R, Bonnes pratiques

Christophe Genolini

Table des matières

1 Des bonnes pratiques, pour quoi faire ?

Quand les hommes ont commencé à envoyer des fusées dans l'espace 3 et qu'elles ont explosé en plein vol, ils ont écrasé une petite larme et ont cherché les causes de l'échec. Comme il fallait bien brûler quelqu'un, ils ont cherché un coupable. Et ils ont trouvé... les informaticiens. "C'est pas d'not' faute, ont déclaré les informaticiens tous marris, c'est un fait avéré intrinsèque aux ordinateurs : tous les programmes sont buggués !" Sauf que dans le cas présent, la facture du bug était plutôt salée... Des gens très forts et très intelligents ont donc cherché des moyens de rendre la programmation moins bugguée. Ils ont fabriqué des nouveaux langages et défini des règles de programmation. On appelle ça la *programmation propre* ou les *bonnes pratiques*.

Les *bonnes pratiques* sont des règles que le programmeur choisit de suivre pour améliorer la qualité de sa programmation et diminuer le nombre de bugs de ses programmes. Les règles que nous proposons ici sont soit adaptées des bonnes pratiques qu'on trouve dans les livres sur les langages objets, soit issues des discussions de la liste de diffusion de R [3] et du forum GuR [1], ou encore librement inspirées du document de Martin Mächler [2].

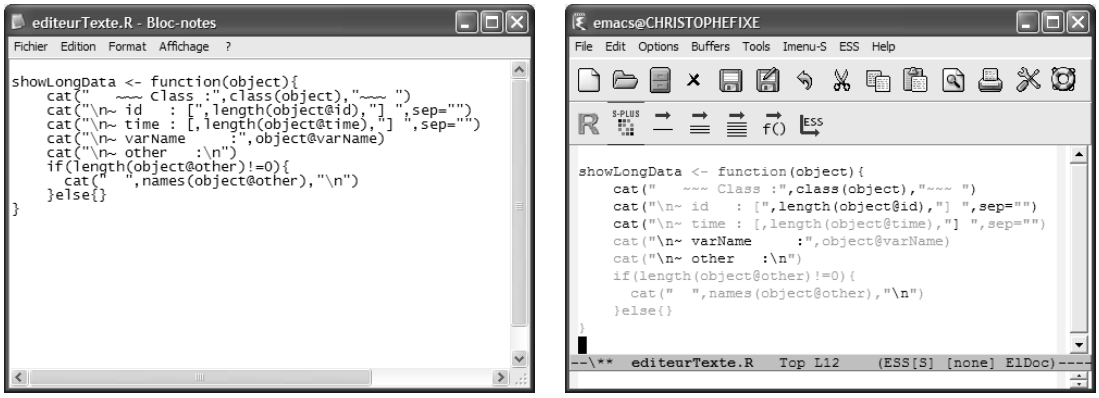
Avant d'entrer dans le vif du sujet, un petit avertissement : toutes ces règles et les situations qui les justifient donnent l'illusion d'être balayables d'un haussement d'épaules : "Ça, en faisant un peu attention, ça ne m'arrivera pas, pas besoin de règle." La pratique nous affirme l'inverse : *Même en suivant les règles*, on arrive tout de même à faire les erreurs qu'elles essaient de prévenir. D'où, ne vous y trompez pas, ces règles ne sont que des petites astuces pour étourdis : bien les utiliser fera toute la différence entre le bon programmeur et le programmo-touriste...

2 Choix de l'éditeur de texte

- E1 -
Utilisez un éditeur de texte intelligent
(avec coloriage, détection des parenthèses et indentation automatique).

