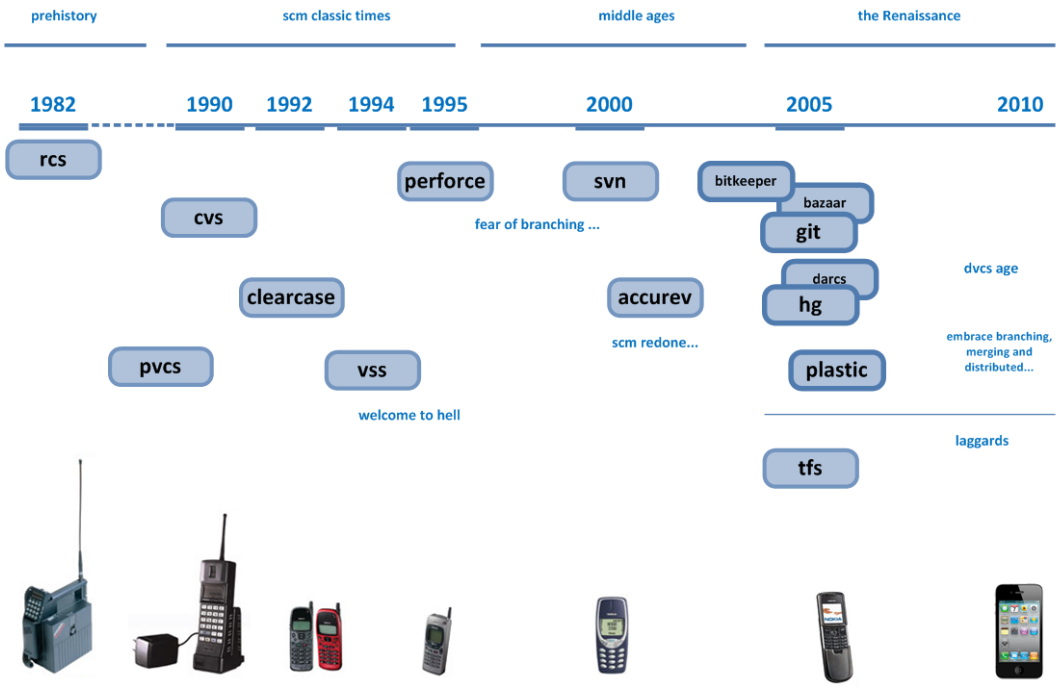
Git, GitHub et le versionnement

Qu'est ce que le versionnement?

- Suivre l'évolution d'un fichier et garder trace de ses évolutions
  - pouvoir revenir en arrière
  - comprendre le pourquoi d'un nouveau comportement du programme
- Les outils de versionnement intègrent la plupart du temps un système de modification concurrente du code, ce qui permet :
  - de savoir qui a fait quelles modifications
  - de gérer des conflits de modifications (sur une même partie du code)

Histoire des outils de verionnement

- Ex : cvs, svn



Edition concurrentielle (1/4)

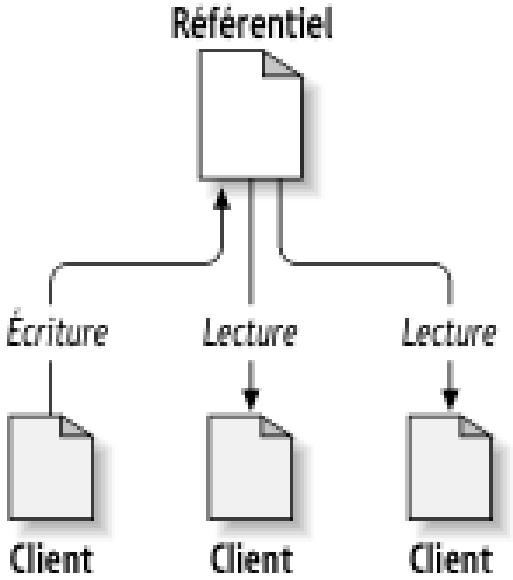


Figure 2: svn

Edition concurrentielle (1/4)

