Construire un Package

Classic et S4

Christophe Genolini

Table des matières

Chapitre 1

Création d'un package classique

1.1 Configuration de votre ordinateur [1]

1.1.1 Installation de programmes

Pour créer un package, il vous faut installer sur votre ordinateur un certain nombre de logiciels (tous disponibles gratuitement sur le web), puis les configurer.

- **Perl** : Perl est un langage optimisé pour l'extraction d'informations de fichiers textes et la génération de rapports.
  `http://www.activestate.com/Products/ActivePerl/Download.html`
- **Rtools** : Les Rtools sont des outils Unix qui fournissent une couche d'émulation pour le système Windows. Ils rendent possible l'exécution de programmes Unix sous Windows.
  `http://www.murdoch-sutherland.com/Rtools/tools.zip`
- **MinGW** : MinGW permet de compiler du code C, C++ et FORTRAN. Si vous n'incluez pas de langage autre dans votre code, vous n'avez pas besoin de `MinGW`. Sinon :
  `http://prdownloads.sourceforge.net/mingw/MinGW-5.0.0.exe`
- **HTML Help Workshop** : Ce logiciel permet de produire des aides au format `.chm`, le format d'aide propriétaire de Windows.
  `http://msdn.microsoft.com/library/en-us/htmlhelp/html/`
  `/hwmicrosofthtmlhelpdownloads.asp`
- **Un compilateur LaTeX** : LaTeX permet de produire une aide au format pdf. Plusieurs compilateurs sont disponibles, MiKTeX est assez stable.
  `http://www.miktex.org/`
- **FTP** : quand vous aurez terminé votre package, il vous faudra le poster sur le site du CRAN. Cela se fait grâce à un logiciel gérant le FTP (File Transfert Protocol). Là encore, plusieurs choix sont possibles.

1. Les sections 1.1 et 1.7 sont fortement inspirées (avec l'aimable autorisation de l'auteur) de l'excellent travail de Sophie Baillargeon. Pour des informations plus complètes, nous vous recommandons chaudement de lire son tutorial [1] ou le diaporama qui l'accompagne [2].

`http://filezilla-project.org/`

1.1.2 Modification du `PATH`

La création d'un package se fait via une fenêtre de commande `DOS` ou une fenêtre terminal sous `Linux`. Tous les nouveaux programmes qui viennent d'être installés doivent être accessibles depuis cette fenêtre. Pour cela, il faut préciser l'endroit où ils ont été installés sur votre ordinateur. Cela se fait en modifiant la variable `PATH`.

Le plus simple est de créer un fichier `Rpath.bat` contenant la définition de la variable `PATH`. Á noter que les espaces ne sont pas acceptés dans les noms de fichier. Sous Windows, `Programme File` doit donc être abrégé en `Progra~1`. La séparation des différents chemins peut se faire grâce à `;`. Pour plus de lisibilité, il est possible de définir le `PATH` sur plusieurs lignes :

```
SET PATH=C:\Rtools\bin
SET PATH=%PATH%;C:\Perl\bin
SET PATH=%PATH%;C:\Rtools\MinGW\bin
SET PATH=%PATH%;C:\PROGRA~1\R\R-2.9.1\bin
SET PATH=%PATH%;C:\PROGRA~1\R\R-2.9.1\include
SET PATH=%PATH%;C:\PROGRA~1\MIKTEX~1.6\miktex\bin
SET PATH=%PATH%;C:\PROGRA~1\HTMLHE~1
SET PATH=%PATH%;C:\WINDOWS
SET PATH=%PATH%;C:\WINDOWS\system32
```

Si vous sauvegardez `Rpath.bat` dans le répertoire racine `C:/`, il vous suffit ensuite de taper `C:/Rpath` dans la fenêtre système que vous venez d'ouvrir et votre variable `PATH` est modifiée comme il convient. Nous reviendrons sur l'utilisation de ce fichier dans la section sur la compilation finale 1.7.

Il est également possible de modifier la variable `PATH` en allant explorer les variables d'environnement. Mais ces modifications sont permanentes (jusqu'à ce qu'elles soient rectifiées). Cela veut dire qu'à chaque fois que vous allumerez votre ordinateur, le `PATH` sera modifié même si vous n'avez pas de package à compiler ce jour-là. Votre ordinateur sera un zeste moins rapide. Aussi, il est plus intéressant de créer un fichier `Rpath.bat` que l'on exécutera les jours où c'est nécessaire.

1.1.3 Répertoire `TMPDIR`

La construction d'un package nécessite l'existence d'un répertoire temporaire. Le plus simple est de créer le répertoire `C:/temp`. Il est également possible de modifier la variable `TMPDIR` (voir [1] pour plus de détail sur cette option.)

1.1.4 Ajustements pour MiKTeX

Si on utilise MiKTeX comme compilateur LaTeX, il y a une dernière manipulation à effectuer pour le rendre compatible. Il faut :

- Créer un sous répertoire nommé `/tex/` dans le répertoire `C:/localtexmf/`
- Copier tous les dossiers du répertoire `C:/Program Files/R/R-2.2.1/ /share/texmf/` dans le répertoire `C:/localtexmf/tex/`
- Pour s'assurer que tout fonctionne, aller dans : *Menu démarrer* → *Programmes* → *MiKTeX* → *MiKTeX Options* → *General* et cliquer sur *Refresh Now* et *Update Now*.

1.2 Création d'un programme... qui fonctionne !

Votre ordinateur est prêt, il faut maintenant vérifier que votre programme l'est aussi.

1.2.1 Quelques règles

- Votre programme doit fonctionner dans un environnement vierge, sans aucun préalable. Ouvrez R (sans restauration d'une session précédente), lancez votre programme, il doit s'exécuter.
- Votre programme doit être "propre". En particulier, il ne doit contenir QUE les variables et les fonctions que vous souhaitez voir apparaitre dans la version finale de votre package. Si vous avez utilisé des fonctions auxiliaires ou des variables qui vous ont servi à faire des tests, prenez soin de les effacer en fin de programme.
- Votre programme doit fonctionner... Or, la programmation est une tâche particulièrement ardue, les programmes sont buggués. Il faut les débuguer. Pour cela, il faut vérifier que les fonctions se comportent bien comme vous le souhaitez. Cela passe par des tests [2]. Beaucoup de tests. Ou plus précisément, des tests bien choisis et le plus exhaustifs possibles.
- Il est préférable de séparer la création des fonctions et les tests que vous effectuez dans deux fichiers différents.
- Les noms de fichiers ou de répertoires ne doivent pas comporter d'accent ni d'espace (les caractères spéciaux ne sont pas gérés de la même manière sous Linux et sous Windows, les transferts de l'un vers l'autre posent problèmes).

1.2.2 Les tests

Qu'est-ce que tester ?

Tester, c'est vérifier que ce que vous avez programmé se comporte bien comme vous le voulez. Par exemple, si vous programmez la fonction `carre` qui calcule le carré d'un nombre, vous vous attendez à ce que `carre(2)` donne 4 et `carre(-3)` donne 9. Si `carre(-3)` donne -9, c'est qu'il y a une erreur dans votre programme. Les tests sont là pour débusquer ce genre d'erreur.

Les tests en pratique

2. Notez que le *test informatique* est un concept complètement différent du *test statistique*...

Définir un test pour une fonction, c'est donner le résultat attendu relativement à une certaine entrée. La conception d'un test est donc complètement indépendante de la programmation de la fonction. En théorie, les tests se construisent avant l'écriture du programme.

Exemple : la fonction `impute` du package `kml` a pour mission de remplacer une valeur manquante au temps `i` par la moyenne des autres mesures présentes. Considérons les trajectoires $\begin{pmatrix} 3 & 12 & 23 \\ 4 & NA & NA \\ NA & NA & 25 \end{pmatrix}$.

La valeur manquante de la première colonne devrait être remplacée par la moyenne de $\{3, 4\}$ c'est à dire 3.5 Les valeurs manquantes de la deuxième colonnes devraient être remplacées par 12. La valeur manquante de la troisième colonne devrait être remplacée par la moyenne de $\{23, 25\}$ c'est-à-dire 24

Au final, après imputation, nos trajectoires devraient être $\begin{pmatrix} 3 & 12 & 23 \\ 4 & 12 & 24 \\ 3.5 & 12 & 25 \end{pmatrix}$.

Nous venons de définir un test informatique. Notez qu'il est indépendant de la fonction implémentée. Le test est l'expression de ce que l'on souhaite, pas de ce qui existe. Une fois le test conçu, on peut vérifier que ce que l'on a programmé est bien conforme à ce que l'on veut programmer. Dans notre cas, il s'agit de vérifier que la fonction `impute` réagit bien comme elle devrait.

Naturellement, un unique test est très insuffisant, il en faut bien d'autres.

Combien de tests ? Quoi tester ?

Les tests doivent vous permettre de vérifier que votre programme fonctionne correctement et cela dans tous les cas *même ceux auxquels vous ne pensez pas mais qui se présenteront sûrement un jour !* Difficile de prévoir même ce à quoi il nous est impossible de penser. Pour palier à ce "manque d'imagination", il est important de bien choisir ses tests : en prévoir un grand nombre mais surtout, prévoir des tests les plus variés possibles.

Á ce sujet, une petite anecdote : mon binôme et moi avions 16 ans ; c'était l'époque des premiers compatibles PC. Nous avions cours d'informatique (turbo pascal). Le problème du jour était de construire la fonction $f(x, y) = x^y$. Après avoir fait au moins 50 tests, nous appelons l'enseignant. Ex prof de math, un pince sans rire. Il arrive sans un mot, il tape 2^0 et crack, notre programme plante. Il repart, toujours sans un mot... Nous n'avions pas pensé à la puissance zéro. Rageurs, nous reprenons notre code, nous corrigeons, nous effectuons au moins 200 tests (parce qu'appeler le prof quand ça ne marche pas, à cet âge-là, c'est un peu la honte) et nous rappelons le prof. Il tape 0^2, ça marche. Victoire ! De courte durée : il essaie 0^0 et crack, ça replante... Une demi-heure plus tard, il nous fait boire notre calice jusqu'à sa lie avec -2^2...

Cette petite anecdote illustre bien le fait qu'il est plus important de bien choisir ses tests que de les multiplier. Faire 50 tests comme 2^3, 3^4, 8^2, ... a bien moins d'intérêt que les 9 petits tests 2^3, 0^3, -4^3, 2^0, 0^0,-4^0, 2^{-5}, 0^{-5} et -4^{-5}

Au final, pour tester une fonction, il faut essayer de trouver tous les types d'entrées qu'un utilisateur pourrait avoir à saisir (dans la mesure où il respecte les spécifications de la fonction : nous n'aurions pas à nous occuper d'un utilisateur qui essaierait `carre("bonjour")`) et de vérifier que le résultat est bien conforme à ce qu'on attend.

Des tests compatibles R Core team... ou pas ?

La R Core Team a mis au point un système de gestion de tests automatiques très performant mais un peu difficile à utiliser pour le programmeur non expert. Il nécessite en particulier d'avoir un package compilable (selon les conditions que nous verrons section 1.7.2 page 28) même pendant sa construction. L'expert développe son package en s'assurant qu'il est compilable et écrit en parallèle un fichier de test qui commence par l'instruction `library(monPackage)`. Quand il veut faire ses tests, il compile son package, puis exécute son fichier test. La première instruction, le chargement du package, lui assure que toutes les fonctions qu'il a définies sont effectivement chargées en mémoire en conditions réelles. Ainsi fait l'expert.

Á notre niveau, une telle démarche est impossible. En effet, la compilation d'un package présente en elle-même des difficultés, mélanger ces difficultés avec celles de la création d'un programme serait informatiquement suicidaire. Autre problème, les tests automatiques ne permettent de tester que les fonctions qui seront au final accessibles à l'utilisateur mais pas les fonctions intermédiaires qui seraient utilisées en interne dans le pacakge (les concepts de fonctions *internes* et *accessibles à l'utilisateur* seront abordés section 1.4.1 page 16). Dans ce livre, nous allons donc laisser de côté la gestion des tests automatiques pour experts et proposer une gestion alternative, certes manuelle, mais plus accessible.

