

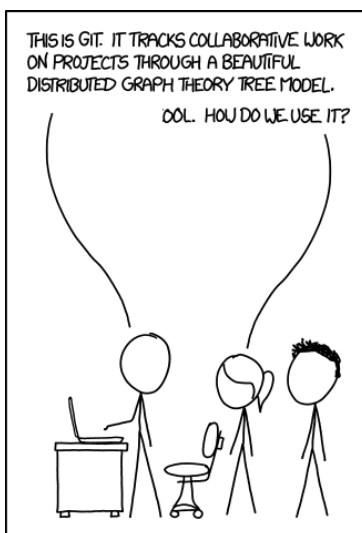


---


# Formation Git


---

Jonathan SCHAEFFER  
jonathan.schaeffer@univ-brest.fr



2015

Ce document est mis à votre disposition sous les termes de la licence Creative Commons Attribution 4.0 

Les illustrations issues de la documentation **progit** <https://git-scm.com/book/fr/v2> sont sous licence 

L'illustration ci dessus est issue du comic-strip XKCD <http://xkcd.com/1597>, sous licence 

# Table des matières

<b>1</b>	<b>Premiers pas</b>	<b>1</b>
1.1	Préparation . . . . .	1
1.2	Un dépôt git . . . . .	1
<b>2</b>	<b>Théorie de Git</b>	<b>4</b>
2.1	Différences et Versionning . . . . .	4
2.2	Fonctionnement de Git . . . . .	5
2.2.1	Un dépôt . . . . .	5
2.2.2	Représentation interne du dépôt . . . . .	5
2.3	Les instantanés du projet . . . . .	6
2.3.1	lister les instantanés . . . . .	7
2.3.2	Créer un instantané . . . . .	7
2.3.3	Commenter un instantané . . . . .	7
2.3.4	Configuration pour les commits . . . . .	7
2.3.5	Choisir les fichier à ignorer . . . . .	9
2.4	Conclusion . . . . .	11
<b>3</b>	<b>Manipuler Git</b>	<b>12</b>
3.1	Visualiser l'historique . . . . .	12
3.2	Annuler des actions . . . . .	13
3.2.1	Corriger un commit . . . . .	13
3.2.2	Désindexer un fichier indexé . . . . .	14
3.2.3	Annuler les modifications locales . . . . .	14
3.3	Manipuler l'historique . . . . .	16
3.3.1	La représentation de l'historique . . . . .	16
3.3.2	Se balader dans l'historique . . . . .	16
3.3.3	Annuler le dernier commit . . . . .	17
3.3.4	Les pointeurs . . . . .	18
3.3.5	Gérer des étiquettes simples . . . . .	18
3.4	Les branches de développement . . . . .	19

<i>TABLE DES MATIÈRES</i>	<i>2</i>
3.4.1 Création de branches . . . . .	19
3.4.2 Fusion de branches . . . . .	21
3.4.3 Résolution de conflits . . . . .	21
<b>4 Git : un système décentralisé</b>	<b>24</b>
4.1 Utiliser un dépôt distant . . . . .	24
4.1.1 Cloner le dépôt . . . . .	24
4.1.2 Synchroniser un dépôt cloné . . . . .	25
4.2 Travail collaboratif . . . . .	26
<b>5 Conclusion</b>	<b>28</b>
5.1 Résumé du contenu du cours . . . . .	28
5.2 Ce qu'il reste à explorer . . . . .	28
5.3 Références utiles . . . . .	28
<b>Solution des exercices</b>	<b>30</b>

# Chapitre 1

## Premiers pas

Dans ce chapitre, nous allons commencer à découvrir l'outil de versionning git.

### 1.1 Préparation

Avant de nous aventurer vers la manipulation de Git, nous allons créer un compte sur le service en ligne "Framagit", afin de préparer les TP du chapitre 4.

Vous allez d'abord créer un compte chez <https://git.framasoft.org/>. En attendant le courriel de confirmation, nous pouvons démarrer le TP.

### 1.2 Un dépôt git

Notre première opération va consister à créer un dépôt git à partir d'un répertoire de travail.

Pour cela, utilisez le répertoire d'exemple `mon_depot`. Constatez qu'il contient déjà quelques fichiers, et copiez-le sur le bureau.

À partir d'un terminal (**Menu démarrer** ⇒ **Console**), nous allons nous positionner dans ce répertoire d'exemple (commande `cd`), et listez (commande `ls`) son contenu :

```
$ cd ~/Bureau/mon_depot
$ ls -a
```

Pour créer un dépôt git (on parle d'initialisation), utilisez la commande `git init` :

```
$ git init
```

Cette commande vient de transformer le répertoire de travail en dépôt git. La différence entre un répertoire de travail et un dépôt git est la présence d'un sous-répertoire `.git` à l'intérieur de notre dépôt.

En listant le contenu de notre dépôt, ce sous-répertoire doit apparaître :

```
$ ls -a
.  ..  .git  salut.py  salut.py~  salut.rb  salut.rb~
```

Dès lors, nous pouvons commencer à utiliser d'autres commandes git. Interrogez l'état du dépôt (`git status`) :

```
$ git status
Sur la branche master
Validation initiale
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    salut.py
    salut.py~
    salut.rb
    salut.rb~
aucune modification ajoutée à la validation mais des fichiers non suivis
sont présents (utilisez "git add" pour les suivre)
```

Nous allons commencer par suivre l'ensemble des fichiers de ce dossier. Comme proposé par `git status` :

```
$ git add *
$ git status
Sur la branche master
Validation initiale
Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)
    nouveau fichier : salut.py
    nouveau fichier : salut.py~
    nouveau fichier : salut.rb
    nouveau fichier : salut.rb~
```

Cette commande nous liste les modification prêtes à être validées. Allons y, validons, en utilisant la commande `git commit` :

```
$ git commit --message "Première validation git"
[master (commit racine) a2fa537] Première validation
 4 files changed, 34 insertions(+)
 create mode 100644 salut.py
 create mode 100644 salut.py~
 create mode 100644 salut.rb
 create mode 100644 salut.rb~
$ git status
Sur la branche master
rien à valider, la copie de travail est propre
```

Voilà, nous venons de valider le suivi des premiers changement dans notre dépôt.

