Name Space Management for R

Luke Tierney

Introduction

In recent years R has become a major platform for the development of new data analysis software. As the complexity of software developed in R and the number of available packages that might be used in conjunction increase, the potential for conflicts among definitions increases as well. The name space mechanism added in R 1.7.0 provides a framework for addressing this issue.

Name spaces allow package authors to control which definitions provided by a package are visible to a package user and which ones are private and only available for internal use. By default, definitions are private; a definition is made public by *exporting* the name of the defined variable. This allows package authors to create their main functions as compositions of smaller utility functions that can be independently developed and tested. Only the main functions are exported; the utility functions used in the implementation remain private. An incremental development strategy like this is often recommended for high level languages like R—a guideline often used is that a function that is too large to fit into a single editor window can probably be broken down into smaller units. This strategy has been hard to follow in developing R packages since it would have lead to packages containing many utility functions that complicate the user's view of the main functionality of a package, and that may conflict with definitions in other packages.

Name spaces also give the package author explicit control over the definitions of global variables used in package code. For example, a package that defines a function `mydnorm` as

```
mydnorm <- function(z)
    1/sqrt(2 * pi) * exp(- z^2 / 2)
```

most likely intends `exp`, `sqrt`, and `pi` to refer to the definitions provided by the **base** package. However, standard packages define their functions in the global environment shown in Figure 1.

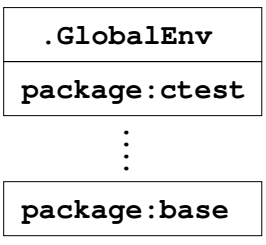

Figure 1: Dynamic global environment.

The global environment is dynamic in the sense that top level assignments and calls to `library` and `attach` can insert shadowing definitions ahead of the definitions in **base**. If the assignment `pi <- 3` has been made at top level or in an attached package, then the value returned by `mydnorm(0)` is not likely to be the value the author intended. Name spaces ensure that references to global variables in the **base** package cannot be shadowed by definitions in the dynamic global environment.

Some packages also build on functionality provided by other packages. For example, the package **modreg** uses the function `as.stepfun` from the **stepfun** package. The traditional approach for dealing with this is to use calls to `require` or `library` to add the additional packages to the search path. This has two disadvantages. First, it is again subject to shadowing. Second, the fact that the implementation of a package **A** uses package **B** does not mean that users who add **A** to their search path are interested in having **B** on their search path as well. Name spaces provide a mechanism for specifying package dependencies by *importing* exported variables from other packages. Formally importing variables form other packages again ensures that these cannot be shadowed by definitions in the dynamic global environment.

From a user's point of view, packages with a name space are much like any other package. A call to `library` is used to load the package and place its exported variables in the search path. If the package imports definitions from other packages, then these packages will be loaded but they will not be placed on the search path as a result of this load.

Adding a name space to a package

A package is given a name space by placing a 'NAMESPACE' file at the top level of the package source directory. This file contains *name space directives* that define the name space. This declarative approach to specifying name space information allows tools that collect package information to extract the package interface and dependencies without processing the package source code. Using a separate file also has the benefit of making it easy to add a name space to and remove a name space from a package.

The main name space directives are export and import directives. The `export` directive specifies names of variables to export. For example, the directive

```
export(as.stepfun, ecdf, is.stepfun, stepfun)
```

exports four variables. Quoting is optional for standard names such as these, but non-standard names need to be quoted, as in

```
export("[<-.tracker")
```

As a convenience, exports can also be specified with a pattern. The directive

```
exportPattern("\\.test$")
```

exports all variables ending with `.test`.

Import directives are used to import definitions from other packages with name spaces. The directive

```
import(mva)
```

imports all exported variables from the package **mva**. The directive

```
importFrom(stepfun, as.stepfun)
```

imports only the `as.stepfun` function from the **stepfun** package.

Two additional directives are available. The `S3method` directive is used to register methods for S3 generic functions; this directive is discussed further in the following section. The final directive is `useDynLib`. This directive allows a package to declaratively specify that a DLL is needed and should be loaded when the package is loaded.

For a package with a name space the two operations of loading the package and attaching the package to the search path are logically distinct. A package loaded by a direct call to `library` will first be loaded, if it is not already loaded, and then be attached to the search path. A package loaded to satisfy an import dependency will be loaded, but will *not* be attached to the search path. As a result, the single initialization hook function `.First.lib` is no longer adequate and is not used by packages with name spaces. Instead, a package with a name space can provide two hook functions, `.onLoad` and `.onAttach`. These functions, if defined, should not be exported. Many packages will not need either hook function, since import directives take the place of `require` calls and `useDynLib` directives can replace direct calls to `library.dynam`.

Name spaces are *sealed*. This means that , once a package with a name space is loaded, it is not possible to add or remove variables or to change the values of variables defined in a name space. Sealing serves two purposes: it simplifies the implementation, and it ensures that the locations of variable bindings cannot be changed at run time. This will facilitate the development of a byte code compiler for R (Tierney, 2001).