1.2.3 Le programme

Notre package classique s'appelle `packClassic`. C'est ce qu'on appelle un "toy example", il n'a d'intérêt que sa simplicité... Il est constitué d'une base de données appelée `dataAges.Rda` et de deux fichiers `progClassic.R` et `testsProgClassic.R`.

1. Notre base de données `dataAges.Rda` est située dans le répertoire `/packClassic/data/`. En théorie, c'est une gigantesque base de données d'une valeur incalculable dont la collecte vous a demandé des années. Dans notre exemple, c'est simplement la sauvegarde d'un `data.frame` obtenu par le code :

```
##############################################################################
###                                  donnees                                 ###
##############################################################################

dataAges <- data.frame(
    sex=c("H","H","F","H","F"),age=c(5,6,5,4,8)
)
save(dataAges,file="packClassic/data/dataAges.rda")
```

Il est également possible de sauvegarder directement des données aux formats .csv, .txt ou .tab. Enfin, nous aurions pu placer dans le répertoire `/data/` un fichier R "fabriquant" les données.

2. `progClassic.R` est situé dans le répertoire `/packClassic/R/`. Il contient le code source du programme, la définition de 6 fonctions.

```
##############################################################################
###                              progClassic.R                             ###
##############################################################################

publicA     <- function(x){plot(x+1)}
privateA    <- function(x){x+2}
.publicB    <- function(x){x+10}
.privateB   <- function(x){x+20}
publicC     <- function(x){publicA(privateA(x))}
privateC    <- function(x){.publicB(.privateB(x))}
```

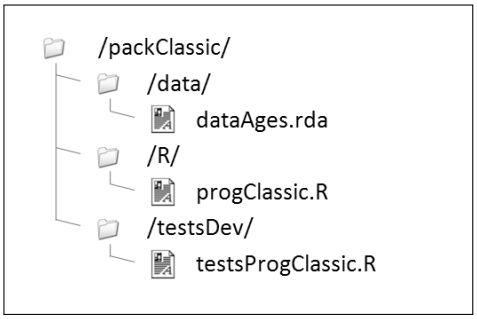
3. `testsProgClassic.R` est situé dans le répertoire `/packClassic/testsDev/`. Il contient les tests pour chaque fonction du code source.

```
##############################################################################
###                          testsProgClassique.R                          ###
##############################################################################

load("../data/dataAges.rda")
source("../R/progClassic.R")

publicA(1)
privateA(2)
.publicB(3)
.privateB(4)
publicC(5)
publicC(dataAges$age)
privateC(6)
```

Au final, on obtient l'arborescence suivante :



Note : nous venons de présenter ici une version terminée de nombre programme. Mais la structure que nous sommes en train de mettre en place est également utilisable pendant la phase développement. Ainsi, après avoir développé la fonction `publicA` et `privateA`, l'état des fichiers aurait été :

```
################################################################################
###                                progClassic.R                             ###
################################################################################

publicA     <- function(x){plot(x+1)}
privateA    <- function(x){x+2}
```

et

```
################################################################################
###                            testsProgClassique.R                          ###
################################################################################

load("../data/dataAges.rda")
source("../R/progClassic.R")

publicA(1)
privateA(2)
```

Pour vérifier que notre programme fonctionne (quand il est finalisé comme pendant la phase développement), il faut définir le répertoire `/packClassic/ /testsDev/` comme étant le répertoire courant de R, copier les tests puis les coller dans la console de R. Si on veut faire simplement le programme sans les tests, il faut se placer dans le repertoire `/packClassic/R/` puis exécuter `source("progClassic.R")`.

Note : pour utiliser la gestion des tests automatiques de la R Core team, il faudrait placer le fichier tests `testsProgClassic.R` dans le répertoire `/packClassic/ /tests/` et le modifier de la manière suivante :

```
################################################################################
```

```
###                          testsProgClassique.R                          ###
##############################################################################

library(packClassic)
data(dataAges)

publicA(1)
.publicB(3)
publicC(5)
publicC(dataAges$age)
```

1.3 Création de l'arborescence et des fichiers

Un package est un ensemble de plusieurs fichiers et répertoires, tous réunis dans un répertoire racine. Le répertoire racine a pour nom le futur nom du package (`packClassic` dans notre exemple). Il contient un fichier nommé `DESCRIPTION`, un fichier nommé `NAMESPACE` (optionnel) plus les répertoires `/R/`, `/man/`, `/data/` (optionnel) et `/tests/` (optionnel).

- Le répertoire `/R/` contient le code des programmes (dans notre cas `progClassic.R`).
- `/man/` contient les fichiers d'aide.
- `/data/` contient les jeux de données que nous avons décidé d'inclure dans notre package (présentement `dataAges.Rda`).
- `/tests/` contient les fichiers permettant de tester notre programme (tests selon la R Core Team).
- Comme nous avons pris l'option "tests simplifiés non compatibles", nous le remplacerons par `/testsDev/` pour définir des tests utilisables pendant le développement.

Dans notre exemple, nous supposerons de plus que le repertoire racine est lui-même situé dans le repertoire `/TousMesPackages/`.

1.3.1 `package.skeleton`

`package.skeleton` est une fonction qui crée pour vous l'arborescence des fichiers (répertoire racine, répertoires `/R/`, `/man/` et `/data/`). Elle crée aussi les fichiers d'aide (dans `/man/`), les fichiers codes (dans `/R/`), éventuellement les données (dans `/data/`), le fichier `DESCRIPTION` et éventuellement le fichier `NAMESPACE`. L'idée est d'utiliser `package.skeleton` pour la création initiale de votre package, puis de modifier ensuite "à la main" les fichiers qu'il a créés.

Pour mémoire, nous voulons créer un package `packClassic` à partir du programme `progClassic.R` qui se trouve déjà dans le répertoire `/packClassic/R/` lui même situé de `/TousMesPackages/`. Le plus simple est de se placer dans le répertoire `/TousMesPackages/` puis de taper :

```
> package.skeleton("packClassic",
+     code_file="packClassic/R/progClassic.R",
```

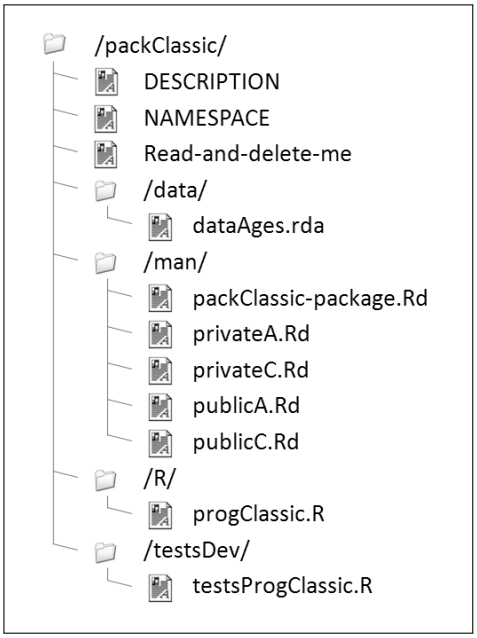
```
+      namespace=TRUE,
+      force=TRUE
+ )
```

Cela a pour effet de :

- Créer fichier `DESCRIPTION` dans le répertoire `/packClassic/`.
- Créer le répertoire `/packClassic/R/`. Comme nous avons choisi l'option `force=TRUE`, celui qui existe déjà est effacé.
- Copier le fichier `progClassic.R` dans le répertoire `/packClassic/R/`.
- Créer le répertoire `/packClassic/man/`.
- Créer de nombreux fichiers d'aide dans `/packClassic/man/`.
- Créer le répertoire `/packClassic/data/`. Ce répertoire n'est créé que dans le cas où le programme `progClassic.R` contient la définition de données.
- Créer un fichier appelé `Read-and-delete-me`. Ce fichier contient des informations sur la création du package et peut être effacé sans problème après lecture.
- Le répertoire `/packClassic/testsDev/` n'est pas modifié.
- `namespace=TRUE` est une option que l'on peut supprimer si on ne compte pas préciser l'accessibilité des fonctions. Dans ce livre, nous allons préciser...

La "destruction / recréation" des fichiers sources permet à R de vérifier que les noms donnés à nos noms de fichiers sont corrects. S'ils ne le sont pas, `package.skeleton` les renomme en ajoutant des `ZZZ` ou des `_` à la place des caractères non acceptés.

Au final, après l'exécution de `package.skeleton` :



1.3.2 Création des aides pour les données

`package.skeleton` a créé plusieurs fichiers d'aide, mais il n'a pas créé un fichier pour `dataAges`. Or, la création d'un package nécessite aussi la documentation des jeux de données. Il faut donc ajouter un fichier d'aide. Cela se fait grâce à l'instruction `prompt`, instruction que l'on peut utiliser après s'être assuré que les données sont effectivement en mémoire :

```
> ### Charge des données en mémoire
> load("packClassic/data/dataAges.Rda")
> ### Ajoute le fichier d'aide à /packClassic/man/
> prompt(dataAges,filename="packClassic/man/dataAges.Rd")
```

Si, bien plus tard dans la création du package, à l'heure où tout est presque terminé et où il devient coûteux d'utiliser `package.skeleton`, on se rend compte qu'on a oublié de créer une fonction, on peut toujours la créer et créer son aide grâce à `prompt`. Il suffit de donner à `prompt` le nom de la fonction et de lui préciser que c'est une fonction (et non des `data` comme dans le cas précédent). Cela se fait grâce à l'argument `force.function` :

```
> funcDerniereMinute <- function(x){cat(x)}
> prompt(funcDerniereMinute,force.function=TRUE)
```

Cette instruction crée le fichier d'aide `funcDerniereMinute.Rd` dans le répertoire courant.

1.3.3 DESCRIPTION

Le fichier `DESCRIPTION` contient, vous l'aurez deviné, une description du package. Voilà celui que `package.skeleton` vient de créer pour nous :

```
Package: packClassic
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2009-05-23
Author: Who wrote it
Maintainer: Who to complain to <yourfault@somewhere.net>
Description: More about what it does (maybe more than one line)
License: What license is it under?
LazyLoad: yes
```

Pour l'adapter à notre package, il suffit de l'ouvrir dans un éditeur de texte et de le modifier. C'est assez rapide :

- `Package: packClassic` : nom du package. Il est important que le nom du package soit le même partout, en particulier le nom du répertoire racine doit être le même que celui présent sur cette première ligne. Le nom ne doit pas contenir de caractères spéciaux ou d'espace.
- `Type: Package` : pas de modification.

- `Title: Toy Pack Classique` : titre du package. Contrairement au nom du package, le titre peut contenir des espaces et des caractères spéciaux.
- `Version: 0.5` : La version 1.0 est la première version stable (c'est-à-dire utilisable et raisonnablement peu bugguée). Les versions 0.x sont les versions prêtes à x%. Si nous estimons notre package prêt à 50%, nous pouvons choisir `Version: 0.5`. Si nous ajoutons une petite amélioration, elle sera publiée dans la version `0.5.1`. Quand tout sera finalisé, nous publierons une version `0.9` pour que d'autres la testent. Quand les testeurs auront rapporté les bugs les plus importants et que nous aurons corrigé, il sera alors temps de publier la `Version 1.0`. Pour l'instant, 0.5 semble raisonnable.
- `Date: 2008-07-22` : Pas de modification. Á noter, la date est au format international : année puis mois puis jour.
- `Author: Christophe Genolini (and some helpfull readers...)` : les auteurs.
- `Maintainer: <genolini@u-paris10.fr>`: Un package est un programme qui a pour vocation d'être utilisé par d'autres. Ces autres vont trouver des bugs ou exprimer des besoins. L'adresse mail est l'adresse de la personne à qui ils doivent écrire.
- `Description: This package comes to illustrate the book "Petit manuel de Programmation Orientée Objet sous R"` : La description permet de préciser de manière un peu plus détaillée ce que fait le package.
- `License: GPL (>=2.0)` R est un logiciel libre. Votre package doit également être défini comme libre, c'est à dire sous licence GPL (Gnu Public Licence). Pour les non experts, la "GPL 2 et suivantes" semble assez classique.
- `LazyLoad: yes`, pas de modification.