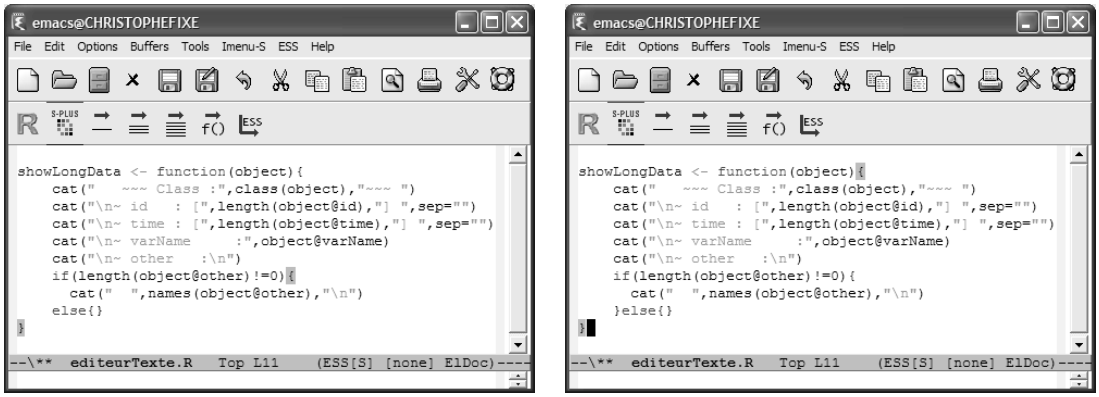
Installer un éditeur intelligent (comme `emacs` ou `Tinn-R`) ne vous prendra pas beaucoup de temps, mais cela vous fera gagner des heures de recherche de parenthèses ou de guillemets "mal placés"...

Voilà un exemple de code tiré du pacakge `kml`. Á gauche, il est non colorié. Où est l'erreur ?



Á droite, le même code est colorié. On voit que dans le bas du texte, des instructions comme `cat` sont dans la couleur réservée normalement au texte entre guillemets. On en conclut qu'il y a un guillement mal fermé... C'est celui du troisème `cat`.

De même, un éditeur évolué peut détecter automatiquement les couples de parenthèses et accolades. Plus précisément, dans un langage informatique, chaque parenthèse ouvrante doit être couplée à une parenthèse fermante ; même chose pour les accolades et les crochets. Un éditeur évolué a généralement la gentillesse de surligner la parenthèse ouvrante correspondant à la parenthèse fermante sur laquel est positionné le curseur, et réciproquement (même chose pour les accolades ou les crochets).



Ainsi, sur le code de gauche, l'accolade du bas que le programmeur pensait être l'accolade finale fermant la fonction s'avère fermer l'accolade du `if`. On peut en conclure qu'il manque l'accolade fermante du `if`. Sur le code de droite, l'accolage manquante a été ajoutée ; l'accolade du bas ferme bien l'accolade d'ouverture de la fonction.

3 Architecture du code

Un programme est généralement un code long et compliqué fait à partir d'instructions rudimentaires. Une suite de 10 000 instructions rudimentaires serait incompréhensible. Il est donc courant de les "regrouper" : des instructions rudimentaires sont assemblées en bloc, les blocs sont regroupés en fonctions, les fonctions sont elles même utilisées pour d'autres fonctions et au final, le programme principal sera composé de quelques fonctions. Ce découpage constitue "l'architecture d'un programme". Il est important que cette architecture soit visible. Pour cela, plusieurs règles :

- A1 -
Deux instructions distinctes doivent être sur deux lignes séparées.

Pour s'en convaincre, il suffit de se demander ce que fait le code suivant :

```
long <- 13; triangleRect <- function(long){result <- 0;
for(i in 1:long){cat("\n",rep("*",i));
result <- result+i};return(result);}
```

Une instruction par ligne rendrait les choses bien plus lisibles :

```
long <- 13
triangleRect <- function(long){
result <- 0
for(i in 1:long){
cat("\n",rep("*",i))
result <- result+i}
return(result)}
```

C'est plus lisible, mais l'architecture générale reste cachée. Pour la mettre en évidence, d'autres règles :

- A2 -
Les lignes doivent être indentées de manière à mettre les blocs constituant le code en valeur.

- A3 -
Chaque accolade fermante doit être verticalement alignée à l'instruction définissant l'accolade ouvrante correspondante.

