Introduction à la gestion de versions avec Git

Thibault Cholez
thibault.cholez@loria.fr

TELECOM Nancy, première année

27/09/19

Outline

Outline

Définition

Définition

- Système permettant le suivi (dans le temps) des différentes modifications apportées à un ensemble de fichiers observés.
- Précision : indifférent au langage utilisé et ne se limite pas au code (fichiers de configuration, sources LaTeX, etc.).

Plusieurs accronymes

- SCM : Source Code Management
- VCS : Version Control System
- RCS : Revision Control System

Pourquoi les SCM ?

Problème

Maîtriser le code produit durant un développement logiciel implique de savoir ce qui a été fait :

- Par l'ensemble des développeurs (*Qui ?*)
- Dans le temps (*Quand ?*)
- Pour quel motif/fonctionnalité (*Pourquoi ?*)
- Impliquant de nombreuses fonctions, dans de nombreux fichiers (*Où ? Comment ?*)

→ Leur utilité ne se limite pas au travail à plusieurs.

Pourquoi les SCM ?

Utilité

- Permettre la traçabilité d'un développement (historique des changements)
- Faciliter la collaboration, éviter les conflits ou aider à la leur résolution
- Garder une version du code source toujours fonctionnelle, tout en travaillant sur plusieurs fonctionnalités (notion de branche)
- Permettre des schémas organisationnels structurant les développements (workflows)

Historique

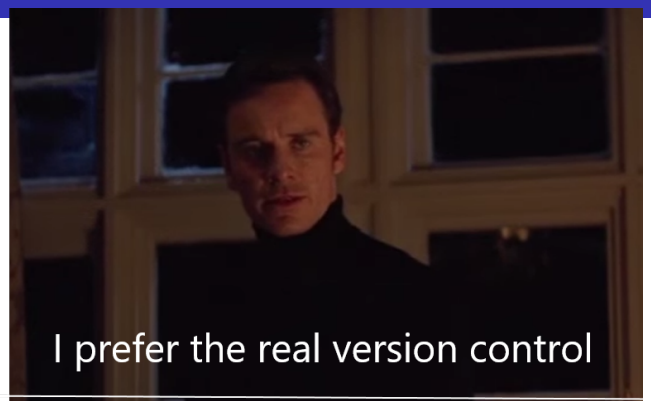
Principaux SCM

- cp -old (vieille blague, mais toujours utilisé)
- CVS (1990) : centralisé, travaille sur des fichiers, limité
- SVN (2000) : centralisé, travaille sur des fichiers, workflows limités
- Git (2005) : décentralisé, travaille sur des arborescences de contenus
- ... et beaucoup d'autres, libres (Mercurial, Bazaar) ou propriétaires (Microsoft, HP, IBM)

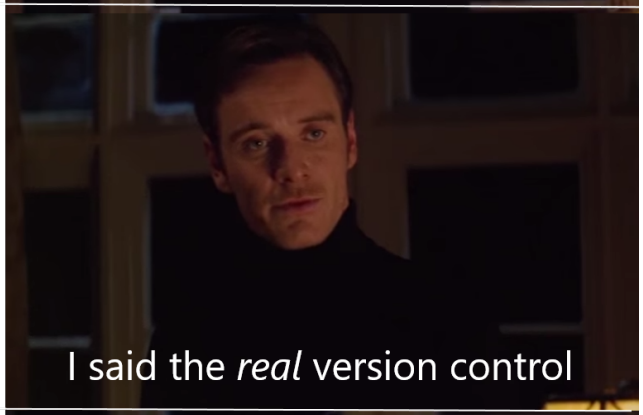
$\rightarrow$ Quelque soit le SCM utilisé, il est important d'en utiliser un et de maîtriser les opérations de base.

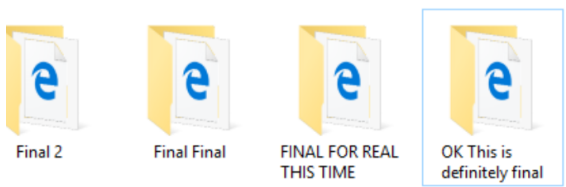
Git


git


I prefer the real version control


SUBVERSION


I said the *real* version control


Final 2
Final Final
FINAL FOR REAL THIS TIME
OK This is definitely final


Perfection

Git

Ne rigolez pas !

- Chaque année des groupes de projet, des stagiaires, etc. tentent de se passer de SCM...
- Partage du code par divers moyens archaïques et dangereux : clé USB, email, Google Drive, Instant Messaging, etc.
- Sauvegarde aléatoire, pas de traçabilité, N versions en parallèle difficiles à regrouper, etc.

On va vous aider à prendre les bonnes habitudes

- Rendu des projets en groupe via des dépôts git
- Tout élève n'ayant pas d'activité enregistrée dans le dépôt est considéré démissionnaire du projet

Un peu de vocabulaire

Principaux termes

- Repository ou dépôt : répertoire versionné (peut être local ou distant)
- Commit : Enregistrement des dernières modifications dans le dépôt
- Version ou revision : état du code source arrêté par un commit
- Branche : version alternative du code source liée à une tentative de développement spécifique
- Head : pointeur sur la version du code chargée (en général le dernier commit)
- Trunk ou tronc ou master : branche principale du code source
- Merge : tentative d'unification de deux branches
- Conflit : problème de merge nécessitant une prise de décision

Git

Pourquoi étudier Git ?

- Créé par Linus Torwarld pour gérer les sources du noyau Linux
- Très souple et puissant mais complexe et peu intuitif → apprentissage ardu
- SCM le plus utilisé actuellement, libre (GPL), communauté très active
- Devient un standard (l'est déjà dans le monde du libre)
- SVN est trivial en comparaison

Git

Avantages par rapport à SVN

- Pas de serveur central (élément critique)
- Utilisable même si déconnecté
- Organisation du travail plus souple

Git

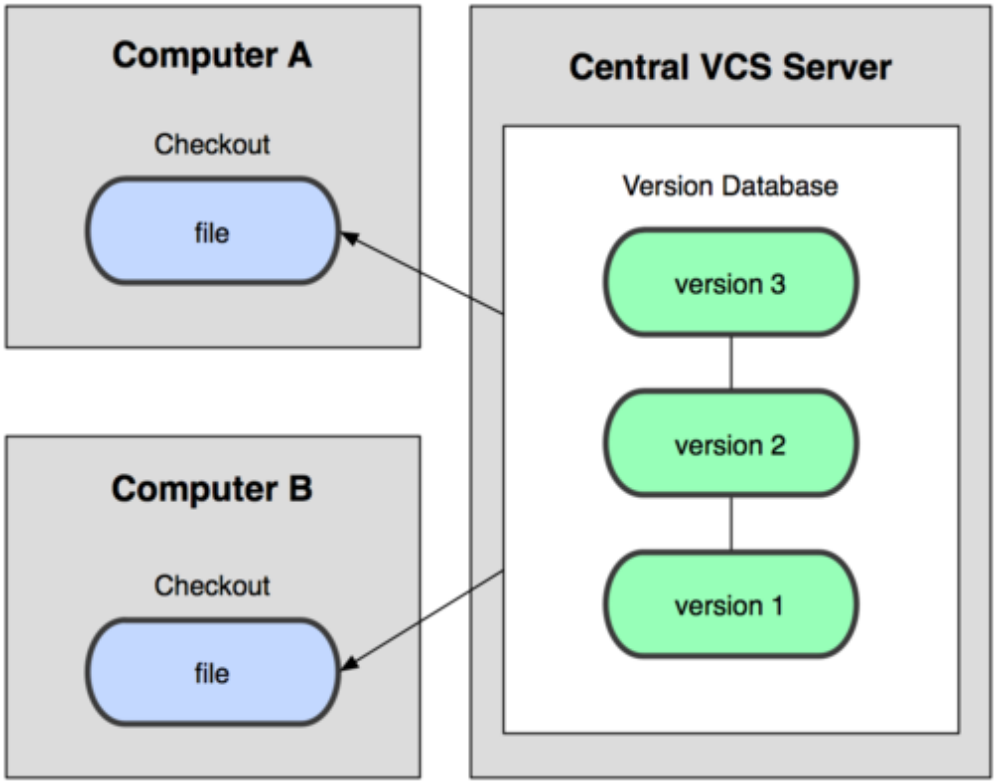


FIGURE – Gestionnaire de version centralisé vs décentralisé [1]

