Encapsulated Classes in R (Rough Draft)

Charlotte Maia

November 23, 2009

Abstract

A new class/object system for R is introduced, that is highly encapsulation based, as well as using an object reference system. A class definition has it's methods inside the the body of the definition. Object's can use self-reference to allow methods to access other members.

Introduction

There are a number of class/object systems available for R, including S3, S4 and R.oo. These are largely based on the principle of defining generic methods. Here we present an approach based on defining classes that (syntactically) encapsulate their methods. The approach also uses an object reference system, including self-reference.

A major advantage of this approach is that good object oriented implementations should match object oriented designs, and as encapsulation is a major principle of object oriented design, it is an important requirement of language (or language extension) supporting object oriented programming. Another major advantage is that we can avoid clashes in function names. e.g. The author has got her self in trouble in the past naming her own functions eval and cat (not a good idea).

The approach used here, which we shall refer to as m classes or m objects, uses a language extension implemented in the oosp package.

Class definitions are essentially functions (although they are not used in the way that functions are typically used) and objects are essentially environments. Object environments contain the object's attributes, however do not contain objects' methods. A non-standard system of method despatch is used.

Note that the system has had little testing and requires considerable more work. It is assumed that the method despatch process is relatively inefficient. However is also assumed that space usage is relatively efficient, which in principle, should be good for programs with large numbers of objects. These assumptions have not been tested.

Syntax

M classes are defined using functions. The name of the function represents the name of the class. The body of the function contains only other functions, except for the last line which should be a call to the mclass function. If the name of a nested function is the same as the name of the class it represents the constructor, otherwise it represents a method. It is possible to have automatic constructors (more on this later). If providing your own constructor, the arguments of the class definition should be just dots. A self reference is a single dot and attributes (not methods) are accessed in the same way as lists and environments. A simple example:

```
> MyObject = function (...)
 {
        MyObject = function (x) .$x = x
        double = function () .$x = .$x * 2
        mclass ()
 }
```

We create an m object by calling the m class definition as a function with whatever arguments are required for it's constructor. We access the attributes in the same way. However a formula operator is used to access object methods.

```
> myobject = MyObject (10)
> myobject$x
[1] 10
> myobject~double ()
> myobject$x
[1] 20
```

There are some objects created to represent the class which are hidden from the user. By default, they are only created once. One slight problem here, is that if we change our class definition, we need to explicitly delete the first class object before calling the new one's constructor. This is a likely to be changed in the next release.

```
> #only needed if changing the class definition
> mdelete ("MyObject")
```

The author has a preference for succinct constructors, and the succinctness possible constructor is an automatic one. Often we simply want to create an object with a set of values. The above example can be rewritten to use automatic construction.

```
> MyObject = function (...)
 {
        double = function () .$x = .$x * 2
        mclass ()
 }
> myobject = MyObject (x=10)
> myobject$x
[1] 10
```

Methods can call other methods, likewise a constructor can call other methods.

```
> MyObject2 = function (...)
 {
        MyObject2 = function (myname) {.$myname = myname; .~mymethod1 ()}
        mymethod1 = function () {cat (.$myname, "mymethod1\n"); .~mymethod2 ()}
        mymethod2 = function () {cat (.$myname, "mymethod2\n"); .~mymethod3 ()}
        mymethod3 = function () {cat (.$myname, "mymethod3\n"); .~mymethod4 ()}
        mymethod4 = function () cat (.$myname, "mymethod4\n")

        mclass ()
 }
> myobject2 = MyObject2 ("Charlotte's Object")
```

```
Charlotte's Object mymethod1
Charlotte's Object mymethod2
Charlotte's Object mymethod3
Charlotte's Object mymethod4
```

Finally we can extend classes, by including the argument super="mysuperclass" in the mclass call. We can (optionally) call the superclass constructor by calling super inside the subclass constructor.

```
> MySuperClass = function (...)
 {
         MySuperClass = function (x) .$x = x
         mymethod = function () cat (.$x, "\n")
         mclass ()
 }
> MySubClass = function (...)
 {
         MySubClass = function (y) super (y)
         mclass (super="MySuperClass")
 }
> myobj = MySubClass (10)
> myobj~mymethod ()
10
```

Support for print methods and operator overloading is currently being considered.

Technical Details

todo