

Développer un *package* R avec RStudio et git

Avril 2018



Martin CHEVALIER (DMS)

Développer un *package* R avec RStudio et git

Objectifs et plan de la présentation

1. Introduire des concepts fondamentaux pour lancer le développement d'un *package* R
2. Faire un panorama des outils qui facilitent le développement collaboratif de *packages* R

La plupart des exemples sont illustrés par le *package* gustave d'estimation de variance développé par la division Sondages.

Plan de la présentation

1. Développer un *package* R avec RStudio
2. Développer un *package* R avec git

Développer un *package* R avec RStudio

Développer un *package* R avec RStudio

Qu'est-ce qu'un *package* R ?

Définition Ensemble de fonctions R documenté et structuré de façon à être facilement réutilisé par d'autres.

Avantages

- ▶ facile à utiliser pour d'autres utilisateurs (internes ou externes) ;
- ▶ facile à publier pour les développeurs (numéro de version, etc.) ;
- ▶ meilleure qualité : documentation, tests de chaque fonctionnalité (« tests unitaires »), correction de bugs détectés par d'autres utilisateurs.

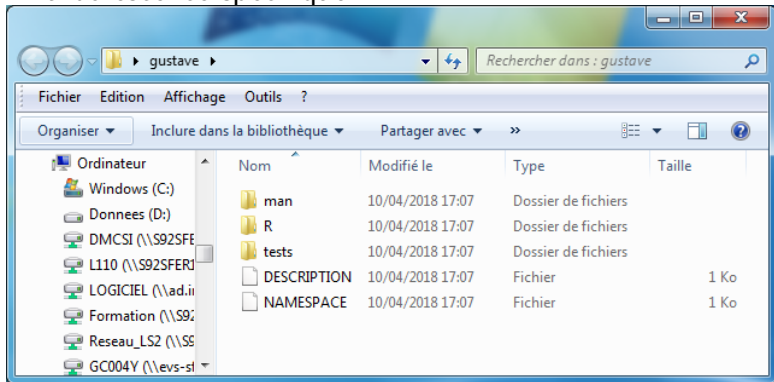
Inconvénients

- ▶ plus complexe à développer que du code R standard ;
- ▶ contraintes pour que le *package* soit accepté sur le CRAN.

Développer un *package* R avec RStudio

Structure d'un *package* R

- ▶ nom unique (attention à la casse!);
- ▶ arborescence spécifique :



Références Writing R Extensions, R packages, cheatsheet

Structure d'un *package* R : DESCRIPTION

Le fichier DESCRIPTION est le fichier texte qui contient les méta-données du *package* :

- ▶ son nom, son titre et une courte description ;
- ▶ sa version (par exemple 0.2.9 pour gustave) et son mainteneur ;
- ▶ les informations relatives à la propriété intellectuelle : licence, auteur, etc. ;
- ▶ la version de R et la liste des *packages* dont il dépend (mots-clés Depends et Imports) ;
- ▶ d'autres informations techniques : encodage des fichiers, ordre dans lequel les fichiers R doivent être soumis (si nécessaire), etc.

Développer un *package* R avec RStudio

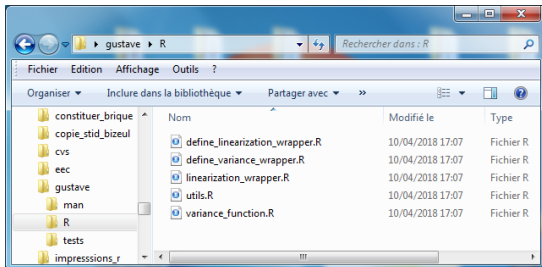
Structure d'un *package* R : R/

Le dossier R/ contient l'ensemble des codes R utilisés par le *package*, organisés comme le souhaite les développeurs.

Deux extrêmes :

- ▶ toutes les fonctions du *packages* dans le même fichier .R ;
- ▶ un fichier .R par fonction.

Exemple de gustave



Développer un *package* R avec RStudio

Structure d'un *package* R : `man/` et `NAMESPACE`

Le dossier `man/` contient l'ensemble des fichiers d'aide du *package* (un fichier par fonction).

