SMP : Spécification et Modélisation de Programme

Gestion de version - Git

Myriam Servières

Ecole Centrale de Nantes
-
sur la base du cours de Guillaume Moreau

Objectifs

- Comprendre les bases d'un système de gestion de version
- Savoir utiliser git dans un projet
  - travailler dans un repository (dossier) local
  - se synchroniser sur un repository distant
  - avoir une idée des workflows collaboratifs (gestion des tâches et des acteurs)

Plan

Définition d'un système de gestion de version

D'après http://en.wikipedia.org/wiki/Revision_control

C'est la gestion des changements dans les documents, les programmes, les grands sites web et de façon plus générale dans les collections d'information.

Les changements sont généralement identifiés par des codes utilisant des lettres et des chiffres, appelés le numéro de révision. Par exemple, si un ensemble initial de fichiers sera appelé révision 1, après un premier ensemble de changements effectués, il sera appelé révision 2 et ainsi de suite.

Chaque révision est associée à un horodatage (timestamp) et à un auteur.

Les révisions pourront être comparées, restaurées et dans certains cas fusionnées.

Cas d'usage 1: conservation de l'historique

La vie de votre programme/document est enregistrée depuis son début

- A tout moment on peut revenir à la version précédente (si vous n'êtes pas contents de vos modifs)
- L'historique est accessible, on peut inspecter chaque révision (pour comprendre et résoudre les bugs)
  - quand a-t-elle été faite ?
  - qui l'a faite ?
  - quelle était la nature du changement ?
  - pourquoi ?
  - dans quel contexte ?
- tous les changements effacés restent accessibles dans l'historique

Cas d’usage 2 : travailler à plusieurs

Les outils de gestion de version vous aident à :

- partager une collection de fichiers avec votre équipe
- fusionner les changements effectués par les autres
- vous assurer que rien n’est effacé par erreur
- ~~savoir qui engu… quand quelque chose ne marche pas~~

Cas d'usage 3 : les branches

Il arrive qu'on ait de multiples versions d'un programme, matérialisées comme des branches.
Par exemple :

- une branche principale
- une branche de maintenance (pour corriger les bugs dans les anciennes versions)
- une branche de développement (pour effectuer des changements de fond)
- une branche de release (pour figer le code avant une livraison)

Les outils de gestion de version vont vous aider à :

- gérer plusieurs branches de façon concurrente
- fusionner (tout ou partie) des branches

Cas d’usage 4 : travailler avec des contributeurs externes

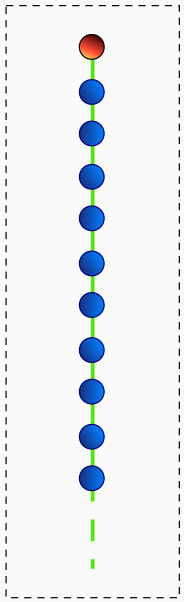
Les outils de gestion de version vous aident à travailler avec des développeurs externes :

- cela leur donne de la visibilité sur ce qui se passe dans le projet
- cela les aide à soumettre des modifications (patches) et vous aide à intégrer leurs patchess
- permet aux développeurs de créer leur propre version du projet, puis à revenir vers la branche principale

Cas d'usage 5 : passage à l'échelle

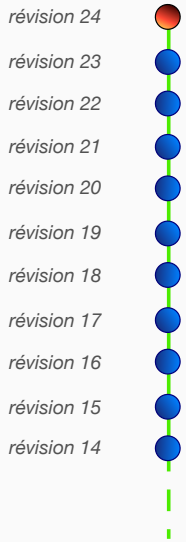
- Quelques éléments sur le noyau de Linux
  - Linus Torvald a créé `git` pour gérer le flot de révisions !
  - environ 10000 changements dans chaque nouvelle version (tous les 2-3 mois)
  - plus de 1000 contributeurs

Illustrations : le repository



Il contient l'historique complet du projet (toutes les révisions depuis le début)

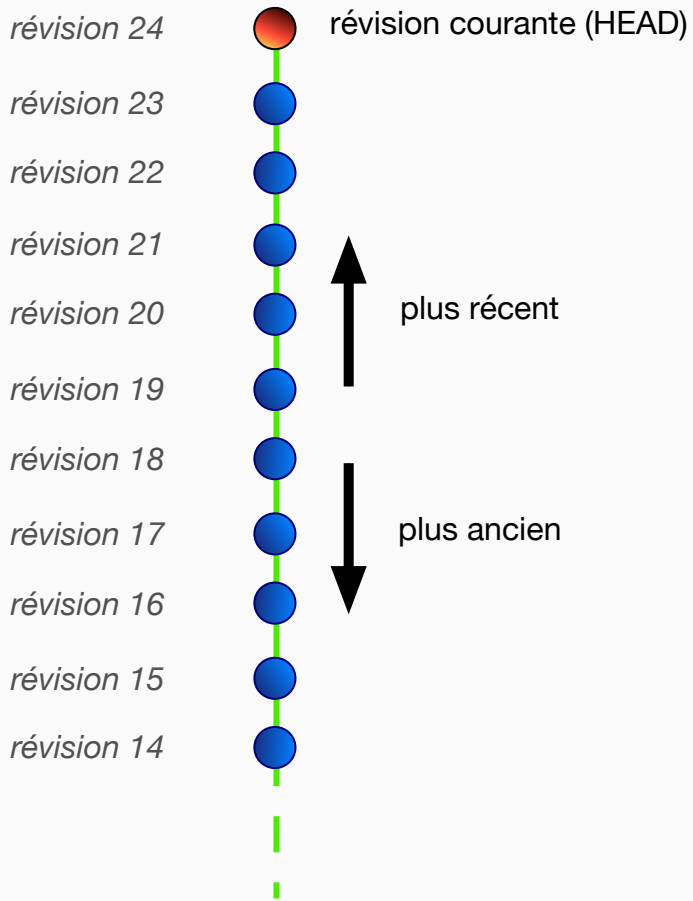
Illustrations : les révisions



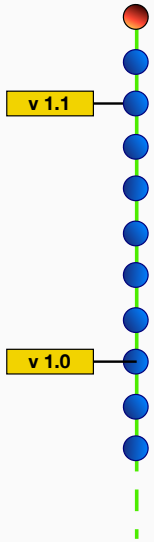
Chaque révision

- introduit des changements par rapport à la révision précédente
- a un auteur identifié
- est horodatée
- contient un message qui décrit les changements

Illustrations : HEAD



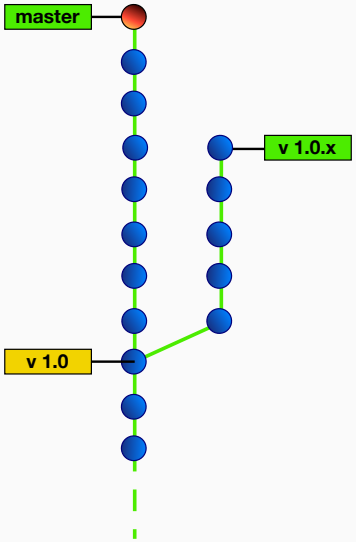
Illustrations : les tags (étiquettes)



Un tag

- identifie une révision particulière
- par exemple les livraisons du logiciel

Illustrations : les branches



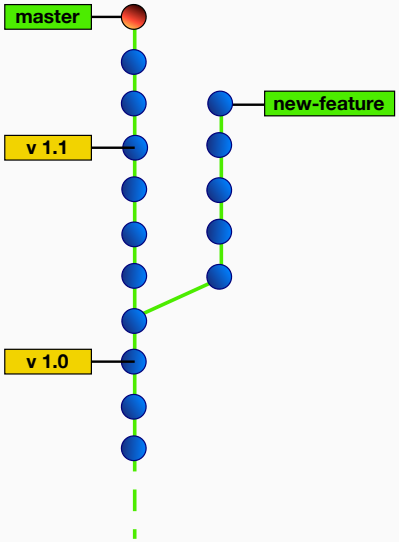
Une branche

- est une variante de la collection de fichiers
- évolue indépendamment des autres branches
- des opérations de fusion sont possibles

Exemples :

- `master` : branche principale
- `v1.0.x` : branche de maintenance de la version 1.0

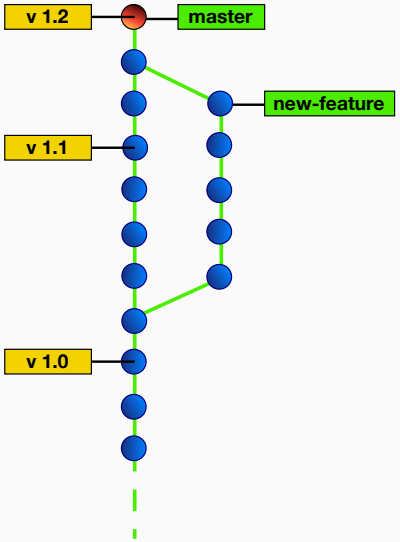
Illustrations : utilisation des branches



La branche `new-feature` permet de

- développer une nouvelle fonctionnalité
  - intrusive
- conserver le développement normal
  - sans impact sur le processus

Illustrations : fusion (merge)



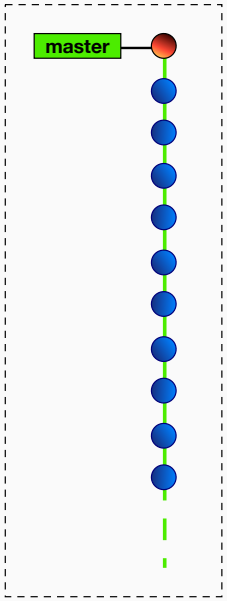
La branche `new-feature` est fusionnée à la branche `master`

