Converting Packages to S4

by Douglas Bates

Introduction

R now has two systems of classes and methods, known informally as the 'S3' and 'S4' systems. Both systems are based on the assignment of a *class* to an object and the use of *generic functions* that invoke different *methods* according to the class of their arguments. Classes organize the representation of information and methods organize the actions that are applied to these representations.

'S3' classes and methods for the S language were introduced in Chambers and Hastie (1992), (see also Venables and Ripley, 2000, ch. 4) and have been implemented in R from its earliest public versions. Because many widely-used R functions, such as `print`, `plot` and `summary`, are S3 generics, anyone using R inevitably (although perhaps unknowingly) uses the S3 system of classes and methods.

Authors of R packages can take advantage of the S3 class system by assigning a class to the objects created in their package and defining methods for this class and either their own generic functions or generic functions defined elsewhere, especially those in the `base` package. Many R packages do exactly this. To get an idea of how many S3 classes are used in R, attach the packages that you commonly use and call `methods("print")`. The result is a vector of names of methods for the `print` generic. It will probably list dozens, if not hundreds, of methods. (Even that list may be incomplete because recent versions of some popular packages use namespaces to hide some of the S3 methods defined in the package.)

The S3 system of classes and methods is both popular and successful. However, some aspects of its design are at best inconvenient and at worst dangerous. To address these inadequacies John Chambers introduced what is now called the 'S4' system of classes and methods (Chambers, 1998; Venables and Ripley, 2000). John implemented this system for R in the `methods` package which, beginning with the 1.7.0 release of R, is one of the packages that are attached by default in each R session.

There are many more packages that use S3 classes and methods than use S4. Probably the greatest use of S4 at present is in packages from the Bioconductor project (Gentleman and Carey, 2002). I hope by this article to encourage package authors to make greater use of S4.

Conversion from S3 to S4 is not automatic. A package author must perform this conversion manually and the conversion can require a considerable amount of effort. I speak from experience. Saikat DebRoy and I have been redesigning the linear mixed-effects models part of the `nlme` package, including conversion to S4, and I can attest that it has been a lot of work. The purpose of this article is to indicate what is gained by conversion to S4 and to describe some of our experiences in doing so.

S3 versus S4 classes and methods

S3 classes are informal: the class of an object is determined by its class attribute, which should consist of one or more character strings, and methods are found by combining the name of the generic function with the class of the first argument to the function. If a function having this combined name is on the search path, it is assumed to be the appropriate method. Classes and their contents are not formally defined in the S3 system - at best there is a "gentleman's agreement" that objects in a class will have certain structure with certain component names.

The informality of S3 classes and methods is convenient but dangerous. There are obvious dangers in that any R object can be assigned a class, say `"foo"`, without any attempt to validate the names and types of components in the object. That is, there is no guarantee that an object that claims to have class `"foo"` is compatible with methods for that class. Also, a method is recognized solely by its name so a function named `print.foo` is assumed to be the method for the `print` generic applied to an object of class `foo`. This can lead to surprising and unpleasant errors for the unwary.

Another disadvantage of using function names to identify methods is that the class of only one argument, the first argument, is used to determine the method that the generic will use. Often when creating a plot or when fitting a statistical model we want to examine the class of more than one argument to determine the appropriate method, but S3 does not allow this.

There are more subtle disadvantages to the S3 system. Often it is convenient to describe a class as being a special case of another class; for example, a model fit by `aov` is a special case of a linear model (class `lm`). We say that class `aov` inherits from class `lm`. In the informal S3 system this inheritance is described by assigning the class `c("aov","lm")` to an object fit by `aov`. Thus the inheritance of classes becomes a property of the object, not a property of the class, and there is no guarantee that it will be consistent across all members of the class. Indeed there were examples in some packages where the class inheritance was not consistently assigned.

By contrast, S4 classes must be defined explicitly. The number of slots in objects of the class, and the names and classes of the slots, are established at the time of class definition. When an object of the class

 ISSN 1609-3631

is created, and at some points during computation with the object, it is validated against the definition. Inheritance of classes is also specified at the time of class definition and thus becomes a property of the class, not a (possibly inconsistent) property of objects in the class.

S4 also requires formal declarations of methods, unlike the informal system of using function names to identify a method in S3. An S4 method is declared by a call to `setMethod` giving the name of the generic and the "signature" of the arguments. The signature identifies the classes of one or more named arguments to the generic function. Special meta-classes named `ANY` and `missing` can be used in the signature.

S4 generic functions are automatically created when a method is declared for an existing function, in which case the function becomes generic and the current definition becomes the default method. A new generic function can be declared explicitly by a call to `setGeneric`. When a method for the generic is declared with `setMethod` the number, names, and order of its arguments are matched against those of the generic. (If the generic has a `...` argument, the method can add new arguments but it can never omit or rename arguments of the generic.)

In summary, the S4 system is much more formal regarding classes, generics, and methods than is the S3 system. This formality means that more care must be taken in the design of classes and methods for S4. In return, S4 provides greater security, a more well-defined organization, and, in my opinion, a cleaner structure to the package.

Package conversion

Chambers (1998, ch. 7,8) and Venables and Ripley (2000, ch. 5) discuss creating new packages based on S4. Here I will concentrate on converting an existing package from S3 to S4.

S4 requires formal definitions of generics, classes, and methods. The generic functions are usually the easiest to convert. If the generic is defined externally to the package then the package author can simply begin defining methods for the generic, taking care to ensure that the argument sequences of the method are compatible with those of the generic. As described above, assigning an S4 method for, say, `coef` automatically creates an S4 generic for `coef` with the current externally-defined `coef` function as the default method. If an S3 generic was defined internally to the package then it is easy to convert it to an S4 generic.

