Les logiciels de gestion de version
exemple de GIT et application avec SmartGIT

Rémi SHARROCK
www.remisharrock.fr

Plan du polycopié

- Plan du polycopié
- Problématique
- La gestion de versions: généralités, vocabulaire
  - Version, révisions et modifications, différences (deltas, diff, patch)
  - Modifications et ensemble de modifications
  - Étiquetage ou marquage de version (tag)
  - Dépôt et copies locales
  - Soumissions, conflits, contrôle de conflits et résolution de conflit
    - éviter les conflits
    - résoudre les conflits
  - Arbre de gestion de versions, Tronc, Branches, Fusion de branches (merges)
  - Gestion de versions centralisée et décentralisée
    - Gestion de versions centralisée
    - Gestion de versions décentralisée
  - Les différents logiciels de gestion de versions
    - CVS, centralisé
    - SVN (Subversion), centralisé
    - Mercurial, décentralisé
    - Bazaar, décentralisé
    - Git, décentralisé
- Exemple de GIT
  - Particularités de Git
  - Vocabulaire spécifique à GIT
  - Ordre de création des dépôts, clonage de dépôt et fork de dépôt.
  - Collaboration de deux personnes avec GIT
  - Le forking workflow
  - GIT et les branches: commit, checkout, HEAD, merge, rebase, reset.
  - GIT et les conflits
    - Commiter des modifications qui sont en relation les unes des autres
    - Tester le code avant de commiter
    - Utiliser les branches
    - Commiter souvent
    - Ecrire de bonnes descriptions dans les commits

Ne pas commiter un travail à moitié fait.
GIT n'est pas un système de sauvegarde/backup.

Problématique

Imaginons la gestion archaïque d'un projet faisant intervenir plusieurs personnes travaillant avec de multiples fichiers dans de multiples dossiers sans gestionnaire de versions:

- Les personnes travaillant sur plusieurs ordinateurs utilisent un support amovible (clé USB, un disque dur externe....) pour avoir une copie des fichiers/dossiers sur lesquels travailler.
- Certaines personnes utilisent un service de synchronisation dans le cloud (Dropbox, SugarSync...) pour synchroniser leurs fichiers/dossiers sur tous les ordinateurs.
- Pour partager leur travail, les personnes s'échangent les clés usb, s'envoient les fichiers par courrier électronique, compressent un dossier en ZIP pour l'envoyer, ou utilisent des dossiers partagés (Google Drive, SkyDrive...).

Voici un exemple de problèmes engendrés par cette gestion archaïque:

- Le disque dur tombe en panne, la clé USB tombe dans les toilettes: destruction de la dernière version des fichiers/dossiers. Il serait préférable de conserver la dernière copie en lieu sûr.
- Une personne veut partager par courrier électronique la dernière version d'un très gros fichier, malheureusement il est trop volumineux et ne "passe par par courrier électronique". Il serait préférable d'envoyer uniquement la différence (modifications, ajouts/suppressions) sûrement plus petite en volume que le fichier entier.
- On voudrait savoir qui a supprimé un fichier important qui reste maintenant introuvable. Il serait préférable de garder un historique de tout se qui se passe et de qui a effectué des modifications (création/suppression/modification de fichiers). Il serait également préférable d'avoir des copies de secours et de connaître le "pourquoi" des changements.
- On voudrait retrouver un dossier dans l'état qu'il était il y a deux mois. Il serait préférable d'avoir des historiques non seulement des fichiers mais des dossiers entiers et de leur structuration.
- Quatre personnes ont travaillé sur un même fichier en même temps mais sur des parties différentes, ils perdent un temps fou à savoir comment fusionner leurs fichiers. Il serait préférable d'avoir une fusion automatique.
- Deux personnes ont travaillé en même temps sur le même fichier et au même endroit, ils doivent refaire le travail ensemble. Il serait préférable d'avoir un outil pour l'aide à la gestion de conflits dans un fichier.

On pourrait multiplier ainsi les problèmes et cela justifie l'utilisation d'un **logiciel de gestion de versions**. Ce type de logiciel est devenu indispensable lorsqu'on travaille à plusieurs sur un même projet avec des multiples fichiers et dossiers. Même si vous travaillez seuls, il vous offrira

de nombreux avantages, comme la conservation d'un historique de chaque modification des fichiers par exemple. Dans le cas d'un développement en équipe, surtout si elles sont réparties dans le monde entier, il est nécessaire de partager une base commune de travail, et c'est tout l'intérêt des systèmes de gestion de version. Mais, il faut aussi veiller à coordonner les équipes de développement grâce à des outils de communication, un logiciel de suivi de problèmes, un générateur de documentation et/ou un logiciel de gestion de projets.

Concernant la gestion de version il existe de nombreux logiciels comme CVS, SVN (aussi appelé Subversion), Mercurial ou Git. Ces logiciels suivent l'évolution de vos fichiers, gardent les anciennes versions de chacun d'eux, retiennent qui a effectué chaque modification de chaque fichier et pourquoi (justification des modifications). Ils sont par conséquent capables de dire qui a effectué chaque modification, à quel endroit du fichier et dans quel but. Si deux personnes travaillent simultanément sur un même fichier, ils sont capables d'assembler (de fusionner) leurs modifications et d'éviter que le travail d'une de ces personnes ne soit écrasé.

Nous illustrerons les propos avec l'utilisation de **Git** (prononcez « guite ») qui est un des plus récents et l'un des plus puissants logiciels de ce genre.

La gestion de versions: généralités, vocabulaire

La gestion de versions[1] consiste à maintenir l'ensemble des versions d'un projet constitué de plusieurs fichiers (généralement en texte) et dossiers. Essentiellement utilisée dans le domaine de la création de logiciels, elle concerne surtout la gestion des codes source.

Cette activité étant fastidieuse et relativement complexe, un appui logiciel est presque indispensable. À cet effet, il existe différents logiciels de gestion de versions qui, bien qu'ayant des concepts communs, apportent chacun leur propre vocabulaire et leurs propres usages. À titre d'exemple, on trouve un mécanisme rudimentaire de gestion de versions dans Wikipédia : pour chaque article, l'historique est disponible en cliquant sur le lien Afficher l'historique ; chaque ligne est une version de l'article. Un tel système est linéaire, par opposition à une gestion de contenu plus élaborée, selon une structure arborescente.

Version, révisions et modifications, différences (deltas, diff, patch)

Les logiciels évoluant, chaque étape d'avancement est appelée version. Les différentes versions sont nécessairement liées à travers des modifications : une modification est un ensemble d'ajouts, de modifications et de suppressions de données (fichiers/dossiers). Il y a donc un delta, une différence entre deux versions appelée "diff" ou "patch".

[1] en anglais version control ou revision control

Schématiquement, le passage de la version N à la version N + 1 engendre un delta, une différence (un “diff”) et on pourra passer de la version N à la version N+1 en appliquant une modification M correspondant à cette différence (en appliquant un “patch”). Un logiciel de gestion de versions applique ou retire ces modifications une par une pour fournir la version voulue.

On préfère parfois le terme « révision » afin de ne pas confondre la version d'un fichier et la version d'un logiciel, qui est une étape de distribution sous forme "finie", c'est-à-dire principalement binaire.

Modifications et ensemble de modifications

Une modification constitue donc l'évolution entre deux versions. On peut aussi bien parler de la différence entre deux versions que de modifications ayant amené à une nouvelle version.

On utilise généralement la gestion de versions à un ensemble de fichiers qui constitue un projet. De ce fait, il est courant de parler de modifications pour un seul fichier et d'ensemble de modifications lorsqu'il s'agit du projet (et donc de plusieurs fichiers). En effet, les deux n'évoluent pas au même rythme.

Pour illustrer, prenons l'exemple d'un logiciel nommé « Toto ». Il est constitué des fichiers A, B et C. À la version 1.0 de « Toto » correspondent les versions 1.0 de chacun des fichiers. Admettons que l'ajout d'une fonctionnalité à « Toto » impose la modification de A et de C. Présentons la situation à l'aide d'un tableau

Versions de “Toto”	Versions de A	Versions de B	Versions de C
1.0	1.0	1.0	1.0
1.1	1.1	1.0	1.1

Du point de vue du projet, les modifications apportées à A et à C font partie du même ensemble.

Étiquetage ou marquage de version (tag)

L’étiquetage (tag) consiste à associer un nom à une version donnée. Pour certains outils de gestion de versions (comme CVS) qui gèrent les versions à une faible granularité (c’est à dire avec beaucoup de modifications non significatives), c'est un moyen de retrouver facilement une version significative à l’aide de son nom (recherche par tag). Un exemple typique serait le nom des versions d’Android ou de MAC OS X (Puma, Jaguar, Panther, Tiger, Leopard, Lion...).

Dépôt et copies locales

Quand un fichier est géré par un gestionnaire de version on dit de lui qu'il est *versionné*. Les fichiers *versionnés* sont mis à dispositions sur un dépôt, c'est-à-dire un espace de stockage partagé entre les collaborateurs du projet et géré par un logiciel de gestion de versions.

Pour pouvoir effectuer des modifications, la personne doit d'abord faire une copie locale (sur son ordinateur de travail) des fichiers qu'il souhaite modifier, ou de tout le dépôt (on parle de clonage de dépôt). Selon les systèmes de gestion de version, certains fichiers peuvent être verrouillés ou protégés en écriture pour tout le monde, ou pour certaines personnes.

