S4 - Composition vs Inheritance and the package msbase

Witold Eryk Wolski

October 28, 2005

Introduction

In the package *msbase inheritance* was used to extend the `list` class functionality as opposed to the package *biobase*[6] where *composition* was used to the same purpose. I describe here the programming techniques used while implementing the functionality of the package *msbase*. Object composition and inheritance are two techniques for reusing functionality in object-oriented systems. The choice between the *composition* ("has a") relation and the *inheritance* ("is a") relation is a frequent one if we use programming languages with type checking and OOP features. It is a sensible one, because the consequences of this choice are not readily visible. We show various ways how the "is a" and "has a" relation can be expressed in `S4` [1] and explain it using the old style `R list` as parent class because there are some differences compared with extending `S4` classes.

Nomenclature

In `S4` the `contains` keyword in the `setClass` function is used to express object *inheritance*[3, 7]. It means that all methods and *slots* defined for the parent class are also valid for the inheriting subclass. Because the parent class implementation is visible to the subclasses this type of reuse is also called **white-box** reuse. Inherited classes can be assigned to slots and function arguments of the parent class type. The arguments and slots of the parent class type are *polymorphic*.

Object *composition*[3] is expressed in S4 by the keyword `representation` in `setClass`. The methods and slots of a class stored in a slot can not be accessed directly, *e.g* to access the the object it must be dereferenced. The internals of the composed objects are unknown. Because objects are treated as "black boxes," this type of reuse is often called **black-box** reuse [7].

In the sample code we label classes defined using inheritance with a leading upper `I` and classes definied using composition with a upper leading `C`. Slot names start always with a lower letter.

Motivation

The `base:list` class is one of the generic classes in `R`. It can store objects of any class and it provides a huge amount of functionality e.g `unlist`, `lapply`,

`sapply, do.call` etcetera. Here we provide examples how to specialize the `list` class. *e.g.* how to implement a list with an additional `slotA` field to store e.g. some additional information. From the OO design point of view this is clearly(?) a is a `list` with `slotA` slot[1].

The Inheritance Relation

The *inheritance* relation in S4 can be defined in various ways [1]. Either the parent class is passed as not named argument to the `representation` function or as the argument to the `contains` keyword in the class definition *e.g:*

```
> setClass("Ia", representation("list", names = "character", slotA = "character"))

[1] "Ia"

> setClass("Ia", contains = "list", representation(names = "character",
+     slotA = "character"), prototype(slotA = "content"))

[1] "Ia"

> getClass("Ia")

Slots:

Name:      .Data      names      slotA
Class:      list character character

Extends:
Class "list", from data part
Class "vector", by class "list"
```

These declarations are synonymous. I prefer the second definition because the additional keyword `contains` emphasises that inheritance is different than composition. The class definition is printed by the function `getClass`. Note that the class `Ia` has an additional slot `.Data` which was generated automatically. This slot keeps the parent class.

Normally if inheriting, all slots of the parent class are present in the child class. The R `list` is a *thingy* from pre-class times, and has no class definition[2]. Hence, an object inheriting from a `list` does not have it automatically, and if we want that the inheriting class has names, we have to define a slot `names` ourselves. By our definition the type of the names is limited to `character` which may be of advantage on the "C side" of the object.

By the `new` function a instance of a class is created. There are several ways to intitialize an object: The first is to provide all the data to the `new` function, *e.g.*

---

[1] An example how to to provide type checking of list content can be found at the S Programming Workshop page [4]

[2] To "Register an old-style (a.k.a. 'S3') class as a formally defined class" the function `setOldClass` is used. All old-style classes in the base and recomended packages are already registered.

```
> xx <- as.list(1:4)
> names(xx) <- letters[1:4]
> ea <- new("Ia", xx, slotA = "my first S4 class")
```

We can aslo assign the data afterwards using @ to access the slots. To assign an object of the `list` class to the `Ia` class we are either use the function `as`. There is a default `as` function provided to *cast* objects of the parent class into an object of the inheriting class.

```
> ia <- new("Ia")
> ia@slotA <- "my first S4 class"
> as(ia, "list") <- xx
```

