Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)
March 20–22, Vienna, Austria *ISSN 1609-395X*
Kurt Hornik, Friedrich Leisch & Achim Zeileis (eds.)
http://www.ci.tuwien.ac.at/Conferences/DSC-2003/

The R.oo package – Object-Oriented Programming with References Using Standard R Code

Henrik Bengtsson

Abstract

An easy to install and platform independent package named R.oo, which provides support for references and mutable objects via a specific class model using standard R code, has been developed. The root class Object implements and encapsulates all the mechanisms needed for references in a way such that object fields are accessed similarly to how elements of a list are accessed with the important difference that the fields can be reassigned within methods. The class model also provides an easy way for defining classes that inherit directly or indirectly from the Object class. Any instance of a class that inherits from the Object class can be passed to functions by reference. Supplementary utility functions for defining constructors and methods in a simple and robust way are also made available. For instance, generic functions are created automatically and if non-generic functions with the same name already exist, they are, if possible, modified to become default functions. Currently, S3 classes and S3 methods are defined, but future versions of the package are likely to support S4 too. We also suggest an R coding convention, which the utility functions test against, with the intention to bring additional structure to the source code. The package also extends the current exception handling mechanism in R such that exception objects can be caught based on their class. The R.oo package has successfully been used in a medium-size microarray project.

Keywords: Object-oriented programming, reference variables, mutable objects, environments, coding conventions, exception handling, root class, generic functions, S3, UseMethod.

1 Introduction

The R.oo package is a spin-off product of a larger project for developing a microarray analysis package in R (com.braju.sma; Bengtsson, 2002a), which was designed

to be used by both statisticians who want to develop and test new methods, but also by other scientists who want to make use of the latest microarray analysis methods available on a daily bases. Requirements such as scalable and maintainable code, efficient memory usage, but also a user-friendly environment led us to an object-oriented design and implementation that made use of references. Moreover, to make it possible for also non-programmers to easily contribute to the project in a robust and sustainable way, we decided upon a common coding convention. A well defined coding convention would then also work as a programming guide for newcomers to the project. The R.oo package provides a standardized way of programming with reference via a specific class model, which also allows the user to define new classes easily. The package also has a mechanism for defining new methods, a mechanism that automatically asserts that the coding convention is followed.

R allows itself to two different styles of *programming with classes*. We will refer to the first (and the intended) style, which is found in languages such as S and Dylan (Shalit, Moon, and Starbuck, 1996), as the function-object-oriented programming (FOOP) style, and the second one, which is found in languages such as Java and C++, as the class-object-oriented programming (COOP) style. Shortly, the nature of the first style is that *methods belong to generic functions* (Chambers, 1998) and the nature of the second style is that *methods belong to classes*. This difference has implications on how classes and methods are designed and implemented, but also technical details such as how methods are dispatched. When methods belong to classes dispatching is done on *one* object only, whereas when methods belong to generic functions dispatching can be done on *multiple* objects. The latter is natural when, for instance, binary operators such as the "+" method are to be called. For another discussion about the two styles in R see Chambers (2002a). We have found that the COOP style is more useful for high-level designs where large and complex classes such as microarray data are used and that the FOOP style is more suitable for less complex data types such as matrices etc. to which we for instance apply algebraic operators. Moreover, even though the intension is to use the FOOP style, it is not rare to see hybrid models where it is clear that the programmer thinks of the methods such that some belong to classes and some belong to generic functions.

In R there are currently two implementations for programming with classes. They are referred to as the S3 (or S3/UseMethod) style (Venables and Ripley, 1999) and the S4 style (Chambers, 1998). The S3 style has been supported by R from the very beginning and support for the S4 style was added with the release of the methods package (Chambers, 2002b), which became part of the core distribution as of R v1.4.0 and is loaded by default from R v1.7.0. The intention of both S3 and S4 is to follow the FOOP style. Although it is possible to call methods based on multiple arguments in S3 (see for instance `help(.Group)`), the dominating approach is to dispatch methods based on the class of the first argument passed to the generic function. This has opened a door for programming using the COOP style in R. Even if method definitions in practice are more bound to the corresponding generic functions in S4 than S3, it is still possible to use S4 for the COOP style.

With references it is possible not only to write more memory efficient code, but also more user-friendly methods as for instance fewer arguments need to be specified by the end user. Even though references are often mentioned in relation to the COOP style, they can equally well be used under the FOOP style. There is also

nothing in the S3 schema or the S4 schema that prevents the idea of using references. However, R *is* a functional language and arguments are supplied to functions by a *pass-by-value* semantic, also known as call-by-value. Inside the function, the arguments behave like local variables and any change made to a value inside the function is not reflected in the original value. The rational for this is that a function by definition takes one or several input values, returns something else and it should never modify the input values. When passing large data objects to a function a true pass-by-value approach would be too inefficient. To overcome this, an argument in R is in practice *passed by reference* as long as the variable is not modified by the function. As soon as the function is changing the value of the argument, a local copy of it is created to obtain *pass by value*. Occasionally there are requests for adding a true *pass-by-reference* semantic so that functions *can* modify the original object. From now on we refer to such functions as *methods* and reserve the word *function* for its original meaning.