Soumissions, conflits, contrôle de conflits et résolution de conflit

En général une personne fait des modifications localement (sur son ordinateur de travail), indépendamment des modifications faites sur le dépôt du fait du travail simultané d'autres personnes. Il doit ensuite faire une soumission, c'est-à-dire soumettre ses modifications, afin qu'elles soient enregistrées sur le dépôt. C'est là que peuvent apparaître des conflits entre ce que la personne souhaite soumettre et les modifications effectuées par d'autres personnes.

Il n'est en effet pas rare que certaines modifications soient contradictoires (lorsque deux personnes ont apporté des modifications différentes à la même partie d'un fichier). On parle alors de conflit de modifications car le logiciel de gestion de versions n'est pas en mesure de savoir laquelle des deux modifications il faut appliquer. Ces conflits doivent absolument être résolus pour que la modification soit acceptée sur le dépôt. En général, si un conflit persiste, aucune modification ne peut être acceptée par la personne tant qu'elle ne résout pas le conflit. Les outils de gestion de version peuvent aider à cette résolution de conflits.

éviter les conflits

Pour éviter ces conflits de modifications on peut faire du contrôle de conflits pessimiste. C'est un problème classique en informatique: on le retrouve par exemple dans les systèmes de gestion de base de données ou en programmation système. Le contrôle de conflits qui évite les conflits est dit pessimiste car il impose à chaque personne de demander un verrou avant de modifier une ressource ; ce verrou lui garantit qu'il sera le seul à modifier la ressource. Ce modèle s'impose quand on considère que le coût de résolution des conflits de modification (coût unitaire pondéré par leur probabilité d'occurrence) est plus important que celui de la gestion du verrou. En gestion de version, il correspond au modèle "verrouiller-modifier-déverrouiller" qui était utilisé par les systèmes les plus anciens. Il s'est avéré que la gestion manuelle des verrous par les utilisateurs n'était pas toujours satisfaisante : les outils de résolution de conflits s'améliorant, il est progressivement devenu moins pénalisant de corriger les conflits d'éditions simultanées que de traiter les problèmes de fichiers verrouillés en écriture.

résoudre les conflits

La résolution de conflits est dite optimiste car elle permet à chaque personne de modifier les données sans contrainte. Au moment d'appliquer ces modifications le système vérifie si une autre personne n'a pas déjà soumis des modifications pour ces mêmes données (soumission

qui a été acceptée dans le dépôt). S'il y a conflit, il demande alors à la personne de le résoudre avant de re-soumettre ses données. En gestion de version, c'est le modèle "copier-modifier-fusionner" qui a été popularisé par CVS.

Arbre de gestion de versions, Tronc, Branches, Fusion de branches (merges)

Il faut s'imaginer la gestion de version comme un arbre (appelé arbre de gestion de versions) avec un tronc, des branches et des sous-branches.

Par défaut, l'évolution d'un projet au travers de l'historique de ses versions est un phénomène linéaire :

la personne A apporte ses modifications sur le projet,

B ajoute ses fonctionnalités,

C corrige le code de A

etc.

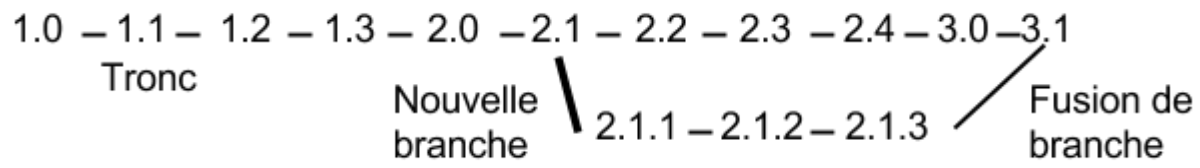
On parle alors de Mainline ou de tronc.

Cependant, au cours de l'évolution d'un logiciel, il se peut que cette MainLine doive à la fois suivre son cours, mais aussi répondre à une contrainte qui va amener le projet à suivre une évolution parallèle à la MainLine. On a besoin alors de créer une branche.

Une branche est une dérivation dans l'histoire de l'évolution du projet. C'est une évolution ayant pour origine une version précise, produisant une "branche de version". Une branche de version correspond à un axe d'évolution de versions, c'est à dire qu'elle est rattachée à une branche source et peut découler sur plusieurs sous-branches. Au sein d'une branche, l'évolution des versions se fait de façon linéaire, comme pour la MainLine ou le tronc, c'est-à-dire que les versions se suivent chronologiquement. L'apport des branches permet donc:

- la maintenance d'anciennes versions du logiciel (sur les branches) tout en continuant le développement des futures versions (sur le tronc) ;
- le développement parallèle de plusieurs fonctionnalités volumineuses sans bloquer le travail quotidien sur les autres fonctionnalités.

Par exemple, le logiciel « Toto » a subi plusieurs évolutions linéaires depuis sa première version, le tronc de l'arbre. À partir de la version 2.1, ses concepteurs décident de publier une version qui devient vite populaire. Ils continuent à ajouter des fonctionnalités jusqu'à atteindre la version 3.0 du tronc, qui changera fondamentalement le logiciel. Peu de temps après que cette dernière version soit sortie, ils sont amenés à devoir corriger des problèmes dans la version 2.1 encore très utilisée; il est alors nécessaire de créer une nouvelle branche. On peut imaginer que plus tard, certaines modifications dans cette nouvelle branche soient intégrées au tronc, on parle alors de fusion de branche. Une fusion de branche n'est pas nécessairement faite avec le tronc mais entre deux branches ou deux sous-branches.



La fusion de branches (merge) consiste à combiner des modifications ou des suites de modifications (qu'elles viennent de branches différentes ou non) pour créer une nouvelle version. Dans l'exemple on a fusionné la version 2.1.3 avec la version 3.0 pour créer la version 3.1.

On peut décliner plusieurs intérêts à la fusion de branches:

- la synchronisation entre plusieurs personnes, qui travaillent habituellement séparément, donc sur des branches de version différentes
- l'annulation d'une ancienne modification (par opposition à la toute dernière), à appliquer sur la version actuelle
- l'importation d'une modification d'une branche vers une autre.

La fusion de branches engendre souvent des conflits et il faudra obligatoirement appliquer la résolution de conflits pour terminer la fusion.

Gestion de versions centralisée et décentralisée

Gestion de versions centralisée

Avec les logiciels de gestion de versions centralisés, comme CVS et Subversion (SVN), il n'existe qu'un seul dépôt qui fait référence.

Cela simplifie la gestion mais est contraignant pour certains usages comme le travail sans connexion au réseau, ou tout simplement lorsque l'on travaille sur des branches expérimentales ou contestées.

Gestion de versions décentralisée

La gestion décentralisée consiste à voir le logiciel de gestion de versions comme un outil permettant à chacun de travailler à son rythme, de façon désynchronisée des autres, puis d'offrir un moyen à ces personnes de s'échanger leur travaux respectifs. De fait, il existe plusieurs dépôts pour un même logiciel. Ce système est très utilisé par les logiciels libres.

Avantages de la gestion décentralisée :

- permet de ne pas être dépendant d'une seule machine comme point de défaillance ;
- permet aux contributeurs de travailler sans être connecté au gestionnaire de version ;
- permet la participation à un projet sans nécessiter les permissions par un responsable du projet (les droits de soumission peuvent donc être donnés après avoir démontré son travail et non pas avant) ;

- la plupart des opérations sont plus rapides car réalisées en local (sans accès réseau) ;
- permet le travail privé pour réaliser des brouillons sans devoir soumettre ses modifications et gêner les autres contributeurs ;
- permet toutefois de garder un dépôt de référence contenant les versions livrées d'un projet.

Un désavantage peut être que récupérer l'arbre complet de gestion de versions est plus long que de récupérer une version car tout l'historique est copié.

Dans la pratique, les logiciels distribués sont capables de fonctionner dans ce mode distribué (plusieurs dépôts) mais on utilise très souvent un seul serveur qui sert de point de rencontre entre les personnes. Le serveur connaît l'historique des modifications et permet l'échange d'informations entre les personnes, qui eux possèdent également l'historique des modifications.

Pas besoin de faire de sauvegarde du serveur étant donné que tout le monde possède l'historique des fichiers, et le serveur simplifie la transmission des modifications.

C'est dans ce dernier mode que nous allons fonctionner avec Git.

Les différents logiciels de gestion de versions

CVS, centralisé

C'est un des plus anciens logiciels de gestion de versions. Bien qu'il fonctionne et soit encore utilisé pour certains projets, il est préférable de ne pas l'utiliser car il est dans l'incapacité à suivre les fichiers renommés par exemple, il est le moins puissant et n'est plus très bien mis à jour.

Utilisé par: OpenBSD

SVN (Subversion), centralisé

Probablement l'outil le plus utilisé à l'heure actuelle. Il est assez simple d'utilisation, bien qu'il nécessite comme tous les outils du même type un certain temps d'adaptation. Il a l'avantage d'être bien intégré à Windows avec le programme Tortoise SVN, là où beaucoup d'autres logiciels s'utilisent surtout en ligne de commande dans la console. Bien qu'étant le plus connu et le plus utilisé à l'heure actuelle, de nombreux projets commencent à passer à des outils plus récents.

Utilisé par: Apache, Redmine, Struts.

Mercurial, décentralisé

Plus récent, il est complet et puissant. Il est apparu quelques jours après le début du développement de Git et est d'ailleurs comparable à ce dernier sur bien des aspects.

Utilisé par: Mozilla (Firefox...), Python, OpenOffice.

Bazaar, décentralisé

Un autre outil, complet et récent, comme Mercurial. Il est sponsorisé par Canonical, l'entreprise qui édite Ubuntu.

Utilisé par: Ubuntu, MySQL, Inkscape.