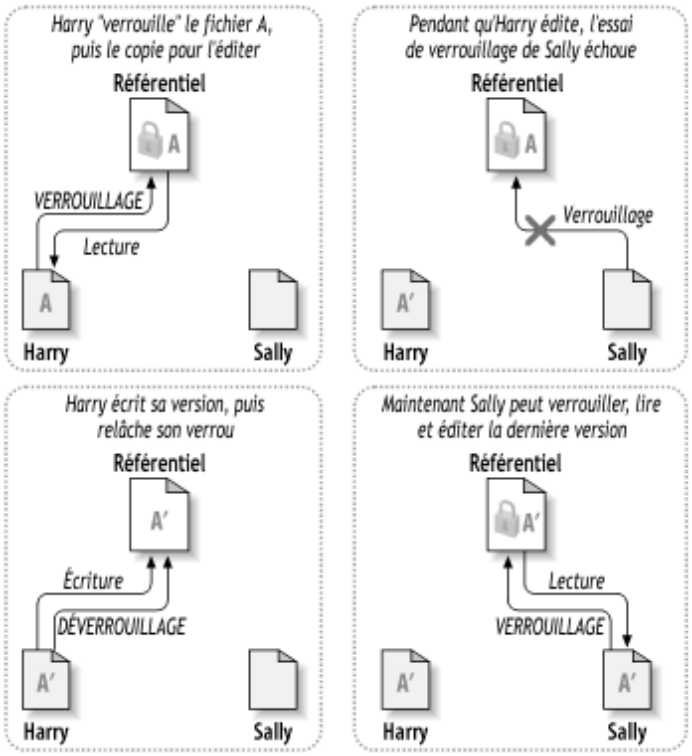


Figure 3: svn

Edition concurrentielle (1/4)

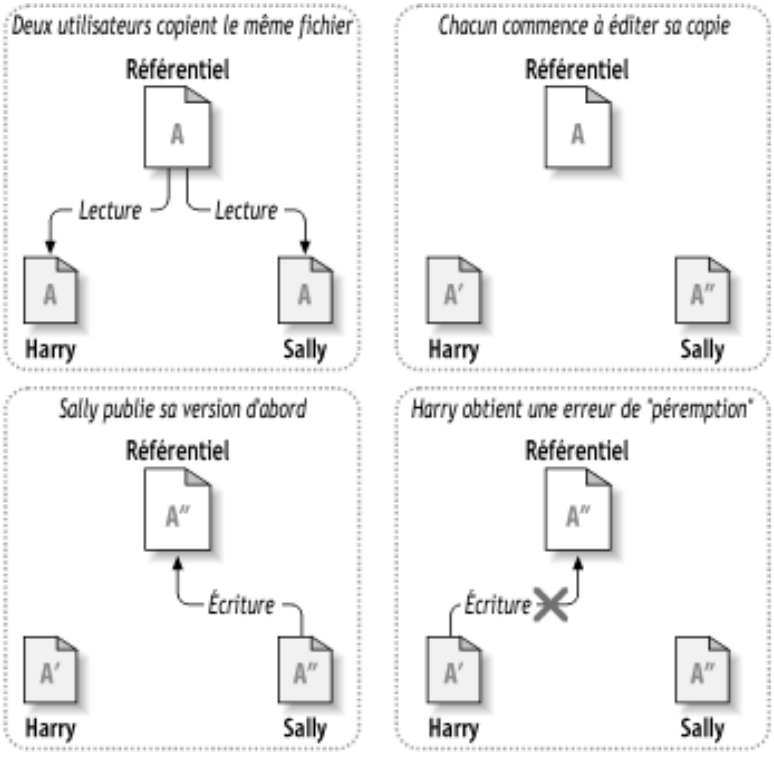


Figure 4: svn

Edition concurrentielle (1/4)