Git

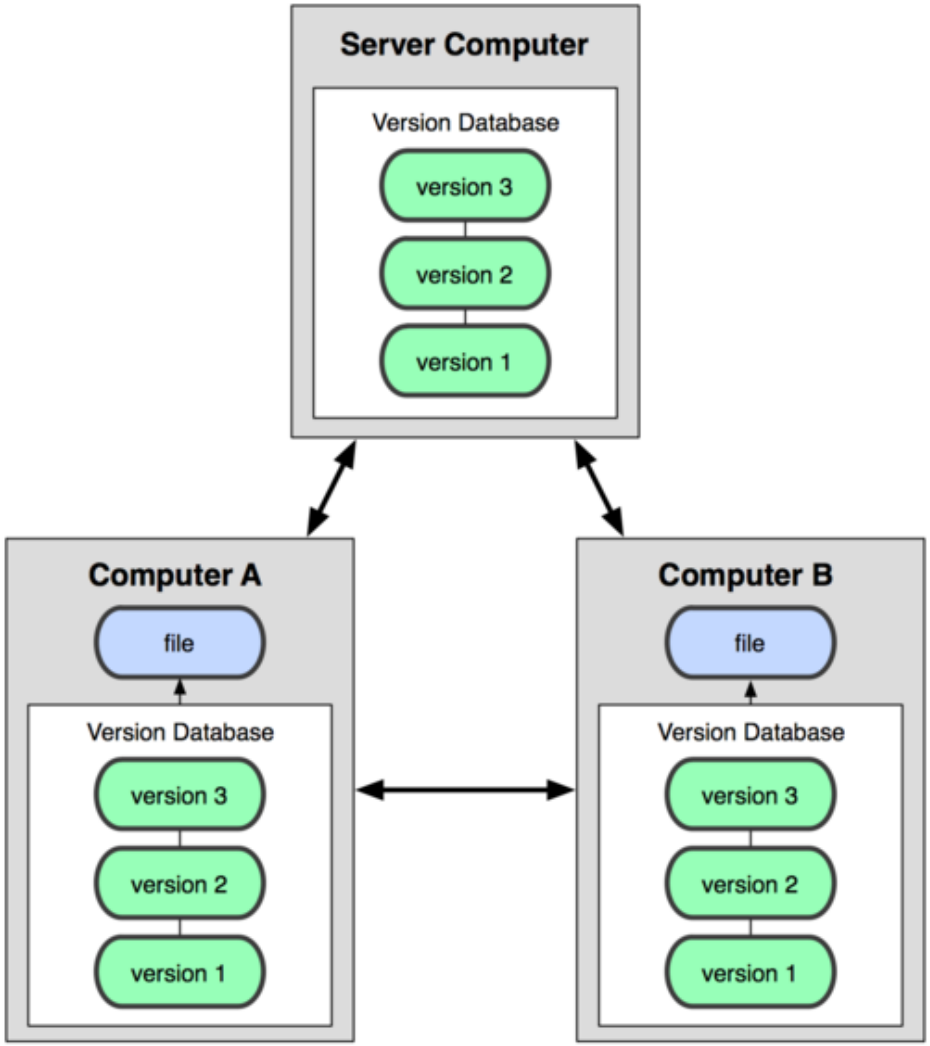


FIGURE – Gestionnaire de version centralisé vs décentralisé [1]

Git

Objectifs

- A court terme (module MIT) : comprendre le fonctionnement de git et maîtriser les commandes de base
- A moyen terme (diplôme) :
  - utiliser git dans tous vos projets à l'école (serveur GitLab : `https://gitlab.telecomnancy.univ-lorraine.fr/`)
  - maîtriser les commandes avancées

Vous AUREZ à utiliser un SCM tôt ou tard → profitez de votre scolarité pour (faire des erreurs et) monter en compétence !

Git

FIGURE – xkcd 1597 : Git

Outline

Outline

SCM décentralisé

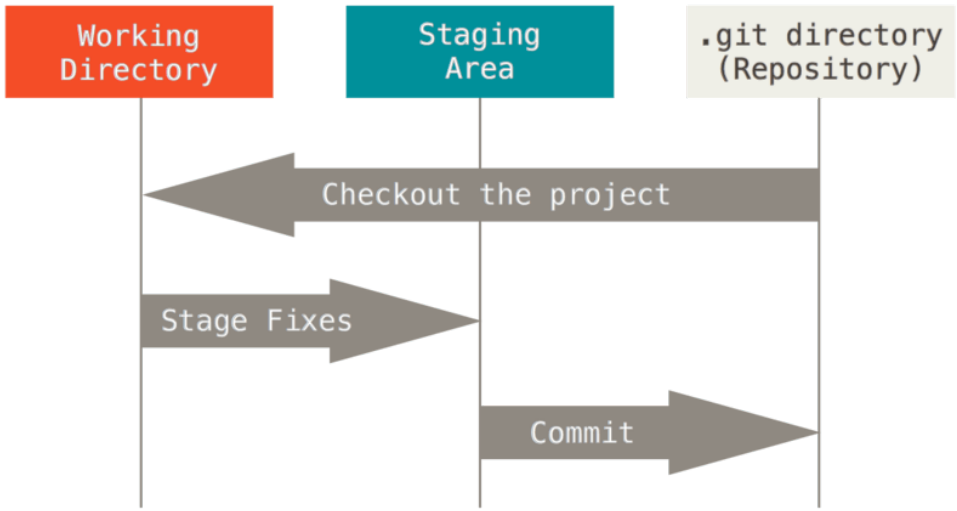
Principes fondateurs

- Chaque client git exécute son propre dépôt en local
- Chaque utilisateur d'un dépôt partagé possède une copie de tout l'historique des changements enregistrés (full mirroring)
- Abandon d'une vision chronologique des changements (pas d'ordre strict entre les commits) pour une vision structurelle (graphe de commits)
- But : faciliter les développements parallèles, permettre au code de diverger/converger rapidement

Zones de stockage dans git

Découpage interne en trois zones

- le répertoire de travail (working directory) local où sont réalisés les changements
- la "staging area" (aussi appelé index) où sont pré-enregistrés les changements (en attente de commit)
- le dépôt git où sont enregistrés les changements



Zones de stockage dans git

Découpage interne en trois zones

- le répertoire de travail est un bac à sable
- l'Index (staging area) est un instantané proposé de la prochaine validation
- HEAD est l'instantané de la dernière validation (commit), et sera le prochain parent

Staging area

Index intermédiaire

- Sert à préparer les commits progressivement
- git commit enregistre les modifications indexées
- La staging area peut être bypassée : git commit -a

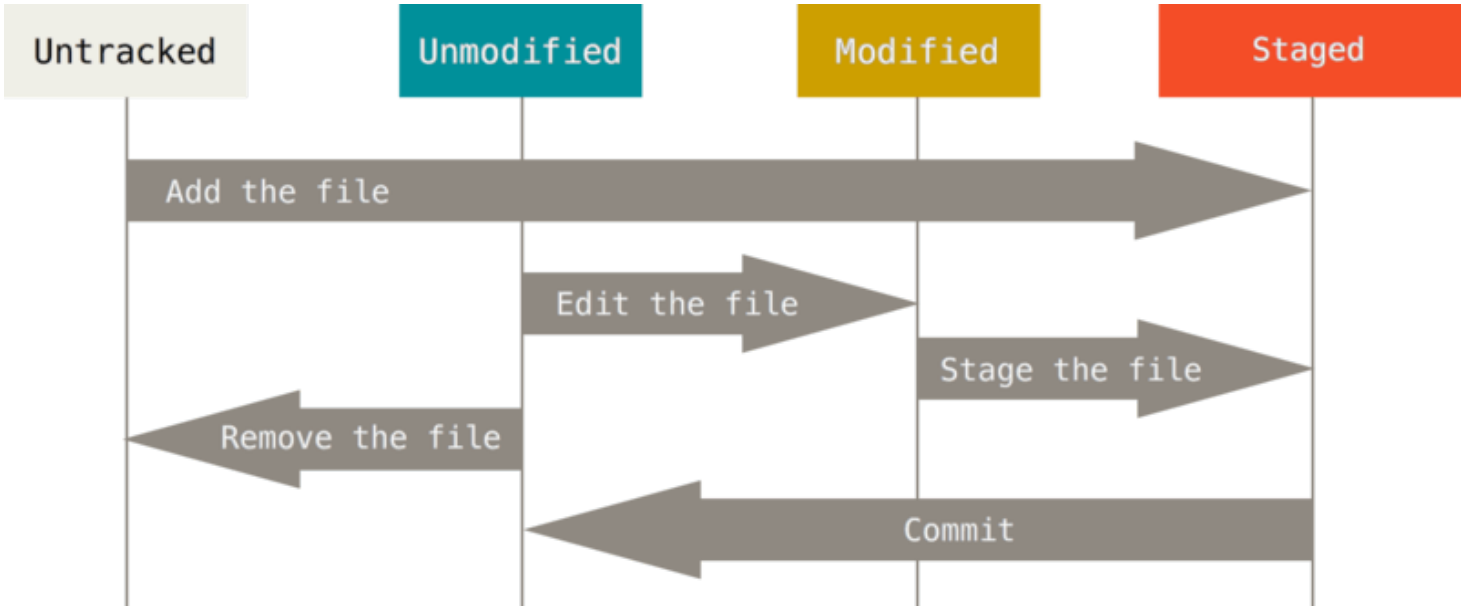


FIGURE – Différents états d'un fichier dans git [1]

Organisation des informations

Stockage par snapshots

- CVS, SVN ne stockent que les deltas des fichiers modifiés par un commit
- Git stocke tout le contenu du répertoire versionné à chaque commit (mais utilise une compression intelligente basée sur la version antérieure la plus proche)
- Permet une grande souplesse

