

Git, Essayons de reprendre le contrôle !

Antoine de ROQUEMAUREL

 satenske

Développeur Java consultant chez Tech Advantage



Meetup Java / C# du 28 Mars 2019



Cette œuvre est mise à disposition selon les termes de la Licence Creative Commons By 4.0

LE LOGICIEL DE GESTION DE VERSIONS



Have you ever:

- Made a change to code, realised it was a mistake and wanted to revert back?
- Lost code or had a backup that was too old?
- Had to maintain multiple versions of a product?
- Wanted to see the difference between two (or more) versions of your code?
- Wanted to prove that a particular change broke or fixed a piece of code?
- Wanted to review the history of some code?
- Wanted to submit a change to someone else's code?
- Wanted to share your code, or let other people work on your code?
- Wanted to see how much work is being done, and where, when and by whom?
- Wanted to experiment with a new feature without interfering with working code?

In these cases, and no doubt others, a version control system should make your life easier.

To misquote a friend: A civilised tool for a civilised age.

share improve this answer

edited Nov 6 '13 at 0:52

answered Sep 11 '09 at 0:42



si618

14.4k

12

60

79

FIGURE – Pourquoi devrais-je utiliser le contrôle de version ?¹

1. <https://stackoverflow.com/questions/1408450/why-should-i-use-version-control>

GIT



- ▶ Créé en 2005 par Linus Torvalds
- ▶ Décentralisé
- ▶ Excellente gestion des branches
- ▶ Efficace sur de gros projet

GIT



- ▶ Créé en 2005 par Linus Torvalds
- ▶ Décentralisé
- ▶ Excellente gestion des branches
- ▶ Efficace sur de gros projet
 - ▶ Microsoft Windows :
 - ▶ 3 500 000 fichiers, soit 300 Go
 - ▶ 440 branches
 - ▶ 4 000 utilisateurs
 - ▶ 10 000 merges

GIT



- ▶ Créé en 2005 par Linus Torvalds
- ▶ Décentralisé
- ▶ Excellente gestion des branches
- ▶ Efficace sur de gros projet

*« I'm an egotistical bastard, and I name all my projects after myself.
First 'Linux', now 'git'. »*

Fonctionnement de Git

Dans les coulisses

La collaboration

Utilisation avancée

LE SYSTÈME DÉCENTRALISÉ

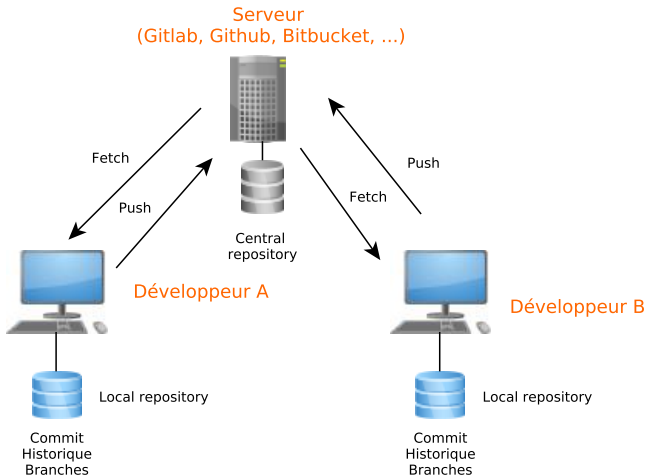


FIGURE – Système décentralisé

LA ZONE DE TRANSIT (*staging area*)