### Exercice 1 Premier pas

Continuez cet exercice en modifiant un des fichiers du dépôt :

1. Ouvrez "salut.py" avec un éditeur de texte.
2. Ajoutez un commentaire dans ce fichier (en le faisant commencer par un #).
3. Sauvegardez le fichier
4. Utilisez les commandes suivantes pour valider vos modifications :
  - `$ git status`
  - `$ git add mon-fichier`
  - `$ git commit --message ""`
5. Utilisez la commande `git log` pour visualiser le journal de vos modifications.

```
$ git log
```

# Chapitre 2

## Théorie de Git

### 2.1 Différences et Versionning

Manipuler des différences entre 2 fichiers de type texte est une opération très courante. Dans un environnement de programmation, on fait appel à une commande **diff** pour cela.

Par exemple, la commande diff entre 2 fichiers donne :

```
$ diff -u collection.rb collection2.rb
--- collection.rb      2015-07-24 14:15:07.416308012 +0200
+++ collection2.rb     2015-08-17 11:45:53.305468727 +0200
@@ -7,7 +7,6 @@
 # unit_id
 class Collection < ActiveRecord::Base
   belongs_to :station
-  belongs_to :unit
+  has_many :units
   belongs_to :samplemethod
   has_many :samples
   has_many :results
```

C'est de cette manière que les gestionnaires de version peuvent enregistrer et garder un historique des versions successives d'un projet de développement.

#### Exercice 2 Travailler avec la commande diff

Dans le dossier TP, prenez le fichier d'exemple "exercice1.txt"

Faites une copie de ce fichier et renommez-la "exercice1\_mine.txt"

Insérez des modifications à plusieurs endroits, et testez la sortie des commandes suivantes :

```
$ diff -u exercicel.txt exercicel_mine.txt
$ diff -u exercicel_mine.txt exercicel.txt
```

## 2.2 Fonctionnement de Git

### 2.2.1 Un dépôt

Un dépôt git est un répertoire (ou dossier) dans lequel toute modification peut être suivie dans le temps.

On peut initialiser un dépôt git dans n'importe quel répertoire. Tous les sous-répertoires feront partie du dépôt.

Lorsqu'on appelle le programme git, il cherche à savoir s'il se trouve dans un dépôt ou non. Pour cela, il cherche la présence d'un sous-répertoire caché `.git/` en remontant l'arborescence.

### 2.2.2 Représentation interne du dépôt

Dans un dépôt git, tout fichier est dans 3 états possibles. On peut faire un bilan de l'état de chaque fichier dans un dépôt avec la commande `status` :

*git status*

Les trois états possibles sont les suivants :

**modifié** Un fichier dans cet état a été localement modifié, mais git n'a pas validé les différences dans sa base de données locale. Les fichiers dans cet état sont dans le répertoire de travail (ou **working directory**).

**indexé** Un fichier dans cet état fait partie des modifications qui sont prêtes à être validées pour le prochain instantané. Les fichiers dans cet état sont dans l'index (ou dans le **staging area**).

**validé** Les données dans cet état sont en sécurité dans la base de donnée locale de git (aussi appelée **git directory**). Les modifications ainsi sauvegardées font partie d'un instantané du projet (appelé aussi **commit**).

Le schéma 2.1 représente ces 3 états et les opérations de transition.

De manière classique, on utilise git en suivant les étapes suivante :

1. On modifie un fichier localement
2. On ajoute la modification à l'index (**staging**)
3. On enregistre un instantané du projet à partir de l'index (**commit**)

Ces notions sont fondamentales dans git et il est primordial de bien les comprendre.



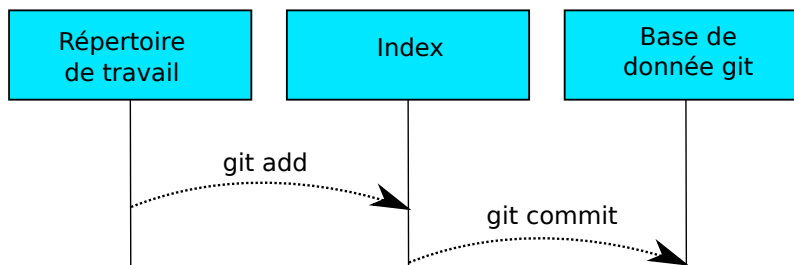


FIGURE 2.1 – Les 3 états dans un dépôt Git et leurs transitions

### Ajout à l'index

Ajouter une modification à l'index se fait par la commande **add**.

Après avoir modifié un fichier ou créé un nouveau fichier, on place ces changements dans l'index :

```
git add mon_fichier
```

L'ensemble des modifications de l'index permettent de maintenant d'enregistrer un instantané.

### Enregistrement d'un instantané

Enregistrer un instantané, c'est écrire dans la base de donnée locale de git les modifications indexées. La commande git pour cette opération est **commit**.

```
git commit
```

Nous détaillerons cette opération dans la section 2.3.

## 2.3 Les instantanés du projet

Git enregistre dans une base de donnée locale une succession d'instantanés (ou **commits**).

La figure 2.2 illustre cette succession dans le temps.

Un instantané a plusieurs caractéristiques fondamentales :

**unicité** Un instantané est unique et identifié par un numéro de série (par exemple `f31a288e90dae7712f49b061a56486f6d489a657`)

**auteur** Un instantané est lié à un auteur (nom + adresse mail)

**date** Un instantané est horodaté

**commentaire** Un instantané est obligatoirement accompagné d'un commentaire de son auteur.

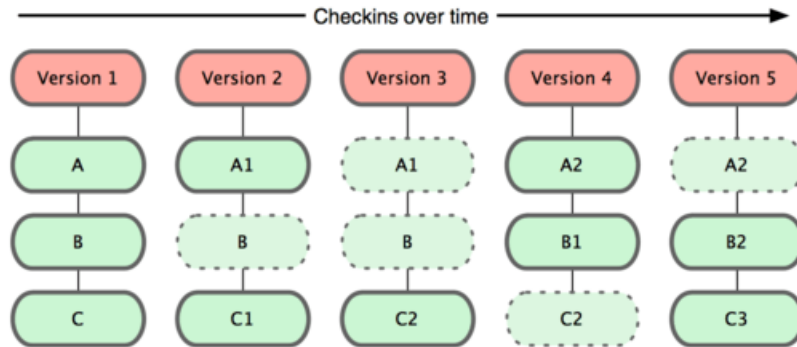


FIGURE 2.2 – Les instantanés dans le temps reconstituent l’ensemble des fichiers du projet à un instant

### 2.3.1 lister les instantanés

La commande **log** permet de lister tous les instantanés successifs du projet :

```
git log
```

### 2.3.2 Créer un instantané

La commande **commit** permet de créer un instantané à partir de l’index :

```
git commit
```

### 2.3.3 Commenter un instantané

La commande **commit** ouvre une fenêtre d’édition dans laquelle l’auteur va écrire le commentaire attaché à son commit.

