (and on the class structure that they induce – ie. who do they inherit from)

Different programming languages have different ways of defining this. The specific details are not yet fully documented for S.

In S3 dispatching is done on the first argument only (most of the time). In S4 dispatching is done on all arguments.

There is a function called `NextMethod` that can be used to call the next most specific method from within the body of any method.

Method Dispatch

Getting the right semantics can be difficult. In S, where the language is dynamic and the inheritance tree could change during the course of evaluation, it is particularly difficult.

In S3 the dispatch mechanism is quite loose. Objects can change their class (or have it changed for them). This is quite bad programming style.

In S4 the mechanism is much tighter and objects cannot change their type or class (except through coercion).

The S3 Class System

This is a very loose collection of functionalities that can be used to achieve certain goals.

I don't like to call it an object system since there is no real notion of objects within this system.

Any S3 object can become an instance of any class it wants to simply by attaching a class attribute with the correct name.

There is no notion of one class extending another (at least in terms of which slots are available).

The S3 Class System

Method dispatch is done entirely by convention. The generic functions have no notion of which methods are associated with them.

The location of methods is determined by string concatination. A function named `foo.bar` is presumed to be a `bar` method for the class `foo`.

There is no way to prevent that presumption.

There is no way to know what the role of a function named `foo.bar.baz` is. It will be interpreted as both a `bar.baz` method for the class `foo` **and** as a `baz` method for the class `foo.bar`.

The S3 Class System

When a method is accessed through the generic function the type of the first argument is known.

In many cases the programmer would like to take advantage of that and not do much checking.

Since S3 methods are just ordinary functions this is not possible. They can be called directly and so all checking must be done.

There is no safety. You cannot presume that because you got an object of class `matrix` that it is a matrix. It can be anything.

S4 Object System

The object system in S4 is much richer.
It has:

- a way of defining classes, handling inheritance
- generic functions that register specific methods
- multiple dispatch

It should be possible to develop large scale extensible software systems using this functionality. There is a reasonable amount of effort being expended to both define the langauge and to implement a version that satisfies both the users and the developers.

S4 Object System

Defining classes in S4 is done with the function `setClass` which has the following arguments (in order):

- `Class`: a character string to name the class
- `representation`: the names and types of all slots.
- `prototype`: a specification of what new instances should look like at creation time. This can also be controlled by an `initialize` method for the class.
- `validity`: a function that checks the validity of an instance (is it a valid member of the class.
- `access`, `where`, `version`, left for the interested reader to explore.

S4 Object System

We can define a class and create an instance from that class.

```
  setClass("foo", representation(a="character"),
                  prototype=list(a="ccc"))
  x1 <- new("foo")
  x1
An object of class "foo"

Slot "a":
[1] "ccc"
```

S4 Object System

This class `foo` can easily be extended:

```
  setClass("bar", representation("foo", b="numeric"))
  x2 <- new("bar")
  x2
An object of class "bar"
Slot "a":
[1] "ccc"
Slot "b":
numeric(0)
```

To include `foo` in the class definition we simply gave its name. Notice that the prototype for `foo` was used in initializing the new instance of `bar`.

Slot Access

Access to the values in a slot in S4 is via the `@` operator for *getting* values and via `@<-` for setting values.

For example, `x2@a` will return `"ccc"`.

And `x2@b <- 10` will set the `b` slot to have value 10. You can then print `x` to see this.

Accessing slots directly **breaks** the data abstraction. You are now relying on the implementation.

Abstract Data Types: Revisited

Consider a class that represents triangles (which we will do some exercises on shortly).

That class can represented in many different ways; the three angles involved, the lengths of the three sides, two sides and one angle and so on.

We might want to do some calculations based on triangles (find their area).

If we use `x@area` then we are relying on there being an area slot.

Abstract Data Types: Revisited

If we use `area(x)` then we are free to implement it as we see fit.