Les lignes suivantes sont optionnelles et ne sont pas dans le fichier par défaut :

- `Depends:graphics` : Si votre package fait appel à d'autres packages, vous devez ajouter une ligne `Depends:nomAutrePackageA nomAutrePackageB`.
  De manière plus précise, toutes les fonctions de R appartiennent à un package [3]: `ls` est dans le package `base`, `plot` est dans `graphics`, `anova` est dans `stats`... Tous les packages auquel notre programme fait appel doivent être listés. Seule exception à la règle, le package `base` est automatiquement inclus, il n'a pas besoin d'être déclaré. Dans le cas présent, nous utilisons `plot`, il faut donc préciser que nous avons besoin du package `graphics`.
- `URL:www.r-project.org,christophe.genolini.free.fr/webTutorial` : le but de faire un package est ensuite de le distribuer. `URL` permet de préciser les sites sur lesquels votre package est disponible. Classiquement, on devrait pouvoir le trouver au moins sur le site du CRAN plus éventuellement sur un site dédié.
- `Encoding` : si votre fichier DESCRIPTION contient des caractères non internationaux (par exemple é, è, à, ç ...), il faut ajouter la ligne `Encoding: latin1`. Mais

3. Pour connaitre le package auquel une fonction appartient, il suffit de consulter l'aide de la fonction. Sur la premier ligne en haut à gauche, le nom du package est donné entre parenthèses, juste à côté du nom de la fonction.

le package est alors moins portable, il vaut mieux éviter d'utiliser cette option.
– `Collate` : Si votre programme se décompose en plusieurs fichiers et que ceux-ci doivent être exécutés dans un ordre précis, alors cet ordre doit être précisé à l'aide de `Collate` (pour nous, la question de l'odre ne se pose pas dans le cas de notre package classique mais nous en aurons besoin lors de la création de notre package S4).

Au final, voila notre fichier `DESCRIPTION` :

```
Package: packClassic
Type: Package
Title: Toy Pack Classic
Version: 0.5
Date: 2009-05-25
Author: Author: Christophe Genolini (and some helpfull readers...)
Maintainer: <genolini@u-paris10.fr>
Description: This package comes to illustrate the book
  "Petit Manuel de Programmation Orientee Objet sous R"
License: GPL (>=2.0)
LazyLoad: yes
Depends: graphics
URL: www.r-project.org,christophe.genolini.free.fr/webTutorial
```

1.4 Visibilité des fonctions : `NAMESPACE`

Le fichier `NAMESPACE` sert à définir la visibilité de nos fonctions et des fonctions des autres packages, via `import` et `export`. Contrairement à ce qui est présenté dans de nombreux tutoriaux, `import` et `export` cachent deux concepts différents. En particulier, on peut avoir besoin de définir `export` mais pas `import` (l'inverse est plus rare).

1.4.1 `export`

Lorsqu'on crée un package, on peut définir plusieurs niveaux d'accessibilité des fonctions.

Public et privé

On peut choisir que certaines fonctions soient accessibles à l'utilisateur, que d'autres soient réservées au programmeur et à son programme. Les fonctions accessibles sont appelées *publiques*), les réservées sont dites *privées*. Le statut public ou privé se définit dans le fichier `NAMESPACE`. Les fonctions publiques doivent être déclarées dans `export`. Les fonctions privées doivent simplement ne pas être déclarées.

Visible et invisible

Parallèlement à public et privé, il est possible de définir des fonctions *visibles* ou *invisibles*. Les fonctions invisibles sont simplement des fonctions dont le nom commence

par un point "." . Ces fonctions ne sont pas affichées lorsque l'utilisateur demande un affichage général `ls()`. Par contre, elles sont utilisables normalement.

A noter que le `.` ne change pas le statut (public ou privé) d'une fonction, il affecte simplement l'instruction `ls()`.

Public, invisible, privé

Théoriquement, il est donc possible de définir quatre types de fonctions : publiques visibles, publiques invisibles, privées visibles et privées invisibles. En pratique, la dernière catégorie ne présente aucun intérêt. On parlera donc de publiques (pour publiques visibles), d'invisibles (pour publiques invisibles) et de privées (pour privées visibles et privées invisibles).

Publique visible	⟶ Publique
Publique invisible	⟶ Invisible
Privée visible	⟶ Privée
Privée invisible	⟶ Privée

Définition de `export`

Un petit exemple va permettre de fixer les choses. On souhaite que les fonctions `publicA` et `publicC` soient publiques ; que `.publicB` soit invisible ; que `privateA`, `.privateB` et `privateC` soient privées.

Pour cela, on écrit dans le fichier `NAMESPACE` :

```
> export("publicA",
+        ".publicB",
+        "publicC"
+ )
```

Ainsi, l'utilisateur peut utiliser `publicA` (publique), `.publicB` (invisible) et `publicC` (publique) mais pas `privateA`, `.privateB` et `privateC` (privées).

A noter que tout cela concerne l'utilisateur et non pas les fonctions : `publicC`, une fonction faite par le programmeur (définie dans le package), peut sans problème utiliser `privateA`. Par contre, les fonctions programmées par l'utilisateur ne pourront pas l'utiliser.

`export` par défaut

Petite digression informatique : une *expression régulière* est la représentation synthétique d'un ensemble de chaînes de caractères ayant des caractéristiques communes. Par exemple, si on considère les dix chaînes `Ind0`, `Ind1`, `Ind2`, ... `Ind9`, il est assez pénible de les noter une par une. Une expression régulière résumant ces dix chaînes sera *"Ind" + un chiffre*. Naturellement, nous donnons ici un exemple illustratif, les vraies expressions régulières utilisent des codifications plus précises.

R et `NAMESPACE` permettent à l'utilisateur de travailler avec des expressions régulières. Cela permet de raccourcir la liste des `export`. Par défaut, `package.skeleton` crée un

`export` utilisant une expression régulière signifiant : *exporte toutes les variables et fonctions qui NE commencent PAS par un point*, autrement dit *n'exporte pas les invisibles*. Pour plus d'information sur les expressions régulières, vous pouvez consulter l'aide en ligne `?regex`.

1.4.2 `import`

En parallèle avec le mécanisme cachant ou dévoilant l'intimité de notre programme, il existe un principe permettant d'aller explorer les cuisines des autres. Cela se fait via l'instruction `import`, située dans le fichier `NAMESPACE`. `import` vient en complément de la ligne `Depends` du fichier `DESCRIPTION`.

Plus précisément, toutes les fonctions définies dans d'autres packages doivent être importées. Pour cela, on peut soit importer le package définissant les fonctions dont on a besoin intégralement, soit importer juste une fonction.

L'import intégral d'un package se fait grâce à l'instruction `import` :

```
### Pour le package integralement
import(graphics)
```

L'import d'une fonction particulière se fait avec `importFrom` en lui précisant le nom du package puis celui de la fonction :

```
### Pour une unique fonction
importFrom(graphics,plot)
```

1.4.3 Bilan

Au final, voila notre fichier `NAMESPACE` :

```
export("publicA",
   ".publicB",
   "publicC"
)

importFrom(graphics,plot)
```

1.5 Préparation de la documentation

L'ordinateur est configuré, le programme tourne, l'arborescence est faite, les fichiers `DESCRIPTION` et `NAMESPACE` sont modifiés. Nous touchons au but, il ne reste *plus que* les docs...

Erreur, grave erreur... car sous le "*plus que*" se dissimulent deux monstres terriblement chronophages : la pédagogie et la lisibilité...

1.5.1 C'est long, c'est fastidieux, ça n'est pas le moment...

Il y a trois grosses difficultés lorsqu'on aborde la rédaction de la documentation :

- C'est long,
- C'est long...
- C'est long et ça tombe au mauvais moment !

Plus précisément, examinons ensemble le cheminement qui aboutit à la création d'un package. D'abord, on a une idée. On commence à la programmer. L'idée évolue, le programme aussi. Il se complexifie. On s'y perd, on le retravaille, on change tout. Il murit. Finalement, après un épuisant travail, il est quasi terminé. On supprime les derniers bugs, on met la petite touche finale. On sent que la fin est proche. On crée les fichiers `DESCRIPTION`, le `NAMESPACE`, l'arborescence. On pense que l'essentiel du travail est derrière nous. On a résolu toutes les difficultés techniques, tout ce qui est intellectuellement stimulant. Reste... la documentation. Á ce stade, sa rédaction apparait comme un petit extra, une partie annexe, un bonus. De plus, notre programme marchera, avec ou sans.

Erreur fatale, la documentation est fondamentale. Elle est fondamentale à au moins deux titres :

- Une documentation est faite pour expliquer aux autres. Elle doit donc être pédagogique. La pédagogie, ça se travaille, ça se pense, ça se relit et ça se fait relire. Et si personne ne comprend rien à ce que vous avez écrit, ça n'est pas parce que les autres sont des tirs au flanc qui ne prennent pas le temps de lire, c'est parce que vous n'êtes pas aussi clair que vous le pensez...
- Si vous faites un package, ça n'est pas pour le plaisir de torturer votre code, c'est pour donner votre travail à la communauté (puis pour avoir la joie de savoir que d'autres utilisent votre travail ce qui en montre la qualité). Hors, il y a assez peu de chance pour que votre package soit unique : il existe des solutions alternatives plus ou moins proches. Si, au détour d'un forum, votre package est cité parmi ceux susceptibles d'aider un utilisateur mais qu'il n'arrive pas à le comprendre, il se tournera vers les autres [4]. Votre package tombera en désuétude, toute votre programmation (qui est peut-être de grande qualité par ailleurs, là n'est pas la question) n'aura servi à rien.

Bilan, une programmeur averti en valant deux, il faut le savoir : rédiger la documentation est une partie longue, fastidieuse, qui arrive au moment où l'on pense avoir enfin fini... mais INDISPENSABLE. Pour vous-même quand vous aurez oublié les détails de votre package ; pour les autres qui aimeraient utiliser votre travail ; pour la qualité des logiciels libres.

4. Si besoin est, les statisticiens changent même de logiciel : par exemple les gens de SAS viennent faire leurs imputations multiples sous R et les gens de R vont faire leur régression polytomique sous SAS ou sous STATA et cela simplement parce que les résultats d'une régression polytomique sous R ne sont pas très clairs... En tout état de cause, il est important de ne pas hésiter à utiliser un autre logiciel si on a des doutes sur une fonction R, même si c'est simplement à titre confirmatoire.

1.5.2 Que faut-il documenter ?

Légitime question : Que faut-il documenter ?

Réponse simple : TOUT ! Et plutôt deux fois qu'une... Cela résume assez bien la pensée de ceux qui ont programmé votre pire ennemi et votre théorique allié, j'ai nommé `R CMD check` et `package.skeleton` (lequel est lequel ? Il faut avouer que parfois, il est difficile de savoir...). Plus précisément, il faut documenter le package dans sa globalité, tous les fichiers de données, toutes les fonctions publiques et toutes les invisibles (Généralement, toutes les invisibles sont documentées dans le même fichier qui a pour titre et pour unique texte *For internal use only...*) Seules les privées sont exemptées.

La documentation doit être rédigée sous forme de fichiers d'aides au format `.Rd`. Les fichiers doivent être enregistrés dans le répertoire `/man/`. Ça permet ensuite à l'utilisateur qui se sert de notre package de faire afficher les différents fichiers d'aide grâce à la commande `help`.

1.5.3 Les alias

Quand l'utilisateur veut de l'aide sur une fonction, il tape généralement `help(fonctionMystere)`. Dans notre cas, il peut vouloir en savoir plus sur `publicA`. Il lui suffit de taper `help(publicA)` (ou `?publicA`) et le fichier d'aide `publicA.Rd` s'ouvre. Petite vérification du travail qu'avaient effectué `package.skeleton` et `prompt` quelques paragraphes plus tôt: dans le répertoire `/packClassic/man/`, on trouve 6 fichiers :

- `dataAge.Rd`
- `packClassic-package.Rd`
- `privateA.Rd`
- `privateC.Rd`
- `publicA.Rd`
- `publicC.Rd`

Les quatre derniers portent les noms de fonctions que nous avons définies. Vous l'aurez compris, des fichiers ont été créés pour toutes les fonctions sauf les invisibles.