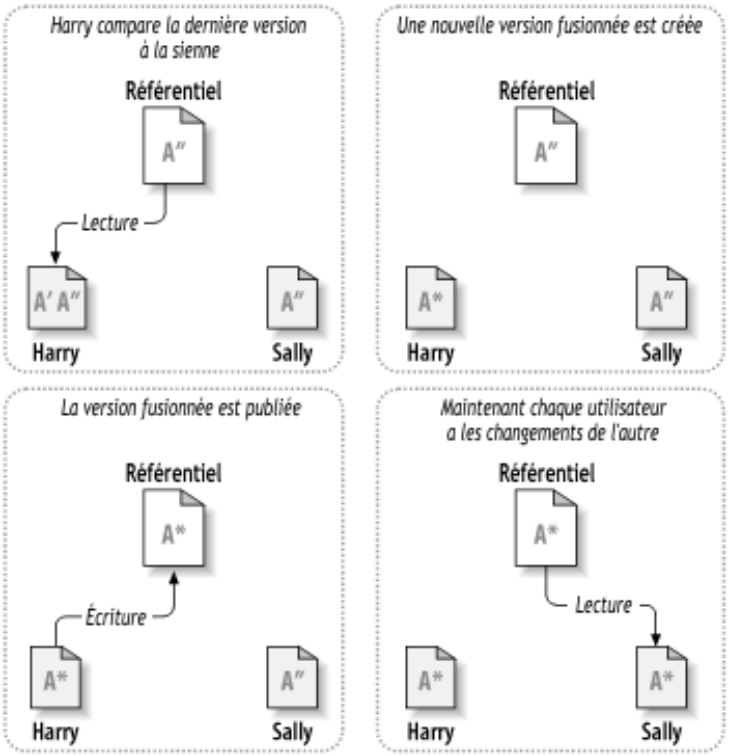


Figure 5: svn

quelques actions principales, donc :

- checkout, clone : récupère le projet depuis le serveur
- update, pull : mise à jour locale des modification réalisée sur le serveur
- commit : valide ses modifications locales
- status : état de mes modifications locales?
- diff : différences d'état entre le local et le serveur
- et un principe général : effectuer le plus de modifications locales sur la structure du code (renommage de fichiers, de répertoires, déplacement, supression, etc.) à l'aide de l'outil de versionnement.

Git

Figure 6: git

Concepts de base

- Décentralisé
- Rapide
- Flexible

Note: Inspiré par BitKeeper et créé en 2005 par Linux Torvalds (et Junio Hamano) pour le remplacer après que la gratuité de ce dernier soit révoqué. Linus Torvalds est le créateur de Linux...

A propos du nom "je ne suis q'un égocentrique, donc je n'appelle mes créations que du nom de ma propre personne". Git est assez péjoratif en anglais...

Décentralisé

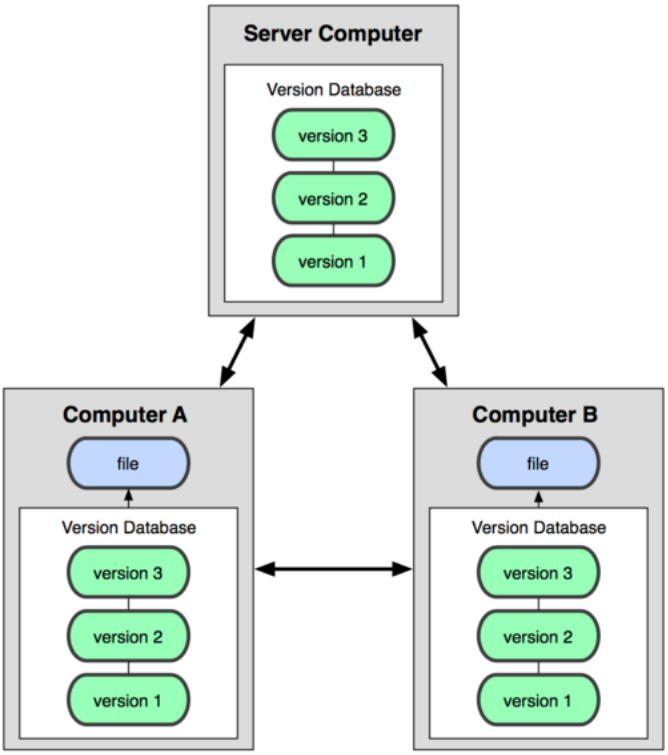


Figure 7: schema decentralisé

Avantages

- Chaque `clone` est un `fork`
- Chaque `clone` est un backup
- Possibilité de versionner hors ligne
- **Très** rapide

Inconvénients

- Pas très performant pour les gros fichiers binaires

Note: Chaque clone est un backup : a condition de maintenir a jour toutes les branches.