Voici les règles courantes implicites d’un bon message de commit :

1. Un titre (maximum 80 caractères)
2. Un saut de ligne
3. Des détails sur les modifications de ce commit

L’auteur, la date et l’identifiant du commit sont automatiquement ajoutés par git.

### 2.3.4 Configuration pour les commits

On peut configurer globalement un auteur :

```
$ git config --global user.name "Gaston Lagaffe"  
$ git config --global user.mail "gaston.lagaffe@journal-de-spirou.fr"
```

On peut configurer l'éditeur que git vous présentera pour rédiger les messages de commit (ici, je propose vim, mais on peut aussi choisir d'autres alternatives comme nano) :

```
$ git config --global core.editor nano
```

Pour informations, ces configurations sont stockées dans un simple fichier : `/.git-config`.

### Exercice 3 Un premier commit

Avant tout, nous allons configurer Git comme indiqué dans la section 2.3.4. Maintenant, nous allons nous intéresser à nouveau à la création de nouveaux commits. Commençons par examiner le journal de notre dépôt :

*git log*

Ensuite, vérifions l'état de notre dépôt :

```
$ git status  
Sur la branche master  
rien à valider, la copie de travail est propre
```

Créez un nouveau fichier README dans le dépôt, écrivez un message à l'intérieur et ajoutez ce fichier dans l'index.

Ensuite, créez un commit, en respectant les règles énoncées plus haut pour le message.

```
$ git commit  
[master (commit racine) bf67c41] Titre court  
1 file changed, 1 insertion(+)  
create mode 100644 mon_fichier.txt
```

Notez les informations renvoyées par la commande commit, en particulier :

*create mode 100644 mon\_fichier.txt*

Ceci indique que git a enregistré la création d'un nouveau fichier.

```
$ git log
commit 2e510ae71d021486cc86ec744c0320c2ad665cca
Author: Jonathan Schaeffer <schaeffer@univ-brest.fr>
Date: Thu Aug 27 16:06:54 2015 +0200
    Ajout d'un fichier README

    Ce sera la documentation du projet

commit a2fa537bda968d92ac98d9813157c2e44e021126
Author: Jonathan Schaeffer <schaeffer@univ-brest.fr>
Date: Mon Aug 24 16:31:44 2015 +0200
    Première validation
```

Maintenant, le but de l'exercice est de créer deux nouveaux commits :

1. modification du fichier README existant
2. ajout d'un nouveau fichier (c'est à vous de choisir un nom cette fois-ci)

Pensez à toujours ajouter à l'index les modifications que vous allez valider. Nous allons voir comment supprimer un fichier de notre dépôt.

1. Supprimez le fichier `salut.rb~`
2. Examinez l'état du dépôt ainsi que son contenu. Que constatez-vous ?
3. Récupérez le fichier supprimé en suivant l'indication donnée par la commande `git status`
4. Vérifiez l'état du dépôt ainsi que son contenu. Que constatez-vous ?
5. Utilisez la commande `git rm` afin de supprimer ce fichier
6. Examinez l'état du dépôt ainsi que son contenu. Que constatez-vous ?
7. Enregistrez un instantané de votre dépôt. Indiquez un message bien explicite.
8. Examinez l'état du dépôt ainsi que son contenu. Que constatez-vous ?

### 2.3.5 Choisir les fichier à ignorer

Dans notre projet, il reste un fichier dont le nom termine par un `"~"`. Cela indique un fichier utilisé par l'éditeur de texte comme copie temporaire. Chaque fois qu'on édite un fichier, ce genre de fichier temporaire est créé. On ne veut pas que ce type de fichiers soit suivis par git.

On peut préciser une liste de fichiers que git va ignorer. Cette liste est dans un fichier `.gitignore`, chaque ligne de ce fichier indique un modèle de nom de fichier à ignorer.

Ainsi, si on veut que git ignore :

- tous les fichiers finissant par "~",
  - tous les fichiers d'un sous-répertoire log
- on va créer le fichier `.gitignore` afin qu'il contienne :

```
*~  
log/
```

Les fichiers correspondant à cette description seront ignorés. Attention, s'ils sont déjà dans le versionning, ils continueront d'être suivis.

### Exercice 4 Git Ignore

Nous allons dire à git d'ignorer les fichiers finissant par "~" et les fichiers d'un répertoire log.

Commencez par créer un répertoire `log` ainsi qu'un fichier dans ce répertoire, et vérifiez que git s'en est bien rendu compte :

```
$ mkdir log  
$ touch log/exemple.log  
$ git status  
Fichiers non suivis:  
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)  
  log/
```

Créer un fichier `.gitignore` avec comme contenu :

```
log/
```

Puis voyez comment git voit le dépôt :

```
$ git status  
Fichiers non suivis:  
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)  
  .gitignore
```

Le répertoire log n'est plus vu par git, il l'ignore !

Maintenant, pour les fichiers terminant par un "~", procédons en 2 étapes :

D'abord retirer les fichiers du versionning :

```
$ git rm --cached *~
```

Cette étape nous laisse avec les fichiers terminant par "~" toujours présents dans le répertoire de travail, mais git les a supprimés du versionning.

Maintenant, pour les ignorer à tout jamais, ajouter la ligne correspondante dans `.gitignore` :

```
*~
```

On termine en ajoutant `.gitignore` dans le suivi de git et en faisant un commit.

## 2.4 Conclusion

Nous savons maintenant :

- comment fonctionne un dépôt git
- comment faire un instantané de notre projet
- consulter l'état d'un dépôt git

Nous allons maintenant voir comment manipuler les instantanés pour remonter l'histoire.

## Chapitre 3

# Manipuler Git

Dans ce chapitre, nous verrons comment manipuler l'historique de notre projet, faire des changements importants dans notre projet sans avoir peur de perdre notre travail.

