S4 Classes and Methods

Friedrich Leisch

R Development Core Group

useR! 2004, Vienna, 22.5.2004

Acknowledgements

S4 has been designed and written by

- John Chambers

These slides contain material by

- Robert Gentleman
- Paul Murrell
- Roger Peng

Overview

- general remarks on object oriented programming
- the S3 class system:
  short, but almost complete
- the S4 class system
  much longer, but not nearly as complete

Introduction

Some reasons to perform object oriented programming:

1. productivity increase
2. easier to maintain code
3. reusable code
4. the design tends to follow the objects being modeled
5. encapsulate the representation of objects
6. specialize the behavior of functions to your objects

Object Oriented Design

- One identifies real objects and the operations on them that are interesting. These operations can then be systematically implemented.
- For example: we might have pdf's and cdf's as different objects, operations might be means, median, maxima and so on.
- A basic principle (or perhaps hope) is that by faithfully representing the objects we get easier to implement functions.
- A cdf object should know how to answer all the cdf questions.
- We want to think of the object in terms of what we want to do with it, not in terms of how we have implemented it.

Do you (as user) care how it is implemented?

Classes

A class is an abstract definition of a concrete real world object.

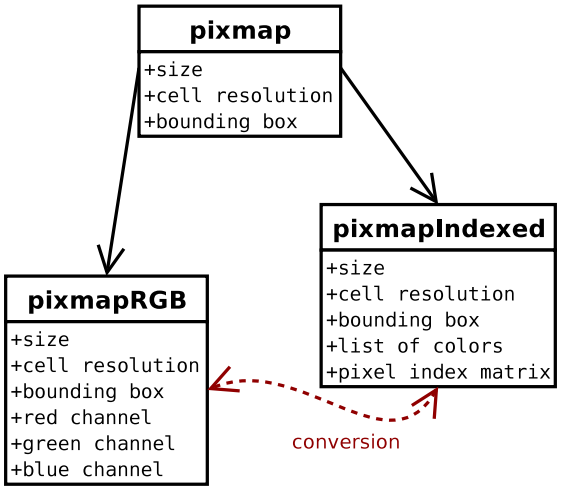
Suppose you are writing software to manipulate images. There are different ways of representing images (JPEG, GIF, PNG, BMP, ...) and different types of images (black/white, grey, color, ...). Then the objects are images of various types and any program that implements those objects in a format that reflects that will be both easier to write and easier to maintain.

A class system is software infrastructure that is designed to help construct classes and to provide programmatic support for dealing with classes.

Example: Pixmap Images

- How can we design a class or family of classes to hold pixmap images?
- Do we need two separate classes for RGB and indexed images or one class or ...?
- What happens if we get an indexed image and need an RGB image for some computations (e.g., to reduce the amount of red)?

Example: Pixmap Images



Classes and Objects

- A class is a static entity written as program code designed to represent objects of a certain type using slots (which in turn have other classes etc.).
- The class defines how an object is represented in the program.
- An object is an instance of the class that exists at run time.
- E.g., `pixmapRGB` is a class, the R logo read into R is an object that class.

Inheritance

- The hierarchical structure of classes we have seen for pixmap images is very typical for object-oriented programming.
- Classes `pixmapRGB` and `pixmapIndexed` extend class `pixmap` by defining additional slots, they inherit from `pixmap`.
- A class inheriting from another class must have all slots from the parent class, and may define additional new slots.

Methods

- Once the classes are defined we probably want to perform some computations on objects. E.g., a natural operation for images is to `plot()` them on the screen.
- In most cases we don't care whether the computer internally stores the image in RGB or indexed format, we want to see it on the screen, the computer should decide how to perform the task.
- The S way of reaching this goal is to use generic functions and method dispatch: the same function performs different computations depending on the types of its arguments.

Methods, Classes and the Prompt

- S is rare because it is both interactive and has a system for object-orientation.
- Designing classes clearly is programming, yet to make S useful as an interactive data analysis environment, it makes sense that it is a functional language.
- In "real" object-oriented languages like C++ or Java class and method definitions are tightly bound together, methods are part of classes (and hence objects).
- We want incremental/interactive additions (e.g., at the prompt), like user-defined methods for pre-defined classes.
- S tries to make a compromise, and although compromises are never optimal with respect to all goals they try to reach, they often work surprisingly well in practice.