Stockage par snapshots

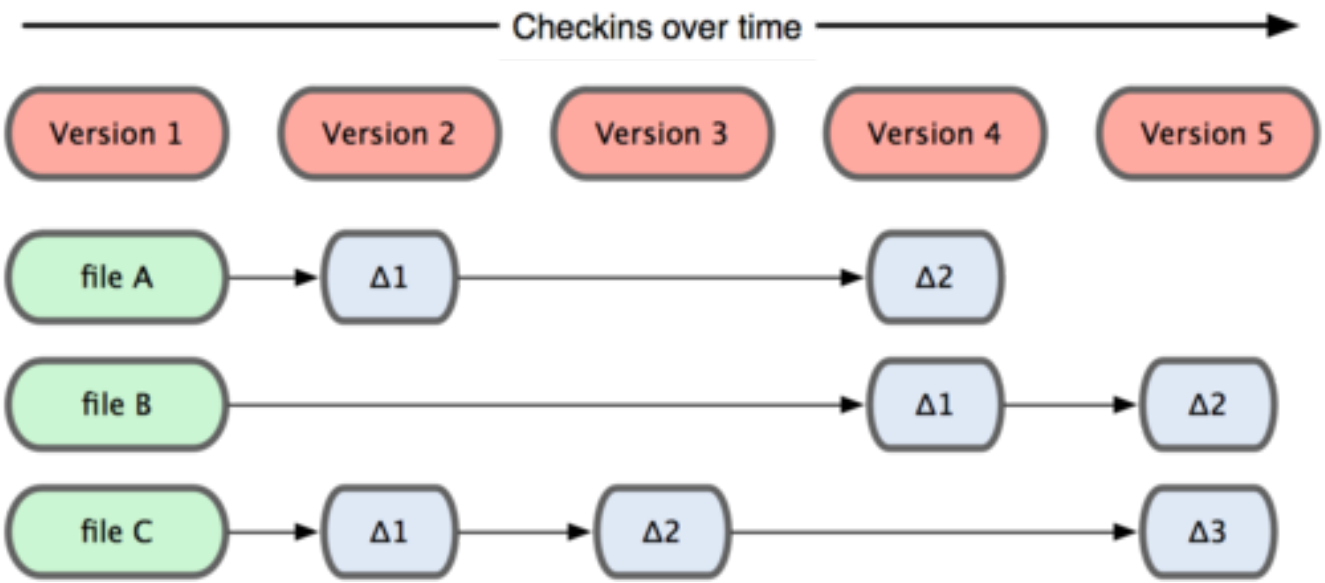


FIGURE – Delta vs Snapshot storage [1]

Stockage par snapshots

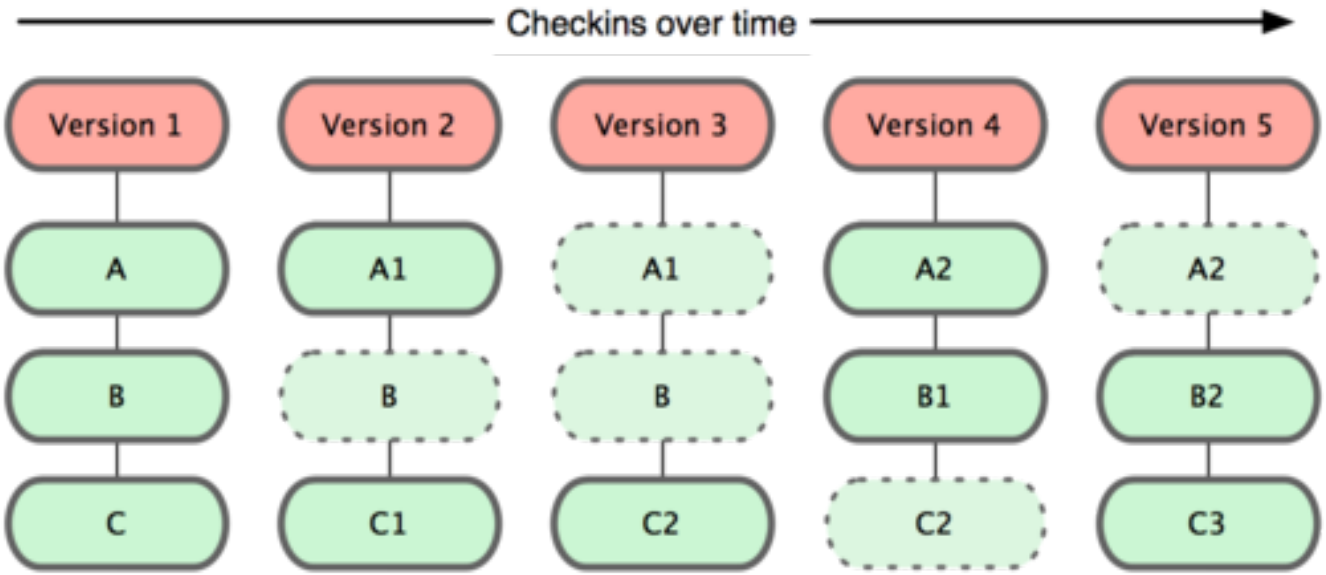


FIGURE – Delta vs Snapshot storage [1]

Organisation des informations

Types de données Git

- **Blob** : contenu d'une version d'un fichier (Binary Large Object)
- **Tree** (structure récursive) : arborescence de références vers d'autres Trees et Blobs
- **Commit** (structure récursive) : pointe sur un Tree, sur le Commit parent et contient des metadonnées (date, auteur, etc.)
- **Tag** : annotation manuelle d'un commit et créant une branche statique (équivalent à un pointeur)
- Identifiant unique pour tout objet : hash SHA1 (Secure Hash Algorithm). Peut être tronqué si non ambigu.

Organisation des informations

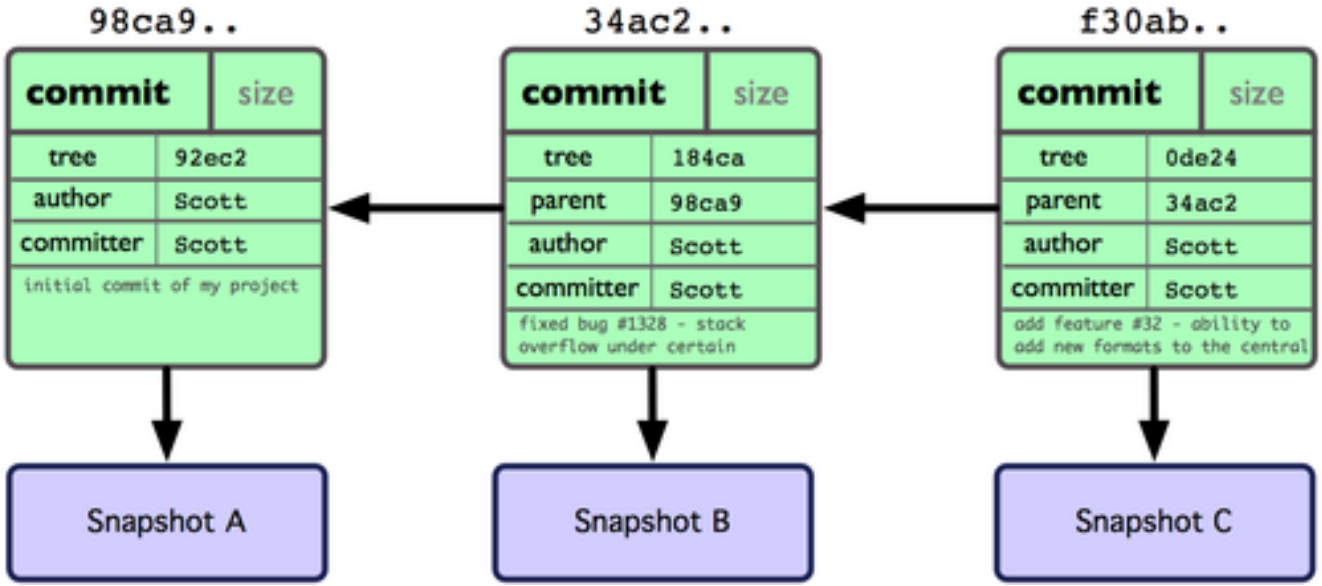


FIGURE – Data structure d'un commit [1]

Organisation des informations

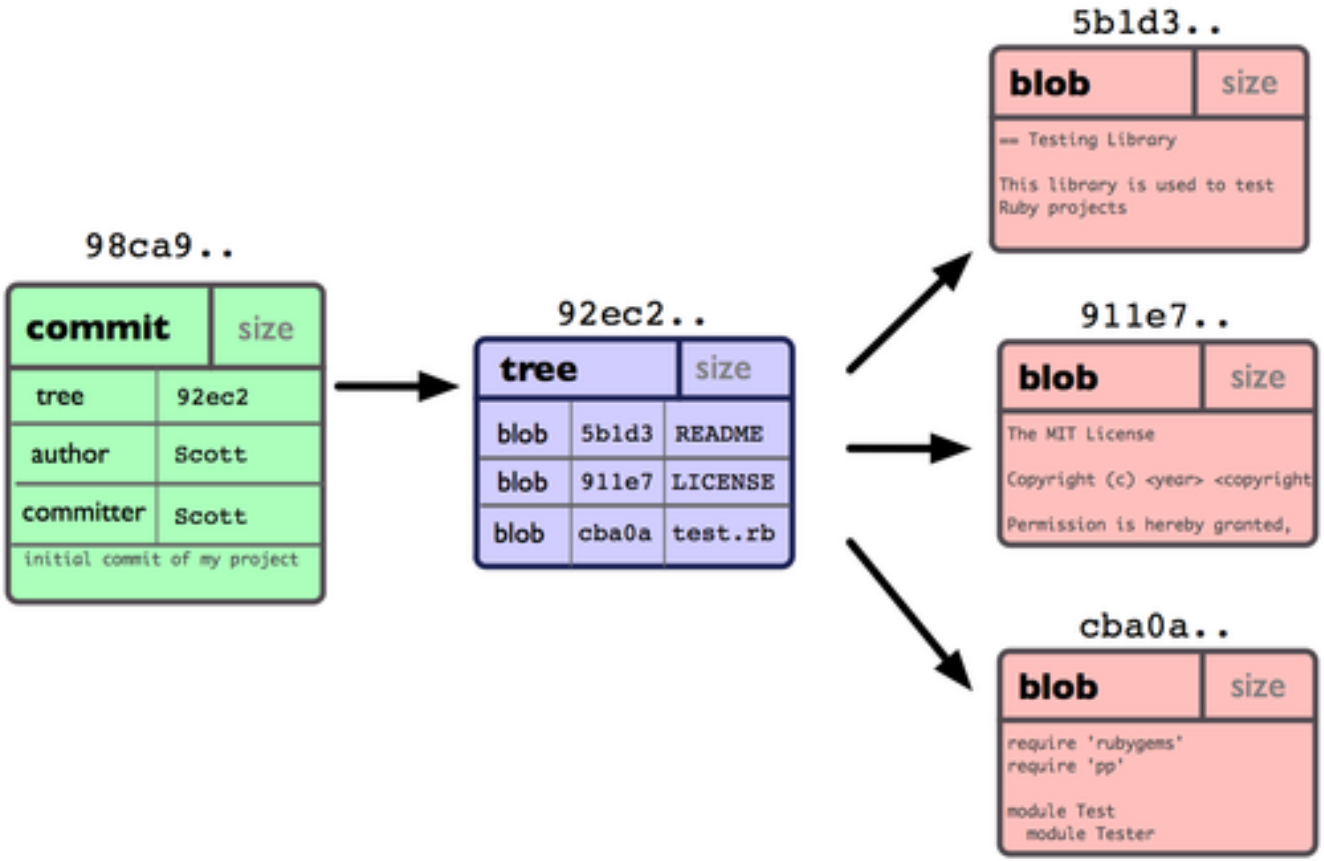


FIGURE – Data structure d'un commit [1]

Organisation des informations

Emplacement du dépôt local

Les données propres à git sont stockées dans un unique répertoire **.git** à la racine du projet. C'est le dépôt local.