### 3.1 Visualiser l'historique

Nous avons déjà fait connaissance avec la commande `git log`. Voyons plus en détail ses possibilités pour visualiser un historique.

Pour connaître les différences introduites par un instantané, on utilise l'option `-p` (pour patch) :

```
$ git log -p
commit 2f051e6a237ea54a01bc9377c9f1508d6332b61f
Author: Jonathan Schaeffer <schaeffer@univ-brest.fr>
Date:   Tue Aug 18 15:36:24 2015 +0200
    Suppression du fichier

diff --git a/mon_fichier.txt b/mon_fichier.txt
deleted file mode 100644
index 7d72abc..0000000
--- a/mon_fichier.txt
+++ /dev/null
@@ -1,0,0 @@
-Contenu
$
```

Pour une sortie moins verbeuse, plus résumée, on peut utiliser :

```
$ git log --pretty='%h %ad | %s%d [%an]' --graph --date=short
* 2f051e6 2015-08-18 | Suppression du fichier (HEAD, master)
* bf67c41 2015-08-18 | Premier commit [Jonathan Schaeffer]
$
```

Pour éviter de se souvenir de cette commande pratique, on peut créer un alias, qu'on appellera `git hist` :

```
$ git config --global alias.hist \
"log --pretty=format:%h %ad | %s%d [%an]' --graph --date=short"
$
```

## 3.2 Annuler des actions

On a vu jusqu'à présent comment déplacer les modifications de notre dépôt vers son état de droite (cf. la représentation 2.1) :

- état modifié  $\Rightarrow$  indexé : `git add mon_fichier`
- état indexé  $\Rightarrow$  instantané enregistré : `git commit`

### 3.2.1 Corriger un commit

Voyons d'abord comment corriger un commit, par exemple dans le cas où notre index n'était pas complet au moment du dernier commit (oubli d'ajouter un fichier).

La première étape est de compléter notre index, à l'aide de la commande `git add`. Lorsque l'index est complet, on veut le charger, non pas dans un nouveau commit, mais dans le dernier commit effectué. Pour cela, on utilise la commande :

```
git commit --amend
```

Le message du dernier commit apparaît, ce qui nous permet de le changer, et on voit dans la liste des changements les nouvelles modifications apportées par l'index.

Ainsi, le workflow complet est le suivant :

```
$ git commit -m 'validation initiale'
$ git add fichier_oublie
$ git commit --amend
$
```



### 3.2.2 Désindexer un fichier indexé

Dans le cas où on a indexé plusieurs modifications (avec la commande `git add`), imaginons qu'on souhaite retirer un des fichiers indexés pour ne pas qu'il soit intégré au prochain instantané.

La commande `git status`, qui nous informe de l'état du dépôt, nous donne les indications nécessaires pour effectuer cette opération :

```
$ git status
Sur la branche master
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    plonegroup.rb
    schema.rb
aucune modification ajoutée à la validation mais des fichiers non suivis
sont présents (utilisez "git add" pour les suivre)
$ git add .
$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)
    nouveau fichier : plonegroup.rb
    nouveau fichier : schema.rb
$ git reset HEAD schema.rb
$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)
    nouveau fichier : plonegroup.rb
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    schema.rb
$
```

Avec la commande `git reset HEAD unfichier` on peut donc extraire un fichier de l'index.

### 3.2.3 Annuler les modifications locales

Pour annuler toutes les modifications faites à un fichier qui n'ont pas encore été ajoutées à l'index, la commande est `git checkout -- monfichier`

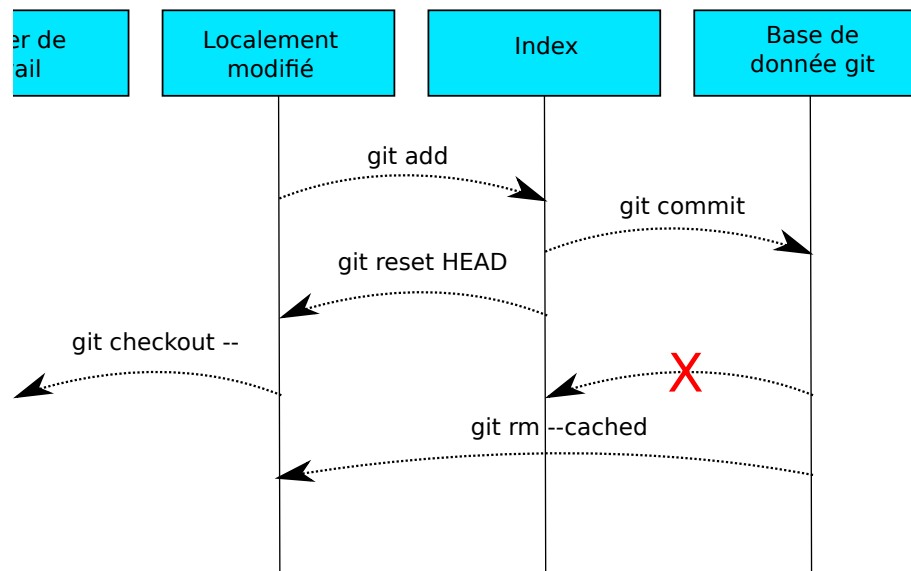


FIGURE 3.1 – Les 3 états dans un dépôt Git et leurs transitions

Il est important de noter que cette commande est destructive ! En l'utilisant vous retrouvez votre fichier dans l'état de son dernier commit, les modifications non indexées ont été supprimées pour de bon.

Cette commande n'a aucun effet si le fichier est déjà ajouté à l'index.

La figure 3.1 illustre l'effet de la commande d'annulation. Notez qu'une fois que des données sont en base, il n'est pas possible de revenir en arrière.

### Exercice 5 Manipuler les états

Dans votre dépôt, commencez par créer 2 nouveaux fichiers en y mettant du contenu.

