Annexe B: Git

INF3135
Construction et maintenance de logiciels

Alexandre Blondin Massé

Université du Québec à Montréal

v253

Plan

Généralités

Logiciel de contrôle de versions (LCV)

- Permet de stocker un ensemble de **fichiers**
- Conserve la **chronologie** de modifications effectuées
- Permet de **partager** des fichiers entre **personnes**
- Conserve différentes **versions** des sources d'un projet
- Gère des versions **concurrentes** d'un projet
- Facilite la **sauvegarde** de fichiers
- Permet de visualiser les **différences** entre versions

Quelques LCV connus

BitKeeper, CVS, Darcs, **Git**, Mercurial, Subversion (SVN)

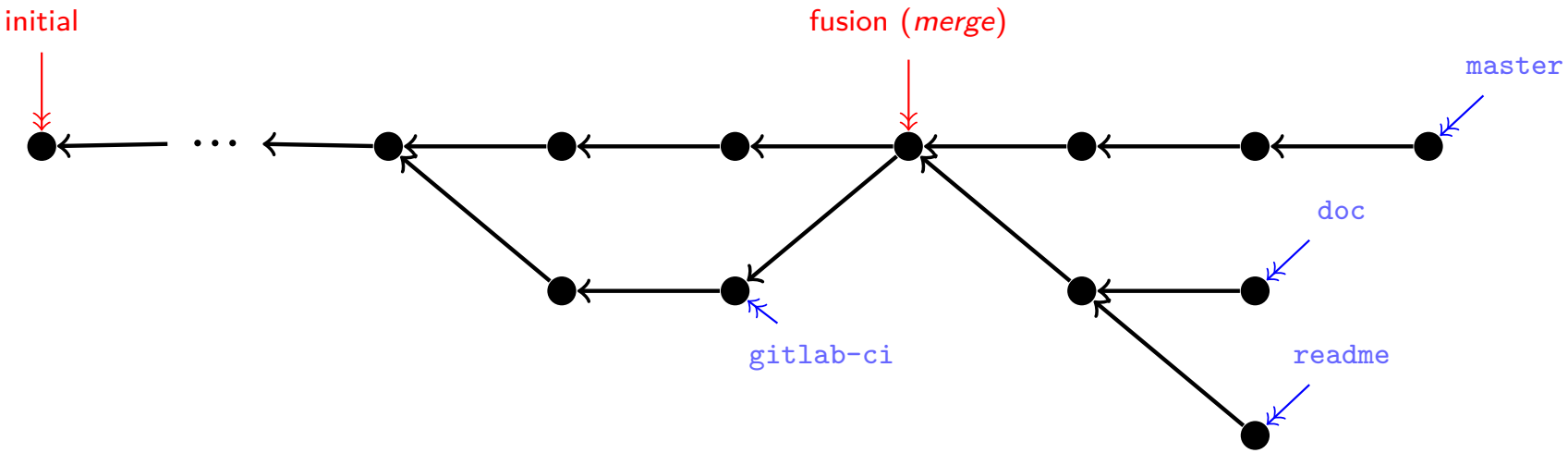
Naissance de Git

- **2002**: Linus Torvalds utilises BitKeeper pour conserver l'historique de Linux
- **6 avril 2005**: La version **gratuite** de BitKeeper est supprimée: Torvalds décide de créer son propre logiciel de contrôle de version, Git
- **18 avril 2005**: Git supporte l'opération de fusion de fichiers
- **16 juin 2005**: Git est officiellement utilisé pour conserver l'historique de Linux
- **Fin juillet 2005**: Junio Hamano devient le développeur principal de Git

Historique d'un projet/dépôt

- **Projet/Dépôt** = ensemble de fichiers dont on suit l'évolution

Graphe orienté acyclique (DAG = *directed acyclic graph*)



