

# Contrôle de versions et Git

Millian Poquet

2020-12-07

# Situations récurrentes

## Problèmes

- Ce rapport était bien mieux hier.
- Mon PC a planté...
- Ce n'est pas le bon fichier de configuration !
- Mais qui a écrit ça ? pourquoi ?
- Comment appliquer ce patch de sécurité sur ces versions ?
- Comment intégrer ces deux changements en même temps ?

## Solution ?

- Sauvegardes régulières : `rapport.docx`

# Situations récurrentes

## Problèmes

- Ce rapport était bien mieux hier.
- Mon PC a planté...
- Ce n'est pas le bon fichier de configuration !
- Mais qui a écrit ça ? pourquoi ?
- Comment appliquer ce patch de sécurité sur ces versions ?
- Comment intégrer ces deux changements en même temps ?

## Solution ?

- Sauvegardes régulières : rapport2.docx

# Situations récurrentes

## Problèmes

- Ce rapport était bien mieux hier.
- Mon PC a planté...
- Ce n'est pas le bon fichier de configuration !
- Mais qui a écrit ça ? pourquoi ?
- Comment appliquer ce patch de sécurité sur ces versions ?
- Comment intégrer ces deux changements en même temps ?

## Solution ?

- Sauvegardes régulières : `rapport_final.docx`

# Situations récurrentes

## Problèmes

- Ce rapport était bien mieux hier.
- Mon PC a planté...
- Ce n'est pas le bon fichier de configuration !
- Mais qui a écrit ça ? pourquoi ?
- Comment appliquer ce patch de sécurité sur ces versions ?
- Comment intégrer ces deux changements en même temps ?

## Solution ?

- Sauvegardes régulières : rapport\_FINAL.docx

# Situations récurrentes

## Problèmes

- Ce rapport était bien mieux hier.
- Mon PC a planté...
- Ce n'est pas le bon fichier de configuration !
- Mais qui a écrit ça ? pourquoi ?
- Comment appliquer ce patch de sécurité sur ces versions ?
- Comment intégrer ces deux changements en même temps ?

## Solution ?

- Sauvegardes régulières : rapport\_FINAL2.docx

# Situations récurrentes

## Problèmes

- Ce rapport était bien mieux hier.
- Mon PC a planté...
- Ce n'est pas le bon fichier de configuration !
- Mais qui a écrit ça ? pourquoi ?
- Comment appliquer ce patch de sécurité sur ces versions ?
- Comment intégrer ces deux changements en même temps ?

## Solution ?

- Sauvegardes régulières : rapport\_FINAL2.docx
- Partage par clé usb ? mail ? transferts réseau ?

# Situations récurrentes

## Problèmes

- Ce rapport était bien mieux hier.
- Mon PC a planté...
- Ce n'est pas le bon fichier de configuration !
- Mais qui a écrit ça ? pourquoi ?
- Comment appliquer ce patch de sécurité sur ces versions ?
- Comment intégrer ces deux changements en même temps ?

## Solution ?

- Sauvegardes régulières : rapport\_FINAL2.docx
- Partage par clé usb ? mail ? transferts réseau ?

## Solution

Utiliser un gestionnaire de versions !



# Aspects traités par ce cours

- Gestion de version : concepts clés
- Pourquoi utilise-t-on du décentralisé aujourd'hui ?
- Git : kit de survie et utilisation plus avancée
- Bonnes pratiques (qualité, traçabilité, intégration)

# Définitions

## Contrôle de version

*version control, revision control, source code management (scm)*

- Gérer l'ensemble des versions de un ou plusieurs fichiers
- Devenu important avec l'informatisation
- Marche sur tout type de fichier, marche **bien** sur fichier textuel
- Surtout utilisé sur du code source

## Gestionnaire de version

*version control system (vcs)*

- Un logiciel permettant le contrôle de version
- Dédié (e.g., Git) ou non (e.g., dans Wikipedia ou Google Docs)

# Un vieux problème...

Le besoin d'identifier et contrôler la version d'œuvres est quasiment aussi vieux que l'écriture. Il croît avec l'amélioration des techniques d'imprimerie.

- Écriture (Mésopotamie), -3000
- Imprimerie xylographique (Chine), -200
- Imprimerie par caractères mobiles (Chine), 1040
- Imprimerie de Gutenberg (Europe), 1454
- Institut International de Bibliographie, 1895

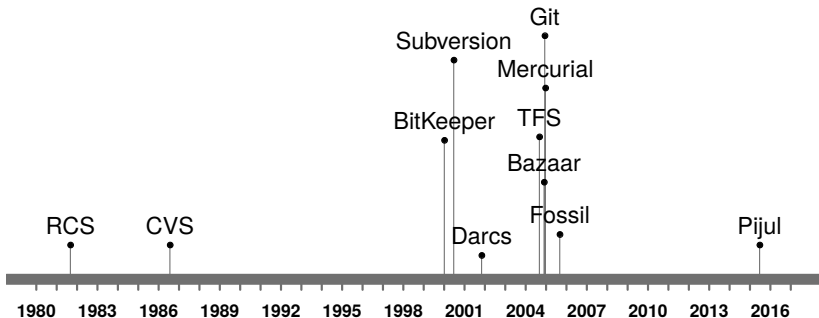
Standards bibliographiques actuels.

- ISBN, 1970 — publications non périodiques
- ISSN, 1971 — publications périodiques
- PubMed, 1996 — recherche en médecine et biologie
- DOI, 2000 — tout objet numérique

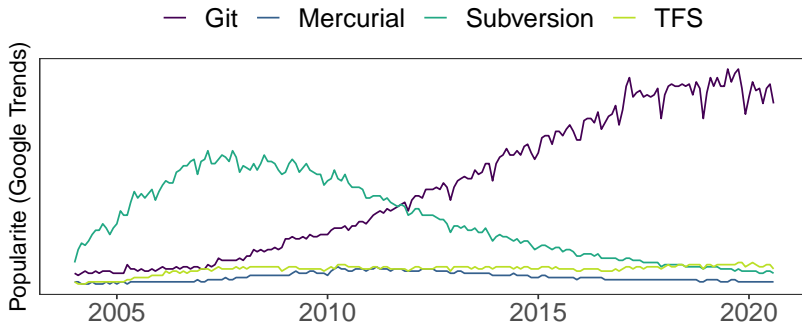
# Un vieux problème, même en informatique

Principaux gestionnaires de versions.

- Locaux : SCCS (1972), RCS
- Centralisés : Subversion, CVS, TFS
- Décentralisés : Git, Mercurial, BitKeeper...