S3 and S4

- The S language has two object systems, known informally as S3 and S4.
- S3 objects, classes and methods have been available in R from the beginning, they are informal, yet "very interactive". S3 was first described in the "White Book" (Statistical Models in S).
- S4 objects, classes and methods are much more formal and rigorous, hence "less interactive". S4 was first described in the "Green Book" (Programming with Data). In R it is available through the `methods` package, attached by default since version 1.7.0.

The S3 System

```
> x <- rep(0:1, c(10, 20))
> x
 [1] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
> class(x)
[1] "integer"
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.0000  1.0000  0.6667  1.0000  1.0000

> y <- as.factor(x)
> class(y)
[1] "factor"
> summary(y)
 0  1
10 20
```


The S3 System

- S3 is not a real class system, it mostly is a set of naming conventions.
- Classes are attached to objects as simple attributes.
- Method dispatch looks for the class of the first argument and then searches for functions conforming to a naming convention:
  - `foo()` methods for objects of class `bar` are called `foo.bar()`, e.g., `summary.factor()`.
  - If no `bar` method is found, S3 searches for `foo.default()`.
  - Inheritance can be emulated by using a class vector.
- The system is simple, yet powerful things can be done (take almost all of R as proof).

S3 Problems

There is no validation whatsoever if objects are valid for a certain class:

```
> nofactor <- "This is not a factor"
> class(nofactor) <- "factor"
> summary(nofactor)
numeric(0)
Warning message:
NAs introduced by coercion

> class(nofactor) <- "lm"
> summary(nofactor)
Error in if (p == 0) { : argument is of length zero
```

The computer should detect the real problem: a character string is neither a factor nor an `lm` object.

S3 Problems

Is `t.test()` a transpose methods for `test` objects?

```
> t
function (x)
UseMethod("t")
<environment: namespace:base>
> methods("t")
[1] t.data.frame t.default    t.ts*

    Non-visible functions are asterisked
> methods(class = "test")
no methods were found
```


S3 Problems

In package `tools`:

```
.makeS3MethodsStopList <-
function(package)
{
    ## Return a character vector with the names of the functions in
    ## @code{package} which 'look' like S3 methods, but are not.
    ## Using package=NULL returns all known examples
...
```

```
> tools:::.makeS3MethodsStopList(package = "stats")
[1] "anova.lmlist"       "fitted.values"      "lag.plot"
[4] "influence.measures" "t.test"
```

Of course this works only for base packages!

The S4 system

- define classes: `setClass()`
- create objects: `new()`
- define generics: `setGeneric()`
- define methods: `setMethods()`
- convert objects: `as()`, `setAs()`
- check object validity: `setValidity()`, `validObject()`
- access registry: `showClass()`, `showMethods()`, `getMethod()`
- ...

Defining S4 Classes

```
setClass("pixmap",
         representation(size="integer",
                        cellres="numeric",
                        bbox="numeric"),
         prototype(size=integer(2),
                   cellres=c(1,1),
                   bbox=numeric(4)))
```

```
> new("pixmap")
An object of class "pixmap"
Slot "size":
[1] 0 0

Slot "cellres":
[1] 1 1

Slot "bbox":
[1] 0 0 0 0
```

Defining S4 Classes

```
setClass("pixmapRGB",
         representation(red="matrix",
                        green="matrix",
                        blue="matrix"),
         contains="pixmap",
         prototype=prototype(new("pixmap")))


setClass("pixmapIndexed",
         representation(index="matrix",
                        col="character"),
         contains="pixmap",
         prototype=prototype(new("pixmap")))
```


Defining S4 Classes

```
> mypic = new("pixmapIndexed")
> mypic
An object of class "pixmapIndexed"
Slot "index":
<0 x 0 matrix>

Slot "col":
character(0)

Slot "size":
[1] 0 0

Slot "cellres":
[1] 1 1

Slot "bbox":
[1] 0 0 0 0
```


Defining S4 Classes

```
> mypic@index <- matrix(1:16, 4)
> mypic@col <- heat.colors(16)
> mypic@size <- dim(mypic@index)
> mypic
An object of class "pixmapIndexed"
Slot "index":
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16

Slot "col":
 [1] "#FF0000" "#FF1700" "#FF2E00" "#FF4600" "#FF5D00" "#FF7400" "#FF8B00"
 [8] "#FFA200" "#FFB900" "#FFD100" "#FFE800" "#FFFF00" "#FFFF20" "#FFFF60"
[15] "#FFFF9F" "#FFFFDF"

Slot "size":
[1] 4 4

Slot "cellres":
[1] 1 1

Slot "bbox":
[1] 0 0 0 0
```