4 types d'objets

- **Blob** = *Binary Large OBject*: données binaires
- **Arbre**: arborescence de blobs ($\approx$ arborescence de fichiers)
- **Commit**: référence à un arbre
- **Annotation**: référence vers un *commit*

Commit

Métadonnées

- **Message**: décrit les modifications apportées
- **Auteur** (*author*): auteur original
- **Date de création**: date, heure, fuseau horaire
- **Livreur** (*committer*): personne qui a intégré le *commit*
- **Date de livraison**: date, heure, fuseau horaire

Identification

- **Clé**: 40 caractères hexadécimaux
- Produite par l'algorithme **SHA-1** (*Secure Hash Algorithm*)
- Probabilité que la clé soit **unique** $\rightarrow 1$

Exemple

- Souvent, **auteur** = **livreur**, mais pas toujours
- `git show`: voir informations sur un *commit*
- `--quiet`: cacher *diff*, `--pretty=fuller`: toutes les métadonnées

```
$ cd bats-core
$ git show --quiet --pretty=fuller 3b33a5a
commit 3b33a5ac6afd7f01ff4120659e2a72b851081178
Merge: 7b032e4 eb120d9
Author:     Sam Stephenson <sstephenson@gmail.com>
AuthorDate: Thu Oct 16 09:44:15 2014 -0500
Commit:     Sam Stephenson <sstephenson@gmail.com>
CommitDate: Thu Oct 16 09:44:15 2014 -0500

    Merge pull request #76 from jwerle/patch-1

    Update package.json

$ git show --quiet --pretty=fuller 3be8246 | head -n 5
commit 3be82466a7355b3a6f40f428d8c6520b63241593
Author:     Henrique Moody <henriquemoody@gmail.com>
AuthorDate: Wed Oct 30 22:10:00 2013 -0200
Commit:     Ross Duggan <rduggan@engineyard.com>
CommitDate: Wed Aug 13 14:32:35 2014 +0100
```

Références

Référence courante

- `HEAD`: référence vers l'état courant
- `HEAD~n`: référence vers le `n`-ième *commit* parent
- `detached HEAD`: référence à un *commit* non pointé par une branche

Branches

- Référence **nommée** vers un *commit*
- `master`: branche locale « principale »

Références distantes (*remote*)

- **Dépôt distant** nommé `origin` par défaut
- `origin/master`: branche « principale » du dépôt distant

Installation

```
# Distributions Debian
$ sudo apt install git-all
# Fedora/RPM/RHEL
$ sudo dnf install git-all
```

- **MacOS**: livré avec XCode
- **Windows**: télécharger l'installateur sur le site de Git

Utilisation

Git s'utilise de plusieurs façons:

- À l'aide d'un **terminal**
- À l'aide d'un **client graphique**:
  - GitHub Desktop (Windows/MacOS),
  - SourceTree (Windows/MacOS)
  - Magit (Linux/Windows/MacOS)
- À l'aide de votre **EDI**:
  - VScode
  - Sublime Text

Configuration

- **Par utilisateur**: `~/.gitconfig`
- **Par projet**: `.git/config`
- Syntaxe textuelle de type **INI**
  - Sections: `[nom de la section]`
  - Paires clé-valeur: `cle = valeur`

Exemple:

```
[user]
    name = Alexandre Blondin Massé
    email = blondin_masse.alexandre@uqam.ca
[alias]
    st = status -s
    co = checkout
    ci = commit
    br = branch
```

Opérations de base

Création d'un dépôt

Nouveau dépôt

```
git init
```

Copie d'un dépôt existant

- **Commande**: `git clone`
- *Fork*: gérée par l'hébergeur (GitLab, Github), pas par Git
- Deux **protocoles**: HTTPS et Git (SSH)

Répertoire spécial `.git`

- **Décentralisé**: tout l'historique y est contenu
- Chaque dépôt Git est **équivalent**
- Le répertoire qui contient le sous-répertoire `.git` peut donc être **déplacé** n'importe où