- quand elle est prête
- tous les changements de la branche sont importés

Typologie de la gestion de versions

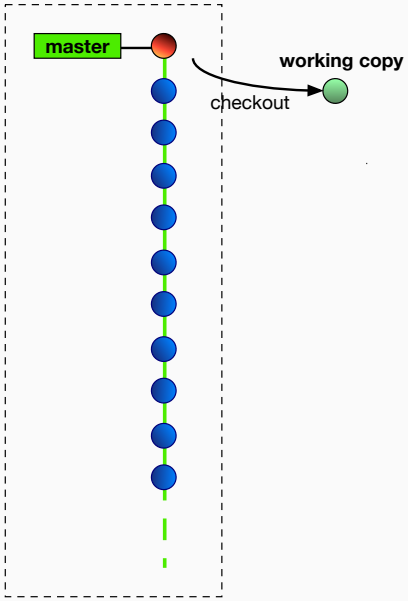
- Architecture
  - **centralisée** : tout le monde travaille sur le même référentiel
  - **décentralisée** : chacun a son propre référentiel
- Modèle de concurrence
  - **verrou avant édition** : exclusion mutuelle
  - **fusion après édition** : gestion des conflits à prévoir
- Gestion de l'historique
  - **arbre** : pas de gestion des fusions
  - **DAG** : *directed acyclic graph* - graphe acyclique orienté
- Atomicité
  - niveau **fichier**
  - niveau **arborescence**

Interactions avec le repository



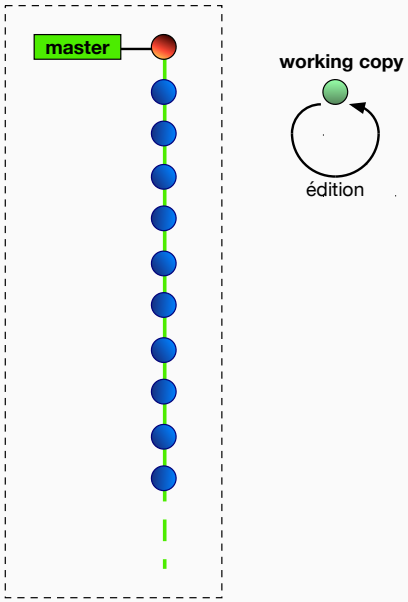
- un référentiel (repository) est une entité opaque, pas question de l'éditer directement
- On doit extraire sa copie de travail (working copy) du repository

Interactions avec le repository : checkout



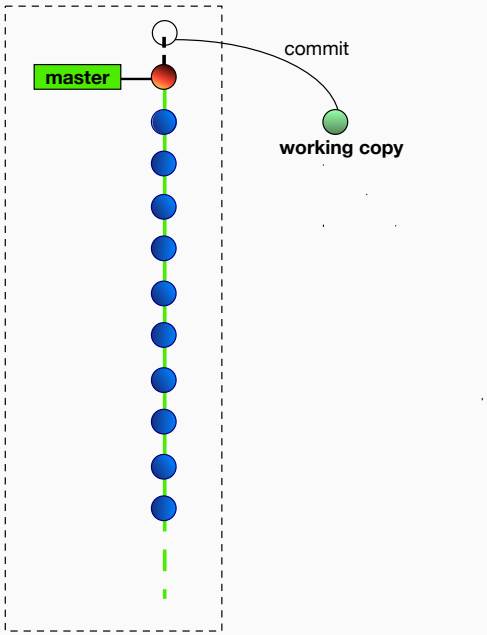
- la commande **checkout** extrait une révision (le plus souvent la dernière) du référentiel

Interactions avec le repository : édition



- la copie de travail est stockée sur le disque local
- elle peut être modifiée, compilée...

Interactions avec le repository : commit



- la commande `commit` crée une nouvelle révision à partir de la copie de travail
- le processus édition puis `commit` peut se répéter plusieurs fois

Que stocke-t-on dans le repository ?

- Tous les fichiers qui ne sont pas générés par un outil
  - des fichiers sources (`.c .cpp .java .tex .sql`)
  - des scripts de build, des fichiers de projets (`Makefile CMakefile.txt ant.xml`)
  - des fichiers de documentation (`.txt README`)
  - des fichiers de ressources (images, audio…)
- Il ne faut pas stocker les fichiers générés par un outil (sous peine de gérer des conflits inutiles)
  - `.o .obj .a .lib .so .dll .class .jar .exe`…
  - les scripts de build quand ils sont générés par un autre outil
    - exemple : `cmake` qui génère des `Makefile`

Quelques bonnes pratiques de `commit`

- Faire des `commit` souvent
- Faire des `commit` relatifs à des changements indépendants dans des révisions différentes
- Ecrire des messages de `commit`
  - c'est obligatoire !
  - décrire le pourquoi du changement plutôt que le changement lui-même

Un peu d'histoire de la gestion de version centralisée

- 1ère génération (mono fichier, utilisation locale, exclusion mutuelle)
  - 1972 : SCCS
  - 1982 : RCS
  - 1985 : PVCS
- 2ème génération (multi-fichiers, client-serveur, fusion après commit)
  - 1986 : CVS
  - 1992 : Rational ClearCase
  - 1994 : Visual SourceSafe
- 3ème génération (2ème + atomicité)
  - 1995 : Perforce
  - 2000 : **subversion**
  - plein d'autres...

Gestion de version décentralisée

- les prémisses au début des années 2000
  - Bitkeeper, GNU Arch
- les outils actuels (2005)
  - Bazaar
  - Mercurial
  - git
- tendance à l'intégration
  - gestion de version + gestion des bugs et du projet
  - github, gitlab, bitbucket
  - fossil

Histoire

- Avant 2005 : les sources de Linux étaient gérées avec Bitkeeper, outil propriétaire
- Avril 2005 : révocation de la licence de libre utilisation
- Rien d'autre dans le paysage en termes de maturité
  - Linus Torvald a commencé à développer `git`
- Juin 2005 : première version de Linux gérée avec `git`
- Décembre 2005 : sortie de `git`1.0

Hébergement : GitHub, GitLab (I)

GitHub

Software developer

github.com

GitHub, Inc. is a platform and cloud-based service for software development and version control, allowing developers to store and manage their code. Wikipedia

Parent: Microsoft

Founders: Tom Preston-Werner, Chris Wanstrath, Scott Chacon, P. J. Hyett

Subsidiary: npm, Inc.

Founded: 2008, San Francisco, California, United States

CEO: Thomas Dohmke (Nov 15, 2021–)

Headquarters: San Francisco, California, United States

Employees: 2,500

GitLab

Software

about.gitlab.com

GitLab Inc. is an open-core company that operates GitLab, a DevOps software package that can develop, secure, and operate software. The open source software project was created by Ukrainian developer Dmytro Zaporozhets and Dutch developer Sytse Sijbrandij. Wikipedia

Written in: Ruby, Go, Vue.js, JavaScript

Launch date: 2014

Written in: JavaScript, Go, Ruby

Available in: English Language

Employees: 1,630 (January 2022)

Founder(s): Dmytro Zaporozhets; Sytse "Sid" Sijbrandij

Hébergement : GitHub, GitLab (II)

- Fonctionnent sur des serveurs Linux; proposent des outils-tiers
- Proposent des CLI (Command Line Interface); dépôts basés sur le web
- Proposent des abonnements gratuits

[GitHub]

- Une partie du code open source, mais son intégralité non
- GitHub est acquis par Microsoft en 2018
- Enquête de la société d'outils de programmation JetBrains : 77% d'utilisateurs
- Localisation de serveurs : US, Artic Code Vault

[GitLab]

- Open source et gratuite : édition Community
- Payant : édition Enterprise (support et plus des fonctionalitées)
- Enquête de la société d'outils de programmation JetBrains : 40% d'utilisateurs
- Localisation de serveurs : US east-region

Plan

Utilisation de git - Installation

- Linux
  - Installé par défaut
  - Si non installée `apt-get install git-all` (sous Ubuntu - Debian)
  - Pour d'autres versions de Unix `http://git-scm.com/download/linux`
- Mac
  - Avec Xcode Command Line Tools.
  - Sur Mavericks (10.9) ou postérieur, vous pouvez simplement essayer de lancer git dans le terminal la première fois. S'il n'est pas déjà installé, il vous demandera de le faire.
- Utilisation avec interface graphique
  - Turtoise Git (Windows)
  - plugins git (Eclipse, Netbeans, Atom...)

Paramétrage à la première utilisation de Git

Identité

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

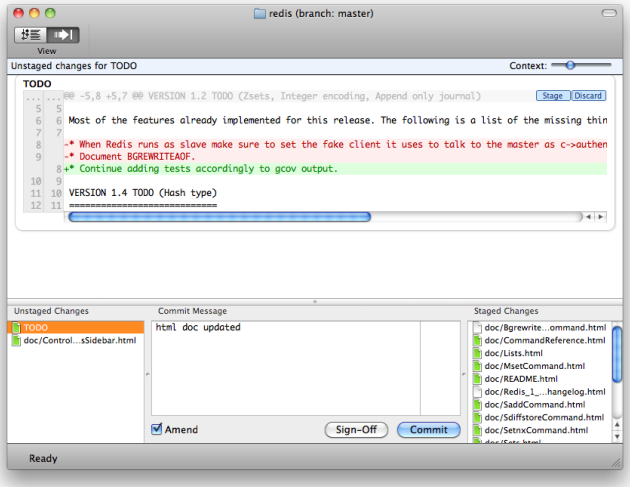
Choix de l'éditeur de texte

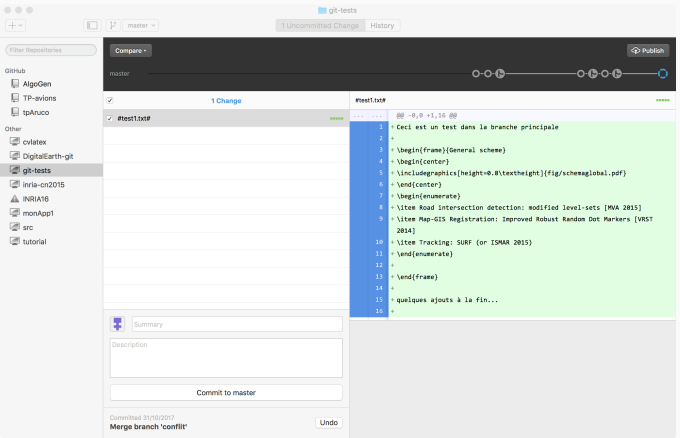
```
$ git config --global core.editor emacs
```