There are different ways to emulate a *pass-by-reference* semantic in R. An intuitive way is to make use of *lexical scoping* (Gentleman and Ihaka, 2000, see also `demo(scoping)`). However, with such an approach each object carries a *copy* of every method belonging to its class and all its superclasses. This *methods-belong-to-objects* style is a very special case of the COOP style, which we do not find to be a reasonable approach in real-world applications where class hierarchies with a large number of methods are commonly used. Another approach is to let regular R variables represent reference variables and pass these, and not the objects themselves, to the methods. Using standard language features, e.g. *environments*, references can be used to look up the actual instances, which then can be queried and modified from anywhere. This idea works perfectly well under the FOOP and the COOP style (and therefore also under both S3 and S4).
A package that provides references is the Omegahat OOP package (Chambers and Lang, 2001, 2002), which provides programming with classes in an explicit COOP style. Like R.oo, it makes use of environments to emulate reference and is (currently) based on an S3 class model, but it has its own built-in method-dispatching mechanism and a more formal way of defining classes, similar to the one in S4.

When our microarray project started in early 2001 we investigation different options for programming with references and we found Omegahat's OOP to be very promising. However, it was in its early stages, we did not know where it was heading or even if it was going to be supported in the long run. Also, OOP did not and does not meet our criteria that our microarray package should be straightforward to install and update[1]. Moreover, although an explicit COOP-style syntax for calling methods (see also section 3.3) has several advantages, we have split feelings about it and we are not convinced that a separate implementation for it is desirable. Instead, we exclusively want to make use of the already existing S3 and S4 mechanisms. These are widely used and tested by many, guaranteeing reliability and high quality. This makes the R.oo package (and the microarray package) as portable, as maintainable, and as compatible with R (also with future versions) as possible. Further, contrary to OOP, our object model enforces the use of a common

[1]To install OOP on non-Unix systems C-code compilation is required, which in turn requires compilers etc. and this is something we can not expect our microarray end-users to have installed on their machines. This can of course be solved by providing binaries for different platforms, but this is currently not done.

root class, which we believe improves the structure of any object-oriented design.

For reasons like the above, but also others, the R.oo package was developed. We want to underline that the intention is not to replace S3 or S4, but to extend them with an extra layer to provide reference variables and to make object-oriented design and programming in R easier and more robust. The outline of this report is as follows. In section 2, we briefly introduce the special "data type" *Object*. As this class implements the functionality for references, any object of a class that inherits from this *root class* will be passed to functions as a reference. In section 3, we describe how to use classes that inherits from the root class Object. Additional object-oriented features that come with proposed class model are also explained. To further improve the structure of an object-oriented implementation, we suggest an R Coding Convention (RCC) in section 4. Another purpose of the package is to relieve the developer from implementation details to make it possible to focus on the object-oriented design. In section 5, the utility functions `setConstructorS3()` and `setMethodS3()` are introduced. They create constructor functions and methods and make sure that required generic functions are created (or not), and at the same time assert that the RCC is followed. How to use references and how they are implemented by the Object class is discussed in section 6. The R.oo package also provides an extended exception handling mechanism, which is described in section 7. Additional utility functions and classes are briefly described in section 8 and section 9, respectively. Limitations of the package and its class model are commented on in section 10. Section 11 explains how to install the package. Conclusions are given in section 12.

2 The root class Object

The most fundamental class in the R.oo package is the class named *Object*. In addition to providing references, it also provides a way to define new classes. By enforcing that all classes are derived (directly or indirectly) from the Object class, we know that there exists a set of methods that are common to all such classes. This idea exists to some extent in R, but with a common root class it will become more explicit to the end user. We believe that having a single root class will bring additional structure to the design as well as the code of any software. Next we will give a short description of all methods coupled to the Object class. See also figure 1. For simplicity, we refer to an instance of the Object class or a class that inherits from it as *an Object*. Moreover, where it is clear we will when referring to methods exclude the first argument, which is always an Object.

The method ***as.character()*** returns a string with short information about an Object. This is the same string that by default is displayed by *print()*.

The ***print()*** method prints information about an Object. By default the string returned by *as.character()* is printed.[2]

[2]Regardless of data type of class, the *print()* method will be called on *any* object whose name is typed at the command line followed by an ENTER, e.g. `3+1` + ENTER calls `print(4)`.