Consulter l'historique

Commandes fréquentes

- `git log`: journal détaillé des modifications
- `git log --graph --all --color`: historique en graphe
- `git show`: voir un *commit* spécifique
- `git diff`: différence entre deux versions

Astuce dans `.gitconfig`

- Ajouter un synonyme (*alias*) pour la commande `git gr`
- Permet de **visualiser** l'historique dans le terminal

État d'un projet

États possibles

- **Propre** (*clean*): aucun fichier versionné n'a été modifié
- **Modifié**: il y a eu certaines modifications
- **Indexé** (*staged*): certaines modifications ont été indexées

Connaître l'état d'un projet

```
$ git status      # Affichage long
$ git status -s   # Affichage compact
```

Astuce dans `.gitconfig`

Ajouter un synonyme pour `git st`:

```
[alias]
    st = status -s
```

Changer l'état du projet

Commandes

- `git checkout`
- `git switch`

Astuce dans `.gitconfig`

```
[alias]
    co = checkout
```

Astuce shell

- Afficher l'état dans l'**invite de commande**
- **Manuellement**: `~/.bashrc`, `~/.profile`
- **Gestionnaire de configuration**: ohmyz.sh

Exemples

```
$ git co master          # Se placer sur la branche principale

$ git co c648a85         # Se placer au commit c648a85
Note: checking out 'c648a85'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

$ git co origin/master   # Branche principale du dépôt distant
Note: checking out 'origin/master'.

You are in 'detached HEAD' state. You can look around, make experimental
[...]
```

- Possible d'**ajouter** des *commits*
- Et de créer des nouvelles **branches**
- Ou de **déplacer** des branches (*rebase*)
- On va y revenir plus tard

Préparer une sauvegarde (*commit*)

Cycle de développement de base:

1. `git st`: vérifier que l'état est **propre**
2. Apporter des modifications au projet (tâche **atomique**)
3. `git st`: vérifier l'état des fichiers modifiés
4. `git diff`: vérifier les modifications
5. `git add`: indexer des modifications (voir option `-p|--patch`)
6. `git ci`: confirmer la sauvegarde (alias `ci = commit`)

Raccourci

`git ci -a`: combiner l'indexation et la sauvegarde de **toutes** les modifications

Message de *commit*

Les 7 règles d'un bon message (plus de détails)

1. Limiter le sujet à **50 caractères**
2. Séparer le **sujet** du **corps** par une ligne vide
3. Commencer le sujet par une **majuscule**
4. Ne pas terminer le sujet avec un **point**
5. Utiliser l'**impératif** dans le sujet
6. Limiter à **72 caractères** la largeur du corps
7. Dans le corps, décrire **quoi** et **pourquoi** plutôt que **comment**

- Ne pas **mélanger** les langues

Conventions

- Dans un projet existant, respecter **les conventions**
- **Exemple**: commits conventionnels (issu du projet Angular)

Réinitialiser l'état du projet

Annuler l'indexation

`git reset`: annuler les commandes `git add` précédentes

Fichier spécifique

- `git checkout <fichier>`: annuler les modifications apportées à `<fichier>` depuis le dernier *commit*
- **Attention:** on ne peut pas revenir en arrière

Annuler les modifications

- `git reset --hard`: restaure les fichiers avant modifications
- **Attention:** on ne peut pas revenir en arrière

Mitigation

Vérifier avec `git st` et/ou `git diff` avant

Ignorer des fichiers

Plusieurs types de fichier

- **Sources**: code source, documentation texte
- **Ressources**: images, sons, vidéos, documentation
- **Artéfacts**: fichiers générés

Bonnes pratiques

- Versionner seulement les **sources**
- Fichiers **binaires**: dépend du volume
- Ignorer des motifs de fichier (*glob*): `.gitignore`
- Profiter de l'organisation **arborescente** des fichiers (héritage)
  - Exemple: `*.png` à la racine, `!*.png` dans le répertoire `images/`
- Éviter les gabarits **génériques**