1. Ajoutez un des 2 fichiers à l'index
2. Faites un commit de votre index
3. Examinez l'état de votre dépôt (`git status`)
4. Ajoutez le 2eme fichier à l'index
5. Faites un nouveau commit
6. Regardez les logs
7. Corrigez le dernier message du dernier commit
8. Ajoutez une modification à l'un des fichiers
9. Corrigez le dernier commit pour qu'il intègre cette modification

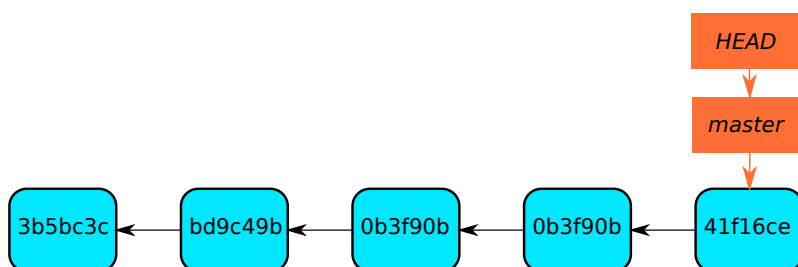


FIGURE 3.2 – Représentation de la chaîne des instantanés

### Exercice 6 Retour en arrière

Nous allons maintenant nous exercer à revenir en arrière dans notre dépôt :

1. Modifiez un fichier et ajoutez-le à l'index
2. Retirez votre modification de l'index
3. Annulez votre modification pour retrouver l'état du dernier commit

## 3.3 Manipuler l'historique

Nous avons présenté de nouvelles commandes dans la section 3.2, sans vraiment explorer toutes leurs possibilités ni expliciter certains détails. Qu'est ce que **HEAD** ? Qu'est ce qu'un **checkout** ?

Nous allons étudier ces aspects en manipulant notre historique d'instantanés.

### 3.3.1 La représentation de l'historique

Les instantanés successifs peuvent être représentés comme formant une chaîne dans le temps. La figure 3.2 illustre cette représentation.

**HEAD** est un pointeur utilisé par git pour référencer le commit courant.

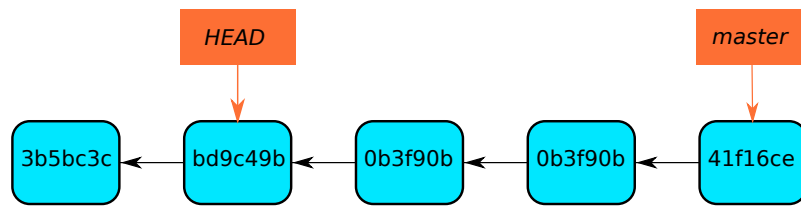
**master** est un autre pointeur. Il représente le commit courant de la branche "master".

Nous aborderons la notion de branches dans la section 3.4

La représentation de la figure 3.2 nous accompagnera pour comprendre l'effet des commandes que nous allons manipuler maintenant.

### 3.3.2 Se balader dans l'historique

Remonter l'historique d'un projet et explorer son état à un instant, revient à déplacer le pointeur **HEAD** dans notre historique. Notre dépôt présente le projet dans l'état pointé par **HEAD**.

FIGURE 3.3 – Effet de la commande `git checkout bd9c49b`

On déplace le pointeur `HEAD` grâce à la commande `checkout`, qui prend en paramètre le numéro de série de l'instantané ciblé :

```
$ git hist
* 7632086 2015-08-20 | Commit supplémentaire (HEAD, master)
* 2f051e6 2015-08-18 | Suppression du fichier [Jonathan Schaeffer]
* bf67c41 2015-08-18 | Premier commit [Jonathan Schaeffer]
$ git checkout bf67c41
Note: checking out '2f051e6'.
$ git hist
* bf67c41 2015-08-18 | Premier commit (HEAD) [Jonathan Schaeffer]
```

On est remontés dans le temps. On peut regarder l'état du projet à ce moment là.

Notez que le message affiché par git lors du checkout explique la situation "detached HEAD", ou "HEAD détaché". Nous détaillerons cela dans la section 3.4. Pour le moment, nous n'allons rien modifier. La figure 3.3 représente ce qui vient de se passer suite à la commande `git checkout`.

Mais où sont passés tous les commits postérieurs ? Remarquez que, dans l'affichage des logs, on ne voit plus apparaître le pointeur `master`. Pour retrouver notre projet complet, il faut retourner à `master` :

```
$ git checkout master
```

### 3.3.3 Annuler le dernier commit

Avec git, il ne faut pas prendre la mauvaise habitude de vouloir réécrire l'histoire. Si on souhaite annuler le dernier commit, par exemple, la meilleure pratique est de créer un nouveau commit qui va inverser toutes les modifications du dernier instantané.

Pour y parvenir, la commande à utiliser est :

```
git revert HEAD
```

Cette commande va créer un nouveau commit qui annule les modifications enregistrées dans le dernier commit. Ainsi, on voit la série de commande suivante :

```
$ git revert HEAD
[master f061f93] Revert "Commit supplémentaire"
 2 files changed, 107 deletions(-)
 delete mode 100644 plonegroup.rb
 delete mode 100644 schema.rb
$ git hist
* f061f93 2015-08-20 | Revert "Commit supplémentaire" (HEAD, master)
* 7632086 2015-08-20 | Commit supplémentaire [Jonathan Schaeffer]
* 2f051e6 2015-08-18 | Suppression du fichier [Jonathan Schaeffer]
* bf67c41 2015-08-18 | Premier commit [Jonathan Schaeffer]
```

Et nous nous retrouvons dans le même état que le commit 2f051e6.

### 3.3.4 Les pointeurs

Encore un mot sur les pointeurs utilisés par git. Bien comprendre leur comportement s'avérera très utile dans votre utilisation de cet outil.

Lorsqu'on crée un nouveau commit, git crée l'instantané avec son numéro de référence et déplace le pointeur HEAD, ainsi que le pointeur de la branche courante (**master** par défaut) pour les placer sur ce dernier commit.

### 3.3.5 Gérer des étiquettes simples

Au cours de votre projet, vous voudrez marquer certains instantanés particulièrement importants. Par exemple, votre développement a atteint une version stable.

Git permet de mettre un pointeur personnalisé, appelé **tag**, sur n'importe quel instantané :

```
$ git tag version-1.0
$ git hist
* f061f93 2015-08-20 | Revert ... (HEAD, tag: version-1.0, master)
```

On peut ensuite retourner sur un tag passé avec la commande **checkout**

## Exercice 7 Gérer l'historique

Avant de manipuler l'historique, ajoutez à votre configuration l'alias **git hist** qui vous sera très pratique :

```
$ git config --global alias.hist "log --pretty=format:'%h %ad'
```

Maintenant, vous allez remonter au premier instantané de votre projet. Vérifiez, avec la commande `git hist` l'historique de votre projet. Vérifiez ensuite que votre projet est bien dans l'état initial.