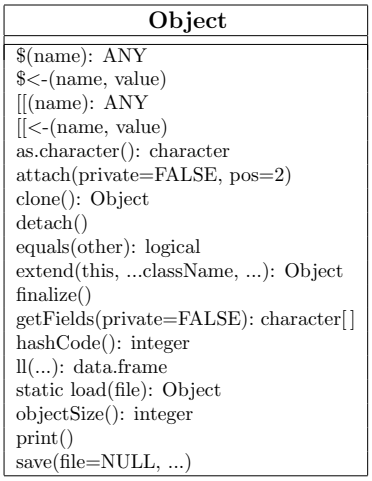

Figure 1: UML representation of the root class Object, which all classes should be derived from directly or indirectly through other classes. ANY is not a defined data type, but refers to any data type or class.

The method ***getFields(private=FALSE)*** returns the name of all fields in an Object. By default only names of non-private fields are returned.

The method ***ll(...)*** returns a data frame with detailed information about the fields of an Object. By default only non-private fields are listed. For more details see section 8.

The ***hashCode()*** method returns an integer hash code for an Object.

The ***objectSize()*** method returns the (approximate) size of an Object. Compare this with *object.size()*, which returns the size of the reference variable and not the Object.

The ***equals(other)*** method compares one Object with another. If they are equal the method returns `TRUE`, otherwise `FALSE`. If argument `other` is `NULL`, then `FALSE` is always returned. The default implementation of *equals()* compares the *hashCode()* values of both objects.

The ***clone()*** method creates an identical *copy* of an Object[3]. See also section 6.

When an Object is deallocated from memory by the garbage collector the ***finalize()*** method is first called. Subclasses can override this method to make sure that any instances of such classes clean up after themselves. For instance, objects that allocate shared resources such as connections should make sure that these resources are closed and deallocated upon deletion.

The methods ***attach(private=FALSE, pos=2)*** and ***detach()*** attaches and de-

[3]Doing `ref2 <- ref` will only create a new reference to the same instance.

taches an Object to and from the search path, respectively. By default only public fields (private=FALSE) of an Object are attached and by default they are attached to the beginning of the search path (pos=2) just after the global environment. Any modification to such attached fields will *not* be reflected (saved) in the actual Object when *detach()* is called.

The method ***save(file=NULL, ...)*** saves an Object to a file (or a connection) and the *static* method ***load(file)*** loads a previously saved Object and returns a reference to it.

The somewhat special method ***extend(...className, ...)*** extends an Object (class) into a subclass named according to the string ...className[4] and which contains all fields as given by the ... arguments. This method is not intended to be overridden by any subclass. For more details see section 5.

Finally, as explained in detail in section 6, the functionality for references is hidden inside the Object class. Hence, all subclasses will support references automatically and the programmer does not have to think about how reference variables should be implemented. They are always provided and they always behave in the same way.

3 Using classes inheriting from the Object class

Throughout this document we make use of a classic case study example to describe the major parts of the package. For a more extensive case study see Bengtsson (2002c). Let the class SavingsAccount represents a bank account, which in the simplest case can be described by its balance. To secure against illegal modifications of the balance we represent the balance with a private field named `.balance`. To obtain the balance of the account the function *getBalance()* is provided. Using *setBalance()*, it is possible to modify the account balance directly, but it is not possible to set it to a negative balance. More commonly used are the methods for withdrawal and depositing, i.e. *withdraw(amount)* and *deposit(amount)*, respectively. The withdrawal method will not accept withdrawals if the balance becomes negative. Moreover, SavingsAccount inherits from Object. When a SavingsAccount object is created, the balance will by default be set to zero.

3.1 Creating an object

The implementation of the class is described in section 5, but for now assume that the usage of the constructor is `SavingsAccount(balance=0)` and that

```
account <- SavingsAccount(100)
```

creates a SavingsAccount object with initial balance 100. The object is referred to by the reference variable `account`.

[4] The second argument to extend() has three dots as a prefix to make it possible to name fields such as className or similar.

3.2 Accessing fields

The fields of an instance of a class inheriting from root class Object can be accessed directly in a way similar to how elements of a list are accessed. For example, the balance field of the account object can the be retrieved by either `account$.balance` or `account[[".balance"]]`. To set the balance of the account either `account$.balance <- newBalance` or `account[[".balance"]] <- newBalance` will do.

Note that there is no way to prevent the access to *private* fields. However, if one follows the RCC rule (section 4) that private fields and only private fields should have a . (period) prefix, it should be clear which fields can be accessed from outside and which should be accessed only from inside the SavingsAccount class. Moreover, private fields named this way will, by default, not be listed by the functions *getFields()* and *ll()* (section 2), and neither by *ls()* (R Language Definition, 2003).

3.3 Calling methods coupled with a class

Under the S3 schema, a method coupled with a class is called in the same way as a regular function, but with the object as the first argument. While specifying the withdrawal and depositing methods above, we excluded the object argument for simplicity, e.g. *withdraw(amount)*. However, when calling the method one has to include it, e.g. `withdraw(account, amount)`. The method dispatching mechanism in S3/UseMethod will then make sure that the method of the correct class will be called. For more detailed information on how method dispatching is done in S3 see R Language Definition, 2003.