Ces deux rêgles permettent la mise en évidence des blocs. Notre code devient

```
long <- 13
triangleRect <- function(long){
    result <- 0
    for(i in 1:long){
```

```
        cat("\n",rep("*",i))
        result <- result+i
    }
    return(result)
}
```

La structure commence à apparaitre : Au plus bas niveau (le plus décalé à droite), on distingue un bloc. Ce bloc est contenu dans une boule `for`. La boucle `for` plus l'initialisation de `result` sont eux-mêmes dans une fonction nommée `triangleRect`.

Enfin, on peut marquer un peu plus l'existence de blocs ou de fonctions en sautant des lignes :

- A4 -

Les blocs d'instruction ou les fonctions doivent être séparés par des lignes.

```
long <- 13

triangleRect <- function(long){
    result <- 0
    for(i in 1:long){
        cat("\n",rep("*",i))
        result <- result+i
    }
    return(result)
}
```

Il devient plus facile de comprendre ce que fait la fonction `triangleRect` : elle dessine un rectangle avec des étoiles et calcule sa surface :

```
> triangleRect(4)
```

```
 *
 * *
 * * *
 * * * *[1] 10
```

Pour certaines instructions, les accolades sont facultatives. Par exemple, quand un bloc d'instruction suit un `for` :

```
for(i in 1:5)
    cat(" I=",i)
```

```
 I= 1 I= 2 I= 3 I= 4 I= 5
```

Le résultat est le même que celui produit en utilisant le code avec accolades :

```
for(i in 1:5){
    cat(" I=",i)
}
```

```
 I= 1 I= 2 I= 3 I= 4 I= 5
```

Mais supposons que l'on souhaite ajouter une instruction dans la boucle. Dans le premier cas, on obtient :

```
for(i in 1:5)
    cat(" I=",i)
    cat(" I^2=",i^2)
```

```
I= 1 I= 2 I= 3 I= 4 I= 5 I^2= 25
```

Dans le deuxième :

```
for(i in 1:5){
    cat(" I=",i)
    cat(" I^2=",i^2)
}
```

```
 I= 1 I^2= 1 I= 2 I^2= 4 I= 3 I^2= 9 I= 4 I^2= 16 I= 5 I^2= 25
```

Dans le deuxième cas, on a effectivement ajouté une instruction dans la boucle. Mais dans le premier cas, comme il n'y a pas d'accolade, la ligne ajoutée est "hors boucle". Bien sûr, *en étant attentif*, on se serait rendu compte qu'il fallait ajouter les accolades. Mais les bonnes pratiques ont justement pour but de traiter les erreurs d'inattention. Omettre les accolades facultatives augmente le risque d'erreur de programmation. D'où la règle de "prudence architecturale" :

- A5 -
N'omettez pas les accolades facultatives.

De même, le `else` d'une instruction `if` est facultatif. Mais omettre un `else` peut introduire des ambiguités. En effet, considérons le code

```
if(cond1)if(cond2)cat("A")else cat("E")
```

Que veut le programmeur ? Veut-il :

```
if(cond1)
    if(cond2)
        cat("A")
else
    cat("E")
```

Ou veut-il plutôt :

```
if(cond1){
    if(cond2)
        cat("A")
    else
        cat("E")
```

Dans le premier cas, le `else` se rapporte au premier `if`. Dans le deuxième, le `else` se rapporte au deuxième. En théorie des langages, on appelle ça une “ambiguité syntaxique” : il manque quelque chose. Bien sûr, on peut *essayer* et voir comment R réagit. Mais la version de R peut changer ; si vous utilisez un autre logiciel de programmation, peut-être que ce dernier se comportera différament. Bref, il vaut mieux écrire sans faire de pari sur les réactions de l'interpréteur du langage. Selon ce que l'on souhaite :

```
if(cond1){
    if(cond2){
        cat("A")
    }else{
        cat("E")
    }
}else{
}
```

ou

```
if(cond1){
    if(cond2){
        cat("A")
    }else{
    }
}else{
    cat("E")
}
```

Il n'y a plus d'ambiguité. D'où la rêgle de désambiguiation :

- A6 -
Toutes les conditions doivent comporter un `else`, même s'il est vide.