Placez une étiquette (**tag**) sur ce commit initial.

Retournez maintenant au dernier commit de votre projet.

Puis, utilisez l'étiquette posée précédemment pour retourner à cet instantané.

Enfin, retournez dans le "présent".

## 3.4 Les branches de développement

Jusqu'ici nous avons pensé notre historique de façon linéaire. Mais il est possible, et très facile, de créer points de divergence dans notre projet. Par exemple, vous arrivez à un point où votre projet est stable. Il vous faut maintenant étudier une nouvelle fonctionnalité, mais vous ne voulez pas toucher à la version stable. Par contre, vous voulez garder la possibilité de corriger des éléments dans la version stable, donc y revenir ensuite.

Git propose un mécanisme très simple pour la gestion des points de divergence dans votre projet. On appelle cela des **branches**.

### 3.4.1 Création de branches

Dans l'exemple illustré par la série de figures 3.4, nous distinguons 3 étapes du développement.

Étape 3.4a, on crée une nouvelle branche, grace aux commandes :

```
$ git branch devel
$ git checkout devel
```

Ces 2 commandes peuvent être raccourcies en :

```
$ git checkout -b devel
```

Étape 3.4b, on travaille sur la branche `devel`, en créant deux nouveaux commits.

À l'étape 3.4c, on doit revenir sur la branche `master` pour travailler sur une autre phase du projet. Pour cela, on commence par retourner sur la branche `master` (déplacer HEAD sur `master`), puis on fait de nouvelles modifications que l'on commit :

```
$ git checkout master
$ git commit
$ git commit
```

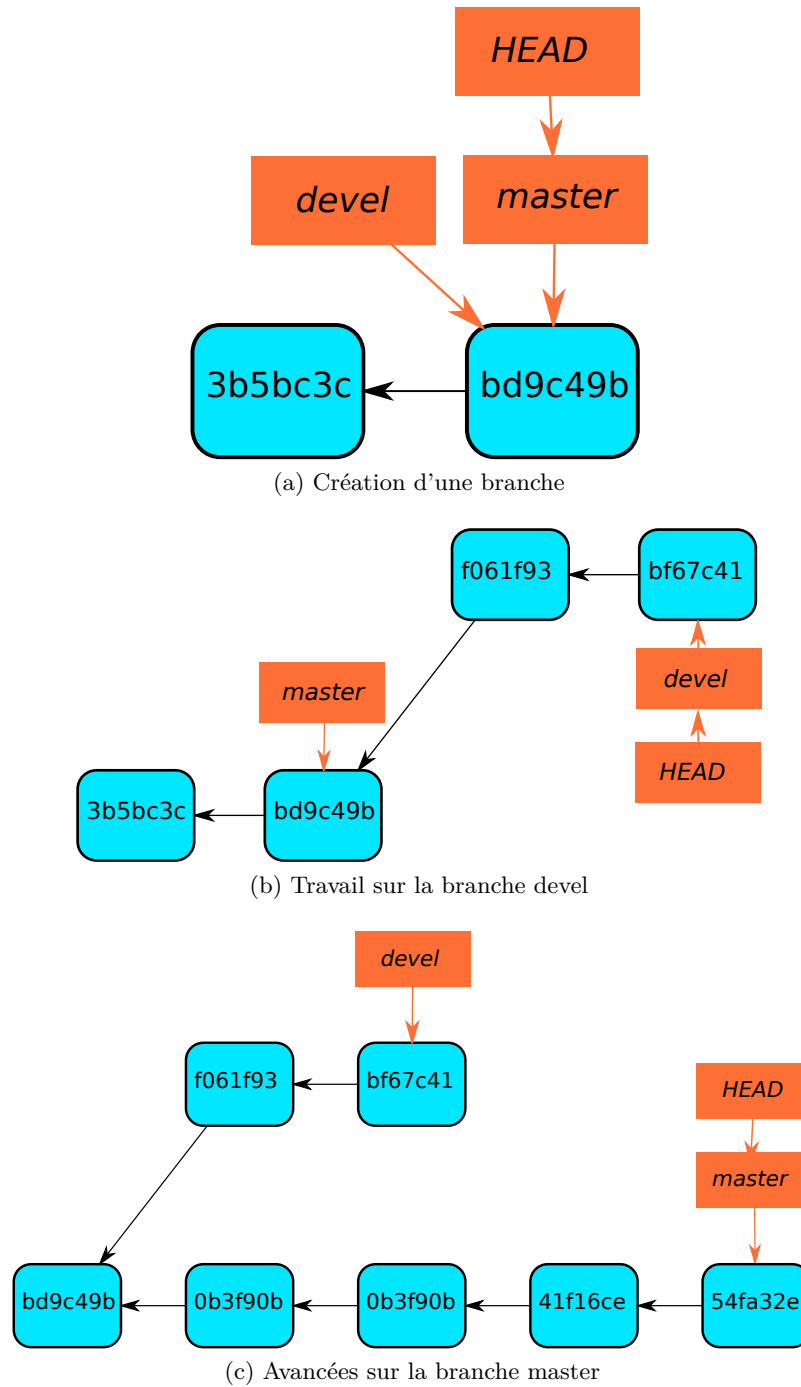
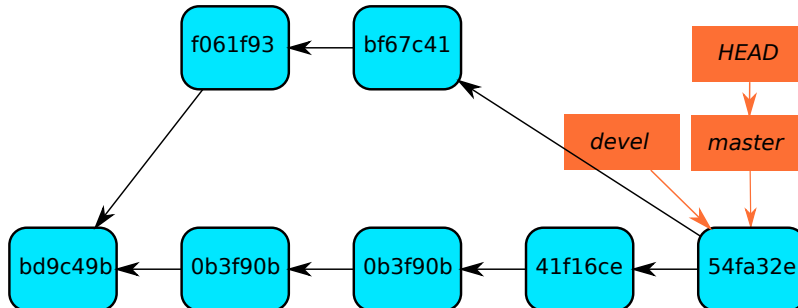


FIGURE 3.4 – États successifs de l'historique dans la gestion des branches

FIGURE 3.5 – Appel à `git merge devel` depuis l'état de la figure 3.4c

### 3.4.2 Fusion de branches

Après un travail dans une branche, on souhaite intégrer notre développement dans la branche principale.

