Good Programming Practice

Martin Mächler
Seminar für Statistik, ETH Zürich

20. Mai 2004

maechler@R-project.org

This talk is . . .

- *not* a one or two days' course (from Insightful or . . .)
- *not* systematic and comprehensive like a *book* such as
  Chambers "Programming with Data" (1998),
  Venables + Ripley "S Programming" (2000),
  Uwe Ligges "R Programmierung" (2004) [in German]
- *not* for complete newbies
- *not* really for experts either
- *not* about C (or Fortran or C++ . . .) programming
- *not* always entirely serious ☺

This talk is . . .

- on R language programming
- my own view, and hence *biased*
- hopefully helping userR s to improve
- . . . . . somewhat entertaining ?

"Good Programming Practice"

- "Good", not "best practice"
- "Programming" using R : 

- "Practice": What I've learned over the years, with examples; *but*

. . . Practice . . .

but "The times they are a-changing" : Speed, memory and the software (R 1.9.x as opposed to S-plus 3.4) have improved much!

⋆ $\longrightarrow$ many 'tricks' no longer needed (nor would some still apply).
⋆ tradeoff speed $\longleftrightarrow$ memory is shifting:
Saving intermediate results may no longer be more efficient (not even in C), but
are still nicer to read and maintain.

Programming = ?

Is Programming

- like car driving, something you learn and then know to do?
- a scientific process to be undertaken with care?
- a creative art?

$\longrightarrow$ all of them, but not the least an *art*.
$\longrightarrow$ Your 'programs' should become *works of art* . . . ☺

In spite of this, *Guidelines* or *Rules for Good Programming Practice*:

Rule 1: Work with Source files!

Source files aka 'Scripts' (but more).

- obvious to some,
not intuitive for useRs used to GUIs.
- *Paradigm* (shift):
Do not `edit` *objects* or `fix()` them, but modify (and re-evaluate) their source!

In other words (from the ESS manual):

The source code is real.
The objects are realizations of the source code.

- Use a *smart* editor:
⋆ syntax-aware: parentheses matching "(..)"
highlighting (differing *fonts* & colors syntax dependently)
⋆ able to *evaluate* R code, by line, whole selection (region), `function`, and the whole file
⋆ command completion on R objects

such as

- Emacs + ESS ('Emacs Speaks Statistics') (all platforms)
- WinEdt + R-WinEdt (MS Windows)
- Alpha (Mac)
- Kate + R-Kate (KDE: Linux etc), (?), ...
- . . . . . (there are more)

Good source code

1a. is well readable by humans
1b. is as much self-explaining as possible

```
\end{Rule 1:  Work with Source files}
```


Rule 2: Good source code is well maintainable

(hence 'well readable' ('1a.' above))

2a. Do indent lines! (i.e. initial spaces)
2b. Do use spaces!
e.g., around $<-$, $=$, $<=$,. . . , $+$, $-$, ... ;
after ',' ; before '{'
2c. Do wrap long lines!
(at column 70–80; $\longrightarrow$ do not put the editor in fullscreen mode)

well maintainable (*Rule 2* cont.)

2d. Do use comments copiously! (about every 10 lines)
We recommend
'##' for the usually indented comments,
'#' for end-of-line comments (ESS: align to comment-column = 40),
and
'###' for the (major) beginning-of-line ones.

2e. Even better (but more laborious): Use Sweave (or another "weave & tangle" system such as `noweb`)

... well readable code and the assignment operator

Beware: this is very controversial, and I am severely biased!

Some (including me, but by far not all!) believe that
using $<-$ instead of $=$ leads to *far* easier readable code:

' $=$ ' is also used much in function *calls* (incl. `list(a=.., b=..)` and definitions (argument defaults) and

$<-$ stands out visually

and can be marked up (by font/color) quite easily in syntax-aware editors or pretty-printers, something really hard to achieve with $=$

```
\end{really-controversial}
```

. . . . . well maintainable (*Rule 2* (end))

2 x. Do follow *naming conventions* for function **argument names**, and if available also for new functions and/or classes.