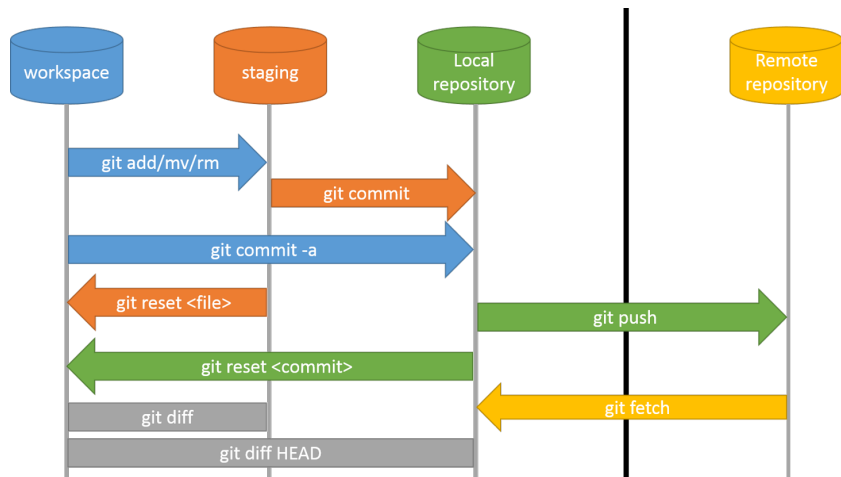


FIGURE – Fonctionnement de Git

Fonctionnement de Git

Dans les coulisses

La collaboration

Utilisation avancée

LE STOCKAGE DES DONNÉES DU DÉPÔT : LES OBJETS

- Les changements de fichiers sont stockés dans le **commit**

1df4fc
first commit

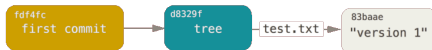
LE STOCKAGE DES DONNÉES DU DÉPÔT : LES OBJETS

- ▶ Les changements de fichiers sont stockés dans le **commit**
- ▶ Un **tree** représente un niveau de hiérarchie de fichiers



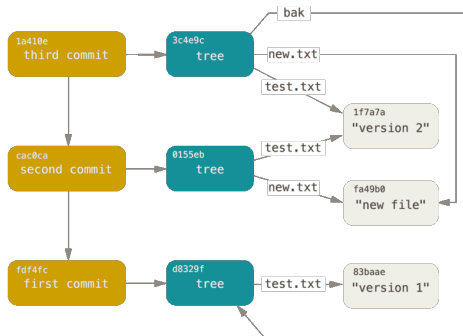
LE STOCKAGE DES DONNÉES DU DÉPÔT : LES OBJETS

- ▶ Les changements de fichiers sont stockés dans le **commit**
- ▶ Un **tree** représente un niveau de hiérarchie de fichiers
- ▶ Chaque version de chaque fichier est un **blob**



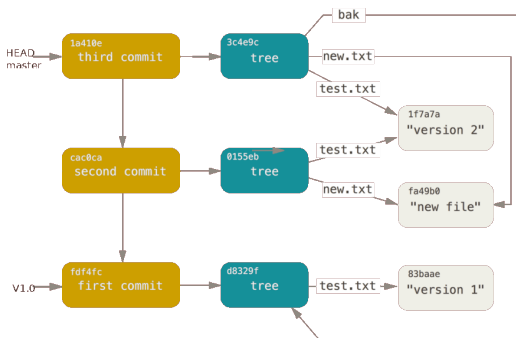
LE STOCKAGE DES DONNÉES DU DÉPÔT : LES OBJETS

- ▶ Les changements de fichiers sont stockés dans le **commit**
- ▶ Un **tree** représente un niveau de hiérarchie de fichiers
- ▶ Chaque version de chaque fichier est un **blob**
- ▶ Les **commits** sont chaînés entre eux



LE STOCKAGE DES DONNÉES DU DÉPÔT : LES OBJETS

- ▶ Les changements de fichiers sont stockés dans le **commit**
- ▶ Un **tree** représente un niveau de hiérarchie de fichiers
- ▶ Chaque version de chaque fichier est un **blob**
- ▶ Les **commits** sont chaînés entre eux
- ▶ Les **tags**, **branches** et **HEAD** sont des pointeurs de commit



LE STOCKAGE DES INFOS DU WORKSPACE : L'INDEX

- ▶ Le fichier contient toutes les informations nécessaires à la génération d'un *tree object*
- ▶ Il permet la comparaison rapide entre un *tree object* et le *working tree*
- ▶ Il contient les informations sur les *merges conflicts*

GIT DÉBITE À LA HASH !