Defining S4 Methods

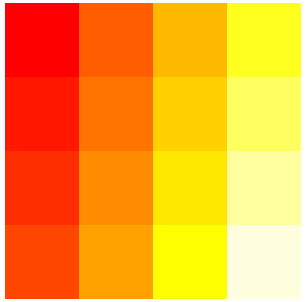
```
setMethod("show", "pixmap",
function(object){
    cat("Pixmap image\n")
    cat("  Type           :", class(object), "\n")
    cat("  Size           :", paste(object@size,
                                    collapse="x"), "\n")
    cat("  Resolution     :", paste(object@cellres,
                                    collapse="x"), "\n")
    cat("  Bounding box   :", object@bbox, "\n")
    cat("\n")
})

setMethod("plot", "pixmapIndexed",
function(x, y, xlab="", ylab="", axes=FALSE, asp=1, ...){

    image(t(x@index[x@size[1]:1,]), col=x@col,
          xlab=xlab, ylab=ylab, axes=axes, asp=asp, ...)
})
```

Defining S4 Methods

```
> new("pixmap")
Pixmap image
  Type           : pixmap
  Size           : 0x0
  Resolution     : 1x1
  Bounding box   : 0 0 0 0
> mypic
Pixmap image
  Type           : pixmapIndexed
  Size           : 4x4
  Resolution     : 1x1
  Bounding box   : 0 0 0 0
```


Defining S4 Methods

```
> plot(mypic)
```



Object Conversion

For `plot()`ting we may want to convert RGB images to indexed images such that we can also use the `image()` function.

There are already automatic coercion methods provided for us up and down along the edges of the inheritance tree:

```
> slotNames(mypic)
[1] "index"   "col"     "size"    "cellres" "bbox"

> mypic1 <- as(mypic, "pixmap")
> slotNames(mypic1)
[1] "size"    "cellres" "bbox"

> mypic2 <- new("pixmapRGB", mypic1)
> slotNames(mypic2)
[1] "red"     "green"   "blue"    "size"    "cellres" "bbox"
```


Object Conversion

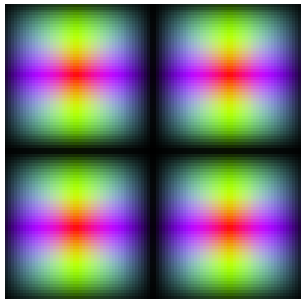
```
setAs("pixmapRGB", "pixmapIndexed",
function(from, to){

    x <- rgb(from@red, from@green, from@blue)

    col <- unique(x)
    x <- match(x, col)
    x <- matrix(x,  nrow=from@size[1], ncol=from@size[2])

    new("pixmapIndexed", size=from@size, index=x, col=col)
})

setMethod("plot", "pixmapRGB",
function(x, y, ...){
    plot(as(x, "pixmapIndexed"), ...)
})
```

Object Conversion

```
> x <- seq(-3, 3, length = 100)
> z1 <- outer(x, x, function(x, y) abs(sin(x) * sin(y)))
> z2 <- outer(x, x, function(x, y) abs(sin(2 * x) * sin(y)))
> z3 <- outer(x, x, function(x, y) abs(sin(x) * sin(2 * y)))
> mypic = new("pixmapRGB", size = dim(z1), red = z1, green = z2,
+     blue = z3)
> mypic
Pixmap image
  Type          : pixmapRGB
  Size          : 100x100
  Resolution    : 1x1
  Bounding box  : 0 0 0 0

> as(mypic, "pixmapIndexed")
Pixmap image
  Type          : pixmapIndexed
  Size          : 100x100
  Resolution    : 1x1
  Bounding box  : 0 0 0 0
```


Object Conversion

```
> plot(mypic)
```



Validity Checking

S4 objects are automatically checked for correct types whenever a slot is modified:

```
> mypic@size <- "Hello"
Error in checkSlotAssignment(object, name, value) :
  Assignment of an object of class "character" is not valid
  for slot "size" in an object of class "pixmapIndexed";
  is(value, "integer") is not TRUE
```

In addition, a function can be defined which checks whether an object is valid. E.g., for `pixmap` objects one of the two slots `size` and `cellres` is redundant given `bbox` the other. For valid objects the information should be consistent.