Références

L'historique du projet est un graphe de commit. Certaines références sur ce graphe sont utiles :

- master : référence la branche principale
- HEAD : par défaut, référence le commit le plus récent de la branche courante (sera le parent du prochain commit)
- HEAD-2 : deux commits avant la HEAD

Références sur l'arbre des commits

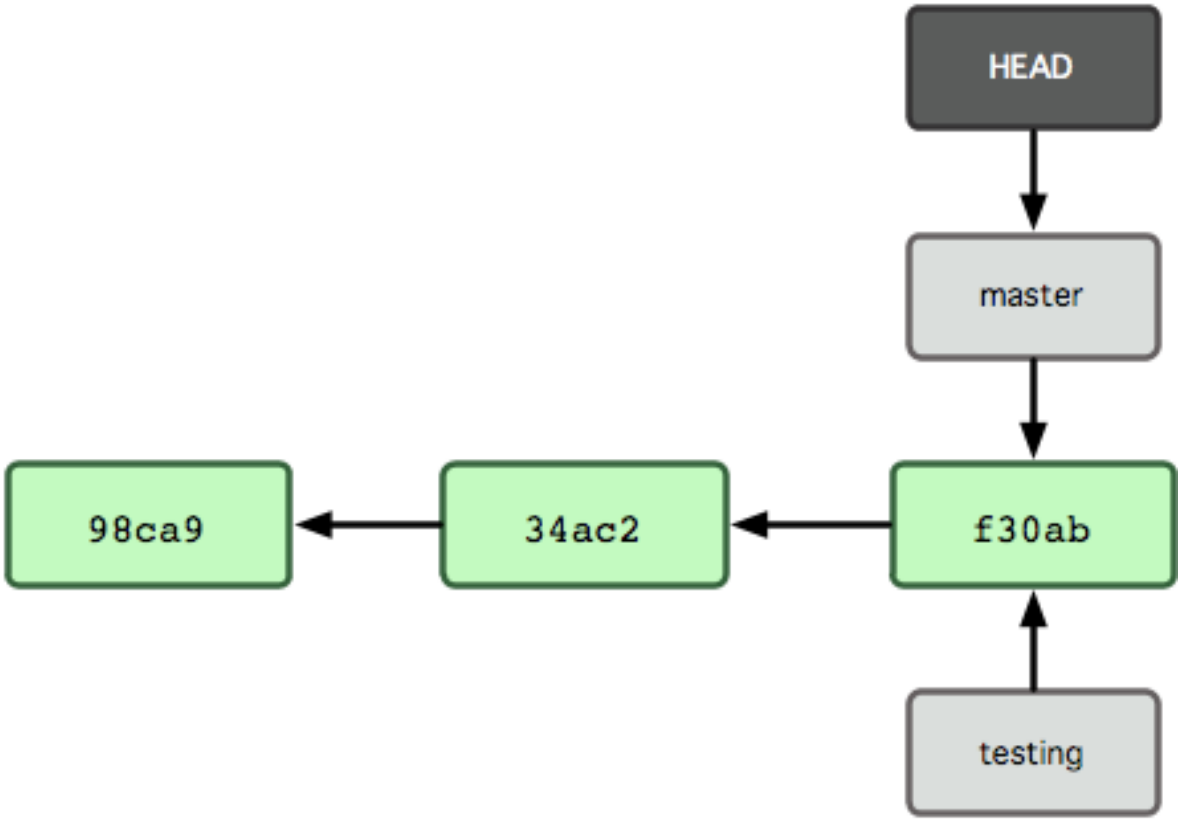


FIGURE – Références sur l'arbre des commits [1]

Bonnes pratiques

- Ne **pas versionner** de fichiers générés automatiquement (logs, pdf, exécutables, etc.) ou personnels
- Faire de petits commits réguliers et facile à intégrer, leur donner un nom explicite
- Utiliser les branches pour :
  - les développements à plusieurs
  - chaque développement conséquent d'une nouvelle fonctionnalité
- Ne pas développer sur la branche master à plusieurs pour éviter les conflits lors des pull
- Faire de petits commits locaux, et pusher des commits plus conséquents, toujours testés et fonctionnels !
- Faire des pull régulièrement

Que versionner ?

Du code source et rien que du code source !

- Git (comme les autres SCM) et language-agnostique
- Il ne "voit" que des lignes texte...
- ... et ne peut gérer intelligemment les versions que des fichiers texte

Ne pas versionner les autres formats

- Tout fichier non textuel (image, binaire exécutable, etc.)
- Tout fichier résidu de compilation (logs, etc.)
- Utiliser le fichier .gitignore pour définir les extensions de fichiers qui ne seront PAS versionnées : `https://github.com/github/gitignore`

Comment Git fusionne-t-il les versions ?

Principes

- Structure des données git : DAG (Directed Acyclic Graph) de commits
- Comparaison réalisée grâce à diff3 : utilitaire pour comparer 3 fichiers
- Fusion réalisée par l'algorithme *three-way merge*

3-way merge

- Pour fusionner A&B : A&B=A+B-C, C étant l'ancêtre commun le plus récent à A et B
- Revient à appliquer A-C à B et B-C à A
- Fait intervenir le *longest common subsequence problem*

Comment Git fusionne-t-il les versions ?

Application du 3-way merge

- Supposons A aligné avec C et B aligné avec C (application du LCS)
- On retient pour le merge tout contenu :
  - commun aux 3
  - présent dans A mais absent dans B et C
  - présent dans B mais absent dans A et C
  - *présent dans A et B mais absent dans C*
- On supprime pour le merge tout contenu :
  - présent dans A et C mais absent dans B
  - présent dans B et C mais absent dans A
  - *présent dans C mais absent dans A et B*

Exemple de 3-way merge

```
A:      C:      B:      -> merged:
milk    milk    milk        milk
juice                       juice
flour   flour   flour       flour
                sausage     sausage
eggs    eggs    eggs        eggs
        butter  butter
```

Visualisation des conflits

Quand un même bloc de C est modifié différemment par A et B. Git présente par exemple :

```
<<<<<<< A
I had one egg and three sausages for breakfast.
=======
I had two eggs and two sausages for breakfast.
>>>>>>> B
```

Marqueurs de conflits

- `<<<<<<<` : Indique les début des lignes en conflit (version locale)
- `=======` : Sépare la version locale (au dessus) de l'autre version (en dessous)
- `>>>>>>>` : Indique la fin de l'autre version en conflit.

Visualisation des conflits

Possibilité de visualiser la version antérieure :

```
<<<<<<< A
I had one egg and three sausages for breakfast.
||||||| C
I had one egg and two sausages for breakfast.
=======
I had two eggs and two sausages for breakfast.
>>>>>>> B
```

Marqueurs de conflits

- ||||||| : Indique la version de l'ancêtre commun

Outline

Configuration et aide

Aide

- Manuel d'utilisation : man git
- Aide générale : git help
- Aide d'une commande : git help *commande*

Configuration globale

- Configuration commune à tous les dépôts d'un utilisateur, voir le fichier $HOME/.gitconfig
- git config –global user.name "Thibault Cholez"
- git config –global user.email thibault.cholez@telecomnancy.eu
- git config –global color.ui true

Premières commandes

Initialisation du dépôt

- Versionner un répertoire courant : git init (créer un dépôt en générant le répertoire .git à la racine)
- Télécharger un dépôt existant : git clone *url*

Enregistrement des modifications

- Ajouter un fichier à suivre pour le prochain commit :
  git add *fichier*
- Enregistrer les modifications dans le dépôt local :
  git commit *fichier*
  ou git commit -m "Description du commit"
  ou git commit -a (pour all, commit tous les fichiers modifiés sans passer par la staging area)

Partage des versions

Propagation et récupération

Un dépôt distant est toujours accessible et permet de partager les commits entre développeurs (ex : GitHub).

- Si besoin, lier un dépôt local avec un dépôt distant :
  git remote add *nom_serveur adresse_depot*
- Envoyer les derniers enregistrements vers le dépôt distant :
  git push (*nom_serveur*
  nom_branche_locale :*nom_branche_distante*)
- Récupérer et intégrer les derniers enregistrements :
  git pull (*nom_serveur nom_branche_distante*)
  Le nom par défaut d'un serveur est **origin**.
- Récupérer les derniers enregistrements sans les intégrer :
  git fetch *nom_serveur nom_branche_distante*

Fichiers

Manipulation de fichiers

Attention : pour être enregistrés, les commandes manipulant le système de fichiers doivent être réalisées par l'intermédiaire de git !