Objectif de git

- Rapidité
- Simplicité
- Prise en charge de milliers de branches parallèles
- Architecture distribuée
- Capable de gerer efficacement de gros projet *(ex:Le noyau Linux)*
- Garantir l'intégrité

Les trois états

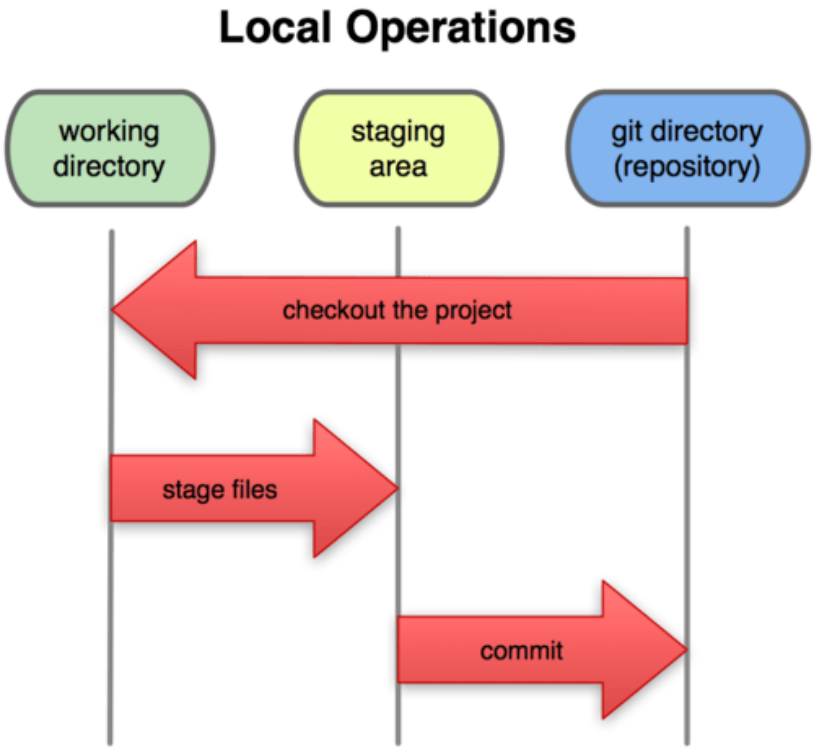


Figure 8: schema 3 states

Utilisation standard

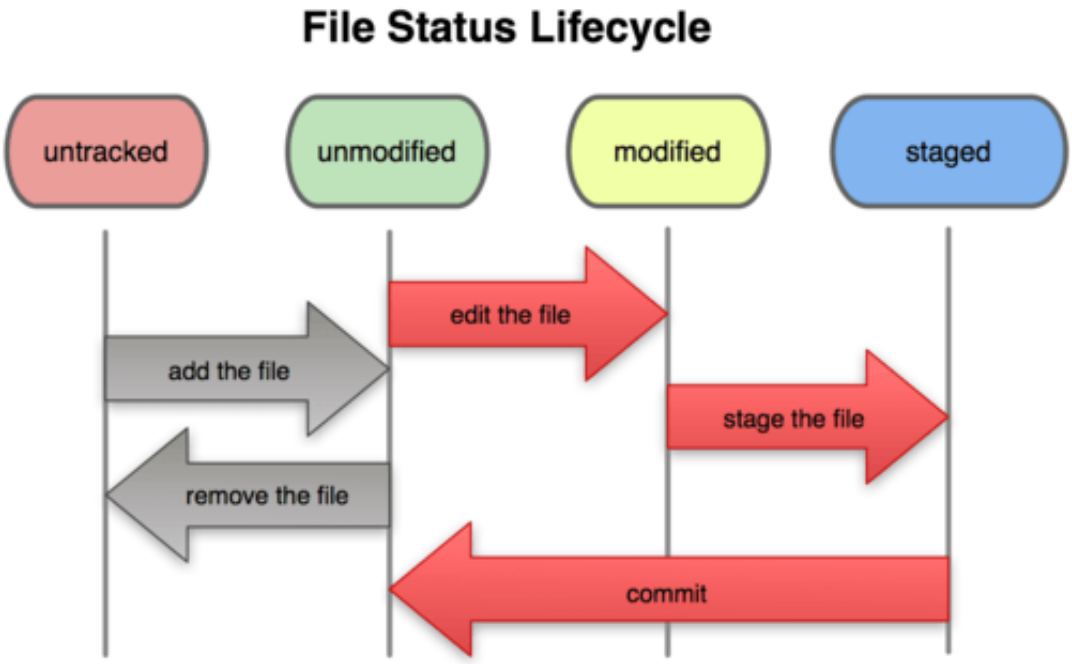


Figure 9: schema file lifecycle

Installer git

Figure 10: one does not simply install git on windows

Officiel

http://git-scm.com/downloads

Note: Binaire, installeur et source pour Windows, OS X, Linux, Solaris

Windows

http://msysgit.github.io/

LINUX

- Ubuntu: `sudo apt-get install git`

OS X

Avec Homebrew `brew install git`

Note: Ok mais a quoi ça ressemble ?

Interface graphique ?

http://git-scm.com/downloads/guis

Recommandé :

- SmartGit (crossplatform, gratuit pour usage personnel)
- SourceTree (sous mac, le plus populaire)
- GitHub for Mac / Windows (attention : spécifique à GitHub)

Configuration

http://git-scm.com/book/fr/Personnalisation-de-Git-Configuration-de-Git

Note: Et voilà, `git` est pret !

Commandes de base

L'aide dans git

```
$ git --help
$ git add --help
$ git <sub-command> --help
```

Créer un dépôt local

```
$ git init
```

Cloner un dépôt existant

```
$ git clone https://github.com/p-j/isep-git.gi
```

Note: Par défaut, cela créera un dossier `isep-git` à l'endroit où vous avez executé la commande. Ce dossier contiendra votre copie du dépôt.

Récupérer le dépôt complet

```
$ git clone https://github.com/p-j/isep-git.gi
$ git fetch --all
```

Note: Permet de récupérer toutes les branches et pas uniquement la branche courrante (ie: `master` par défaut)

Créer un instantané

Changements spécifiques:

```
$ git add *.html
$ git add README.md
$ git commit -m 'First commit'
```

Tous les changements:

```