An alternative way to assign the data is to access the `.Data` slot directly.

```
> ia@.Data <- xx
```

Note that the names slot was updated automatically to match the content of the `names` attribute of the `list`. This update is done by the `.mergeAttrs` function which is called every time when assigning an object to the `.Data` slot. The function `.mergeAttrs` is *exporting* all slots of the parent class.

```
> ia@names
```

```
[1] "a" "b" "c" "d"
```

Object composition

A class definition with member variables only, gives the objects of that class full control over all objects in its slots. By *composition* practically an unlimited number of different classes can be composed within the new class. It means that if calling a function with an object of this class as argument only a function with this class in the signature can be executed. In case of *inheritance* method dispatching can occur, this means that if there are no function with an argument of this class a function with an argument of the parent class is executed. The consequence is that if we like to export all of the functionality of the classes stored in member variables we must either reimplement them in the objects interface or dereference the object.

The class `Ca` is composed of two member variables of class `list` and `character`. To be able to call the function `apply` on the member variable `list` we implement the method `lapply`.

```
> library(methods)
> setClass("Ca", representation(list = "list", slotA = "character"),
+     prototype(slotA = "hello"))
```

```
[1] "Ca"
```

```
> setMethod("lapply", signature(X = "Ca"), def = function(X, FUN,
+     ...) {
+     X@list <- lapply(X@list, FUN, ...)
+     return(X)
+ })
```

```
[1] "lapply"

> ca <- new("Ca")
> ca@list <- xx
```

Note that the function definition of `lapply` must have the same signature (argument names) as the function `lapply` in the package `base` being the template for the generic. For the `Ca` class an *as list* function can be defined using `setAs`.

```
> setAs("Ca", "list", def = function(from) {
+     return(from@list)
+ }, replace = function(from, value) {
+     from@list <- value
+ })

[1] "coerce<-"
```

We see that using this design we can provide the same user interface and to this point functionality as by using inheritance.

Inherits vs. Compose

For the `Ia` class we do not need to provide an implementation of the method `apply` to be able to call it directly on the object `ea`.

```
> lapply(ca, "^", 2)

An object of class "Ca"
Slot "list":
$a
[1] 1

$b
[1] 4

$c
[1] 9

$d
[1] 16


Slot "slotA":
[1] "hello"

> lapply(ea, "^", 2)

$a
[1] 1
```

```
$b
[1] 4

$c
[1] 9

$d
[1] 16
```

Note that the first call returns an object of class `Ca` while the second one returns an object of class `list` since the method `lapply` of the parent class was executed. If we want a function `lapply` which returns an object of class `Ia` we will have equal work to do as for the class `Ia`.

The examples provided show that using the inheritance relation we need to implement less functions (no implementation of the function `lapply`, `as` are necessary). Furthermore, all the `list` functions will work for the objects of class `Ca`. The downside of this approach is that good knowledge of the methods provided by the parent classes is required.

The main reason why inheritance was introduced is to allow functions and slots to be `polymorphic` in type safe programming languages, and not just to make some lines of code superfluous. By `polymorphism` all inheriting classes can be assigned to a slot or passed to a function defined for the parent class type (polymorphic arguments slots). Composition alone is not able to provide this functionality. Let assume that we have several related classes. The classes `Ib` and `Cb` are almost the same as the classes `Ia`,`Ca` respectively, except that the *type* of `slotA` is now `numeric`.

```
> setClass("Cb", representation(list = "list", slotA = "numeric"),
+     prototype(slotA = 3))

[1] "Cb"

> cb <- new("Cb", list = xx)
> setClass("Ib", contains = "list", representation(slotA = "numeric"))

[1] "Ib"

> eb <- new("Ib", xx, slotA = 4)
```