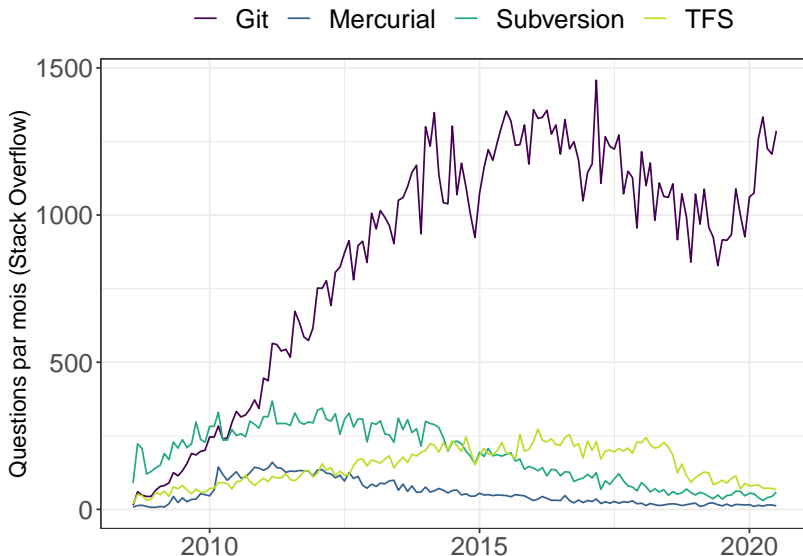
# Popularité écrasante de Git



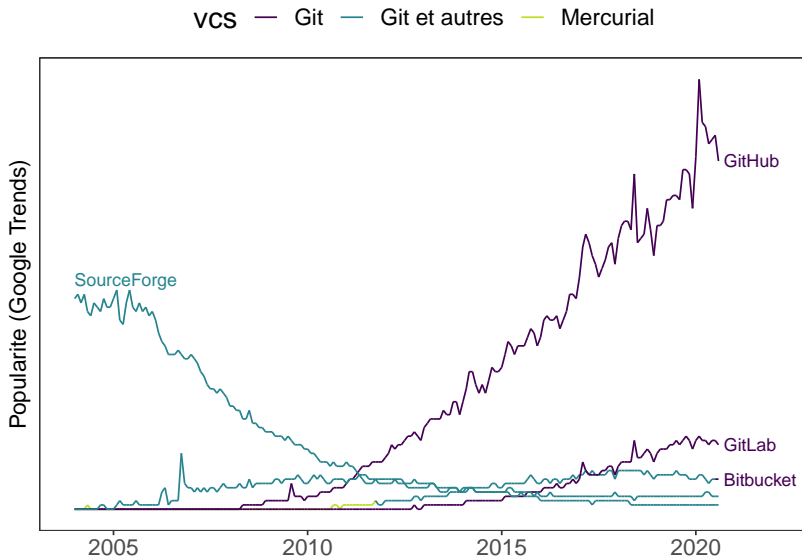
## Une référence pour des concurrents

- Pijul for Git users (Pijul doc, page 1, section 1)
- Fossil vs. Git (Fossil doc, menu principal du site)
- Azure DevOps a deux VCS: son moteur *ad hoc* et Git

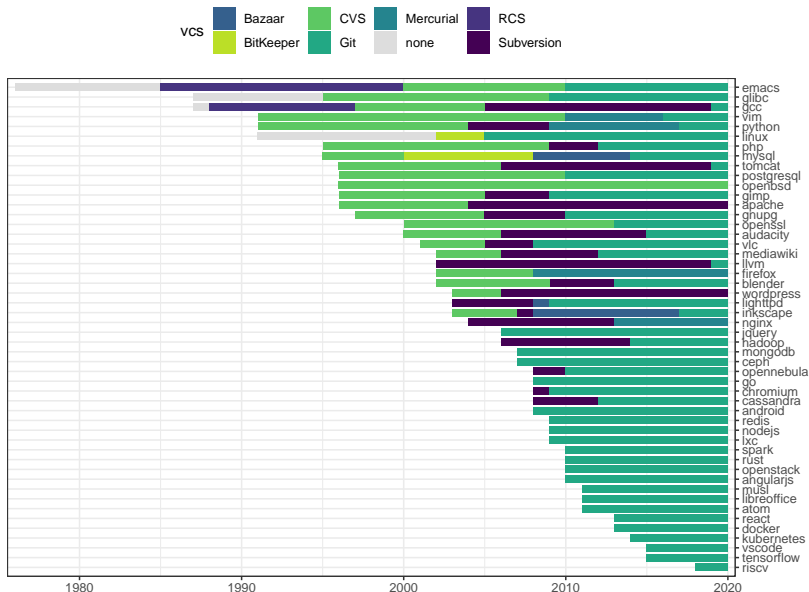
# Popularité écrasante de Git (Stack Overflow)