Validity Checking

```
setValidity("pixmap",
function(object){

    retval <- NULL
    if((object@bbox[3]-object@bbox[1]) !=
           object@size[1]*object@cellres[1])
    {
        retval <- c(retval, "cellres/bbox mismatch for rows")
    }
    if((object@bbox[4]-object@bbox[2]) !=
           object@size[2]*object@cellres[2])
    {
        retval <- c(retval, "cellres/bbox mismatch for columns")
    }

    if(is.null(retval)) return(TRUE)
    else return(retval)
})
```

Validity Checking

```
> mypic
Pixmap image
  Type          : pixmapRGB
  Size          : 100x100
  Resolution    : 1x1
  Bounding box  : 0 0 0 0

> validObject(mypic, test=TRUE)
Error in validObject(mypic) :
Invalid "pixmapIndexed" object: 1: cellres/bbox mismatch for rows
Invalid "pixmapIndexed" object: 2: cellres/bbox mismatch for columns

> mypic@bbox <- c(0,0,100,100)
> validObject(mypic)
[1] TRUE
```


Validity Checking

Of course one could simply use functions to check object validity, the advantages of `setValidity()` are:

- Validity checking methods are stored together with class definitions.
- If slots are themselves objects of classes with validity checks, they are also (recursively) checked.

Access to the Registry

All classes and methods are stored in a central registry

```
> showClass("pixmap")
Slots:

Name:     size cellres    bbox
Class: integer numeric numeric

Known Subclasses: "pixmapRGB", "pixmapIndexed"
```

on a per-package level:

```
> plot
standardGeneric for "plot" defined from package "graphics"

function (x, y, ...)
standardGeneric("plot")
<environment: 0x92f9530>
Methods may be defined for arguments: x, y
```


Access to the Registry

```
> showMethods("show")
Function "show":
object = "ANY"
object = "traceable"
object = "ObjectsWithPackage"
object = "MethodDefinition"
object = "MethodWithNext"
object = "genericFunction"
object = "classRepresentation"
object = "pixmap"
object = "pixmapIndexed"
    (inherited from object = "pixmap")
object = "pixmapRGB"
    (inherited from object = "pixmap")
object = "standardGeneric"
    (inherited from object = "genericFunction")
object = "function"
    (inherited from object = "ANY")
```

Access to the Registry

```
> getMethod("show", "pixmap")
Method Definition (Class "MethodDefinition"):

function (object)
{
    cat("Pixmap image\n")
    cat("  Type          :", class(object), "\n")
    cat("  Size          :", paste(object@size, collapse = "x"),
        "\n")
    cat("  Resolution    :", paste(object@cellres, collapse = "x"),
        "\n")
    cat("  Bounding box  :", object@bbox, "\n")
    cat("\n")
}

Signatures:
        object
target  "pixmap"
defined "pixmap"
```


Access to the Registry

- `selectMethod()`
- `existsMethod()`, `hasMethod()`
- `removeClass()`, `removeMethod()`, ...

Defining Generics

So far we have only defined new methods for existing generics (`plot`, `show`). To define a new generic simply do

```
> setGeneric("mygeneric", function(arg1, arg2, arg3) standardGeneric("mygeneric"))
[1] "mygeneric"
> mygeneric
standardGeneric for "mygeneric" defined from package ".GlobalEnv"

function (arg1, arg2, arg3)
standardGeneric("mygeneric")
<environment: 0x8e7e908>
Methods may be defined for arguments: arg1, arg2, arg3
```

where the number and names of arguments is of course arbitrary (and may include the special `...` argument). One of the main advantages of S4 generics is that dispatch can depend on the class of all arguments, not only the first argument.

Defining Generics

A special case is to turn S3 generics to S4:

```
> boxplot
function (x, ...)
UseMethod("boxplot")
<environment: namespace:graphics>

> setGeneric("boxplot")
[1] "boxplot"
> boxplot
standardGeneric for "boxplot" defined from package "graphics"

function (x, ...)
standardGeneric("boxplot")
<environment: 0x9278c14>
Methods may be defined for arguments: x
```

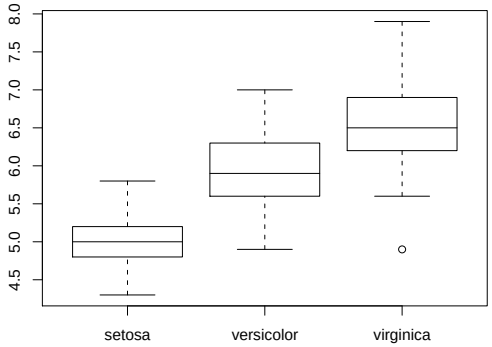
Multiple Dispatch

S3 methods have to do a lot of `if()...` `else...` computations to check what their arguments actually are:

```
> graphics:::plot.factor
function (x, y, legend.text = levels(y), ...)
{
    if (missing(y) || is.factor(y)) {
        ...
    }
    if (missing(y)) {
        barplot(table(x), axisnames = axisnames, ...)
    }
    else if (is.factor(y)) {
        barplot(table(y, x), legend.text = legend.text, axisnames = axisnames,
            ...)
    }
    else if (is.numeric(y))
        boxplot(y ~ x, ...)
    else NextMethod("plot")
}
```