For developers who prefer an explicit COOP programming style, methods can also be accessed via the `$` (or the `[[`) operator, e.g. `account$withdraw(amount)`, which is very similar to how Omegahat's OOP package, but also how other object-oriented languages such as Java and C++ do it. However, until there is a well defined standard for doing this in R we do not encourage this style (except for static methods described next).

3.4 Calling static methods

A *static method* of a class is a method that by definition belongs to a class and it is invoked using only the class, i.e. it does not require an instance of a class. All classes extending the Object class can define static methods. The most readable way to call a method of a class is via the `$` operator, e.g. `Object$load(file)` and `Exception$getLastException()`, even though FOOP style, e.g. `load(Object(), file)` and `getLastException(Exception())`, is also supported.

3.5 Accessing virtual fields

For Object instances, there is a third way of calling methods. Methods with a name of format `get<`*`Field`*`>(object)` or `set<`*`Field`*`>(object, value)` can be accessed by what we denote as *virtual fields*, i.e. as `object$<`*`field`*`>`. For instance, the methods `getBalance(account)` and `setBalance(account, newBalance)` will be called whenever `account$balance` and `account$balance <- value` are evaluated,

respectively.[5] There are at least three real advantages of using virtual fields. First, it is possible, as the name suggest, to make it look like a class has a certain field, whereas it internally might use something else. For instance, a Circle class can have the two redundant fields named `radius` and `diameter` where one is a virtual field and the other is the actual field. Indeed, both might be declared virtual at the same time. We find that the use of virtual fields reduces the redundancy, which in turn reduces the risk for inconsistency. It also reduces the memory usage. One can also image that virtual fields are used to map rows and columns in tab-delimited data files, spreadsheets, database tables etc. Another advantage is that it is possible to restrict what values or data types a field can be assigned. For instance, we can prevent the user from setting a negative radius, e.g. `circle$radius <- -20`. Finally, virtual fields can prevent direct access to private fields or modification of constants. That is, they provide a mechanism for *encapsulation* (data hiding).

3.6 Accessing class fields

A *class field*, also known as a *static field*, is a field associated with the class itself, not with a particular instance of the class. A class field of a class is shared by all objects of that class. A common role of a class field is that of a global variable (with the important difference that it is not a global variable), e.g. `Colors$RED.HUE`. A class field is accessed as a regular field except that the object is now the static class object, e.g. `SavingsAccount$count <- SavingsAccount$count + 1`. Any class extending the Object class can have static fields. Static fields can also be implemented by virtual fields.

4 Coding conventions

An important part of object-oriented design and implementation is to follow a standard to describe the design and to implement it. There are several standards for describing object-oriented design of software in COOP style, e.g. Unified Modelling Language (UML) (Object Management Group, 2002). It is unknown to us if there is a corresponding one for the FOOP style. For implementation standards, also referred to as *coding conventions*, some languages have a well defined specification to follow whereas others do not. Unfortunately, there is no explicit and official coding convention for R. A well defined coding convention is useful because it helps to make the code more structured and more readable and it reduces the risk for mixing up field names with class names or reassign fields that are supposed to be constants etc. It is also fundamental for being able to efficiently share source code between developers and over time. Moreover, it provides a smoother and more robust way for statisticians that otherwise lack programming experience to quickly start contributing to existing in-house packages. A well defined coding convention can also be verified automatically, which decreases the amount of time needed for peer reviewing the source code. For reasons like these we are working on a *R Coding Convention* (RCC) draft (Bengtsson, 2002d). Next we will present an excerpt of its naming conventions.

[5] By default, virtual fields have higher priority than regular fields in case both exist. However, it is (on a reference-to-reference or an object-to-object basis) possible to change the order which fields, virtual fields, methods and static methods are accessed.

4.1 Naming conventions

Some of the naming convention rules of the RCC apply to object-oriented design and programming. One of the most important is how classes, fields and methods should be named. According to the RCC, names representing classes must be nouns and written in mixed case starting with upper case, e.g. `SavingsAccount`. Both field and method names must be in mixed case starting with lower case, e.g. `balance` and `getBalance()`. Private fields should have a . (period) as a prefix, e.g. `.balance`, to make it clear that it is a private field. Reserved keywords (R Language Definition, 2003) and unsafe method names must also be avoided according to the RCC. The methods `setConstructorS3()` and `setMethodS3()`, described next, enforce these naming rules and if not followed, an `RccViolationException` is thrown. Not all rules are enforced to be backward compatible with some basic R functions that (for obvious reasons) do not comply with the RCC. As a last resort, it is always possible to turn of the test against RCC by using the argument `enforceRCC=FALSE` when using the above functions. For all rules and the rationals behind them see Bengtsson (2002d).