Ces fichiers sont structurés par des balises et permettent de générer le code HTML d'aide (accessible *via* ?) ainsi que le manuel `.pdf` en ligne.

Le fichier `NAMESPACE` liste en particulier l'ensemble des fonctions du *package* qui ont vocation à être accessibles aux utilisateurs.

Remarque importante Les fichiers du dossier `man/` et le fichier `NAMESPACE` peuvent être générés automatiquement par le *package* `roxygen2`.

Développer un *package* R avec RStudio

Structure d'un *package* R : tests/

Le dossier tests/ (optionnel) a vocation à contenir un ensemble de tests qui vérifient le bon fonctionnement du *package*.

On parle en particulier de « test unitaire » pour désigner le test d'une fonctionnalité précise (indépendamment des autres).

Coder des tests unitaires nombreux qui sont relancés à chaque évolution du *package* permet de garantir la **non-régression** entre les versions.

Remarque importante Le *package* testthat offre un écosystème pour simplifier la mise en œuvre de tests unitaires.

Développer un *package* R avec RStudio

Structure d'un *package* R : `src/`

Quand un *package* fait appel à d'autres langages (C++ par exemple), le dossier `src/` contient le code source des fonctions en question.

Pour développer ce type de *package*, il est indispensable que l'ordinateur soit en mesure de **compiler** le ou les langages concernés.

Remarque importante Sous Windows, il est recommandé d'installer le programme Rtools (qui comporte notamment les compilateurs nécessaires) pour développer ce type de *package*.

Structure d'un *package* R : autres dossiers

Plusieurs autres dossiers optionnels peuvent être ajoutés à la racine d'un *package* R :

- ▶ `data/` : il s'agit d'un dossier contenant des données (compressées) utilisées en particulier dans les exemples du *package*;
- ▶ `vignettes/` : le dossier `vignettes/` comporte le code (souvent au format Markdown, `.md`) des documents d'explication longs du *package*, qui sont nettement moins contraints que les éléments d'aide classiques (dans `man/`).

Développer un *package* R avec RStudio

Code source, compilation et installation

L'arborescence décrite dans les diapositives précédentes est celle d'un *package* au moment de son développement, c'est-à-dire quand il est à l'état de **code source**.

Pour être utilisé par R, un *package* doit être **compilé** et **installé** pour le système sur lequel R s'exécute.

Quand un *package* est **compilé**, il n'est plus possible de lire directement le code des fonctions et le contenu de l'aide.

Remarque importante

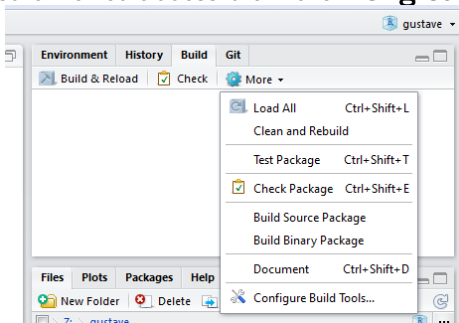
- ▶ sous Windows, en règle générale la compilation est faite par le CRAN en amont du téléchargement et de l'installation ;
- ▶ sous Linux, l'installation est effectuée à partir du code source et le *package* est compilé lors de l'installation.

Développer un *package* R avec RStudio

Utiliser RStudio pour développer un *package*

Le mode « projet » de RStudio facilite considérablement le développement de *packages* R.

L'ensemble des commandes complexes nécessaires pour produire un *package* est en effet accessible *via* un **onglet spécifique**.



Remarque Cette interface graphique s'appuie notamment sur les *packages* devtools, roxygen2 et testthat.