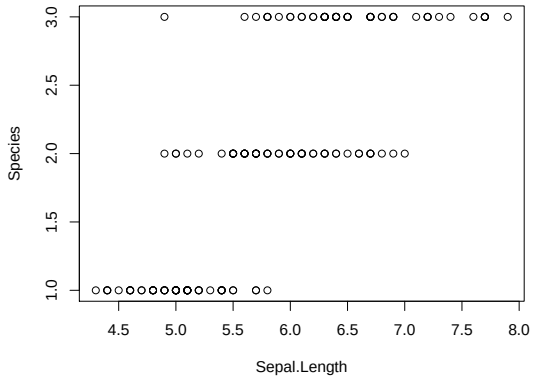

Multiple Dispatch

```
> plot(Species, Sepal.Length)
```



Multiple Dispatch

```
> plot(Sepal.Length, Species)
```



Multiple Dispatch

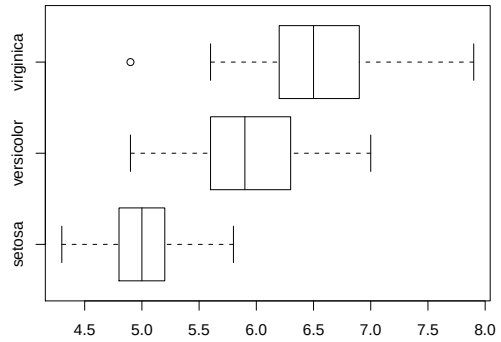
Using dispatch on multiple arguments makes the code much more transparent:

```
setMethod("plot", signature(x="numeric", y="factor"),
function(x, y, ...){
   boxplot(x~y, horizontal=TRUE, ...)
})

setMethod("plot", signature(x="factor", y="numeric"),
function(x, y, ...){
   boxplot(y~x, horizontal=FALSE, ...)
})
```

```
> showMethods("plot")
Function "plot":
x = "ANY", y = "ANY"
x = "pixmapIndexed", y = "ANY"
x = "pixmapRGB", y = "ANY"
x = "numeric", y = "factor"
x = "factor", y = "numeric"
```

Multiple Dispatch

```
> plot(Sepal.Length, Species)
```



Multiple Dispatch

There are two special "classes" that can be used in method signatures:

`ANY`: matches any class, corresponds to S3 default methods

`MISSING`: the call to the generic does not include this particular argument, corresponds to `if(missing(x)) ...` constructs in the body of function, but makes usage much clearer.

Example:

```
signature(arg1="factor", arg2="ANY", arg3="matrix",
          arg4="MISSING", arg5="ANY")
```


Getting Help

The question mark operator now has much more features than just `?topic`:

```
> class ? pixmap
> methods ? plot
> method ? plot("pixmap")
> ? plot(mypic)
```


Writing Help

Basically you can write documentation for S4 classes and methods like you do with S3. To enable R to actually find the correct help pages Rd files need entries like

```
\alias{pixmap-class}
\alias{pixmapRGB-class}

\alias{plot,pixmap-method}

\alias{coerce,pixmapRGB,pixmapIndexed-method}
\alias{coerce,ANY,pixmapIndexed-method}
```

`promptClass()` and `promptMethods()` create skeleton Rd files for a given class or method.

Other S4 Features

- group generics & methods
- replacement methods
- calling "next" method
- class unions
- ...

S4 Problems

- documentation, reference material
- still evolving (although much less than it used to)
- slower than S3
- glitches with name spaces

Current S4 Packages

R base: `stats4`

CRAN: less than 20 (out of 350+), including `Matrix`, `lme4`, `orientlib`, `flexmix` and `pixmap`

Bioconductor: almost all 50+ packages are S4

Conclusions

- S4 provides object oriented programming within an interactive environment.
- It can help you a lot to write clean and consistent code, and checks automatically if objects conform to class definitions.
- Multiple dispatch rather than nested `if() ... else` constructs in the body of functions.
- I personally start all new packages using S4.