En France, après un examen, les possibilités sont les suivantes : si vous avez moins de 8, vous avez échoué. Si vous avez plus de 10, vous êtes reçu. Si vous avez entre 8 et 10, vous pouvez présenter un oral de rattrapage deux semaines plus tard. Qu'est-ce qui est faux dans le code suivant ?

```
for(x in 1:100){if(note[x]<10){if(note[x]<8){cat("Fail")
}else{cat("You get it")}}
```

Voilà le même code après application des règles précédentes :

```
for(x in 1:100){
    if(note[x]<10){
        if(note[x]<8){
            cat("Fail")
        }else{
            cat("You get it")
        }
    }
```

Les erreurs sont bien plus facilement identifiables :

- Il manque une accolade
- Il n'y a qu'un seul `else` pour deux `if`. Le `else{cat("You get it !")}` est utilisé à la mauvaise place.

Le code correct est donc :

```
for(x in 1:100){
    if(note[x]<10){
        if(note[x]<8){
            cat("Fail")
        }else{}
    }else{
        cat("You get it")
    }
}
```

4 Variables

Les variables constituent généralement le cœur d'un programme. Bien les nommer est fondamental. Par exemple, que fait le code suivant ? Y a-t-il des bugs ? Que représente `m` ?

```
> n <- c(9,18,5,14)
> a <- c(17,18,18,17)
> nn <- 4
> (m <- sum(n)/a)
```

```
[1] 2.705882 2.555556 2.555556 2.705882
```

Mêmes questions avec le code suivant :

```
> noteEleves <- c(9,18,5,14)
> ageEleves <- c(17,18,18,17)
> nombreEleves <- 4
> (moyenneNotes <- sum(noteEleves)/ageEleves)
```

```
[1] 2.705882 2.555556 2.555556 2.705882
```

Comme vous pouvez le constater, le résultat final est le même. Mais dans le premier cas, on ne sait pas ce que sont `m`, `n` et `a` ; dans le deuxième, non seulement on sait de

quoi il retourne (selon toute vraisemblance, `moyenneNotes` est utilisée pour calculer la moyenne des notes des élèves), mais il est clair que le résultat devrait être un nombre unique et non un vecteur. Il serait également surprenant qu'une moyenne de notes tourne autour de 2.6. L'erreur est facilement identifiable : la somme des notes des élèves a été divisée par les âges au lieu du nombre d'élèves. D'où l'importance de bien choisir le nom des variables.

La première régle est celle que notre exemple vient de mettre en évidence :

- V1 -
Nommez vos variables explicitement

Une manière de faire est de choisir pour nom une suite de mots décrivant la variable mais sans les séparer par des espaces. Pour plus de lisibilité, chaque mot commence par une majuscule sauf le premier. Par exemple, `nombreDeFils`, `noteEleves` ou `ageEleves` sont des noms explicites dont la lecture explique le contenu.

Le côté "explicite" n'est cependant pas le seul à considérer. En effet, des noms de variables trop longs nous obligeraient à écrire un code sur plusieur lignes. Les instructions du langage seraient alors noyées, cela rendrait le code illisible :

```
> notesDesElevesDuGroupeDeTravauxDiriges6 <- c(9,18,5,14)
> nombreDElevesDuGroupeDeTravauxDiriges6 <- 4
> (moyenneDesNotesDuGroupeDeTravauxDiriges6 <-
+     sum(notesDesElevesDuGroupeDeTravauxDiriges6)/
+     nombreDElevesDuGroupeDeTravauxDiriges6
+ )
```

`notesDesElevesDuGroupeDeTravauxDiriges6` est clairement trop long... mais `ndedgdtd6` (uniquement les initiales de la variable précédente) n'est pas explicite. D'où un raffinement de la règle **V1** :

- V2 -
Cherchez un compromis :
les noms de variables doivent être de taille raisonnable...
tout en restant explicites.

La majorité des langages sont sensibles à la case (ils font la distinction les majuscules des minuscules). Il est possible d'utiliser cette propriété pour distinguer les variables, les fonctions et les classes. Dans ce tutorial, nous avons utilisé le principe suivant :

- V3 -
Utilisez des noms commençant par une majuscule pour les classes
par une minuscule pour les variables et les fonctions

Bien sûr, des variantes sont possibles. En particulier, si vous n'utilisez pas la programmation objet [1], vous pouvez commencer vos variables par une minuscule et distinguer vos fonctions par une majuscule.