Git, décentralisé

Très puissant et récent, il a été créé par Linus Torvalds, qui est entre autres l'homme à l'origine de Linux. Il se distingue par sa rapidité et sa gestion des branches qui permettent de développer en parallèle de nouvelles fonctionnalités.

Utilisé par: VLC, Kernel de Linux, Debian, Android, Gnome, Qt…

Il existe aussi des logiciels propriétaires : Perforce, BitKeeper, Visual SourceSafe de Microsoft.

Exemple de GIT

Particularités de Git

Mercurial, Bazaar et Git se valent globalement, ils sont récents et puissants, chacun a des avantages et des défauts.

Concernant les avantages de Git sur les autres, on retiendra surtout que Git :

- est très rapide ;
- sait travailler par branches de façon très flexible ;
- est assez complexe, il faut un certain temps d'adaptation pour bien le comprendre et le manipuler, mais c'est également valable pour les autres outils ;
- est à l'origine prévu pour Linux. Il existe des versions pour Windows ou Mac OS X avec des interfaces graphiques simplifiées que depuis peu.

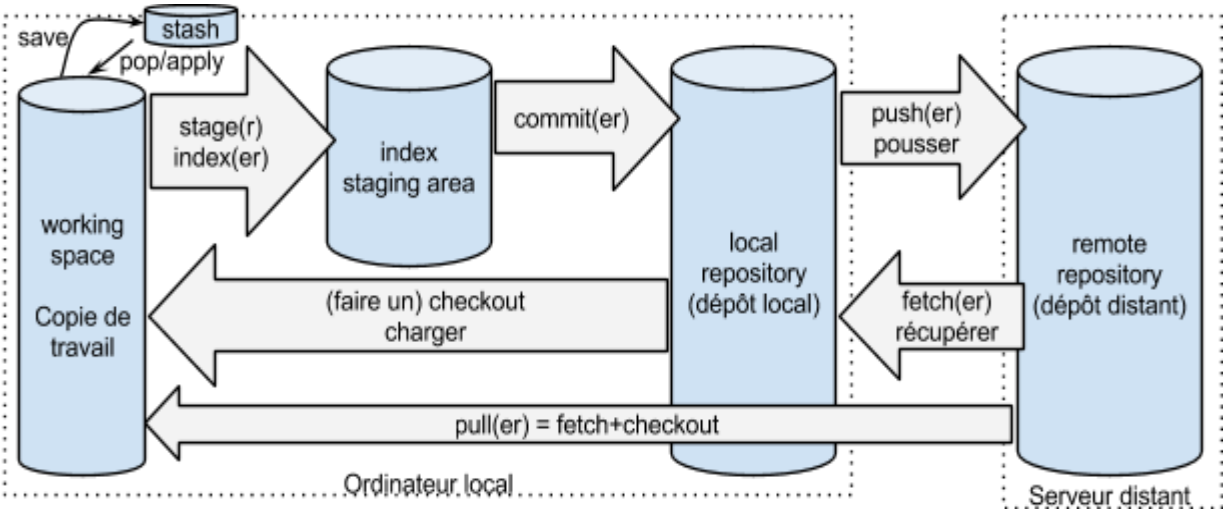
Une des particularités de Git est l'existence de sites web collaboratifs comme GitHub et Gitorious. GitHub, par exemple, est très connu et utilisé par de nombreux projets : jQuery, Symfony, Ruby on Rails…

Ces sites ressemblent à une sorte de réseau social pour développeurs : n'importe qui peut regarder tous les projets évoluer et décider de participer à l'un d'entre eux. Il est également

possible d'y créer son propre projet ouvert à tous et gratuit pour les projets open source publics (payante pour ceux qui l'utilisent pour des projets propriétaires).

GitHub fournit le dépôt où les développeurs qui utilisent Git se rencontrent. C'est un excellent moyen de participer à des projets open source et de publier son projet sur un serveur géré par une entité extérieure.

Vocabulaire spécifique à GIT



En imaginant qu'un dépôt local (sur l'ordinateur local de travail) et un dépôt distant (sur un serveur distant) existent déjà et soient bien configurés, cette figure retrace l'ordre des actions généralement constaté. Un ensemble de modifications dans la copie de travail (working space) est d'abord mis dans l'index (staging area) avec un "stage(r)" ou "index(er)". On peut choisir à cette étape d'indexer une partie seulement des modifications. Les modifications sont ensuite commitées dans le dépôt local (on parle d'un commit). On peut commiter plusieurs fois dans le dépôt local avant de pousser tous les commits d'un coup dans le dépôt distant. Par la suite on peut récupérer les modifications du dépôt distant vers le dépôt local puis charger (faire un checkout) les modifications du dépôt local vers la copie de travail; ou tout simplement puller les modifications, c'est à dire puller=récupérer+charger(faire un checkout) d'un coup. Il existe aussi une action appelée "sync" qui fait un push suivit d'un pull, c'est à dire une synchronisation du dépôt local avec le dépôt distant.

On remarque une petite zone "stash" comparable à un répertoire temporaire sur lequel on peut faire "save" ou "pop/apply" depuis la copie de travail. Elle est utile par exemple si la personne est dans cette situation: elle a fait des modifications dans la copie de travail mais ne veut pas encore les commiter (les bonnes pratiques explicitées à la fin l'en empêche) et elle veut faire un checkout (ou un pull) qui efface les changements non commités.

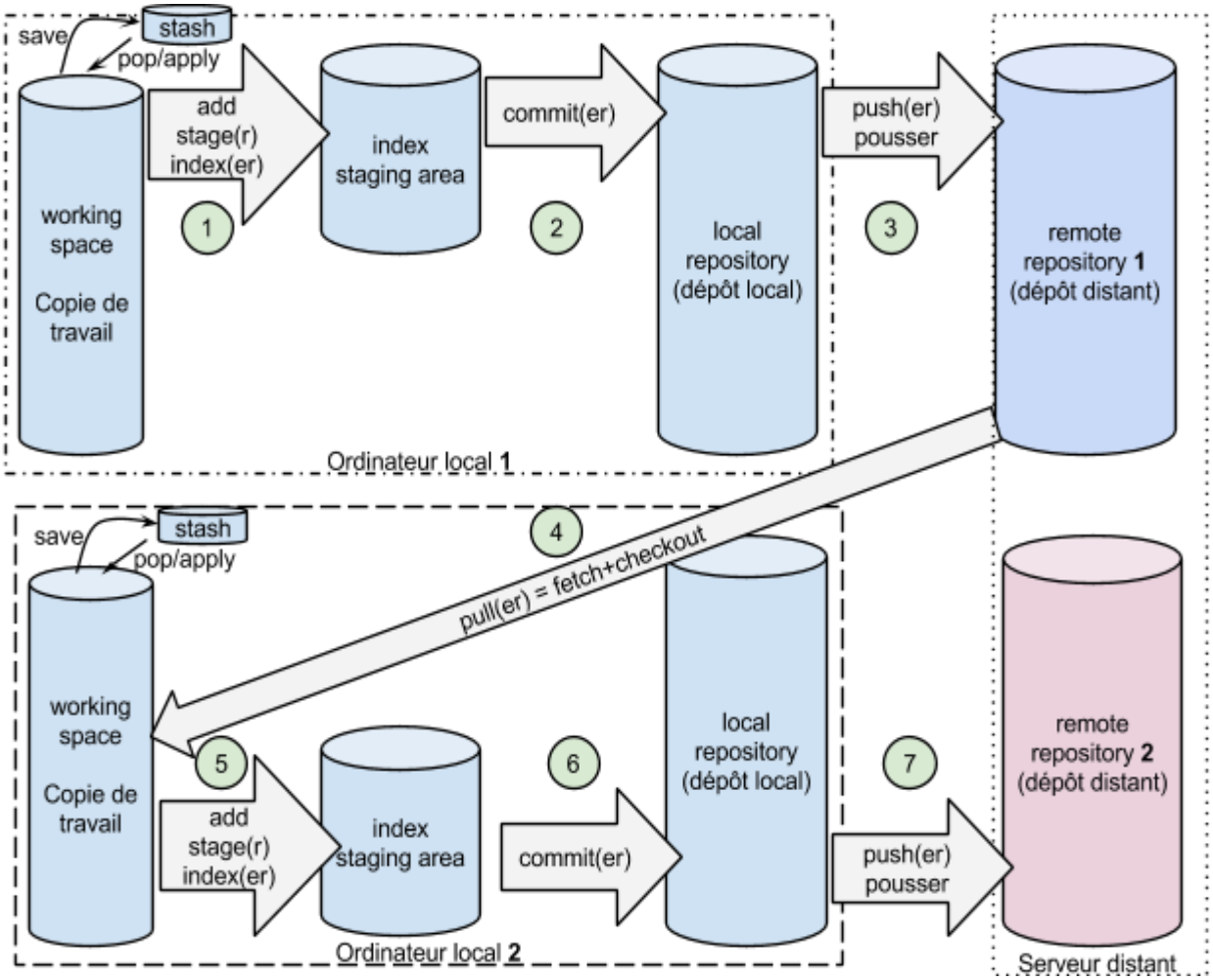
Ordre de création des dépôts, clonage de dépôt et fork de dépôt.

Dans le cas de l'utilisation uniquement en local (sur une seule machine, pour garder l'historique des version en local) on ne fera que créer un dépôt local (faire un "init") et on n'utilisera aucun dépôt distant. On ne fera donc que des commit et checkout et pas de push ni de pull.

Dans le cas de l'utilisation avec des dépôts distants, **il est préférable de ne pas créer de dépôt local** mais de cloner le dépôt distant dans un dépôt local qui sera créé automatiquement (faire un "clone"). Le clonage va rapatrier toutes les versions et branches du projet. Sauf si le projet est vraiment gigantesque il vaut mieux cloner l'intégralité du dépôt distant. On clone donc un dépôt distant vers un dépôt local.