# Popularité des forges

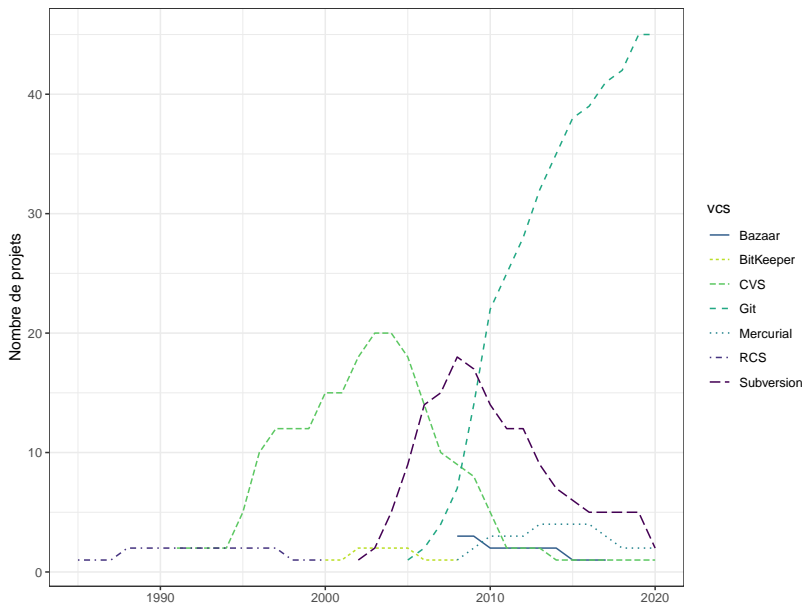


# Adoption dans les projets open source





# Adoption dans les projets open source (2)

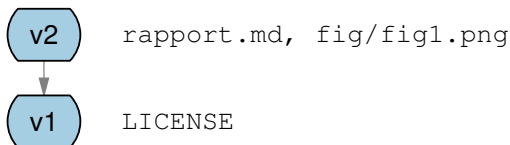


# Historique des versions

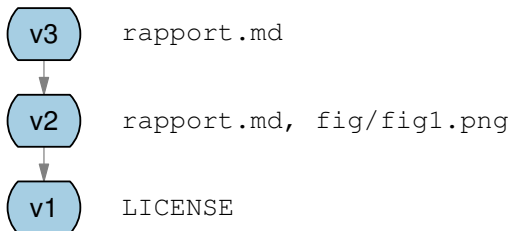


LICENSE

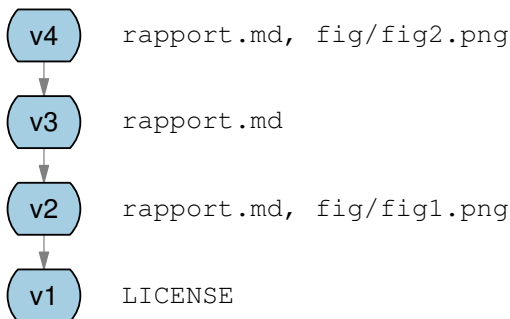
# Historique des versions



# Historique des versions



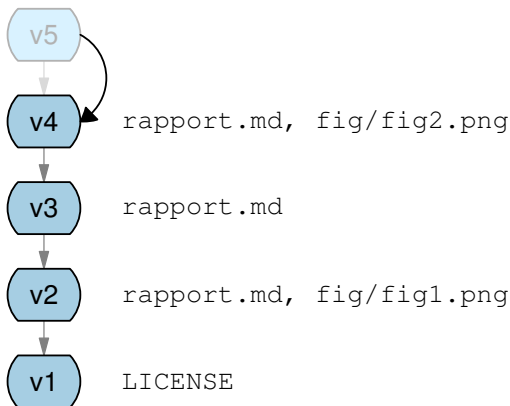
# Historique des versions



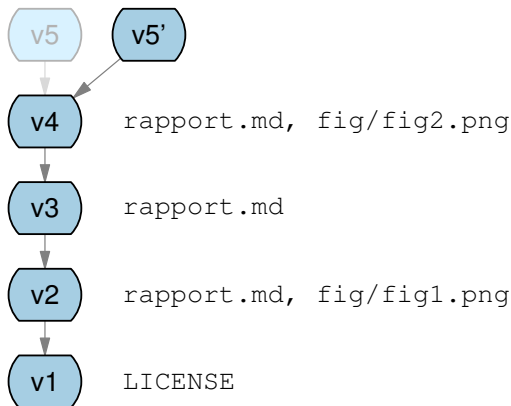
# Historique des versions



# Historique des versions

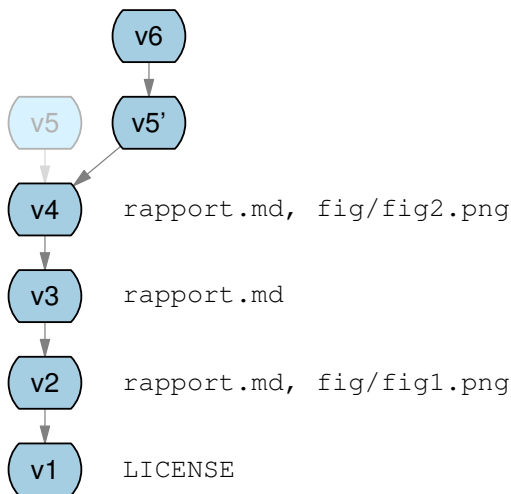


# Historique des versions

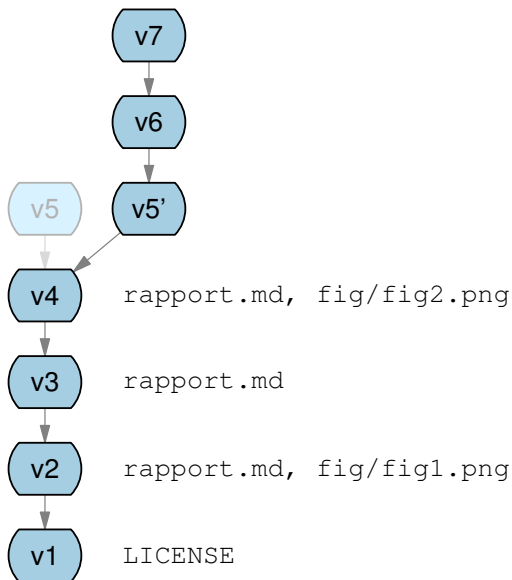




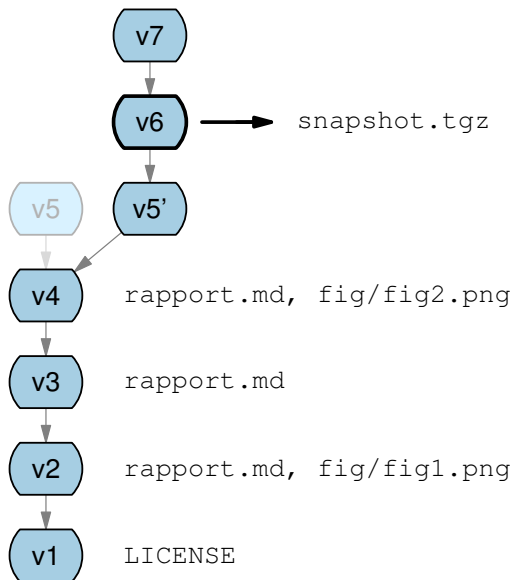
# Historique des versions



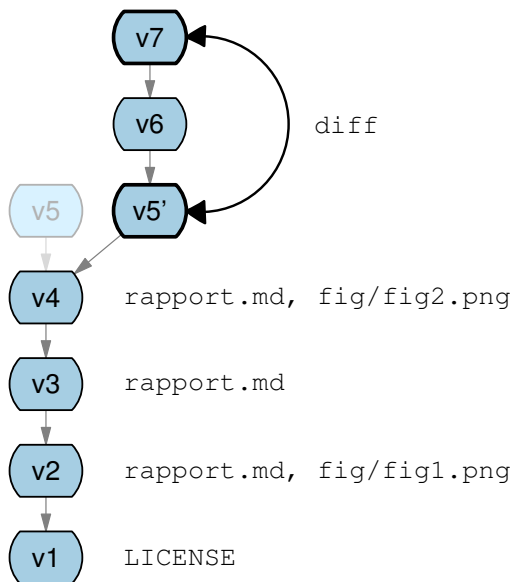
# Historique des versions



# Historique des versions

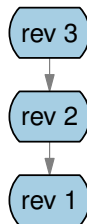


# Historique des versions



# Contrôle de versions centralisé — e.g., Subversion

Le serveur gère l'historique



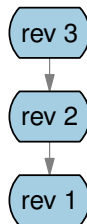
Serveur SVN

# Contrôle de versions centralisé — e.g., Subversion

Alice veut accéder au dépôt



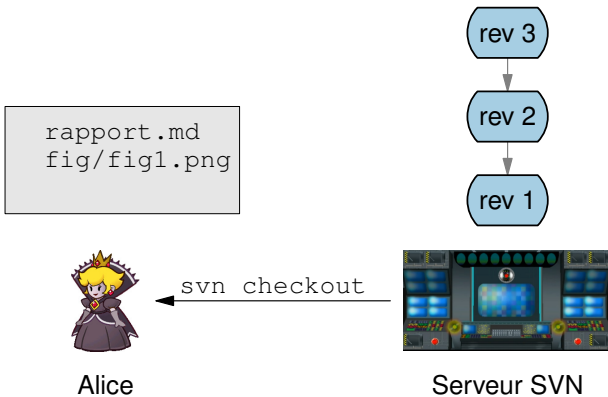
Alice



Serveur SVN

# Contrôle de versions centralisé — e.g., Subversion

Alice fait une copie locale



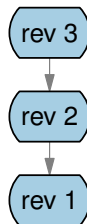
# Contrôle de versions centralisé — e.g., Subversion

Alice modifie/ajoute des fichiers

```
* rapport.md  
  fig/fig1.png  
* fig/fig2.png
```



Alice

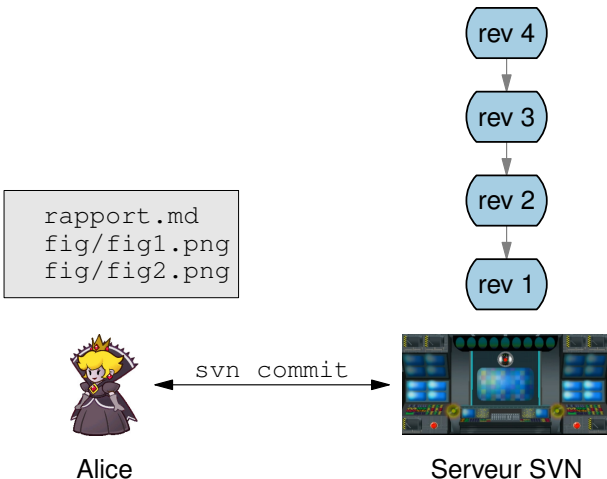


Serveur SVN



# Contrôle de versions centralisé — e.g., Subversion

Alice crée une nouvelle version



# Contrôle de versions centralisé — e.g., Subversion

Alice modifie un fichier

```
* rapport.md  
  fig/fig1.png  
  fig/fig2.png
```



Alice



Serveur SVN

# Contrôle de versions centralisé — e.g., Subversion

Alice crée une nouvelle version

```
rapport.md  
fig/fig1.png  
fig/fig2.png
```



Alice

svn commit



Serveur SVN

rev 5

rev 4

rev 3

rev 2

rev 1

# Contrôle de versions centralisé — e.g., Subversion

## Fonctionnement résumé

- Un serveur **central** gère les versions
- Les utilisateurs sont des clients qui interagissent avec le serveur
  - Pour récupérer une version
  - Pour soumettre une nouvelle version
  - Pour faire diverses requêtes (différences entre versions. . .)
- Par conséquent, l'historique est très linéaire

# Contrôle de versions centralisé — e.g., Subversion

## Fonctionnement résumé

- Un serveur **central** gère les versions
- Les utilisateurs sont des clients qui interagissent avec le serveur
  - Pour récupérer une version
  - Pour soumettre une nouvelle version
  - Pour faire diverses requêtes (différences entre versions...)
- Par conséquent, l'historique est très linéaire

## Problèmes

- Impossible de créer des versions locales  
(client sans accès internet, serveur en panne...)
- Conflits fréquents, intégration délicate  
(intégration incluse dans *vraies* modifs d'une feature/bugfix)
- Difficile de travailler en parallèle  
(plusieurs développeurs sur les mêmes fichiers,  
un développeur qui travaille sur plusieurs features...)

