

1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# LOG1000

# Ingénierie logicielle

## ***Gestion de configuration***

mathieu.lavallee@polymtl.ca

Département de génie informatique et de  
génie logiciel

1

Avec du matériel produit par Bram Adams et Michel Gagnon

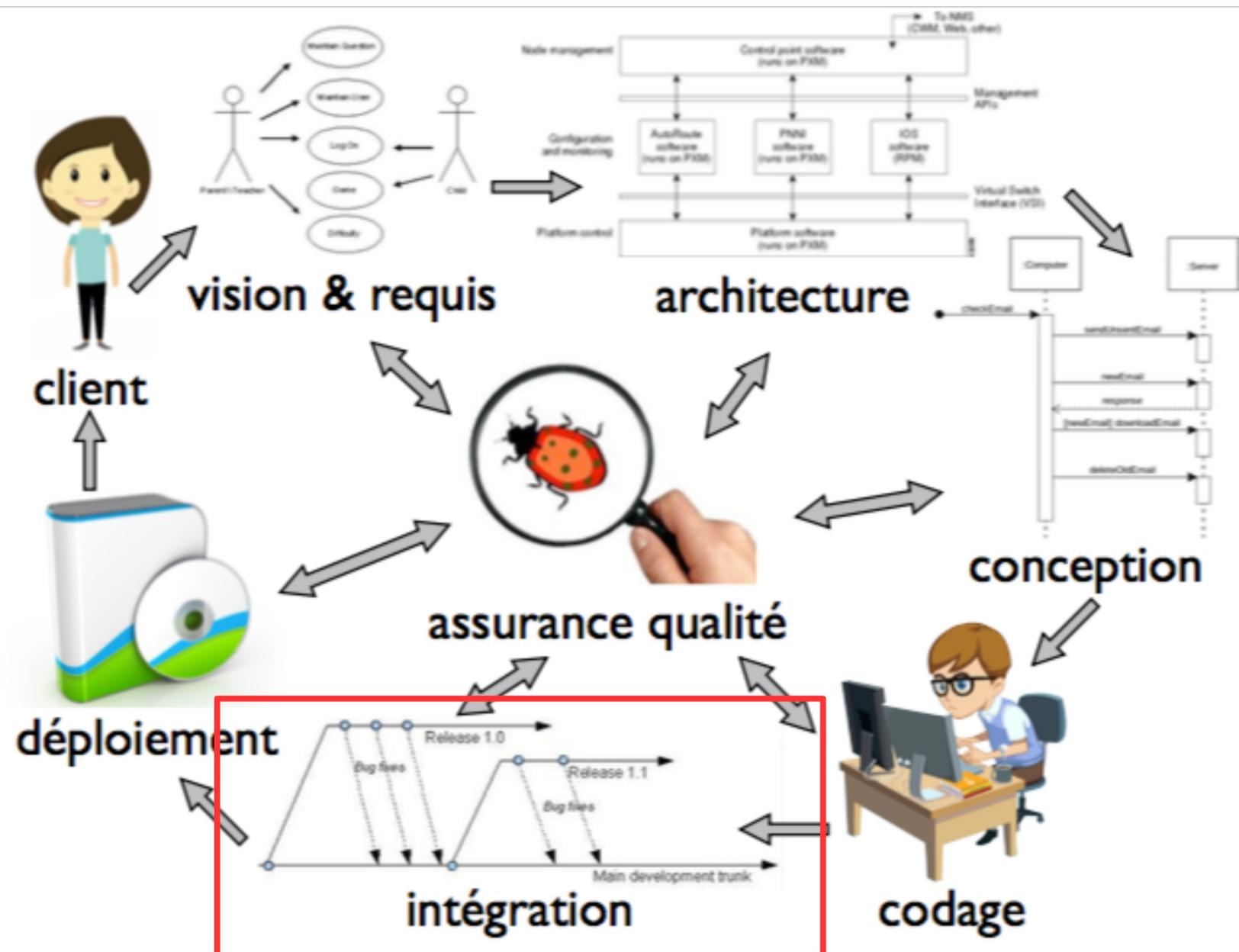
1

1. Définition

2. Principes de fonctionnement et outils

3. Commandes de base Git

# Gestion de configuration logicielle



## 1. Définition

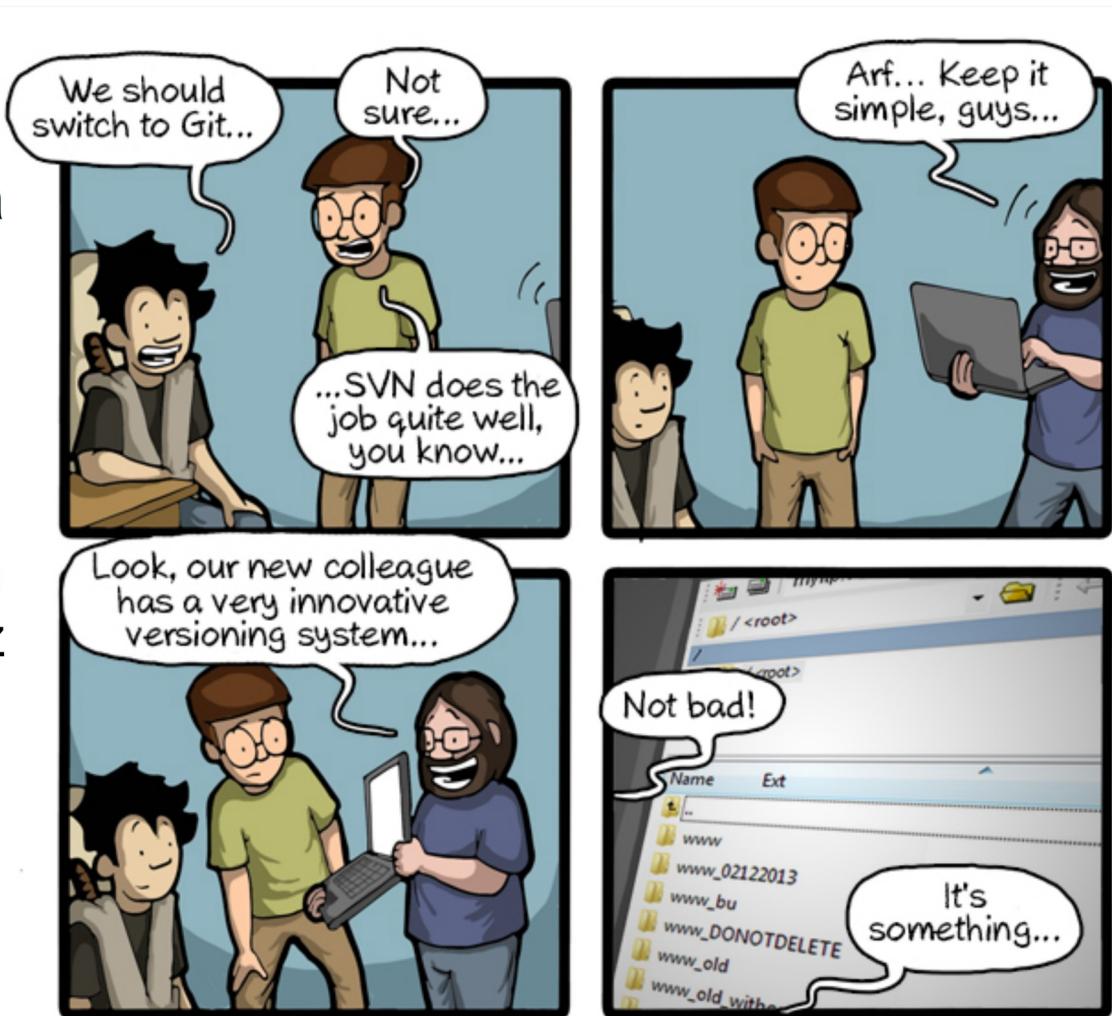
## 2. Principes de fonctionnement et outils

## 3. Commandes de base Git

# Gestion de configuration logicielle

- Gestion de versions (*versioning*) : comment s'assurer que toute l'équipe travaille sur les bonnes versions des fichiers de code ?

- On s'échange des fichiers par courriel ? On met ça sur Dropbox ?
- Touchez pas au fichier main.cpp, je suis en train de le modifier !
- J'ai corrigé le bogue dans xyz.cpp, soyez sûrs d'avoir la dernière version !
- Zut, j'ai brisé quelque chose ... comment revenir en arrière ?



1. Définition

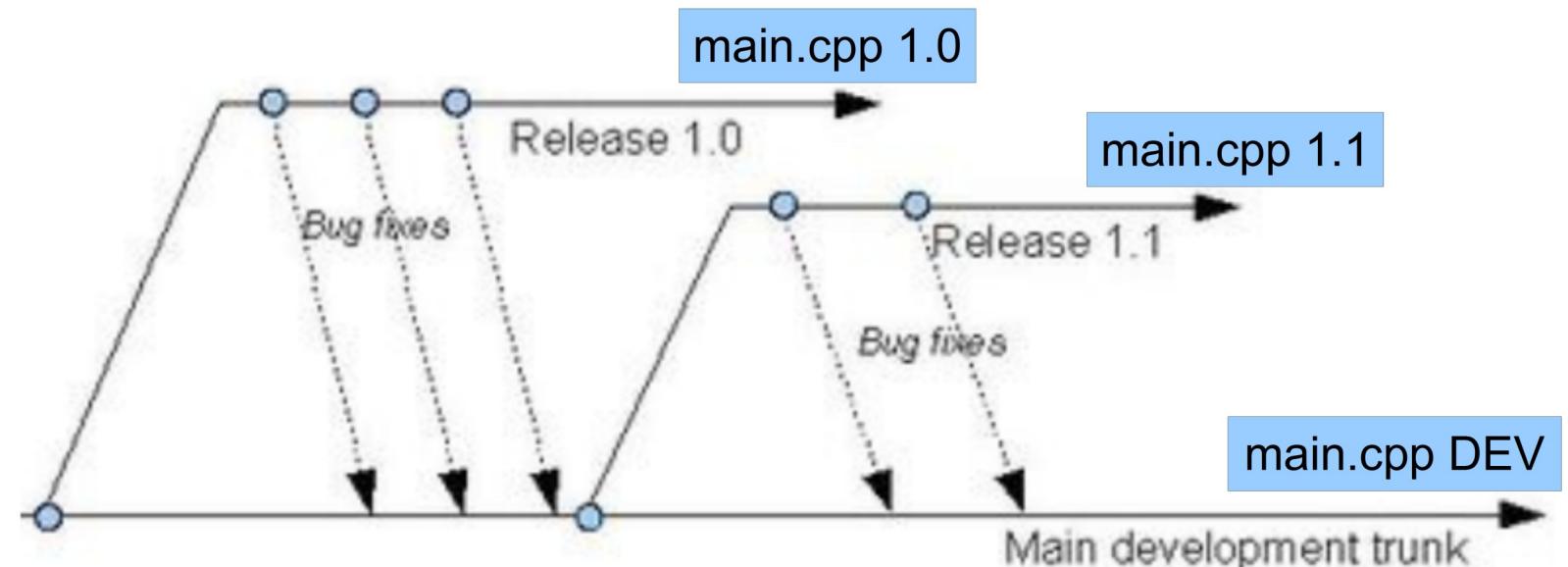
2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Gestion de configuration logicielle

- Une bonne gestion des versions facilite l'intégration du code en identifiant clairement quel code va avec quelle version.