1. Ce conseil peut paraître étrange dans un livre dédié à la programmation objet. Mais nous nous sommes laissé dire que ce manuel était aussi utilisé par des lecteurs intéressés uniquement par la construction de package classique et par les bonnes pratiques...

Variante : la notation Hongoise

Il est également possible de nommer les variables en commençant par une lettre qui donne leur type : par exemple, le nombre d'enfants est une variable entière (`integer`), la taille est un numérique (`numeric`) et la corrélation des valeurs propres d'une échelle de mesure est une matrice (`matrix`). Ces trois variables seraient donc nommées `iNombreFils`, `nTaille` et `mCorEchelle`. Pour un langage non typé comme R, cela présente un intérêt certain.

Concernant les noms de variables et fonctions, il est préférable que chacun ait un usage unique :

- V4 -
Un même nom ne doit jamais avoir deux usages

Nous l'avions déjà évoqué lors de la nomenclature du constructeur d'une classe mais le principe est généralisable. En particulier, des lettres comme `c` et `t` sont des fonctions R. Il est déconseillé de les utiliser pour stocker des valeurs (même si cela ne vous serait pas venu à l'esprit en vertu des règles **V1** et **V2** ?)

De même, il est peu souhaitable d'utiliser le même nom pour une fonction et pour un des arguments de la fonction. Par exemple, il serait maladroit de définir la fonction `salaire` :

```
salaire <- function(salaire){
    cat("Salaire horaire =",salaire/35)
}
```

5 Commentaires et documentation

Un programme est quelque chose de compliqué. Les commentaires permettent d'en faciliter la lecture.

- C1 -
Commentez votre code.

Les commentaires servent à écrire en français ce que le programme fait. On peut ainsi suivre son évolution calculs plus facilement. L'art du commentaire n'est pas aisé : en particulier, il ne faut pas trop "coller" au programme, tout en restant explicite. Par exemple, dans

```
> ### Affecte 2 à i
> i <- 2
```

le commentaire ne sert strictement à rien...

- C2 -
Commentez votre code intelligemment.

Les commentaires peuvent intervenir au niveau local, mais également global. En effet, un utilisateur (vous-même six mois plus tard) doit pouvoir utiliser votre fonction sans avoir à lire le code. Il est donc capital de bien préciser les variables à fournir en entrée et ce qu'il récupèrera à la sortie.

- C3 -
Documentez les entrées et sorties de chaque fonction / méthode.

Exemple, le package `kml` définit plusieurs fonctions qui travaillent sur des trajectoires. L'une d'entre elles les impute. Voilà ce qu'on peut lire dans le code :

```
### imputeTraj needs two arguments
###  - matrixToImpute is a matrix with or without missing values
###  - method is a character string in "LOCF", "mean", "multiple"
###
### ImputeTraj returning a matrix without missing values
###
imputeTraj <- function(matrixToImpute,method){
    ....
    return(matrixImputed)
}
```

Même sans lire le code, on sait ce que cette fonction prend comme argument et retourne comme valeur.

6 Divers

L'initialisation par défaut n'a plus vraiment de raison d'être.

- D1 -
N'utilisez pas de valeurs par défaut :
Une variable non initialisée doit provoquer une erreur.

En particulier, dans un langage statistique comme R, les valeurs manquantes jouent en rôle important (hélas !). Une initialisation malencontreuse peut fausser le résultat.