Deux méthodes se présentent alors : le **merge** et le **rebase**.

Pour intégrer notre travail de la branche `devel`, sur la branche `master`, il faut se placer sur la branche `master`, et demander à git de fusionner la branche `devel`.

```
$ git checkout master
$ git merge devel
```

Cette opération crée un nouveau commit constitué des modifications de la branche `devel` appliquées au dernier commit de la branche `master`. Cette opération est illustrée par la figure 3.5.

Afin d'éviter les confusions, ce cours n'aborde que les fusions par merge.

### 3.4.3 Résolution de conflits

Il se peut que notre fusion concerne des parties de codes qui ont été modifiées indépendamment dans les 2 branches. Dans ce cas, git ne peut pas déterminer seul quel est la modification à intégrer.

Par exemple les commit `0b3f90b` et `f061f93` concernent un même paragraphe du même fichier. On dit alors qu'il y a un conflit.

Pas de panique, la résolution de conflits est extrêmement simple, mais il faut agir avec méthode.

Voici un exemple :

1. Création d'une branche `devel`



2. Modifications dans un fichier "index.html" sur la branche devel
3. Commit des modifications
4. Retour sur la branche master
5. Modification au même endroit dans le fichier "index.html"
6. Commit
7. Fusion de la branche devel

Cette dernière étape se présente ainsi :

```

<<<<<<< HEAD
<h1>Nouveau titre</h1>
=====
Nouvelle table :
</table>

<table>
>>>>>>> devel

```

Elle nous montre 2 zones :

- séparées par des signes =====
- bornées en haut par <<<<<<< HEAD
- et en bas par >>>>>>> devel

C'est à nous de choisir quelle zone est celle qu'il faut conserver, et c'est à nous de supprimer la zone à jeter. Il faut être très attentif à bien supprimer les balises que git a ajouté dans notre code, afin qu'elles ne soient pas sauvegardées.

Une fois le conflit résolu, il faut valider les modifications, avec un `git commit`.

## Exercice 8 Les branches

Dans cet exercice, vous allez manipuler des branches.

Premièrement, créez une nouvelle branche et apportez-y des modifications (un nouveau fichier par exemple). Faites plusieurs commits.

Ensuite, vous avez besoin de revenir sur la branche master afin de corriger quelque chose. Ramenez votre projet sur la branche master et faites une série de commits.

Enfin, vous pouvez revenir sur la branche de développement pour continuer à travailler.

## Exercice 9 Plus de branches

Trouvez et écrivez la séquence de commandes permettant d'arriver à l'enchaînement d'instantanés de la figure 3.6.

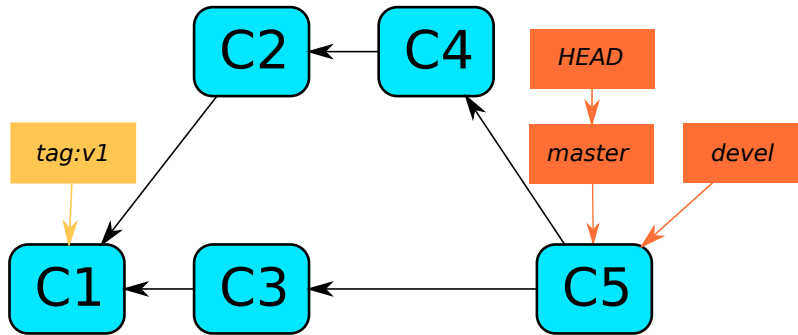


FIGURE 3.6 – Objectif de l'exercice 9

## Chapitre 4

# Git : un système décentralisé

Le grand avantage de git est qu'il permet de gérer très simplement un dépôt local sans lourdeur.

Mais git a aussi été conçu pour favoriser le travail collaboratif. Sa philosophie est d'être complètement décentralisé. Vous pouvez connecter votre dépôt git à un serveur distant et le synchroniser, travailler localement sans connexion et le resynchroniser plus tard.

Vous pouvez également partager avec plusieurs personnes un même dépôt distant. Chacun peut travailler de façon indépendante et se resynchroniser avec un autre dépôt.

Chaque dépôt connecté à un autre est un clone. Voyons les possibilités de git dans ce domaine.

### 4.1 Utiliser un dépôt distant

Dans un souci de facilité, voyons comment cloner un dépôt distant déjà existant.

#### 4.1.1 Cloner le dépôt

La commande **clone** va créer localement un dépôt, en lien avec un autre dépôt distant (il peut aussi être sur la même machine).

Par exemple :

```
$ git clone git://tucuxi.univ-brest.fr/formation_git
Clonage dans 'formation_git'...
Vérification de la connectivité... fait.
$ cd formation_git
$ git hist
```

Le dépôt d'origine est nommé "origin" on peut voir la liste des dépôts distants avec la commande suivante :

```
$ git remote show
origin
$ git remote show origin
```

Cette dernière commande vous montrera les références du dépôt distant d'origine.

On travaille dans un dépôt git cloné exactement de la même manière que ce qui a été présenté jusqu'ici.

Toutes les commandes git utilisées vont concerner notre dépôt local.

#### 4.1.2 Synchroniser un dépôt cloné

En partant de l'hypothèse que nous sommes le seul utilisateur du dépôt, il nous suffit, lorsqu'on souhaite se synchroniser, de *pousser* notre base de donnée git locale vers le dépôt distant.

La commande **push** permet de faire cette opération :

```
$ git push origin master
X11 forwarding request failed on channel 0
Décompte des objets: 42, fait.
Delta compression using up to 4 threads.
Compression des objets: 100% (39/39), fait.
Écriture des objets: 100% (42/42), 464.02 KiB | 0 bytes/s, fait.
Total 42 (delta 17), reused 0 (delta 0)
To gitium:formation_git
 * [new branch]      master -> master
```

Cette commande peut être raccourcie, si vous synchronisez la branche master avec le dépôt d'origine, en :

```
$ git push
```

Maintenant, la commande **status** donne une nouvelle information :

```
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
```

### Exercice 10 Travail avec un dépôt distant

Plutôt que de partir d'un dépôt distant vide, nous allons brancher votre dépôt utilisé jusqu'à présent sur un serveur distant et nous utiliserons pour cela le compte Framagit que

vous avez créé au début du cours (référez-vous au courriel de confirmation que vous devez avoir obtenu depuis).