- ▶ Tout objet du store est adressable par son contenu
- ▶ Le nom unique de chaque objet est obtenu avec SHA-1
 - ▶ Valeur sur 160 bits (nombre hexadécimal à 40 chiffres)
- ▶ Toute modification du contenu produira un changement du hash

Fonctionnement de Git

Dans les coulisses

La collaboration

Utilisation avancée

RESTONS BRANCHÉS : LE GIT-FLOW

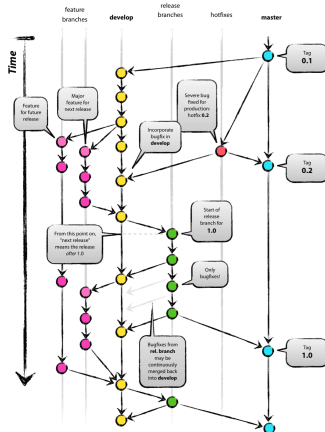
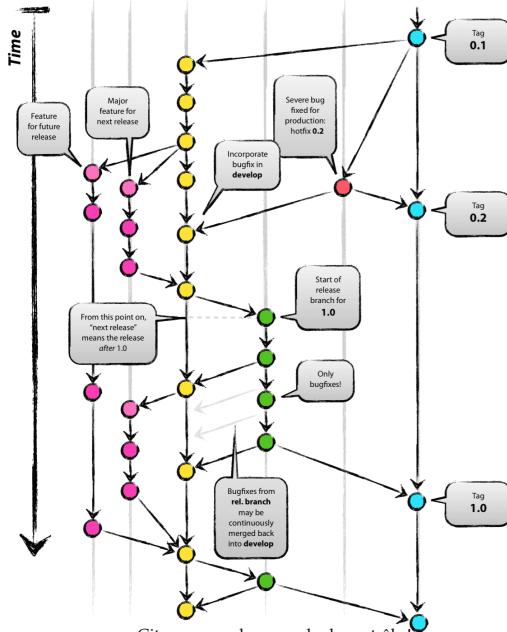


FIGURE – Un modèle de branchement²

2. <https://nvie.com/posts/a-successful-git-branching-model/>

RESTONS BIEN



Git, essayons de reprendre le contrôle !

DE L'IMPORTANCE DE L'HISTORIQUE

Ingrédients :

- ▶ 200g de beurre
- ▶ 200g de chocolat
- ▶ 4 œufs
- ▶ 150g de sucre
- ▶ 60g de farine
- ▶ $\frac{1}{2}$ sachet de levure

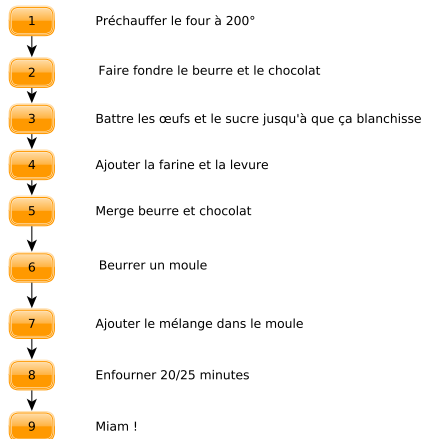
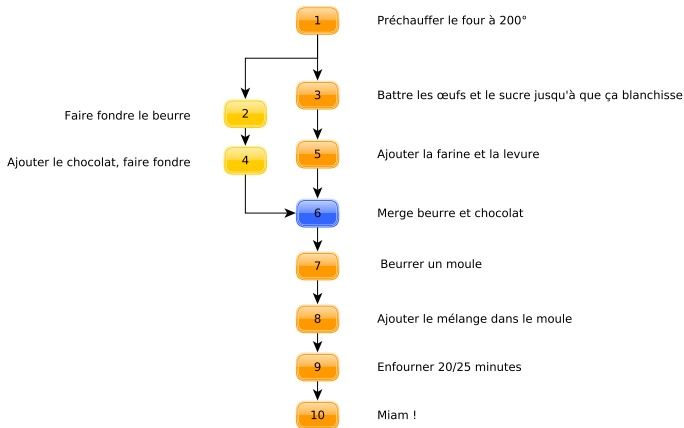