$ git commit -am 'First commit'
```

Connaitre le status du dépôt

```
$ git status
```

Enlever un fichier

De l'index

```
$ git rm --cached file.html
```

De l'index et du disque dur

```
$ git rm file.html
```

Déplacer des fichiers

Bien que git ne se préocuppe que du contenu, il y a une commade pour déplacer des fichiers.

```
$ git mv file1 file2
```

Qui est l'équivalent de

```
$ mv file1 file2
$ git rm file1
$ git add file2
```

Voir l'historique du dépôt

Complet

```
$ git log
```

Filtrer par date

```
$ git log --since=2.weeks
$ git log --since="2 years 1 day 3 minutes ag
```

Format court avec graphique

```
$ git log --pretty=oneline --decorate --graph
```

Voir les différences

Les fichiers modifiés

```
$ git diff
```

Les fichiers indexés

```
$ git  diff --cached
```

Par rapport a une version spécifique

```
$ git diff 7be56a
git diff HEAD^
```

Voir les commits

Le dernier commit

```
$ git show
```

Un commit spécifique

```
$ git show 7be56a
git show HEAD^
```

Revenir en arrière

Modifier le dernier commit

```
$ git commit --amend
```

Désindexer un fichier

```
$ git reset HEAD file.html
```

Annuler les modifications d'un fichier modifier

```
$ git checkout -- file.html
```

Créer des tags

Tags léger

```
$ git tag v0.1.0
```

Tags annotés

```
$ git tag -a v0.1.0 -m 'Version 0.1.0'
```

Note: Un tag est simplement un commit avec un nom un peu plus sympa qu'un hash. On s'en sert pour marquer des versions stable ou des points important dans le développement qui serviront de

.gitignore

```
$ cat .gitignore
.DS_Store
.svn
log/*.log
tmp/**
node_modules/
```

- Les lignes vides ou démarrant par un # sont ignorées
- Il est possible d'utiliser des jokers standards
- Terminer un modèle par un slash / pour spécifier un dossier
- Inverser un modèle avec un point d'exclamation !