- Renommer un fichier : git mv *old_name new_name*
- Supprimer un fichier : git rm *fichier*

Consultation des méta-données

- Consulter les modifications du working dir depuis le dernier commit : git status
- Consulter les modifications en cours du contenu des fichiers suivis : git diff
- Consulter les modifications du contenu des fichiers entre deux commits : git diff *sha1 autre_sha1*
- Consulter l'historique des commits : git log
- Consulter l'historique des manipulation du graphe (déplacement de HEAD) : git reflog
- Consulter qui a modifié quelles lignes d'un fichier : git blame *nom_fichier*
- Consulter un objet git : git show *sha1*

Annuler des modifications

Sur un **fichier**

- Retirer un fichier indexé dans la staging area (inverse de git add) tout en gardant ses modifications dans le working dir : git reset *nom_fichier*
- Annuler les modifications courantes d'un fichier ou répertoire du working dir en rappelant celles :
  - de la staging area : git checkout *nom_fichier*
  - d'un commit : git checkout *SHA1 nom_fichier*

Annuler des modifications

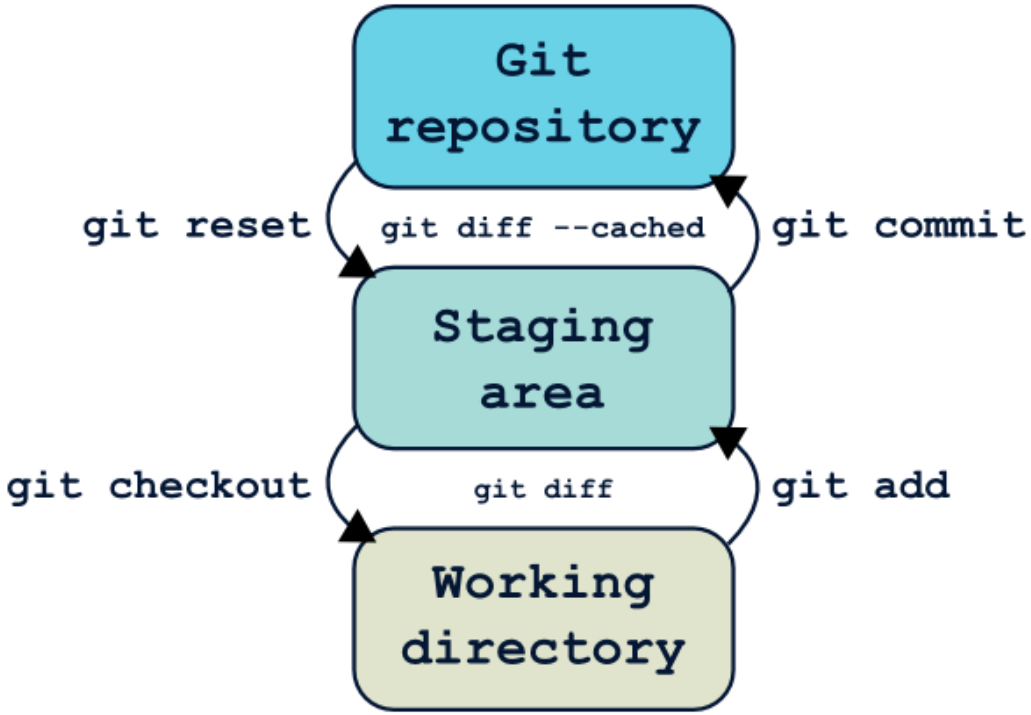


FIGURE – Manipulations de fichiers entre zones de git [1]

1. schéma de Thibault Jouannic : https://www.miximum.fr/blog/enfin-comprendre-git/

Annuler des modifications

Sur un ensemble de modifications en cours

- Changer la HEAD, équivalent à un "go to" : git reset *SHA1* –soft
- Changer la HEAD et réinitialiser la staging area à cette version : git reset *SHA1* –mixed (option par défaut)
- Changer la HEAD et réinitialiser la stating area et le working dir à cette version : git reset *SHA1* –hard

Sur un **commit**

- Annuler un commit en créant un nouveau commit annulant les modifications du premier : git revert *SHA1*
- Modifier (corriger) un commit avec les modifications courantes : git commit –amend

Gestion des branches

Création et navigation

Les branches permettent de réaliser un développement en parallèle de la branche principale afin de limiter les impacts.

- Lister les branches existantes : git branch
- Créer une nouvelle branche : git branch *nom_branche* (*nom_racine ou SHA1*)
- Changer la branche active : git checkout *nom_branche* (revient à déplacer la référence HEAD vers le commit le plus récent de cette branche)
- Revenir sur la branche principale : git checkout master
- Placer HEAD sur un commit particulier : git checkout *SHA1*

Gestion des branches

Regroupement et suppression

Les branches ont vocation à être réintégrées à la branche principale.

- Inclus dans la branche courante les modifications de la branche nommée : git merge *nom_branche* (génère un commit avec deux parents, un dans chaque branche)
- Supprimer une branche : git branch -d *nom_branche*

Gestion des branches

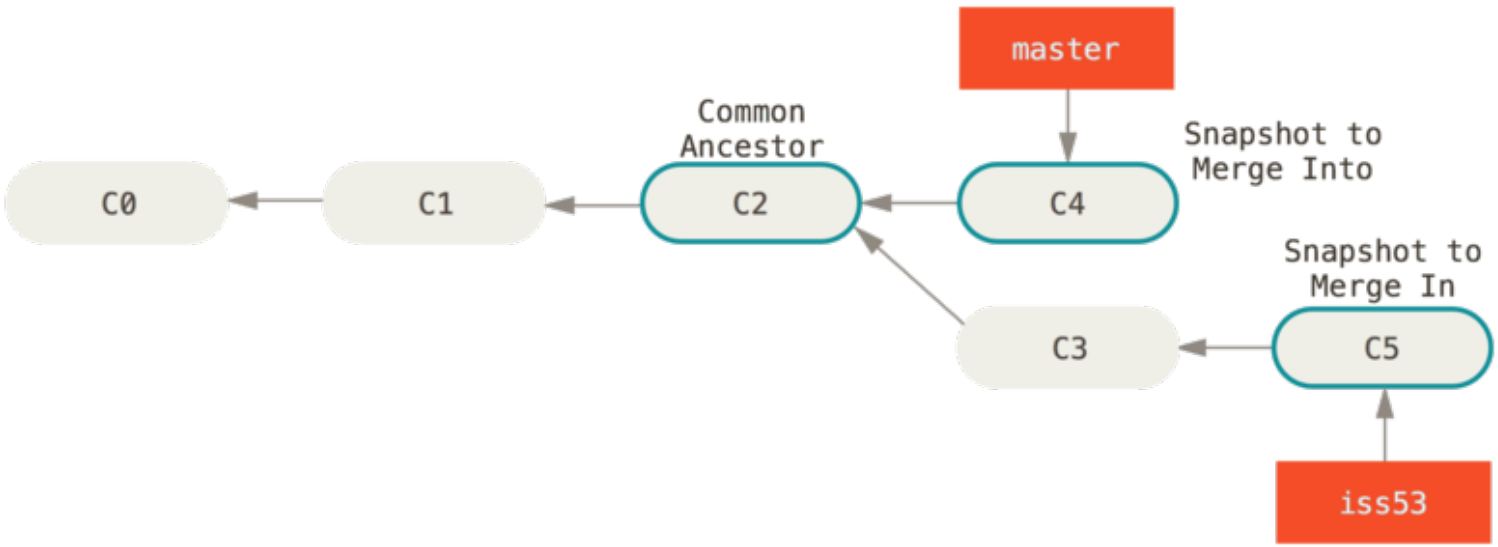


FIGURE – Branches divergentes avant fusion [1]

Gestion des branches

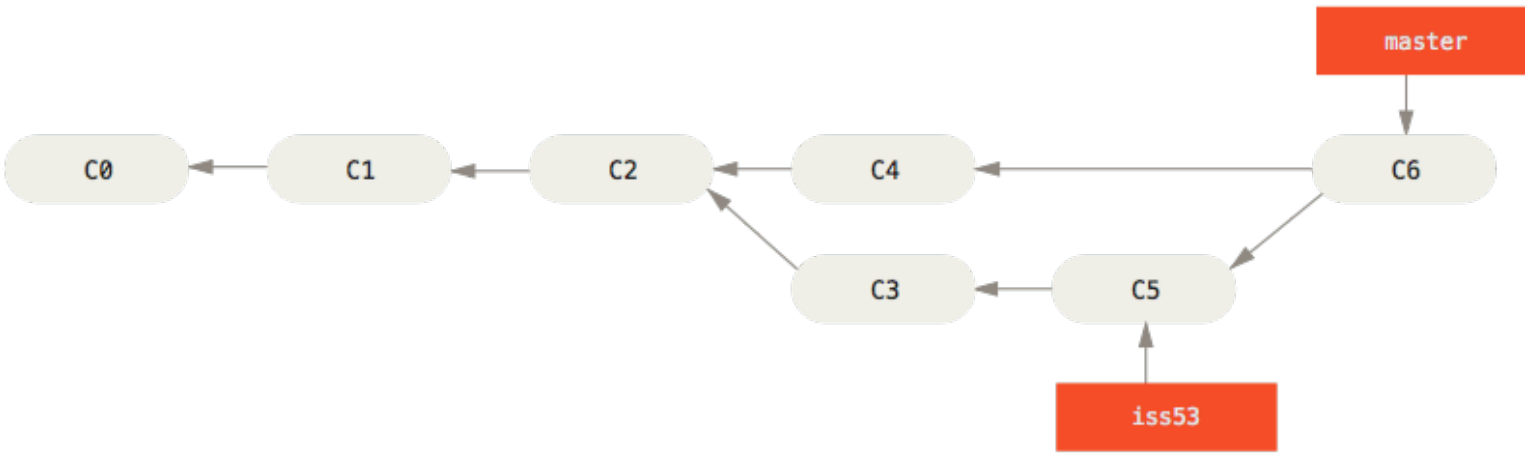


FIGURE – Branches après fusion [1]

Gestion des conflits

Dus à des modifications différentes au même endroit dans un même fichier.