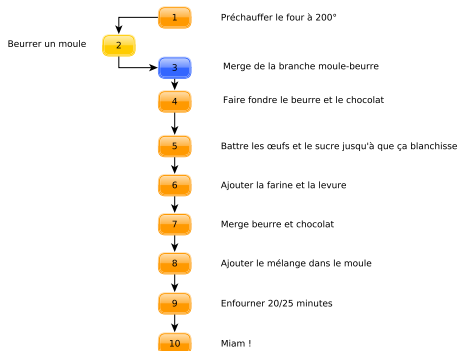
FIGURE – Historique sans collaboration

LE MERGE : ON PARRALÉLISE



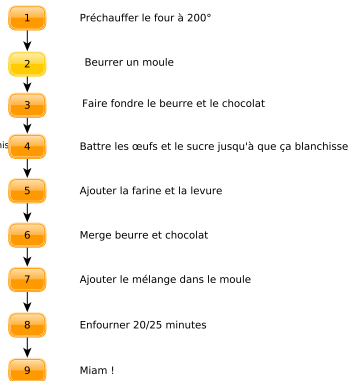
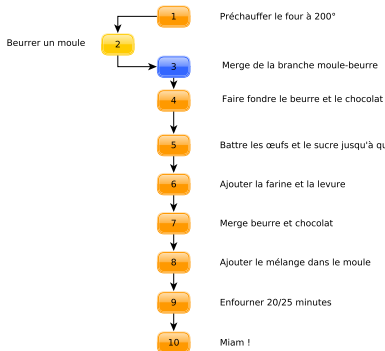
```
1 | $ git merge beurre-chocolat
```

LE MERGE : LE FAST-FORWARD



```
1 | $ git merge --no-ff ↔  
   |     moule-beurre
```

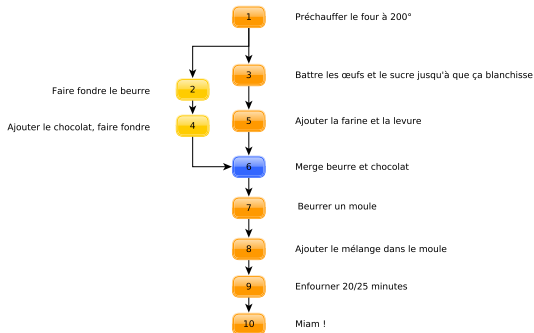
LE MERGE : LE FAST-FORWARD



```
1 | $ git merge --no-ff ←
    moule-beurre
```

```
1 | $ git merge moule-beurre
```

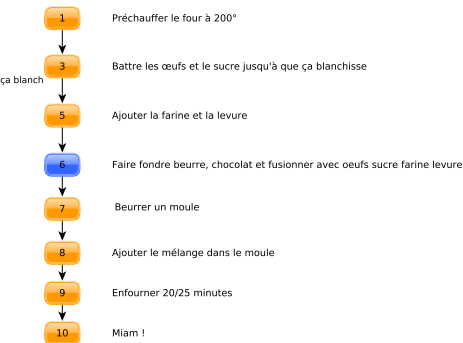
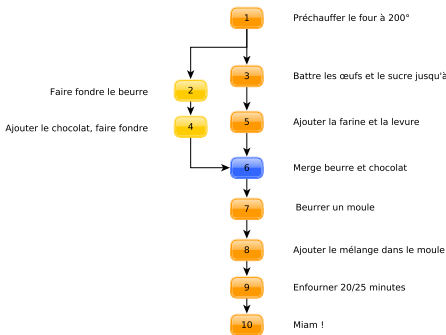
LE MERGE : LE SQUASH



```

1 | $ git merge ←
    beurre-chocolat
  