Vérification de vos paramètres

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
```

Utilisation de git - ligne de commande





- la ligne de commande (installé par défaut avec les autres outils ou certains IDE)
- Turtoise Git (Windows)
- Github (toutes plate-formes)
- plugins git (Eclipse, Netbeans, Atom...)

Création d'un référentiel local

`git init myrepository`

Cette commande crée un répertoire myrepository, le repository lui même est contenu dans myrepository/.git, une copie de travail (initialement vide) est créée dans myrepository/

```
(base) mservier@mcl-21-001-1 code % git init helloworld
Initialized empty Git repository in /Users/mservier/Documents/pedagogie/Apprentissage/appc-co
(base) mservier@mcl-21-001-1 code % ls -a helloworld
.    ..    .git
(base) mservier@mcl-21-001-1 code % ls -a helloworld/.git
.           HEAD        config      hooks       objects
..          branches    description info        refs
```

Premiers commits

```
git add file
git commit [-m message]
```

Avec `git`, il y a deux opérations :

- add
- commit

Par défaut les commits sont effectués dans la branche master

Exemple

```
(base) mservier@mcl-21-001-1 code % cd helloworld
(base) mservier@mcl-21-001-1 helloworld % vi hello.txt
(base) mservier@mcl-21-001-1 helloworld % git add hello.txt
(base) mservier@mcl-21-001-1 helloworld % git commit -m "ajout du fichier hello.txt"
[master (root-commit) 0c85b0b] ajout du fichier hello.txt
1 file changed, 2 insertions(+)
create mode 100644 hello.txt
(base) mservier@mcl-21-001-1 helloworld %
```

L'index (staging area)

Les systèmes de gestion de version utilisent deux espaces :

- le référentiel
  - toute l'histoire de votre projet
- la copie de travail
  - les fichiers que vous éditez et qui feront partie du prochain commit

`git` introduit un espace intermédiaire : la **staging area** parfois appelée **index** qui stocke les fichiers en vue du prochain commit[1] :

- `git add` files copie les fichiers vers l'index
- `git commit` effectue le commit du contenu de l'index

[1] la motivation et l'utilisation sortent du cadre de ce cours

Exemple : modification d'un fichier

```
(base) mservier@mcl-21-001-1 helloworld % echo 'blablabla' >> hello.txt
(base) mservier@mcl-21-001-1 helloworld % git commit
On branch master
Changes not staged for commit:
modified:   hello.txt

no changes added to commit
```

`git` se plaint qu'il y a rien de changé. En réalité, il faut faire

```
(base) mservier@mcl-21-001-1 helloworld % git add hello.txt
(base) mservier@mcl-21-001-1 helloworld % git commit -m "quelques changements"
[master 8e252fc] quelques changements
1 file changed, 1 insertion(+)
```

Illustration avec github

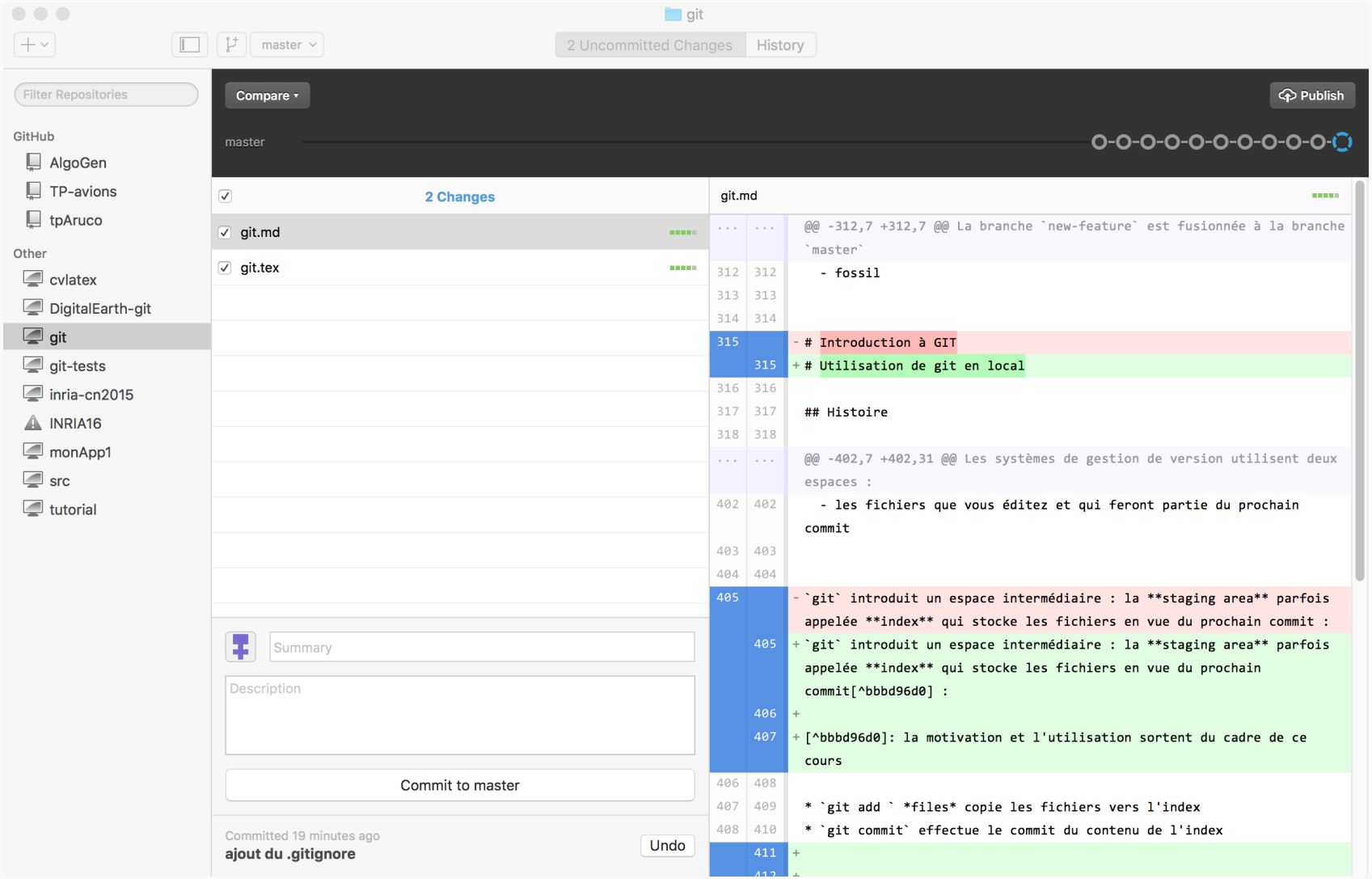
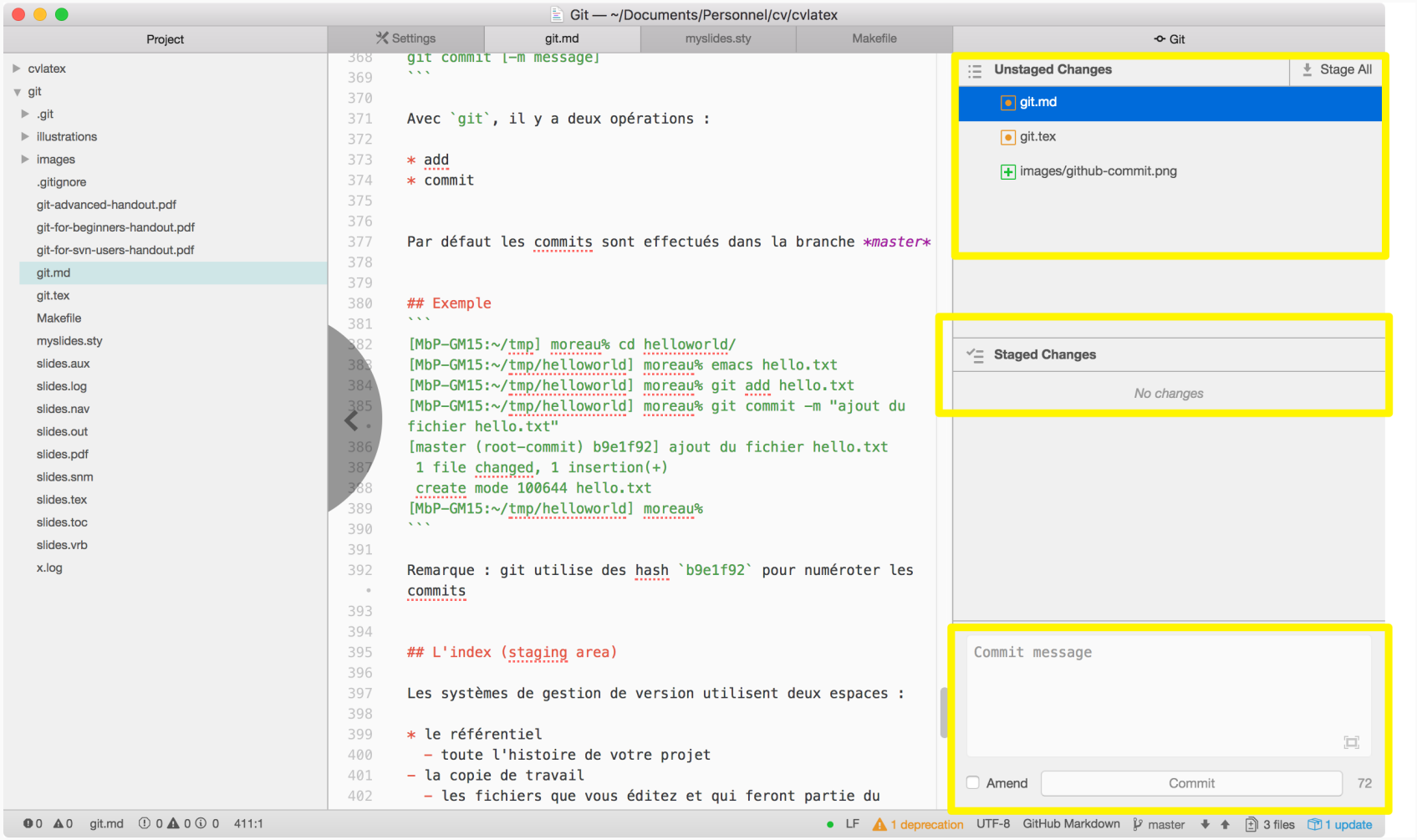


Illustration avec Atom



Suppression de fichiers

`git rm file`
`git commit`

supprime un fichier de la copie de travail et du référentiel