But do *not* impose rigid rules here, since

1. programming is *art* (☺)
2. The S language has a long history with many contributers: We will live with some historical misnomers and have sometimes deprecated and replaced others.

2 . . . Modularity, Clarity: *"refine and polish your code"* (V&R): More on "well maintainable" in the following rules

Rule 3: Do read the documentation

and read it again and again . . .
(and—only then—submit bug reports ☺)

1. Books: V&R's, . . .
2. The manuals "An Introduction to R" (early),
   "Writing R Extensions" (when you're mutating from useR to programmeR)
3. The help pages! and try their **examples** (in ESS)
4. Do use `help.search()`!! (and read its help page to find out about fuzzy matching and the `agrep` argument!)

Rule 4: Do learn from the masters

An art is learned from the master artists:

Picasso, Van Gogh, Gauguin, Manet, Klimt . . .

John Chambers, Bill Venables, Bill Dunlap, Brian Ripley, Luke Tierney, . . .

Read others' source

Read the source – of packages

Nota bene: The R source of a package (in *source* state) is inside `<pkg>/R/*.R`, and *not* what you get when you print the function! e.g., `plot` or `dev.print` from `package:graphics`.

If the package source is not easily available to you, ***and if*** the package is not installed "binary", e.g.,

```
system.file("../graphics/R/graphics"))
```

gives you the name of a file with all the R source files *concatenated*. Inside this file, you'll find the real source, e.g., of `dev.print`.

Rule 5: Do not Copy & Paste !

because the result is *not* well maintainable:

Changes in one part do not propagate to the copy!

a) write functions instead

b) break a long function into *several* smaller ones, if possible

c) Inside functions : still *Rule 5*: "Do not Copy & Paste !!"
   → write local or (package) global helper functions
   ⟶ use many small helper functions in NAMESPACE.

d) Possibly use

```
mat[complicated, compcomp] <-
  if(A) A.expression else B.expression
```

instead of

```
if(A)  mat[complicated, compcomp] <- A.expression
else   mat[complicated, compcomp] <- B.expression
```


Rule 6: Strive for clarity and simplicity

first! . . . and second . . . and again e.g., think about naming of intermediate results ("self-explainable") but use short names for extended formulae

> V.&R: "Refine and polish your code in the same way you would polish your English prose" (using 'dictionary': your reference material)

⟶ modularity ("granularity")

Optimization: much much later, see below

Rule 7: Test your code!

a. Carefully write (small) testing examples, for each function ("modularity", "unit testing")

b. Next step: Start a 'package' via `package.skeleton()`. This allows (via `R CMD check` *<pkg>*)

- auto-testing (all the help pages examples).
  use `example(`*your_function*`)`
- specific testing (in a `./tests/` subdirectory, with or without strict comparison to previous results)
- documenting your functions (and data, classes, methods):
  takes time, but almost always leads you to improve your code !

Test your code! (*Rule 7* cont.)

c. Use software tools for testing: Those of `R CMD check` are in the standard R package `tools`.

Advanced (at version 0.0-0): Luke Tierney's *codetools*
http://www.stat.uiowa.edu/~luke/R/codetools/

Optimizing code

Citing from V&R's "S Programming" (p.172):

> Jackson (1975) "Principles of Program Design" . . . . . two much quoted rules (on 'code optimization'):
>
> - Rule 1 Don't do it.
> - Rule 2 (for experts only) Don't do it yet—that is not until you have a perfectly clear and unoptimized solution.
>
> to which we might add *'to the right problem by an efficient method'*.

Optimizing code - 2

1. Really do clean up and *test* your code and think twice before you even start contemplating optimizing the code . . .
2. do ***measure***, not guess:

> From: Thomas Lumley (tlumley@u.washington.edu)
> Date: 28 Feb 2001
> To: R-help
>
> There are two fundamental principles of optimisation
> 1) Don't do it unless you need it
> 2) Measure, don't guess, about speed.
>
> The simple way to answer questions about which way is slower/more memory intensive is to try it and see. Between Rprof(), unix.time() and gc(), you have all the information you need. . . . . .