- Par exemple, on peut donc se retrouver avec plusieurs *versions* du fichier "main.cpp" :



- Cela nous permet de modifier le code sans affecter des versions fonctionnelles précédentes.

# Gestion de configuration logicielle

- Définition de la gestion de configuration logicielle :
  - Technique qui permet d'**identifier les artéfacts** du projet et de **gérer systématiquement les demandes de modification**, afin que le système puisse conserver son intégrité au fil du temps.
  - En bref : contrôle des demandes de changement (DDC), des *change request* (CR).



# Gestion de configuration logicielle

- Quels **artéfacts** sont suivis par la gestion de configuration ?

1. Définition

2. Principes de fonctionnement et outils

3. Commandes de base Git

Ex.:  
SVN,  
Git.

**Code source**  
(ex.: fichiers  
.cpp, .h, makefile,  
texte brut)

Ex.: Script de  
création des  
bases de  
données.

**Données**  
(contenu que le  
logiciel doit  
manipuler)

**Documents**  
(ex.: fichiers  
binaires DOC, PDF  
de conception,  
exigences)

Ex.:  
SharePoint,  
*Content Management Systems*  
(CMS).

**Outils**  
(configuration des  
outils utilisés, du  
*workspace*)

Outils de  
gestion des  
espaces de  
travail.

1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Gestion de configuration logicielle

- Dans un grand projet :
  - Procédures de sauvegarde et de suivi des versions de tous les artefacts importants.
  - Ex.: 1M lignes de code  $\approx$  5000 pages de documentation  $\rightarrow$  LibreOffice  $\approx$  15M lignes de code !
  - Cela inclut aussi les tests, les schémas de base de données, les données de configuration des outils, etc. etc. etc.
  - Contrôle systématique des exigences et de la conception.
    - Le logiciel d'une voiture  $\rightarrow$  5000+ exigences qui doivent changer régulièrement ...
    - Il ne faut rien laisser échapper !

1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Activités de gestion de configuration

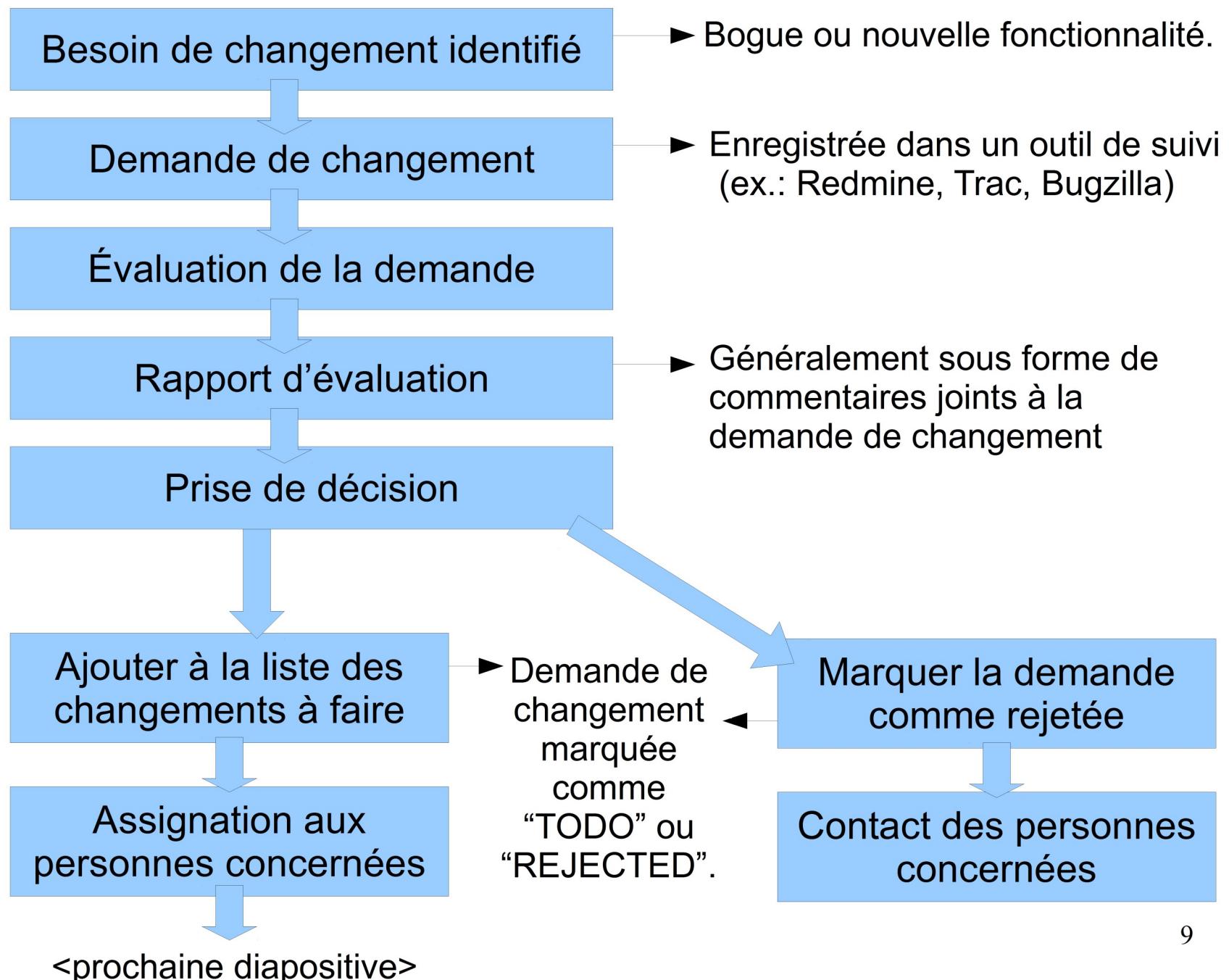
- **Identifier les demandes de modification/changement.**
  - Ex.: bogues à corriger, nouvelles fonctionnalités à implémenter.
- **Contrôler les demandes de modification/changement.**
  - Est-ce un vrai bogue ? Est-ce réellement une fonctionnalité manquante ?
  - Est-ce que le changement en vaut la peine ? Est-ce trop complexe/risqué/coûteux ?
- **Assurer l'implémentation des demandes de modification.**
  - Suivant le processus : communication → modélisation → construction → déploiement.
  - Implique utiliser des **outils de gestion de configuration** comme SVN, Git.
- **Informer les personnes concernées des modifications.**

1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Processus de contrôle des demandes de changement

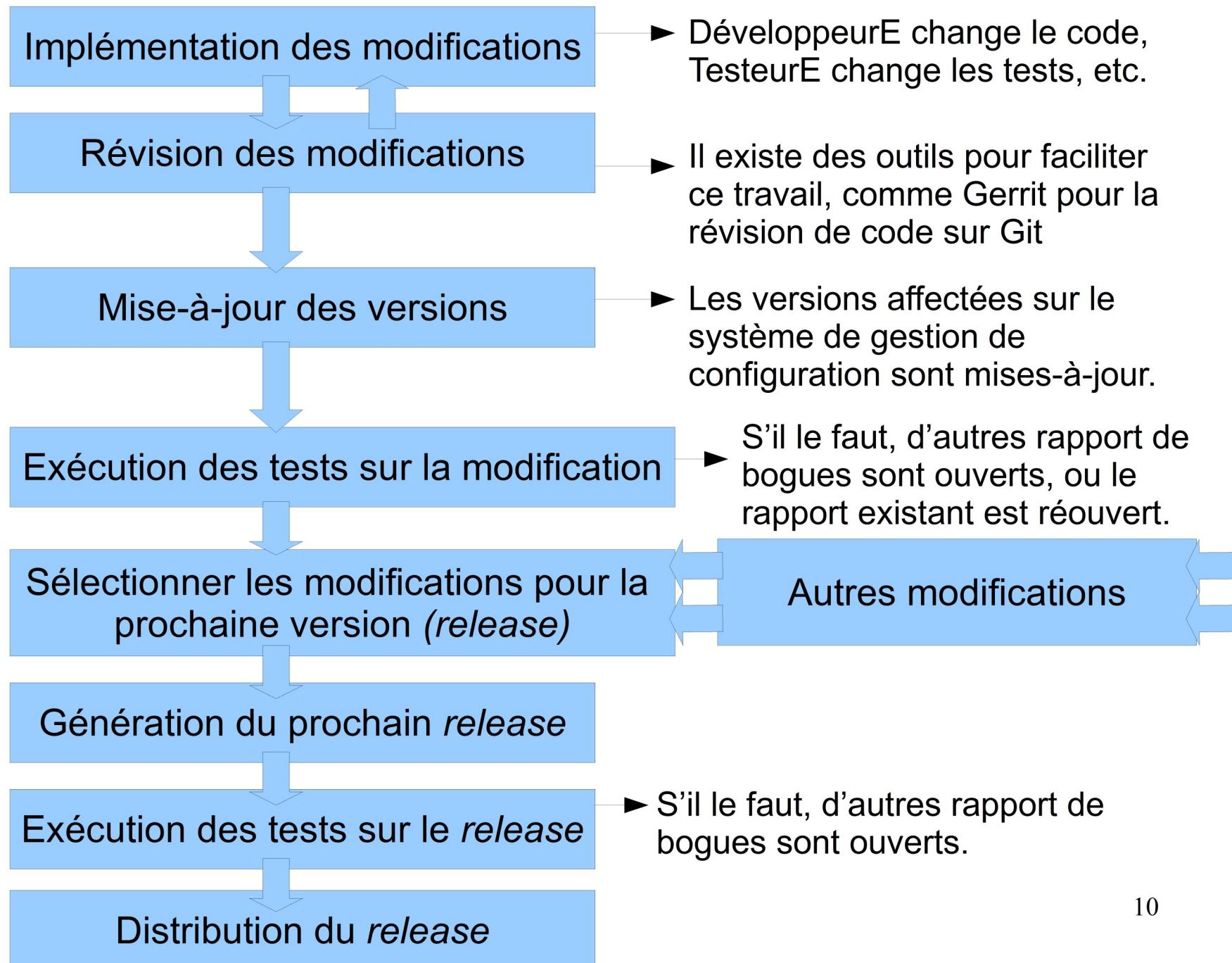


## 1. Définition

## 2. Principes de fonctionnement et outils

## 3. Commandes de base Git

# Processus de contrôle des demandes de changement



1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Exemple concret avec le logiciel *open source* Eclipse

- Un bogue (en fait une demande de nouvelle fonctionnalité) est soumis :  
[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=408543](https://bugs.eclipse.org/bugs/show_bug.cgi?id=408543)
- Un développeur soumet une première proposition de correction : <https://git.eclipse.org/r/#/c/28831/>
- La proposition est rejetée après révision : c'est trop complexe → il faut scinder le changement en trois parties afin de faciliter les tests.
- Une deuxième proposition est soumise :  
<https://git.eclipse.org/r/#/c/29350/2>
- Cette proposition n'est pas fusionnée (*merged*) telle qu'elle, des réviseurs trouvent des problèmes à corriger.
- Après plusieurs versions, le code est finalement fusionné au code du logiciel Eclipse !



# Révision des changements

## 1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

- **Les modifications autorisées ont-elles toutes été faites ?**
  - Ex.: est-ce que le code modifié règle le bogue ?
- **Y a-t-il des modifications supplémentaires qui ont été apportées ?**
  - Ex.: y a-t-il un problème imprévu qui nous a forcé de changer un autre fichier → qui pourrait causer des problèmes à d'autres développeurEs ?
- **Le processus de développement et les normes ont-ils été respectés ?**
  - Ex.: est-ce que le code est commenté de la manière recommandée par l'organisation ?
- **Les modifications ont-elles été clairement indiquées dans les items concernés ?**
  - Ex.: marquer clairement qu'on a ajouté une nouvelle exigence dans le document d'exigence pour que tout le monde le sache.
- **Tous les items ont-ils été mis-à-jour ?**
  - Ex.: voir le cas de Knight Capital ...