Corriger un *commit*

Corriger le message

- `git ci --amend`
- Puis on réécrit le message

Corriger le contenu du dernier *commit*

- Apporter les **modifications** souhaitées normalement
- Indexer avec `git add` et `git add -p`
- Puis faire `git ci --amend` plutôt que `git ci`

Attention

- **Réécrit** l'historique
- À éviter sur les branches **principales partagées**

Bonnes pratiques

Faire des *commits* atomiques

- Correspond à **une itération**
- Soigner le **message** (surtout la première ligne)
- Peut toucher **plusieurs fichiers**
- Au besoin, utiliser `git add -p`

Vérifier avant chaque opération risquée

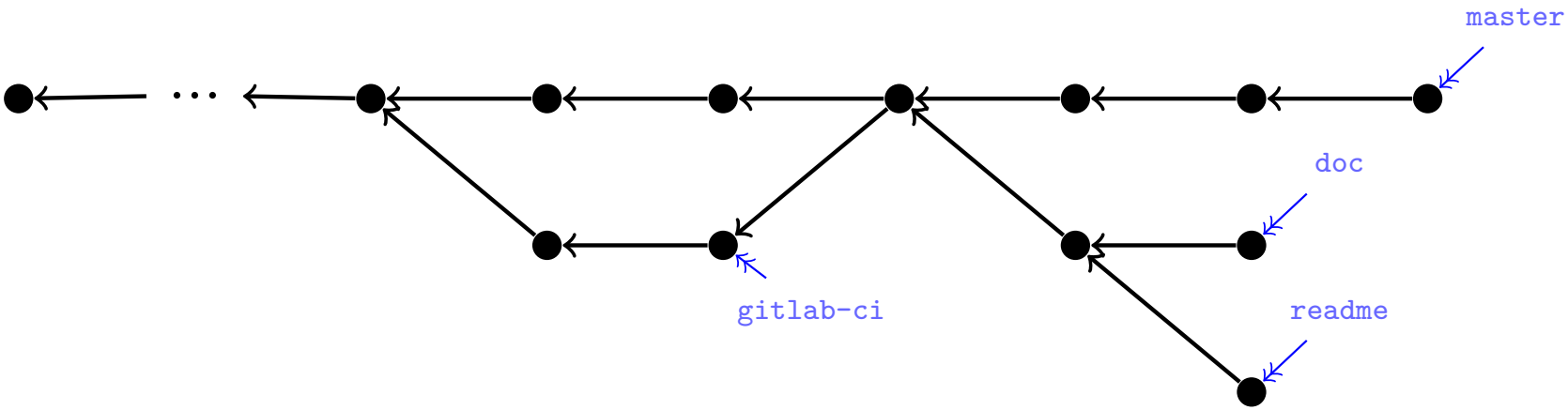
- `git status`
- `git diff`

Préférer des *commits* stables

- L'état du projet reste **valide**
  - Le programme **compile**
  - Les tests **réussissent**
- Sinon, **l'indiquer** dans le message (`WIP = work in progress`)

Branches

Branches



Rappels

- **Branche**: référence **nommée** vers un *commit*
- **Changer** de branche: `git checkout` ou `git switch`
- Chaque *commit* est ajouté sur la branche **courante**
- Puis la référence est **mise à jour** automatiquement

Opérations de base

- **Création**: `git checkout -b` ou `git switch -c`
- **Renommage**: `git branch -m`
- **Suppression**: `git branch -d` ou `git branch -D`

Dépôts distants (*remote*)