# Contrôle de versions décentralisé — e.g., Git

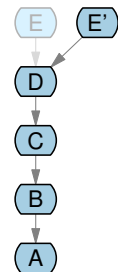
A

Alice crée un dépôt local



Alice

# Contrôle de versions décentralisé — e.g., Git

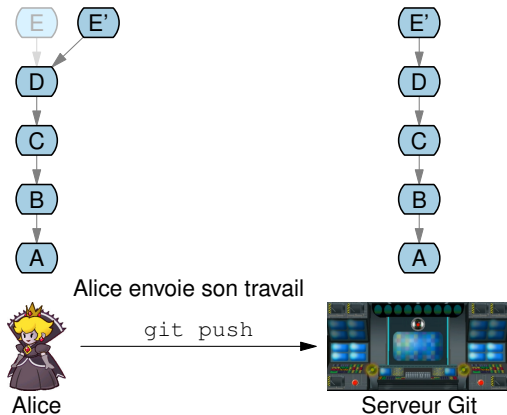


Alice travaille en local



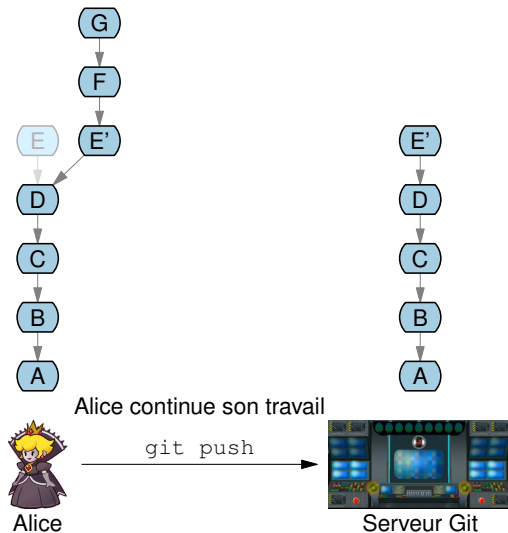
Alice

# Contrôle de versions décentralisé — e.g., Git

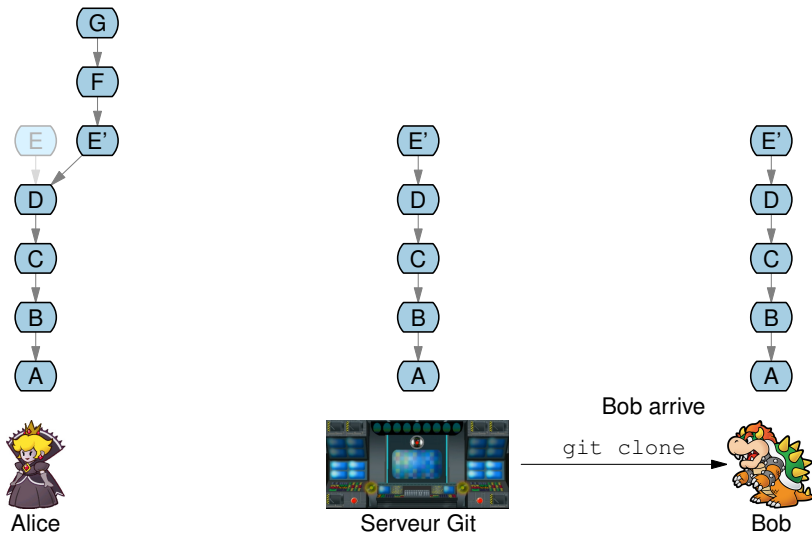




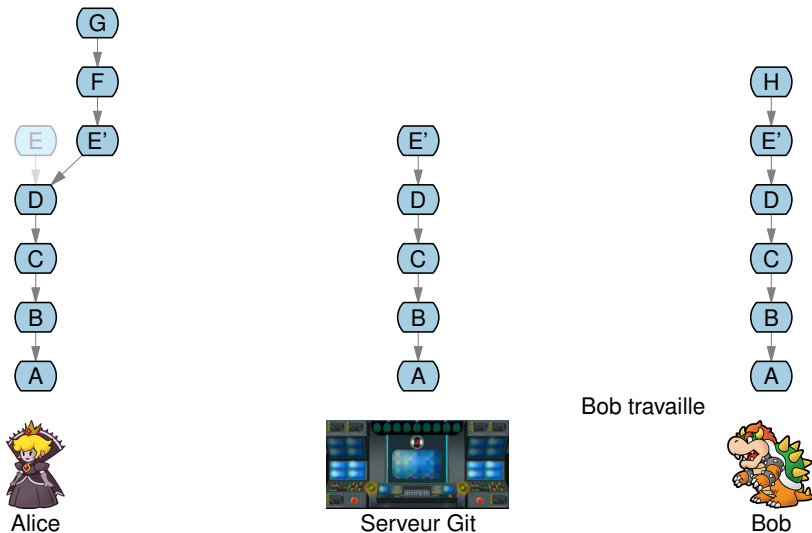
# Contrôle de versions décentralisé — e.g., Git



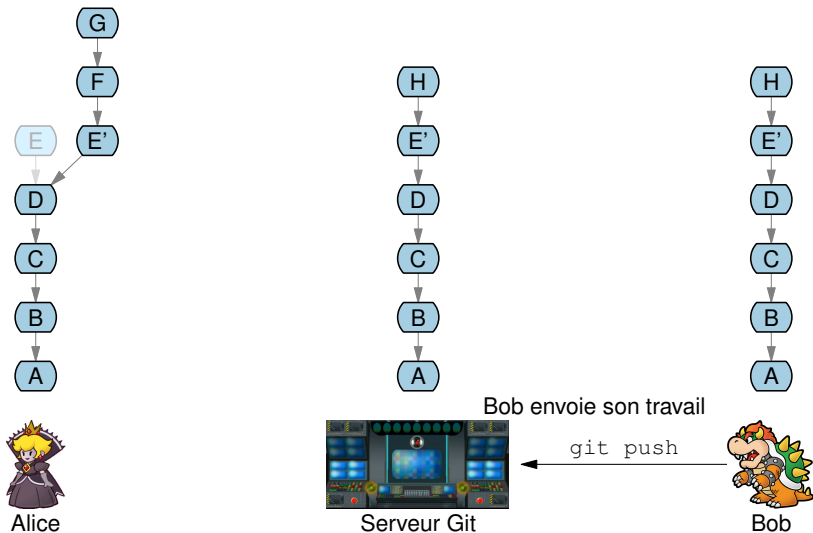
# Contrôle de versions décentralisé — e.g., Git



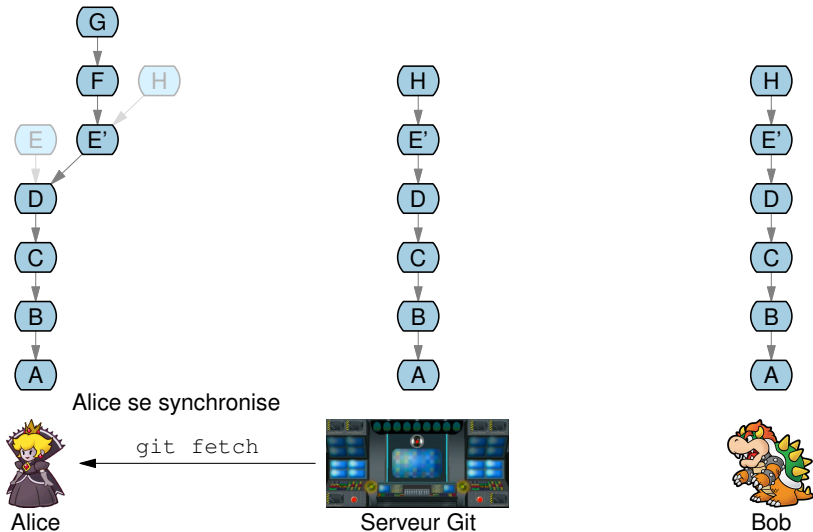
# Contrôle de versions décentralisé — e.g., Git



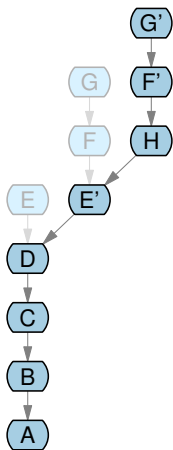
# Contrôle de versions décentralisé — e.g., Git



# Contrôle de versions décentralisé — e.g., Git



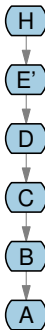
# Contrôle de versions décentralisé — e.g., Git