Développer un *package* R avec RStudio

Utiliser RStudio pour développer un *package*

- ▶ « Load All » charge l'ensemble des fonctions du *package* ;
- ▶ « Document » lance les fonctions de documentation automatique de roxygen2 ;
- ▶ « Test Package » lance les codes R du dossier tests/, en particulier les tests conçus avec le *package* testthat ;
- ▶ « Build and Reload » réinstalle le *package* et le lance ;
- ▶ « Clean and Rebuild » supprime le *package* et le réinstalle ;
- ▶ « Build Source Package » et « Build Binary Package » produisent un fichier .tar.gz facilement exportable, au format code source ou pré-compilé respectivement ;
- ▶ « Check Package » vérifie que la structure du *package*

Développer un *package* R avec RStudio

Documenter un *package* avec roxygen2

roxygen2 est un package qui permet d'**intégrer dans le même code R** le code et la documentation d'une fonction.

Concrètement, les éléments de documentation d'une fonction sont intégrés dans le code *via* des commentaires commençant par '#' et structurés par un ensemble de mots-clés :

```
#' Ma super fonction
#' @description C'est une super fonction.
#' @param arg1 Un premier argument
#' @param arg2 Un deuxième argument
#' @examples ma_super_fonction()
#' @export ma_super_fonction
ma_super_fonction <- function(arg1, arg2) "Pouêt !"
```

roxygen2 parcourt les fichiers R du dossier R/ pour produire automatiquement les fichiers du dossier man/ ainsi que le fichier NAMESPACE.

Développer un *package* R avec RStudio

Tester un *package* avec testthat

Le *package* testthat fournit tout un ensemble de fonctions pour **automatiser les tests** des fonctionnalités du *package*.

```
context("Arithmétique")
test_that("les maths, ça marche !", {
  expect_equal(2 + 2, 4)
  expect_true(1 < 2)
  expect_error(1 * 2, NA)
})
```

Le résultat des tests menés est compris par l'outil de vérification de *package* (« Check Package ») ainsi que par l'interface de RStudio : dès qu'un test unitaire échoue, il est identifié.

Ce type de fonctionnalités pousse à systématiser les tests unitaires dans le développement d'un *package* (*cf. test driven development*).

Développer un *package* R avec git

Développer un *package* R avec git

Qu'est-ce que git ?

git est un des nombreux outils de gestion de version (*version control system* ou VCS en anglais) utilisés dans le monde du développement logiciel.

Les outils de gestion de version (CVS, SVN par exemple) ont été inventés pour répondre à **deux problèmes majeurs** :

- ▶ la **conservation** de versions successives du code d'un projet : par défaut, les développeurs recourent à la technique du CPOLD (COPY + OLD), c'est-à-dire

```
mon_code.R  
mon_code_copie_1.R  
mon_code_vdef.R  
mon_code_180411_192600.R
```

- ▶ la **collaboration** autour du même projet sans passer par le verrouillage de fichiers (exemple : fichier LibreOffice).

Développer un *package* R avec git

Pourquoi développer un *package* R avec git ?

1. **Sécuriser** le code : conservation de l'ensemble des lignes de codes, même celles qui ne figurent plus dans la version actuelle du projet ;
2. **Améliorer** la qualité du code : méta-données autour de chaque modification (donc suppression de commentaires superflus), relecture par d'autres facilitée ;
3. **Simplifier** la diffusion du *package* : installation directe des *packages* depuis un dépôt git (sans passer par le CRAN) ;
4. **Faciliter** la collaboration : travail en parallèle sur le code du *package* puis consolidation des modifications.

Référence Pro Git (Scott Chacon, Ben Straub)

Développer un *package* R avec git

En pratique : le répertoire (caché) `.git/`

Quand un projet est suivi en version par git, le répertoire (caché) `.git/` est ajouté à sa racine (on parle de dépôt git).

C'est dans ce répertoire `.git/` qu'est stockée l'**ensemble de la mémoire du projet** : il est possible à partir de ce seul dossier de récupérer toutes ses versions.

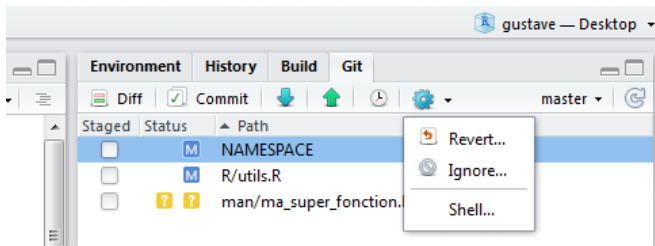
Remarque Aucun « serveur » n'est nécessaire pour utiliser git en tant que tel, tout est contenu dans le répertoire `.git/`.