Git est un LCV **distribué**

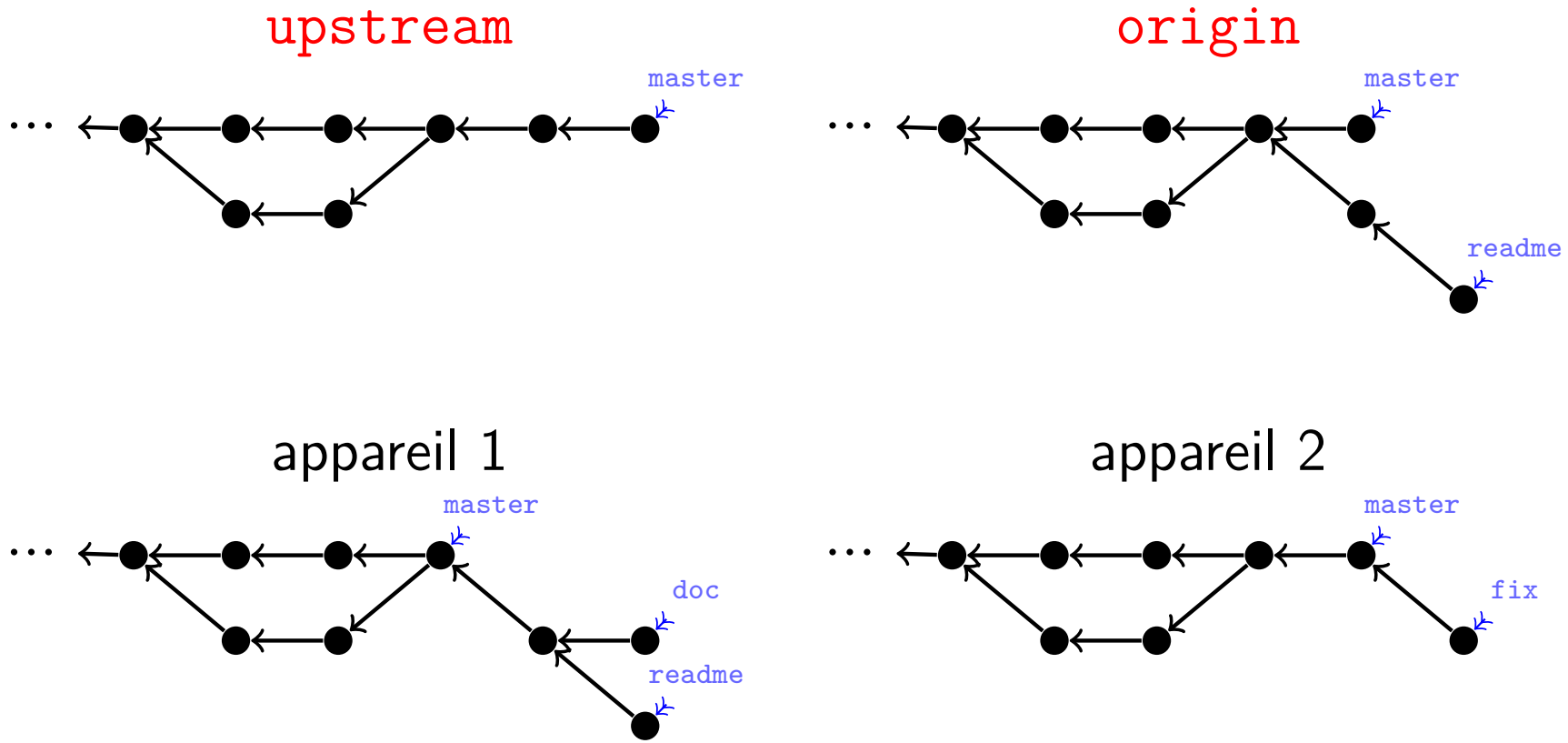
Conséquence

- **Plusieurs** copies d'un dépôt peuvent exister
  - Différents **appareils**
  - Différentes **copies distantes** (exemple: *fork*)
  - Différents **serveurs**
- Elles deviennent fréquemment **désynchronisées**

Synchronisation

- Action de réconcilier deux **versions**
  - En **partie** (la plupart du temps)
  - Ou en **totalité** (exemple: miroir)
- Synchroniser des dépôts = synchroniser ses branches

Exemple



Question: comment synchroniser ces différents dépôts?

