TD 5: Développement collaboratif décentralisé

ED6 — Licence 3 — Université Paris Diderot

Séances du 7 et 28 avril 2014

Git est un logiciel de contrôle de versions décentralisé. Contrairement à d'autres outils comme Subversion qui se basent sur un unique dépôt avec lequel se synchronisent une ou plusieurs copies de travail (les copies de travail ne peuvent se synchroniser entre elles : elles doivent passer par le dépôt), chaque copie de travail d'un projet versionné avec Git joue aussi le rôle de dépôt, et il est possible de synchroniser entre elles n'importe quelles copies de travail. De plus, Git permet d'utiliser une ou plusieurs *branche de développement* et de fusionner entre elles ces branches de développement.

1 Nouveau repositoire

Nous allons tout d'abord nous intéresser à l'aspect gestionnaire de versions de Git : comment enregistrer l'historique des modifications apportées à un projet. Pour obtenir un repositoire Git sur lequel travailler nous avons deux options :

1. création d'un repositoire vide (typiquement utilisé pour commencer un nouveau projet de développement) ;
2. copie (*clone* dans le langage de Git) d'un repositoire existant pour y travailler dessus (typiquement utilisé pour collaborer avec les développeurs du projet en question).

Nous allons d'abord expérimenter le premier cas.

Pour initialiser un repositoire, il convient d'invoquer la commande `git init monrepo`. Cette commande initialise un dépôt Git dans le répertoire `monrepo` (qui est créé si celui-ci n'existe pas). Ce repertoire contient alors à la fois une version de travail (dans `monrepo`) et un repositoire (ou *dépot*) Git (dans `monrepo/.git`).

Question 1.1. *Initialiser un nouveau dépôt Git dans un répertoire* `sandwich`*, et créez le fichier* `burger.txt` *qui contient la liste des ingrédients d'un burger, un ingrédient par ligne.*

```
steak
salade
tomate
cornichon
fromage
```

Git a plusieurs interfaces utilisateur. La plus complète étant l'interface en ligne de commande (CLI), nous allons d'abord expérimenter avec celle-ci.

2 Commit

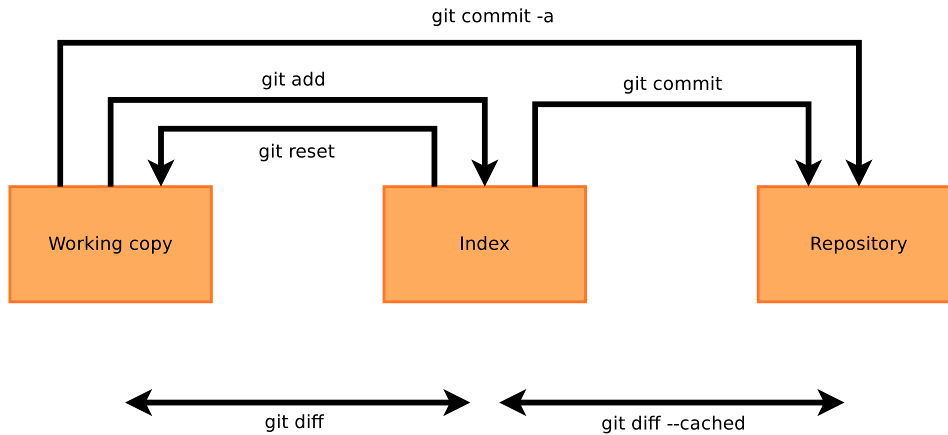
Pour être intégrée dans l'historique des révisions, chaque modification doit suivre le *workflow* montré en Figure 1 :

1. une modification est d'abord effectuée dans la copie de travail ;
2. après, elle doit être préparée pour le commit et placée dans une aire temporaire nommée *index* avec la commande `git add` ;
3. enfin, ce qui a été placé dans l'index peut être commité avec la commande `git commit`.

`git diff`, selon les paramètres d'appel peut être utilisé pour observer les différences entre les états en Figure 1 ; le format d'affichage est le même de `diff -u` que vous connaissez déjà.