```

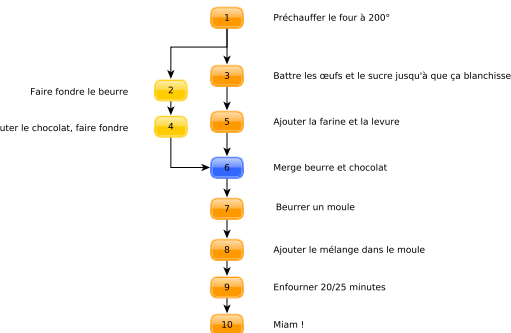

LE MERGE : LE SQUASH



```
1 | $ git merge ←  
    beurre-chocolat
```

```
1 | $ git merge --squash ←  
    beurre-chocolat
```

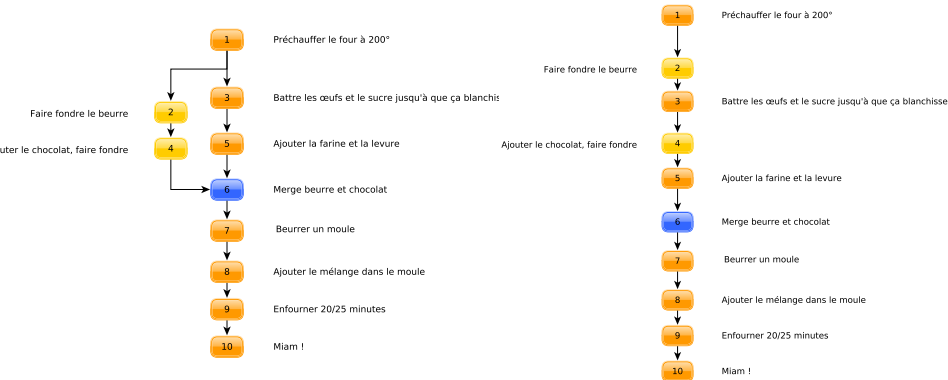
LE REBASE : ON VEUT UN HISTORIQUE LINÉAIRE



```

1 | $ git merge ←
   |     beurre-chocolat
  
```

LE REBASE : ON VEUT UN HISTORIQUE LINÉAIRE



```
1 | $ git merge ←
    beurre-chocolat
```

```
1 | $ git rebase ←
    beurre-chocolat
```

LE REBASE : ATTENTION À SON UTILISATION

- ▶ Jamais sur une branche partagée
 - ▶ Un rebase réécrit l'historique et va donc changer les hash
- ▶ Si la branche à été poussée
 - ▶ Il faut faire un push force... Et donc écraser l'historique de la branche distante!
- ▶ Principalement à utiliser pour :
 - ▶ Mettre à jour sa branche par rapport à la branche mère
 - ▶ Réécrire son historique

LE STASH : METTRE DES MODIFICATIONS DE CÔTÉ

- ▶ Utile pour pouvoir changer de brancher ou se mettre à jour sans commiter
- ▶ Gestion des conflits lors de l'application du stash

LE STASH : METTRE DES MODIFICATIONS DE CÔTÉ

- ▶ Utile pour pouvoir changer de brancher ou se mettre à jour sans commiter
- ▶ Gestion des conflits lors de l'application du stash
- ▶ Le stash est une pile :
 - ▶ `git stash` pour empiler
 - ▶ `git stash pop` pour dépiler
- ▶ Possibilité d'y accéder comme une liste
 - ▶ `git stash list`
 - ▶ `git stash apply stash_name`

LE CHERRY-PICK

Fonctionnement de Git

Dans les coulisses

La collaboration

Utilisation avancée

LE PULL DEVRAIT ÊTRE INTERDIT...

- Par défaut, le pull peut être vu comme un alias :

```
1 | $ git fetch  
  | $ git merge origin/master
```

Un git pull si on est sur master

LE PULL DEVRAIT ÊTRE INTERDIT...

- Par défaut, le pull peut être vu comme un alias :

```
2 | $ git fetch  
  | $ git merge origin/master
```

Un git pull si on est sur master

- Si on est pas en fast-forward, le merge va créer une branche temporaire

LE PULL DEVRAIT ÊTRE INTERDIT...

- Par défaut, le pull peut être vu comme un alias :

```
2 | $ git fetch  
  | $ git merge origin/master
```

Un git pull si on est sur master

- Si on est pas en fast-forward, le merge va créer une branche temporaire
- Ne faire un pull que si on est en fast forward

LE PULL DEVRAIT ÊTRE INTERDIT...