Remotes

- Autre clones du même dépôt
- Peut être local (un autre `checkout`) ou distant (collègue, serveur central)
- On peut configurer une valeur par défaut pour `push` et `pull`

```
$ git remote -v
origin  git@github.com:p-j/isep-fsnp.git (fetc
origin  git@github.com:p-j/isep-fsnp.git (push
```

Push to remote

Sans valeur par défaut

```
$ git push <remote> <rbranch>
```

Configurer une valeur par défaut

```
$ git push -u <remote> <rbranch>
```

Puis

```
$ git push
```

Pull from remote

FETCH & MERGE

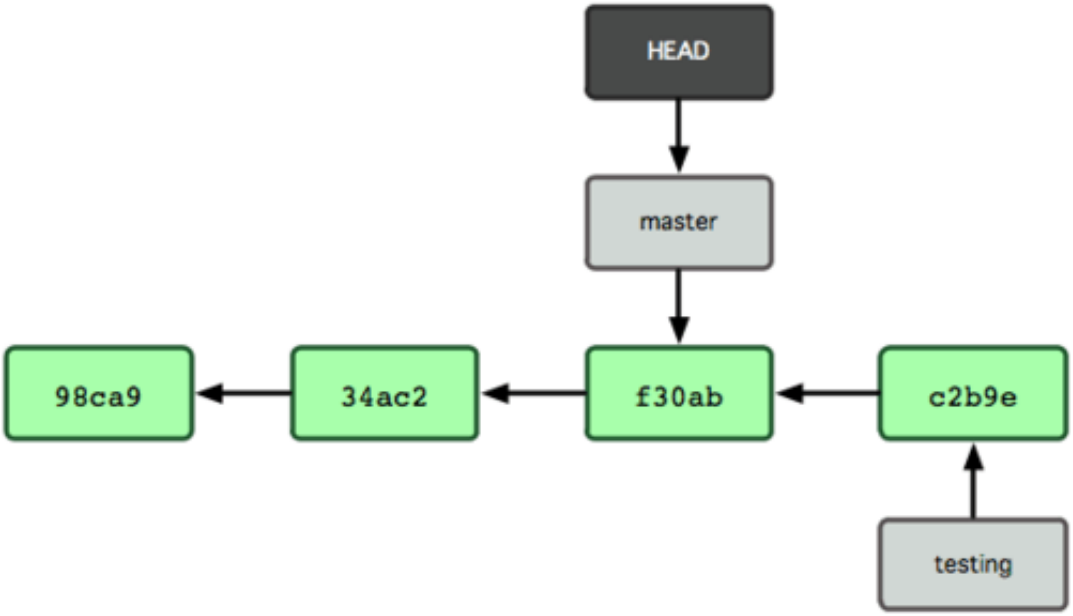
```
$ git pull  [<remote> <rbranch>]
```

Equivalent de

```
$ git fetch <remote>
$ git merge <rbranch>
```

Branches

Les branches sont des “pointeurs” vers des commits.