Un dernier terme est celui de "fork", beaucoup moins utilisé: il s'agit d'un clonage particulier entre deux dépôts distants (et non entre un dépôt distant et un dépôt local).

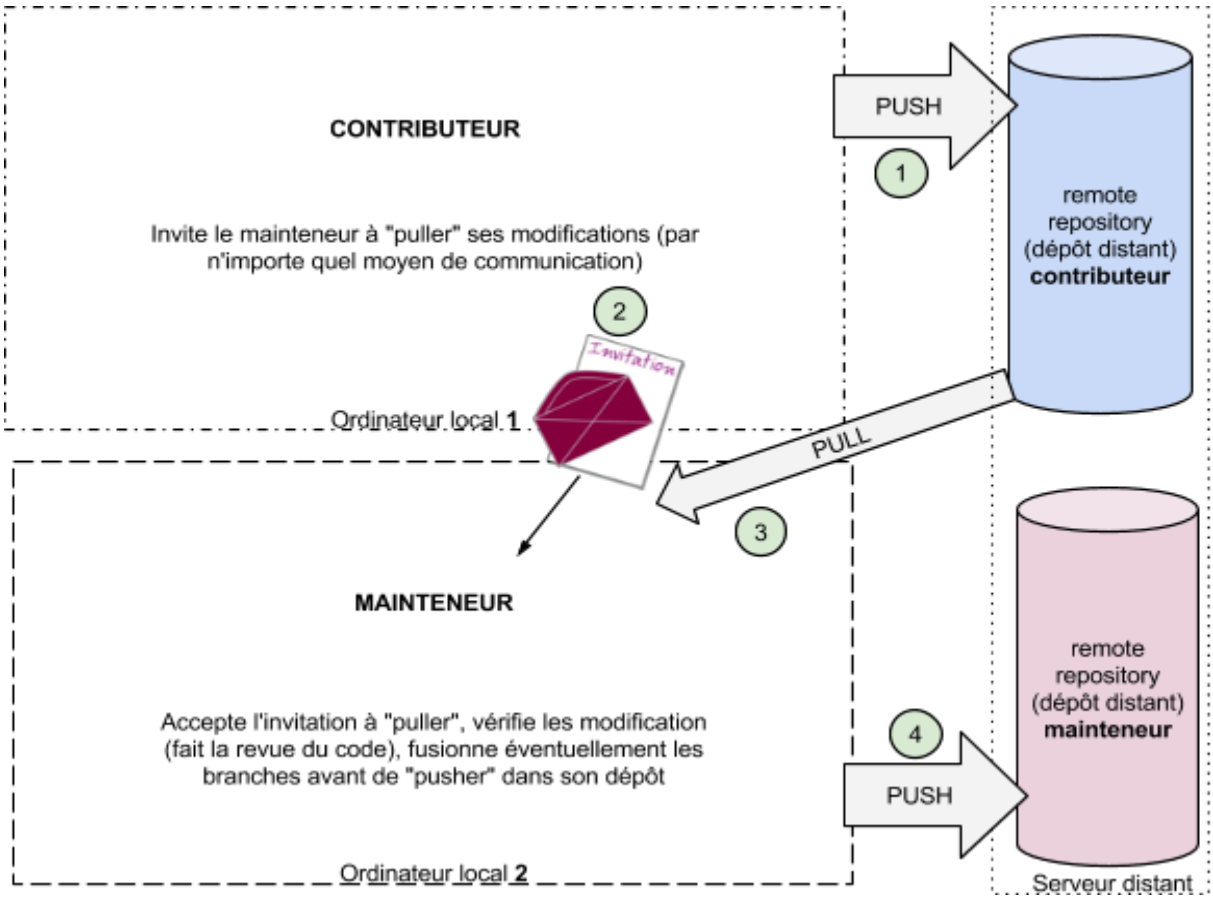
Collaboration de deux personnes avec GIT



Dans le cas de collaboration entre deux personnes avec deux ordinateurs et deux dépôts distants (ici il s'agit de remote repository 1 et remote repository 2 mais sur le même serveur): si

la première personne veut partager des modifications, elle indexe, commite et pousse sur le dépôt distant 1 (actions 1, 2 et 3 sur la figure) puis invite la deuxième personne à puller les modifications depuis le dépôt distant 1 vers la copie locale de la deuxième personne (action 4 sur la figure). Cette invitation à puller est également appelée “pull request”, elle peut se faire par courrier électronique ou avec certains réseaux sociaux de développeurs qui gèrent les “pull request” comme github. Cette deuxième personne peut alors décider (ou non) d’indexer, commiter et pousser ces changements dans son dépôt distant 2 (actions 5, 6 et 7 sur la figure). Ce schéma est simplificateur car en général les deux personnes vont travailler sur des branches différentes et la deuxième personne devra fusionner la branche de la première personne avec sa propre branche.

Voici un schéma montrant les rôle du même scénario:



On voit apparaître deux rôles: le contributeur fait des modifications au projet qu’il “push” dans son dépôt distant (action 1), il envoie une invitation au mainteneur pour lui demander de puller ses modifications (action 2). Le mainteneur accepte ou non de puller les modifications (action 3), fait la révision de code (fusion éventuelle de branche, résolution de conflit) avant de les pusher dans son dépôt local (action 4). En réalité il s’agit d’un type particulier de workfow GIT (flot de travail) qui s’appelle forking workflow (utilisant les contributeurs et un mainteneur). Il existe d’autres workflow GIT: centralized workflow, feature branch workflow, gitflow workflow...

Le forking workflow

Etape 0: Le mainteneur crée le dépôt distant “officiel”

Etape 1: Les contributeurs “fork” le dépôt officiel dans des dépôts distants.

Etape 2: Tout le monde clone les dépôts distants dans des dépôts locaux.

Etape 3: Les contributeurs travaillent et modifient leur dépôt local.

Etape 4: Les contributeurs poussent les modifications dans leurs dépôts distants respectifs.

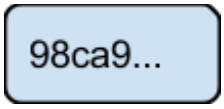
Etape 5: Enfin, les contributeurs invitent le mainteneur à puller leurs modifications et le mainteneur révise le code et pousse les modifications dans le dépôt initial.

Etape 6: Les contributeurs pull le dépôt initial dans leur dépôt local.

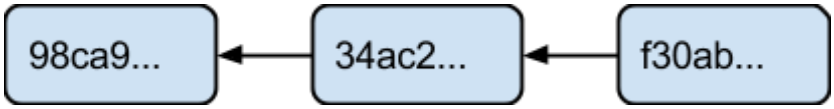
Etape 7 : Retour à l’Etape 3.

GIT et les branches: commit, checkout, HEAD, merge, rebase, reset.

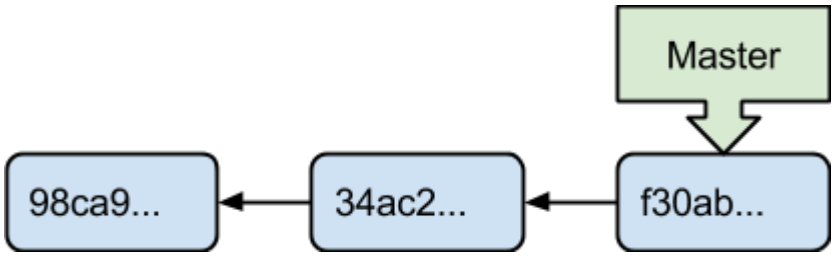
Dans GIT, les commits sont identifiés par un identifiant unique (une valeur hexadécimale sur 160 bits appelée “SHA”), on peut le représenter par exemple comme ceci:



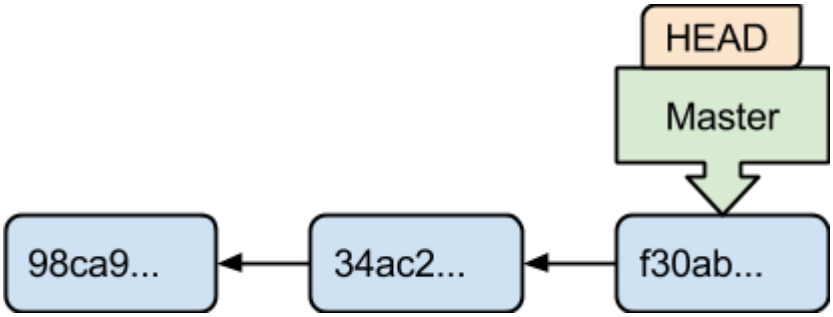
Un commit pointe vers son ou ses commits parents. Un dépôt GIT est finalement un graphe de commits:



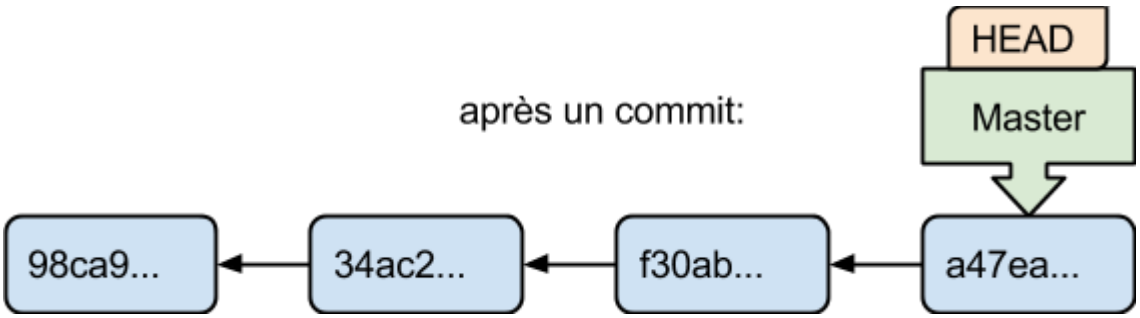
Une branche pointe vers un commit, par exemple la branche master:



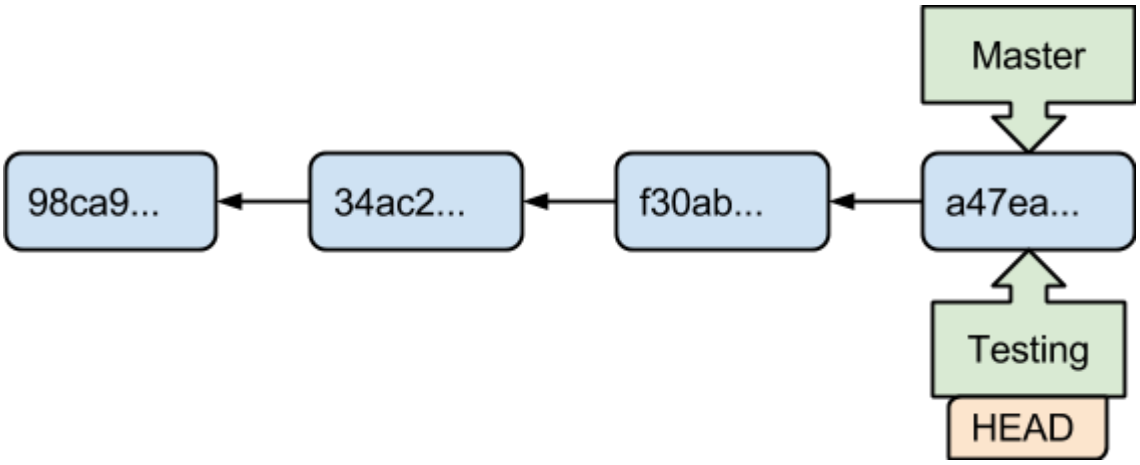
Un pointeur spécial nommé HEAD pointe vers la branche courante de travail:



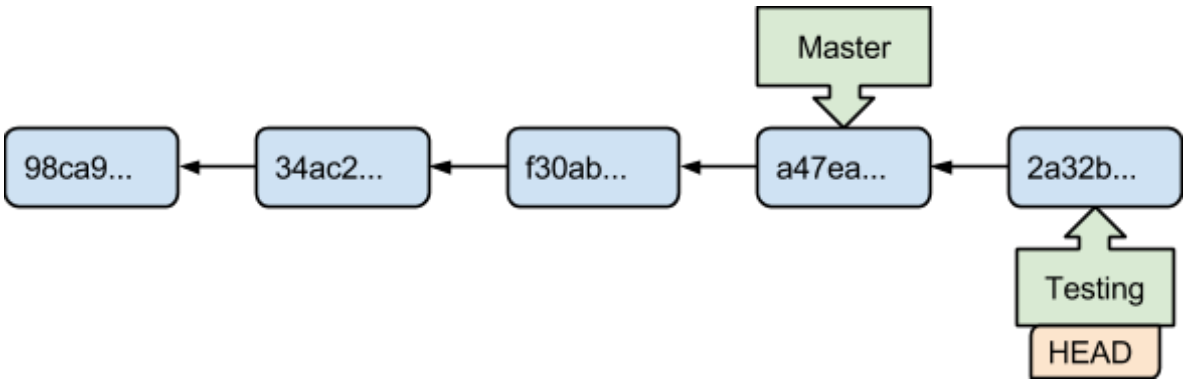
Le pointeur avance quand on fait un commit:



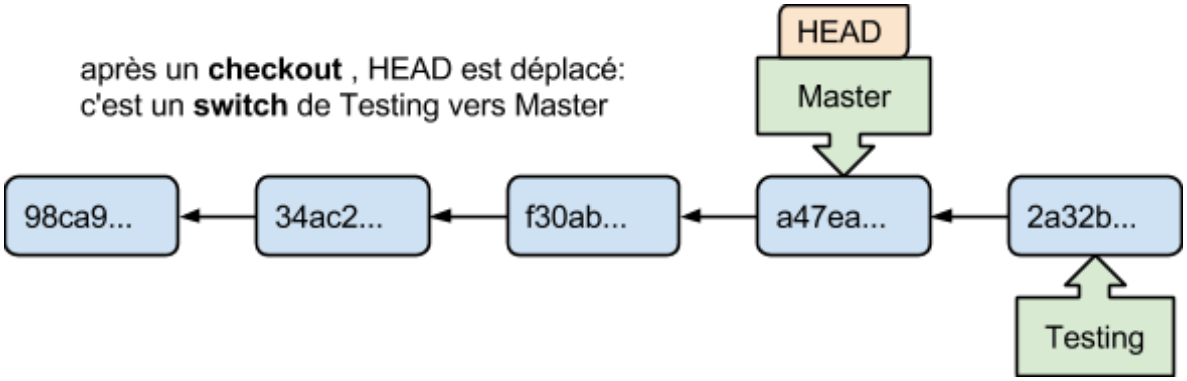
On peut créer des nouvelles branches à partir de n'importe quel commit, créons une branche "Testing" à partir du commit pointé par HEAD (add branch to current HEAD commit) - en général le HEAD se déplace sur la branche nouvellement créée:



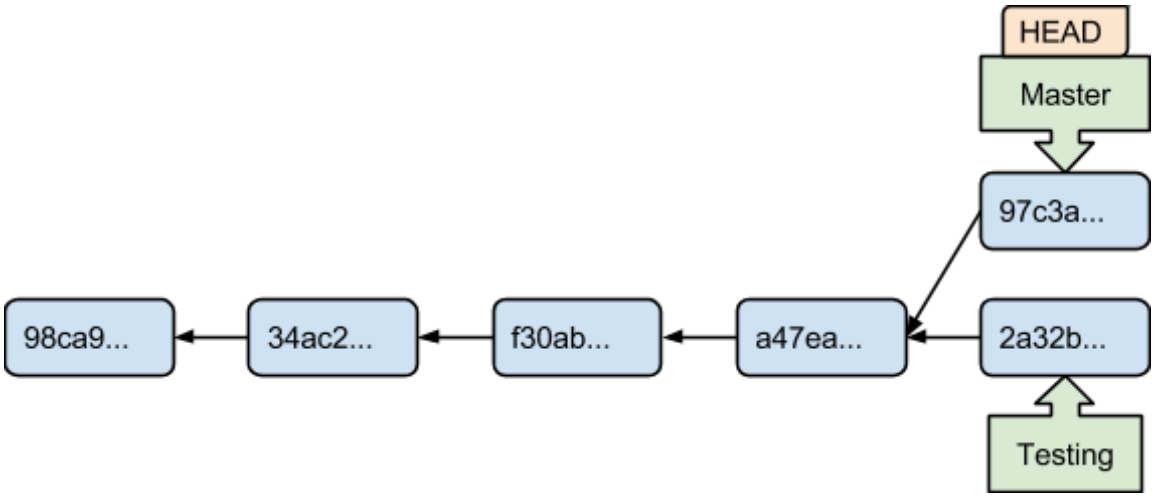
Les commits ne font déplacer que le pointeur de la branche courante (pointée par HEAD). Nous travaillons actuellement sur Testing, faisons un commit:



Un checkout change la position de HEAD, on peut faire un checkout d'une branche ou d'un commit en particulier. Si on fait un checkout d'un commit sans branche (par exemple vers f30ab... sur l'exemple), il vaut mieux créer immédiatement une nouvelle branche sur ce commit. Si on fait un checkout de branche, on dit qu'on "switche" de branche. **Attention: cela replace tous les fichiers/dossiers dans le workspace (espace de travail) par ceux de la nouvelle branche.**
Par exemple "switchons" de Testing vers Master (faisons un checkout de Master):

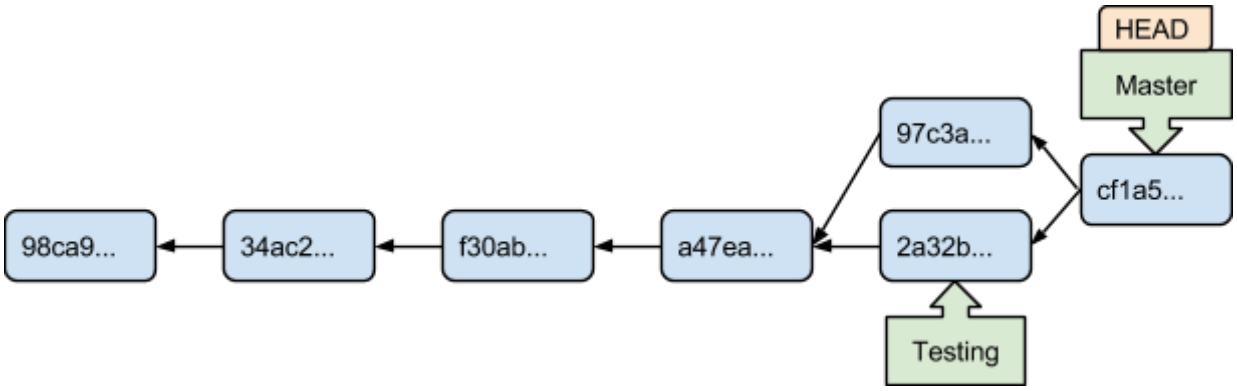


Maintenant que le HEAD est Master, faisons un commit, il va se créer une divergence et on distingue bien les deux "branches" de l'arbre:

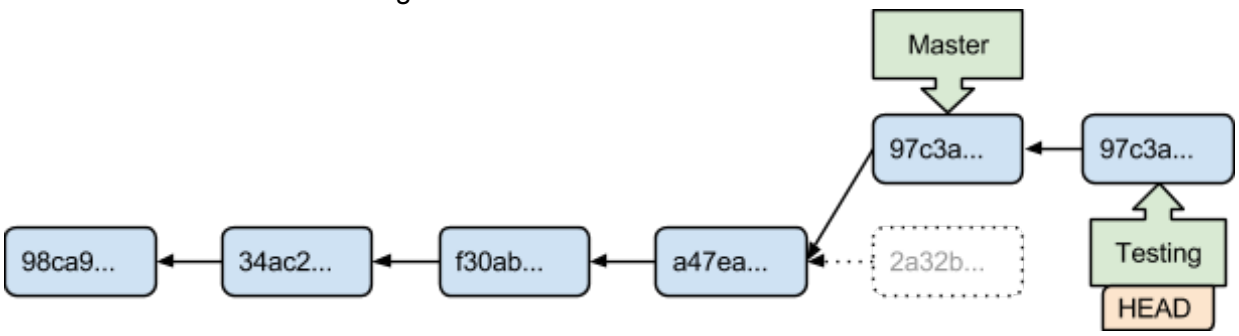


On peut fusionner (merge) ou rebaser (rebase) des branches.

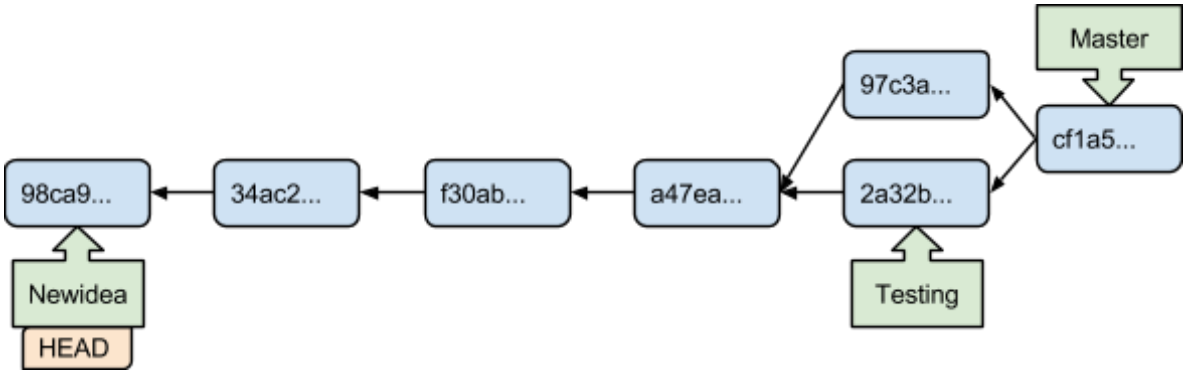
Par exemple si on fusionne Testing dans Master on obtient:



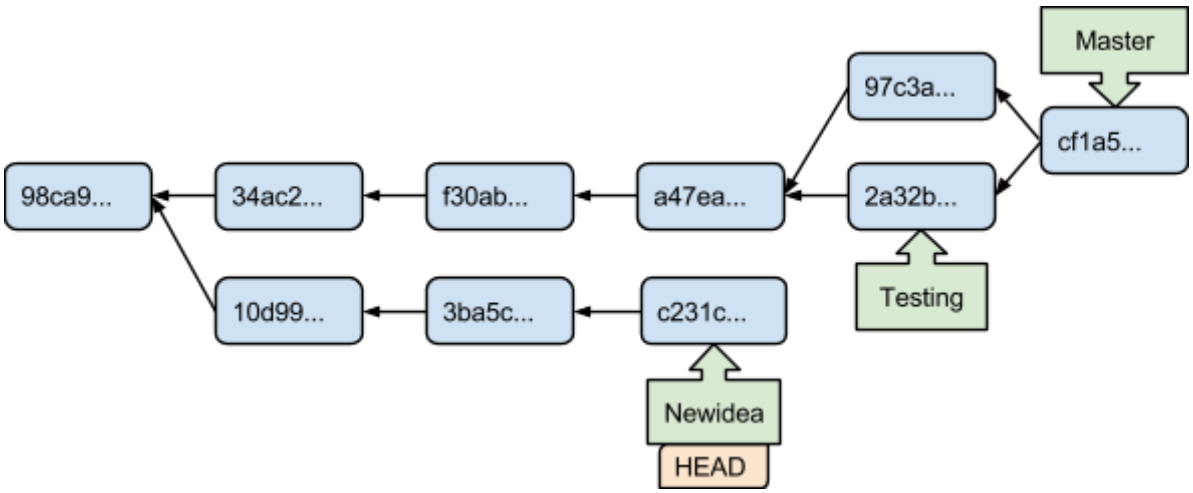
Par contre si on rebase Testing dans Master cela donne:



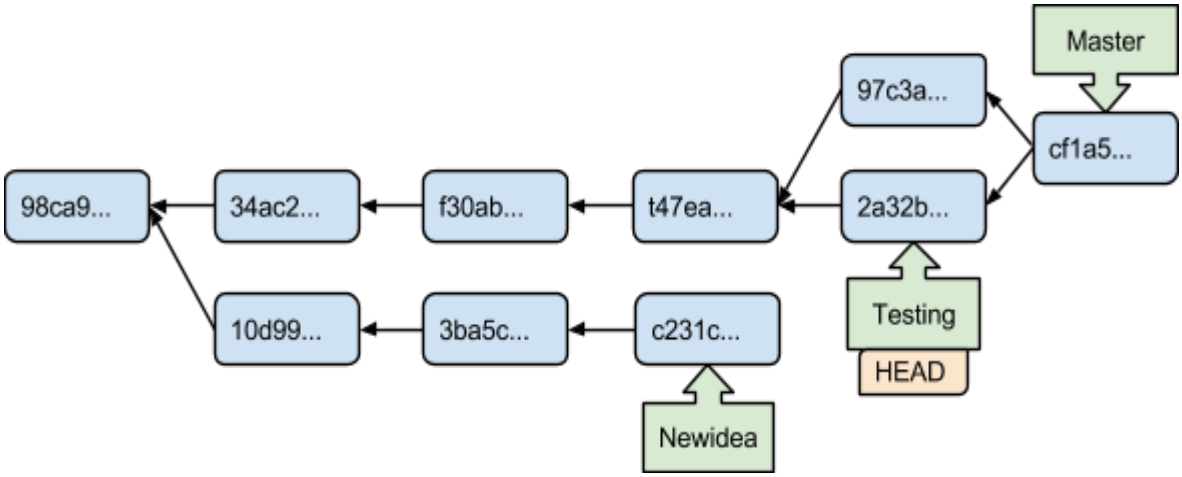
Important: en général si on veut repartir d'une ancienne version il est préférable de créer une nouvelle branche. Par exemple repartons de la version originale car nous avons une nouvelle idée et créons une branche Newidea:



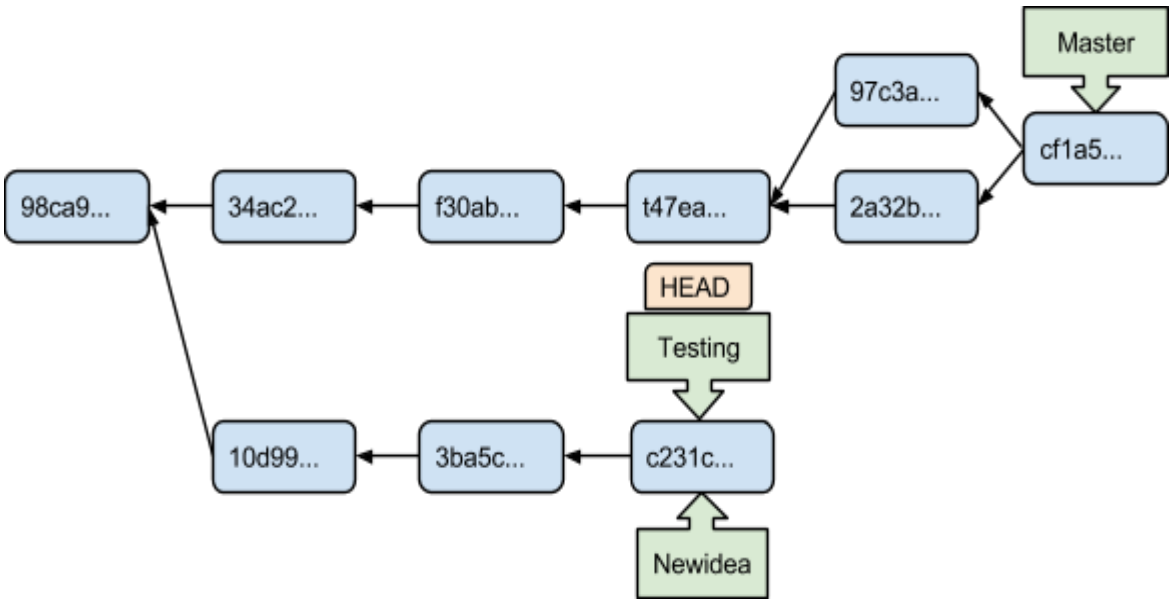
Faisons quelques commits pour tester notre nouvelle idée:



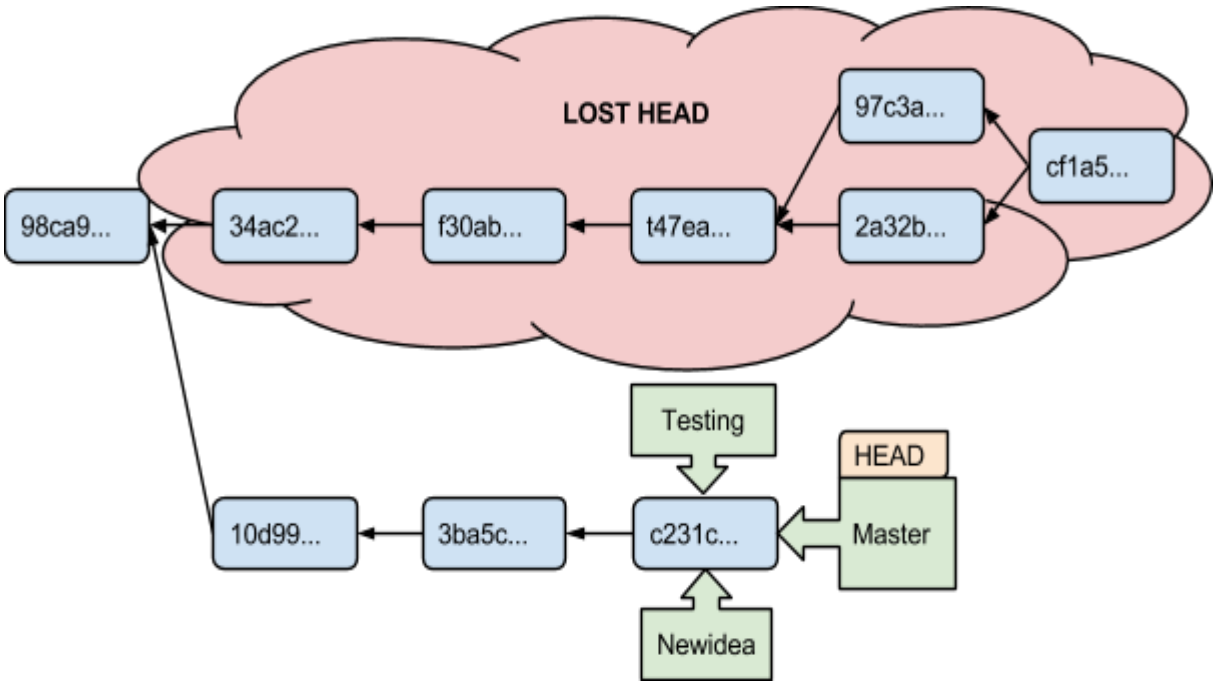
Il existe un mécanisme de “reset” pour déplacer le pointeur de branche mais ATTENTION il vaut mieux ne pas l’utiliser car on peut vite faire n’importe quoi. Par exemple si on veut déplacer Testing pour qu’il pointe sur la même chose que Newidea, on switche d’abord vers la branche Testing avec un checkout (déplace le HEAD):



Puis on fait un “reset” de Testing vers Newidea (ATTENTION ceci est à titre d’exemple mais évitez les “reset” !):



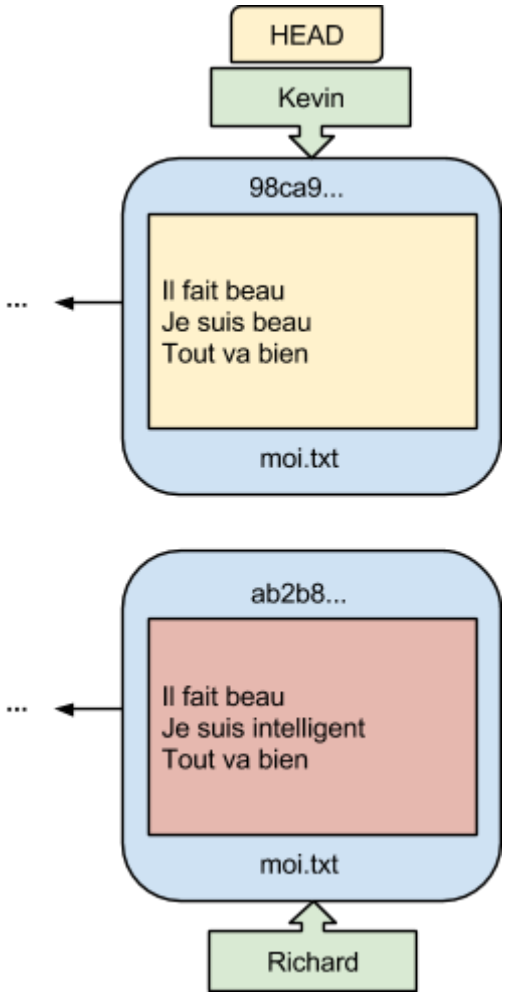
Retenez donc qu'un checkout déplace HEAD, et qu'il vaut mieux créer une nouvelle branche quand on veut repartir d'une ancienne version plutôt que d'utiliser "reset". Retenez plus simplement qu'il ne faut pas utiliser "reset".
Un exemple de mauvaise utilisation de "reset" peut mener à la perte du "bout de la branche", c'est à dire une branche dont le bout n'est plus pointé par aucun pointeur de branche. Par exemple nous faisons un reset de Master vers Newidea, nous obtenons une "Lost Head":



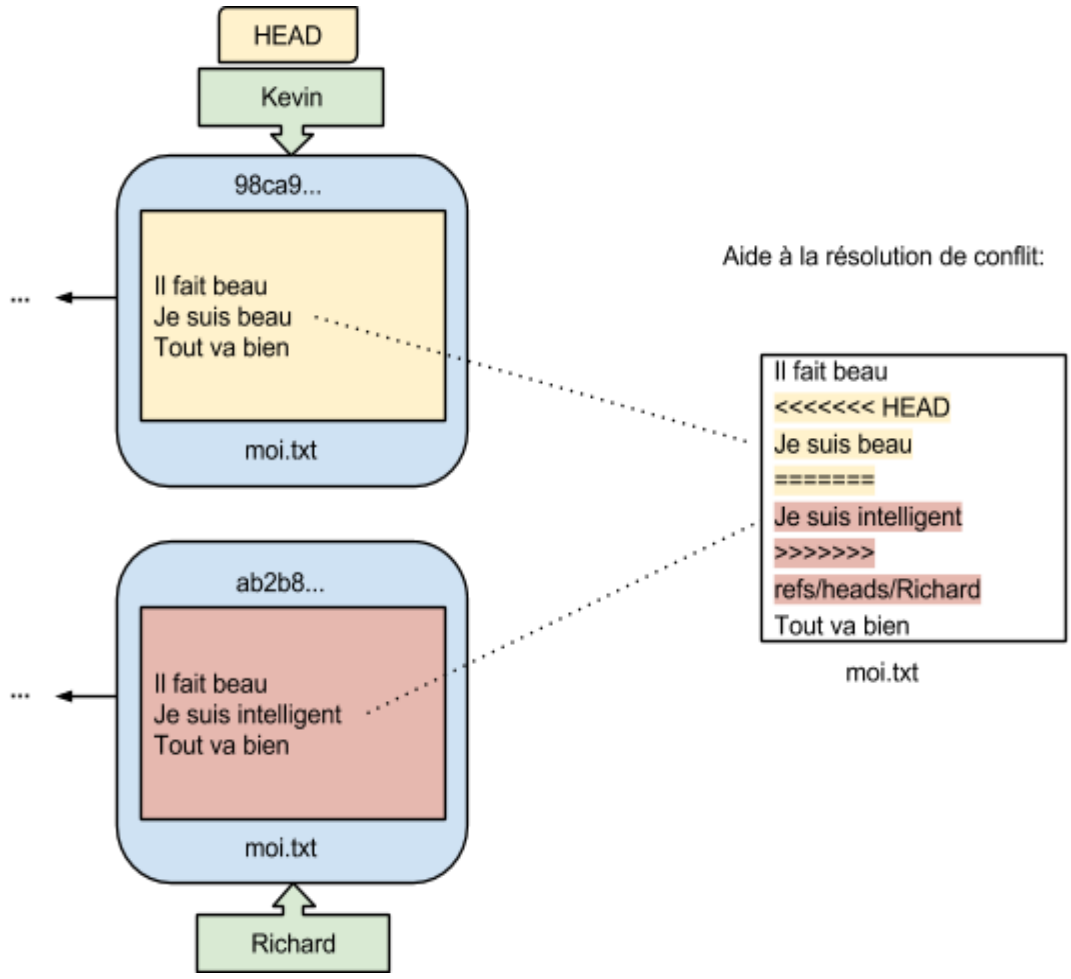
Ne vous inquiétez pas, les LOST HEADs ne sont pas supprimés et sont toujours disponibles sur le dépôt, néanmoins puisqu'aucun pointeur de branche n'est disponible il faudra d'abord en créer un pour accéder aux commits de ce LOST HEAD.