Quand une modification est effectuée dans le projet, il faut que celle-ci soit répercutée (*via* les commandes de git) dans le répertoire `.git/` pour être sauvegardée à jamais (ou presque!).

Développer un *package* R avec git

Une séance de travail avec git : RStudio

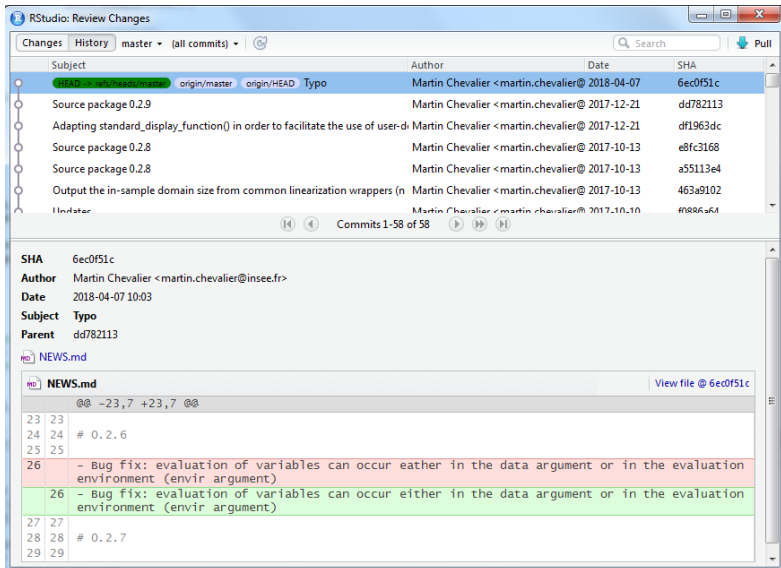
Quand un projet RStudio est configuré pour utiliser git, un onglet supplémentaire apparaît :



Les principales fonctionnalités sont dans l'ordre : « Diff », « Commit », « Pull », « Push » et enfin l'affichage de l'historique des versions.

Développer un *package* R avec git

Une séance de travail avec git : RStudio



The screenshot shows the RStudio 'Review Changes' window. The top panel displays a list of commits, with the selected commit being 'HEAD -> origin/master origin/HEAD Typo' by Martin Chevalier, dated 2018-04-07, with SHA 6ec0f51c. The bottom panel shows the diff for the file 'NEWS.md', highlighting a bug fix in green and a comment in red.

Commits 1-58 of 58

Subject	Author	Date	SHA
HEAD -> origin/master origin/HEAD Typo	Martin Chevalier <martin.chevalier@insee.fr>	2018-04-07	6ec0f51c
Source package 0.2.9	Martin Chevalier <martin.chevalier@insee.fr>	2017-12-21	dd782113
Adapting standard_display_function() in order to facilitate the use of user-defined functions	Martin Chevalier <martin.chevalier@insee.fr>	2017-12-21	df1963dc
Source package 0.2.8	Martin Chevalier <martin.chevalier@insee.fr>	2017-10-13	e8fc3168
Source package 0.2.8	Martin Chevalier <martin.chevalier@insee.fr>	2017-10-13	a55113e4
Output the in-sample domain size from common linearization wrappers (n	Martin Chevalier <martin.chevalier@insee.fr>	2017-10-13	463a9102
Update	Martin Chevalier <martin.chevalier@insee.fr>	2017-10-10	f0886a61

SHA 6ec0f51c
Author Martin Chevalier <martin.chevalier@insee.fr>
Date 2018-04-07 10:03
Subject Typo
Parent dd782113

NEWS.md

```
@@ -23,7 +23,7 @@
23 23
24 24 # 0.2.6
25 25
26 - Bug fix: evaluation of variables can occur either in the data argument or in the evaluation
26 + Bug fix: evaluation of variables can occur either in the data argument or in the evaluation
27 27 environment (envir argument)
28 28 # 0.2.7
29 29
```

Développer un *package* R avec git

Une séance de travail avec git : *diff*

Au début d'une séance de travail, en général aucun fichier n'apparaît dans l'onglet git : les fichiers du projet correspondent exactement à la dernière version du projet connue de git.