Créez un nouveau projet bien à vous : <https://git.framasoft.org/projects/new>. Choisissez un nom, une description et sélectionnez "internal" ou "public".

```
$ cd mon_depot
$ git remote add origin https://git.framasoft.org/glagaffe/mon_depôt.git
$ git push -u origin master
```

Vous venez de connecter votre dépôt local à un dépôt distant et de les synchroniser. Félicitations !

## 4.2 Travail collaboratif

Git facilite grandement le travail collaboratif. Toute une équipe de développeurs peut se brancher sur un même dépôt. Chacun travaille localement et pousse ses modifications régulièrement.

Git permet à une équipe de mettre en oeuvre plusieurs systèmes de collaboration par sa souplesse et sa versatilité.

À partir du moment où l'on utilise un dépôt distant, la règle d'or est de ne jamais toucher à une chaîne de commits qui a déjà été poussée sur le serveur. Notamment, ne pas faire de `git commit --amend`

Dans le cadre de ce cours, nous allons nous pencher sur un workflow où une équipe de développeur travaille directement sur le dépôt distant.

### Exercice 11 Travailler en binôme

Par binôme, choisissez un dépôt distant et clonez-le chacun localement.

Ajoutez du contenu, chacun de son côté, dans deux répertoires distincts.

Une fois que plusieurs commits ont été réalisés par chacun, choisissez la personne qui va synchroniser son dépôt en premier : ce sera Alice. L'autre personne (Bernadette) va ensuite faire un `pull` pour récupérer les derniers commits du dépôt distant. Git voudra créer un nouveau commit pour la fusion de vos 2 dépôts. Une fois ce commit effectué, il suffira à Bernadette de pousser à nouveau son travail vers le serveur. Alice va faire un `git status`. Que constatez-vous ? Synchronisez ensuite le dépôt d'Alice avec un `git pull`.

Répétez l'opération en inversant les rôles.

### Exercice 12 Avec des conflits

Maintenant, nous allons refaire des modifications, mais dans le même fichier.

Commencez par tester les capacités de git à résoudre lui-même les conflits. Alice va éditer du code en tête de fichier. Bernadette va éditer à la fin du fichier.

Réitérez l'opération de synchronisation. Un conflit a-t-il lieu ?

Répétez l'exercice en modifiant chacun une même zone d'un même fichier.

# Chapitre 5

## Conclusion

### 5.1 Résumé du contenu du cours

Après cette mise en pratique de git, vous savez :

- versionner votre travail local
- revenir dans le passé
- manipuler des branches pour oser diverger de votre code
- travailler en collaboration sur un même code

### 5.2 Ce qu’il reste à explorer

Git permet beaucoup de souplesse, notamment dans la gestion des branches, des branches distantes.

Git permet de monter son propre workflow pour le travail collaboratif.

Par ailleurs, git est souvent intégré à vos environnements de développement préférés. Il existe des interfaces graphiques à Git qui peuvent également être utiles. Ce cours les a volontairement évitées afin que vous sachiez manipuler git dans toutes les situations.

### 5.3 Références utiles

Le guide ProGit en français : <https://git-scm.com/book/fr/v2/> est consultable en ligne, illustré et très didactique

Le serveur Git de l’IUEM, adossé à une forge logicielle :

- <https://www-iuem.univ-brest.fr/feiri/services/SYS008>
- <https://www-iuem.univ-brest.fr/feiri/soutiens-scientifique/calcul-scientifique/wiki/utilisation-de-git>

Un serveur Git librement accessible et basé sur des logiciels libres : <http://framacode.org/gitlab.php>, si vous n’avez pas la chance d’avoir accès au serveur git de l’IUEM.

Une feuille de triche Git très utile : <http://www.cheat-sheets.org/saved-copy/git-cheat-sheet.pdf>



Si tout ça n'a pas fonctionné, j'ai un fichier `git.txt` contenant le numéro de téléphone d'un ami qui comprend git. Attends quelques minutes de « C'est vraiment simple, pense à une branche comme si c'était un ... » et à la fin, tu auras la commande qui réparera tout.



# Solution des exercices

## Solution de l'exercice 3

Pour supprimer un fichier dans un dépôt git, on utilise la commande `git rm`.  
Par exemple :

```
$ git rm mon_fichier.txt
$ git commit
[master 2f051e6] Suppression du fichier
1 file changed, 1 deletion(-)
delete mode 100644 mon_fichier.txt
```

On aura la possibilité d'annuler cette suppression comme on le verra dans la section ??  
La séquence de suppression demandée dans l'exercice est la suivante :

1. `rm salut.rb~`
2. Le fichier est supprimé, mais il y a des modifications à indexer.
3. `git checkout -- salut.rb~`
4. le fichier est à nouveau présent
5. `git rm salut.rb~`
6. L'index est prêt pour un instantané
7. `git commit -m "Suppression du fichier salut.rb~"`
8. Le dépôt est dans un état propre.

## Solution de l'exercice 5

```
$ git add fichier1
$ git commit
$ git status
$ git add fichier2
$ git commit
$ git log
$ git commit --amend
$ git add fichier2 # après modification de fichier2
$ git commit --amend
```

#### Solution de l'exercice 6

```
$ git add fichier1
$ git reset fichier1
$ git checkout -- fichier1
```

#### Solution de l'exercice 7

Les commandes sont les suivantes :

```
$ git checkout bf67c41
$ git tag init
$ git checkout master
$ git checkout init
$ git checkout master
```

#### Solution de l'exercice 8

```
$ git checkout -b devel
$ git commit
$ git commit
$ git checkout master
$ git commit
$ git commit
```

#### Solution de l'exercice 9

C1 Faire des modifications  
\$ git commit -a

```
$ git tag v1
```

**C2**    \$ git branch devel  
         \$ git checkout devel  
         Faire des modifications  
         \$ git commit -a

**C3**    \$ git checkout master  
         Faire des modifications  
         \$ git commit -a

**C4**    \$ git checkout devel  
         Faire des modifications  
         \$ git commit -a

**C5**    \$ git checkout master  
         \$ git merge devel