```
(base) mservier@mcl-21-001-1 helloworld % git rm hello.txt
rm 'hello.txt'
(base) mservier@mcl-21-001-1 helloworld % git commit -m "plus utile"
[master b328f30] plus utile
1 file changed, 3 deletions(-)
delete mode 100644 hello.txt
```

Différences entre versions

`git diff [rev_a [rev _b] ]`

montre les différences entre 2 révisions `rev_a` et `rev_b`

Attention, par défaut :

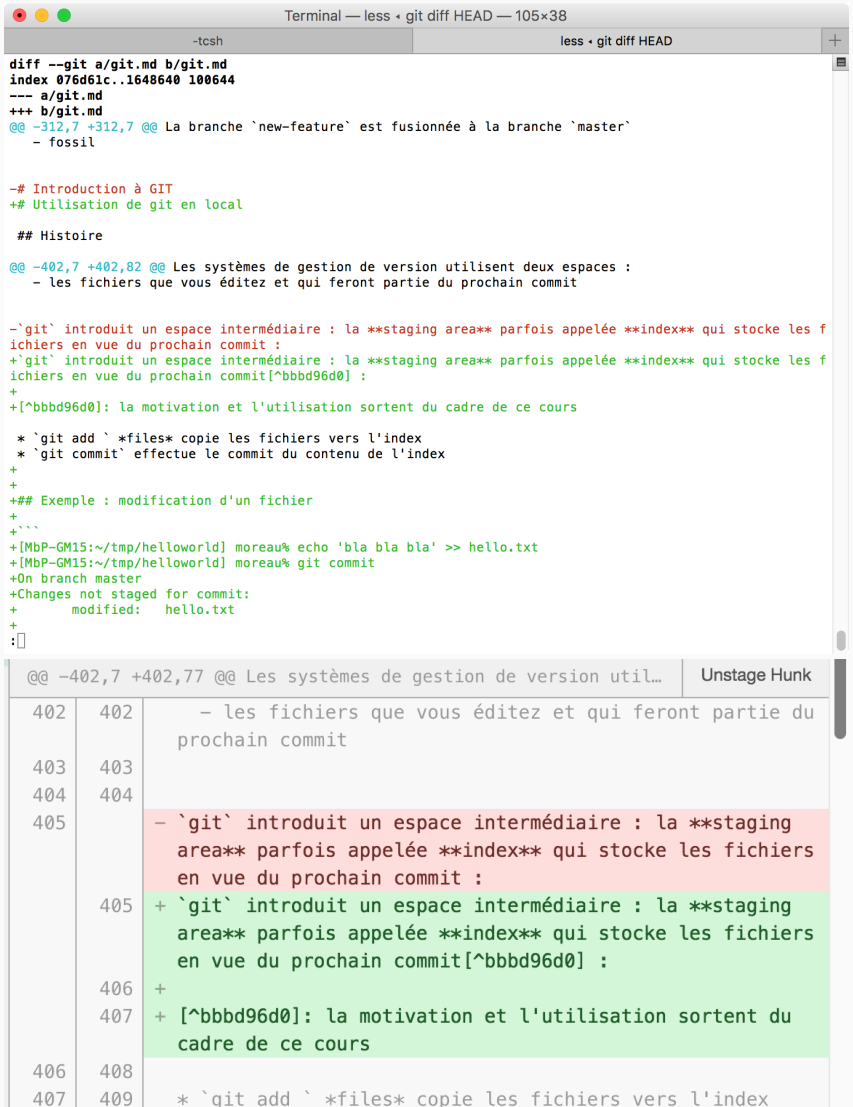
- `rev_a` est l'index
- `rev_b` est la copie de travail

Si on veut savoir où l'on en est par rapport au référentiel :

```
git diff HEAD
git diff master
```

Visualisation des différences entre versions

Les outils intégrés prennent ici tout leur sens : exemples avec la ligne de commande et Atom



Revenir en arrière

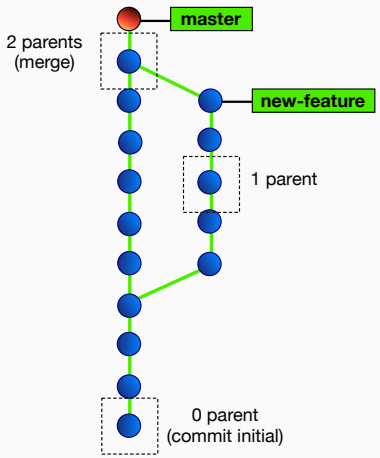
- `git reset` annule les changements dans l'index
- `git reset --hard` annule les changements dans l'index **et** dans la copie de travail
- `git checkout --`path restaure un fichier ou un répertoire tel qu'il apparait dans l'index (revient de la copie de travail à l'état au dernier `git add`)

Autres commandes locales

- `git status` : donne l'état de l'index et de la copie de travail (la liste des modifications prises en compte ou non)
- `git show` : donne les détail d'un commit
- `git log` : donne l'historique
- `git mv` : déplace ou renomme un fichier/répertoire
- `git tag` : création et suppression des tags

Plan

Comment git gère-t-il son historique ?



Chaque objet de **commit** possède sa propre liste de **commit** parents :

- 0 parent : commit initial
- 1 parent : commit ordinaire
- 2+ parents : résultat d'une fusion (merge) de branches

d'où cette notion de graphe acyclique orienté.

en réalité, une branche est juste un pointeur sur le dernier commit.

Création d'une nouvelle branche

`git checkout -b nouvelle_branche  [ depart ]`

- nouvelle_branche est le nom de la nouvelle branche
- depart est le point de départ (commit id, tag...). Par défaut `git`utilise l'emplacement courant

```
(base) mservier@mcl-21-001-1 helloworld % git status
On branch master
nothing to commit, working tree clean
(base) mservier@mcl-21-001-1 helloworld % git checkout -b dev
Switched to a new branch 'dev'
(base) mservier@mcl-21-001-1 helloworld % git status
On branch dev
nothing to commit, working tree clean
```

Passer d'une branche à l'autre

`git checkout [-m]` branch_name

```
(base) mservier@mcl-21-001-1 helloworld % git status
On branch dev
nothing to commit, working tree clean
(base) mservier@mcl-21-001-1 helloworld % git checkout master
Switched to branch 'master'
```

La commande ne fonctionne que si la copie de travail est propre. L'option `-m` permet de demander une fusion vers la branche de destination.

Fusion de branches

`git merge` other_branch

fusionne les changements effectués dans other_branch vers la branche courante.

Attention : cette commande n'est pas symétrique !

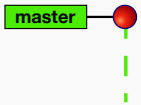
```
(base) mservier@mcl-21-001-1 helloworld % git checkout dev
Switched to branch 'dev'
(base) mservier@mcl-21-001-1 helloworld % vi toto.txt
(base) mservier@mcl-21-001-1 helloworld % git add toto.txt
(base) mservier@mcl-21-001-1 helloworld % git commit -m "ajout de toto.txt"
(base) mservier@mcl-21-001-1 helloworld % git checkout master
Switched to branch 'master'
(base) mservier@mcl-21-001-1 helloworld % git merge dev
Updating b328f30..36ad28d
Fast-forward
toto.txt | 1 +
1 file changed, 1 insertion(+)
create mode 100644 toto.txt
```

Quelques remarques sur la fusion

- Le résultat d'un `git merge` fait immédiatement l'objet d'un commit (sauf en cas de conflit)
- Le nouvel objet de commit a **deux parents**
  - l'historique de la fusion est donc enregistré
- `git merge` s'applique uniquement aux changements effectués dans l'autre branche depuis le dernier ancêtre commun
  - s'il y a déjà eu un `merge`, seuls les changements depuis le dernier `merge` seront pris en compte

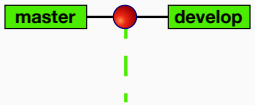
Exemple (1/12)

Situation initiale



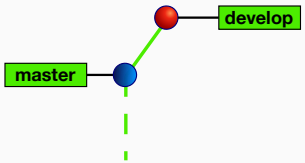
Exemple (2/12)

```
git checkout -b develop
```



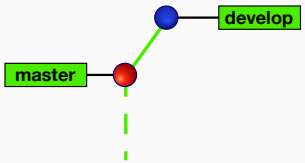
Exemple (3/12)

```
git commit
```



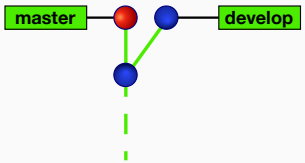
Exemple (4/12)

```
git checkout master
```



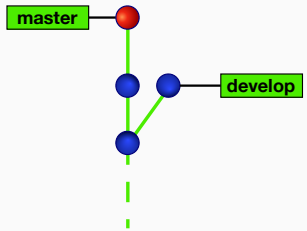
Exemple (5/12)

```
git commit
```



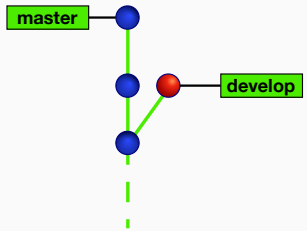
Exemple (6/12)

```
git commit
```



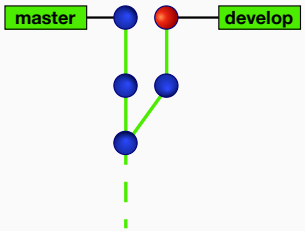
Exemple (7/12)