Question 2.1. *Vérifiez avec* `git status` *l'état dans les quel le repositoire se trouve. Vos modifications (l'ajout du fichier* `burger.txt`*) doivent être présentes seulement dans la copie de travail, est-ce le cas ?*

Question 2.2. *Préparez* `burger.txt` *pour le commit avec* `git add burger.txt`*. Utilisez* `git status` *à nouveau pour vérifier que les modifications ont bien était placés dans l'index. Maintenant, utilisez* `git diff --cached` *pour observer les différences entre l'index est la dernière version présent dans l'historique de révision (qui est vide).*



Source : http://thomas.enix.org/pub/conf/git2011/presentation.pdf

FIGURE 1 – `git add` *workflow*

Question 2.3. *Commitez votre changement avec `git commit`. Git va lancer pour vous un éditeur de texte pour entrer un message de commit appropriez. Le format doit être celui que nous avons vu en cours, i.e. une ligne courte (≤ 65 caractères) de description générale puis, si nécessaire, une ligne vide et plusieurs lignes des description plus détaillé du changement.*

Question 2.4. *Exécutez à nouveau `git status`, pour vérifier que tous changements ont été commités.*

Question 2.5. *Essayez maintenant la commande `git log` pour afficher la liste des changements effectués dans ce repositoire ; combien y en a ? Quel est le numéro (un hash cryptographique en format SHA1) du dernier commit effectué ?*

Question 2.6. *Rajoutez un ingrédient dans* `burger.txt`*, puis créez quelques autres sandwich.txt et commitez toutes ces modifications* séparément*. Chaque commit doit contenir un et un seul changement à un sandwich ou l'autre. Effectuez au moins 5 modifications différentes (et donc 5 commits differents). À chaque étape essayez les commandes suivantes :*
- `git diff` avant `git add` *pour observer ce que vous allez ajouter à l'index ;*
- `git diff --cached` après `git add` *pour observer ce que vous allez committer.*

Question 2.7. *Regardez à nouveau l'historique des modifications avec `git log` et vérifiez avec `git status` que vous avez tout commité. Git offre plusieurs interfaces, graphiques et non, pour afficher l'historique, essayez les commandes suivantes :*
- `git log`
- `gitg` *(onglet "Historique")*
- `git log --graph --pretty=short`
- `gitk --all`

3 Voyage dans le temps

Question 3.1. *Et si vous voulez changer d'avis entre les différents états en Figure 1 ? Faites une modification à un sandwich, ajoutez le avec `git add` et vérifiez avec `git status` qu'il est bien dans l'index, mais ne le commitez pas. Maintenant, exécutez `git reset` sur le fichier (ou les fichiers) que vous avez préparés pour le commit ; vérifiez avec `git status` le résultat.*

Question 3.2. *Votre changement a été repoussé "arrière" de l'index vers la copie de travail. Vous pouvez maintenant le jeter à la poubelle avec la commande `git checkout` sur le nom du fichier, qui récupère la dernière version* committée *d'un fichier de l'historique. Essayez le et vérifiez après avec `git status` qu'il n'y a maintenant plus aucune modification à committer.*

`git checkout` est une commande très puissante. Il vous permet de voyager entre différentes branches (voir plus loin) et aussi de revenir *temporairement* à une version précédente de votre copie de travail.

Question 3.3. *Regardez l'historique de votre repositoire avec `git log` ou avec `gitk --all` ; choisissez dans la liste un commit (autre que le dernier). Exécutez `git checkout COMMITID` ou "COMMITID" est le numéro de commit qui vous avez choisi. Vérifiez que l'état de vos burgers est maintenant revenue à l'arrière dans le temps, au moment du commit que vous avez choisi. Que dit maintenant `git status` ?*

Attention, avec `git checkout` les fichiers de votre copie de travail sont modifiés directement par Git pour le remettre à l'état que vous avez demandé. Si les fichiers modifiés sont ouverts par d'autres programmes (e.g. un éditeur de texte comme Emacs), il faudra les ré-ouvrir pour observer les modifications.