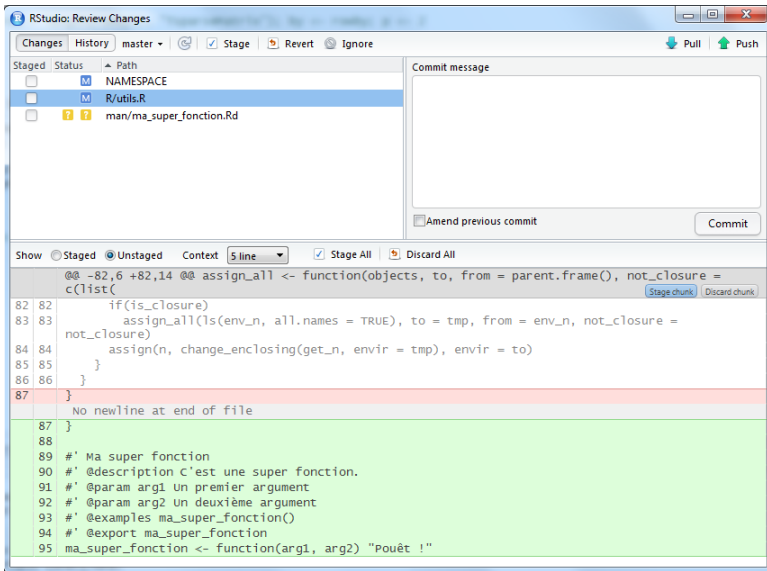
Au fur et à mesure que des modifications sont effectuées, des fichiers apparaissent avec une lettre à côté de leur nom :

- ▶ « M » pour les fichiers modifiés ;
- ▶ « A » pour les fichiers ajoutés ;
- ▶ « D » pour les fichiers supprimés.

À tout moment, le menu « Diff » permet de vérifier d'un coup d'œil toutes les modifications effectuées.

Développer un *package* R avec git

Une séance de travail avec git : *diff*



The screenshot shows the RStudio 'Review Changes' window. The top bar indicates the current branch is 'master'. The 'Changes' tab is active, showing a list of files: 'NAMESPACE' (modified), 'R/utls.R' (modified), and 'man/ma_super_fonction.Rd' (modified). The 'Commit message' field is empty. The 'Amend previous commit' checkbox is unchecked. The 'Commit' button is visible. Below the file list, the 'Show' section is set to 'Unstaged' and 'Context' is set to '5 line'. The 'Stage All' checkbox is checked. The diff view shows the changes in 'ma_super_fonction.Rd'. The diff starts with a hunk header: '@@ -82,6 +82,14 @@ assign_all <- function(objects, to, from = parent.frame(), not_closure = c(list(' if(is_closure) assign_all(is(env_n, all.names = TRUE), to = tmp, from = env_n, not_closure = not_closure) assign(n, change_enclosing(get_n, envir = tmp), envir = to) } } }'. The diff shows that line 87 was added, closing the function definition. The diff ends with a hunk footer: 'No newline at end of file'. The diff view shows the changes in 'ma_super_fonction.Rd'. The diff starts with a hunk header: '@@ -82,6 +82,14 @@ assign_all <- function(objects, to, from = parent.frame(), not_closure = c(list(' if(is_closure) assign_all(is(env_n, all.names = TRUE), to = tmp, from = env_n, not_closure = not_closure) assign(n, change_enclosing(get_n, envir = tmp), envir = to) } } }'. The diff shows that line 87 was added, closing the function definition. The diff ends with a hunk footer: 'No newline at end of file'.

```
@@ -82,6 +82,14 @@ assign_all <- function(objects, to, from = parent.frame(), not_closure =  
c(list(  
82 82     if(is_closure)  
83 83         assign_all(is(env_n, all.names = TRUE), to = tmp, from = env_n, not_closure =  
not_closure)  
84 84     assign(n, change_enclosing(get_n, envir = tmp), envir = to)  
85 85     }  
86 86     }  
87 }  
No newline at end of file  
87 }  
88 }  
89 #' Ma super fonction  
90 #' @description c'est une super fonction.  
91 #' @param arg1 Un premier argument  
92 #' @param arg2 Un deuxième argument  
93 #' @examples ma_super_fonction()  
94 #' @export ma_super_fonction  
95 ma_super_fonction <- function(arg1, arg2) "Pouêt !"
```