Lier un fichier d'aide à une demande d'aide ne se fait pas grâce au nom du fichier mais à l'aide de l'instruction `alias`. La règle est simple :

Le fichier qui doit s'ouvrir lorsque l'utilisateur tape `help(fonctionMystere)`
doit contenir l'instruction `\alias(fonctionMystere)`

Ainsi, comme on veut que `publicA.Rd` s'ouvre lorsque l'utilisateur tape `help(publicA)`, le fichier `publicA.Rd` doit contenir `\alias(publicA)`. Pour vérifier, ouvrons le fichier `publicA.Rd`. Voilà les trois premières lignes :

```
\name{publicA}
\Rdversion{1.1}
\alias{publicA}
...
```

Cette aide s'ouvrira donc à l'appel de `help(publicA)`.

Remarque : `alias` est le seul lien entre un fichier et la commande `help`. En particulier, si l'on renomme un fichier d'aide sans changer son alias, cela ne modifiera pas le comportement du package.

1.5.4 L'alias modifiés

Il est possible d'affiner un peu le processus : étant donné un fichier d'aide `aide.Rd`, on peut spécifier le mot que l'utilisateur doit taper pour l'ouvrir. Cela se fait grâce à la commande `alias`. Si dans le fichier `publicA.Rd` on remplace `\alias(publicA)` par `\alias(superPublicA)`, l'instruction `help(publicA)` produira une erreur alors que `help(superPublicA)` ouvrira le fichier. Cela semble étrange de vouloir faire ce genre de modification. Ça l'est sans doute dans un package classique, ça le sera beaucoup moins dans un package S4.

1.5.5 Les alias multiples

Un fichier peut contenir plusieurs `alias`. Ainsi, si le fichier `publicA.Rd` contient les alias `publicA` et `publicC`, l'utilisateur pourra le visualiser soit en tapant `help(publicA)`, soit `help(publicC)`. Ceci est en particulier utile pour documenter des fonctions qui traitent de sujets proches (comme `numeric`, `as.numeric` et `is.numeric`). Par contre, il pourra être souhaitable de renommer le fichier `publicA.Rd` en `publicAC.Rd` pour bien indiquer qu'il ne correspond plus à une aide unique.

Ainsi, nous pouvons :
- Renommer `publicA.R` en `publicAC.R`
- Supprimer `publicC.R`
- Ajouter `\alias(publicC.R)` au fichier `publicAC.R`.

Notez que réciproquement, un fichier d'aide sans alias ne serait plus accessible (du moins, plus par la fonction `help`).

1.5.6 Les extra

Nous venons de le voir, une aide est liée à un fichier grâce à un alias. Il est donc possible d'ajouter des fichiers d'aide non prévus par `package.skeleton`. Pour cela, il suffit de créer un nouveau fichier `.Rd` et de lui ajouter l'alias désiré. C'est en particulier utile si plusieurs fonctions gravitent autour d'un concept central sans qu'aucune ne l'englobe totalement. On peut alors ajouter une aide `conceptCentral.Rd` qui sera accessible par `help(conceptCentral)`.

1.5.7 \alias{nomDuPackage}

R n'impose pas l'existence de `\alias(nomDuPackage)` [5]. Or, quand un utilisateur vient d'installer un package tout beau tout neuf qu'il ne connait pas, son premier réflexe est de taper `help(nomDuPackage)`. Si cet alias n'existe pas, l'utilisateur ne sait pas par où commencer pour avoir une aide. C'est un peu déroutant.

Cet alias a assez naturellement sa place dans le fichier `nomDuPackage-package`. En effet, ce fichier d'aide a un rôle un peu particulier : il sert à donner une vue globale du package. L'instruction `help(nomDuPackage)` ouvrira donc directement la description générale grâce à cet alias. Notez que ça n'est pas une obligation, rien ne vous empêche de créer un autre fichier d'aide (un extra) et de lui affecter `\alias(nomDuPackage)`.

Mais dans un fichier ou dans l'autre, cet alias doit exister parce qu'il est vraiment désagréable, lorsqu'on vient de télécharger `nouveauSuperPackage` et qu'on s'attelle à la lourde tâche de comprendre comment il fonctionne, de ne pas recevoir d'aide après avoir tapé `help(nouveauSuperPackage)`.

1.5.8 Internal

Pour finir, `package.skeleton` génère le fichier `packClassic-internal.Rd` seulement si l'option `namespace=FALSE`. Ce fichier est là pour opposer une fin de non recevoir aux curieux qui voudraient des informations sur les fonctions invisibles. Il est poli, mais ferme : "These are not to be called by the public"...

En fait, il fait un peu double emploi avec `NAMESPACE` :

- Soit le programmeur cache ses fonctions en les rendant invisibles (avec des noms commençant par des points) : dans ce cas, R exige des documentations pour les invisibles, mais comme le programmeur estime que l'utilisateur n'en a pas besoin, il redirige les demandes d'aide vers `packClassic-internal.Rd` (en ajoutant dans ce fichier les alias de toutes les fonctions invisibles).
- Soit le programmeur cache ses fonctions en les rendant privées (grâce à `NAMESPACE`) auquel cas il n'a plus besoin de les documenter.

1.5.9 Effacer les docs des privés

`package.skeleton` ne sait pas que les fonctions `private` sont privées, il a donc créé des fichiers d'aide pour chacune d'entres elles. Ces fichiers sont inutiles, les privés n'ont pas besoin d'être documentés. Nous pouvons donc supprimer `privateA.Rd` et `privateC.Rd`.

1.5.10 Bilan

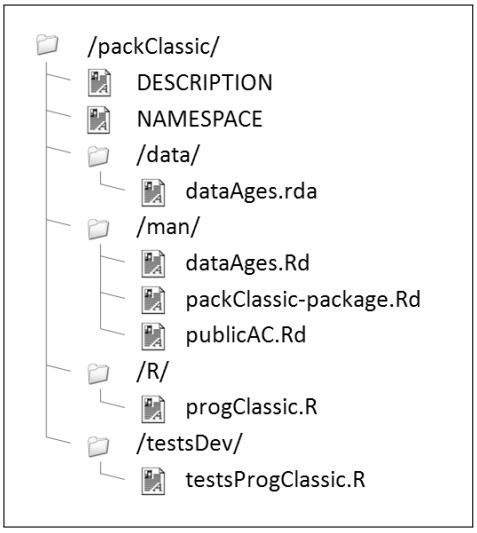
Au final, l'aide se construit de la manière suivante :

- Ajouter dans `NAMESPACE` le nom des fonctions publiques et invisibles.

5. Depuis, la version R 2.9.0, cet alias est inséré dans `nomDuPackage-package`, mais il n'est toujours pas obligatoire.

– Documenter les publiques, éventuellement en regroupant certaines dans un même fichier.
– Documenter les fichiers de données.
– Documenter le package dans son ensemble.
– Le cas échéant, documenter les invisibles dans un unique fichier qui ne donnera pas d'information.

Appliqué à notre package `packClassic`, cela donne :



1.6 Rédaction des documentations

Il est maintenant temps d'éditer les fichiers d'aide et d'adapter les fichiers créés par `package.skeleton` à nos besoins. La rédaction des documentations se fait dans un langage *interprété* (assez proche de LaTeX). Le texte est tapé tel quel et est égayé instructions spéciales précédées de \ et permettant la mise en forme du document.

Les textes saisis doivent obligatoirement figurer dans une rubrique (dans `\detail{ }`, dans `\author{ }`,...) Le texte `%% ~Make other sections like Warning with \section{Warning }{....} ~` est une entorse à cette règle, il faut donc le supprimer, sinon le fichier ne sera pas syntaxiquement correct. Avant de présenter les différentes rubriques que doivent comprendre les fichiers d'aides, quelques remarques sur la syntaxe générale.

1.6.1 Syntaxe générale

Connaitre la syntaxe de trois ou quatre instructions suffit pour la majorité des aides :
– Pour mettre du code informatique en valeur, il faut utiliser `\code{mon code}`

- L'instruction `\link` permet de rendre un mot cliquable, pour que l'utilisateur puisse aller sur une autre page d'aide. `\link{AutreAide}` affiche le mot `AutreAide` et redirige vers le fichier `AutreAide.Rd`.
- L'instruction `\link[=fichierAide]{motClef}` affiche `motClef` et redirige vers le fichier `fichierAide.Rd`, à condition toutefois que celui-ci fasse partie du package. Pour une aide dans un autre package, il faut utiliser l'instruction `\link[nomAutrePackage]{motClef}` pour afficher `motClef` et ouvrir le fichier `motClef.Rd` ; `\link[nomAutrePackage:fichierAide]{motClef}` pour afficher `motClef` et ouvrir le fichier `fichierAide.Rd`.
- Pour faire une liste à puce, il faut utiliser la structure `\itemize{` pour commencer la liste. Puis pour chaque puce, on ajoute `\item texte associé`. Enfin, `}` cloture la liste.
- Pour faire une liste numérotée, on remplace `\itemize{` par `\enumerate{`.
- Pour faire une liste avec des items "només", il faut utiliser la structure `\describe{` pour commencer la liste. Puis pour chaque item, on ajoute `\item{item 1}{texte associé}`, `\item{item 2}{texte associé 2}`... Enfin, comme dans les cas précédents, `}` cloture la liste.

De nombreux exemples illustratifs sont présentés dans les pages suivantes.

1.6.2 Généralités pour toutes les documentations

Pour être correcte, une aide doit obligatoirement comporter un certain nombre de rubriques. Voilà les rubriques communes à toutes les aides :

- L'`\alias`, nous en avons déjà longuement parlé, définit l'instruction qui permet d'ouvrir le fichier d'aide.
- Le `\title` sert à décrire l'aide. Autre utilité, lorsqu'un utilisateur fait une recherche avec `help.search` sur un mot clef, la liste des packages et fonctions qui se rapportent à son mot clef est affichée. Chaque fonction est assortie d'un court descriptif. Ce descriptif, le programmeur le spécifie via `\title` de l'aide associé à la fonction.
- Les `\references` servent à citer des articles ou des livres qui ont servi de support à la création du package.
- `\seealso` permet de rediriger vers d'autres aides qui éventuellement complètent celle que nous sommes en train d'écrire.
- `\examples` contient un mini programme R illustrant l'aide que l'on est en train d'écrire. Il est appelé lorsque l'utilisateur utilise l'instruction `example` Ce code doit être directement exécutable sans modification. Dans le cas de la présentation globale du package, il doit reprendre les fonctions les plus importantes. Dans le cas de fonctions, il doit présenter différentes manières d'utiliser la fonction.
- `\keyword` permet de définir des mots clefs. Quand un utilisateur qui ne connait pas l'existence de votre package fait une recherche sur un concept, il le fait en saisissant un mot. Si le mot correspond à un de vos mots clef, votre package sera signalé à l'utilisateur comme étant potentiellement intéressant pour lui. La liste des mots clefs autorisés est donnée en annexe **??** page **??**.

1.6.3 Présentation générale du package

La présentation générale de notre package se fait dans le fichier d'aide `packClassic-package.Rd`. Par rapport à ce qui précède, on doit ajouter `\details`.

– `\details` reprend les informations du fichier `DESCRIPTION`.