If we like to have a function which prints the length of the `list` for all related classes we can program easily a function `printlength` with the base class in the signature and it will work for all child classes.

```
> setGeneric("printlength", function(object, ...) standardGeneric("printlength"))

[1] "printlength"

> setMethod("printlength", signature(object = "list"), def = function(object) {
+     cat("length : ", length(object), " object=", class(object),
+         "\n")
+ })
```

```
[1] "printlength"

> printlength(ea)

length :  4  object= Ia

> printlength(eb)

length :  4  object= Ib
```

It is also easily possible to assign objects of the child class to slots of parent class type.

```
> cb.tmp <- cb
> cb.tmp@list <- ea
> cb.tmp@list <- eb
> class(cb.tmp@list)

[1] "Ib"
attr(,"package")
[1] ".GlobalEnv"
```

The objects of two different classes can be easily passed to the method defined for the class `list`. For composed classes `S4` provides a different way to have a similar functionality. But they are then either less secure or limited.

Virtual Classes

setClassUnion

The "is a" relation can be defined in `S4` using the `setClassUnion`[2] method. This method defines a new `virtual` parent class (no instance can be created). For the composed classes `Ca` and `Cb` we define a virtual parent class `VC`. Then we can define a function which has in the function `signature` an argument of the virtual parent class `VC` type. To illustrate the danger inherent in this design we define a third class `Cc` which does not contain the slot *list e.g*:

```
> setClass("Cc", representation(slotA = "character"))

[1] "Cc"

> setClassUnion("VC", c("Ca", "Cb", "Cc"))

[1] "VC"

> setGeneric("printlengthC", function(object, ...) standardGeneric("printlengthC"))

[1] "printlengthC"

> setMethod("printlengthC", signature(object = "VC"), def = function(object) {
+     cat("length : ", length(object@list), "\n")
+ })

[1] "printlengthC"
```

```
> printlengthC(ca)

length :  4

> printlengthC(cb)

length :  4
```

The difference between this designs and the inheritance is that by inheritance each function defined for the parent class and working properly with it, will work for the inheriting classes. Hence, defining a class union we must ensure that all classes in the union define a common "interface" or implement appropriate checking, otherwise errors at runtime may happen e.g:

```
> cc <- new("Cc", slotA = "hello")
> print(try(printlength(cc)))

[1] "Error in printlength(cc) : no direct or inherited method for function 'printlength' f
attr(,"class")
[1] "try-error"
```

It is also possible to define a slot of the *virtual* class type and assign objects of the inherited classes to it.

```
> setClass("MC", representation(list = "VC"))

[1] "MC"

> mc <- new("MC")
> mc@list <- ca
> class(mc@list)

[1] "Ca"
attr(,"package")
[1] ".GlobalEnv"

> mc@list <- cb
> class(mc@list)

[1] "Cb"
attr(,"package")
[1] ".GlobalEnv"

> mc@list <- cc
```

setIs

When using `setClassUnion` we first must define the composed classes and then we can assigned them to an virtual class. If we already have a virtual class and like to set it as a parent class afterward we can use the `setIs` function [5, 1]. The advantage of defining an virtual class by `setClass` instead of `setClassUnion` is that we can define slots that all inheriting classes must define. In the first line of the next example we define a `virtual` class (by passing the keyword `"VIRTUAL"` to the class `representation`). Furthermore we define a slot `list` of type `list`.

```
> setClass("VC2", representation(list = "list"), contains = "VIRTUAL")

[1] "VC2"

> print(try(new("VC2")))

[1] "Error in new(\"VC2\") : trying to use new() on a virtual class\n"
attr(,"class")
[1] "try-error"

> setIs("Ca", "VC2")
> print(try(setIs("Cc", "VC2")))

[1] "Error in .validExtends(class1, class2, classDef, classDef2, obj@simple) : \n\tclass \
attr(,"class")
[1] "try-error"

> setClass("MC", representation(list = "VC2"))

[1] "MC"

> mc <- new("MC")
> mc@list <- ca
> class(mc@list)

[1] "Ca"
attr(,"package")
[1] ".GlobalEnv"
```