1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Bonnes pratiques (*best practices*) en gestion de configuration

- Adopter une méthode **systématique** de contrôle des demandes de modification.
  - Dans la mesure du possible, éviter de *bypasser* le processus en place → source d'erreurs.
- Traiter les demande de modifications **par groupes**.
  - En *releases* idéalement.
  - Estimer le **coût** de chaque modification demandée.
    - En temps, en argent, mais aussi le risque que ça ne se passe pas tel que prévu ...
- Se méfier des volumes élevés de demandes de modification.
  - Ça peut indiquer que le travail fait en communication (exigences) et modélisation était incorrect ...
- Choisir un **point milieu** entre une bureaucratie trop lourde et une absence de contrôle.

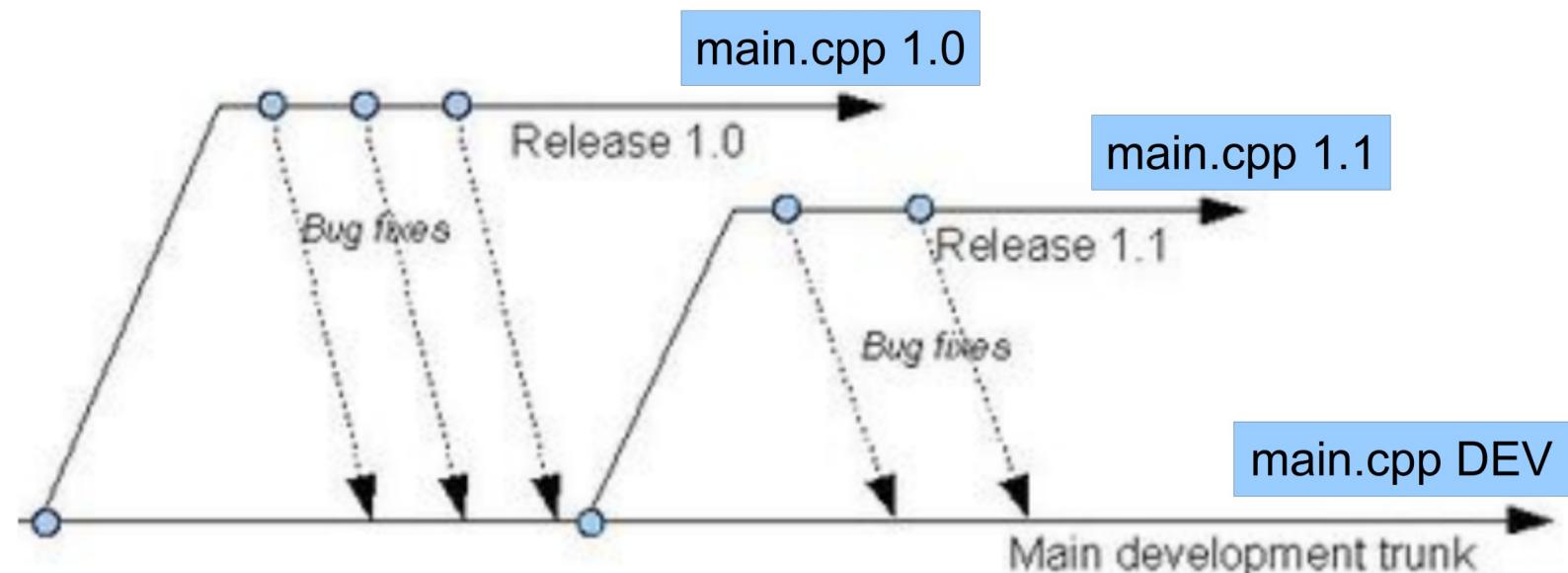
## 1. Définition

## 2. Principes de fonctionnement et outils

## 3. Commandes de base Git

# Système de gestion des versions

- Outil pour gérer les implémentations des modifications demandées, c'est-à-dire les changements ("patches").
- Gère l'accès de manière à ce que plusieurs personnes puissent partager les mêmes fichiers.
- Garde l'historique des modifications.



## 1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Version stable, version de référence

- Appelée « *baseline* » en anglais.
- Spécification ou produit ayant été formellement révisé et accepté, servant de base pour le développement futur, et qui peut être modifié seulement par le biais de procédures formelles.
- Exemples :

« baseline » d'un document



« baseline » d'un logiciel



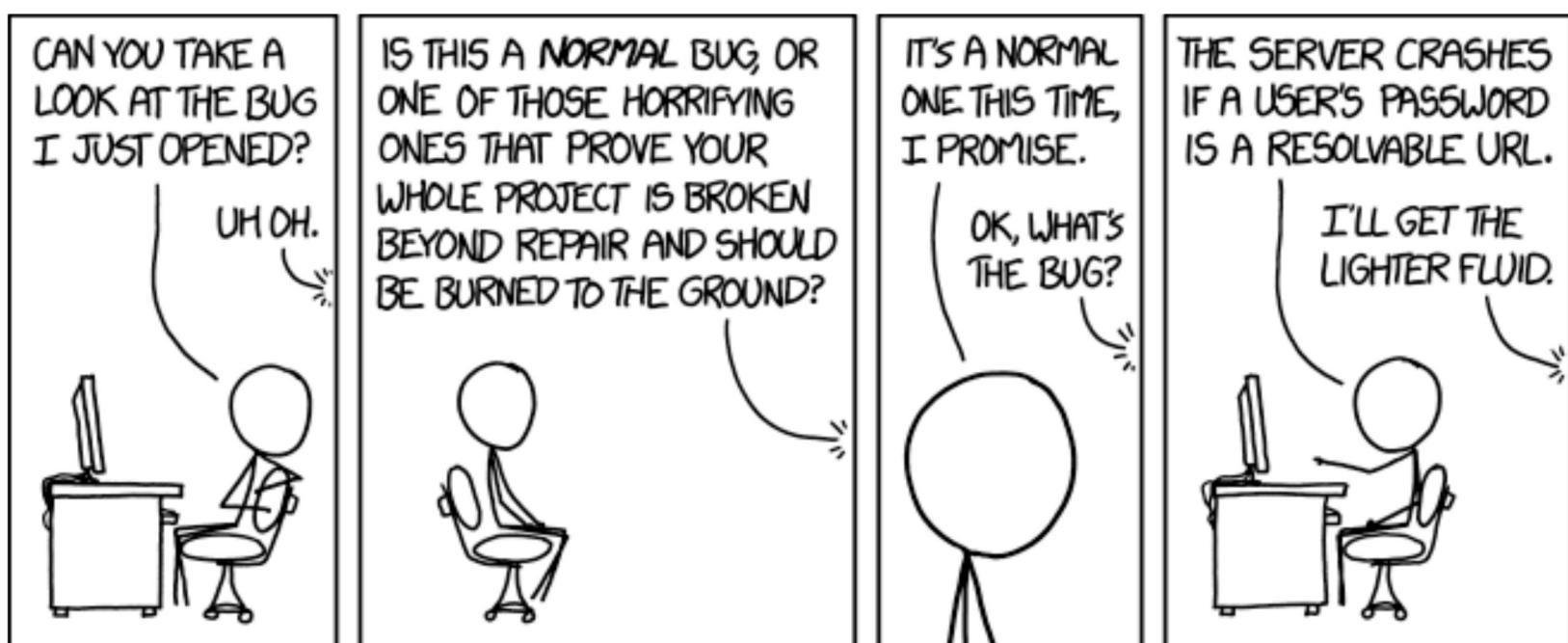
1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Système de gestion des versions

- Deux approches principales :
  - **Blocage de fichiers** : Je bloque l'accès au fichier exigences.odt tant que je suis en train de le modifier.
  - **Fusion de fichiers** : Tout le monde peut modifier le fichier main.cpp et on réglera les incohérences (conflits) par la suite.



Source : XKCD

# Blocage de fichier

## 1. Définition

## 2. Principes de fonctionnement et outils

## 3. Commandes de base Git

- Tant que le développeur ne libère pas le fichier, personne d'autre ne peut y apporter des modifications.
- **Avantages :**
  - Utile quand le fichier modifié n'est pas en texte brut.
    - Ex.: Fichier DOC, PPT, PDF, images.
    - Ces fichiers sont enregistrés en binaire et il n'est pas possible de fusionner des versions différentes.
  - Parfois utile pour du travail urgent ou très complexe ...
- **Inconvénients :**
  - Empêche le travail en parallèle.
  - Cause des problèmes si on oublie de libérer le fichier et qu'on est absent ...
  - Peut être utilisé de manière abusive par des collègues qui n'aiment pas fusionner les versions différentes ...
  - N'empêche pas les incompatibilités avec d'autres fichiers.
    - Ex.: Si un collègue renomme une dépendance.

## 1. Définition

2. Principes de fonctionnement et outils

3. Commandes de base Git

# Fusion de fichiers

- Les développeurEs sont responsables de résoudre les conflits avant de pouvoir sauvegarder ses modifications dans l'entrepôt.
- **Avantages** ( $\approx$  inverse du blocage) :
  - Permet le travail en parallèle.
  - Il n'y a pas de risque d'oublier de libérer un fichier.
- **Inconvénients** ( $\approx$  inverse du blocage) :
  - Avoir des versions conflictuelles de fichiers binaires (DOC, PPT, PDF ...) est une bonne source de migraine ...
  - La résolution de conflit peut prendre du temps, surtout si deux personnes ont modifié le même fragment de code.



# Principes de fonctionnement

```
int main(int argc, char **argv) {  
    return 0;  
}
```

*REMOTE*

J'obtiens le code du dépôt (*repository*) partagé en ligne.

```
int main(int argc, char **argv) {  
    int x = mesure();  
    std::cout << x;  
    return 0;  
}
```

```
int mesure() {  
    return 5;  
}
```

*LOCAL*

Je fais mes modifications localement.

```
int main(int argc, char **argv) {  
    int x = mesure();  
    std::cout << x;  
    return 0;  
}
```

```
int mesure() {  
    return 5;  
}
```

*REMOTE*

Quand je suis satisfait, je soumet (*commit*) le changement sur le dépôt partagé en ligne.

Le dépôt partagé possède maintenant ma dernière version de code. Tout le monde peut l'obtenir. Yay !

1. Définition

2. Principes de  
fonctionnemen  
t et outils

3. Commandes  
de base Git

# Principes de fonctionnement

```
int main(int argc, char **argv) {  
    int x = mesure();  
    std::cout << x;  
    return 0;  
}  
  
int calcul() {  
    return 5;  
}
```

*REMOTE v52*

J'obtiens le code du dépôt partagé en ligne.

Ça ne marche pas ... On a fait des changements qui ont brisé quelque chose ...

```
int main(int argc, char **argv) {  
    int x = mesure();  
    std::cout << x;  
    return 0;  
}  
  
int mesure() {  
    return 5;  
}
```

*REMOTE v51*

Pas de problème, je n'ai qu'à retourner à une version précédente (*revert*) qui fonctionnait !

Je peux aussi comparer (*diff*) les versions 51 et 52 pour voir ce qui a été changé et pourquoi ça ne marchait pas.

