S4 Classes in 15 pages, more or less

February 12, 2003

Overview

The preferred mechanism for object–oriented programming in R is described in Chambers (1998). The actual implementation is slightly different (in some parts influenced by Dylan, (Shalit, 1996)). This document provides a short introduction to programming using these classes and methods. It is worth noting that R also supports an older class system (as described in Chambers and Hastie (1992) but we recommend developing all new programs using the new system and will not refer to the older system further.

The object system is class–based with multiple inheritance and generic functions that can dispatch on any set of arguments (*signature*). In some sense the concepts of classes and methods are distinct and we will deal with them separately. The final section of this document will deal with documenting classes and methods.

Classes

A class is defined using `setClass`.

```
setClass(Class, representation, prototype,
  contains=character(), validity, access, where=1, version=FALSE)
```

Where the arguments are:

Class character string; the name of the class

representation the slots that the new class should have and/or other classes that this class extends. Usually a call to the 'representation' function.

prototype An object providing default data for the slots. New instances will pick up these values unless explicitly overridden at time of construction.

contains The classes that this class extends. All slots of parent classes will be propagated to new instances of the extending class.

validity, access, version Control arguments included for compatibility with S-Plus.

where The environment to use to store or remove the definition as metadata.

Once a class has been defined *instances* of that class can be created using the function `new`. The class defines the structure of an object. The instances of that class represent the objects themselves.

Classes can provide an abstraction of complex objects that helps to simplify programming with them. Appropriate design is essential and this is where much of the programming effort (rather than in actual coding) should be focused.

A class can extend one or more other classes. We can think of the class hierarchy as being a directed graph (or tree). The extended classes are considered to be parents of the class that extends them. The graph must be acyclic – no class can be a parent of itself. Extension means that the new class will contain all of the slots that the parent classes have. It is an error to create a class with duplicate slot names (either by inclusion or directly).

```
> library(methods)
> setClass("foo", representation(a = "character", b = "numeric"))

[1] "foo"

> setClass("bar", representation(d = "numeric", c = "numeric"))

[1] "bar"

> setClass("baz", contains = c("foo", "bar"))

[1] "baz"

> getClass("baz")
```

```
Slots:

Name:          a         b         d         c
Class: character   numeric   numeric   numeric

Extends:
Class "foo", directly.
Class "bar", directly.
```

Now we can create an instance of *baz*. You do not need instances of *foo* or *bar* to do so, *baz* objects can be instantiated directly.

Access to the values in a slot is through a special operator, the `@` symbol. There are many advantages to not accessing the values in slots directly but rather to define and use special methods for accessing the values. We will discuss some of these issues in the next section when methods are discussed.

```
> x <- new("baz", a = "xxx", b = 5, c = 10)
> x

An object of class "baz"
Slot "a":
[1] "xxx"

Slot "b":
[1] 5

Slot "d":
numeric(0)

Slot "c":
[1] 10

> x@a

[1] "xxx"
```

Virtual Classes

Virtual classes are classes for which no instances will be (can be) created. Rather, they are used to link together classes which may have distinct representations (and hence cannot inherit from each other) but for which we

want to provide similar functionality. A fairly standard mechanism is to create a virtual class and to then have several other classes extend it.

In practice the virtual classes are then used in different ways:

1. Methods for the virtual class will apply to any of the classes that extend the virtual class.

2. A slot in a new class can have as its type the virtual class. This allows slots to be polymorphic.

3. If a virtual class has slots they will be common to all classes that extend the virtual class.

Suppose that we want to define a class that represents a dendrogram. Each node in a dendrogram has three values associated with it. The height, the left node and the right node. Terminal nodes are different they have a height and a value (possibly an instance of a yet another class).

One way to implement this structure is to use a virtual class called `dendNode`, say. Then there are two classes that we can define that extend this virtual class; terminal and non-terminal nodes.

```
> setClass("dendNode")

[1] "dendNode"

> setClass("dnode", representation(left = "dendNode", right = "dendNode",
+     height = "numeric"), contains = "dendNode")

[1] "dnode"

> setClass("tnode", representation(height = "numeric", value = "numeric",
+     label = "character"), contains = "dendNode")

[1] "tnode"
```

Now we can create dendrograms whose nodes are either terminal or non–terminal. The virtual class *dendNode* has been used to allow two different classes of objects as the `left` and `right` nodes of a dendrogram. This design makes recursive manipulation of dendrograms somewhat simpler.

A situation that seems to arise frequently is the desire to allow a slot in an object to either contain a *list* or to be `NULL()`. Since the object `NULL()` is not itself a list these cannot ordinarily *share* a slot. To overcome this

we could create a new virtual class that extends both the *list* class and the `NULL()` class.