setIs and "normal" classes

To the *is* function defined by `setIs` a `coerce` and `replace` function can be
provided. These functions are executed if an object is assigned to a slot or
a function argument or a slot of the parent class type. `setIs` is still a little
buggy when working with old style classes in the current R2.0.0 release. It is
not enough to define just the *is* (`setIs`) relation but we must also define the *as*
(`setAs`) relation.

```
> setIs("Ca", "list", coerce = function(obj) {
+     return(obj@list)
+ }, replace = function(obj, value) {
+     obj@list <- value
+ })
> setIs("Cb", "list", coerce = function(obj) {
+     return(obj@list)
+ }, replace = function(obj, value) {
+     obj@list <- value
+ })
> setAs("Cb", "list", def = function(from) {
+     return(from@list)
+ }, replace = function(from, value) {
+     from@list <- value
+ })
```

```
[1] "coerce<-"

> setIs("Cc", "list", coerce = function(obj) {
+     return(vector("list", 0))
+ }, replace = function(obj, value) {
+ })
> setAs("Cc", "list", def = function(from) {
+     return(vector("list", 0))
+ }, replace = function(from, value) {
+ })

[1] "coerce<-"

> printlength(cb)

length :  4  object= list

> printlength(ca)

length :  4  object= list

> printlength(cc)

length :  0  object= list
```

By providing a proper definition of the functions `coerce` and `replace` using `setIs` we can ensure for classes that do not have a `list` slot that a valid object of the *base* class is returned. The limitation of `setIs` is that we are not able to assign to a slot of the parent class type an object of the child class type. At the time of assignment the inheriting class is cast into the parent class.

```
> cb.tmp@list <- cb
> class(cb)

[1] "Cb"
attr(,"package")
[1] ".GlobalEnv"
```

Memory

With respect to memory the extends relation seems to be a little more efficient.

```
> object.size(ea)

[1] 840

> object.size(ca)

[1] 908
```

Conclusion

By composition we are able to provide a similar user interface as with the extends relation. Hence, the only serious reason for using inheritance is to express the "is a" relation to be able to assign objects of different but similar classes to a slot or argument of the parent class type. In addition to inheritance defined in the `setClass` function by the keyword `contains`, we can use class unions or the `setIs` function in `S4`. The use of virtual classes defined via `setClassUnion` implies the danger of runtime errors and requires programming discipline. A safer way to define the "is a" relation is to use `setClass` to define *virtual* classes with slots and `setIs`. Using the `coerce` and `replace` function defined in `setIs` we can mimic some aspects of object inheritance. Many examples how to combine inheritance and composition to obtain a good design can be found in E. Gammas et al. "Design Patterns" [3].

1 Aknowledgments

Thanks to Mathias Kohl for valuable disussion.

References

[1] John M. Chambers. *Programming with Data*. Springer, New York, 1998. ISBN 0-387-98503-4.

[2] John M. Chambers. Documentation for package 'methods' version 2.0.0, 2004.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In O. M. Nierstrasz, editor, *ECOOP'93: Object-Oriented Programming - Proc. of the 7th European Conference*, pages 406–431. Springer, Berlin, Heidelberg, 1993.

[4] R. Gentleman. Container.r, February 2003.

[5] Robert Gentleman. S4 classes in 15 pages, more or less, 2003.

[6] Robert C Gentleman, Vincent J. Carey, Douglas M. Bates, Ben Bolstad, Marcel Dettling, Sandrine Dudoit, Byron Ellis, Laurent Gautier, Yongchao Ge, Jeff Gentry, Kurt Hornik, Torsten Hothorn, Wolfgang Huber, Stefano Iacus, Rafael Irizarry, Friedrich Leisch Cheng Li, Martin Maechler, Anthony J. Rossini, Gunther Sawitzki, Colin Smith, Gordon Smyth, Luke Tierney, Jean Y. H. Yang, and Jianhua Zhang. Bioconductor: Open software development for computational biology and bioinformatics. *Genome Biology*, 5:R80, 2004.

[7] Paul J. Rajlich. An object oriented approach to developing visualization tools portable across desktop and virtual environments. Master's thesis, University of Illinois at Urbana-Champaign, 1998.