```
git checkout develop
```

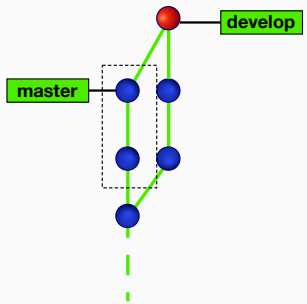


Exemple (8/12)

```
git commit
```



Exemple (9/12)

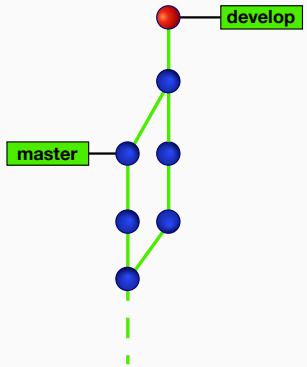


`git merge master`

ici on demande bien la fusion de la branche master vers la branche develop

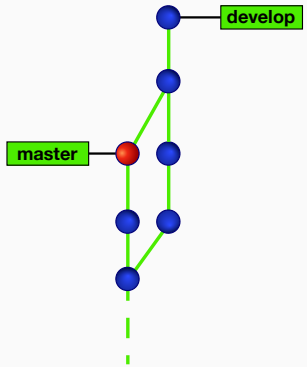
Exemple (10/12)

```
git commit
```



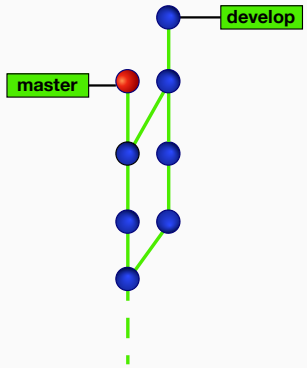
Exemple (11/12)

```
git checkout master
```



Exemple (12/12)

```
git commit
```

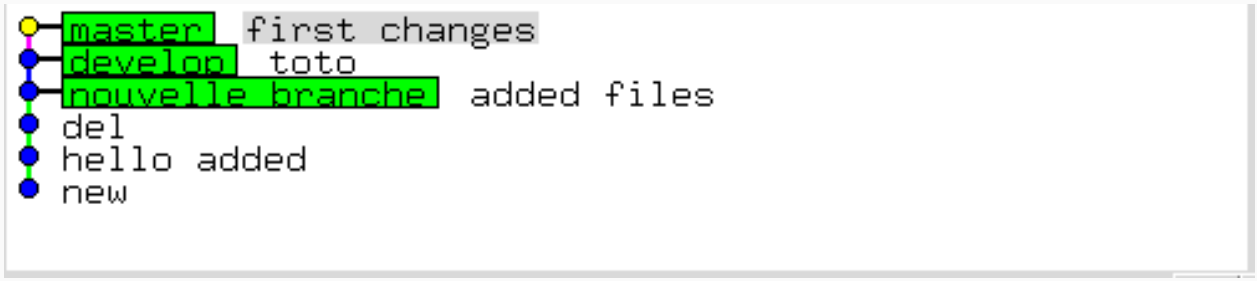


Comment git fusionne-t-il les fichiers ?

Si le même fichier est modifié dans les deux branches, git doit fusionner deux variantes :

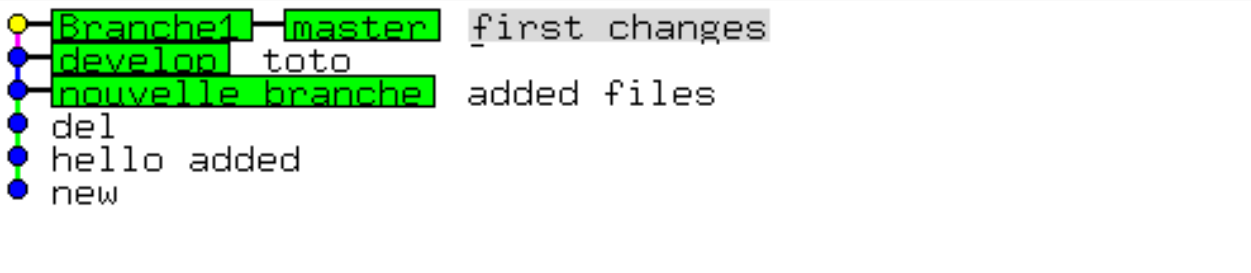
- les **fichiers texte** sont fusionnés ligne par ligne
  - les lignes modifiées dans une seule branche sont automatiquement fusionnées
  - si une ligne est modifiée dans deux branches, `git` rapporte un conflit
  - les zones de conflits sont repérées par `<<<<<<<` et `>>>>>>>`
- les **fichiers binaires** génèrent toujours un conflit
- on résout alors les conflits manuellement (en éditant les fichiers)
- `git` refusera le commit tant que tous les conflits ne sont pas réglés

L'histoire d'un conflit sur GIT (1)




```
$ git checkout master
$ vi victor_hugo.txt
$ git add victor_hugo.txt
$ git commit -m ``first changes''
[master 2fee11f] first changes
1 file changed, 3 insertions(+)
create mode 100644 victor_hugo.txt
```

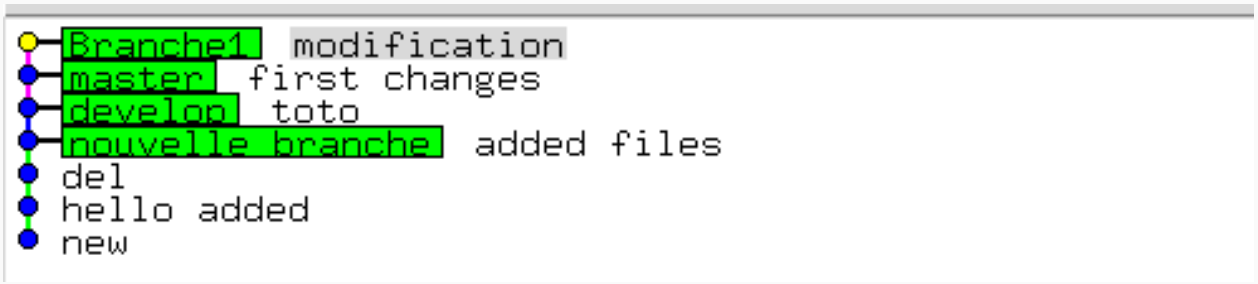
L’histoire d’un conflit sur GIT (2)




```
$ git checkout -b Branche1
Switched to a new branch 'Branche1'
```

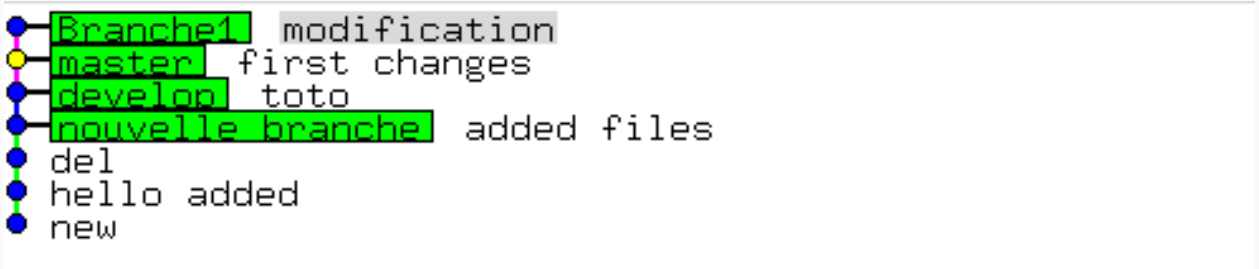
L’histoire d’un conflit sur GIT (3)

On modifie la même ligne du fichier dans la branche “Branche1”



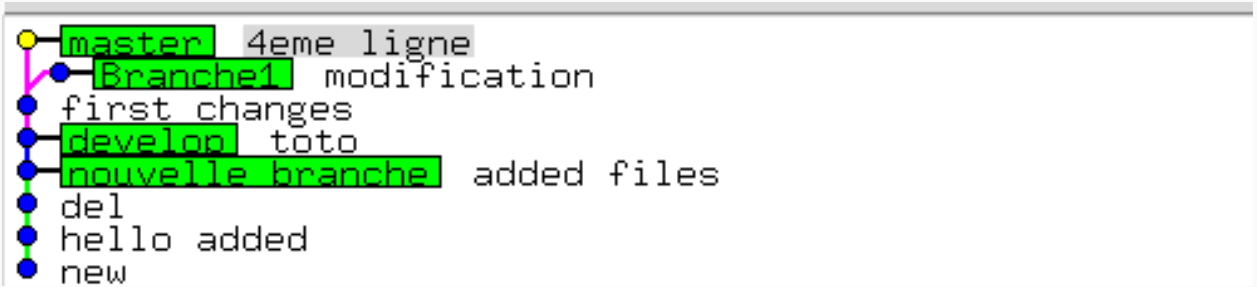

```
$ vi victor_hugo.txt
$ git commit -am "modfication"
[Branche1 09ee474] modification
1 file changed, 1 insertion(+), 1 deletion(-)
```

L’histoire d’un conflit sur GIT (4)




```
$ git checkout master
Switched to branch 'master'
```

L'histoire d'un conflit sur GIT (5)

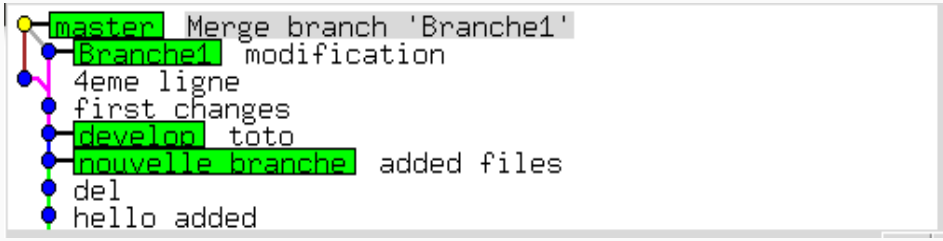