On obtient le fichier :

```
\name{packClassic-package}
\Rdversion{1.1}
\alias{packClassic-package}
\alias{packClassic}
\docType{package}

\title{Toy example of a classic package}

\description{
  This package is a toy example build to illustrate the
  construction of a package as explain in the tutorial
  \emph{Petit Manuel de Programmation Orientee Objet en R}.
}

\details{
  \tabular{ll}{
    Package: \tab packClassic\cr
    Type: \tab Package\cr
    Version: \tab 0.5\cr
    Date: \tab 2009-05-23\cr
    License: \tab GPL(>=2.0)\cr
    LazyLoad: \tab yes\cr
  }
  This package is a toy example build to illustrate the
  construction of a package as explain in the tutorial
  \emph{Petit Manuel de Programmation Orientee Objet en R}.
  There is mainly two functions. There are documented in
  \link[packClassic]{publicAC}.
  May be there is another one, but it is a secret...
}

\author{Christophe Genolini <genolini@u-paris10.fr>}

\references{
  Book: "Petit Manuel de Programmation Orientee Objet sous R"
}

\seealso{
  \code{\link[packS4:packS4-package]{packS4}} is another toy
  example build to illustrate the construction of an S4 package.
}

\examples{
### PublicC is the main fonction of this package
publicC(3)
}
```

```
\keyword{package}
```

1.6.4 Fonctions

Dans les fichiers d'aide expliquant les fonctions, il faut également détailler l'utilisation de la fonction et les arguments utilisés :

- `\usage` présente la manière d'appeler la fonction, en particulier la liste des arguments et éventuellement les valeurs par défaut de certains d'entre eux. La liste des arguments doit être rigoureusement identique à celle de la définition de la fonction.
- `\arguments` détaille la liste des arguments en précisant les valeurs qu'ils doivent prendre et leur utilité.
- `\value` précise le type de ce qui est retourné par la fonction (un `numeric`, un `character`, une `list`...)

Pour le fichier `publicAC.Rd`, on obtient :

```
\name{publicAC}
\Rdversion{1.1}
\alias{publicA}
\alias{publicC}

\title{publicA and publicC}

\description{
  Two great function that do very simple mathematical operation.
}

\usage{
  publicA(x)
  publicC(x)
}

\arguments{
  \item{x}{\code{x} is a numeric}
}

\details{
  So simple, no details are needed.
}

\value{A numeric}

\references{
  Book: "Petit Manuel de Programmation Orientee Objet sous R"
}

\author{Christophe Genolini <genolini@u-paris10.fr>}

\examples{
publicA(3)
```

```
publicC(4)
}
\keyword{documentation}
```

1.6.5 Données

Enfin, dans les fichiers d'aide pour les données, il faut ajouter `\format`.
– `\format` détaille chaque colonne.

Pour le fichier `dataAges.Rd`, on obtient :

```
\name{dataAges}
\Rdversion{1.1}
\alias{dataAges}
\docType{data}

\title{Toy data frame for packClassic}

\description{
  This data.frame is a fake toy example made up to illustrate the
  inclusion of data in a package.
}

\usage{data(dataAges)}

\format{
  A data frame with 5 observations on the following 2 variables.
  \describe{
    \item{\code{sex}}{a factor with levels \code{F} \code{H},
        which denote the gender of the subject}
    \item{\code{age}}{a numeric vector for teh age.}
  }
}

\details{
  So simple, no detail are needed.
}

\source{Fake data.}

\references{
  Book: "Petit Manuel de Programmation Orientee Objet sous R"
}

\examples{
data(dataAges)
str(dataAges)
}

\keyword{datasets}
```

1.7 Finalisation et création

Lorsque les fichiers sources (programme, aides et données) sont prêts, il reste à les vérifier, à construire le package et éventuellement à construire des documentations au format `pdf`. Ces trois étapes se font dans une fenêtre de commandes `DOS` sur Windows ou une fenêtre terminal sur `Linux`.

1.7.1 Vérification

Pour vérifier le package, il faut ouvrir une fenêtre de commande DOS ou une fenêtre terminale Linux, se placer dans le répertoire contenant le répertoire du package (pas dans le répertoire portant le nom package, mais un niveau au dessus ; dans notre cas, il s'agit du répertoire `/TousMesPackages/`). Si vous avez créé un ficher `Rpath.bat` comme conseillé section 1.1 page 5, il vous faut taper la commande `C:/Rpath` pour modifier le path. Ensuite, la vérification de votre package se fait avec l'instruction : `R CMD check monPackage`

`R CMD check` classe les problèmes en deux catégories : les erreurs et les avertissements (warnings). Les erreurs sont fatales, elles indiquent qu'un problème empêchera la construction future de votre package. Entrent dans cette catégorie les erreurs de programmation, les erreurs de syntaxe dans les fichiers DESCRIPTION, NAMESPACE ou dans les aides et bien d'autres encore. Quand il détecte une erreur, `R CMD check` la signale puis s'arrête.

Les avertissements sont moins graves : ils signalent des problèmes qui n'empêcheront pas la construction du package mais plutôt sa bonne utilisation. Par exemple, si les fichiers d'aide ne sont pas tous présents, si des alias existent en double, si des attributs obligatoires manquent, cela donnera lieu à des avertissements. Il sera donc possible de construire votre package. Par contre, un tel package sera refusé sur le site du CRAN. La R Core Team n'accepte en effet que les packages sans erreur ni avertissement. Pour la petite histoire, il existe des packages "dissidents" de personnes estimant qu'elles n'ont pas à corriger les avertissements et qui diffusent leur package via des sites personnels... Mais cela est un peu excessif, la correction des avertissements se fait assez rapidement.

Enfin, il peut arriver que la vérification d'un package génère des avertissements sur un système d'exploitation mais pas sur l'autre. Si vous travaillez sur Linux, vous n'avez pas à vous en soucier puisque la R Core Team vérifie les package à partir de Linux et gèrera elle-même l'export vers windows. L'inverse est plus problématique, mais est heureusement très rare.

Si vous voulez en savoir plus sur les options, une aide en ligne est disponible via `R CMD check -help`.

1.7.2 Construction (ou compilation)

Après la vérification d'un package, on peut passer à sa construction proprement dite. Cette dernière étape se fait à l'aide de la commande `R CMD build <option> monPackage`. Les options dépendent de la plateforme informatique sur laquelle on travaille et

de la plateforme à laquelle est destiné le package.

Pour ce qui est d'une soumission d'un package au CRAN, la R Core Team n'accepte que les packages Linux et se charge de les compiler sous windows. Trois types de compilations sont donc nécessaires :

- Windows → Windows : vous travaillez sous Windows et vous voulez compiler votre package pour vous même.
- Windows → Linux : vous travaillez sous Windows et vous voulez compiler votre package pour soumission à la R Core Team.
- Linux → Linux : vous travaillez sous Linux et vous voulez compiler votre package pour vous même ou pour soumission à la R Core Team.

Sur Windows pour Windows : `R CMD build -binary monPackage`

Cette commande crée un package binaire précompilé (pre-compiled binary package) zippé. Ce fichier se nomme `monPackage_version.gz` où version est le numéro de la version inscrit dans le fichier `DESCRIPTION`. Il permet l'installation du package sous Windows.

Sur Windows pour Linux : `R CMD build monPackage`

Cette commande crée un fichier nommé `monPackage_version.tar.gz` comprenant les fichiers sources (c'est à dire les fichiers originaux) du package. Au passage, le fait que les sources soient inclues permet à celui qui reçoit votre fichier de le *recompiler*, y compris de le recompiler pour un autre système que Linux. C'est en particulier ce que fait la R Core Team : à partir de vos sources, ils compilent un fichier pour Windows, un fichier pour Linux et un fichier pour Mac.

Sur Linux pour Linux : `R CMD build monPackage`

C'est la même commande que Windows pour Linux. Elle crée un fichier nommé `monPackage_version.tar.gz` comprenant également les fichiers sources du package.

Sur Linux pour Windows

Il s'agit de la construction la plus compliquée à effectuer. Heureusement, rares sont ceux qui en ont besoin puisque la création d'un package pour Windows à partir de Linux est assurée par la R Core Team à partir des sources. Pour ceux qui voudraient quand même faire la conversion, nous vous renvoyons encore une fois à l'incontournable tutorial de Sophie Baillargeon [1].

Si vous voulez en savoir plus sur les options, une aide en ligne est disponible via `R CMD build -help`.

1.7.3 Documentation

Enfin, on peut créer un fichier documentation au format `pdf` avec la commande : `R CMD Rd2dvi --pdf monPackage`. Sur Windows, le document `Rd2.pdf` est alors créé

dans le répertoire `./Rd2dv/`. Sur Linux, le document `monPackage.pdf` est créé dans le répertoire du package.

1.7.4 upload

Quand tout est terminé, il vous faut soumettre votre package. Cela se fait grâce à un *dépôt* (un *upload*, c'est à dire l'inverse d'un *téléchargement* ou *download*) sur le site du CRAN. Pour cela, il vous faut ouvrir une connexion `ftp` vers l'adresse `ftp://cran.r-project.org/incoming/` puis déposer votre package compilé pour Linux (c'est à dire le fichier `monPackage_version.tar.gz`) dans le répertoire `incoming`.

Le lendemain, vous recevrez un petit mail disant :

```
Dear package maintainer,

this notification has been generated automatically.
Your package packClassic_0.5.tar.gz has been built for Windows
and will be published within 24 hours in the corresponding CRAN
directory (CRAN/bin/windows/contrib/2.9/).
R version 2.9.0 Patched (2009-04-27 r48414)

All the best,
Uwe Ligges
(Maintainer of binary packages for Windows)
```

Fini !

Merci Uwe !

Enfin, ça, c'est dans le meilleur des mondes. Dans la réalité, vous allez probablement recevoir un message vous informant que vous avez oublié de corriger tel et tel warnings, que vous avez une version de R un peu vieillote, que tel fichier d'aide est mal fait...

Il vous faudra donc reprendre, corriger, modifier et ressoumettre. Et recommencer jusqu'à recevoir le tant attendu message. Et là, ça sera vraiment fini. Au moins pour le cas d'un package classique. Reste le passage à la S4...

Chapitre 2

Création d'un package en S4

Les six étapes de la création sont les mêmes que pour un package classique.

2.1 Configuration de votre ordinateur

Rigoureusement identique à la création d'un package classique.

2.2 Création d'un programme... fonctionne !

Les règles définies section 1.2.1 s'appliquent toujours mais le découpage du programme est un peu plus complexe.

2.2.1 Quelques règles de plus

Nous l'avons brièvement évoqué dans l'introduction, l'encapsulation est le fait de rassembler à un seul endroit le code définissant un objet, ce qui permet de ne plus avoir à s'occuper de ses méandres internes lorsqu'il est terminé. R ne force pas le programmeur à encapsuler ses objets, mais il est possible d'encapsuler malgré tout. Pour cela, il suffit de suivre une règle toute simple :

Á chaque objet son fichier

De même, chaque objet doit être testé. D'où une deuxième règle tout aussi simple que la première :

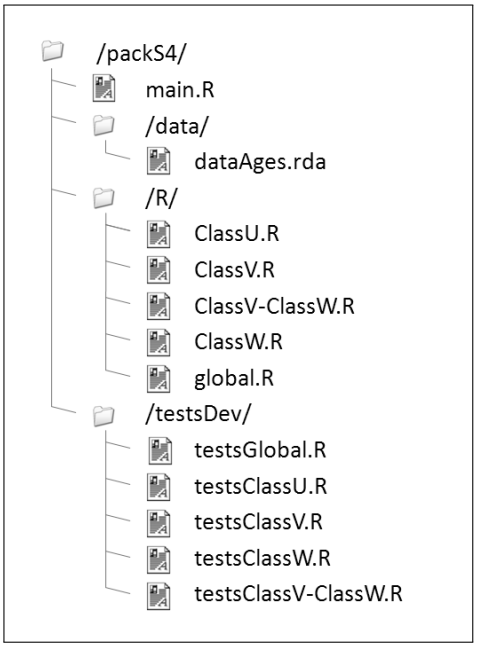
Á chaque fichier de code son fichier test.