As described on page 294 of Chambers (1998) we can operationalize these ideas with the following code:

```
> setClass("listOrNULL")

[1] "listOrNULL"

> setIs("list", "listOrNULL")
> setIs("NULL", "listOrNULL")
```

Now if we define an object with a slot that is of type `listOrNULL` it should accomodate either.

```
> setClass("c1", representation(value = "listOrNULL"))

[1] "c1"

> y <- new("c1", value = NULL)
> y

An object of class "c1"
Slot "value":
NULL

> y2 <- new("c1", value = list(a = 10))
> y2

An object of class "c1"
Slot "value":
$a
[1] 10
```

One can create a virtual class in two different ways. First, as shown above, if the call to `setClass` has no representation then the class will be a virtual class. Otherwise, include the class *VIRTUAL* in the representation.

```
> setClass("myVclass", representation(a = "character", "VIRTUAL"))

[1] "myVclass"

> getClass("myVclass")
```

```
Virtual Class

Slots:

Name:          a
Class: character
```

And we see that `myVclass` is indeed a virtual class with a single slot, `a`.

Initialization and prototyping

In some situations it is desireable to control the creation of new instances of a class. In some cases we might want to perform some computations or other initialization processes.

There are two different mechansims for doing this. The first is to use the `prototype` argument of `setClass`. Using this argument any of the slots can be provided with initial values.

```
> setClass("xx", representation(a = "numeric", b = "character"),
+     prototype(a = 3, b = "hi there"))

[1] "xx"

> new("xx")

An object of class "xx"
Slot "a":
[1] 3

Slot "b":
[1] "hi there"
```

And we see that new instances of the class `xx` will all have the values `3` and `"hi there"` associated with the slots `a` and `b`, respectively.

In some cases this is not sufficiently general and we might, for example want more control and flexibility. To achieve this we define an `initialize` method for our class. This method, if defined will be applied after the `prototype`, if one was defined.

```
> setMethod("initialize", "xx", function(.Object, b) {
+     .Object@b <- b
```

```
+     .Object@a <- nchar(b)
+     .Object
+ })

[1] "initialize"

> new("xx", b = "yowser")

An object of class "xx"
Slot "a":
[1] 6

Slot "b":
[1] "yowser"
```

Note that the last statement in the initialize method must return the *object*. This is required because of the pass–by–value semantics of R. While `initialize` gives the appearance of changing the values of the slots of its arguments it does not do so. Rather it creates a whole new object sets the values of its slots and returns that new object.

Generics and Methods

An equally important aspect of object oriented programming is the use of generic functions and methods. A generic function is essentially a dispatching mechanism. The methods are specialized functions that perform the required task on a specific type of input. The job of the generic function is to determine which of the methods is most applicable for a given set of arguments. Once this decision has been made the method selected is invoked with the appropriate arguments.

Much of the S language already uses generic functions and methods. The new system is a big improvement for several reasons

- Dispatching can be done on multiple arguments.
- Methods should not be called directly but only via the generic.
- The old style of dispatching made it virtually impossible to distinguish the appropriate dispatching for a function named `foo.bar.baz`.

When creating methods you usually need to first determine whether a generic function exists. If there is no generic function you must create one. An issue that may arise is the signature of the generic. The signature of the generic function limits the signatures of all methods defined for that generic function.

Once the signature for the generic has been specified methods can implement some or all of the arguments of the generic function but they cannot add any new arguments. This implies that the construction of the argument list for the generic is especially important. All potentially relevant arguments should be included.

Accessor Functions

Slots can always be accessed directly using the `@` operator. This practice is not recommended (for Bioconductor) since it produces code that depends on the actual class representation. Should that representation change, for example some slots dropped in favour of them being computed or renamed for some reason, then all of the code that uses the `@` operator must be identified and modified. Instead we recommend using the convention of accessing the slots via methods (while this produces a lot of generic functions the abstraction gained more than offsets this).

Suppose that the class `foo` has a slot named `a`. To create an accessor function for this slot you can proceed as shown below.

```
> setClass("foo", representation(a = "ANY"))

[1] "foo"

> if (!isGeneric("a")) {
+     if (is.function("a"))
+         fun <- a
+     else fun <- function(object) standardGeneric("a")
+     setGeneric("a", fun)
+ }

[1] "a"

> setMethod("a", "foo", function(object) object@a)

[1] "a"
```

```
> b <- new("foo", a = 10)
> a(b)

[1] 10
```

0.1 Replacement Methods

As noted previously one of the conceptual hurdles (and hence a real one) is dealing with the pass-by-value semantics of the S language. One of the reasons for object oriented programming is to use the computer representation to mimic real-world objects. Since real-world objects are mutable and their behavior depends on their history we would like to mimic that in the computer representation.

However, R has pass-by-value semantics and so every function operates on a copy of its arguments. Thus, change can not occur through function calls in a straightforward manner. The replacement methods are one mechansim that can be used to provide the appearance of pass-by-reference semantics.

Basically a replacement method *silently* replaces the whole object with a suitably modified copy of itself. For example, in `x[1]<-10`, it appears to the user that the 1st element of `x` has had its value changed to 10. But that is not what happens. First an analysis is done to determine that the symbol being operated on is `x()`. Then a copy of `x()` is made, the first value of the copy is changed, and finally the binding for `x()` is changed to the new value.