It could be a slot,

```
setMethod("area", "triangle", x@area)
```

S4 Object System

If instead you wanted to have a slot in your class that contained an object of class `foo` that is different. Then you do

```
setClass("baz", representation(a="foo", b="numeric"))
new("baz")
```

A `baz` instance has two slots (like a `bar` instance) but one of those slots is a `foo` instance, while for a `bar` instance the slot is a `character`.

There is in principle no problem with having self-referential definitions (although the S-PLUS implementation seems to get annoyed),

```
setClass("xx", representation(a="xx", b="numeric"))
```

Virtual Classes

A *virtual class* is a class for which no instances can be made.

The purpose of a virtual class is to provide some common stucture that is shared by a number of classes for which instances can be created.

There are two ways to create a virtual class

1. have no representation in the call to `setClass`.
2. include the class `VIRTUAL` in the representation.

Virtual Classes

The `"vector"` class is a virtual class.

All the specific types of vectors, e.g. `character`, `numeric` extend the vector class.

Try `getClass("vector")`.

Then `getMethods("length")` will show a method for the virtual class.

By using a virtual class, we can show the common structure inherited by all vectors and allow each of them to extend that structure in appropriate ways.

S4 Generic Functions

Defining generic functions is quite simple.

```
setGeneric("myFoo", function(object)
             standardGeneric("myFoo"))
```

This defines a generic function and installs a default method.

This method is called if no other method is found to handle a call to the generic function `myFoo`.

In most cases the appropriate action is to signal an error and indicate that no method was found. That is basically what the call to `standardGeneric` does.

S4 Generic Functions

Once the generic function has been defined you can start adding methods.

The signature of the generic function (and of the methods) is the set of names and types of arguments.

The choice of arguments for the generic can limit the methods that can be defined on it, so some care may be needed.

```
setMethod("myFoo", "character",
          function(object) print(object))
```

Defines a method for the generic `myFoo` that simply prints its value.

S4 Generic Functions

The behavior of the call `myFoo(1)` is to signal an error. There is no method for numeric arguments.

On the other hand, `myFoo("a")` will print `a`.

You can remove generic functions, `removeGeneric`, and methods, either individually (`removeMethod`) or as a group (`removeMethods`).

These can be quite handy for program development.

S4 Methods

To get some feel for polymorphism. Imagine that in your application you believe that it is sensible to interpret the addition operator, `+`, as operating on strings by concatenating them.

Then defining a method on `+`:

```
setMethod("+", c("character","character"),
    function(e1,e2) paste(e1,e2, sep=""))
```

has the desired affect.

Addition on numbers still works as it did before. We say that the addition function is polymorphic.

Replacement Methods

One of the conceptual hurdles in dealing with the S language is learning to understand the pass-by-value semantics.

If every operation is carried out by a function call and arguments are always copied then how does `x[1]<-10` work?

The understanding of this is important for understanding replacement methods.

The semantics of `x[1]<-10` are:

```
x<- do.call("[<-", list(x, 1, value=10))
```

Replacement Methods

So what happens is first, `x` is copied, an the value of the first element of `x` is changed to 10.

The function `[<-` returns an object just like `x` but with the appropriate values changed.

Finally, S's evaluator rebinds the symbol `x` to this new value.

Thus, you seem to have pass-by-reference in a pass-by-value language.

Replacement Methods

For replacement methods the strategy is similar. First a generic function needs to be created (with a `<-` suffix), then the replacement method defined.

Note that the last argument is named `value` and that the method returns the altered copy of its first argument.

These are both necessary for the approach to work.

Replacement Methods: Example

```
setGeneric("a<-", function(x, value)
           standardGeneric("a<-"))

setReplaceMethod("a", "foo",
 function(x, value) {
   x@a <- value
   x
 })

 a(b) <- 32
```

Documentation

There are some tools being developed in the R language for documentation of S4 methods and classes.

These are at quite an early stage of development, but they should be ready for users in either S-PLUS or R sometime this year.

There are tools, macros and so on, available at the C level for working with S4 classes and methods.

See Appendix A.6 in *Programming with Data* for specific details on the implementation. You should also consult any on-line materials as this interface may be implemented in different ways on different platforms.