Il va donc nous falloir créer un fichier et un fichier test par objet. En outre, à ces fichiers vont venir s'ajouter divers fichiers auxiliaires. Prenons un exemple, là encore artificiel et simple. Son nom est `packS4`. Il est constitué de trois classes `ClassU`, `ClassV` et `ClassW`. Pour cela, il nous faut créer les fichiers suivants :

- `global.R` : Même si notre programme est orienté objet, il arrive que nous fassions appel à des fonctions classiques non S4 et n'appartenant à aucune classe en particulier. Ces fonctions ne peuvent pas être placées dans le fichier associé à une classe et doivent donc être déclarées dans un fichier à part. Nous les mettons dans `global.R` lui-même dans `/packS4/R/`.
  `global.R` contient également tous les `setGeneric` (nous avions vu section **??** qu'il valait mieux les regrouper dans un même fichier et non les inclure dans une classe particulière).
- `ClassU.R`, `ClassV.R` et `ClassW.R` : *A chaque classe son fichier.* Les fichiers définissant les classes sont dans le répertoire `/packS4/R/`.
- `ClassV-ClassW.R` : Exception, il peut arriver qu'une méthode dépende de deux classes, `ClassV` et `ClassW`. Si cette méthode appartient à la classe `ClassV` mais qu'il faut définir `ClassW` *après* `ClassV`, alors la méthode mixte ne peut pas être testée lors de sa création (car `ClassW` n'existe pas encore). Dans ce cas, il est préférable de définir les méthodes mixtes dans un fichier à part (petite entorse à la règle *Á chaque objet son fichier*) que l'on appellera `ClassV-ClassW.R` et qui sera exécuté après `ClassV.R` et après `ClassW.R`. Il sera dans `/packS4/R/`.
- `testsClassU.R`, `testsClassV.R`, `testsClassW.R` et `testsClassV-ClassW.R` : *Á chaque fichier de code, son fichier test.* Pour chaque fichier présent dans `/packS4/R/`, on crée un fichier test que l'on place dans `/packS4/testsDev/` [1].
- `dataAges.rda` : pas de changement par rapport au cas classique, les fichiers de données sont dans le répertoire `/data/`.
- `main.R` : enfin, un fichier `main.R` appelle tous les fichiers tests. Il ne fait pas vraiment partie du package, il permet juste de l'exécuter pendant sa conception. Il ne faut donc pas l'inclure dans `/packS4/R/` puisqu'il ne définit pas de fonctionnalité du langage. On peut le mettre dans le répertoire `/packS4/`.

On a donc :

1. Comme pour le package classique, ce répertoire n'est à utiliser que si vous choisissez de suivre la gestion des tests "alternative et manuelle" telle que nous l'avons définie section 1.2.2 page 7. Pour la gestion automatique et officielle de la R Core Team, utilisez le répertoire `/packS4/tests/`.



2.2.2 Les tests

Mêmes règles que dans le cas classique. Là encore, chaque fichier test `testsClassU.R` doit commencer par l'inclusion du programme associé : `source("../R/ClasseU.R")`.

Il est possible de concevoir les fichiers tests comme une suite de fichiers appelés les uns après les autres. Par exemple, on peut décider que `testsClassV.R` sera toujours exécuté après `testsClassU.R`. Cela permet de considérer que les objets définis dans `testsClassU.R` seront utilisables dans `testsClassV.R`. Ce genre d'imbrication est facilement gérable grâce au fichier `main.R`. L'intérêt est de ne pas avoir à reconstruire de but en blanc les jeux de données tests pour chaque fichier mais de pouvoir réutiliser ce qui a été défini avant.

2.2.3 Toy exemple S4

Reprenons notre exemple. Il est constitué de trois classes `ClassU`, `ClassV`, `ClassW` et de données `dataAges`. `ClassW` hérite de `ClassU`. Nous souhaitons que l'utilisateur puisse avoir accès à la classe `ClassW` intégralement, à la classe `ClassV` partiellement (certaines méthodes mais pas toutes) et qu'il n'ait pas accès à `ClassU`. Le code complet (tests inclus) est téléchargeable sur le site web du livre [2].

```
################################################################################
###                                  main.R                                  ###
################################################################################

load("./data/dataAges.rda")
```

2. `http://christophe.genolini.free.fr/webTutorial`.

```
source("./testsDev/testsGlobal.R")
source("./testsDev/testsClassU.R")
source("./testsDev/testsClassV.R")
source("./testsDev/testsClassW.R")
source("./testsDev/testsClassV-ClassW.R")
```

```
##############################################################################
###                                global.R                                ###
##############################################################################

setGeneric("privateA",
    function(object){standardGeneric("privateA")}
)
setGeneric("publicA",
    function(object){standardGeneric("publicA")}
)
setGeneric("publicB",
    function(objectV,objectW){standardGeneric("publicB")}
)
functionClassicA <- function(age){return(age*2)}
```

```
##############################################################################
###                                ClassU.R                                ###
##############################################################################

setClass("ClassU",representation(u1="numeric",u2="numeric"))
setMethod("privateA","ClassU",function(object){object@u1^2})
setMethod("plot","ClassU",function(x,y){barplot(c(x@u1,x@u2))})
```

```
##############################################################################
###                                ClassV.R                                ###
##############################################################################

setClass("ClassV",representation(v1="numeric",v2="numeric"))
setMethod("privateA","ClassV",function(object){object@v1^3})
setMethod("publicA","ClassV",function(object){sqrt(object@v2^3)})
setMethod("plot","ClassV",function(x,y){barplot(c(x@v1,x@v2^2))})
```

```
##############################################################################
###                                ClassW.R                                ###
##############################################################################

setClass(
    "ClassW",
    representation(w1="character"),
    contains="ClassU"
)
classW <- function(u1,u2,w1){new("ClassW",u1=u1,u2=u2,w1=w1)}
setMethod("publicA","ClassW",function(object){sqrt(object@u2^5)})
setMethod("plot","ClassW",
    function(x,y){barplot(c(x@u1,x@u2),main=x@w1)}
)
```

```
setMethod("[","ClassW",
    function(x,i,j,drop){
        switch(EXP=i,
            "u1"={return(x@u1)},
            "u2"={return(x@u2)},
            "w1"={return(x@w1)}
        )
    }
)

setReplaceMethod("[","ClassW",
    function(x,i,j,value){
        switch(EXP=i,
            "u1"={x@u1<-value},
            "u2"={x@u2<-value},
            "w1"={x@w1<-value}
        )
        validObject(x)
        return(x)
    }
)
```

```
################################################################################
###                              ClassV-ClassW.R                             ###
################################################################################
setMethod("publicB",c("ClassV","ClassW"),
    function(objectV,objectW){objectV@v1*objectW@u1}
)
```

2.3 Création de l'arborescence et des fichiers

2.3.1 `package.skeleton`

Quasi identique à ce que nous avons utilisé pour un package classique, il faut néanmoins donner à `package.skeleton` un peu plus d'arguments. En effet, le code source a été séparé en cinq fichiers. Il faut donc les préciser dans `code_file`. Comme dans le cas classique, il faut définir le répertoire `/TousMesPackage/` comme répertoire courant de votre console R.

```
> package.skeleton("packS4",namespace=TRUE, force=TRUE,
+     code_file=c(
+         "packS4/R/global.R",
+         "packS4/R/ClassU.R",
+         "packS4/R/ClassV.R",
+         "packS4/R/ClassW.R",
+         "packS4/R/ClassV-ClassW.R"
+     )
+ )
```

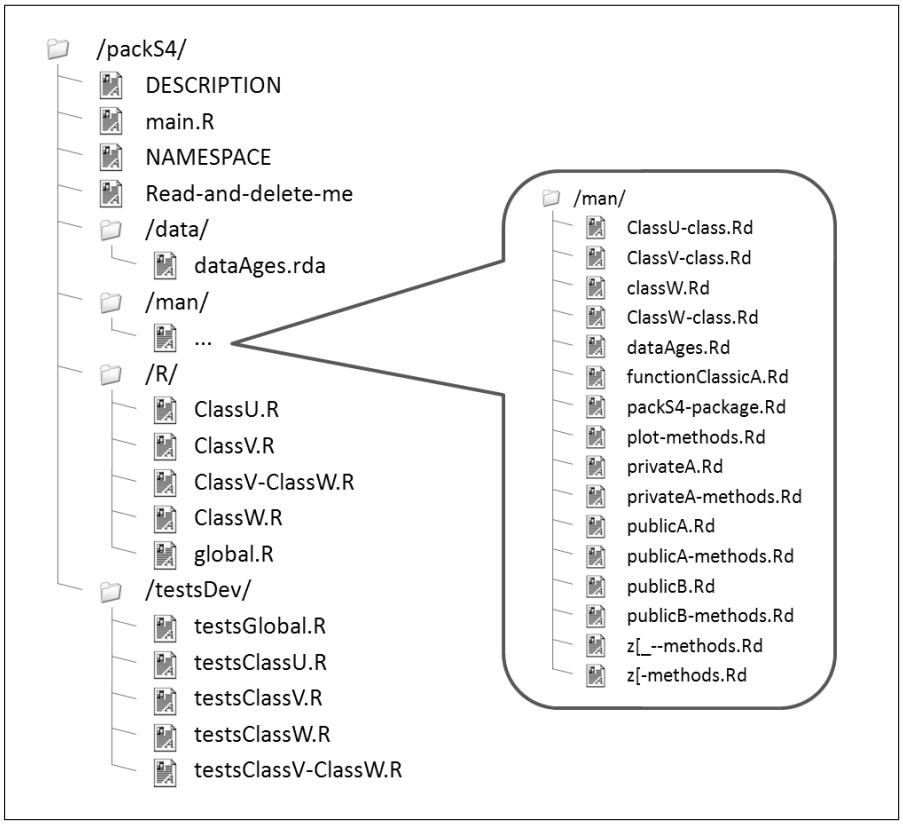
Par défaut, `package.skeleton` a tendance à créer beaucoup d'alias (dont certains en double, comme nous le verrons en détail section 2.5.4), mais ne crée pas d'aide pour les données.

2.3.2 Aides pour les données

Comme dans la création de package classique, il faut donc les ajouter à la main avec `prompt`. L'instruction est la même :

```
load("packS4/data/dataAges.rda")
prompt(dataAges,filename="packS4/man/dataAges.Rd")
```

Á ce stade :



2.3.3 `DESCRIPTION`

Même chose que dans le cas classique. Dans le cas présent, nous avons besoin de préciser l'ordre d'exécution des différents fichiers : `global.R` définit les génériques, donc il doit être lu en premier ; puis la définition des 3 classes ; et enfin, les méthodes communes à deux classes. Pour mémoire, cela se fait grâce à l'instruction `Collate`. Au final, le fichier description est :

```
Package: packS4
Type: Package
Title: Toy example of S4 package
Version: 0.5
Date: 2009-05-26
Author: Christophe Genolini
Maintainer: <genolini@u-paris10.fr>
Description: This package comes to illustrate the book
  "Petit Manuel de Programmation Orientee Objet sous R"
License: GPL (>=2)
LazyLoad: yes
Depends: methods, graphics
URL: www.r-project.org, christophe.genolini.free.fr/webTutorial
Collate: global.R ClassU.R ClassV.R ClassW.R ClassV-ClassW.R
```

2.4 Visibilité des classes et des méthodes (NAMESPACE)

Là encore, on distingue les `export` des `import`.

2.4.1 `export`

Notre package comporte trois types de définitions :
- Les fonctions classiques,
- Les classes,
- Les méthodes.

Chacune de ces catégories s'exporte en utilisant une instruction spécifique :
- **Les fonctions classiques** s'exportent comme dans le cas classique en utilisant `export`.
- **Les classes** s'exportent grâce à `exportClasses`
- **Les méthodes** s'exportent, oh ! surprise, via `exportMethods`

Dans notre cas, nous voulons exporter les classes `ClassV` et `ClassW`, mais pas `ClassU`. D'où :

```
exportClasses(
    "ClassV",
    "ClassW"
)
```

De même, nous voulons exporter la méthode `publicA` de `ClassV`, les méthodes `publicA`, `[` et `[<-` de `ClassW`.

```
exportMethods(
    "[",
    "[<-",
    "plot",
    "publicA",
    "publicB"
```

```
)
```

Enfin, il nous faut exporter les fonctions classiques non S4 :

```
    "classW",
    "functionClassicA"
)
```

Á noter, `plot` a un statut un peu particulier : c'est à la fois une fonction classique et une fonction S4 (pour des soucis de compatibilité). Il est donc possible de l'exporter soit dans `export`, soit dans `exportMethods`.