"Case studies"

Case study 0 – The small features inside `cov2cor()`: Among others, how to improve, for a matrix M on

1. diag(a) %*% M
2. M %*% diag(b)

Case study 1: function() returning function

Good examples:

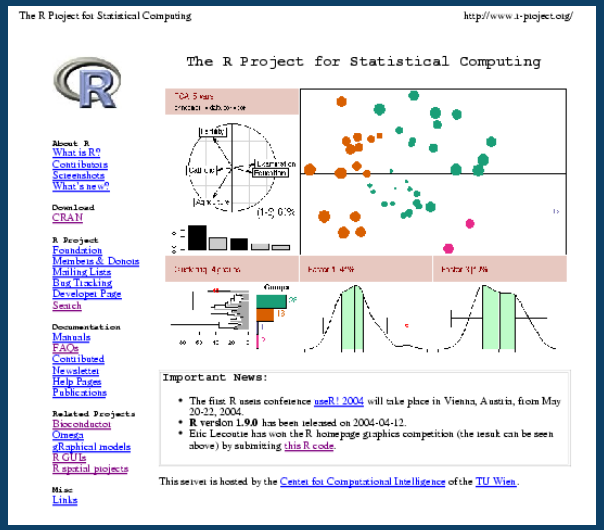
1. `help(ecdf)`, `example(ecdf)` (also `splinefun()`, etc)
2. The 'polynom' package by Bill Venabels et al. → `library(help=polynom)` has an `as.function()` method for polynomials
3. This talk: The '`scatterplotd3d`' package

```
library(scatterplot3d)          ## more modern: library(rgl) [1]
?scatterplot3d
```

Look at the `Value:` section (ESS: "s v" (skip to value)), and then at the Examples one, examples **5** and **6**.

[1] if only `rgl.close()` wouldn't seg.fault anymore . . . . . .

Case study 2 : The R Homepage Graphic Winner



Case study 3 : New `boxplot()` features

(in 'R-devel' or "R version 2.0.0 (unstable)"):

Using "**local functions**" for modularity and clarity.

An e-mail exchange MM with Arni Magnusson (UW, Seattle).

Specific Hints, Tips:

1. Subsetting ("[..]"):
   (a) Matrices, arrays (& data.frames):
   Instead of `x[ind ,]`, use `x[ind, , drop = FALSE]` !
   (b) tricky because of NAs
   For data frames (and vectors): Use `subset(x, ...)` instead of `x[, \dots]`
   Or, inside "[..]", often use `match()` or (a wrapper) `%in%` and `which()`.
2. Not x == NA but **is**.**na**(x)
3. Use '1:n' only when you *know* that n is positive:
   Instead of `1:length(obj)`, use **seq**(along = obj)

4. Do not *grow* objects:
   Replace

```
rmat <- NULL
for(i in 1:n) {
  rmat <- rbind(rmat, long.computation(i, .....))
}
```

   by

```
rmat <- matrix(0., n, k)
for(i in 1:n) {
  rmat[i, ] <- long.computation(i, .....)
}
```

   and if n can be large, it will pay off creating the *transpose*, column by column instead of row by row:

```
tmat <- matrix(0., k, n)
for(i in 1:n) {
  tmat[, i ] <- long.computation(i, .....)
}
```

5. Use `lapply`, `sapply`,
   the new `mapply` (Apply a function to multiple arguments), or sometimes the `replicate()` wrapper:

```
sample <- replicate(1000, median(rt(100, df=3)))
hist(sample)
```

6. Use `with(<d.frame>, .....)` and do *not* attach data frames
7. `TRUE` and `FALSE`, not 'T' and 'F' !
8. know the difference between '|' vs '||' and '&' vs '&&'
   and inside **if** (....) almost always use '||' and '&&' !
9. use `which.max()`, . . . , `findInterval()`
10. Learn about 'Regular Expressions': `?regexp` etc
11. . . . . *(more if time permitted)* . . . . .

Handouts will be available from the useR! web page by next week.

That's all Folks!

.. wishing you joy in R Programming!

Martin.Maechler@R-project.org