Adding a name space to a package may complicate debugging package code. The function `fix`, for example, is no longer useful for modifying an internal package function, since it places the new definition in the global environment and that new definition remains shadowed within the package by the original definition. As a result, it is a good idea not to add a name space to a package until it is completely debugged, and to remove the name space if further debugging is needed; this can be done by temporarily renaming the 'NAMESPACE' file. Other alternatives are being explored, and R 1.7.0 contains some experimental functions, such as `fixInNamespace`, that may help.

Name spaces and method dispatch

R supports two styles of object-oriented programming: the S3 style based on `UseMethod` dispatch, and the S4 style provided by the **methods** package. S3 style dispatch is still used the most and some support is provided in the name space implementation released in R 1.7.0.

S3 dispatch is based on concatenating the generic function name and the name of a class to form the name of a method. The environment in which the generic function is called is then searched for a method definition. With name spaces this creates a problem: if a package is imported but not attached, then the method definitions it provides may not be visible at the call site. The solution provided by the name space system is a mechanism for registering S3 methods with the generic function. A directive of the form

```
S3method(print, dendrogram)
```

in the 'NAMESPACE' file of a package registers the function `print.dendrogram` defined in the package as the `print` method for class `dendrogram`. The variable `print.dendrogram` does not need to be exported. Keeping the method definition private ensures that users will not inadvertently call the method directly.

S4 classes and generic functions are by design more formally structured than their S3 counterparts and should therefore be conceptually easier to integrate with name spaces than S3 generic functions and classes. For example, it should be fairly easy to allow for both exported and private S4 classes; the concatenation-based approach of S3 dispatch does not seem to be compatible with having private classes except by some sort of naming convention. However, the integration of name spaces with S4 dispatch could not be completed in time for the release of R 1.7.0. It is quite likely that the integration can be accomplished by adding only two new directives, `exportClass` and `importClassFrom`.

A simple example

A simple artificial example may help to illustrate how the import and export directives work. Consider two packages named **foo** and **bar**. The R code for package **foo** in file 'foo.R' is

```
x <- 1
f <- function(y) c(x,y)
```

The 'NAMESPACE' file contains the single directive

```
export(f)
```

Thus the variable x is private to the package and the function f is public. The body of f references a global variable x and a global function c. The global refrence x corresponds to the definition in the package. Since the package does not provide a definition for c and does not import any definitions, the variable c refers to the definition provided by the **base** package.

The second package **bar** has code file 'bar.R' containing the definitions

```
c <- function(...) sum(...)
g <- function(y) f(c(y, 7))
```

and 'NAMESPACE' file

```
import(foo)
export(g)
```

The function c is private to the package. The function g calls a global function c. Definitions provided in the package take precedence over imports and definitions in the **base** package; therefore the definition of c used by g is the one provided in **bar**.

Calling `library(bar)` loads **bar** and attaches its exports to the search path. Package **foo** is also loaded but not attached to the search path. A call to g produces

```
> g(6)
[1]  1 13
```

This is consistent with the definitions of c in the two settings: in **bar** the function c is defined to be equivalent to sum, but in **foo** the variable c refers to the standard function c in **base**.

Some details

This section provides some details on the current implementation of name spaces. These details may change as name space support evolves.

Name spaces use R environments to provide static binding of global variables for function definitions. In a package with a name space, functions defined in the package are defined in a name space environment as shown in Figure 2. This environment consists of a set of three static frames followed by the usual dynamic global environment. The first static frame contains the local definitions of the package. The second frame contains

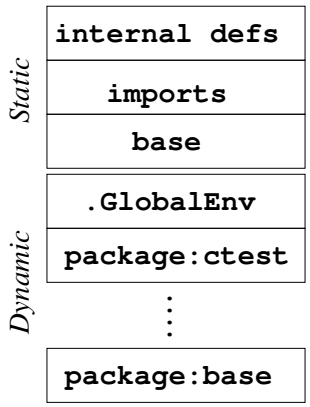

Figure 2: Name space environment.

imported definitions, and the third frame is the **base** package. Suppose the function mydnorm shown in the introduction is defined in a package with a name space that does not explicitly import any definitions. Then a call to mydnorm will cause R to evaluate the variable pi by searching for a binding in the name space environment. Unless the package itself defines a variable pi, the first binding found will be the one in the third frame, the definition provided in the **base** package. Definitions in the dynamic global environment cannot shadow the definition in **base**.

When library is called to load a package with a name space, library first calls loadNamespace and then attachNamespace. loadNamespace first checks whether the name space is already loaded and registered with the internal registry of loaded name spaces. If so, the already loaded name space is returned. If not, then loadNamespace is called on all imported name spaces, and definitions of exported variables of these packages are copied to the imports frame, the second frame, of the new name space. Then the package code is loaded and run, and exports, S3 methods, and shared libraries are registered. Finally, the .onLoad hook is run, if the package defines one, and the name space is sealed. Figure 3 illustrates the dynamic global environment and the internal data base of loaded name spaces after two packages, **A** and **B**, have been loaded by explicit calls to library and package **C** has been loaded to satisfy import directives in package **B**.