Connexion entre deux dépôts

La connexion est orientée

- **Local**: le dépôt dans lequel on travaille
- **Distant**: le dépôt avec lequel on veut se synchroniser

La connexion est flexible

- Peut passer par le **réseau** (HTTPS, SSH)
- Ou sur le **même appareil**

Quelques commandes

`git remote`

- `git remote add NOM CIBLE`: établit une connexion nommée `NOM` entre le dépôt courant et le répertoire ou l'URL `CIBLE`
- `git remote -v`: liste les dépôts « connectés »
- `git remote rename`: renomme une connexion à un dépôt
- `git remote remove`: supprime une connexion à un dépôt
- `git remote set-url`: modifie l'URL d'une connexion à un dépôt

`git clone`

- `git clone CIBLE`: établit d'abord la connexion (avec `NOM = origin` par défaut), puis copie le dépôt

Partage de branches

Du dépôt local vers le dépôt distant

`git push DEPOT BRANCHE`: 3 résultats possibles

- La branche locale est une **sous-branche** de la branche distante: aucune **modification** (*everything is up-to-date*)
- La branche locale **diverge** (a un embranchement) par rapport à la branche distante: l'opération est **refusée**
- La branche locale est un **prolongement** de la branche distante: la branche distante est **prolongée**

Du dépôt distant vers le dépôt local

`git fetch DEPOT`: télécharge les branches **distantes** de `DEPOT`

- Ne modifie pas les branches **locales**
- L'opération réussit **toujours** (sauf si problème de connexion)

Fusion de branches

- Opération **fondamentale** sur les branches (en anglais, *merge*)
- La commande est `git merge`

Note

La commande `git pull` est une combinaison des commandes

- `git fetch`, pour télécharger la branche et
- `git merge`, pour la fusionner

Suggestion

- Éviter `git pull`
- Appeler d'abord `git fetch`
- Puis `git gr` pour visualiser l'historique
- Et finalement appeler `git merge`

Résultats possibles

Fusion automatique

- Pas de modifications **concourantes**
- Donc pas de conflit au niveau **textuel**
- Mais on doit vérifier qu'il n'y a pas de problème **logique**

Conflit

- Il y a des zones de texte conflictuelles
- Il y a des modifications **concourantes**
- On doit alors résoudre les conflits dans toutes ces **zones**
- On peut utiliser un **outil** de résolution (`mergetool`, `vimdiff`, ...)
- Ou résoudre le conflit **manuellement**