Alice gère l'intégration



Alice

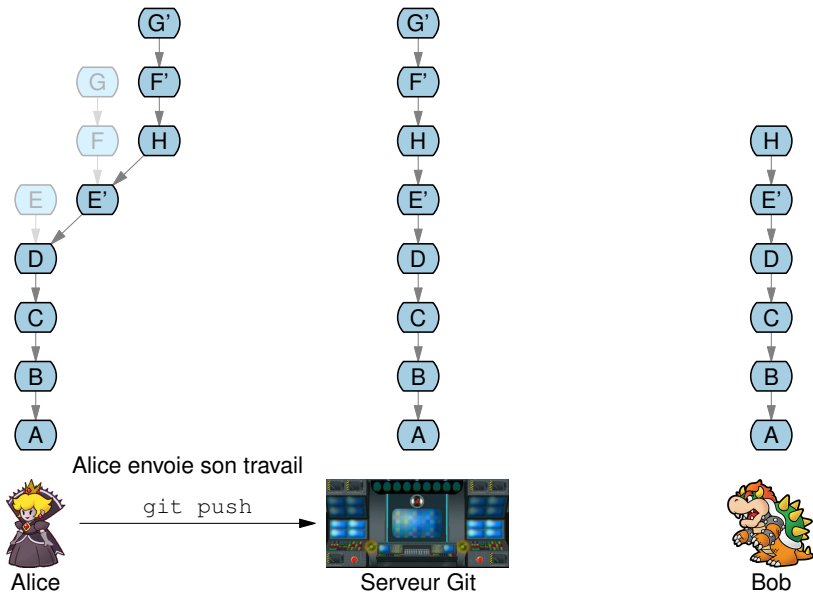


Serveur Git



Bob

# Contrôle de versions décentralisé — e.g., Git



# Contrôle de versions décentralisé — e.g., Git

## Fonctionnement résumé

- Chaque utilisateur a un historique local
- Se synchroniser = échanger des bouts d'historique
- Intégrer = gérer un historique distribué
- Le serveur est inutile<sup>1</sup>, on peut s'échanger l'historique sans
- L'historique est assez arborescent

## Problèmes ?

- Demande une certaine aise en manipulation d'arbres
- Bénéfices du contrôle local et centralisé, sans les inconvénients

---

1. Un dépôt central est très souvent utilisé pour des raisons pratiques : accès persistant, unicité de la "dernière version à jour", documentation, bug tracker...



### 3 Git (1)

- Concepts clés
- Premiers pas, commandes Git...

# Commits

## Définition

- C'est un état (*snapshot*) de tout le dépôt Git
- C'est l'unité de base de travail en Git
- Tout travail *commité* peut être récupéré<sup>2</sup> plus tard
- Identifié par un *hash* unique au dépôt — e.g., e9ed0f

## Contenu

- Contenu des fichiers suivis par Git (et leurs métadonnées)
- Un message de commit. Primordial pour traçabilité (humains)
- Référence vers les commits parents — 1, 2, 0, 66<sup>3</sup>...

---

2. Sauf appel explicite du GC ou suppressions manuelles de fichiers dans `.git`

3. <https://www.destroyallsoftware.com/blog/2017/the-biggest-and-weirdest-commits-in-linux-kernel-git-history>

# Branches, tags et HEAD

## Branch

- C'est un pointeur **mobile** vers un commit
- Lorsqu'on *commite* à partir d'une branche, la branche est mise à jour (elle pointe vers le nouveau commit)
- Branche par défaut : `master`

## Tag

- C'est un pointeur **constant** vers un commit
- Peut avoir un message associé (*annotated tag*)
- Utile pour marquer un commit particulier — e.g., `v1.0.0`

## HEAD

- Un pointeur vers le commit *courant*
- Pointe souvent vers une branche
- Peut pointer directement vers un commit (*detached HEAD*)

# Remotes

## Définition

- Dépôt distant avec lequel on peut se synchroniser
- Contient des branches distantes (*remote branches*)
- La synchronisation se fait le plus souvent via branches distantes

# Illustration des concepts Git

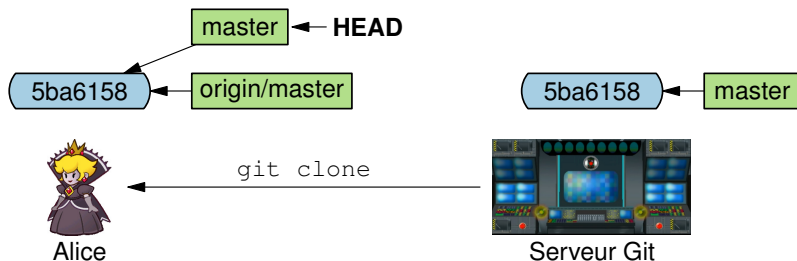
Alice crée un dépôt sur GitHub, avec un README



Serveur Git

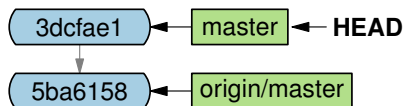
# Illustration des concepts Git

Alice fait une copie locale du dépôt



# Illustration des concepts Git

Alice travaille



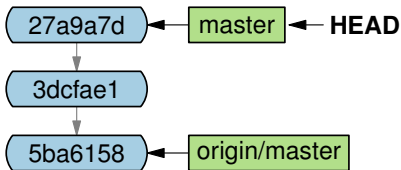
Alice



Serveur Git

# Illustration des concepts Git

Alice travaille (encore)



Alice

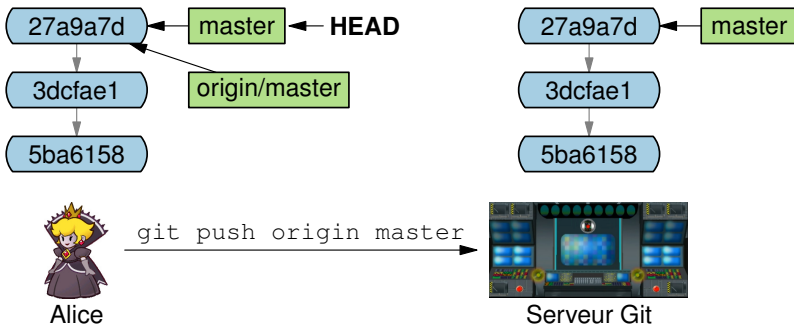


Serveur Git



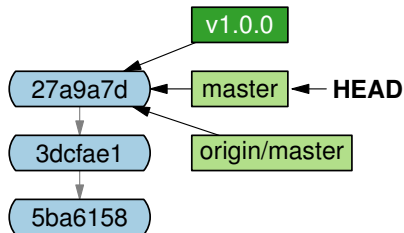
# Illustration des concepts Git

Alice partage sa branche

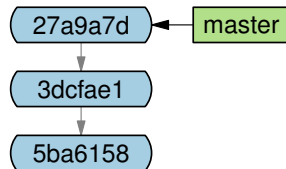


# Illustration des concepts Git

Alice crée un tag



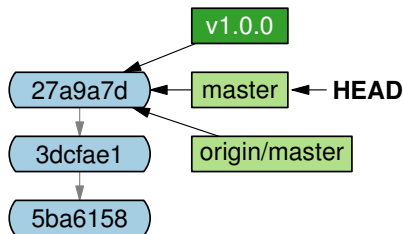
Alice



Serveur Git

# Illustration des concepts Git

Alice partage son tag



Alice

`git push origin v1.0.0`



Serveur Git

# Les trois arbres Git

Les commandes git peuvent être déroutantes à première vue.

Leur organisation est logique quand on sait que Git manipule trois structures de données en interne.

- Le répertoire de travail — *sandbox* de travail
- L'*index* ou *staging area* — zone tampon pour créer un commit
- L'arbre des commits — contient l'historique

### 3 Git (1)

- Concepts clés
- Premiers pas, commandes Git...

# Fichiers internes à Git, configuration

Tous les fichiers internes de Git sont stockés dans le répertoire `.git` (à la racine du dépôt).

Git a beaucoup d'options de configuration.

Elles sont stockées dans des fichiers, dont voici les principaux.

- `${HOME}/.gitconfig` (globale)
- `.gitconfig` à la racine d'un dépôt (spécifique au dépôt)

Éditables directement (`vim ~/.gitconfig`) ou via des commandes Git (`git config --global alias.chk checkout`).

# Initialisation d'un dépôt Git

## Création d'un dépôt local vide

- `git init [<repo>]`

## Récupération d'un dépôt existant

- `git clone <repo-url> [<dir>]`

Options utiles si on ne veut que la dernière version.

- `--single-branch` : synchronise une seule branche
- `--branch <branch>` : spécifie la branche par défaut
- `--depth 1` : 1 seul commit au lieu de tout l'historique

## Exemples

- `git init ${HOME}/projects/my-awesome-project`
- `git clone https://github.com/python/cpython.git`

# Je suis perdu, que faire ?

## git status

- Montre votre état courant (branche, synchronisation distante...)
- Montre votre *staging area*
- Montre comment sortir de situations épineuses (rebase...)