Question 3.4. *Vous pouvez retourner à la version plus récente de votre repositoire avec* `git checkout master`*. Vérifiez que cela est bien le cas. Que dit maintenant* `git status` *?*

4 Clone

Très peu de logiciels sont développés aujourd'hui à parti de zéro. Le cas normal est de plus en plus de se baser sur un logiciel existant, et de chercher à faire remonter aux auteurs du logiciel les changements qui sont importants pour nous. Git rend tout cela très efficace pour les auteurs originels et pour les contributeurs.

Étant donné un repertoire Git publié sur l'Internet (il est possible de le faire sur le Web, via SSH, et aussi à travers beaucoup d'autres protocoles), vous pouvez utiliser la commande `git clone URL` pour créer un nouveau repositoire en locale qui contient toutes les révisions présent sur le repositoire d'origine.

Comme projet d'exemple, nous allons utiliser *purity* un logiciel jouet pour vérifier combien un joueur adhère à un profil (e.g. nerd, hacker, etc).

Question 4.1. *Récupérez le repositoire Git de purity avec la commande*
`git clone http://anonscm.debian.org/git/collab-maint/purity.git`
`git clone` *va créer pour vous un nouveau repositoire* `purity/` *qui contient en plus du code source de* `purity/`*, tout l'historique du repositoire d'origine que vous venez de cloner.*

Question 4.2. *Étudiez l'historique de votre nouveau repositoire à l'aide de* `git log --graph` *ou d'autres outils que nous avons déjà utilisés.*

Question 4.3. *Compilez le projet avec la commande* `make LIBDIR=tests/`*, testez-le avec la commande* `./purity hacker`*, regardez le code source de purity dans le fichier* `pt.c`*.*

5 Branches de développement

Nous allons maintenant nous concentrer sur la notion de *branche*. Lors du développement d'un projet, il peut arriver que l'on veuille introduire une nouvelle fonctionnalité dans le projet, sans "casser" le projet. Nous voudrions donc pouvoir basculer instantanément de la version stable du projet à sa version "en développement". C'est ce que nous permettent de faire les branches.

Plus précisément, vu la facilité avec laquelle Git permet de gérer les branches, en Git on a tendance à utiliser les branches pour chaque développement non trivial, e.g. :

- pour corriger un bug numéroté 1234, on procédera comme suit :
  1. à partir du branche principale de développement (normalement nommé `master`) on créera une nouvelle branche nommée `bug/1234`
  2. on changera la copie de travail pour être sur la branche `bug/1234`
  3. on corrigera le bug, en faisant tous les commits nécessaires
  4. on retournera sur la branche `master`
  5. on fusionnera (*merge*) la branche `bug/1234` avec `master`
- pareil, pour ajouter une nouvelle fonctionnalité nommée *superduper* :
  1. on crée une branche `feature/superduper` à partir de `master`
  2. on passe à la branche `feature/superduper`
  3. on développe *superduper*, on commit (plusieurs fois, si besoin)
  4. on revient sur `master` pour y faire merge de `feature/superduper`

Question 5.1. *Dans le repositoire de purity qui vous aviez cloné, exécutez* `git branch` *et vérifiez que, d'après lui, vous êtes bien sur la branche* `master` *: la branche actuelle est celui avec une étoile.* `"master"` *est le nom de la branch de développement principale, que Git à crée pour vous. Notez aussi que la sortie de* `git status` *contient le nom de la branche actuelle ; essayez !*

On veut maintenant traduire en Français quelques-unes des questions du test pour vrais hacker de purity, et le faire selon le style propre de Git.

Question 5.2. *Créez une nouvelle branche pour le développement à l'aide de `git branch NOM`. L'effet de la commande est de créer une nouvelle branche, nommée NOM, à partir de la branche courante. e.g. vous pouvez essayer avec `git branch fr-translation` pour la nommer "fr-translation". Vérifiez les deux choses suivantes : 1) l'existence de la nouvelle branche (vous pouvez le faire à la fois avec `git branch` et avec `gitk --all`) ; et 2) le fait que vous êtes toujours sur la branche "master" (vous pouvez le faire avec `git status`, `git branch`, et autres outils).*

Vous êtes maintenant prêts à développer vos traductions.