1. Définition

2. Principes de  
fonctionnement et outils

3. Commandes de base Git

# Principes de fonctionnement : fusion

```
int main(int argc, char **argv) {  
    int x = mesure();  
    std::cout << x;  
    return 0;  
}
```

```
<<<<< HEAD  
int calcul() {  
=====  
int mesure() {  
>>>>> 77976da  
    return 5;  
}
```

LOCAL

Vous essayez de soumettre votre code en ligne, mais ça ne fonctionne pas.

Quelqu'unE d'autre a soumis des changements au même fichier !

Il y a un conflit (*conflict*) ... entre votre version (HEAD) et la version qui a été soumise (77976da).

```
int main(int argc, char **argv) {  
    int x = mesure();  
    std::cout << x;  
    return 0;  
}  
  
int calcul_mesure() {  
    return 5;  
}
```

LOCAL

Les marques de conflit ont été mises directement dans le code. Il faut donc l'éditer afin de dire à l'outil quelle version garder (la nôtre, la précédente, ou un mélange des deux).

# Systèmes de gestion de versions

- Il existe une très grande quantité d'outils de gestion des versions. Parmi les plus populaires :

- SVN (Subversion) :** 
  - Gestion des branches difficile.
- Git :** 
  - Très populaire en ce moment.
- Mercurial :** 
  - Similaire à Git.
- CVS :**
  - Ancien, aucun changement depuis 2008.
- Microsoft TFS (Team Foundation System) / VSTS (Visual Studio Team System) :** 
  - Inclut des outils de gestion de projet.

# Systèmes de gestion de versions

- SVN a été populaire pendant très longtemps, mais est lentement remplacé par Git.
  - Entre 30% et 40% des projets à code ouvert sont gérés avec Git, en hausse continue.
  - La même proportion pour SVN, en chute libre.
  - Le projet intégrateur de 1ère année se fait donc sur Git.
    - Il faut donc savoir comment s'en servir !
    - Il existe des interfaces graphiques pour Git, mais il est préférable de commencer en bonne vieille ligne de commande.
  - Git est déjà installé sur les postes de l'école.
    - Si vous travaillez sur un autre ordi commencez pas installer Git : <https://git-scm.com/>

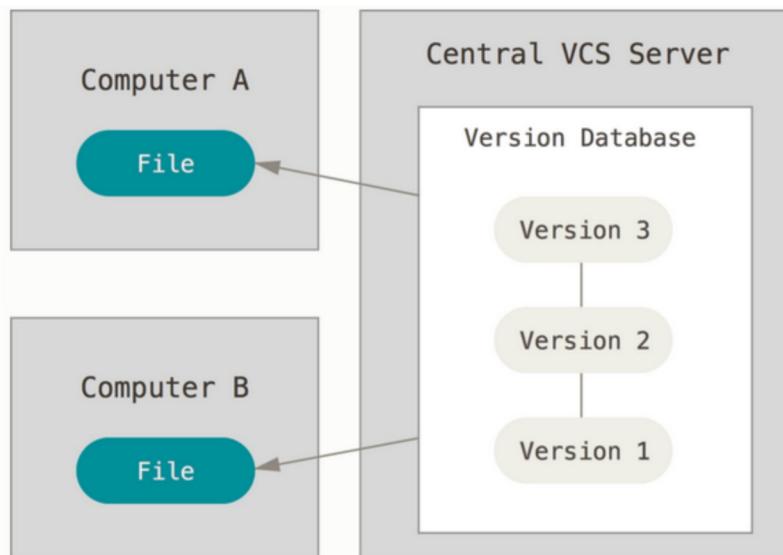
1. Définition

2. Principes de fonctionnement et outils

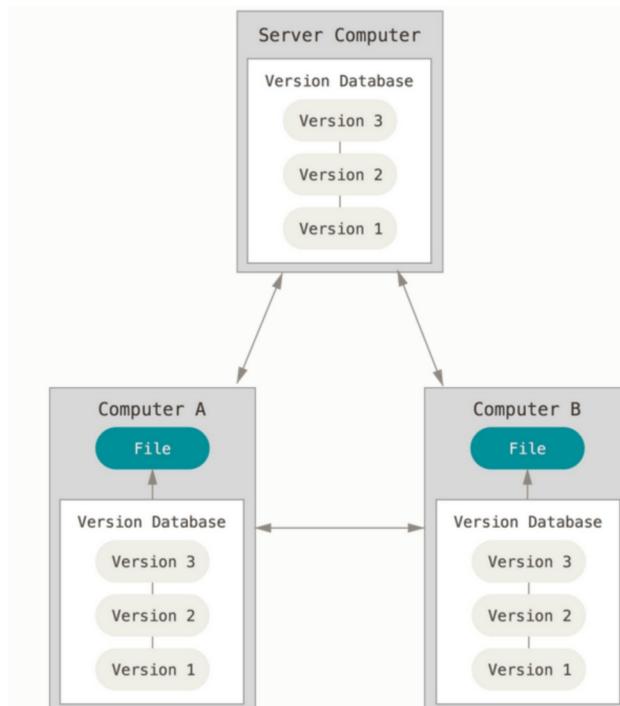
3. Commandes de base Git

# SVN vs. Git

- SVN est un système de gestion de versions **centralisé**.
  - Seul le serveur garde l'historique des versions. Les clients ne font que manipuler les fichiers.
- Git est un système de gestion de versions **distribué**.
  - Les clients n'ont pas seulement des fichiers, mais dupliquent complètement l'historique sur le serveur.



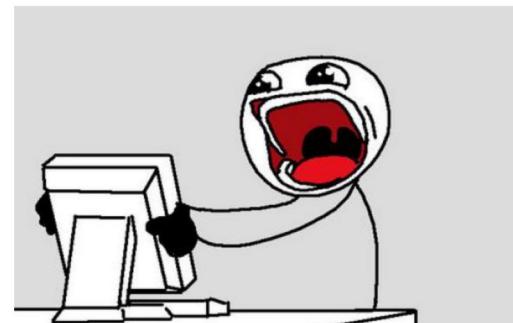
Approche de SVN (centralisé)



Approche de Git (distribué)

# Principes de fonctionnement de Git

- Git est un exemple de gestionnaire de versions fonctionnant par fusion de fichiers.
  - Avec le Git de base, il n'est pas possible de bloquer des fichiers → Pas de fichiers binaires (DOC, PPT, PDF etc.).
- Les publications de changements (*commit*) peuvent être calibrés comme on le désire.
  - On recommande de publier fréquemment des changements de petite taille (atomiques).
- Permet de renommer et déplacer des fichiers et des répertoires sans perdre l'historique des changements.