On branch master

Your branch is up to date with gitlab/master.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: slides/version-control.md

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: fig/branch.asy



# Création de commits

## Enregistrement des modifications à commiter

- `git add <path>...`
- `git rm <path>...`

Options de `git add`.

- `-f, --force` : force l'ajout (cf. `.gitignore`)
- `-p, --patch` : sélection interactive des blocs à ajouter.

**Utile** pour se relire avant de commit une bêtise.

## Création du commit

- `git commit [-m <message>]`

Options.

- `-a, --all` : ajoute toute modification des fichiers suivis par Git.  
**À éviter** car favorise le commit de blocs indésirables.

# Création de commits : exemple

**Historique**

**Index**

**Répertoire de travail**

|  
|  
|  
|  
|  
|  
|  
|

|  
|  
|  
|  
|  
|  
|  
|

```
$ git init
```

Création d'un dépôt local

# Création de commits : exemple

## Historique

|  
|  
|  
|  
|  
|  
|  
|

## Index

## Répertoire de travail

|  
|  
|  
|  
|  
|  
|  
|

LICENSE (untracked)  
README.md (untracked)

```
$ vim LICENSE  
$ vim README.md
```

Création de fichiers initiaux

# Création de commits : exemple

## Historique

## Index

## Répertoire de travail

LICENSE

new file mode 100644

@@ -0,0 +1 @@

+Public domain.

LICENSE (staged)

README.md (staged)

README.md

new file mode 100644

@@ -0,0 +1 @@

+TODO: README

```
$ git add LICENSE README.md
```

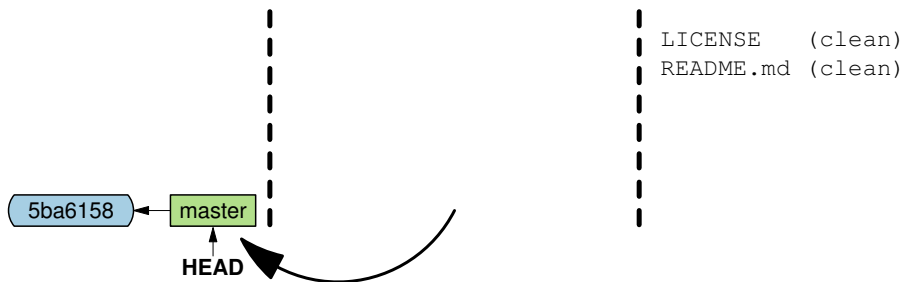
Ajout des fichiers dans l'index

# Création de commits : exemple

## Historique

## Index

## Répertoire de travail



```
$ git commit -m 'initial commit'
```

Création du commit

# Création de commits : exemple

## Historique

## Index

## Répertoire de travail



```
$ vim hello.py
$ vim README.md
```

Travail sur `hello.py` et `README.md`

# Création de commits : exemple

## Historique

## Index

## Répertoire de travail

```
hello.py
new file mode 100644
@@ -0,0 +1,2 @@
+#!/usr/bin/python3
+print('Hello world!')
```

```
README.md
@@ -1 +1 @@
-TODO: README
+Hello world codes.
```

```
LICENSE      (clean)
README.md    (staged)
hello.py     (staged)
```

5ba6158

master

HEAD

```
$ git add hello.py README.md
```

Ajout des fichiers dans l'index

# Création de commits : exemple

## Historique

## Index

## Répertoire de travail

```
hello.py
new file mode 100644
@@ -0,0 +1,2 @@
+#!/usr/bin/python3
+print('Hello world!')
```

```
LICENSE      (clean)
README.md    (modified)
hello.py     (staged)
```

5ba6158

master

HEAD

```
$ git restore --staged README.md
```

Oups : retrait de modifications non souhaitées



# Création de commits : exemple

## Historique

## Index

## Répertoire de travail



```
$ git commit -m 'python example'
```

Création du commit

# Création de commits : exemple

## Historique

## Index

## Répertoire de travail

```
README.md
@@ -1 +1 @@
-TODO: README
+Hello world codes.
```

```
LICENSE      (clean)
README.md    (staged)
hello.py     (clean)
```

3dcfae1

master

5ba6158

HEAD

```
$ git add README.md
```

Ajout dans l'index

# Création de commits : exemple

## Historique

## Index

## Répertoire de travail

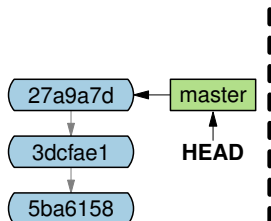


```
$ git commit -m 'doc: README'
```

Création du commit

# Création de commits : exemple

## Historique



## Index

## Répertoire de travail

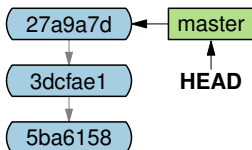
LICENSE (clean)  
README.md (clean)  
hello.py (modified)

```
$ vim hello.py  
$ chmod +x hello.py
```

Travail sur hello.py

# Création de commits : exemple

## Historique



## Index

```
hello.py
old mode 100644
new mode 100755
@@ -1,2 +1,2 @@
-#!/usr/bin/python3
+#!/usr/bin/env python3
print('Hello world!')
```

## Répertoire de travail

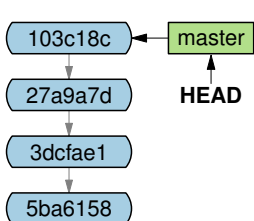
```
LICENSE      (clean)
README.md    (clean)
hello.py     (staged)
```

```
$ git add hello.py
```

Ajout dans l'index

# Création de commits : exemple

## Historique



## Index

## Répertoire de travail

LICENSE (clean)  
README.md (clean)  
hello.py (clean)

```
$ git commit -m 'fix python example: #!, mod+x'
```

Création du commit

# Visualisation de l'historique

## Arbre des commits

- `git log [<revision range>] [--] <path>...`

Options courantes.

- `--oneline` : concis (1 ligne par commit)
- `--graph` : parenté entre commits
- `--all` : affiche toutes les branches
- Celles de `git show...`

## Un commit en particulier

- `git show [<commit>...] [--] <path>...`

Options courantes.

- `-p, --patch` : modifications du commit
- `--stat` : résumé des modifications par commit

# Déplacement dans l'historique

## Déplacement total (HEAD + répertoire de travail)

- `git checkout <branch>` : vers <branch>
- `git checkout <commit>` : vers <commit> (*detached HEAD*)

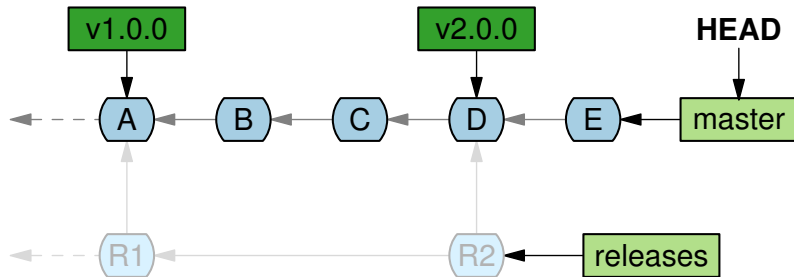
## Ne déplacer que HEAD (conserver répertoire de travail)

- `git reset <t-ish>` : vers une branche, un commit, un tag...