Fusionner des branches

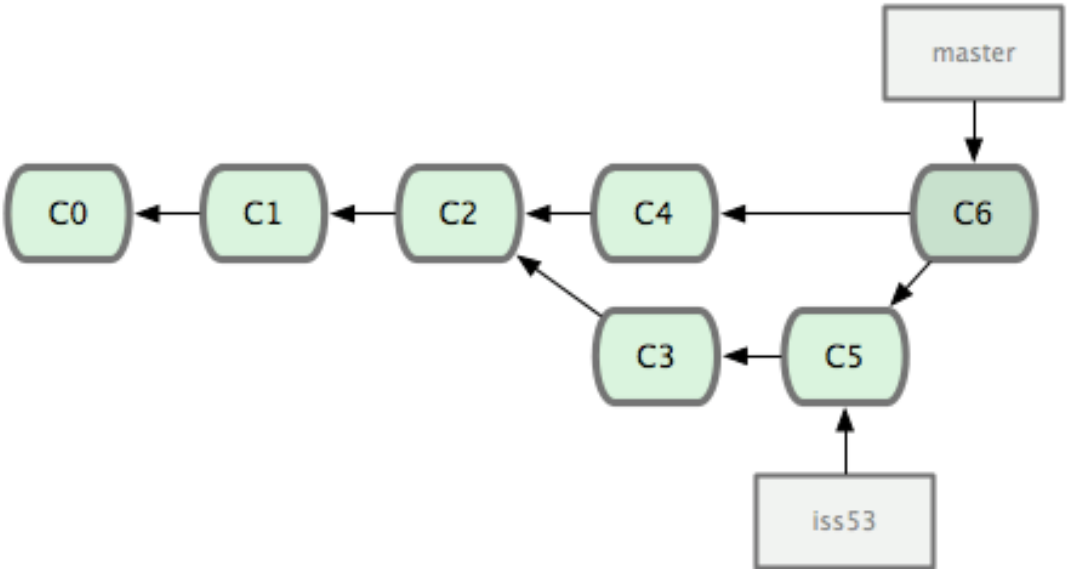


Figure 11: schema common workflow

En général ça marche sans problème

Et maintenant ?

Figure 12: git all the things

GitHub

J'allais oublier...

- Github : plateforme web qui sert de serveur et d'interface à Git
- Non recommandé par Linus Torvalds car limite fortement les possibilités de Git
- Néanmoins adopté par de nombreux projets, et notamment des projets R (Hadley Wickham a l'air d'être un grand fan...)
- cf. le package devtools et ses fonctions dédiées (`install_github()` etc.)

A quoi cela ressemble?

https://github.com/manumart/formation_R

Resources

Très (très) inspirée de `https://github.com/p-j/isep-git/blob/master/data/slides.md`

- "Pro Git" Book
- Git reference
- Github git challenges
- Cheat sheet
- Git Visual Cheat sheet

Questions ?

Compléments

Le dossier git

Le répertoire de travail est une extraction unique d'une version du projet. Ces fichiers sont extraits depuis la base de données compressée dans le répertoire Git et placés sur le disque pour pouvoir être utilisés ou modifiés.

La zone d'index est un simple fichier, généralement situé dans le répertoire Git, qui stocke les informations concernant ce qui fera partie du prochain instantané.

Note: Le répertoire Git est l'endroit où Git stocke les méta-données et la base de données des objets de votre projet. C'est la partie la plus importante de Git, et c'est ce qui est copié lorsque vous clonez un dépôt depuis un autre ordinateur.

Master comme branche stable

- On ne commit dans master que du code stable
- On commit le code instable dans une branche de développement
- On déploie des patchs sur master
- On merge les patchs dans la branche de développement

Master comme branche de développement

- On commit le code instable dans master
- On créer des tags pour marquer les étapes stables
- On créer une branche à partir d'un tag pour les patchs
- On merge les patchs dans master

Gestion des branches

Créer une branche

```
$ git branch iss53
```

```
$ git checkout -b iss53 master
```

Changer de branche

```
$ git checkout iss53
```

Supprimer une branche

Voire toutes les branches

```
$ git branch
  iss53
* master
  testing
```

Voire le dernier commit sur chaque branches

```
$ git branch -v
  iss53 93b412c fix javascript issue
* master 7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list
```

Lister les branches fusionnées

```
$ git branch --merged
  iss53
* master
```

Lister les branches non fusionnées

```
$ git branch --no-merged
  testing
```

Stashing

```
$ git stash
$ git stash pop
$ git stash list
$ git stash apply
$ git stash drop
$ git stash clear
```

Note: Utilisez git stash quand vous voulez enregistrer l'état actuel du répertoire de travail et de l'index, et revenir à un répertoire de travail propre. La commande enregistre vos modifications locales à part et mets à jour le répertoire de travail pour correspondre à la HEAD.

Les modifications mise de côté par cette commande peuvent être listés avec `git stash list`, inspecté avec `git stash show`, et restauré (potentiellement par dessus un autre commit) avec `git stash apply`. `git stash` sans aucun argument équivaut à `git stash save`. Une `stash` est par défaut répertorié comme «WIP sur branchname ...», mais vous pouvez donner un message plus explicite via la ligne de commande lorsque vous en créez un.

Identité

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@examp
```

Couleur

```
$ git config --global color.ui true
```