Développer un *package* R avec git

Une séance de travail avec git : *stage* et *commit*

Les opérations *stage* et *commit* permettent de sauvegarder les modifications effectuées dans git (i.e. dans le répertoire `.git/` du projet).

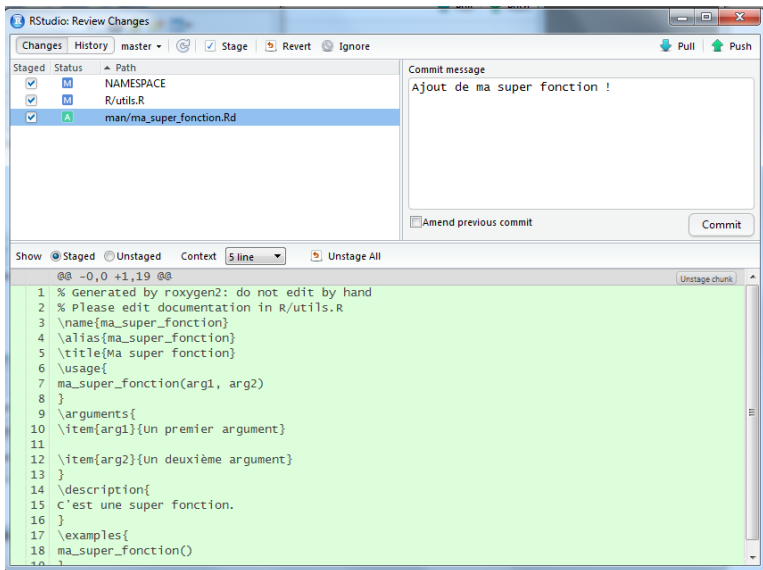
Il est recommandé de les effectuer relativement fréquemment : s'il s'avère nécessaire de revenir en arrière, il est préférable qu'il y ait peu de changements depuis la dernière version.

1. *stage* : sélectionner les fichiers que l'on souhaite. . .
2. *commit* : . . . intégrer dans la mise à jour de la dernière version du projet dans git.

Remarque La plupart du temps, tous les éléments ont vocation à être sélectionnés (*staged*) pour le prochain *commit*.

Développer un *package* R avec git

Une séance de travail avec git : *stage* et *commit*



Développer un *package* R avec git

Une séance de travail avec git : *bis repetita*

diff, *stage* et *commit* sont les trois opérations essentielles à connaître pour travailler au quotidien avec git.

Remarque Il est possible de corriger le *commit* précédent en cochant la case « *Amend previous commit* ».

Elles aident à décomposer le travail en petites opérations faciles à décrire et à valider individuellement.

Exemple Pour développer une nouvelle fonctionnalité :

1. Écrire le test qui correspond au résultat souhaité
2. Écrire une première version minimale de la fonctionnalité
3. Écrire une version plus complète de la fonctionnalité

Développer un *package* R avec git

git : un outil de gestion de version décentralisé

Jusqu'à présent, toutes les opérations ont été faites dans le dossier personnel d'un développeur en particulier.

Pour collaborer avec d'autres développeurs sur le même projet, il suffit que les dossiers `.git/` des uns et des autres **s'échangent l'histoire du projet.**

La principale innovation de git est d'être **complètement décentralisé** : le développement pourrait ne s'appuyer que sur des échanges « de pair-à-pair » entre développeurs.

En pratique cependant, un dépôt particulier est choisi comme dépôt de référence : même si on l'appelle en général le « serveur », il ne présente **aucune particularité technique.**

Développer un *package* R avec git

Collaborer avec git : *remote*

L'opération *remote* permet d'ajouter et de configurer de nouveaux dépôts avec lesquels le dépôt local est susceptible d'échanger.