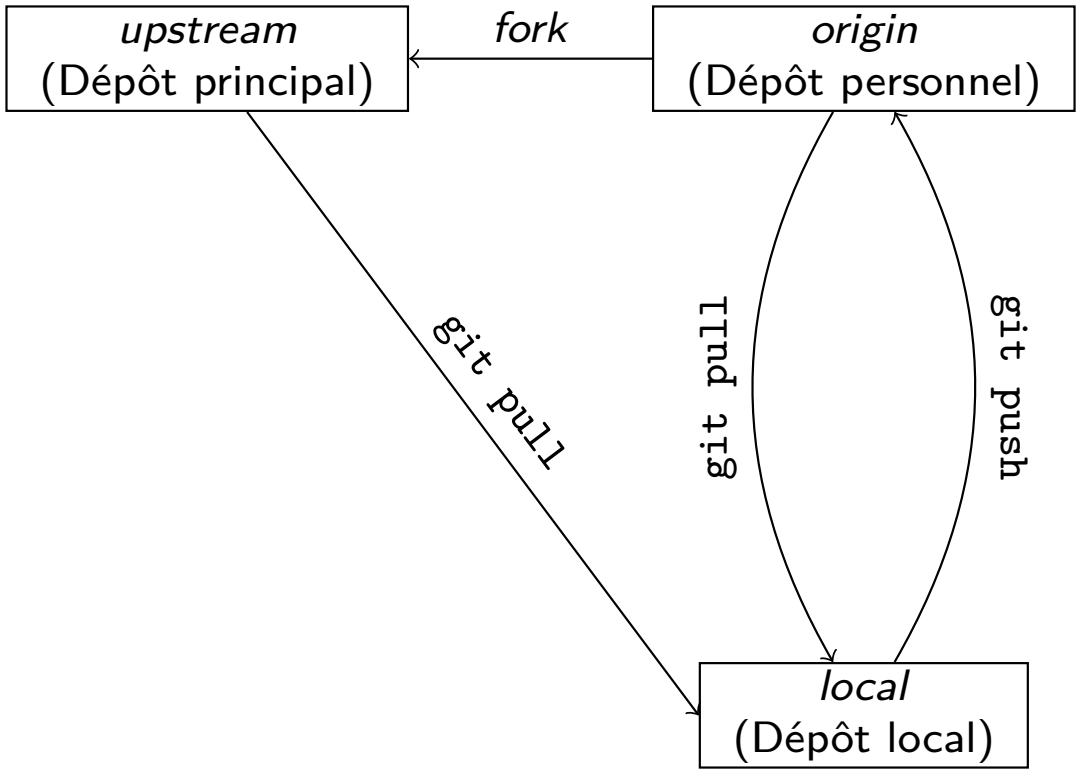
Résolution manuelle

- Lorsqu'il y a un conflit, la commande `git status` indique les fichiers dans lesquels il y a un problème
- Il suffit alors d'éditer **manuellement** chacun des fichiers en conflit, en identifiants les passages de la forme

```
<<<<<<< <nom de la branche 1>
// Bout de branche 1
=======
// Bout de branche 2
>>>>>>> <nom de la branche 2>
```

- Lorsqu'on a fini de régler le fichier, on indique que le conflit est réglé en faisant `git add <nom fichier>`
- Lorsque tous les fichiers ont été corrigés, on fait `git commit`
- Il y a un message de *commit* par défaut que vous pouvez laisser et, au besoin, ajouter des détails sur la stratégie utilisée pour régler le conflit

Se synchroniser avec des dépôts distants



- `git remote -v`: voir les informations sur dépôts distants
- `git remote add upstream <URL>`: ajouter un dépôt distant nommé `upstream`
- **Synchronisation** avec *upstream* sur `master`

Rebasement de branches

- Consiste à « arracher » une branche et à la « recoller » ailleurs.
- Il s'agit d'une opération qui **réécrit** l'historique