5 Defining new classes

The two utility functions *setConstructorS3()* and *setMethodS3()* introduced next help the programmer to create constructors and methods without having to worry about generic functions. These functions can be used to define any S3 class, not only classes derived from Object. Note that there is no *setClassS3()*, cf. *setClass()* in S4. This is because S3, contrary to the richer S4 schema, does not have a formal class definition. For this reason there is also no (need for an) internal class definition database. However, this also means that there is no way to enforce that an instance of a class has the correct format, contains the correct fields, or to assure that the inheritance structure is valid. One reason for the latter weakness is that the class of the object and the inheritance structure of the class are solely specified by the class attribute of the individual objects. This attribute, which is characteristic to the S3 schema, can be modified in any way at any time making the object-oriented implementation vulnerable to programming mistakes, but also to misuse. The S4 schema overcomes some of these lack-of-robustness drawbacks. Moreover, because the Object class relies exclusively on the S3 schema, it implies that neither Object classes can be defined formally. Moreover, as we have no intention to reinvent the wheel, we leave it to the developer to "define" classes. However, as we will see, by means of the suggested class model and especially the *extend()* method of the Object class the definition of classes can still be done in a structured way such that the risk for errors and misuse is minimized.

5.1 Defining constructors

The *setConstructorS3()* sets the constructor function and automatically creates any necessary generic function (there are situations where this might be necessary). When defining a class descending from the Object class, its *extend()* method plays the role of defining the new class (its fields and which class to extend). This is preferably done within a constructor function. For example, to create the SavingsAccount class we write:

```
setConstructorS3("SavingsAccount", function(balance=0) {
  if (balance < 0)
    throw("Trying to create an account with a negative balance: ", balance);

  extend(Object(), "SavingsAccount",
    .balance = balance
  )
})
```

The declaration of the inheritance is done via the *extend()* method of the Object class, which will be called recursively throughout all the superclasses. The first argument to *extend()* should be the object returned by the constructor of the superclass. In the above example, the SavingsAccount inherits directly from the Object class, which is done by calling its constructor. The second argument to *extend()* should be the name of the class to be defined, e.g. SavingsAccount. According to the RCC, the name of the class should be the same as the name of the constructor function. Any other arguments to *extend()* are optional, but they must be named value arguments, e.g. `.balance=balance`, which then declare the fields of the class and their default values. Finally, all classes derived from Object *must* comply with the rule that it is should be possible to create an instance of it by calling its constructor with *no arguments*[6], e.g. `account <- SavingsAccount()`, cf. *prototypes* in S4.

5.2 Defining methods

The *setMethodS3()* method creates methods for any S3 class, not just Object classes, and at the same time encapsulates several details that the programmer should not have to think about. One such thing is if a generic function should be created or not and if so, how it should be created. For a detailed discussion on how generic functions are created automatically if missing see section 5.4. To create the *setBalance(newBalance)* method for the SavingsAccount class, the only thing needed is[7]:

```
setMethodS3("setBalance", "SavingsAccount", function(this, newBalance) {
  if (newBalance < 0)
    throw("Trying to create an account with a negative balance: ", balance);
  this$.balance <- newBalance;
})
```

The complete usage of *setMethodS3()* is:

```
setMethodS3(name, class="default", definition, private=FALSE, protected=FALSE,
                  static=FALSE, abstract=FALSE, trial=FALSE, deprecated=FALSE,
                    envir=parent.frame(), createGeneric=TRUE, enforceRCC=TRUE)
```

where `name` is the name of the method, `class` is the name of the class and `definition` is the definition, i.e. the function itself. If `class == "default"` (or `"ANY"`), a default function (R Language Definition, 2003) is created. Indeed, we highly recommend to use *setMethodsS3()* to define regular functions because this will minimize

[6] The reason for this is that *static class objects* are created by calling the constructor with no arguments.

[7] Note that fields still have to be accessed via the references variable, e.g. `this$.balance`. This is in line with the FOOP style, whereas in the COOP style one could imagine direct access, e.g. `.balance`. Omegahat's OOP package supports the latter.

the risk for future naming conflicts with generic functions, which in turn will make it simpler for other developers. For all other arguments see the help page of the function.

5.3 Details

The *setMethodS3()* method creates a standard S3 method and at the same time makes sure that a generic function for that method is available. For instance, the evaluation of

```
setMethodS3("getBalance", "SavingsAccount", function(this) {
  this$.balance;
})
```

will create the S3 method for the class and the S3 generic function, i.e.

```
getBalance.SavingsAccount <- function(this) {
  this$.balance;
}

getBalance <- function(...) UseMethod("getBalance")
```

It also makes sure that if there already exists a non-generic function with the same name, it will be renamed to `getBalance.default()`. If the latter also exists there is no way *setMethodS3()* can solve the conflict and therefore an exception will be thrown explaining this. If a generic function or an internal function that works as such already exists (determined by source code inspection), a new generic function is not created. For instance

```
setMethodS3("as.character", "SavingsAccount", function(this) {
  paste(data.class(this), ": balance is ", this$.balance, ".", sep="");
})
```

will only create the S3 method and *not* the generic function. In addition to this, *setMethodS3()* will by default verify that the (most important) RCC naming rules are followed. If not, it throws an RccViolationException informing that an RCC rule was violated. The above applies also to *setConstructorS3()*.