Question 5.3. *Mettez-vous sur la branche de développement. Pour le faire il faut utiliser la commande `git checkout` comme nous l'avons fait dans la première partie du TP pour voyager dans le temps : il suffit de passer comme paramètre le nom du branche à la place de l'identifiant de commit, e.g. `git checkout fr-translation`. Vérifiez après avec `git branch` ou `git status` que vous êtes maintenant sur la branche désirée.*

Question 5.4. *Traduisez en français quelques questions du test de purity pour vrais hackers. Ouvrez le fichier `tests/hacker` avec un éditeur de texte et remplacez le texte en anglais de la première en question avec sa traduction en français ; commitez vos changements. Si vous avez des doutes sur le format du fichier, regardez `tests/format` qui en contient une description. Répétez l'opération pour au moins 2 autres questions.*

Important : faites un commit pour chaque traduction. Pour traduire 4 questions, donc, il faudra faire 4 commits differents, avec les messages de commit appropriés.

Question 5.5. *Exécutez purity sur le fichier que vous venez de traduire, pour vérifier que les questions que vous aviez touchées sont maintenant en Français.*

Question 5.6. *Revenez maintenant sur la branche principale, `master`. Pour le faire exécutez `git checkout master`. Vérifiez que vos changements, maintenant ne sont plus dans le fichier que vous avez touché. Néanmoins, ils n'ont pas été oubliés. Lancez `gitk --all` et vérifiez que les changements existent, même s'il ne sont pas (encore) partie de la branche sur laquelle vous êtes maintenant.*

Question 5.7. *Avant de proceder à l'intégration de vos changements dans la branche `master`, nous allons implémenter une autre feature. Téléchargez le patch : `http://upsilon.cc/zack/teaching/1213/ed6/tp5-adding-some-colors.patch`.*

Vérifiez que vous êtes sur la branche master et appliquez-le à `pt.c`. Commitez les changements induits par ce patch.

Vérifiez avec `gitk --all` ou `git log --all --graph --pretty=oneline` l'état de votre repositoire. À ce point, les 2 branches `master` et `fr-translation` sont parties dans 2 directions différentes : chacune a au moins un commit qui n'existe pas dans l'autre branche.

Vous êtes maintenant prêts à fusionner les changements effectués dans la branche `fr-translation` sur `master`. Notez que l'action de fusionner n'est pas une action symétrique : décider de fusionner le changement de `fr-translation` sur `master` n'est pas la même chose que fusionner les changements de `master` sur `fr-translation`. Dans le premier cas `master` sera changé pour contenir *aussi* les changements de `fr-translation` et celui-ci ne sera pas touché ; vice-versa dans le deuxième cas.

Question 5.8. *Vérifiez que vous êtes toujours sur la branche `master`. Fusionnez-y les changements de la branche `fr-translation` avec la commande `git merge fr-translation`. Git vous demandera un message de commit, comme d'habitude, donnez une description de l'action que vous êtes en train de faire, e.g. "merge French translations of first N questions of the hacker test".*

Question 5.9. *Regardez l'historique maintenant l'historique de votre repositoire avec `gitk --all`. Le "commit" que vous venez d'ajouter est un commit de type "merge" entre deux branches.*

Jusqu'à présent, nous n'avons envisagé que des scénarios dans lequels la fusion des branches est simple, mais il peut y arriver qu'il y ait des conflits, par exemple un même bogue corrigé de manière sensiblement différente dans deux branches différentes.