Deux choix

Git entre en mode résolution des conflits. Deux manières d'en sortir :

- Ne pas fusionner : git merge –abort
- Résoudre les conflits :
  - Éditer les fichiers sources de conflits
  - Terminer la résolution par git commit

Gestion des conflits

Commandes utiles pour visualiser les conflits

- Afficher les conflits au format diff3 (permet de visualiser l'ancêtre commun, ce qui aide à la résolution) : git config –global merge.conflictstyle diff3
- Si certains conflits récurrents, réutiliser résolution passée (reuse recorded resolution) : git config –global rerere.enabled 1
- Référence pratique : MERGE_HEAD pointe sur la tête de la branche distante à fusionner
- Comparer les versions HEAD et MERGE_HEAD : git diff

Gestion des conflits

Commandes utiles pour visualiser les conflits

- Trouver l'ancêtre commun : git merge-base *nom_branche_distante nom_branche_locale*
- Voir les différences : git diff *SHA1_ancêtre nom_branche_distante fichier*
- git log *SHA1_ancêtre nom_branche_distante* –follow -p – *fichier*
- git log *SHA1_ancêtre* HEAD –follow -p – *fichier*

Gestion des conflits

Les stratégies automatiques de résolutinon

- Résoudre automatiquement les conflits avec une stratégie pré-définie : git merge -s *stratégie*
- Toujours favoriser notre version : ours
- Toujours favoriser l'autre version : theirs

Gérer le graphe des commits

Transplantation de branches (rebase)

Permet d'intégrer les modifications d'une branche dans une autre afin d'éviter un merge par fusion : donne l'illusion d'un développement linéaire a posteriori.

- Intégreation progressive des commits précédents à la *head* : git rebase -i *SHA1* (du commit à partir duquel commencer)
- Pour chaque commit parcouru, 3 options :
  - pick : commit directement appliqué à la head
  - edit : commit appliqué et modification possible avant l'intégration du suivant
  - squash : commit appliqué et fusionné avec le précédent

Réécrit l'histoire du dépôt. A faire uniquement localement. En cas de conflit git rebase –continue (après résolution) ou git rebase –abort.

Transplantation de commits

État du graphe :

```
- A - B - C - D - remote HEAD
    \
      E - F - G - local HEAD
```

Après un merge :

```
- A - B - C - D - remote HEAD
    \                        \
      E - F - G - local HEAD - new local HEAD
```

Après un rebase :

```
- A - B - C - D - remote HEAD - E' - F' - G' - local HEAD'
```

Gérer le graphe des commits

Pull and rebase

- git pull est équivalent à git fetch + git merge
- git pull –rebase est équivalent git fetch + git rebase
  - Réécrit les commits locaux intermédiaires de manière à s'inscrire dans la continuité de l'autre branche (peut faciliter la résolution des conflits) :
    git pull –rebase (*nom_branche_distante nom_branche_locale*)

Gérer le graphe des commits

Squashing

Permet de regrouper plusieurs commits successifs d'une branche avant un merge.

- git merge –squash
- git pull –squash

Squashing et Rebasing : simplifier le graphe de commits pour laisser le dépôt principal propre et lisible (anyway, toujours suivre la politique de l'entreprise si définie)

Autres commandes utiles

Stash (stash)

- Sauvegarde puis annule les modifications du working dir et de l'index pour les ré-appliquer plus tard : git stash
- Lister les modifications sauvegardées : git stash list
- Inspecter les modifications sauvegardées : git stash show
- Restaurer : git stash apply

Autres commandes utiles

Et bien d'autres...

- Étiqueter une version (copie l'arbre et interdit toute modification) : git tag *nom_tag* (*SHA1*)
- Intégrer à la branche courante les modifications d'un commit précis : git cherry-pick *sha1*
- Recherche dichotomique pour trouver l'origine d'un bug entre un *good* et un *bad* commit : git bisect

Attention

- Git est très puissant mais un grand pouvoir implique...
- Certaines commandes dangereuses
- ex : git push –force
- Ne **jamais** utiliser de nouvelles commandes sans s'être documenté