GIT et les conflits

Imaginons deux branches Kevin et Richard, le fichier moi.txt diffère sur la deuxième ligne pour les deux branches. Branche Kevin “Je suis beau”, branche Richard “Je suis intelligent”:



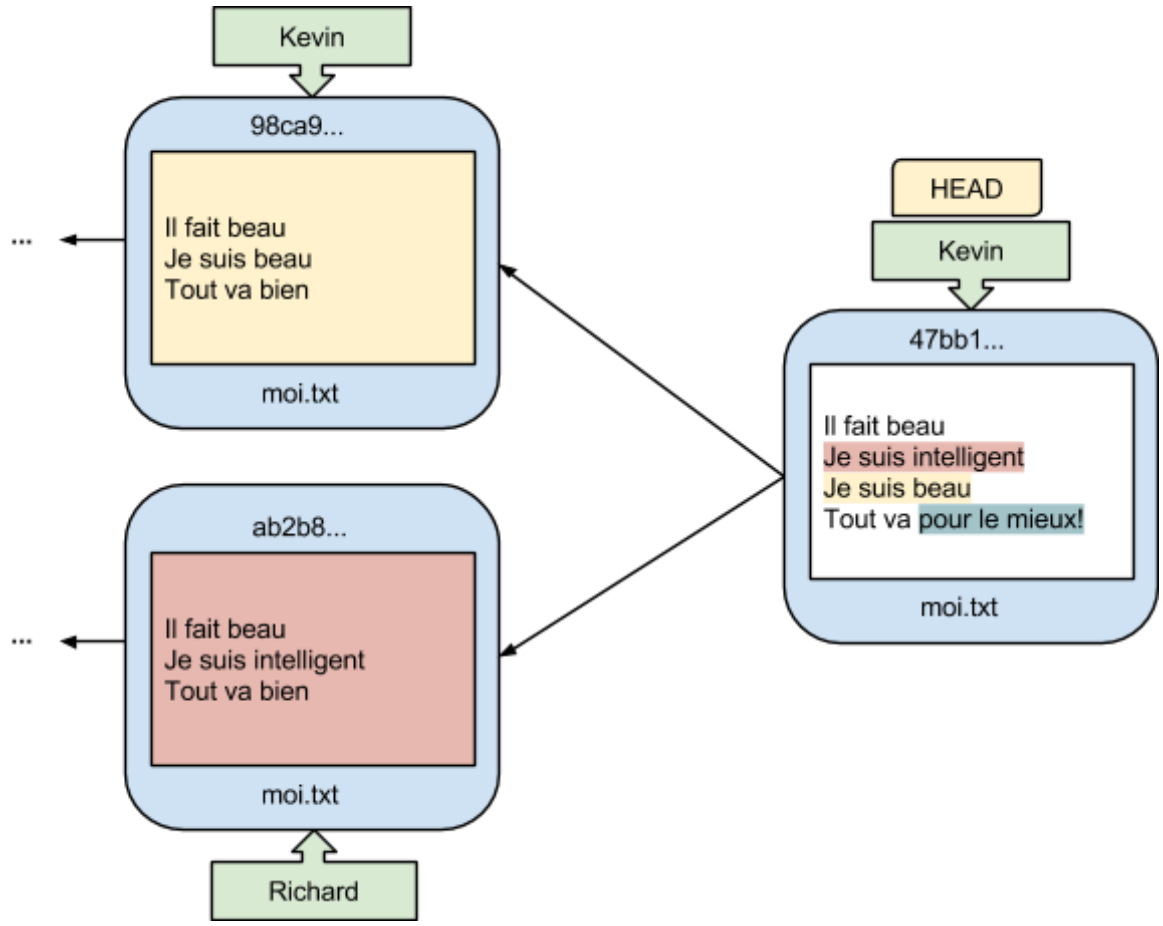
Si nous voulons fusionner (merge) la branche Richard dans la branche Kevin, nous aurons un conflit. Git nous aide à résoudre le conflit avec des annotations:



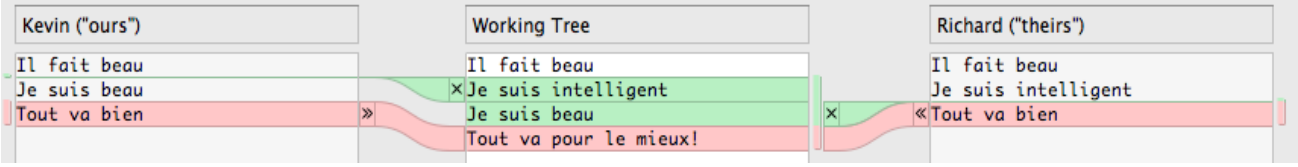
Dans cette annotation, on retrouve:

- entre <<<<<<< HEAD et =======
  la ou les lignes du HEAD (Kevin) qui sont en conflit avec la branche Richard.
- entre ======= et >>>>>>> refs/heads/Richard
  la ou les lignes de la branche Richard (ici refs/heads/Richard est le nom complet de la branche) qui sont en conflit avec la banche HEAD (Kevin).

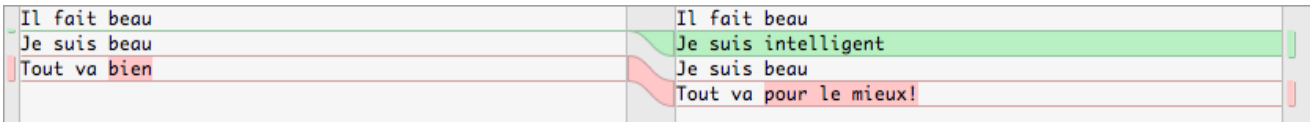
Il faut donc faire un choix: garder l'une ou l'autre ligne ou un autre choix ?
Tant que le choix n'est pas effectué, nous restons dans l'état "en train de fusionner" ou "merging" et la fusion n'est pas faite et ne peut pas être commitée.
Après mûre réflexion, faisons le choix de garder les deux phrases en plaçant "je suis intelligent" en premier et en remplaçant "Tout va bien" par "Tout va pour le mieux!" . Effaçons ensuite les annotations automatiques d'aide à la résolution de conflit et faisons un commit:



Voici un exemple d'un outil intégré à SmartGIT (interface graphique pour GIT) pour l'aide à la résolution de conflit, il montre les différences lors des choix (ajouts/suppressions/modifications):



Enfin, cet outil montre plus précisément (mot à mot) les différences lors du commit:



GIT et les bonnes pratiques

Commiter des modifications qui sont en relation les unes des autres

Un commit doit contenir un ensemble de modifications qui sont en relation les unes des autres. Par exemple, le fait de réparer deux problèmes (deux bugs) doit produire deux commits. Des

petits commits facilitent la compréhension de toutes les personnes pour les modifications et facilitent également le processus d'annulation (roll-back) si quelque-chose se passe mal. Pour faire plusieurs commits en regroupant des modifications, il est possible d'utiliser l'index ou staging area: on ajoute les modifications du premier commit (premier bug à résoudre) dans le staging area puis on commite et dans un deuxième temps on ajoute les modifications du deuxième commit (deuxième bug à résoudre) et on commite.

Tester le code avant de commiter

Résistez à la tentation de commiter quelque-chose que vous "pensez" terminé. Il faut d'abord le tester en profondeur pour être sûr que c'est terminé et qu'il n'y a pas d'effets de bord (interférences) du mieux possible. C'est encore plus vrai quand vous pushez du code sur des dépôts extérieurs car cela veut dire que vous êtes prêts à envoyer une invitation pour puller vos modifications (dans votre invitation vous expliquez que vous avez bien testé votre code et mettez le mainteneur à l'aise afin qu'il puisse puller votre code).

Utiliser les branches

Si les branches sont la fonctionnalité la plus puissante de GIT ce n'est pas par hasard: les branches sont parfaites pour vous empêcher de mixer différentes lignes de développement de votre projet. Vous devez utiliser les branches de manière intensive: pour les nouvelles fonctionnalités ajoutées à votre projet, pour la correction des erreurs (faites une branche à chaque fois que vous tentez de réparer une erreur), pour les nouvelles idées...

Commiter souvent

Si vous commitez souvent, les commits seront plus petits et aideront non seulement les autres personnes à mieux comprendre mais en plus à avoir des modifications qui sont en relation les unes des autres dans un commit. Cela permet de partager votre code de manière plus fréquente avec les autres, il est donc plus facile pour tout le monde d'intégrer vos changements et évite les situations de résolution de conflit. Au contraire si vous faites des gros commits non réguliers il sera plus difficile de résoudre les conflits.

Ecrire de bonnes descriptions dans les commits

Commencez votre message de commit avec un petit sommaire de vos modifications (50 caractères par exemple). Séparez le d'un blanc puis de réponses détaillées sur
- quelle est la motivation de cette modification ?
- quelle est la différence par rapport à l'original ?

Forcez vous à utiliser le présent ou l'impératif et n'utilisez pas le passé ou le futur.

Ne pas commiter un travail à moitié fait.

Vous devez commiter des modifications quand le travail est finalisé et uniquement dans ce cas. Cela ne veut pas dire que vous devez terminer une fonctionnalité ou une grande idée d'un coup, cest plutôt le contraire: divisez la fonctionnalité en petits morceaux logiques et pensez à commiter souvent et assez tôt quand ces morceaux sont terminés. Ne commitez pas avant d'aller prendre votre café ou si quelqu'un vous interrompt dans votre travail ou à la fin d'une journée de travail (enregistrez le fichier normalement dans ce cas). Si vous êtes tentés de commiter dans ces cas là, vous pouvez utiliser les "stashs" de GIT qui ressemblent à des répertoires temporaires avant de commiter.

GIT n'est pas un système de sauvegarde/backup.

Avoir ses fichiers dans un dépôt distant est plaisant mais trompeur! Un gestionnaire de versions est là pour gérer les versions, par pour stocker des sauvegardes régulières de votre travail! Ne faites pas la confusion et n'assimilez pas GIT à un logiciel de sauvegarde/backup ce n'est qu'une toute petite partie de la puissance de GIT. En faisant de la gestion de versions vous devez prêter une attention particulière à la sémantique des versions (signification des versions, description précise des commits) et vous ne devez pas envoyer des fichiers dans le dépôt distant de manière aléatoire pour les "sauvegarder", c'est un mauvais usage, autant utiliser Dropbox.