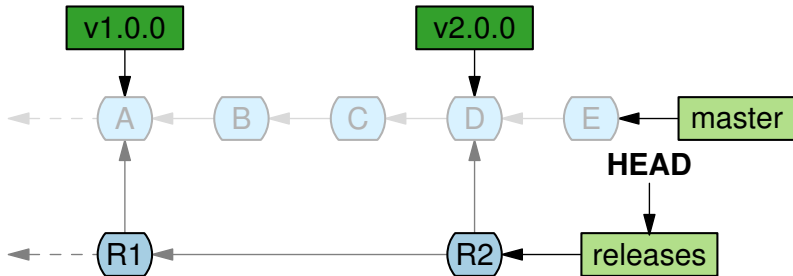
# Déplacement dans l'historique : exemple

```
$ git checkout master
```



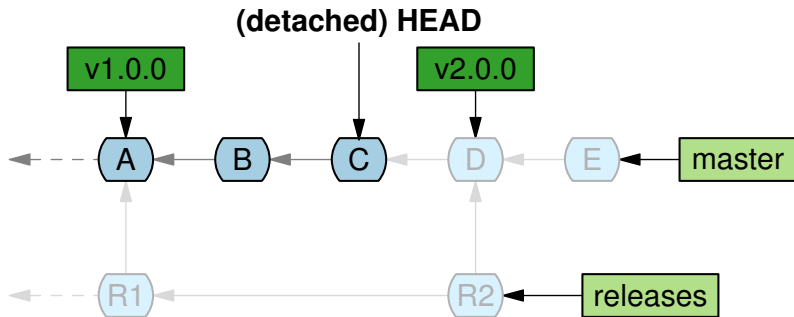
# Déplacement dans l'historique : exemple

```
$ git checkout releases
```



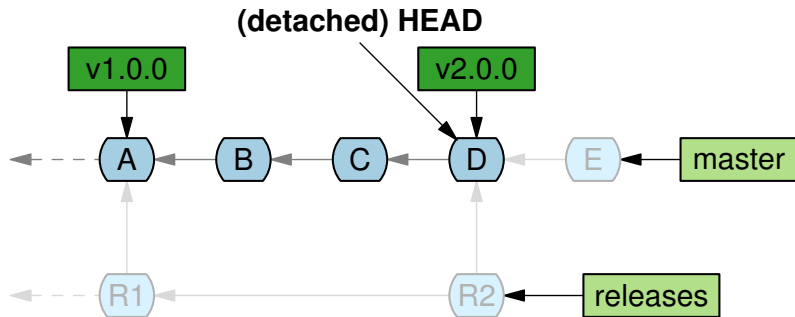
# Déplacement dans l'historique : exemple

```
$ git checkout C
```



# Déplacement dans l'historique : exemple

```
$ git checkout v2.0.0
```



## 4 Git (2)

- Gestion de branches

# Créer une branche

## Sans s'y déplacer (HEAD inchangé)

- `git branch <branch-name> [<start-point>]`

Options courantes.

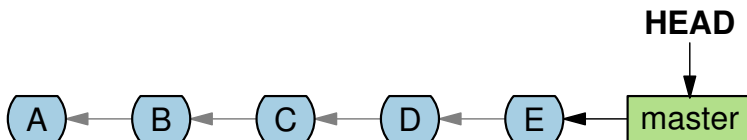
- `-f, --force` : met à jour <branch-name> si elle existe déjà

## En s'y déplaçant (HEAD pointe vers la nouvelle branche)

- `git checkout -B <branch-name> [<start-point>]`

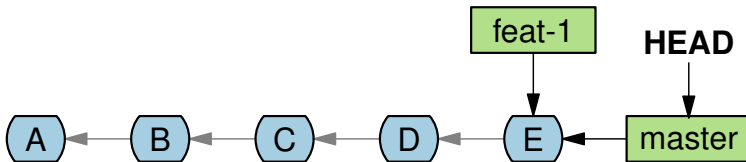
# Créer une branche : exemple

```
$ git checkout master
```



# Créer une branche : exemple

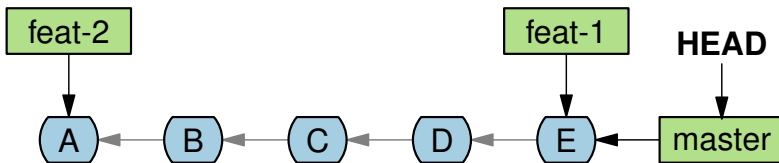
```
$ git branch feat-1
```





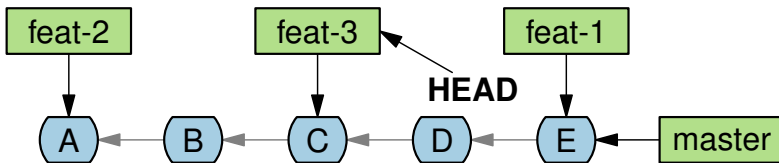
# Créer une branche : exemple

```
$ git branch feat-2 A
```



## Créer une branche : exemple

```
$ git checkout -B feat-3 C
```



# Fusion de branches

- `git merge [-m <msg>] <stuff...>`

Fusionne `stuff` dans `HEAD`. `stuff` peut être :

- Une branche
- Un commit

Peut créer un commit de fusion ou non selon les options :

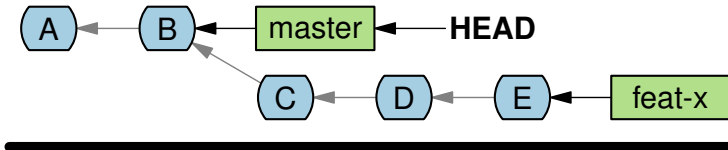
- `--ff-only` en cas de fast-forward<sup>4</sup>, met juste `HEAD` à jour (ne crée pas de commit). Sinon, ne fait rien
- `--no-ff` crée toujours un commit de fusion
- `--ff` (défaut) met juste `HEAD` à jour en cas de fast-forward. Sinon, crée un commit de fusion

---

4. Arrive lorsque les commits à intégrer sont directement au-dessus de `HEAD`.

# Fusion de branches : exemple *fast forward*

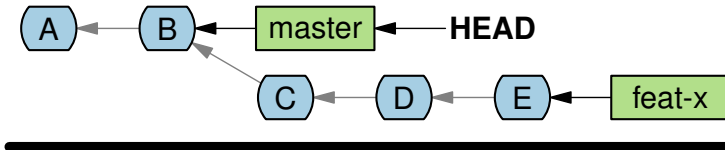
## Avant



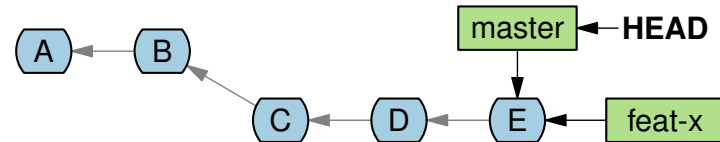
```
$ git checkout master
```

# Fusion de branches : exemple *fast forward*

## Avant



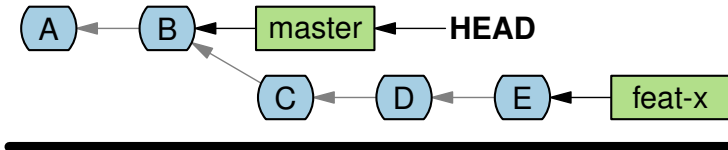
## Après



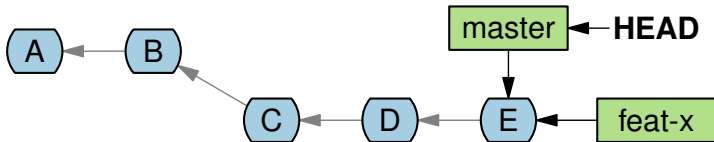
```
$ git merge --ff-only feat-x
```

# Fusion de branches : exemple *fast forward*

## Avant



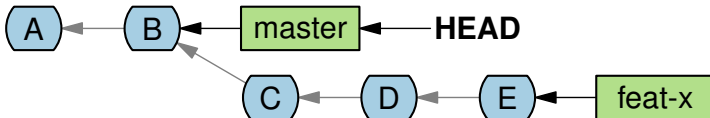
## Après



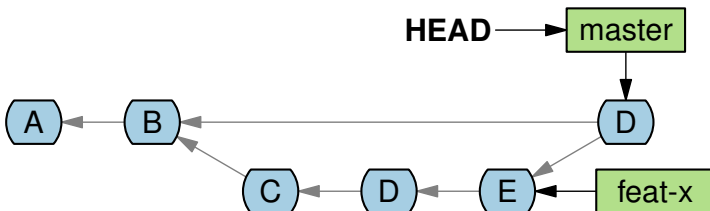
```
$ git merge --ff feat-x
```

# Fusion de branches : exemple *fast forward*

## Avant



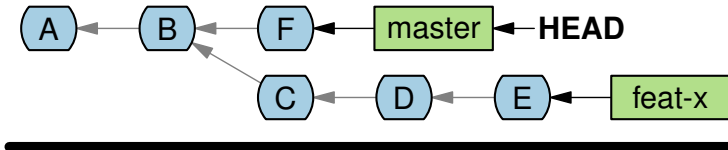
## Après



```
$ git merge --no-ff feat-x
```