2.4.2 `import`

Le fonctionnement est le même que dans le cas classique.

2.4.3 Bilan

Au final, voilà notre fichier `NAMESPACE` :

```
export(
    "classW",
    "functionClassicA"
)
exportMethods(
    "[",
    "[<-",
    "plot",
    "publicA",
    "publicB"
)
exportClasses(
    "ClassV",
    "ClassW"
)

import(graphics)
```

2.5 Préparation de la documentation

Lors de l'appel de `package.skeleton`, un certain nombre de fichiers de documentation et d'alias ont été créés. Tous ne sont pas nécessaires, il va falloir faire le tri. Mais avant tout, quelques détails sur l'utilisation de l'aide en S4.

2.5.1 Aide en S4

Par rapport à la programmation classique, la programmation S4 ajoute trois types d'outils (classes, méthodes génériques, méthodes spécifiques). Á chaque outil, elle associe

un fichier d'aide. Ainsi, on trouve des aides pour les classes, des aides pour les méthodes génériques et des aides pour les méthodes spécifiques.

- Afficher l'aide pour une **classe** se fait en appelant le nom de la classe suivi de `-class`. Exemple, l'aide sur `ClassU` se fait grâce à `help(ClassU-class)`.
- L'aide pour une **méthode générique** se fait en appelant le nom de la méthode. Par exemple, pour connaitre le fonctionnement général de `publicA`, on tape `help(publicA)`
- Afficher l'aide de **méthodes spécifiques** se fait en appelant le nom de la méthode suivi de `-methods` (avec un "s"). Par exemple, pour connaitre *les* fonctionnements spécifiques de `publicA`, on tape `help(publicA-methods)`.
- Afficher l'aide **d'une méthode spécifique** se fait en appelant le nom de la méthode suivi de la signature de la méthode puis `-method` sans "s". Par exemple, connaitre *le* fonctionnement spécifique de `publicA` appliqué à la signature `ClassV,ClassW` se fait avec `help(publicA,ClassV,ClassW-method)`.

2.5.2 Fichiers et alias créées par package.skeleton

L'utilisation de `pacakge.skeleton` crée les fichiers et les alias suivants :

1. **Nom de package** : Le fichier `packS4-package.Rd` est créé pour l'ensemble du package (et donc en un seul exemplaire). Il contient `\alias(packS4)` et `\alias(packS4-package)`. Ce fichier est le fichier qui est ouvert quand l'utilisateur tape `help(packS4)`. Il a pour vocation de décrire le package dans sa globalité.
2. **Classes** : pour chaque classe, un fichier `ClassU-class.Rd` est créé. Il contient deux groupes d'alias : un alias pour la classe propre `\alias(ClassU-class)`. Également, pour toutes les méthodes `publicA` de la classe `ClassU`, il contient un alias vers la méthode spécifique `\alias(publicA,ClassU-method)`, et cela pour toutes les signatures possibles.
3. **Méthodes génériques** : pour chaque générique défini dans notre package, un fichier `publicA.Rd` est créé. Il contient `\alias(publicA)`. Les méthodes génériques qui sont utiliséees dans notre package mais qui sont définies dans d'autres packages (comme `plot`, `print`, `[<-`,...) ne donnent pas lieu à la création d'un fichier générique.
4. **Méthodes spécifiques** : pour chaque méthode utilisée par une classe, un fichier `publicA-methods.Rd` est créé. Il contient un alias vers l'ensemble des méthodes spécifiques `\alias(publicA-methods)` plus un alias pour toutes les signatures possibles de la méthode `publicA,ClassU-methods`.
5. **Fonctions classiques** : enfin, un fichier `functionClassic.Rd` est créé pour chaque fonction classique (non S4). Il contient `\alias(functionClassic)`

On constate que dans son désir d'être exhaustif, `package.skeleton` a créé certains alias deux fois : `help(publicA,ClassU-method)` renvoie vers les aides `ClassU-class.Rd` et `publicA-methods.Rd`. Il va nous falloir choisir lequel concerver et lequel supprimer.

Pour l'instant, voilà l'état de notre package `packS4` :

Fichier	Alias
ClassU-class.Rd	ClassU-class plot,ClassU-method privateA,ClassU-method
ClassV-class.Rd	ClassV-class plot,ClassV-method privateA,ClassV-method publicA,ClassV-method publicB,ClassV,ClassW-method
classW.Rd	classW
ClassW-class.Rd	ClassW-class [,ClassW-method plot,ClassW-method publicA,ClassW-method publicB,ClassV,ClassW-method
dataAges.Rd	dataAge
functionClassicA	functionClassicA
packS4-package.Rd	packS4-package packS4
plot-methods.Rd	plot-methods plot,ANY-method plot,ClassU-method plot,ClassV-method plot,ClassW-method
privateA.Rd	privateA
privateA-methods.Rd	privateA-methods privateA,ClassU-method privateA,ClassV-method
publicA.Rd	publicA
publicA-methods.Rd	publicA-methods publicA,ClassV-method publicA,ClassW-method
publicB.Rd	publicB
publicB-methods.Rd	publicB-methods publicB,ClassV,ClassW-method
z[_--methods.Rd	[<--methods [<--,ClassW-method
z[-methods.Rd	[-methods [,ClassW-method

Ensuite, comme dans le cas classique, chaque fichier doit être édité et modifié. Quel travail de titan, seriez-vous en droit de vous exclamer ! Et vous auriez raison... Et encore, il ne s'agit là que d'un toy example, la réalité est bien pire ! Vous êtes encore là ? Heureusement, il va être possible de faire quelques fusions et simplifications. Mais ne vous y trompez pas, nous ne le répèterons jamais assez, la rédaction d'une documentation claire et compréhensible est un travail chronophage... totalement indispensable mais chronophage. La section 1.5.1 page 19 détaille ce point.

Avant de poursuivre, deux remarques : en premier lieu, un nom de fichier ne peut pas commencer par le symbole `[`. Le fichier `[-methods.Rd` est donc renommé en `z[_methods.Rd`. De même, tous les noms de fichiers non compatibles selon R sont renommés : les caractères "fautifs" sont remplacés par `_` et le préfixe `z` est ajouté au nom du fichier. Par exemple, le fichier d'aide de la méthode de remplacement n'est pas `[<--methods` mais `z[_--methods.Rd`.

Deuxième point, `method(s)` est parfois au singulier, parfois au pluriel. Cela permet de distinguer si on est en train de parler d'une méthode générique (`methods` est alors au pluriel), de plusieurs méthodes spécifiques (pluriel) ou d'une seule méthode spécifique (singulier).

2.5.3 Intuitivement : de quoi a-t-on besoin ?

Laissons un instant `package.skeleton` ; imaginons que vous ayez besoin d'un package qui traite de trajectoires et vous ne connaissez rien à la S4. Sur un forum, on vous conseille `miniKml` en particulier sa fonction `impute`. Vous le téléchargez. Ensuite, vous allez essayer d'apprendre à vous en servir. Intuitivement, quelles sont les aides que vous allez demander ?

Reprenons maintenant nos casquettes de programmeur. Quelles sont les aides que nous allons programmer ? Simple, celles dont nous aurions eu besoin en temps qu'utilisateur ne connaissant rien à la S4...

En S4, il existe des méthodes précises et bien pensées pour appeler des aides sur des méthodes ou sur des classes, comme nous l'avons vu section 2.5.1 page 38. Mais la majorité des utilisateurs ne connaissent rien à la S4 et ils ne savent même pas qu'ils sont en train d'utiliser des objets. Le non-initié à la S4 ne connait qu'un seul type d'aide, la fonction help de base. Cela signifie que s'il cherche de l'aide sur la classe `Trajectories`, il tapera `help(Trajectories)`. S'il cherche de l'aide sur la méthode `impute`, il tapera `help(impute)`. Il ne cherchera pas à faire de différence entre une classe, une méthode générique, une spécifique et une fonction classique.

D'un autre côté, il est important de donner aussi une aide à l'expert, celui qui sait qu'il est en train de manipuler de l'objet. Il va donc nous falloir jongler entre ceux qui ne savent pas et pour lesquels il faut fournir des aides intuitives, et ceux qui savent pour lesquels il faut fournir des aides prévus par la S4...

Pour cela, nous conseillons d'ajouter, en plus des alias prévu par la R Core Team, des alias plus intuitifs, pour "non initiés". Par exemple, en plus de `\alias(MaClass-class)`, on peut ajouter `\alias(MaClass)`. Problème, si `MaClass` désigne à la fois le constructeur et la classe, `help(MaClass)` devra faire référence au constructeur. Si on a distingué les deux en utilisant une majuscule pour la classe et une minuscule pour le constructeur, il n'y a plus de problème. l'aide pour la classe sera appelable via `help(MaClass)` et `help(MaClass-class)`, celle pour le constructeur sera appelable via `help(maClass)`.

Concernant les méthodes inventées par le programmeur (celles qui n'existent pas ailleurs et pour lesquelles le programmeur a le contrôle du générique), il est conseillé de faire un lien de l'aide générique vers l'aide spécifique. Ainsi, si l'utilisateur averti

cherche une documentation spécifique, il tapera `help(maMethod,MaClass-methods)`. De son côté, l'utilisateur non averti tapera simplement `help(maMethod)`. Cette instruction ouvrira l'aide générique de `maMethod` ; grâce au lien que nous suggérons d'insérer dans `maMethod`, l'utilisateur non averti pourra se rediriger vers la description spécifique et trouvera lui aussi l'information qu'il cherche.

Pour les méthodes génériques définies dans d'autres packages (comme `plot`) pour lesquelles vous écrivez une définition spécifique, à moins de gentiment demander à la R Core Team un lien vers notre méthode spécifique [3], il n'y a pas vraiment de solution : tout au plus pourra-t-on ajouter `\alias(plot,MaClass)` ce qui est légèrement plus simple que `\alias(plot,MaClass-methods)`. Toutefois, depuis la version R 2.9.0, il est possible de définir des aides pour les fonctions génériques existant par ailleurs. Dans ce cas, l'appel de l'aide en question (qui renvoie donc à deux fichiers différents) propose à l'utilisateur de choisir entre les deux fichiers. Par exemple, si nous décidons de définir dans notre package un fichier d'aide pour la méthode générique `plot`, l'appel `help(plot)` déclenchera la question : "`plot` du package `base` ou `plot` du package `packS4`" l'appel d'une aide

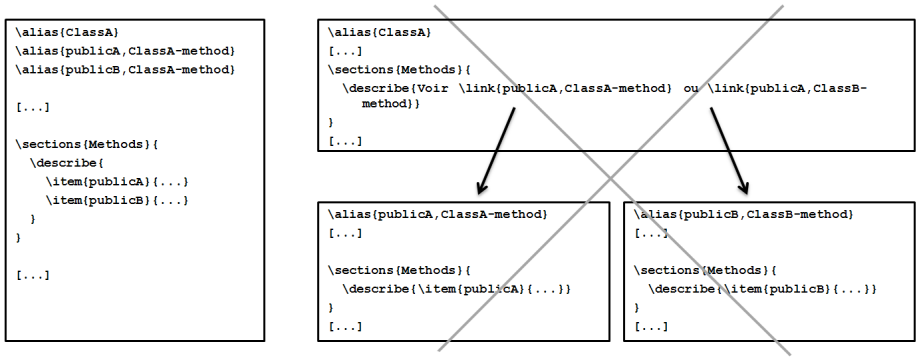
2.5.4 Les alias en double

Les alias des méthodes spécifiques existent dans plusieurs fichiers : dans les fichiers de chaque classe présente dans la signature et dans le fichier décrivant la méthode spécifique. Par exemple, `\alias(publicA,ClassV-method)` est présent dans `ClassV-class.Rd` et dans `publicA-methods.Rd`

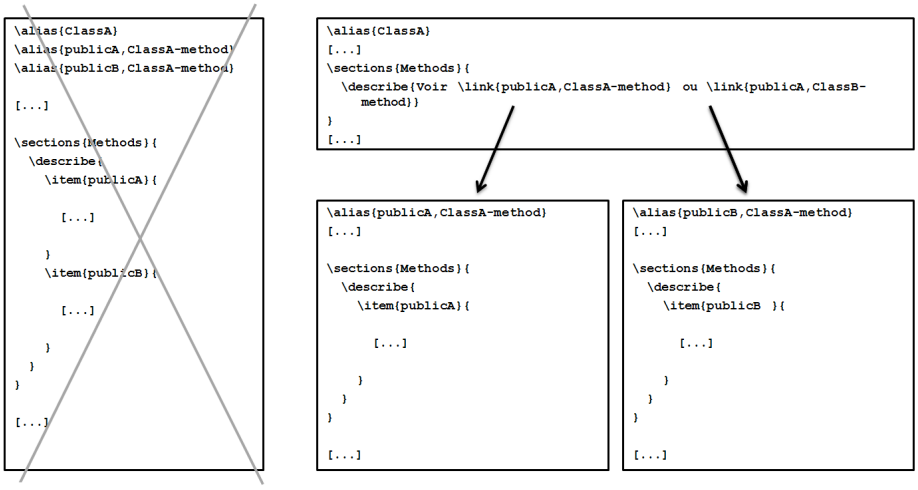
Pire, `\alias(publicB,ClassV, ClassW-method)` est présent dans trois fichiers... Il va nous falloir décider si `help(publicA,ClassV-method)` ouvre le fichier décrivant la classe `ClassV-class.Rd` ou décrivant la méthode `publicA-method.Rd`.