Question 5.10. *Que se passe-t-il dans ce cas-là ? Essayez d'implémenter ce scénario.*

Par exemple : créez à partir de `master` une branche qui traduit en français les messages d'aide en ligne qui sont affichés lorsque vous appuyez sur 'h' pendant l'exécution. ***Avant*** *d'en faire le merge, créez une autre branche à partir de master et dans cette branche supprimez la description de l'option 'l' de l'aide en ligne. Maintenant retournez sur `master` et cherchez de faire merge des 2 branches.*

Comment Git vous permet-il de résoudre les confits ? Écrase-t-il unilatéralement les modifications effectuées dans une branche ?

6 Échange de patch

Rendre efficace l'échange des changements est un des objectifs principaux de Git. Une manière de le faire est d'avoir plusieurs repositoire et permettre aux gérants de chacun d'envoyer des commits d'un repositoire à un autre (voir prochaine section). Une autre manière est d'échanger des patch qui sont générés par Git à partir des commits. La *workflow* habituel dans ce cas est donc le suivant :

— `git clone`
— développement / `git commit` / développement / `git commit` / ...
— choix des commits à envoyer
— *stérilisation* des commits en format patch (avec la commande `git format-patch`
— envoi des patchs par mail
— intégrations des patch avec `git am` et/ou `git apply`

Question 6.1. *Travaillez à deux pour les exercices de cette section. À partir d'un repositoire Git (e.g. celui de purity) développez des changements indépendants (e.g. continuez la traduction de quelques questions dans des test différents). Ensuite, identifiez avec `git log` les numéros du premier et dernier commit que vous voulez envoyer à votre binôme. Notez les quelque part.*

Question 6.2. *Ce que vous devrez envoyer à votre binôme est une* série de patch *(*patch series*) à appliquer l'un après l'autre. Pour la produire exécutez la commande `git format-patch COMMIT1..COMMIT2` ou COMMIT1 est l'identifiant du premier commit (le plus vieux) à envoyer et COMMIT2 est l'identifiant du dernier (le plus récent). Git va vous produire une liste de fichiers, un pour chaque commit, qui commence avec les préfixes 0000, 0001, 0002, etc. Regardez le contenu de ces fichiers, en quel format sont ils écrits ?*

Question 6.3. *Envoyez les fichiers à votre binôme (par mail ou autre moyen). Entretemps, il aura fait la même chose avec vous.*

Question 6.4. *Grâce à la commande `git am` vous pouvez appliquer un ou plusieurs patchs dans votre repositoire et les committer atomiquement. En cas des conflits, Git vous demandera de les résoudre comme d'habitude. Alternativement, `git apply` applique un patch à la copie de travail courante, mais n'effectue pas le commit.*

Essayez les deux commandes et intégrez les changements de votre binôme avec les vôtres.

Comme travailler avec des identifiants de commits très longs n'est pas aisé, Git offre plusieurs astuces. D'abord vous pouvez utiliser seulement un préfixe court d'un identifiant de commit, tant que le préfixe n'est pas ambigu dans un repositoire [1]. Ensuite, vous avez des raccourcis, par exemple :
— `HEAD` correspond au dernier commit de la branche courante
— `HEAD^` correspond à l'*avant*-dernier commit de la branche
— `HEAD^^` correspond toujours à l'*avant-avant*-dernier commit de la branche (etc. pour autres ^)
— `COMMITID^` correspond au commit précédent le commit COMMITID ; donc HEAD^est un cas particulier de cette syntaxe
— `COMMITID~N` corresponde au Nième *avant*-dernier commit du commit COMMITID

7 Synchronisation de plusieurs repertoires

Jusqu'à présent, nous avons vu quelques fonctionalités de Git sans nous intéresser son aspect collaboratif. Git permet un travail collaboratif sur un dépôt. C'est-à-dire qu'il est possible de synchroniser entre elles des branches de deux dépôts différents.

Question 7.1. *Créez un nouveau dépôt (avec* `git init --bare`*).*

Ceci initialise un dépôt Git sans copie de travail. Il y a deux façons de synchroniser entre eux deux dépôts :
— soit en récupérant les commits du dépôt distant (*pull*)
— soit en envoyant des commits vers le dépôt distant (*push*).
Dans le deuxième cas, il faut que le dépôt distant soit un dépôt *bare*, c'est à dire sans copie de travail.