Dans le code suivant, on initie la variable `age` à zéro, puis on la modifie ensuite au fûr et à mesure qu'on reçoit de l'information :

```
> ### Liste de nom
> data <- data.frame(nom=c("Renée","Marcel","Raymonde","Isidore"))
> ### Initialisation avec la valeur 0
> data$age <- 0
> ### Renseignement des lignes "au fur et à mesure"
> data$age[1] <- 43
> data$age[3] <- 56
> data$age[4] <- 51
> ### Calcul de la moyenne
> mean(data$age)
```

```
[1] 37.5
```

Bien évidemment, le calcul de la moyenne est faux puisque l'âge de Marcel n'a pas été renseigné. La non-initialisation de la variable `age` (ou plus exactement son initialisation à NA) aurait permis d'éviter l'erreur :

```
> ### Liste de nom
> data <- data.frame(nom=c("Renée","Marcel","Raymonde","Isidore"))
> ### Initialisation avec la valeur NA
> data$age <- NA
> ### Renseignement des lignes
> data$age[1] <- 43
> data$age[3] <- 56
> data$age[4] <- 51
> ### Calcul de la moyenne
> mean(data$age)
```

```
[1] NA
```

- D2 -
Dans l'appel d'une fonction, spécifiez les arguments par leur nom.

Ne pas respecter cette règle, c'est s'exposer à intervertir involontairement l'ordre des arguments :

```
> ### Définition de la fonction IMC
> IMC <- function(taille,poids){
+     return(poids/taille^2)
+ }
> ### Mes paramètres
> monPoids <- 86
> maTaille <- 1.80
> ### Mon IMC sans specifier le nom des arguments
> IMC(monPoids,maTaille)
```

```
[1] 0.0002433748
```

```
> ### Mon IMC en specifiant le nom des arguments
> IMC(poids=monPoids,taille=maTaille)
```

```
[1] 26.54321
```

- D3 -
N'utilisez jamais de variable globale.
Jamais.

Une variable globale est une variable qui est définie à l'extérieur d'une fonction. Dès lors, l'utilisation de la fonction *dépend* d'autre chose que d'elle-même.

Or, le principe préludant à la construction d'une fonction est le même que celui d'une méthode : elle doit être automome, ne pas avoir besoin de l'environnement global. Une fois terminé, le programmeur ne doit plus avoir besoin de lire son code. Une fonction doit

pouvoir fonctionner même si son environnement change. En particulier, si vous copier-coller votre fonction dans un autre programme, elle doit fonctionner sans aménagement.

Dans l'exemple suivant, la fonction définie n'est utilisable que pour `moi` et non pour `lui` :

```
> ### Variables
> mesAnneesDEtude <- 8
> monSalaire <- 2500
> sesAnneesDEtude <- 5
> sonSalaire <- 3300
> ### Variation sur salaire
> ###    la variable mesAnneesDEtude est globale
> salaireDetail <- function(salaire){
+     cat("Salaire horaire =",salaire/35)
+     cat("\nRentabilité des études =",salaire/mesAnneesDEtude)
+ }
> ### Pour moi
> salaireDetail(salaire=monSalaire)
```

```
Salaire horaire = 71.42857
Rentabilité des études = 312.5
```

```
> ### Pour lui
> salaireDetail(salaire=sonSalaire)
```

```
Salaire horaire = 94.28571
Rentabilité des études = 412.5
```

Aucune erreur n'est signalée. Et pourtant, le calcul de `Rentabilite` pour `lui` est faux...

```
> ### Variables
> mesAnneesDEtude <- 8
> monSalaire <- 2500
> sesAnneesDEtude <- 5
> sonSalaire <- 3300
> ### Variation sur salaire
> ###    la variable mesAnneesDEtude est globale
> salaireDetail <- function(salaire,anneesDEtude){
+     cat("Salaire horaire =",salaire/35)
+     cat("\nRentabilité des études =",salaire/anneesDEtude)
+ }
> ### Pour moi
> salaireDetail(salaire=monSalaire,anneesDEtude=mesAnneesDEtude)
```

```
Salaire horaire = 71.42857
Rentabilité des études = 312.5
```

```
> ### Pour lui
> salaireDetail(salaire=sonSalaire,anneesDEtude=sesAnneesDEtude)
```

```
Salaire horaire = 94.28571
Rentabilité des études = 660
```

- D4 -
N'utilisez pas d'abréviation.

Par exemple, utilisez `FALSE` / `TRUE` et non `0` / `1` ou `F` / `T`. Les abréviations diminuent la lisibilité du code.

Enfin, tout règlement qui se respecte devrait comporter une clause précisant de ne pas *trop* respecter le règlement (sinon, gare au fanatisme...)