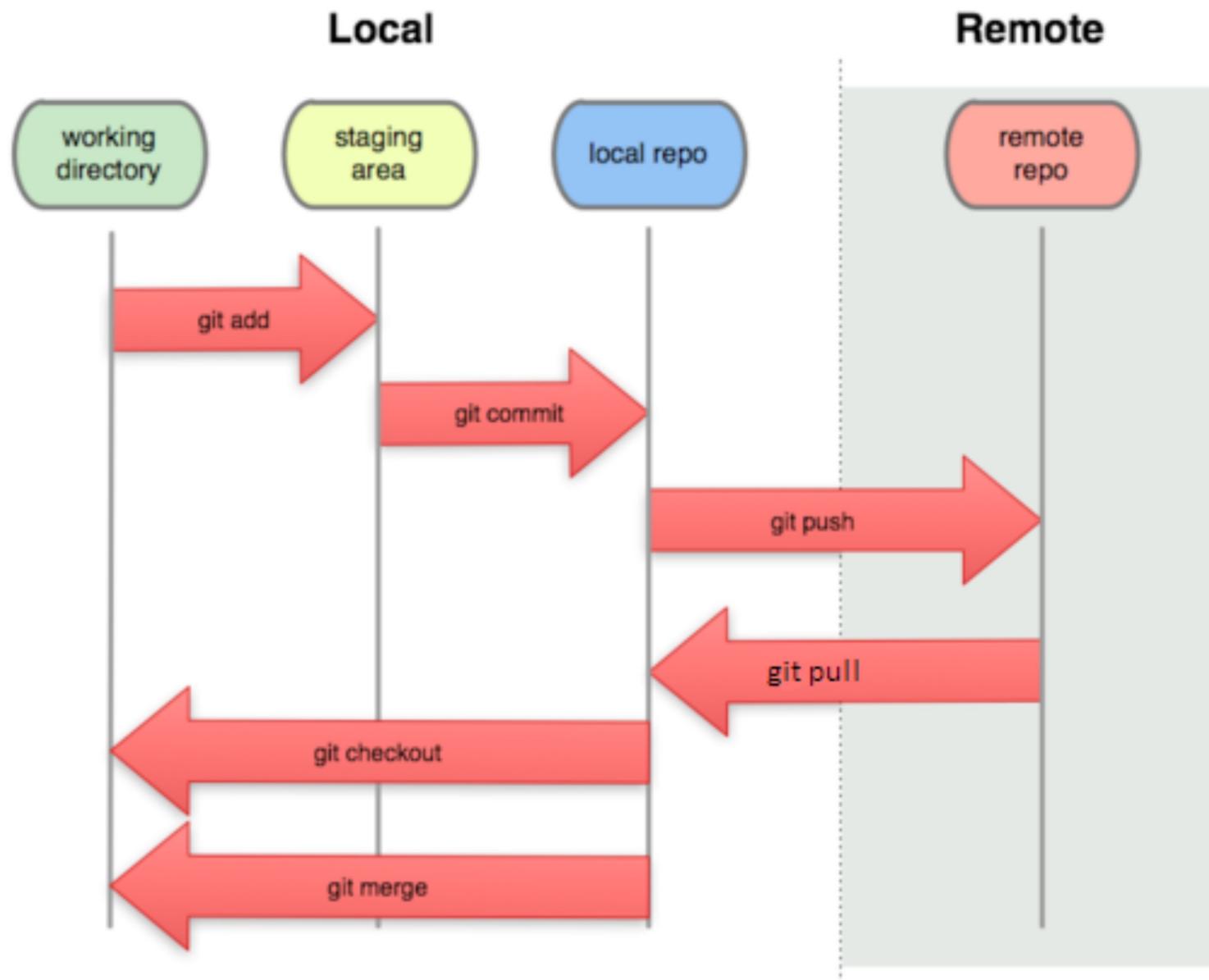


1. Définition

2. Principes de  
fonctionnemen  
t et outils

3. Commandes  
de base Git

# Principe de fonctionnement de Git

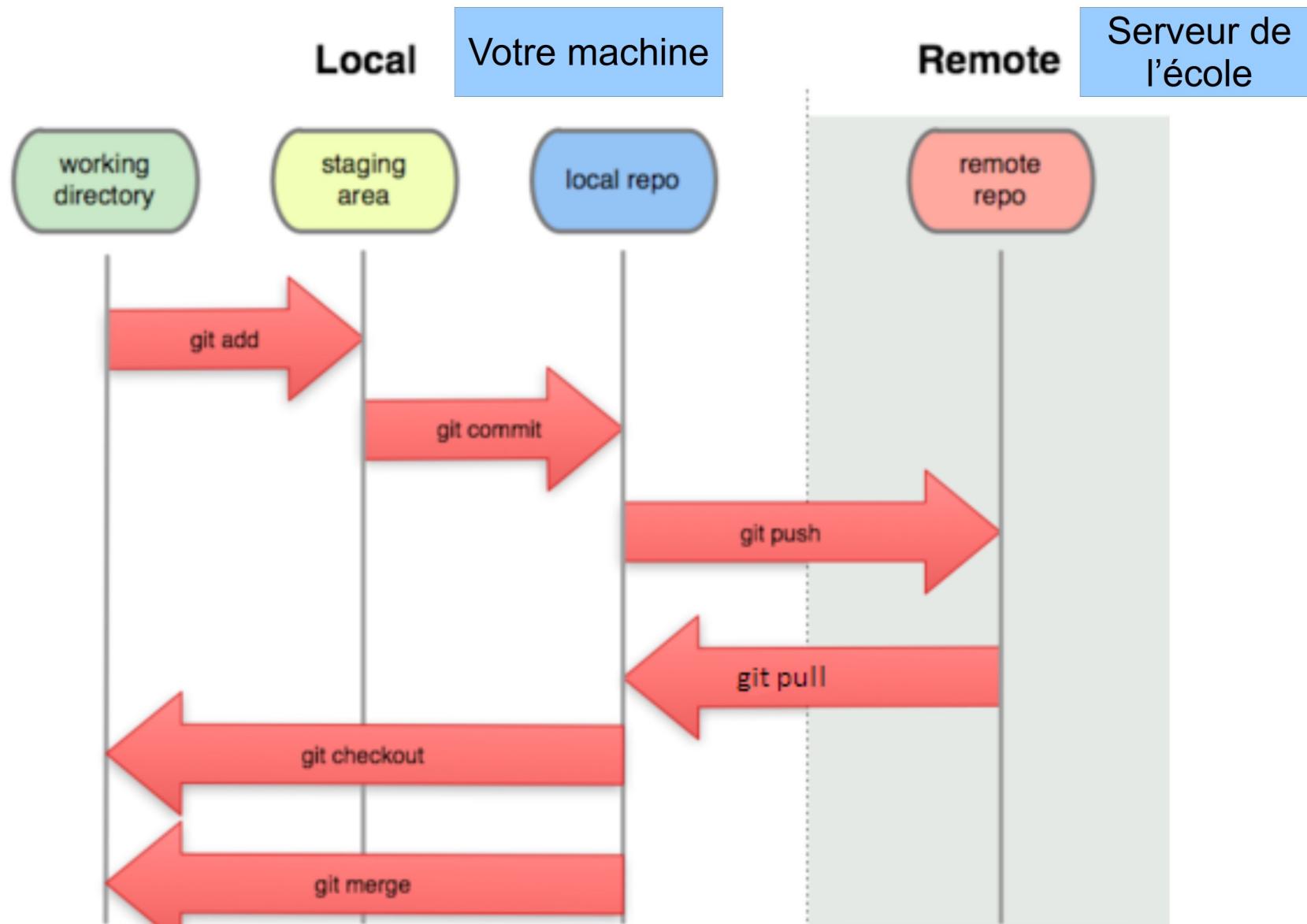


1. Définition

2. Principes de  
fonctionnemen  
t et outils

3. Commandes  
de base Git

# Principe de fonctionnement de Git



Code que  
vous venez  
d'écrire

Ensemble de  
modifications  
que vous  
voulez ajouter

Code prêt à  
être partagé ou  
à être fusionné  
avec le vôtre.

Code partagé  
avec tout le  
monde

1. Définition

2. Principes de fonctionnement et outils

3. Commandes de base Git

# Forces de Git

- Il est facile de s'assurer que nos *push* vers le *remote* (code partagé par tout le monde) contient une version de code qui marche !
  - Il suffit de tester le code du *local repo* avant de faire le *push*.
- On peut définir explicitement quels fichiers font partie d'un *commit*.
  - Ex.: Regroupe ensemble les modifications de code liés à un correctif de bogue.
  - Ex.: Permet de limiter le *commit* au code qu'on a terminé d'écrire.
- Gestion des branches facile.
  - On restera cependant aux notions de base cette session.

## 1. Définition

## 2. Principes de fonctionnement et outils

## 3. Commandes de base Git

# Git cheatsheet

### CREATE

Clone an existing repository

```
$ git clone ssh://user@domain.com/repo.git
```

Create a new local repository

```
$ git init
```

### LOCAL CHANGES

Changed files in your working directory

```
$ git status
```

Changes to tracked files

```
$ git diff
```

Add all current changes to the next commit

```
$ git add .
```

Add some changes in <file> to the next commit

```
$ git add -p <file>
```

Commit all local changes in tracked files

```
$ git commit -a
```

Commit previously staged changes

```
$ git commit
```

Change the last commit

*Don't amend published commits!*

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

### COMMIT HISTORY

Show all commits, starting with newest

```
$ git log
```

Show changes over time for a specific file

```
$ git log -p <file>
```

Who changed what and when in <file>

```
$ git blame <file>
```

### BRANCHES & TAGS

List all existing branches

```
$ git branch -av
```

Switch HEAD branch

```
$ git checkout <branch>
```

Create a new branch based on your current HEAD

```
$ git branch <new-branch>
```

Create a new tracking branch based on a remote branch

```
$ git checkout --track <remote/branch>
```

Delete a local branch

```
$ git branch -d <branch>
```

Mark the current commit with a tag

```
$ git tag <tag-name>
```

### UPDATE & PUBLISH

List all currently configured remotes

```
$ git remote -v
```

Show information about a remote

```
$ git remote show <remote>
```

Add new remote repository, named <remote>

```
$ git remote add <shortname> <url>
```

Download all changes from <remote>, but don't integrate into HEAD

```
$ git fetch <remote>
```

Download changes and directly merge/integrate into HEAD

```
$ git pull <remote> <branch>
```

Publish local changes on a remote

```
$ git push <remote> <branch>
```

Delete a branch on the remote

```
$ git branch -dr <remote/branch>
```

Publish your tags

```
$ git push --tags
```

### MERGE & REBASE

Merge <branch> into your current HEAD

```
$ git merge <branch>
```

Rebase your current HEAD onto <branch>

*Don't rebase published commits!*

```
$ git rebase <branch>
```

Abort a rebase

```
$ git rebase --abort
```

Continue a rebase after resolving conflicts

```
$ git rebase --continue
```

Use your configured merge tool to solve conflicts

```
$ git mergetool
```

Use your editor to manually solve conflicts and (after resolving) mark file as resolved

```
$ git add <resolved-file>
```

```
$ git rm <resolved-file>
```

### UNDO

Discard all local changes in your working directory

```
$ git reset --hard HEAD
```

Discard local changes in a specific file

```
$ git checkout HEAD <file>
```

Revert a commit (by producing a new commit with contrary changes)

```
$ git revert <commit>
```

Reset your HEAD pointer to a previous commit ...and discard all changes since then

```
$ git reset --hard <commit>
```

...and preserve all changes as unstaged changes

```
$ git reset <commit>
```

...and preserve uncommitted local changes

```
$ git reset --keep <commit>
```

Source :  
<https://www.git-tower.com/blog/git-cheat-sheet/>

1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Git : premières commandes

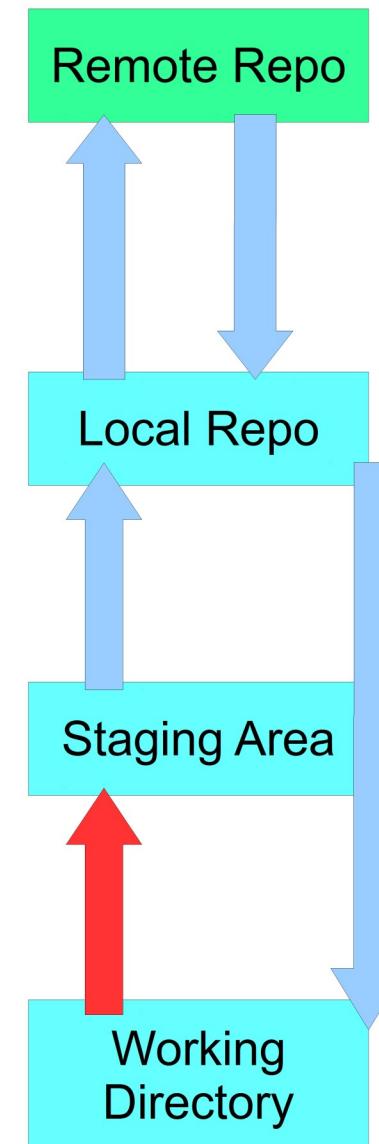
- Une fois Git installé, lancer Git Bash 
- Pour initialiser un répertoire comme dépôt local (*local repo*) :
  - *git init*
  - Cette commande va créer un répertoire `.git` dans votre répertoire courant. Il ne faut pas l'effacer → c'est l'historique des changements de votre *local repo*.
- Pour obtenir le code et l'historique des changements d'un dépôt en ligne (*remote repo*) :
  - *git clone https://userx@bitbucket.org/userx/test.git*

Cet exemple est pour un espace Git appelé “test” hébergé par Bitbucket par l’utilisateur “userx”. Pour utiliser Git à l’école, il faut qu’unE technicienNE vous crée un espace et vous envoie l’adresse de celui-ci.

- Il suffit d’entrer votre mot de passe et d’attendre.<sup>30</sup>

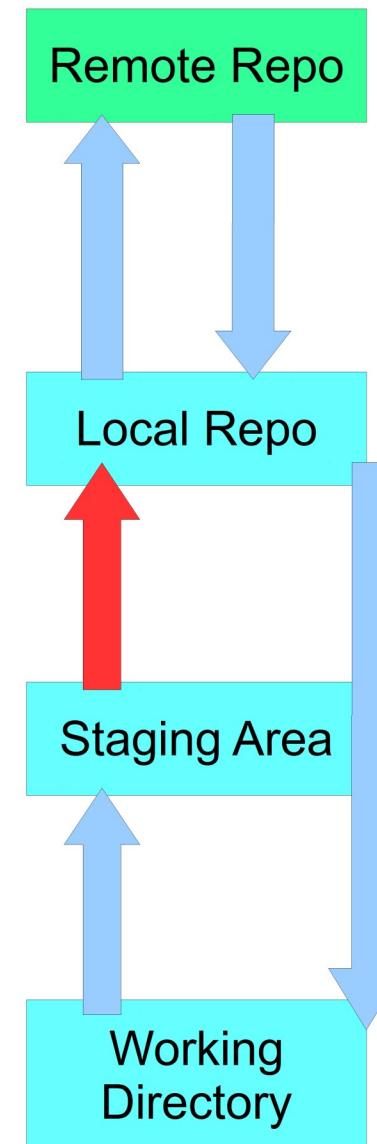
# Git : commandes principales

- *git add nomfichier.cpp*
  - Ajoute le fichier "nomfichier.cpp" au "staging area". Les changements faits à ce fichier seront ajoutés au prochain *git commit*.
- *git add .*
  - Ajoute tous les fichiers modifiés dans le répertoire courant et ses sous-répertoires au "staging area".
- *git status*
  - Décrit l'état du "working directory" et du "staging area" : les fichiers modifiés, les fichiers en "staging", les nouveaux fichiers, les fichiers à effacer ...