```
$ vi victor_hugo.txt
$ git commit -am "4eme ligne"
[master e4107d8] 4eme ligne
 1 file changed, 1 insertion(+)
```

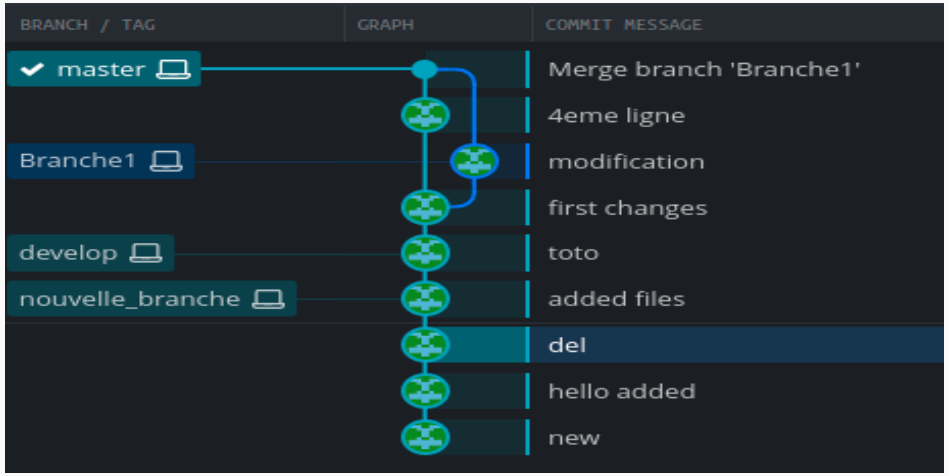
L'histoire d'un conflit sur GIT (6)

Auto-merging fonctionne quand des mots différentes ont été ajoutés dans la même ligne

gitk



gitKraken




```
$ git merge Branche1
Auto-merging victor_hugo.txt
Merge made by the 'recursive' strategy.
 victor_hugo.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

L’histoire d’un conflit sur GIT (6’)

Auto-merging ne marche pas quand le même mot a été changé de deux façons différents

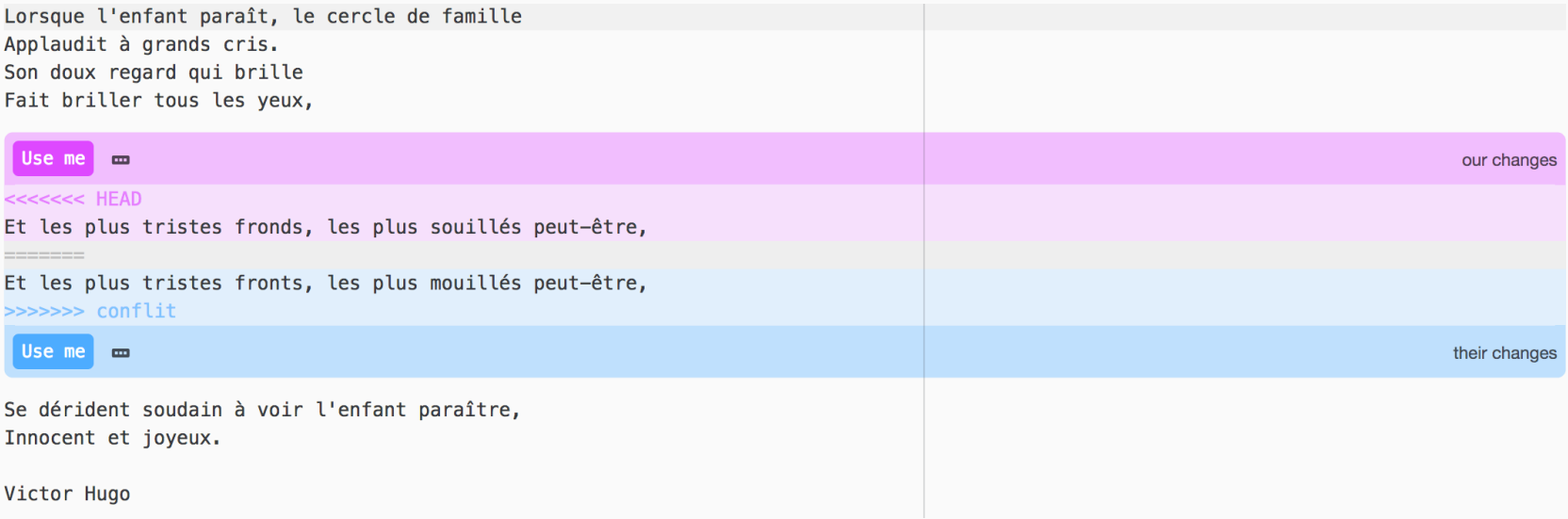
```
$ git merge Branche1
Auto-merging victor_hugo.txt
CONFLICT (content): Merge conflict in victor_hugo.txt
Automatic merge failed; fix conflicts and then commit
the result.
```

Exemple de fusion manuelle

```
Lorsque l'enfant paraît, le cercle de famille
Applaudit à grands cris.
Son doux regard qui brille
Fait briller tous les yeux,
<<<<<<< HEAD
Et les plus tristes fronds, les plus souillés peut-être,
=======
Et les plus tristes fronts, les plus mouillés peut-être,
>>>>>>> conflit
Se déridents soudain à voir l'enfant paraître,
Innocent et joyeux.

Victor Hugo
```

Plus simple avec un outil approprié (Atom) !



- Principe : Editer le fichier en question (victor_hugo.txt) de façon à trouver un consensus, puis `commit -am ``correction''`, puis `git merge Branche1`.

Plan

Introduction

- il n'existe pas une façon unique de travailler à plusieurs sur `git`
- dans tous les cas, il faut un serveur central
- ce serveur peut être :
  - monté par vous-même ou votre organisation
  - un provider quelconque (github est le plus connu)
- dans tous les cas, il faut apprendre quelques commandes `git` supplémentaires

Le workflow centralisé

- Fonctionne à peu près comme `svn`
- permet à tout le monde de travailler plus ou moins en même temps
- la gestion des conflits est du ressort de chacun des développeurs
- il faut avoir des droits sur le serveur pour attribuer à chaque développeur
- pas très scalable

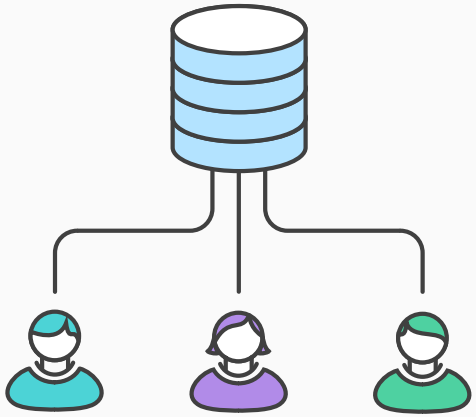


Illustration (1/5)

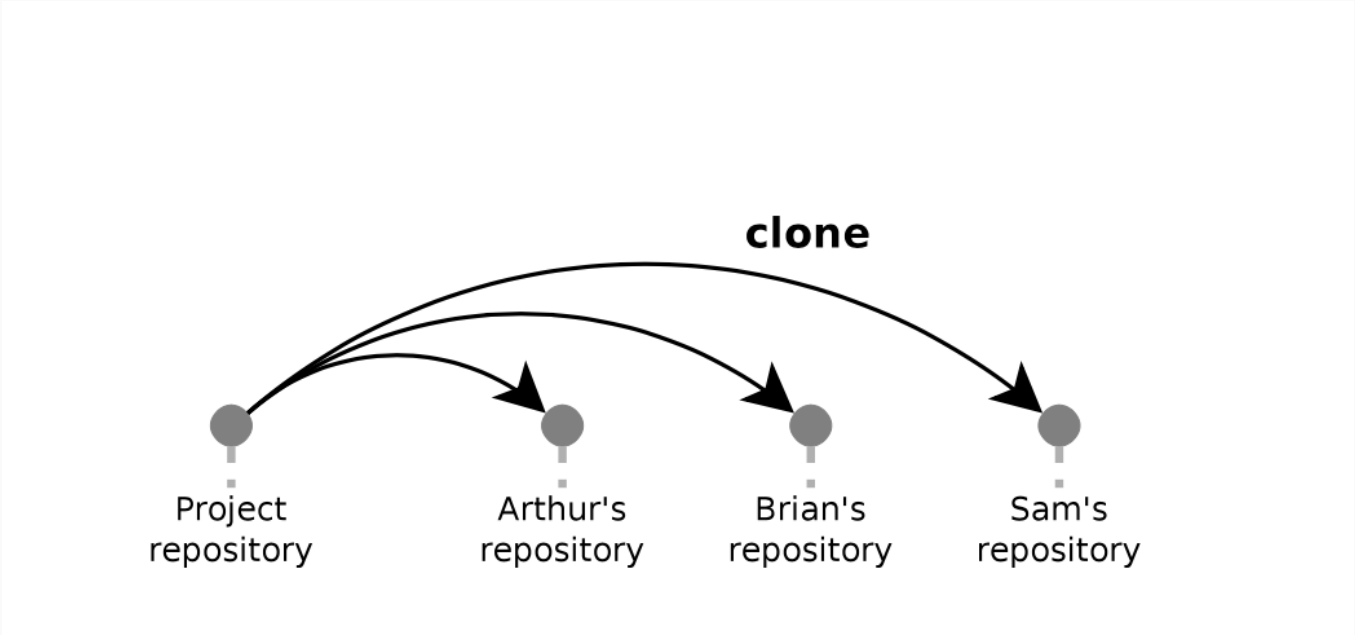


Illustration (2/5)