This process is automatic and most users are not aware of the true underlying semantics. The procedure is largely conventional. One of the requirements is that the last statement in any replacement method (function) is the whole object. A second requirement is that the last argument must be named `value`. This is to ensure that S can always identify the value that is going to be assigned.

We continue the example given above.

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

[1] "a<-"

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

[1] "a<-"
```

```
> a(b) <- 32
```

We also had to define an appropriate generic function. In this case the name of the generic is `a<-` since that is the traditional expression for assignment functions, the name of the function for which an assignment version is being defined concatenated with the assignment operator.

Finally, note that the value in the `a` slot of the object `b` has been *changed* (as noted really the whole object has been changed).

Documentation

Documentation of S4 classes and methods is aided by two functions in the *methods* package. They are `promptClass` and `promptMethods` both create documentation templates in the .Rd markup dialect.

First consider the top of the file `iter-methods.Rd` in the *Biobase* package.

```
\name{iter-methods}
\docType{methods}
\title{ Methods for the generic iter.}
\alias{iter-methods}
\alias{iter}
```

Here we have the name of the file, the documentation type (not sure just how it works yet), the title and some `alias`es. The aliases are used by R's help system to find the correct documentation. From the `help` man page,

```
help(topic, offline = FALSE, package = .packages(),
     lib.loc = NULL, verbose = getOption("verbose"),
     try.all.packages = getOption("help.try.all.packages"),
     htmlhelp = getOption("htmlhelp"),
     pager = getOption("pager"))
?topic
type?topic
```

We can see how the `?` operator works. It is either monadic or dyadic. In the dyadic form it constructs calls to `help` for the string `topic-type`.

Consider the `aggregator` class in the Biobase package. When R receives the command `promptClass("aggregator")`, it will dump the following text to the file `aggregator-class.Rd`:

```
\name{aggregator-class}
\docType{class}
\alias{aggregator-class}
\title{Class aggregator, ~~class for ... ~~ }
\description{  ~~ A concise (1-5 lines) description of what the class is  ~~}
\section{Creating Objects}{
\code{  new('aggregator',}\cr
\code{    aggenv  = ...., # Object of class environment}\cr
\code{    initfun = ...., # Object of class function}\cr
\code{    aggfun  = ...., # Object of class function}\cr
\code{  )}}
\section{Slots}{
  \describe{
    \item{\code{aggenv}:}{Object of class "environment" ~~ }
    \item{\code{initfun}:}{Object of class "function" ~~ }
    \item{\code{aggfun}:}{Object of class "function" ~~ }
  }
}

\section{Prototype}{
  \describe{
    \item{environment aggenv}{<environment> }
    \item{function initfun}{function (name, val) 1 }
    \item{function aggfun}{function (name, current, val) current + 1 }
  }
}
\section{Methods}{
  \describe{
    \item{aggenv}{(aggregator): ... }
    \item{aggfun}{(aggregator): ... }
    \item{initfun}{(aggregator): ... }
  }
}
\keyword{methods}
```

The developer should add narrative content to this skeleton, and copy the final file to the `man` section of the package implementing the class. See `Biobase/man/aggregator-class.Rd` for an example. Note that the `xyz-class` naming convention permits use of the binary version of `?`. Thus

`class?aggregator` is the command that retrieves the manual page for class aggregator.

Using *methods* in R packages

S4 methods and classes depend on the availability of metadata. For this reason suppling class definitions and methods in R packages requires a little more programming than what is needed for developing standard R packages (as described in the R Extensions Manual).

We will consider building a small package that will provide the class definitions defined earlier in this document. The first thing to do is to collect these functions into a single file. A relatively standard mechanism is to place all class, generic and method definitions inside of a function. In the example below the function is called `.initFoo`. The reason it has a name that starts with a dot is so that the user will not see it and inadvertantly call it.

```
.initFoo <- function(where) {
 setClass("foo", representation(a="character", b="numeric"),where=where)
  setClass("bar", representation(d="numeric", c="numeric"), where=where)
  setClass("baz", contains=c("foo", "bar"), where=where)
}
```

We then create a second file, this contains certain startup code needed when loading the R package. This file is traditionally named `zzz.R`.

```
.First.lib <- function(libname, pkgname, where) {
    if( !require(methods) ) stop("we require methods for package Foo")
    where <- match(paste("package:", pkgname, sep=""), search())
    .initFoo(where)
}
```

When a packages is loaded in R one of the initialization mechanisms is to search for a function called `.First.lib`. If such a function is found it is evaluated. In the case described above one of the actions taken is to evaluate the function `.initFoo`. That creates the class definitions and stores the appropriate metadata with the `Foo` package.

References

Chambers, J. M. (1998). *Programming with Data: A Guide to the S Language*. Springer-Verlag New York.

Chambers, J. M. and Hastie, T. (1992). *Statistical Models in S*. Wadsworth & Brooks/Cole.

Shalit, A. (1996). *The Dylan Reference Manual*. Apple Press.