- ▶ Par défaut, le pull peut être vu comme un alias :

```
2 | $ git fetch  
  | $ git merge origin/master
```

Un git pull si on est sur master

- ▶ Si on est pas en fast-forward, le merge va créer une branche temporaire
- ▶ Ne faire un pull que si on est en fast forward
- ▶ Sinon, il faut faire

```
2 | $ git fetch  
  | $ git rebase origin/master
```

Se mettre à jour si on est sur master

LE REBASE INTERACTIF : RÉÉCRIRE L'HISTOIRE

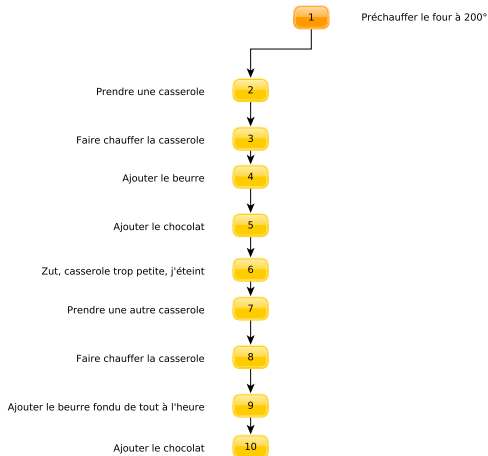


FIGURE – Un historique pourri

LE REBASE INTERACTIF : RÉÉCRIRE L'HISTOIRE

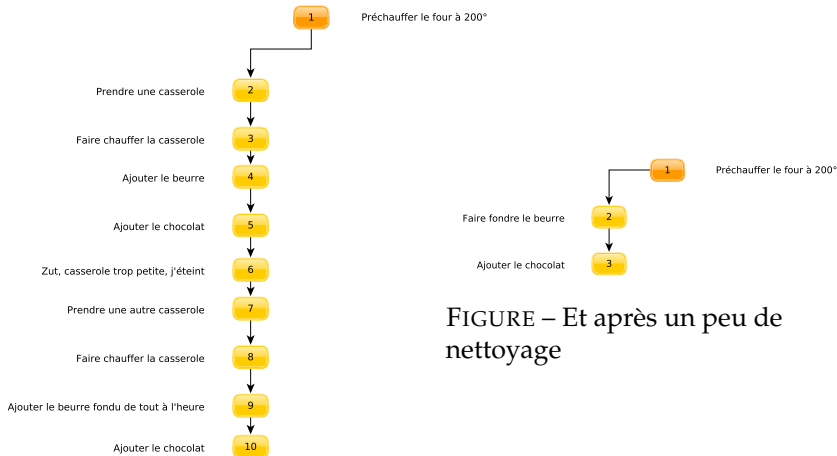


FIGURE – Et après un peu de nettoyage

FIGURE – Un historique pourri

LE REBASE INTERACTIF : RÉÉCRIRE L'HISTOIRE

```
1 | pick f7f3f6d changed my name a bit
2 | pick 310154e updated README formatting and added blame
3 | pick a5f4a0d added cat-file
4 | # Rebase 710f0f8..a5f4a0d onto 710f0f8
```

Utilisation du rebase interactif

- **p, pick** : utiliser le commit (ne change rien)

LE REBASE INTERACTIF : RÉÉCRIRE L'HISTOIRE

```
1 | pick f7f3f6d changed my name a bit
   | pick 310154e updated README formatting and added blame
3 | pick a5f4a0d added cat-file
   |
5 | # Rebase 710f0f8..a5f4a0d onto 710f0f8
```

Utilisation du rebase interactif

- ▶ **p, pick** : utiliser le commit (ne change rien)
- ▶ **r, reword** : utilise le commit, mais permet de changer le message

LE REBASE INTERACTIF : RÉÉCRIRE L'HISTOIRE

```
1 | pick f7f3f6d changed my name a bit
   | pick 310154e updated README formatting and added blame
3 | pick a5f4a0d added cat-file
   |
5 | # Rebase 710f0f8..a5f4a0d onto 710f0f8
```