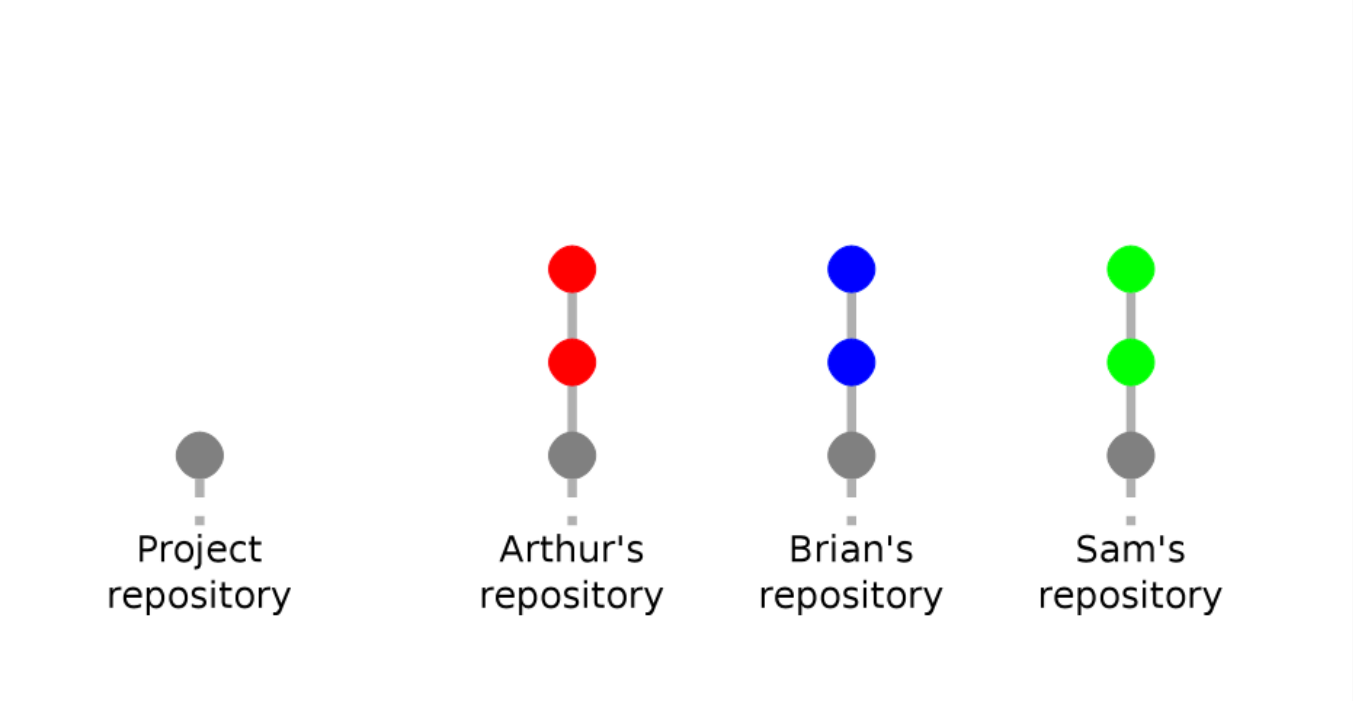


Illustration (3/5)

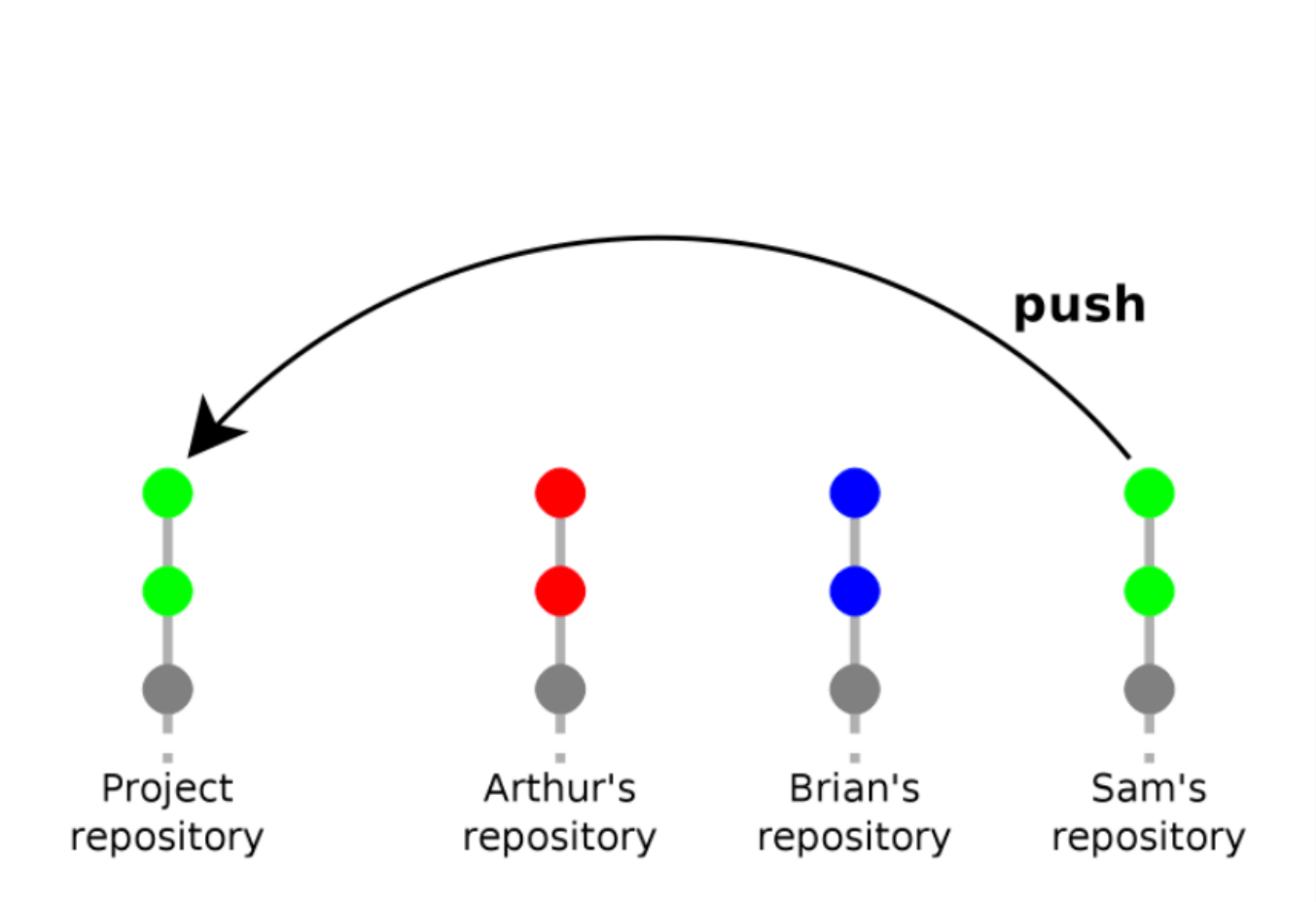


Illustration (4/5)