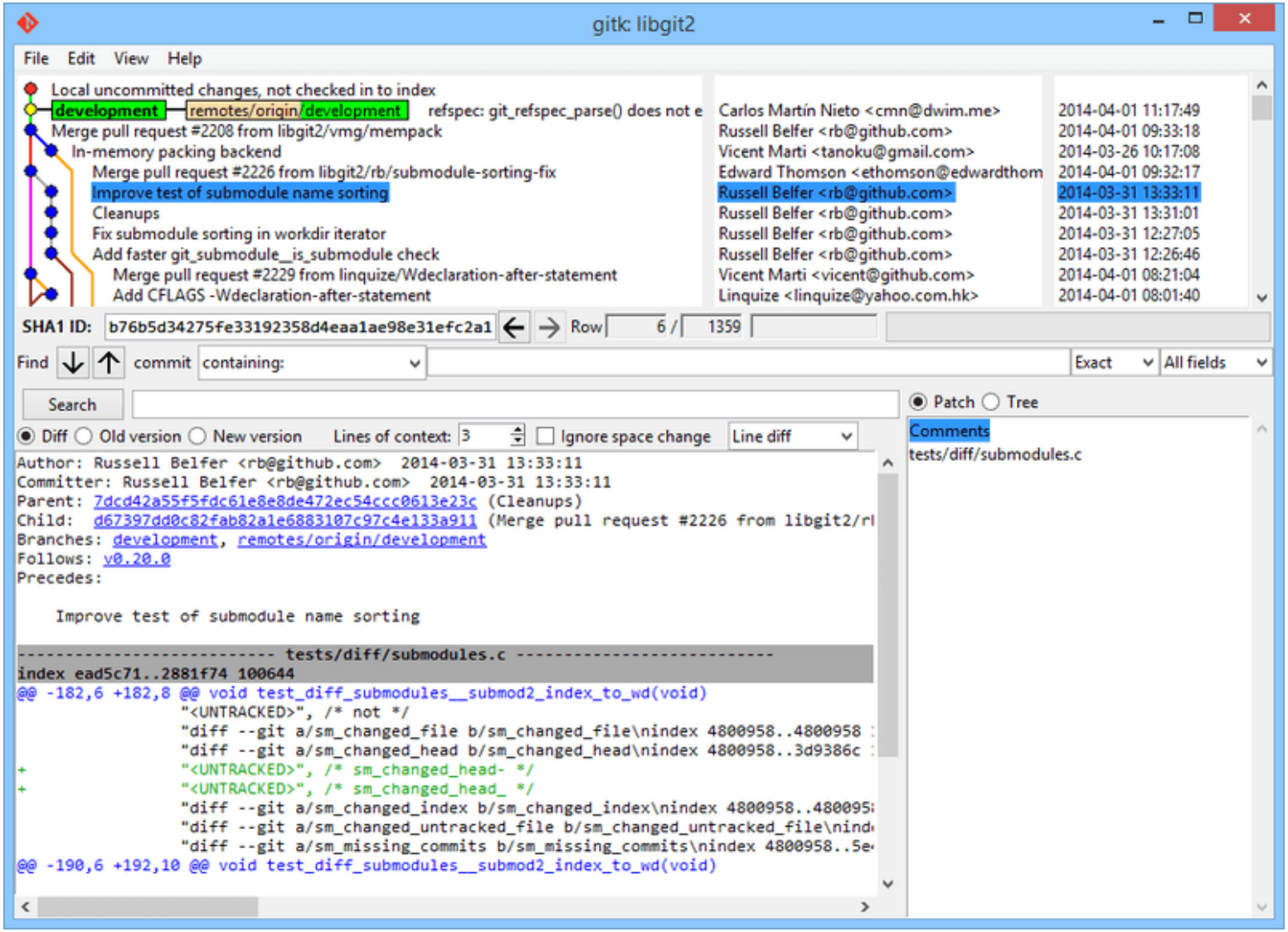
Outline

Outils graphiques

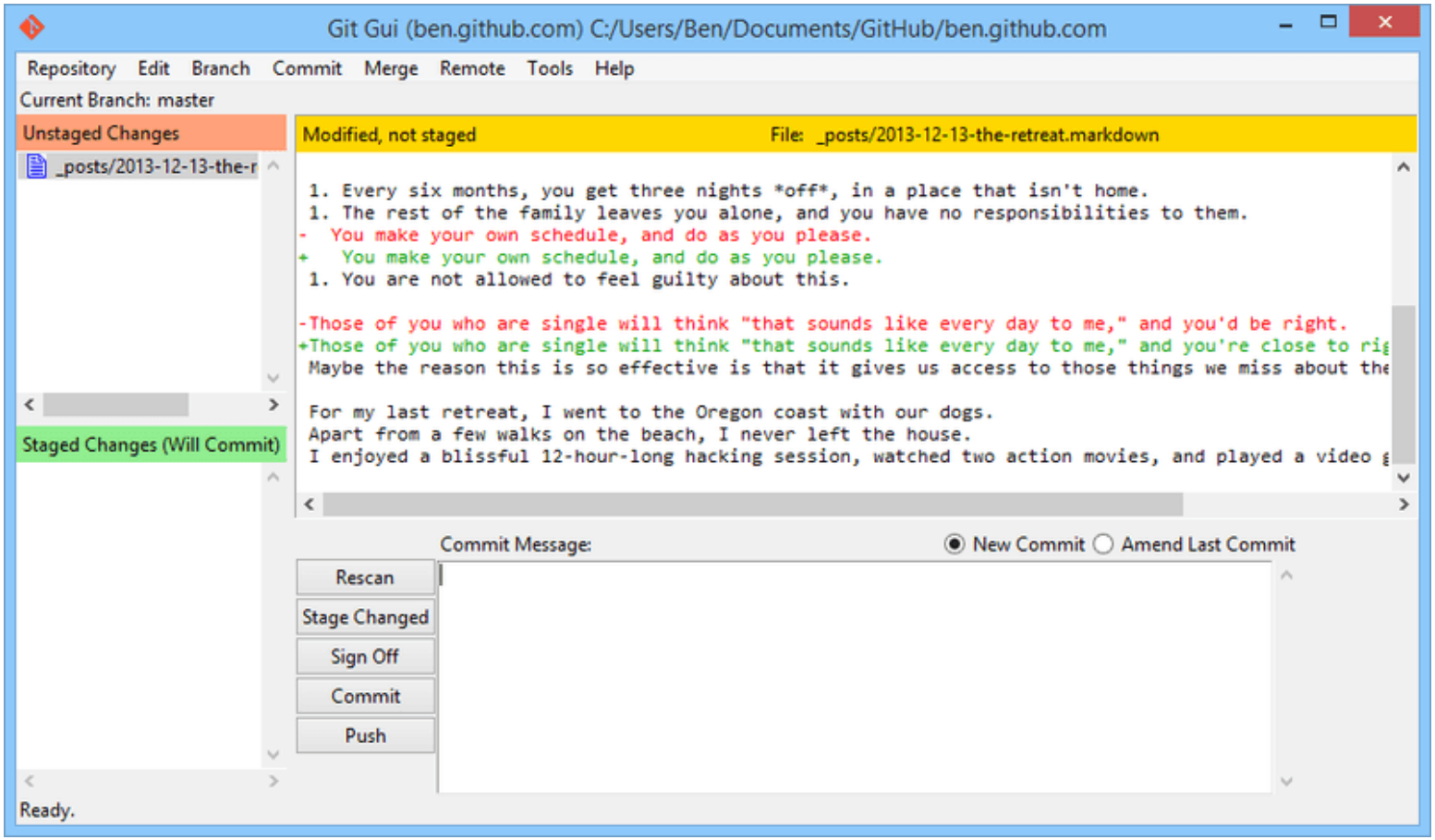
Gérer les arbres à l'aide des commandes peut se révéler délicat. Certains outils graphiques peuvent aider mais sont toujours plus limités que la console :

- gitk ou gitg : simple interface graphique qui permet de visualiser l'historique (branches et commits). Chaque commit peut être visualisé (liste des fichiers modifiés, changements, etc.).
- git-gui : client complet permettant de préparer ses commits
- git-cola : client git graphique (pull, push, commit, diff-viewer, etc.)
- Smart Git : client git complet, nombreuses fonctionnalités, cross-platform
- Giggle : interface de visualisation, équivalent à gitk

Outils graphiques



Outils graphiques



Outils en ligne

Services de dépôts en ligne

GitHub, BitBucket, Gitorious, GitLab, etc.

Plateformes d'hébergement

GitLab, FusionForge, etc.
Offrent souvent beaucoup d'autres services : sites web, wiki, mailing lists, forums, bugtrackers, gestion des releases etc.

Organisation du travail

Plusieurs workflow possibles

- Force de git : souplesse dans l'organisation des workflow
- Avec/sans dépôt commun distant
- Avec/sans intégrateur (notion de pull request)

Organisation du travail

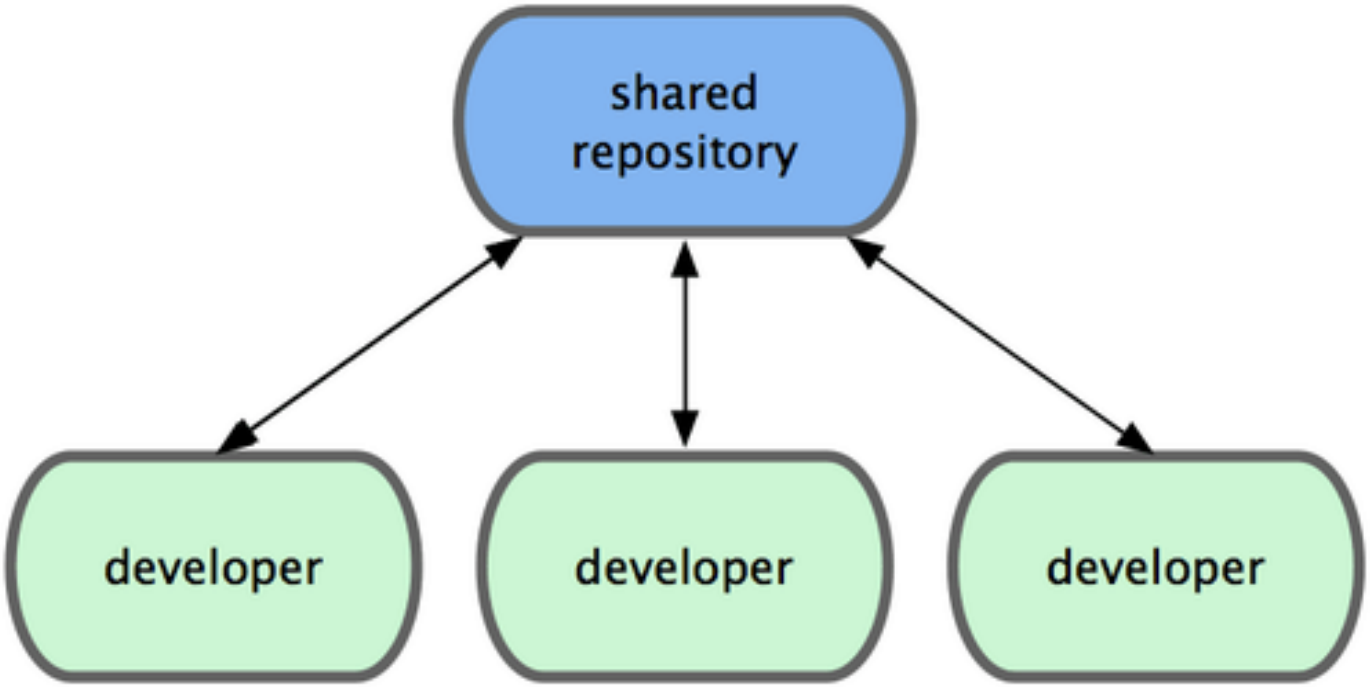


FIGURE – Workflows possibles [1]