- D5 -
La clartée doit primer sur le respect des règles.
Si dans un cas particulier, une règle nuit à la lisibilité, ignorez-la !

Par exemple, plusieurs instructions `if` successives peuvent être plus lisibles si on les note sur une ligne unique. En respectant les règles :

```
setReplaceMethod("[","ClusterizLongData",
    function(x,i,j,value){
        if(i=="id"){
            x@id<-value
        }else{
        }
        if(i=="var"){
            x@var<-value
        }else{
        }
        if(i=="name"){
            x@name<-value
        }else{
        }
    }
)
```

Le code plus lisible quand on applique la règle **D5** :

```
setReplaceMethod("[","ClusterizLongData",
    function(x,i,j,value){
        if(i=="id"){x@id<-value}else{};
        if(i=="var"){x@var<-value}else{};
        if(i=="name"){x@name<-value}else{};
    }
)
```

7 Astuces de programmation

Enfin, voilà quelques astuces de programmation. Ce ne sont pas à précisément parler des *bonnes pratiques*, elles rentrent plus dans la catégorie des méthodes de programmation liées aux spécificités de R.

1. Testez votre code. Testez votre code régulièrement, n'écrivez pas un long code pour ne le tester qu'à la fin, cela rendrait le débuggage très difficile.
2. Écrivez de nombreuses petites fonctions et testez-les au fur et à mesure.
3. N'utilisez pas `x[ind,]`, remplacez-le par `x[ind,drop=FALSE]`.
4. N'utilisez pas `x==NA`, remplacez-le par `is.na(x)`.
5. N'utilisez pas `1:length(x)`, remplacez-le par `seq(along=x)`.
6. N'attachez pas vos `data.frame` à l'environnement (cela les tranformerait en variables globales, infraction **D3**).
7. N'utilisez pas `=` pour vos affectations, remplacez-le par `<-`.
8. N'essayez pas d'écrire un code optimal. Écrivez un code clair et simple. Plus tard, quand votre code sera opérationnel, bien plus tard, il sera temps de penser à l'optimisation.
9. Les boucles ne sont pas efficaces dans R. Il est préférable de les remplacer par les fonctions `lapply` et `sapply`.

Concernant les tests réguliers, il est plus facile de tester une fonction qu'une méthode. Aussi, il est plus simple de déclarer la fonction utilisée par une méthode *à part*. On vérifie ensuite qu'elle fonctionne correctement puis on l'intègre dans la méthode. Enfin, une fois la méthode déclarée, la fonction peut être supprimée sans que cela n'affecte la méthode.

A quoi bon la supprimer, diriez-vous ? C'est simplement pour ne pas laisser "trainer" dans l'espace de travail des variables, fonctions ou objets qui ne seront plus utilisées. C'est un peu comme passer l'éponge sur la table après le petit déjeuner pour enlever les miettes. De manière générale, plus c'est propre, moins il y a de bugs...

Donc, au lieu de :

```
> setMethod(
+     f="methodA",
+     signature="clasA",
+     definition=function(){
+         cat("Bonjour le monde")
+         return(invisible())
+     }
+ )
```

on peut écrire :

```
> ### Définition de la fonction
> .clasA.methodA <- function(){
+     cat("Bonjour le monde")
+     return(invisible())
+ }
> ### Ici, on teste .clasA.methodA :
> ###    - recherche des bugs
> ###    - detection des variables globales avec findGlobals
> ###    - tests préliminaires
> ###    - ...
>
> ### Ensuite, définition de la méthode
```

```
> setMethod(
+     f="methodA",
+     signature="clasA",
+     definition=.clasA.methodA
+ )
> ### Puis nétoyage
> rm(.clasA.methodA)
```

Références

[1] CIRAD. GuR : Groupe des Utilisateurs de R, 2004
http://forums.cirad.fr/logiciel-R.

[2] M. Mächler. Good Programming Practice, 2004
http://www.ci.tuwien.ac.at/Conferences/useR-2004/Keynotes/Maechler.pdf.

[3] R-Core Team. Liste de diffusion,
mailto:r-help@r-project.org.