The serialization mechanism used to save R workspaces handles name spaces by writing out a reference for a name space consisting of the package name and version. When the workspace is loaded, the reference is used to construct a call to loadNamespace. The current implementation ignores the version information; in the future this may be used to signal compatibility warnings or select among several available versions.

The registry of loaded name spaces can be examined using loadedNamespaces. In the current implementation loaded name spaces are not unloaded automatically. Detaching a package with a

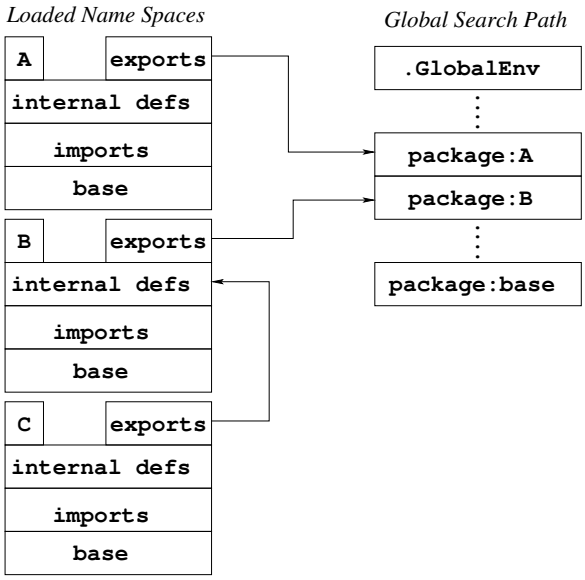

Figure 3: Internal implementation.

name space loaded by a call to `library` removes the frame containing the exported variables of the package from the global search path, but it does not unload the name space. Future implementations may automatically unload name spaces that are no longer needed to satisfy import dependencies. The function `unloadNamespace` can be used to force unloading of a name space; this can be useful if a new version of the package is to be loaded during development.

Variables exported by a package with a name space can also be referenced using a fully qualified variable reference consisting of the package name and the variable name separated by a double colon, such as `foo::f`. This can be useful in debugging at the top level. It can also occasionally be useful in source code, for example in cases where functionality is only to be used if the package providing it is available. However, dependencies resulting from use of fully qualified references are not visible from the name space directives and therefore may cause some confusion, so explicit importing is usually preferable. Computing the value of a fully qualified variable reference will cause the specified package to be loaded if it is not loaded already.

Discussion

Name spaces provide a means of making R packages more modular. Many languages and systems originally developed without name space or module systems have had some form of name space management added in order to support the development of larger software projects. C++ and Tcl (Welch, 1999) are two examples. Other languages, such as Ada (Barnes, 1998) and Modula-3 (Harbison, 1992) were designed from the start to include powerful module systems. These languages use a range of approaches for specifying name space or module information. The declarative approach with separation of implementation code and name space information used in the system included in R 1.7.0 is closest in spirit to the approaches of Modula-3 and Ada. In contrast, Tcl and C++ interleave name space specification and source code.

The current R name space implementation, while already very functional and useful, is still experimental and incomplete. Support for S4 methods still needs to be included. More exhaustive error checking is needed, along with code analysis tools to aid in constructing name space specifications. To aid in package development and debugging it would also be useful to explore whether strict sealing can be relaxed somewhat without complicating the implementation. Early experiments suggest that this may be possible.

Some languages, including Ada, ML (Ullman, 1997) and the MzScheme Scheme implementation (PLT) provide a richer module structure in which parameterized partial modules can be assembled into complete modules. In the context of R this might correspond to having a parameterized generic data base access package that is completed by providing a specific data base interface. Whether this additional power is useful and the associated additional complexity is warranted in R is not yet clear.

Bibliography

John Barnes. *Programming In Ada 95*. Addison-Wesley, 2nd edition, 1998. 5

Samuel P. Harbison. *Modula-3*. Prentice Hall, 1992. 5

PLT. PLT MzScheme. World Wide Web, 2003. URL http://www.plt-scheme.org/software/mzscheme/. 5

Luke Tierney. Compiling R: A preliminary report. In Kurt Hornik and Friedrich Leisch, editors, *Proceedings of the 2nd International Workshop on Distributed Statistical Computing, March 15-17, 2001, Technische Universität Wien, Vienna, Austria*, 2001. URL http://www.ci.tuwien.ac.at/Conferences/DSC-2001/Proceedings/. ISSN 1609-395X. 3

Jeffrey Ullman. *Elements of ML Programming*. Prentice Hall, 1997. 5

Brent Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 1999. 5

Luke Tierney
Department of Statistics & Actuarial Science
University of Iowa, Iowa City, Iowa, U.S.A
luke@stat.uiowa.edu

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