Utilisation du rebase interactif

- ▶ **p, pick** : utiliser le commit (ne change rien)
- ▶ **r, reword** : utilise le commit, mais permet de changer le message
- ▶ **e, edit** : utilise le commit et s'arrête pour pouvoir changer le contenu du commit

LE REBASE INTERACTIF : RÉÉCRIRE L'HISTOIRE

```
1 | pick f7f3f6d changed my name a bit
   | pick 310154e updated README formatting and added blame
3 | pick a5f4a0d added cat-file
   |
5 | # Rebase 710f0f8..a5f4a0d onto 710f0f8
```

Utilisation du rebase interactif

- ▶ **p, pick** : utiliser le commit (ne change rien)
- ▶ **r, reword** : utilise le commit, mais permet de changer le message
- ▶ **e, edit** : utilise le commit et s'arrête pour pouvoir changer le contenu du commit
- ▶ **s, squash** : fusionne avec le commit précédent

LE REBASE INTERACTIF : RÉÉCRIRE L'HISTOIRE

```
1 | pick f7f3f6d changed my name a bit
   | pick 310154e updated README formatting and added blame
3 | pick a5f4a0d added cat-file
   |
5 | # Rebase 710f0f8..a5f4a0d onto 710f0f8
```

Utilisation du rebase interactif

- ▶ **p, pick** : utiliser le commit (ne change rien)
- ▶ **r, reword** : utilise le commit, mais permet de changer le message
- ▶ **e, edit** : utilise le commit et s'arrête pour pouvoir changer le contenu du commit
- ▶ **s, squash** : fusionne avec le commit précédent
- ▶ **d, drop** : supprime le commit

BISECT : TROUVER D'OÙ VIENT LE BUG

- Savoir depuis quel commit l'application ne fonctionne plus

```
1 | $ git bisect start <bad commit> <good commit>
   | # Quand la version courante est mauvaise :
3 | $ git bisect bad
   | # Quand la version courante est bonne :
5 | $ git bisect good
```

Utilisation de git bisect

BISECT : TROUVER D'OÙ VIENT LE BUG

- Savoir depuis quel commit l'application ne fonctionne plus

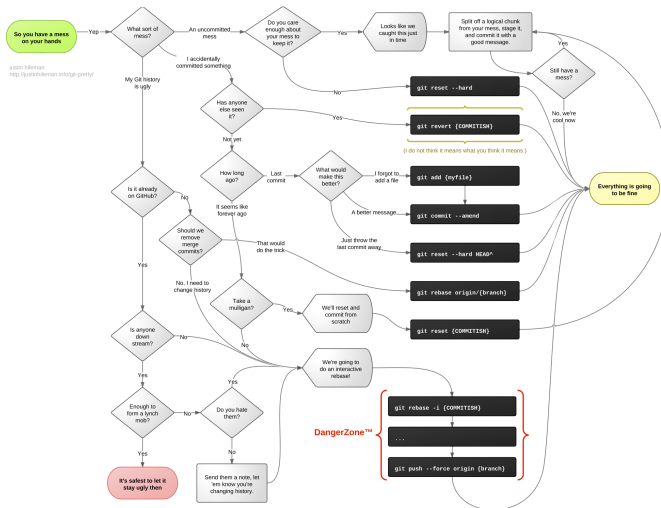
```
1 | $ git bisect start <bad commit> <good commit>
   | # Quand la version courante est mauvaise :
3 | $ git bisect bad
   | # Quand la version courante est bonne :
5 | $ git bisect good
```