Organisation du travail

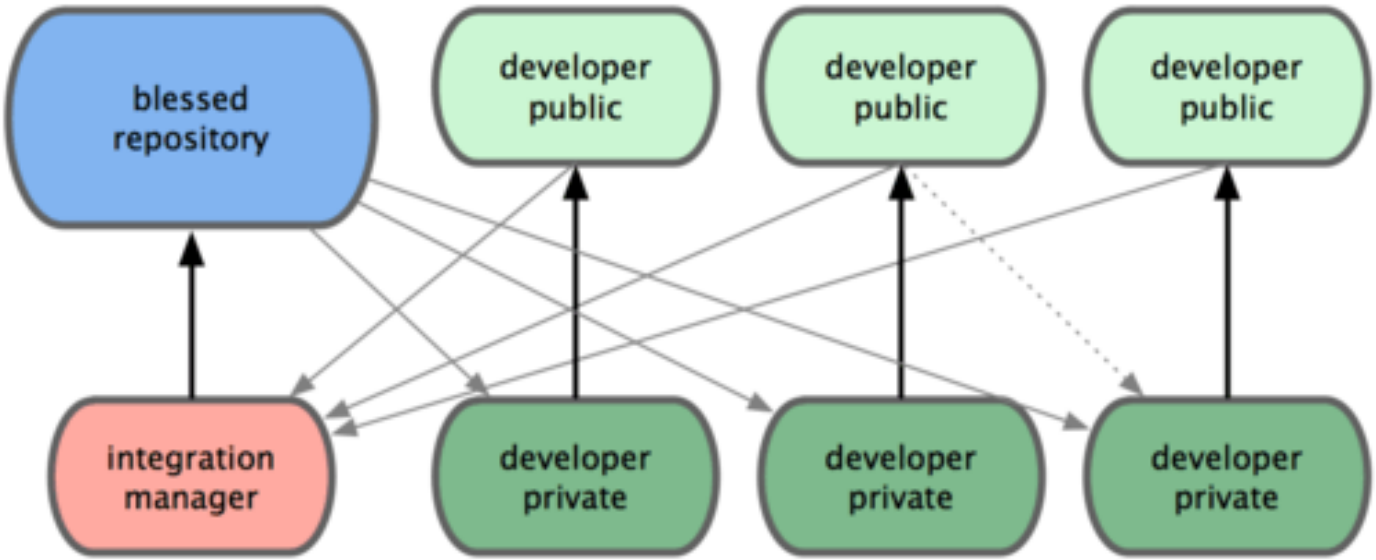
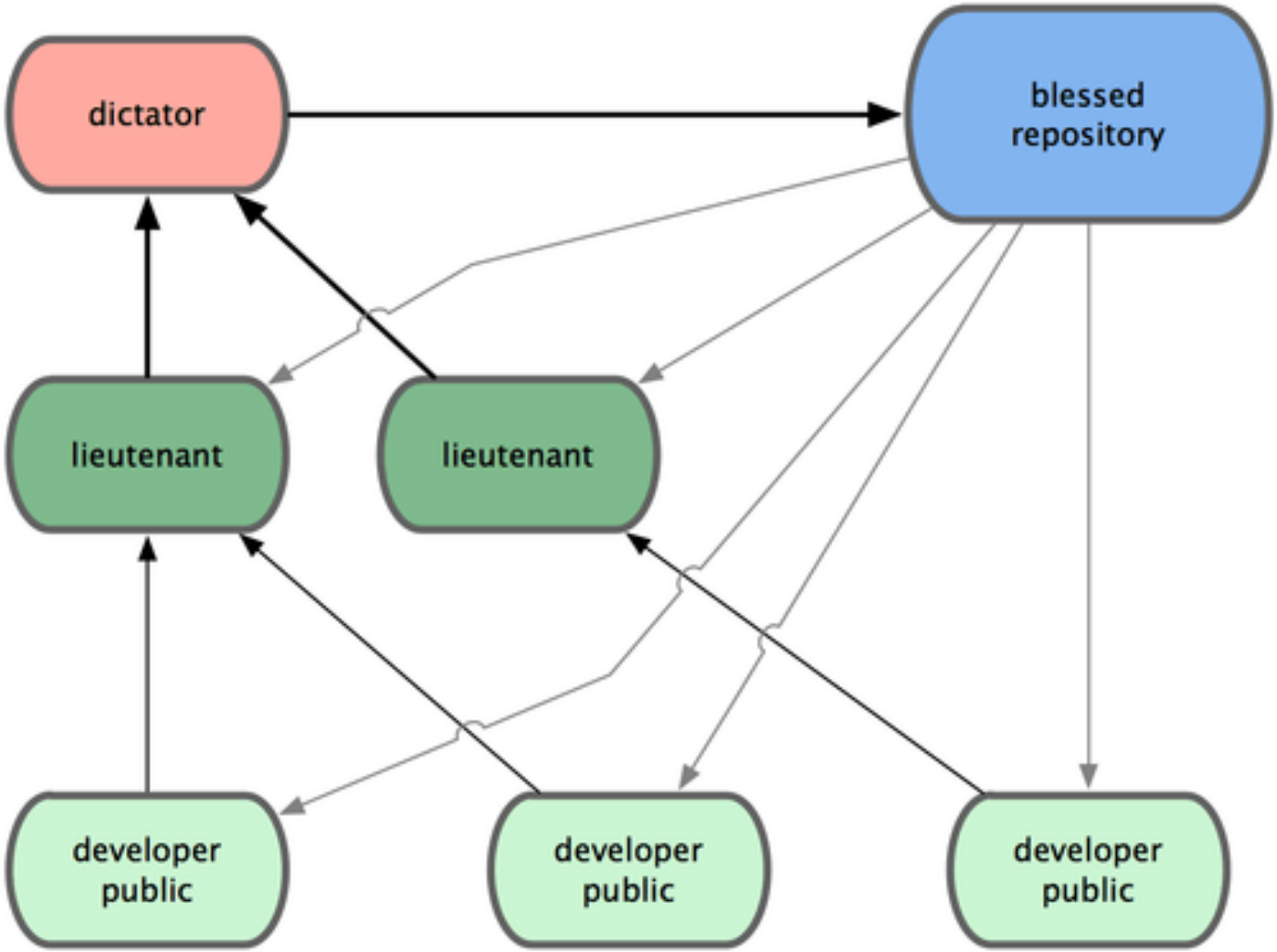


FIGURE – Workflows possibles [1]

Organisation du travail



Organisation du travail

Github Workflow

- Anything in the master branch is deployable
- To work on something new, create a descriptively named branch off of master
- Commit to that branch locally and regularly push your work to the same named branch on the server
- When you need feedback or help, or you think the branch is ready for merging, open a pull request
- After someone else has reviewed and signed off on the feature, you can merge it into master
- Once it is merged and pushed to master, you can and should deploy immediately

Outline

Besoins

Étapes d'un projet de développement logiciel

1. Offre (proposition, budgétisation, présentation),
2. Spécifications fonctionnelles et planification (chef de projet)
3. Gestion des ressources humaines (chef de projet, RH),
4. Spécifications techniques (architecte logiciel), gestion des ressources matérielles
5. Mise en place des environnements (devops)
6. Écriture du code source, tests unitaires, tests d'intégration (développeurs)
7. Tests fonctionnels (équipe dédiée)
8. Corrections des erreurs (développeurs)
9. Livraison, mise en production (devops / ingénieur système)

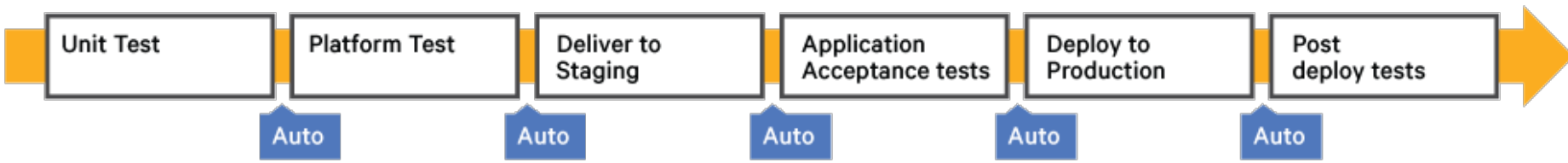
Besoins

Production logicielle

Besoin de traçabilité : par exemple, lien entre un bug identifié et un commit le corrigeant
Besoin d'automatisation :

- Raccourcir les cycles de développement (points 6,7,8, voire 9) (SDLC automation "Software Development Life Cycle")
- Éviter les fautes, pouvoir corriger rapidement
- Dégager du temps utile

Procédés actuels : intégration continue "CI" (voire même livraison continue "CD"). Similaire à la production industrielle en flux tendu.



L'intégration continue

Procédure automatique

- Les développeurs enregistrent les modifications stables sur une branche du SCM
- Le serveur d'intégration continue réalise alors :
  1. récupération du code source depuis le SCM et compilation
  2. récupération des configurations des environnements depuis le SCM
  3. exécution des tests unitaires
  4. exécution des tests d'intégration
  5. création d'un build
  6. déploiement
- report automatique de tout problème dans le processus
- le SCM est un élément essentiel de la chaîne de CI

Exemples d'outils de CI

- Maven / Gradle : gestion et automatisation de production de logiciels Java (description du projet et configuration en XML, gestion des dépendances avec téléchargement automatique, etc.)
- Jenkins : build et tests automatiques de projets
- Docker : automatise le déploiement d'applications dans des conteneurs logiciels dont la construction est paramétrable. Permet de déployer dans un environnement isolé et facilement partageable.
- Ansible : déploiement de logiciels sur clusters et gestion des configurations
- Puppet : gestion de la configuration de serveurs, déploiement de VM et d'applications

Git à TELECOM Nancy

GitLab

- GitLab : logiciel libre de forge (GitLab Community Edition)
- Déployé à l'école : `https://gitlab.telecomnancy.univ-lorraine.fr/`
- Login : prenom.nom@telecomnancy.eu, Mdp : LDAP
- Ne pas confondre avec GitLab.com : le service de forge en ligne, basé sur GitLab Enterprise Edition (similaire à GitHub ou BitBucket)
- Utilisé en entreprise (IBM, Sony, CERN, SpaceX, etc.)

Git à TELECOM Nancy

Fonctionnalités de GitLab

- Dépôt git (of course) avec inferface web, mais pas que
- Outils collaboratif : gestion des membres, Mattermost (slack-like), revue de code, etc.
- Outils de suivi : tableaux de bord, gestion des tickets/issues, des versions, des releases, bug tracker, statistiques, etc.
- Outils de CI : recettes Docker, routines et tests automatiques, etc.
- Outils pour la communauté : pages web, wiki

Voir `https://about.gitlab.com/features/`

Quelques liens

Généraux :

http://git-scm.com/
http://git-lectures.github.io/
https://guides.github.com/introduction/flow/
https://git.wiki.kernel.org/
https://www.miximum.fr/blog/enfin-comprendre-git/
http://people.irisa.fr/Anthony.Baire/

Questions spécifiques :

https://www.quora.com/How-does-Git-merge-work
https://www.miximum.fr/blog/git-rebase/
http://clubmate.fi/
git-dealing-with-branches-merging-and-rebasing/
https://www.atlassian.com/git/tutorials/
resetting-checking-out-and-reverting/

Encore des liens

Mémos :
`http://ndpsoftware.com/git-cheatsheet.html`
`http://gitref.org/`
Tutos :
`http://try.github.io/levels/1/challenges/1`
`https://blog.udemy.com/git-tutorial-a-comprehensive-guide/`
`http://gitimmersion.com/`
gittutorial et gittutorial-2(man page)

References

[1] S. Chacon and B. Straub, "Pro git."
http ://git-scm.com/book/fr/, 2014.

[2] L. Nussbaum, "Gestion de versions avec git."
https ://members.loria.fr/LNussbaum/asrall/o5-git.pdf, 2013.

[3] B. Lynn, "Git magic."
http ://www-cs-students.stanford.edu/ blynn/gitmagic/, 2010.

[4] N. Villa, "Introduction à git."
http ://nathalievilla.org/spip.php ?article91, 2013.