5.4 Safely creating generic functions

When writing a package it is important to make sure that the package does not overwrite preexisting functions. If a preexisting function exists that is not a generic function, in most cases, the conflict can be solved by redefining the function to become a default function. The test whether a function already exists or not is often done by hand. This is a tedious task for a developer as one has to stay up to date, not only will new versions of R, but also with all possible packages that the end user might use simultaneously. This is unrealistic as more and more packages are added to the CRAN.[8] More seriously, other packages might be loaded before or after our package is loaded and there is no way we can know which functions will be defined or

[8]The current effort of adding *name spaces* (R Development Core Team, 2003b) to the R language will remove some of the problems related to conflicting names of generic functions. A possible future support for *multiple generic function* may solve the problem completely.

not. A much safer approach is to check for conflicts and solve them when the package is loaded. Furthermore, it is important to make sure that the generic function will work with all packages and not just the methods in our package. Complete object-oriented programming (COOP style especially) requires that methods can have the same name for different classes, but with different sets of arguments. By not specifying the arguments of the generic functions, but only the special ... argument, e.g. `getArea <- function(...) UseMethod("getArea")`, we make sure the generic function is "as generic as possible"[9]. For a further discussion on how to create generic functions safely see Bengtsson (2002e). These problems are all automatically taken care of by *setMethodS3()* and *setConstructorS3()*.

6 Reference variables

All instances of the Object class or one of its subclasses are accessed via references variables or shortly *references*. In standard R where reference variables are not provided, each instance of a class is accessed by one single variable, the object itself. With references, however, it is possible for several variables to link to the same object. Here is an example where a list contains several references to the same Object:

```
person <- Person("Dalai Lama", 68)
l <- list(a=person, b=person, c=clone(person))
setAge(l$a, 67)
print(person)
[1] "Dalai Lama is 67 years old."
setAge(l$c, 69)
print(person)
[1] "Dalai Lama is 67 years old."
```

If `person` would *not* be a reference, the two elements `a` and `b` would be another two copies (clones) of the Person object and a modification of one of them would not have affected the other instance and neither the original variable `person`. It is possible to create a copy of an Object by using `clone()` as the above code shows.

In addition to being more memory efficient, references make it possible to implement software that otherwise would be tricky or impossible to implement. Using references, more details can be encapsulated and thereby the package will be more user friendly. We believe that a well designed object-oriented method interface based on references can serve as a base for, but also be a good complement to, a graphical user interface. For real-world examples see Bengtsson (2002c,a).

[9] If we *do* specify any arguments we restrict the corresponding methods for all classes in all packages loaded at the same time to have the exactly the same set of arguments. Under the S4 style, it is required and enforced that *all* methods for all classes have exactly the same arguments as the corresponding generic function. This is one of the reasons why we currently are not using the S4 style of programming with classes, but we hope to overcome this problem by making use of name spaces (and a possible future support for multiple generic function). Another option is to move completely to the COOP style and implement a stand-alone method dispatching mechanism similar to what is done in Omegahat's OOP package.

6.1 Garbage collector

The use of references requires memory management, but as we will explain next, R will automatically take of it. Many languages, including R, provide a built-in *garbage collector*, which removes obsolete objects from the memory that are not referred to by anyone. Since objects inherited from the Object class are standard R objects they will also be recognized by the garbage collector. For example, an Object created inside a function and for which no reference is returned, will be deleted by the R garbage collector. In summary, objects do not have to be deleted explicitly, but for an Object to be deleted it is important that all references to it are removed, e.g. by *rm()*, or set to `NULL`. It is always a good custom to do this as soon as an Object is not needed.

6.2 Details

R does *not* support references, but references can be emulated using so called *environments* (R Language Definition, 2003). However, using environments explicitly will quickly fill the source code with a lot of `get(name, envir=ref)`, `assign(name, value, envir=ref)` and/or `eval(..., envir=ref)` statements. This makes the code hard to read and increases the risk for errors. By encapsulating all calls to *get()* and *assign()* in the operator methods *$()*, *[[()*, *$<-()* and *[[<-()* of the root class Object[10], all fields can be accessed like if they were elements in a list. Since environments are used, garbage collection is automatically taken care of by R's memory management system, which minimizes the risk for memory bugs. By using environments it also possible to save Objects to disk or communicate them via a connection.