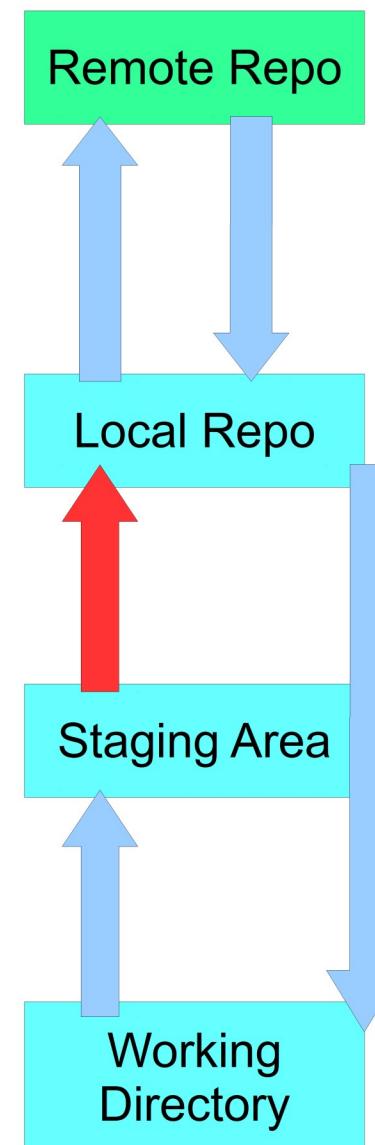
# Git : commandes principales

- *git commit -m 'message'*
  - Soumet l'ensemble des changements dans le "staging area" au "local repo".
  - Un *commit* devrait inclure des changements liés à un objectif particulier : résoudre un bogue, ajouter une fonctionnalité, etc.
    - Il est recommandé de faire des *commits* plus petits et plus fréquents.
  - Le 'message' du *commit* doit inclure une description claire permettant à vos collègues de comprendre les changements qui ont été effectués.



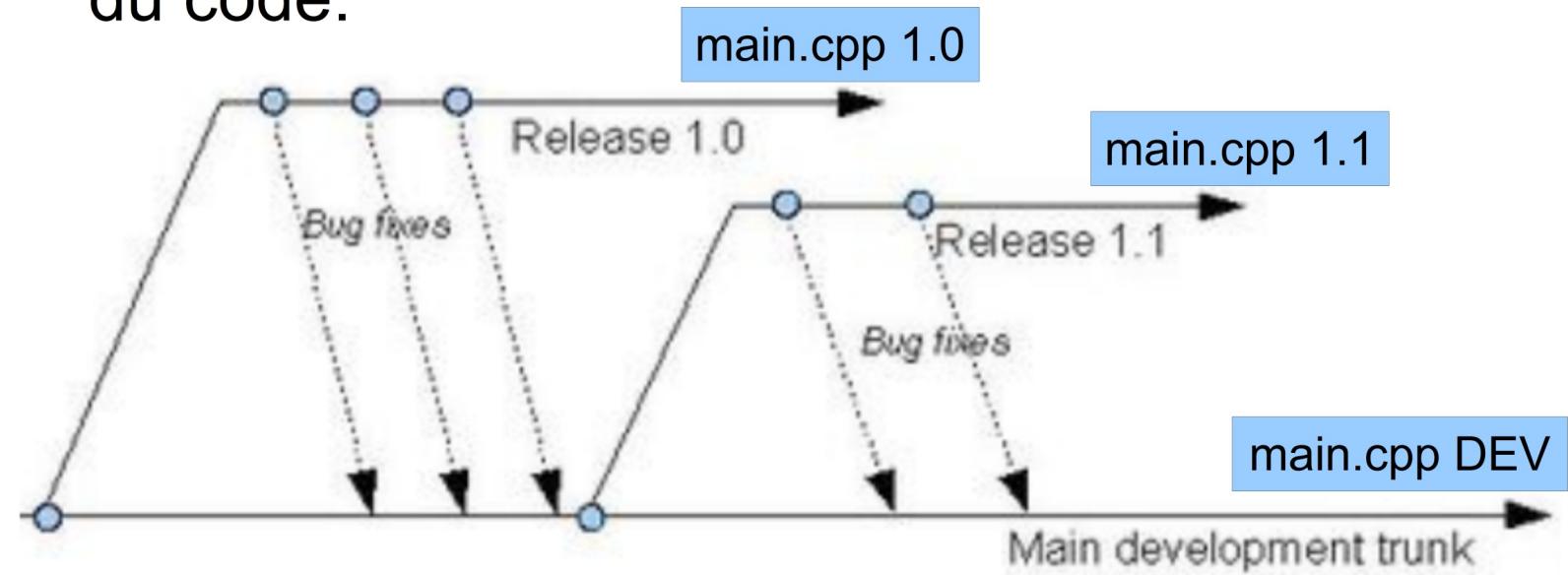
# Git : commandes principales

- Vous avez oublié de mettre un message à votre *commit* et vous vous retrouvez dans une fenêtre bizarre ? WTF!
  - Git a lancé pour vous un éditeur de texte en ligne de commande, généralement Vim (ou Vi).
  - Appuyez sur 'i' (pour *insertion*) pour entrer un message.
  - Une fois votre message fait, tapez sur 'Esc' et entrez ':wq' (pour *write* et *quit*) et tapez sur 'Enter'.



# Git et les branches

- Qu'est-ce qu'une branche ?
  - Les branches représentent les différentes versions du code.



- Git possède des branches par défaut :
  - HEAD pointe vers la branche où vous vous trouvez en ce moment.
  - La branche par défaut de Git s'appelle 'master'.
  - La commande `git status` permet de savoir sur quelle branche HEAD pointe.

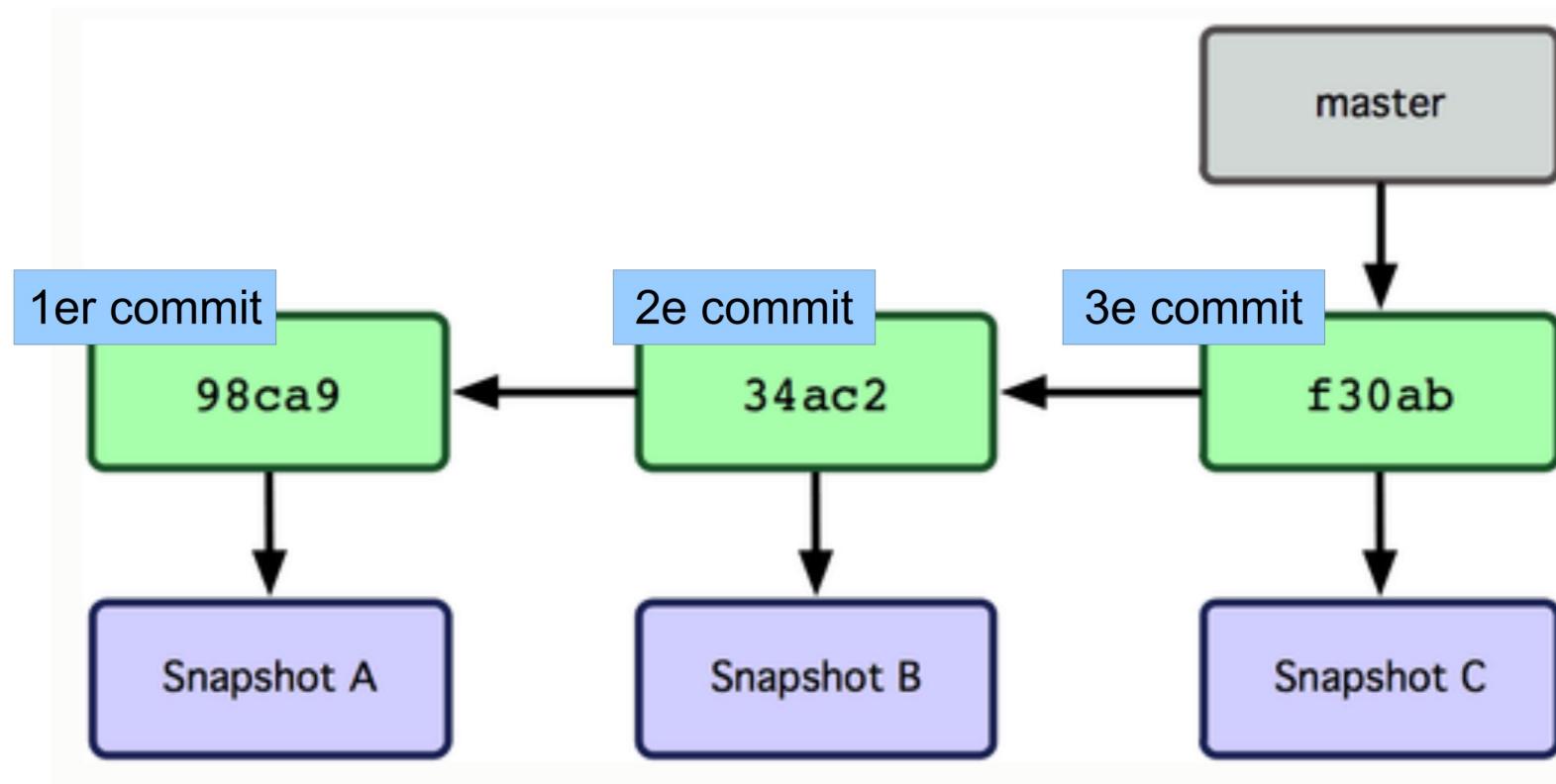
1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Git et les branches

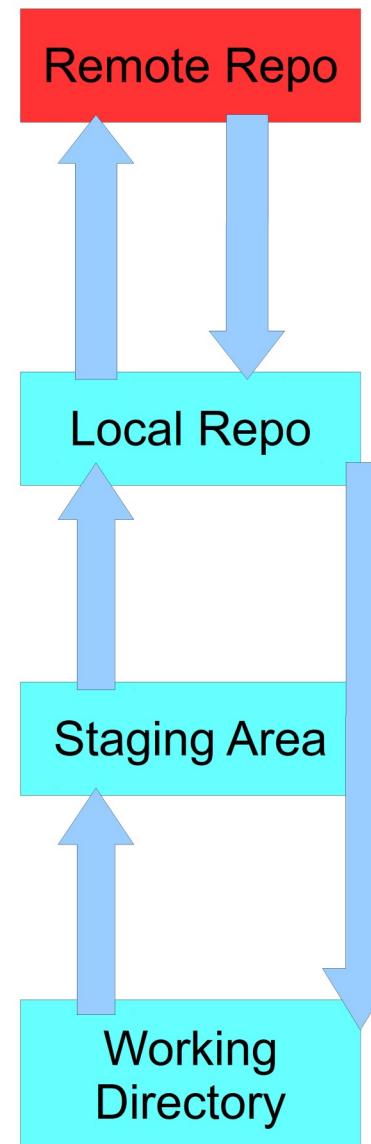
- À mesure que l'on fait des *commits* sur une branche, le pointeur de la branche est mis à jour pour pointer vers le dernier *commit* effectué.



→ Il est donc possible de faire pointer HEAD vers d'autres *commits* afin de voir le code de versions précédentes.