Question 7.2. *Envoyez les commits de votre premier dépôt vers le second avec les commandes (exécutées depuis votre premier dépôt) :*

```
git push file:///$PATH_TO_REPO2 master:master
```

Cette commande va envoyer la branche master du premier dépôt dans une branche appelée master dans le second dépôt. La seconde va effectuer la même chose avec la branche developpement.

```
git push file:///$PATH_TO_REPO2 developpement:developpement
```

Observez le résultat en lançant `gitk --all` depuis le second dépôt.

Pour rendre la synchronisation plus intéressante, nous allons utiliser une deuxième copie de travail.

Question 7.3. *Créez une nouvelle copie de travail à partir du dépôt* bare *:*

```
git clone file:///$PATH_TO_REPO2 copietravail
```

qui crée une copie de travail du second dépôt dans le répertoire `copietravail`.

1. Dans un repositoire contenant 256 commits, il y a une probabilité de 50% qu'il existe 2 commits avec le même prefixe de 4 lettres. Il y a une chance sur deux que dans un repositoire de 65.000 commits il y en ait 2 qui aient le même prefixe de 8 lettres.

Question 7.4. *Effectuez quelques modifications dans votre première copie de travail. Propagez ces modifications dans votre troisième dépôt :*

— *Envoyez ces modifications dans le second dépôt (avec* `git push file:///PATH_TO_REPO2 BRANCHE:BRANCHE`*)*
— *Puis depuis le troisième dépôt, récupérez avec* `git pull`[2] *ces modifications depuis le second dépôt.*

Nous avons mis en place avec les seconds et troisièmes dépôts le schéma de collaboration avec Git le plus courant : il y a un dépôt qui fait office de dépôt maître, et le troisième dépôt qui peut récupérer et envoyer des commits sur le dépôt maître. Nous allons maintenant nous intéresser à l'accès concurrent à ce dépôt maître.

Question 7.5. *Effectuez des modifications dans le premier dépôt, et envoyez ces modifications dans le second dépôt. Sans synchroniser le troisième dépôt avec le second, effectuez (et commitez) des modifications dans le troisième dépôt. Que se passe-t-il maintenant lorsqu'on fait* `git pull` *dans le troisième dépôt ?*

Question 7.6. `git pull` *peut être décomposé en* `git fetch` *suivi de* `git merge`*. Réitérez le scénario de la question précédente mais en faisant* `git fetch` *au lieu de* `git pull`*, observez "toutes les branches" du dépôt 3 dans* `gitk --all`*.*

Un des aspects fondamental de Git est qu'il est décentralisé. Nous avons ici donné un rôle spécial de dépôt central au dépôt 2, mais s'il venait à disparaître, il serait toujours possible de synchroniser entre eux les dépôts 1 et 3.

Question 7.7. *Dans le dépôt 1, nous allons déclarer l'addresse du dépôt 3, nous allons créer pour cela une* remote *appelée repo3.*

```
git remote add repo3 file:///chemin/vers/repo3
```

De même, dans le dépôt 3, nous allons créer une remote *appelée repo1 qui pointe vers le premier dépôt.*

Question 7.8. *Effectuez (et commitez) des modifications dans le dépôt 3 et récupérez-les dans le dépôt 1 au moyen de :*

```
git fetch repo3
git checkout master
git merge remotes/repo3/master
```

Nous pouvons simplifier cette démarche en déclarant que la branche master du dépôt 1 "suit" la branche master du dépôt 3, à l'aide de (depuis le dépôt 1) `git branch master --set-upstream repo3/master`.