# Fusion de branches : exemple non *fast forward*

## Avant

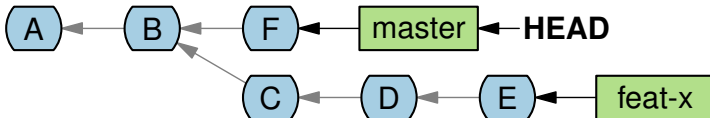


```
$ git checkout master
```

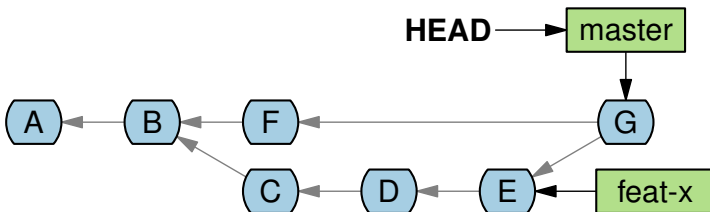


# Fusion de branches : exemple non *fast forward*

## Avant



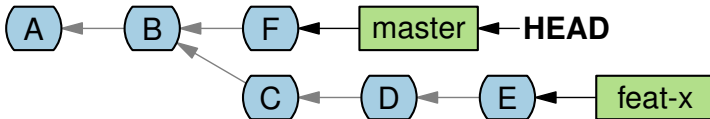
## Après



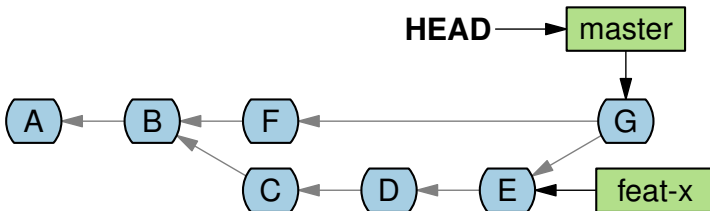
```
$ git merge --no-ff feat-x
```

# Fusion de branches : exemple non *fast forward*

## Avant



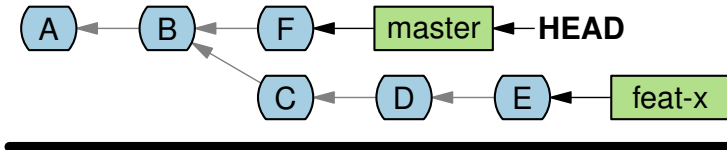
## Après



```
$ git merge --ff feat-x
```

## Fusion de branches : exemple non *fast forward*

**Avant**



**Après**

Not possible to fast-forward, aborting.

```
$ git merge --ff-only feat-x
```

# Changer la base d'une branche

- `git rebase <newbase> [<branch>]`

Change la base de <branch> (par défaut HEAD), de telle sorte que <newbase> devienne sa nouvelle base.

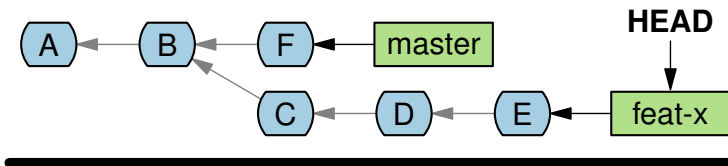
- Ré-applique chaque commit par-dessus la nouvelle base
- Permet de réécrire l'historique
- Très utile pour revenir en *fast forward*

Options courantes :

- `-i`, `--interactive` permet de choisir finement quoi faire de chaque commit

# Changer la base d'une branche : exemple

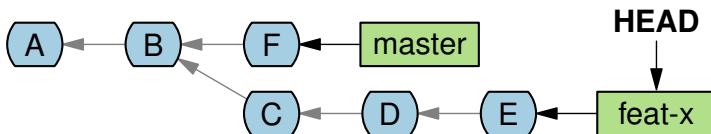
## Avant



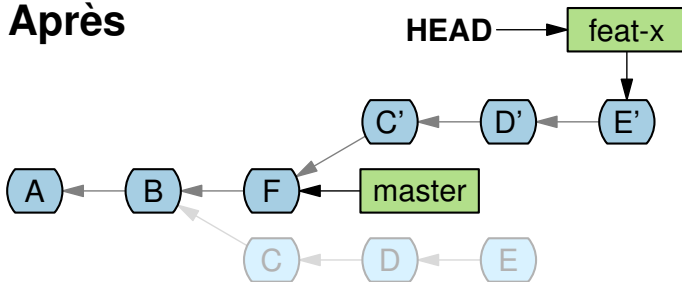
```
$ git checkout feat-x
```

# Changer la base d'une branche : exemple

## Avant



## Après



```
$ git rebase master
```

## 5 Git (3)

### ■ Synchronisation

# Récupérer des objets depuis un dépôt distant

```
git fetch [<repo>]
```

- Met à jour vos branches/tags distants
- Ne change **rien** à vos branches locales
- → Intégration manuelle avec merge ou rebase

## Exemples

- `git fetch`
- `git fetch origin`
- `git fetch --all`



# Récupérer et intégrer depuis des objets distants

```
git pull [<repo> [<ref>]]
```

Raccourci pour :

- fetch + merge : `git pull [--ff-only|--no-ff|--ff]`
- fetch + rebase : `git pull --rebase`

Comportement par défaut configurable (`man git config`).

## Exemples

- `git pull --ff-only origin master`
- `git pull --rebase`

# Envoyer vos objets sur un dépôt distant

```
git push [<repo> [<ref>...]]
```

Met à jour les <ref> distantes (branche, tag...) à partir de vos versions locales. Options courantes :

- `--force` l'acceptation distante de la nouvelle version de <ref>.  
**Très dangereux sur branche publique !**

## Exemples

- `git push origin master`
- `git push`

# Semantic Versioning<sup>7</sup>

Système générique de numérotation de versions.

- Expose aux utilisateurs les cassures d'API  
→ Réduit le *dependency hell*<sup>5</sup>
- Ensemble de règles strictes à suivre

Utilisé dans les dépôts de paquets des langages.

- Python : PyPI a un système proche – cf. PEP440<sup>6</sup>
- D : Dub l'utilise
- Rust : Cargo l'utilise

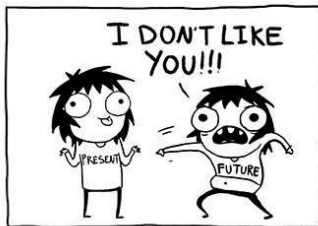
---

5. [https://en.wikipedia.org/wiki/Dependency\\_hell](https://en.wikipedia.org/wiki/Dependency_hell)

6. <https://www.python.org/dev/peps/pep-0440/>

7. <https://semver.org>

# Message de commit



© Sarah Andersen

Une ligne. Informatif. Concis (< 50 char)

Détails si besoin.

- Une ligne vide avant les détails !

# commentaire

# merge ou rebase ?

## rebase

- Réécrit l'historique
  - **jamais sur branche publique**<sup>8</sup>
  - Pratique en local pour réduire branchements inutiles

## merge

- Conserve le *vrai* parent d'un commit
  - Peut aider à comprendre un commit plus tard

---

8. <https://www.atlassian.com/git/tutorials/merging-vs-rebasing#the-golden-rule-of-rebasing>

# Workflow Git

Modélise comment collaborer via Git.

- Défini par projet
- Gitflow<sup>9</sup> est populaire

Bonnes idées pour une bonne qualité.

- Maintenir une branche qui *marche* toujours (tests automatisés)
- Maintenir une documentation à jour (changelog)

→ Sortir *souvent* une nouvelle version devient facile.

---

9. <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

# Remerciements et références

Pour faire ces slides

- Arnaud Legrand — idées de slides de mooc-rr<sup>10</sup>
- Adoption des VCS dans l'open source<sup>11</sup>

Pour aller plus loin

- Documentation officielle de git<sup>12</sup>
- `man git`, `man git checkout...`

---

10. <https://gitlab.inria.fr/learninglab/mooc-rr> (module 1, slides)

11. <https://mpoquet.github.io/blog/2020-08-vcs-adoption-in-floss/index.html>

12. <https://git-scm.com/doc>