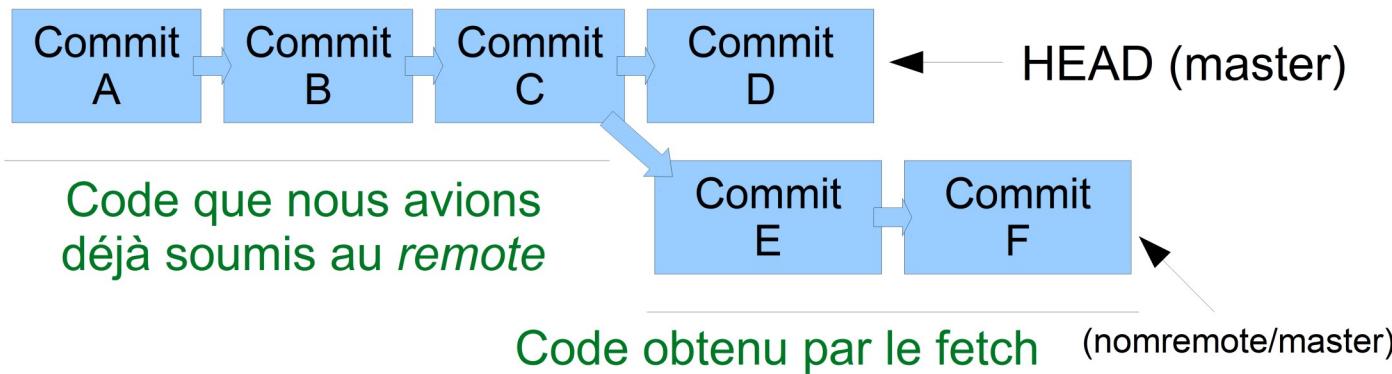
# Git : commandes principales

- Une fois notre code fonctionnel localement, il faut le partager aux autres !
- *git remote -v*
  - Détaille (-v → verbose) les *remote repo* auxquels il est possible d'obtenir ou de soumettre du code.
  - Par défaut, il n'y a rien, il faut donc ajouter un *remote repo*.
  - *git remote add nomremote https://userx@bitbucket.org/userx/test.git*
  - Par défaut, la majorité des gens remplace 'nomremote' par '**origin**'.
  - L'adresse internet devra être fournie par l'école dans votre cas.

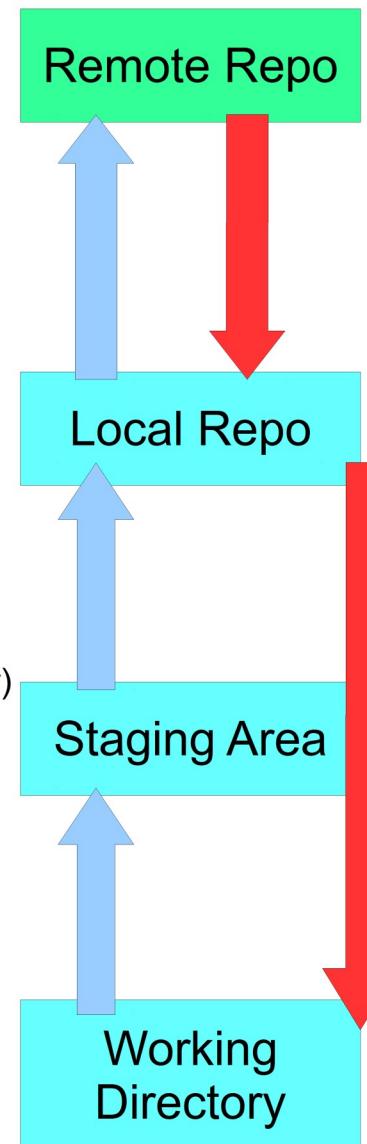


# Git : commandes principales

- *git fetch nomremote*
  - Prend les changements publiés sur le *remote repo* et les mets dans le *local repo*. Ex.:



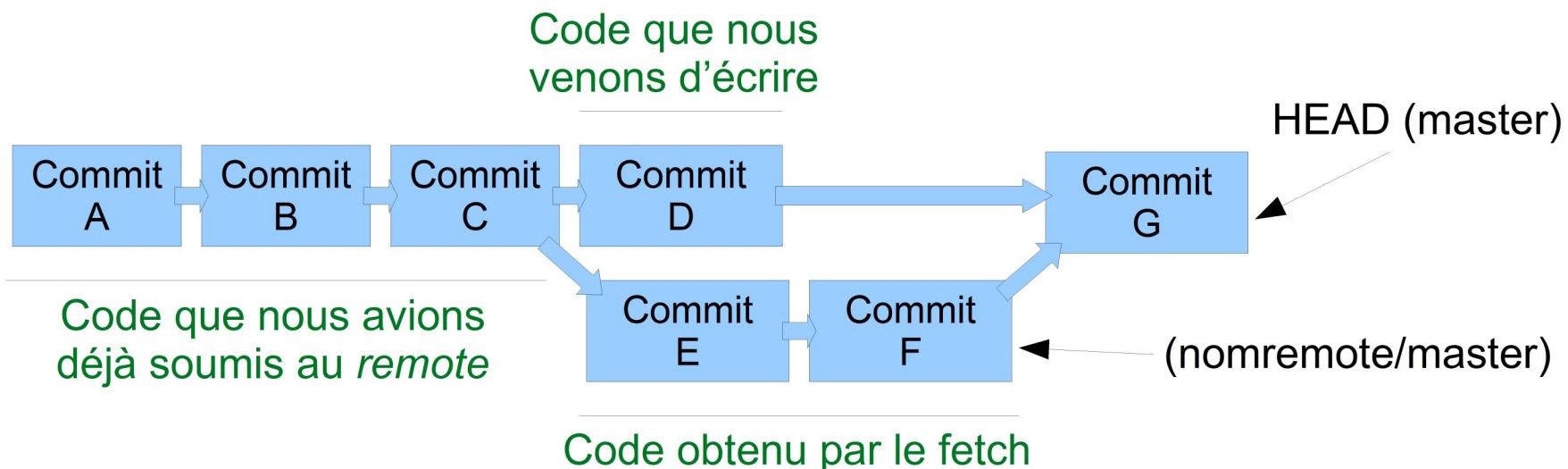
- On a donc des versions différentes du code ... il faut fusionner (*merge*).
- *git pull origin*
  - Effectue un *git fetch nomremote* suivi d'un *git merge*.



Voir : <https://stackoverflow.com/questions/44473483/what-does-git-fetch-exactly-do>

# Git : commandes principales

- *git merge nomremote/master (ou git pull)*
  - Crée un nouveau *commit* avec la fusion de la branche spécifiée avec la branche pointée par HEAD (dans notre cas, master).



- Dans la mesure où les modifications ne touchent pas le même fragment de code, la fusion se fait sans conflits.
  - Mais ça peut introduire des problèmes quand même ! → Ex.: fonction renommée.
- *git reflog* permet de voir quels *commits* se trouvent dans quelles branches et de comprendre où on en est !

# Git : résolution de conflit

- Dans le cas où les deux branches fusionnées touchent le même fragment de code, Git ne sait pas quoi faire.
  - Pour lister les conflits : *git diff*
- Le ou les fichiers affectés vont être marqués. Il revient à vous de choisir quelle version conserver.

```
int main(int argc, char **argv) {  
    int x = mesure();  
    std::cout << x;  
    return 0;  
}  
  
=<<<<< HEAD  
int calcul() {  
===== ←  
int mesure() {  
>>>>> 77976da35 ←  
    return 5;  
}
```

Marque de conflit.  
Début du code de  
votre version (HEAD)

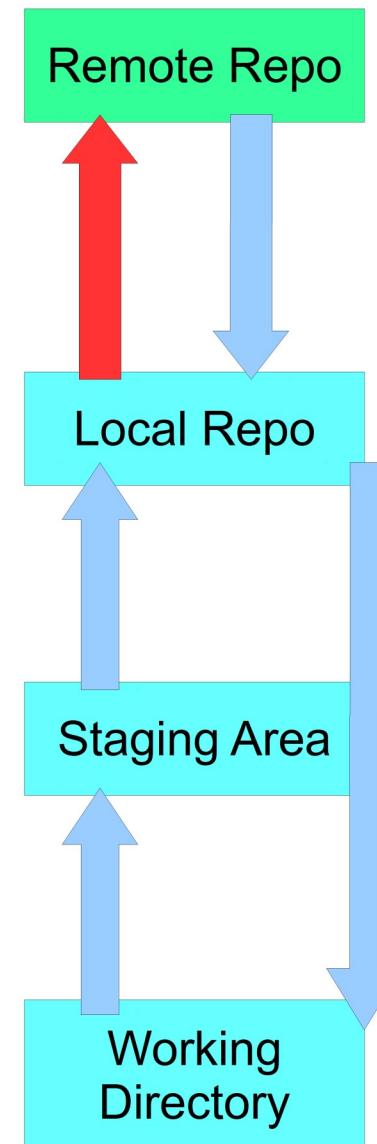
Séparateur des deux  
versions de code

Fin de l'autre version  
(commit 77976da35)

- Par la suite, vous pouvez faire un *commit* comme d'habitude.

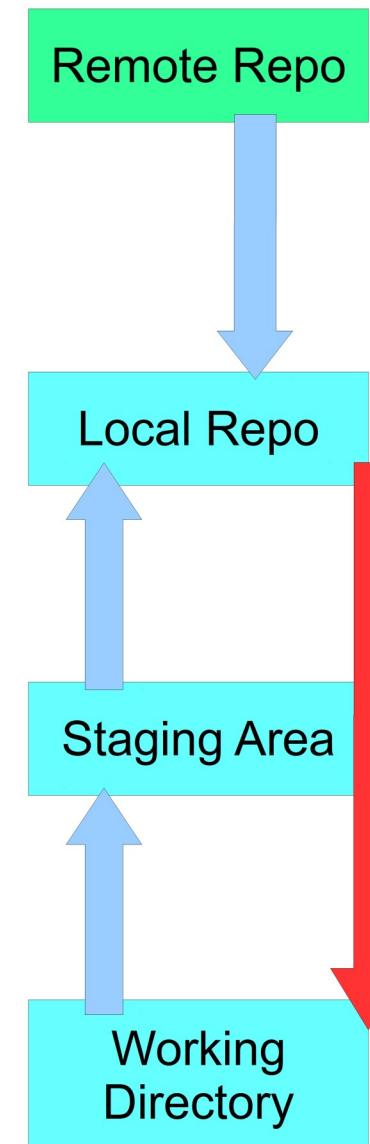
# Git : commandes principales

- *git push nomremote nombranche*
  - La branche 'nombranche' est généralement 'master'.
  - Cette commande publie tous les *commits* fait dans la branche spécifiée du *local repo* vers le *remote repo*.
  - Si d'autres personnes ont fait des *push* avant vous sur le *remote repo* → votre *local repo* n'est pas à jour avec le *remote repo* → votre *push* sera refusé.
    - Vous devez d'abord faire un *git pull* et résoudre les conflits si nécessaire.



# Git : commandes principales

- *git checkout nomcommit*
  - **DANGER !** Cette commande va effacer tout le code qui n'a pas été soumis (*committed*) au *staging area*.
  - Cette commande permet de voir (*to check out*) le code d'une autre branche.
  - Le 'nomcommit' est le code de hachage SHA-1 du commit (ex.: a1e8fb5). Les sept premiers caractères sont suffisants.  
Pour les obtenir :
    - *git reflog*
    - *git log --oneline*
- *git checkout master*
  - Pour revenir à l'espace de travail original.