7 Exception handling

In addition to methods for defining classes and support for references, the package provides an extended *exception handling* mechanism where an exception can be thrown and then caught depending on its class. The core functionalities for exception handling is done by the Exception class (see figure 2). It provides methods to create and throw exceptions and together with its companion *trycatch()* complete exception handling is provided.[11]

7.1 Creating and throwing exceptions

The easiest way to create and throw an exception is by calling *throw()*, e.g.

```
throw("Division by zero.")
```

which is equivalent to calling

```
throw(Exception("Division by zero."))
```

[10] We have considered extracting the functionalities specific to environments into an Environment class to make it more explicit that environments are used to emulate references and that the usage of reference is unrelated to whether the COOP or the FOOP style is used. However, we decided not to do this (yet) in case R will support true references in the future, which we then silently would like to incorporate into our class model.

[11] R developer Luke Tierney presented an almost identical, but built-in, `tryCatch()` method during his talk *Some New Language Features for R* at the DSC 2003 conference.

An object of any class that inherits from Exception contains information about the error and when it occurred. Any Exception object can be thrown using the *throw()* method and then optionally be caught by either *trycatch()* or *try()*. If an Exception is thrown, the last exception thrown can be obtained by the static method *getLastException()* of class Exception. The *as.character()* method for the

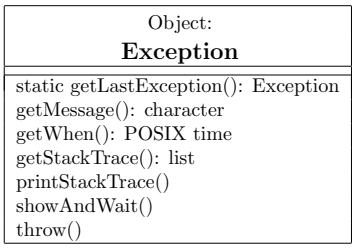

Figure 2: UML representation of the Exception class, which extends the Object class. 'Object:' in the header means that class extends the Object class.

Object class is overridden by the Exception class and the default print message of an Exception has the format:

```
> throw("Division by zero.")
Error: [2002-10-20 10:24:07] Exception: Division by zero.
```

7.2 Catching exceptions depending on class

The *trycatch()* method can catch exceptions based on what class they belong to. Like the *try()* function, the first argument to *trycatch()* is the expression to be evaluated, which might throw an exception. Any further arguments must be named arguments where the name specifies the Exception class to be caught and the value the code to be evaluated if such an exception is thrown. An argument with name `ANY` will catch any kind of Exception (including `try-error` thrown by `stop()`). If an exception is caught and no further exceptions are thrown, then `trycatch()` will return safely. The following code will generate and throw an exception, which will be caught by the `ANY` clause, preventing the R session from being interrupted.

```
trycatch({
  x <- log(2);
  y <- log("a");
}, ANY={
  x <- 0;
  y <- 0;
  print(Exception$getLastException());
})

print("trycatch() did indeed catch the exception.");
```

Moreover, code defined by an argument named `finally` is guaranteed to be evaluated immediately before *trycatch()* returns. This is for instance useful if a connection needs to be closed regardless of whether an exception is thrown or not.

8 Utility functions

In addition to the aforementioned methods, the package defines some useful utility functions, which are applicable to objects of any class or data type. The default method of ***ll()*** lists detailed information about the objects (variables and functions) found in an environment. The returned data frame will by default contain information about the *member* (name of the variable or function), *data.class*, *dimension* and *object.size*, which are the values returned by the functions with the same name. For example

```
> ll()
       member   data.class dimension object.size
1     analyze     function      NULL         248
2          ma       MAData         1         452
3         raw      RawData         1         452
4         gpr  GenePixData         1         460
5           y      numeric       100         828
```

For other utility functions available see the help of the package.

9 Other classes

Other classes that are loaded with this package are *Class*, *Package* and *Rdoc*. The class Class provides an interface for querying classes about methods, fields etc. The class Package represents any kind of package, e.g. `Package("base")`. Given a Package object it is possible to query it for its classes, its author, check for updates (see section 11 for an example) etc. The Rdoc class provides a compiler for Rdoc documentation, which is an extension of the Rd language that minimizes the need for having to update the documentation when the source code is updated. For instance, the tag `@synopsis` generates a correct `\usage` (or `\synopsis`) markup given the other information in the Rdoc code, but also given the source code. The Rdoc documentation can be standalone files similar to Rd files (the simplest Rdoc file is a plain Rd file) or it can be part of the source files in form of comments. The Rdoc code is compiled into standard Rd files, which are then converted into help pages etc. by `R CMD build`. As the Rdoc compiler is run within R with the package to be documented loaded, it is possible to generate parts of the documentation at compilation time. The reason for not generating a Rd skeleton, which is then to be filled in manually, is that at each update the skeleton has to be regenerated and the help text has to be reentered. This is not necessary if Rdoc comments are used. Currently the Rdoc compiler supports S3 and Object classes in COOP style (methods are listed at the same place as the class description), but we intend to extend it to recognize S4 and also generic functions in FOOP style (methods are listed at the same place as the generic function).