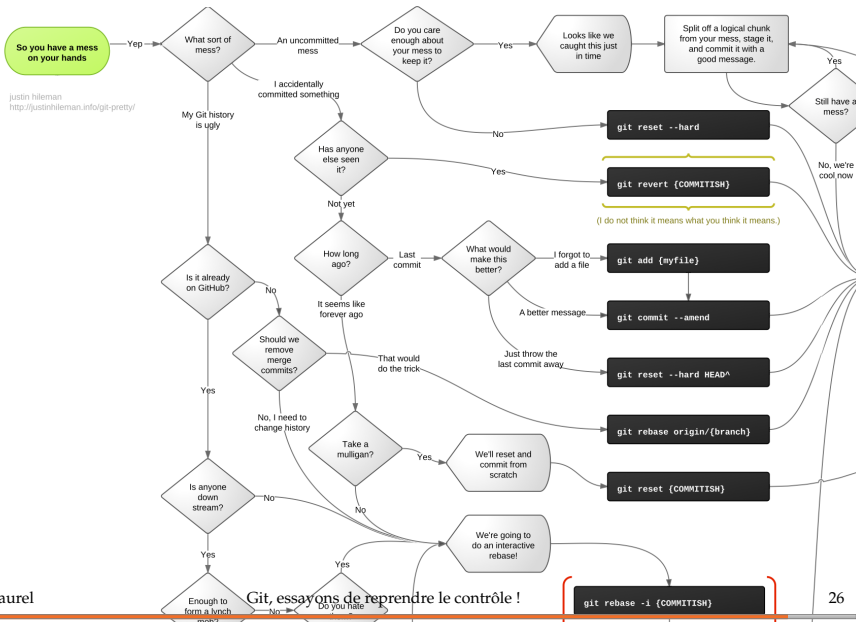
Utilisation de git bisect

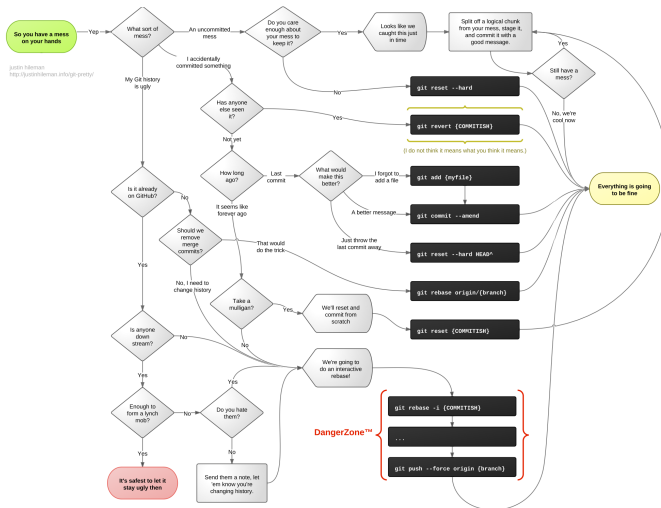
- Et si on a des tests automatisés ?

```
2 | $ git bisect start <bad commit> <good commit>
   | $ git bisect run auto-build-run-tests.sh
```

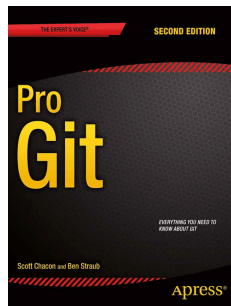
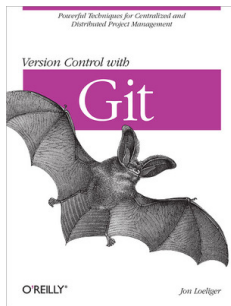
Utilisation de git bisect pour lancer les tests

FIGURE – So, you have a mess on your hands? ²2. <http://justinhileman.info/article/git-pretty/>



FIGURE – So, you have a mess on your hands? ²2. <http://justinhileman.info/article/git-pretty/>

RÉFÉRENCES



- ▶ git-scm.com
Site officiel
- ▶ learngitbranching.js.org
Apprendre Git de manière ludique
- ▶ github.com/aroquemaurel/Presentation-beamer-Git
Les sources \LaTeX de cette présentation

Git, Essayons de reprendre le contrôle !

Antoine de ROQUEMAUREL

 satenske

Développeur Java consultant chez Tech Advantage



Meetup Java / C# du 28 Mars 2019



Cette œuvre est mise à disposition selon les termes de la Licence Creative Commons By 4.0