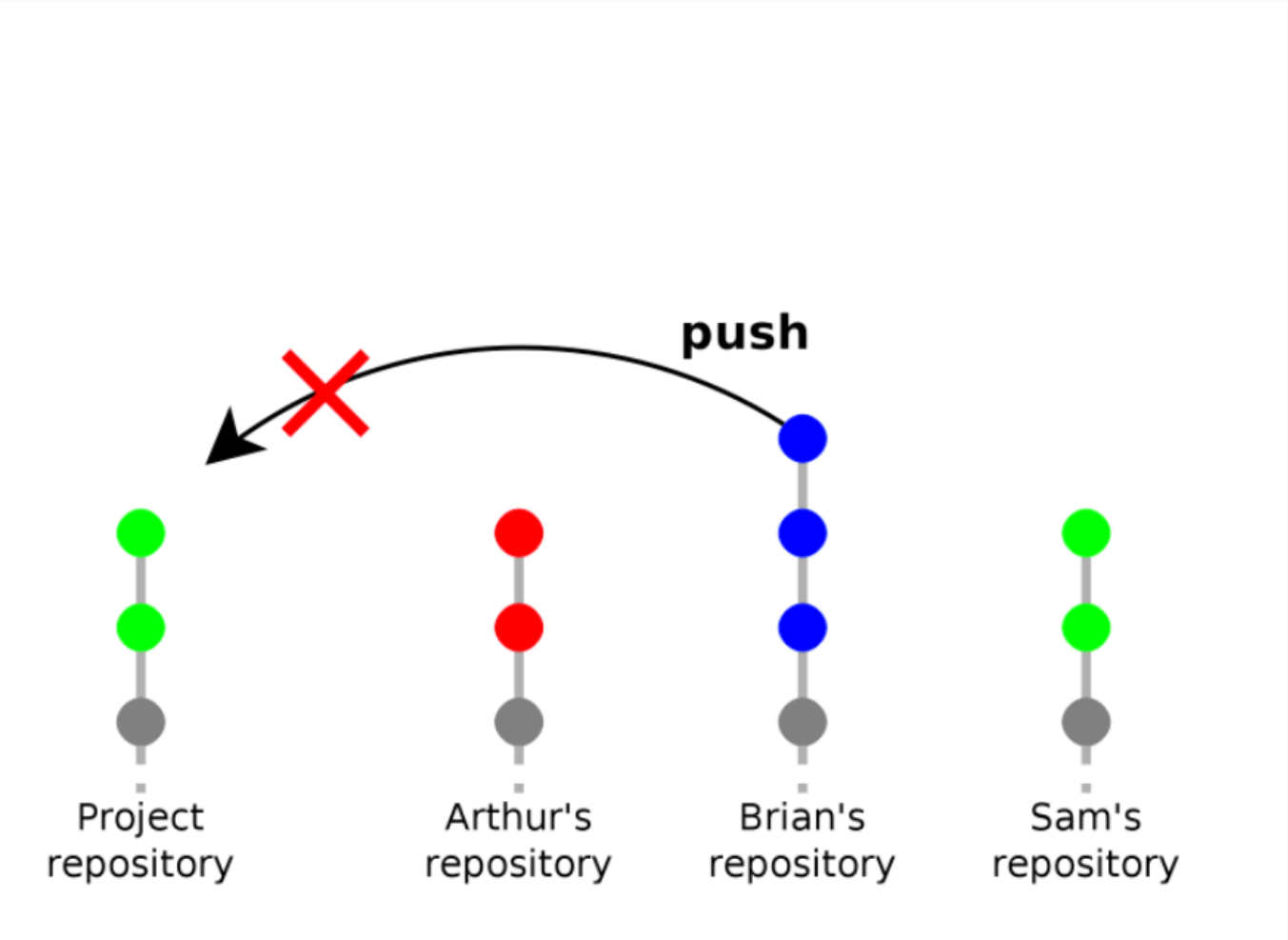
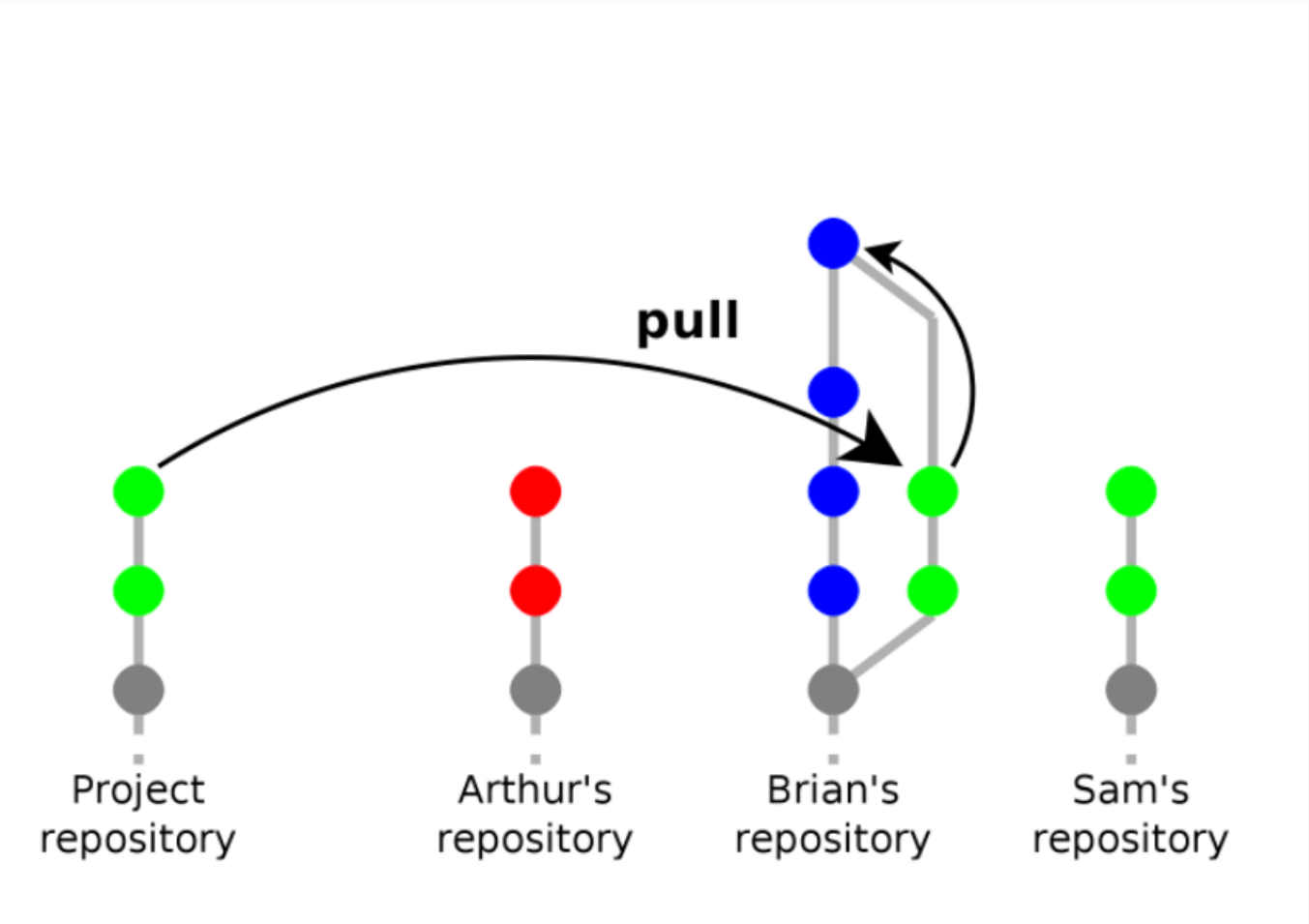
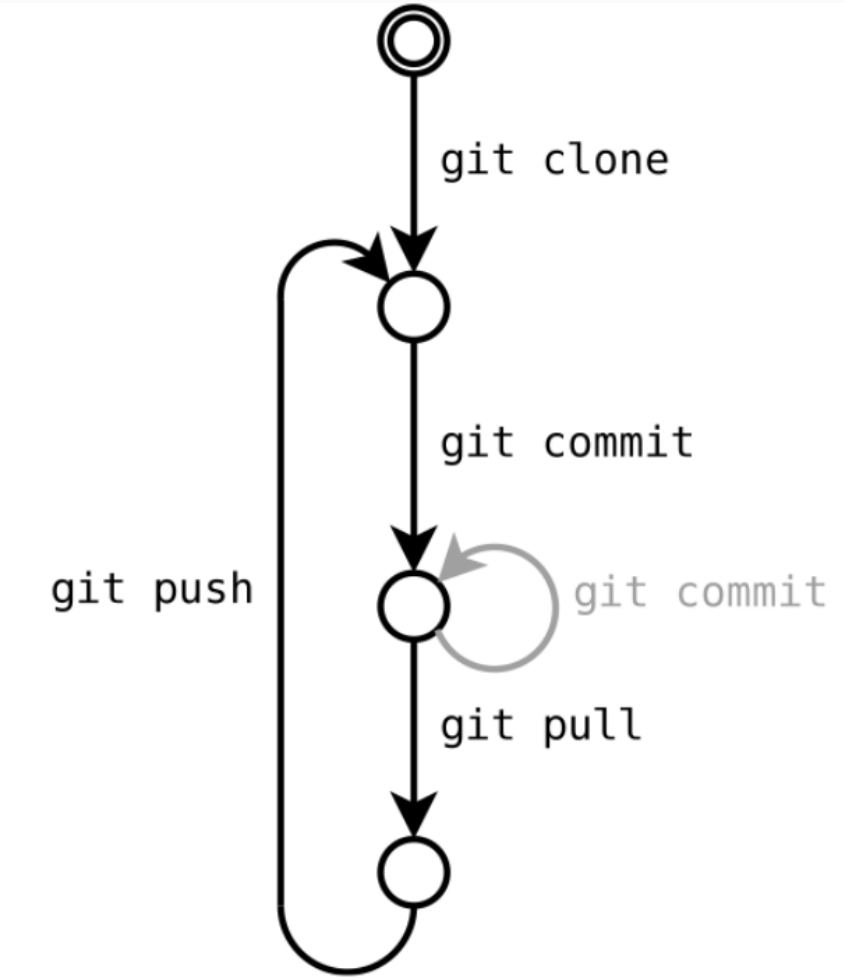


Illustration (5/5)



exemple classique de façon de travailler



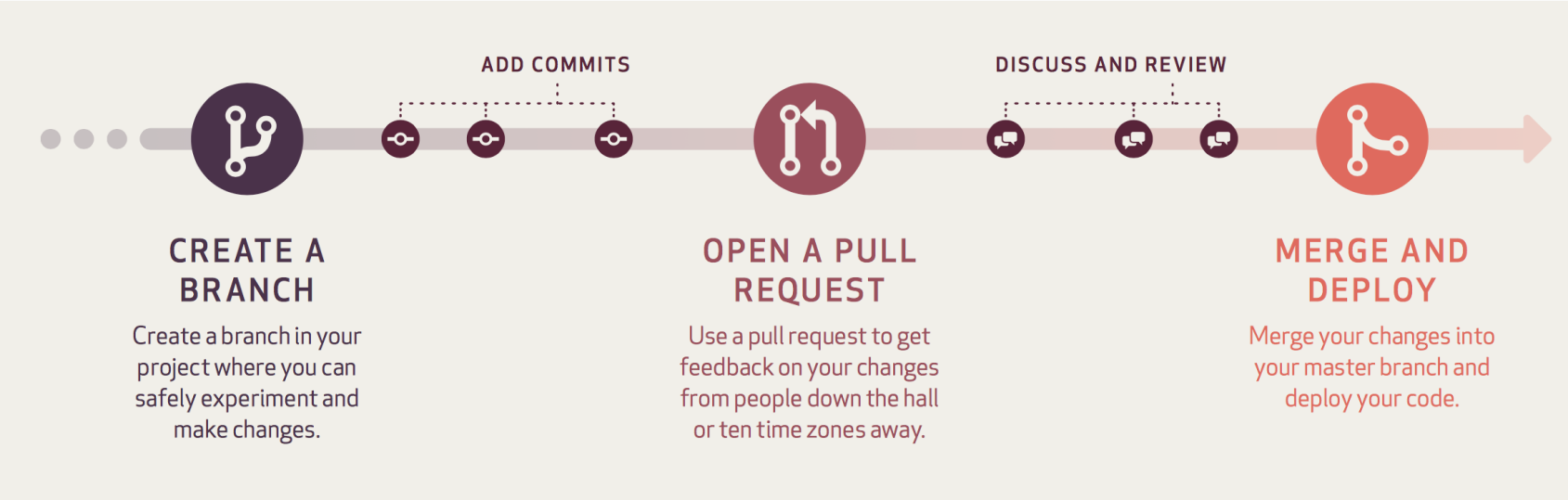
Commandes

`git clone ssh://user@host/path/to/repo.git`

- l'URL peut être différente (github vous les fournit à copier/coller)
- crée automatiquement un raccourci vers l'adresse locale appelé `origin`
- le processus est ensuite classique, il se déroule en local
  - `git status`
  - `git add`
  - `git commit`
- `git push origin master`
- mais ne pas oublier que les autres travaillent
  - `git pull`

Le workflow github

- vrai pour plusieurs hébergeurs `git` : github, bitbucket, gitlab...
- A tester par soi-même



Quelques références

- The `git` book: http://git-scm.com/book
- Documentation github : http://learn.github.com/
- Documentation Atlassian/bitbucket : https://www.atlassian.com/git/tutorial