10 Limitations

First, using environments to emulate references limits the granularity of references to fields, e.g. it is not possible to reference a single element in a matrix etc. Hence, it could be argued that the term *semi-references* is more appropriate. Second, as the access to the fields of an Object is done indirectly via the evaluation of *$()* etc., there

is a noticeable (and unavoidable) overhead, which affects the overall performance. This effect can be significant if the same field is accessed a large number of times within a loop, say. In such cases, we recommend to work with a local copy of the field. Note that the amount of memory required is still much less that working on a local copy of the whole object. Third, multiple inheritance is (yet) not supported by the class model, but *interfaces* are. For more details see the package help. Finally, there is an (unspoken) intention in the R community to migrate to S4 and since the class model described above is relying on S3 this is a weakness.

11 Installation

Since the package was written in "100% R", no native code needs to be compiled and the installation is straightforward. The R.oo package is part of a bundle of packages called R.classes (Bengtsson, 2003). To download and install the R.classes bundle do

```
install.packages("R.classes", contriburl="http://www.maths.lth.se/help/R")
```

from within R. By default, the package is installed in the directory `library/` in the directory where R is installed. The bundle can be installed in a private directory by setting the environment variable `R_LIBS`. See `help(.Renviron)` for details. To install the bundle manually or on a Macintosh that does not have OS X see Bengtsson (2003). For future updates, load the package and do `update(R.oo)`.

12 Conclusions

The R.oo package is open source, it is designed in an object-oriented style and implemented using plain and richly commented R code ("100% R"). Moreover, it is designed and implemented such that any future migration from S3 to S4, but also such that the incorporation of a future language support for true references will be as smooth as possible for the end user. This should also be true if another solution, for instance Omegahat's OOP package, becomes the de facto standard for references in R.

For over two years we have used the R.oo package and the R.classes bundle in a project developing a cDNA microarray analysis package (com.braju.sma; Bengtsson, 2002a). We have found that by using the R.oo package we have never had problems with conflicts related to generic functions and we never have had to create a generic function explicitly. In situations when we by mistake tried to use a reserved word for a method name, *setMethodS3()* immediately notified us. We have also found the Rdoc compiler to be a valuable tool for maintaining its nearly 200 Rd files for over 50 classes and 800 methods. Due to the huge memory load and the large amount of redundancy in microarray data, the use of reference variables has been a natural and successful choice. Moreover, since methods can change the state of objects when references are used, we have been able to decrease the number of arguments that has to be specified in the method calls and therefore we can provide a cleaner and more user-friendly method interface. For the same reason we have been able to speed up subsequent analysis steps by caching intermediate calculations internally. For further discussion on how the R.oo package has been used in the development

of our microarray package see Bengtsson (2002b). To install the com.braju.sma package see Bengtsson (2002a).

References

Henrik Bengtsson. com.braju.sma - object-oriented microarray analysis in 100% R, 2002a. URL http://www.maths.lth.se/help/R/.

Henrik Bengtsson. The com.braju.sma package - a microarray analysis package based on an object-oriented design and reference variables, 2002b. URL http://www.maths.lth.se/help/R/.

Henrik Bengtsson. Programming with references - a case study using the R.oo package, 2002c. URL http://www.maths.lth.se/help/R/.

Henrik Bengtsson. R Coding Conventions (draft), 2002d. URL http://www.maths.lth.se/help/R/.

Henrik Bengtsson. Safely creating S3 generic functions using setGenericS3(), 2002e. URL http://www.maths.lth.se/help/R/.

Henrik Bengtsson. The R.classes bundle (R.oo and friends), 2003. URL http://www.maths.lth.se/help/R/.

John M. Chambers. *Programming with Data*. Springer, 1998.

John M. Chambers. The definition of generic functions and methods, January 2002a. URL http://developer.r-project.org/methodDefinition.html.

John M. Chambers. S language methods and classes, 2002b.

John M. Chambers and Duncan Temple Lang. Object-oriented programming in R. *R News*, 1(3):17–19, September 2001. URL http://CRAN.R-project.org/doc/Rnews/.

John M. Chambers and Duncan Temple Lang. OOP programming in the S language, 2002. URL http://www.omegahat.org/OOP/.

Robert Gentleman and Ross Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9:491–508, 2000.

Object Management Group. UML Resource Page, 2002. URL http://www.omg.org/uml/.

Andrew Shalit, David Moon, and Orca Starbuck. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley Publishing Company, 1996. ISBN 0-201-44211-6.

R Development Core Team. R Language Definition (v1.7.0, draft), April 2003a.

R Development Core Team. Writing R Extensions (v1.7.0), April 2003b.

W.N. Venables and B.D. Ripley. *Modern Applied Statistics with S-PLUS*. Springer, 3rd edition, 1999. ISBN 0-387-98825-4.

Affiliation

Henrik Bengtsson
Mathematical Statistics
Centre for Mathematical Sciences
Lund University, Box 118
SE-221 00 Lund, Sweden
E-mail: hb@maths.lth.se