1. Définition

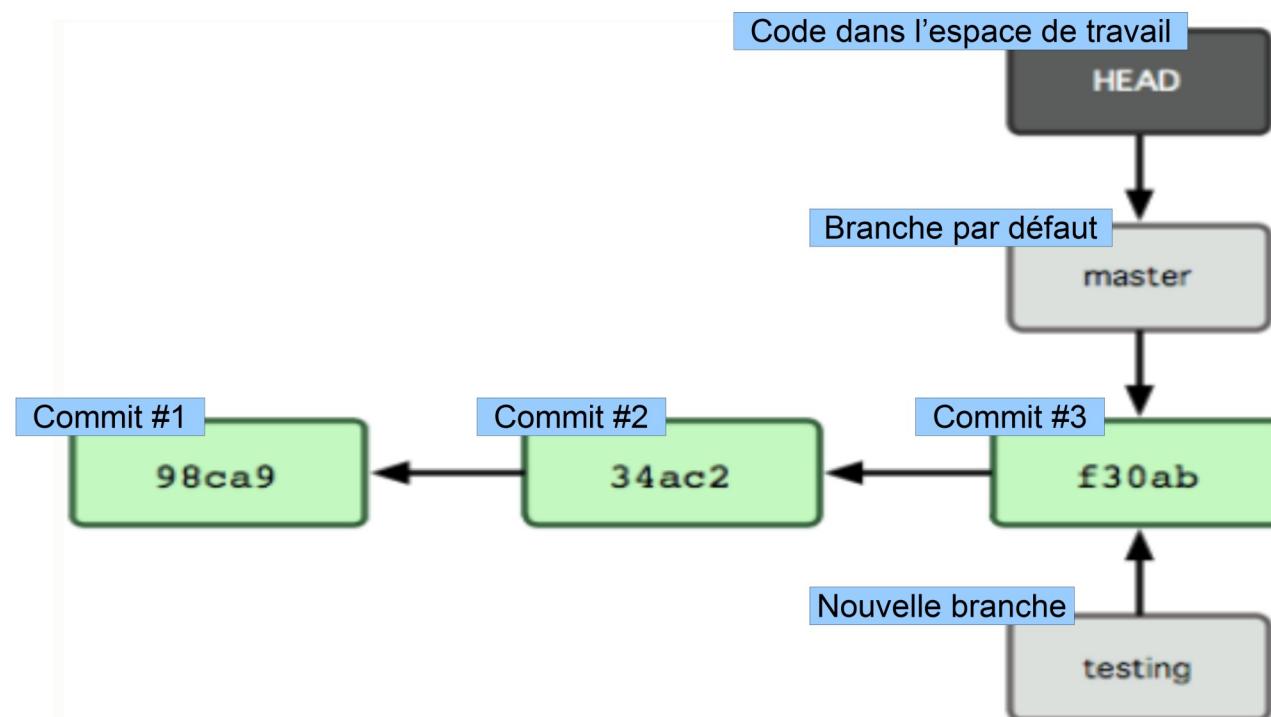
2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Git : naviguer dans les branches

- *git branch testing*

- Crée une nouvelle branche appelée "testing" pointant sur le dernier *commit* effectué.
- Cette commande ne change pas la position de HEAD. Si HEAD pointait sur "master" avant la commande, ce sera toujours le cas.



1. Définition

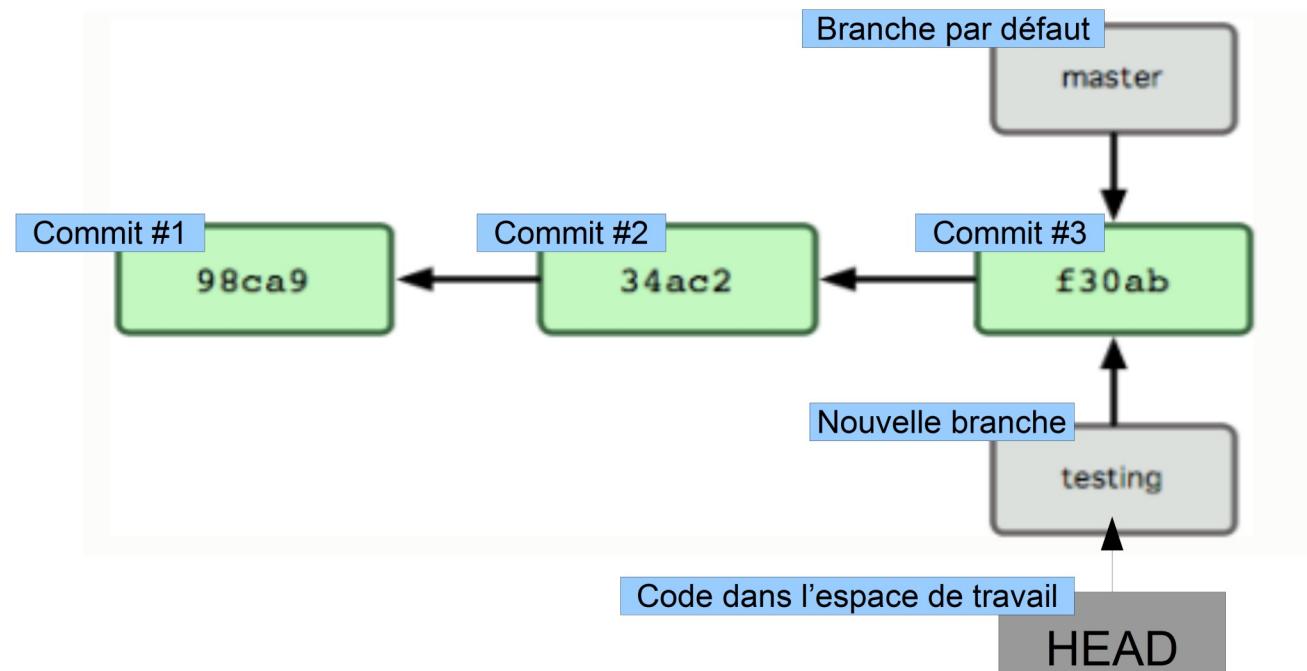
2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Git : naviguer dans les branches

- *git checkout testing*

- ATTENTION ! Vous perdrez tous les changements non-soumis (*committed*).
- Change le pointeur HEAD vers la branche "testing". Dans ce cas-ci, le code de "master" est le même que le code de "testing" parce que nous sommes au même *commit*.



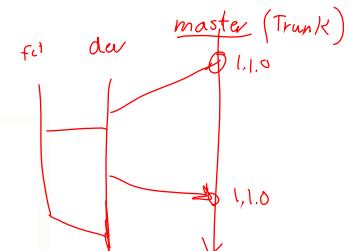
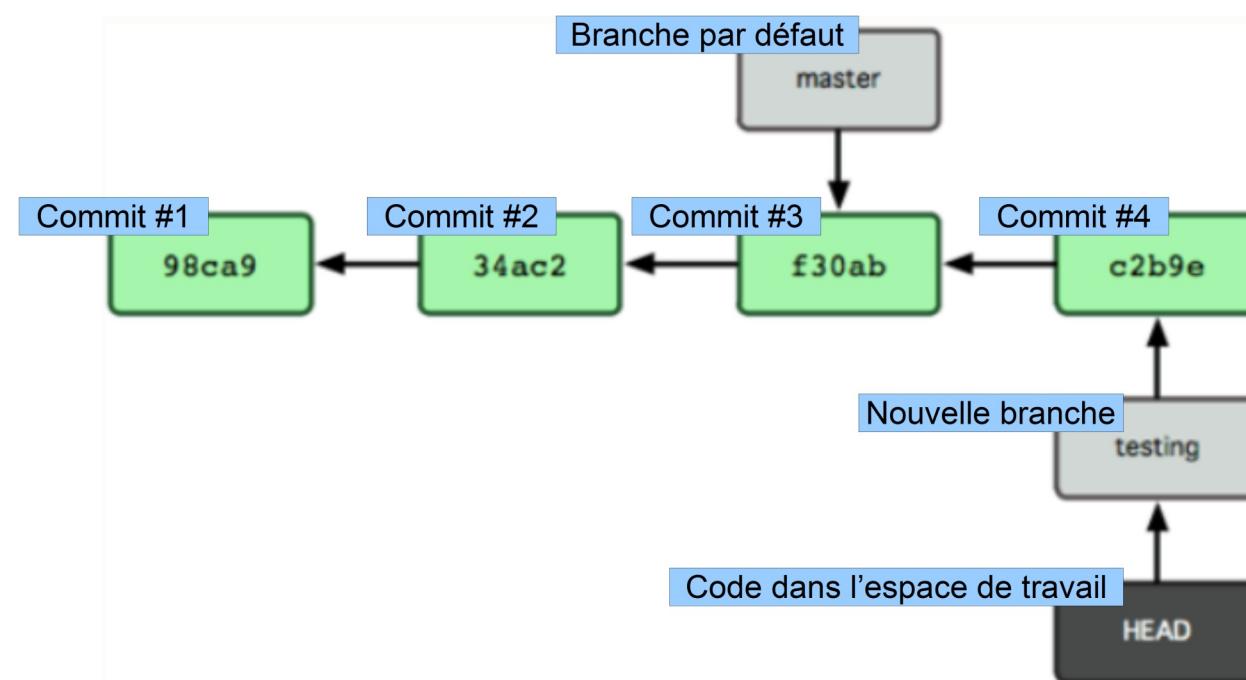
1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Git : naviguer dans les branches

- *vim main.cpp* ← On modifie le fichier « main.cpp »
- *git commit -a -m 'petite modification'*
  - On fait un nouveau *commit* sur la branche courante. Comme nous avons fait un *checkout* de "testing", c'est cette branche qui est affectée au lieu de "master".
  - Un *checkout* de "master" permettrait de revenir au code précédent.
  - Un *merge* de "testing" alors qu'on pointe sur "master" permettrait de fusionner le branche avec le code principal de "master" (souvent appelé "trunk").



# Git : fichier .gitignore

- Il est possible de dire à Git d'ignorer certains fichiers.
  - Utile afin d'éviter d'ajouter par accident des fichiers binaires (PDF, DOCX ...).
- Il faut mettre dans le répertoire de Git un fichier s'appelant .gitignore
  - Dans ce fichier, vous pouvez définir des types de fichiers que Git doit ignorer, séparer par des retours à la ligne. Ex.:
    - \*.pdf
    - guide.docx
    - carnet-\*.docx
  - Pour plus d'infos sur la syntaxe des fichiers .gitignore, voir  
<https://www.atlassian.com/git/tutorials/gitignore>

1. Définition

2. Principes de  
fonctionnement  
et outils

3. Commandes  
de base Git

# Git : cycle de travail normal

- Normalement, dans une séance de travail, vous devriez utiliser les commandes suivantes :
  - ***git pull*** : Avant de commencer afin d'obtenir la dernière version du code.
  - ***git status*** : Pour voir les changements que vous avez effectués.
  - ***git add*** : Pour ajouter les changements au prochain *commit*.
  - ***git commit*** : Régulièrement, avec des messages clairs des modifications faites.
  - ***git pull*** : Juste avant de faire un *push* afin de vous assurer que vous avez la dernière version de code.
  - ***git push*** : Pour publier vos *commits* à vos collègues.

1. Définition

2. Principes de fonctionnement et outils

3. Commandes de base Git

# Git : commandes principales

- Ce guide n'est qu'une vision très rapide des possibilités de Git.
- Il y a beaucoup d'autres commandes (certaines plus dangereuses) qui ne sont pas abordées ici.
  - Ex.: Hook → lancement de scripts après un *commit/push*.
  - Pour plus d'informations, ou si vous êtes coincéEs :
    - Stackoverflow présente d'excellentes réponses pour les questions les plus fréquentes avec Git.
      - <https://stackoverflow.com/questions/tagged/git?sort=frequent>
    - Atlassian présente des bons tutoriels sur l'utilisation des principales commandes de Git.
      - <https://www.atlassian.com/git/tutorials/setting-up-a-repository>



Git est stable, mais pas invulnérable ... il est recommandé de faire des *backups* réguliers de votre code.