Remarque Les dépôts en question peuvent être distants ou sur la même machine que le dépôt local (par exemple sur un espace partagé sur AUS).

Concrètement, cette opération permet d'associer un nom symbolique aux dépôts distants.

Exemple Le dépôt

<https://github.com/martinchevalier/gustave> est le dépôt distant libellé "github" du dépôt git "gustave" de mon poste de travail.

Développer un *package* R avec git

Collaborer avec git : *pull* et *push*

L'opération *pull* permet de mettre à jour le dépôt local à partir du contenu d'un dépôt distant.

Dans le cadre d'un travail coopératif, cette opération a vocation à être effectuée **en début de séance de travail**.

Inversement, l'opération *push* permet de mettre à jour un dépôt distant à partir du contenu du dépôt local.

Dans le cadre d'un travail coopératif, cette opération a vocation à être effectuée **en fin de séance de travail**.

Collaborer avec git : conflits

Il peut survenir que git ne parvienne pas à synchroniser les deux dépôts automatiquement.

Cela survient en général quand des modifications ont été effectuées en parallèle sur les mêmes lignes de code.

Dans cette situation, il est impératif de régler les conflits pour finaliser l'opération :

- ▶ les zones en conflit sont indiquées spécifiquement par git ;
- ▶ le développeur modifie l'ensemble des zones en conflit, teste sa solution et propose un commit de fusion (*merge*).

Développer un *package* R avec git

Collaborer avec git : branches

C'est la manière dont git permet de construire des dérivations (ou « branches ») qui constitue la principale raison de son succès.

Exemple Pour développer une fonctionnalité complexe sur plusieurs semaines, un développeur a le choix entre :

- ▶ effectuer des *push* chaque jour : déstabilisation de l'ensemble du programme (car la fonctionnalité ne sera pas complètement codée) ;
- ▶ effectuer un seul *push* à la fin : pas de relecture par les pairs, gros travail de fusion à effectuer.

Dans cette situation, créer une nouvelle branche permet de poursuivre un développement **sans perturber la branche principale** et tout en intégrant ses évolutions.

Développer un *package* R avec git

Collaborer avec git : plateformes

De nombreuses plateformes se sont construites autour de git pour faciliter le travail des développeurs : github.com, serveur web *open-source* gitlab.

Leur fonction première est d'être des candidats naturels pour accueillir le **dépôt de référence d'un projet**.

Elles comportent néanmoins un certain nombre de **fonctions annexes très appréciables** :

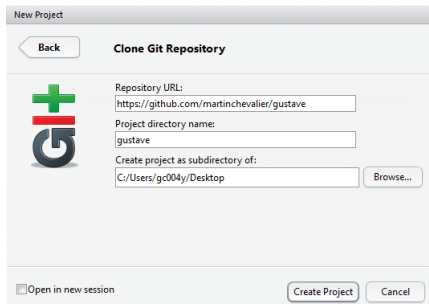
- ▶ exploration et modification du code source des projets dans un navigateur web ;
- ▶ centralisation des rapports de bugs, documentation ;
- ▶ couche sociale permettant la collaboration entre développeurs ;
- ▶ tests et déploiement automatique.

Développer un *package* R avec git

Commencer avec git : *clone*

Une dernière fonctionnalité non-négligeable des plateformes est de simplifier l'initialisation d'un dépôt git.

En effet, il est souvent plus simple de « cloner » un projet depuis une de ces plateformes que de le créer de toute pièce en local.



Remarque Pour obtenir ce menu : File > New project > Version control > Git.

Conclusion

Le développement de *packages* R constitue un enjeu important pour la construction et la diffusion d'outils méthodologiques libres et de qualité.

Les fonctionnalités offertes par l'écosystème de RStudio (IDE, *packages* devtools, roxygen2 et testthat) simplifient considérablement cette opération.

Par ailleurs, l'utilisation d'un outil de gestion de version comme git présente de nombreux avantages : suivi de version, collaboration interne et externe, diffusion simplifiée.