Converting classes is less easy. We found that we could use the informal set of classes from the S3 version as a guide when formulating the S4 classes but we frequently reconsidered the structure of the classes during the conversion. We frequently found ourselves adding slots during the conversion so we had more slots in the S4 classes than components in the S3 objects.

Increasing the number of slots may be an inevitable consequence of revising the package (we tend to add capabilities more frequently than we remove them) but it may also be related to the fact that S4 classes must be declared explicitly and hence we must consider the components or slots of the classes and the relationships between the classes more carefully.

Another reason that we incorporating more slots in the S4 classes than in the S3 prototype is because we found it convenient that the contents of slots in S4 objects can easily be accessed and modified in C code called through the `.Call` interface. Several C macros for working with S4 classed objects, including `GET_SLOT`, `SET_SLOT`, `MAKE_CLASS` and `NEW` (all described in Chambers (1998)) are available in R. The combination of the formal classes of S4, the `.Call` interface, and these macros allows a programmer to manipulate S4 classed objects in C code nearly as easily as in R code. Furthermore, when the C code is called from a method, the programmer can be confident of the classes of the objects passed in the call and the classes of the slots of those objects. Much of the checking of classes or modes and possible coercion of modes that is common in C code called from R can be bypassed.

We found that we would initially write methods in R then translate them into C if warranted. The nature of our calculations, frequently involving multiple decompositions and manipulations of sections of arrays, was such that the calculations could be expressed in R but not very cleanly. Once we had the R version working satisfactorily we could translate into C the parts that were critical for performance or were awkward to write in R. An important advantage of this mode of development is that we could use the same slots in the C version as in the R version and create the same types of objects to be returned.

We feel that defining S4 classes and methods in R then translating parts of method definitions to C functions called through `.Call` is an extremely effective mode for numerical computation. Programmers who have experience working in C++ or Java may initially find it more convenient to define classes and methods in the compiled language and perhaps define a parallel series of classes in R. (We did exactly that when creating the Matrix package for R.) We encourage such programmers to try instead this method of defining only one set of classes, the S4 classes in R, and use these classes in both the interpreted language and the compiled language.

The definition of S4 methods is more formal than in S3 but also more flexible because of the ability to match an argument signature rather than being contrained to matching just the class of the first argument. We found that we used this extensively when defining methods for generics that fit models. We

would define the "canonical" form of the arguments, which frequently was a rather wordy form, and one "collector" method that matched this form of the arguments. The collector method is the one that actually does the work, such as fitting the model. All the other methods are designed to take more conveniently expressed forms of the arguments and rearrange them into the canonical form. Our subjective impression is that the resulting methods are much easier to understand than those in the S3 version.

Pitfalls

S4 classes and methods are powerful tools. With these tools a programmer can exercise fine-grained control over the definition of classes and methods. This encourages a programming style where many specialized classes and methods are defined. One of the difficulties that authors of packages then face is documenting all these classes, generics, and methods.

We expect that generic functions and classes will be described in detail in the documentation, usually in a separate documentation file for each class and each generic function, although in some cases closely related classes could be described in a single file. It is less obvious whether, how, and where to document methods. In the S4 system methods are associated with a generic function and an argument signature. It is not clear if a given method should be documented separately or in conjunction with the generic function or with a class definition.

Any of these places could make sense for some methods. All of them have disadvantages. If all methods are documented separately there will be an explosion of the number of documentation files to create and maintain. That is an unwelcome burden. Documenting methods in conjunction with the generic can work for internally defined generics but not for those defined externally to the package. Documenting methods in conjunction with a class only works well if the method signature consists of one argument only but part of the power of S4 is the ability to dispatch on the signature of multiple arguments.

Others working with S4 packages, including the Bioconductor contributors, are faced with this problem of organizing documentation. Gordon Smyth and Vince Carey have suggested some strategies for organizing documentation but more experience is definitely needed.

One problem with creating many small methods that rearrange arguments and then call `callGeneric()` to get eventually to some collector method is that the sequence of calls has to be unwound when returning the value of the call. In the case of a model-fitting generic it is customary to preserve the original call in a slot or component named `call` so that it can be displayed in summaries and also so it can be used to `update` the fitted model. To preserve the call, each of the small methods that just manipulate the arguments must take the result of `callGeneric`, reassign the `call` slot and return this object. We discovered, to our chagrin, that this caused the entire object, which can be very large in some of our examples, to be copied in its entirety as each method call was unwound.

Conclusions

Although the formal structure of the S4 system of classes and methods requires greater discipline by the programmer than does the informal S3 system, the resulting clarity and security of the code makes S4 worthwhile. Moreover, the ability in S4 to work with a single class structure in R code and in C code to be called by R is a big win.

S4 encourages creating many related classes and many methods for generics. Presently this creates difficult decisions on how to organize documentation and how unwind nested method calls without unwanted copying of large objects. However it is still early days with regard to the construction of large packages based on S4 and as more experience is gained we will expect that knowledge of the best practices will be disseminated in the community so we can all benefit from the S4 system.

Bibliography

J. M. Chambers. *Programming with Data*. Springer, New York, 1998. ISBN 0-387-98503-4. 6, 7

J. M. Chambers and T. J. Hastie. *Statistical Models in S*. Chapman & Hall, London, 1992. ISBN 0-412-83040-X. 6

R. Gentleman and V. Carey. Bioconductor. *R News*, 2(1):11–16, March 2002. URL http://CRAN.R-project.org/doc/Rnews/. 6

W. N. Venables and B. D. Ripley. *S Programming*. Springer, 2000. URL http://www.stats.ox.ac.uk/pub/MASS3/Sprog/. ISBN 0-387-98966-8. 6, 7

Douglas Bates
University of Wisconsin–Madison, U.S.A.
Bates@stat.wisc.edu