Question 7.9. *Effectuez des modifications dans le dépôt 1 puis récupérez-les dans le dépôt 3 au moyen de* `git pull`*.*

Question 7.10. *Synchronisez entre eux les trois dépôts.*

8 Modifications publiées, modifications non publiées

Nous avons vu que les objets que s'échangent les dépôt gits sont des commits. Afin de maintenir une intégrité des arbres de commit, Git utilise des primitives cryptographiques. Chaque commit est en fait signé, en fonction du patch qu'il représente, du nom d'auteur, de la date de création, et aussi de la signature du commit parent (ou des deux parents, dans le cas d'un commit de fusion). Cette signature est un hachage SHA1 de toutes ces informations, et il est possible de se référer à un commit uniquement par cette signature (de la forme f4ccba7ba89d4f6f8f0853056d47912c640a19c1) ou par un préfixe non ambigu de celle-ci (f4ccba7b).

Ainsi Git n'appliquera pas un commit ailleurs que sur son père. L'utilisateur de Git pourra vouloir appliquer un commit ailleurs dans l'arbre de commit (par exemple sur une autre branche), il va pour cela devoir créer un nouveau commit (c'est à dire avec un SHA1 différent) mais qui contient les mêmes modifications.

Supposons que nous clonions un dépôt (qu'on appelera repo et nommons commitA le dernier commit sur ce dépot), et que nous effectuions un commit dans notre copie de travail (commitB dont le père est commitA). Parallèlement, un autre développeur effectue un commit commitC au dessus du commitA et envoie ce commit dans le dépôt repo. Il n'est plus possible d'envoyer notre commit commitB, car le dernier commit du dépot repo n'est pas commitA, mais commitC. La stratégie consiste alors à fusionner notre branche locale avec la branche distante, ce qui va créer un commit de fusion, fils de commitB et commitC et d'envoyer ce commit et le commit commitB vers repo.

Nous allons voir qu'une autre stratégie est possible, de demander (avec la commande rebase) à Git de "modifier" le commitB pour que son père soit commitC

Question 8.1. *Implémentez ce scénario, constater que* `git push` *renvoie une erreur, puis au lieu d'invoquer* `git merge`*, invoquez* `git rebase origin/master` *(origin/master étant le nom de la branche distante avec laquelle on voudrait normalement effectuer un merge). Git va recréer les commits de votre branche master qui ne sont pas dans la branche master du dépôt repo et va les placer au dessus du dernier commit de la branche master de repo).*

2. Comme le troisième dépôt a été créé à partir du second au moyen de `git clone`, il n'est pas nécessaire de préciser ici où Git doit chercher les commits.

Attention, il est très fortement déconseillé de rebase des commits qui ont déjà été publiés, c'est à dire présent sur un autre dépôt. La question suivante va donc vous montrer ce qu'il faut éviter de faire.

Question 8.2. *Synchonisez vos 3 dépôts. Dans le dépôt 3, effectuez un commit. Publiez-le dans le dépot 2. Dans le dépôt 1 effectuez d'autres commits, récupérez ce commit dans le dépôt 3 à l'aide de* `git fetch repo1`[3]*, puis (dans le dépôt 3) rebasez votre branche master au dessus de la branche repo1/master avec* `git rebase repo1/master`*.*

Que se passe-t-il si on essaie de merger cette branche avec la branche présente dans le dépôt 2 ?

Un autre cas de modification de commit est avec la sous-commande amend de Git. amend permet d'éditer, de modifier le contenu d'un commit. Supposons qu'on vienne de commiter un commit intitulé orthographe et qu'il corrige des fautes d'orthographes. Supposons qu'une faute ne soit pas corrigée par ce commit, et qu'on ne veuille pas créer un autre commit par dessus, il est possible de modifier le dernier commit avec `git commit --amend`.

Question 8.3. *Créez un commit, puis apportez d'autres modifications, et éditez le précédent commit au lieu d'en créer un nouveau.*

Là encore, il est impératif de ne pas éditer un commit qui a déjà été publié, c'est ce que la dernière question vous demande de faire et qu'il faut éviter de faire :

Question 8.4. *Créez un commit, envoyez-le vers un dépôt distant, puis amendez votre commit, synchronisez votre dépôt avec le dépôt distant.*

3. car nous avons déclaré une remote appelée repo1 dans le dépôt 3 qui pointe vers le premier dépôt.