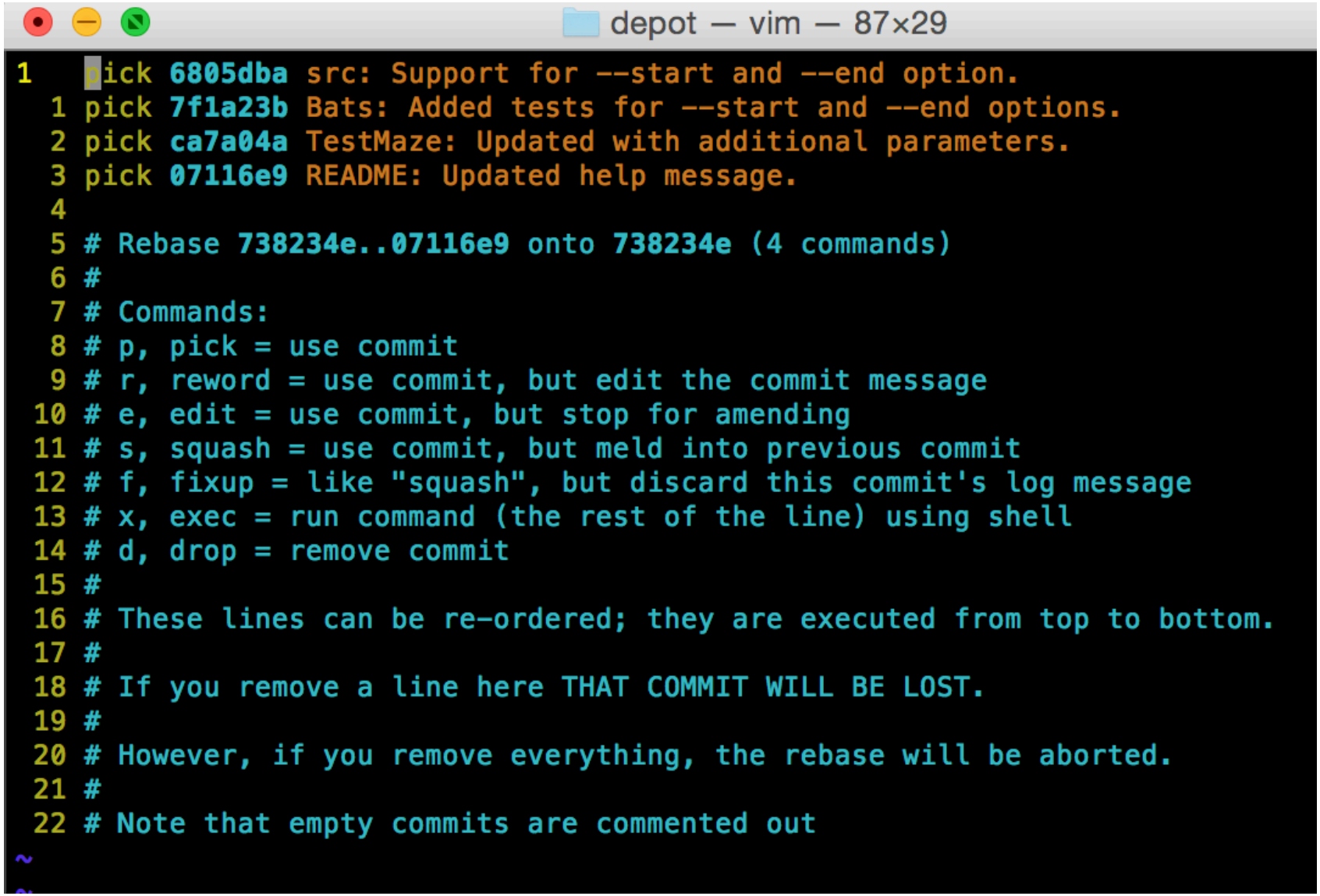
Rebasement interactif (`git rebase -i`)

On peut modifier un ou plusieurs *commits* lors du rebasement

- `p, pick`: conserver le *commit*
- `r, reword`: réécrire le message de *commit*
- `e, edit`: on prend le *commit*, mais on arrête pour le modifier
- `s, squash`: fusion de plusieurs *commits* en un seul
- `d, drop`: on jette le *commit*

On peut aussi **réordonner** les lignes pour inverser l'ordre d'application des *commits*

Rebasement interactif: dialogue



Bonnes pratiques

Profiter des branches

- Une branche = une **thématique**
- Quitte à passer régulièrement d'une branche à l'autre
- Donner un nom **significatif** à la branche
- Utiliser la syntaxe `kebab-case` pour nommer les branches
- Continuer de faire des *commits* **atomiques**
- Configurer le terminal pour afficher la **branche courante**

Utiliser une topologie simple

- **Supprimer** les « vieilles » branches
- Éviter les **embranchements** multiples
- Fusionner **fréquemment** les branches complètes
- Préférer un **rebasement** à une **fusion** (branche non intégrée)