Comment choisir ? Un fichier d'aide contient de nombreuses informations : un titre, des auteurs, des références, des liens externes... D'un fichier d'aide à l'autre, ces informations sont redondantes. Aussi, si l'aide pour `publicA,ClassU` est courte, il est plus simple de ne pas lui créer un fichier spécifique mais de l'inclure dans le fichier décrivant la classe. Pour l'utilisateur, il est agréable d'avoir la description d'une classe puis en quelques lignes la description de ses principales méthodes, méthodes qu'il connait probablement déjà par ailleurs (comme `plot`, `print` ou `summary`) sans avoir à chercher à travers plusieurs fichiers.

3. Même pas en rêve...



Réciproquement, si les aides sur des méthodes sont longues, il sera plus agréable de les trouver dans des fichiers séparés que dans un seul méga fichier. En particulier, des informations sur une méthode précise seront plus difficiles à trouver si elles sont noyées au milieu d'autres méthodes.



Au final, la taille de l'aide doit déterminer la redirection de `help(MaMethod, MaClass)` :

- Si l'aide est courte, elle peut être laissée dans `MaClass-class.Rd`.
- Si l'aide est longue, il vaut mieux lui consacrer une page d'aide personnelle.

Dans le même esprit, si deux méthodes spécifiques sont très proches, on peut les laisser dans le même fichier. Par contre, si elles diffèrent, il pourra être plus pertinent de séparer `maMethod-methods.Rd` en deux fichiers `maMethod-ClassUmethod.Rd` et `maMethod-ClassV-method.Rd`, chacun récupérant ses alias (notez la présence ou absence des "s" à `methode(s)` selon le type de fichier). Il reste néanmoins indispensable de conserver le fichier `maMethod-methods.Rd` parce qu'il sera exigé lors de la compilation du package. Dans le cas présent, il pourra servir à rediriger vers l'un ou l'autre des alias spécifiques.

2.5.5 Bilan

Au final, nous supposons que (suppositions faites uniquement pour simuler ce qui peut se passer dans un package réel) :

- `publicA` a un comportement assez différent avec `ClassV` et `ClassW`. Il faut donc deux aides séparées `publicA-ClassV.Rd` et `publicA-ClassW.Rd`, tout en conservant le fichier `publicA-Methods.Rd` qui servira d'aiguilleur.
- Les accesseurs et constructeurs peuvent être inclus dans la description de la classe (pour pouvoir également rediriger les accesseurs génériques vers ce fichier, il faut être certain qu'ils ne concernent QUE cette classe). `z[-method.Rd`, `z[_--method.Rd` et `classZ.Rd` peuvent donc être supprimées.
- Les autres méthodes spécifiques méritent des fichiers séparés de ceux des classes.

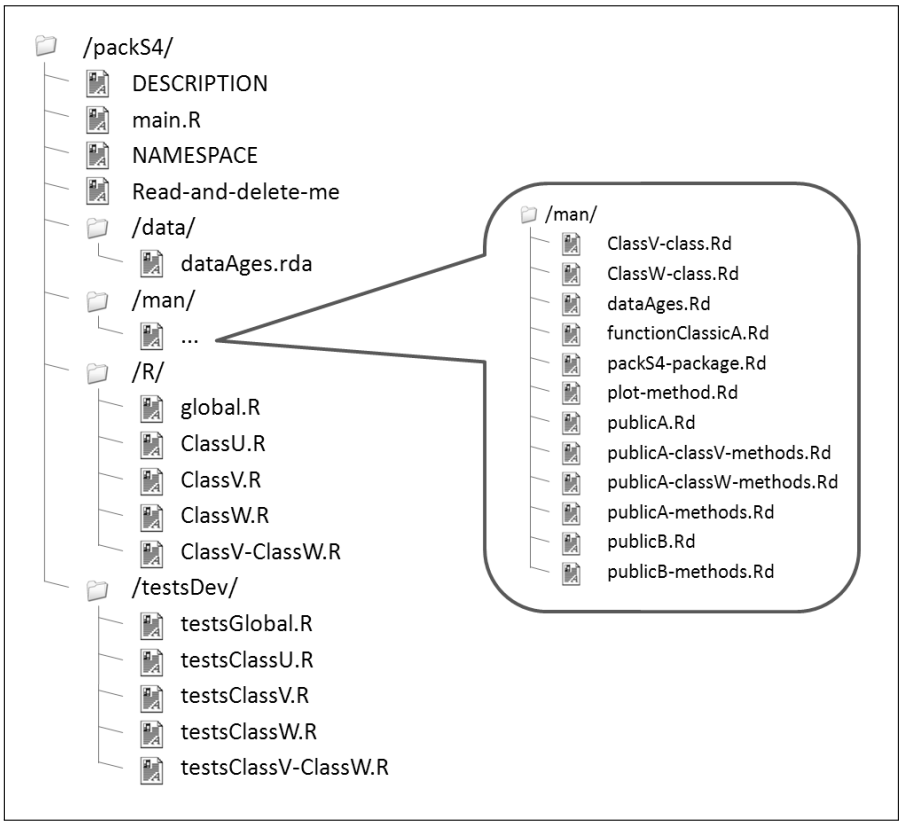
Fort de ces hypothèses, et après élimination de ce qui est privé (`ClassU` et les méthodes `private`), il nous reste (en *souligné* ce qu'il faut ajouter, en gris ce que nous avons supprimé) :

Fichier	**Alias**
ClassU-class.Rd	
ClassV-class.Rd	*ClassV* ClassV-class plot,ClassV-method privateA,ClassV-method publicA,ClassV-method publicB,ClassV,ClassW-method
classW.Rd	
ClassW-class.Rd	*ClassW* ClassW-class [,ClassW-method [<-,ClassW-method plot,ClassW-method privateA,ClassW-method publicA,ClassW-method publicB,ClassV,ClassW-method *classW*
dataAges.Rd	dataAges
functionClassicA.Rd	functionClassicA
packS4-package.Rd	packS4 packS4-package
plot-methods.Rd	plot-methods plot,ClassU-method plot,ClassV-method plot,ClassW-method
privateA.Rd	
privateA-methods.Rd	
publicA.Rd	publicA publicA-methods

publicA-ClassV-method.Rd	*publicA,ClassV-method*
publicA-ClassW-method.Rd	*publicA,ClassW-method*
publicA-methods.Rd	
publicB.Rd	publicB *publicB-methods* *publicB,ClassV,ClassW-method*
z[_--methods.Rd	
z[-methods.Rd	

Á noter, la classe `ClassU` et les méthodes `private` n'apparaissent plus puisque nous les avons définies comme privées dans la section `NAMESPACE`.

Á noter également, si une méthode générique (comme `plot`) existe déjà dans R, il n'est pas possible de créer un `\alias(plot)` ou `\alias(plot-methods)`. Par contre, il faut ajouter la description spécifique de `plot` relativement à la classe qui l'emploie (`\alias(plot,trajPartitioned-methods)` et `\alias(plot,trajPartitioned)`). Au final, notre package sera composé des fichiers suivants :



2.6 Rédaction des documentations

Deux petits changements par rapport au package classique : en premier lieu, `\linkS4class{maClass}` est une nouvelle instruction qui redirige directement vers le fichier d'aide définissant une classe. Deuxième point, les keyword `methods` et `classes` sont à ajouter dans les fichiers d'aide décrivant des méthodes et des classes.

2.7 Finalisation et création

Identique au package classique. Dans notre exemple :
- `R CMD check packS4`
- `R CMD build packS4`
- Puis un upload sur le site du CRAN `ftp://cran.r-project.org/incoming/`

Fini !
Et cette fois, c'est la bonne.
Champagne (bien mérité...)

Annexe A

Remerciements

A.1 Nous vivons une époque formidable

Quand on vit un moment historique, une révolution, une date clef, on ne s'en rend pas toujours compte. Je pense que la création d'Internet sera considérée par les historiens futurs comme une avancée majeure, quelque chose d'aussi énorme que l'invention de l'écriture ou l'imprimerie. L'écriture, c'est la conservation de l'information. L'imprimerie, c'est la diffusion de l'information à une élite, puis à tout le monde, mais avec un coût. Internet, c'est l'instantanéité, la gratuité, le partage global, les connaissances des experts mises à la disposition de tous... Les forums, c'est la fin des questions sans réponses...

Il y a quelque temps, je ne connaissais de la S4 que le nom. En quelques mois, j'ai pu acquérir des connaissances sérieuses grâce à des gens que je ne connais pas, mais qui m'ont aidé. Bien sûr, j'ai beaucoup lu. Mais dès que j'avais une question, je la posais le soir, je dormais du sommeil du juste et le lendemain matin, j'avais ma réponse. C'est gratuit, c'est de l'altruisme, c'est tout simplement merveilleux. Nous vivons une époque formidable...

A.2 Ceux par qui ce tutorial existe...

Un grand merci donc à de nombreuses personnes dont certaines que je ne connais pas. Pierre Bady, inestimable relecteur, m'a fait des remarques très pertinentes, en particulier sur la structure générale du document qui partait un peu dans tous les sens... Martin Morgan, non seulement connait TOUT sur la S4, mais de surcroit dégaine plus vite que son ombre quand il faut répondre à une question sur r-help... Merci aussi à l'équipe du CIRAD qui anime le forum du Groupe des Utilisateurs de R. Jamais on ne se fait jeter, jamais le moindre RTFM ou GIYF [1]. C'est cool. Bruno Falissard m'a donné (par ordre décroissant d'importance) une équipe [2], son amour de R, un thème de recherche, des idées, des contacts, un bureau, un ordi méga puissant... Sans lui, je serais sans

1. *Read The Fucking Manuel* ou *Google Is Your Friend*, réponses classiques faites à ceux qui posent des questions sans avoir pris le temps de chercher par eux-mêmes.
2. La MEILLEEEEUUUURE des équipes !

doute encore en train de végéter dans une impasse. Merci à Scarabette pour ses avis, sa fraicheur et son impertinence. Merci à Cathy et Michèle, mes deux chasseuses de fautes préférées. Enfin, merci à la R Core Team pour “the free gift of R”...

Bibliographie

[1] S. Baillargeon. Programmation en R: incorporation de code C et création de packages, 2006
http://www.mat.ulaval.ca/fileadmin/informatique/LogicielR/ProgR_AppelC_Package_210607.pdf.

[2] S. Baillargeon. Programmation en R: incorporation de code C et création de packages, 2006
http://www.mat.ulaval.ca/fileadmin/informatique/LogicielR/ProgR_AppelC_Package_Pres.pdf.

Index

- U -

- V -

- W -

- Z -