Table des matières

1 Préambule

Git permet de gérer un projet (de programmation généralement) et de garder en mémoire l'historique de toutes les versions d'un ensemble de fichiers. Il permet de gérer un projet à plusieurs, de programmer afin de pouvoir revenir en arrière, comparer avec d'anciennes versions et cie.

Le principe est d'avoir un serveur git (un seul possible) qui va garder en mémoire l'historique de toutes les versions et un client git (plusieurs possibles) qui vont se connecter au serveur pour mettre à jour la version des fichiers ou en récupérer les dernières versions.

Remarque : Il est possible que le serveur soit lui aussi client, dans le cas où il n'y aurait qu'un seul développeur et qu'on ne souhaite pas passer par internet.

1.1 Configuration

Afin d'avoir la coloration syntaxique, il faut faire :

```
git config --global color.diff auto
git config --global color.status auto
git config --global color.branch auto
```

De même, il faut configurer votre nom (ou pseudo) :

```
git config --global user.name "votre_pseudo"
```

Puis votre e-mail :

```
git config --global user.email moi@email.com
```

Vous pouvez aussi éditer votre fichier de configuration .gitconfig situé dans votre répertoire personnel pour y ajouter une section alias à la fin :

```
vim ~/.gitconfig
```

```
[color]
        diff = auto
        status = auto
        branch = auto
[user]
        name = votre_pseudo
        email = moi@email.com
[alias]
        ci = commit
        co = checkout
        st = status
        br = branch
```

1.2 Git sur internet

Je vais prendre l'exemple de google code, qui est celui que j'ai choisi et que je suis en train d'apprendre.

Une fois le projet créé (sur la page http://code.google.com/hosting/createProject), il faut faire la commande suivante pour récupérer le contenu du projet localement (et pouvoir envoyer les modifs par la suite) :

```
cd Formulaire
git clone https://autiwa@code.google.com/p/autiwa-tutorials/
```

Cette commande permet de récupérer le contenu du projet et de le copier dans un dossier **Formulaires** qui sera créé dans le dossier courant.

Définition 1 (Clone)
Opération d'extraction d'une version d'un projet du repository vers un répertoire de travail local.

1.3 Git localement

Le serveur ET le client seront alors sur la même machine. Pour créer un projet, il faut créer un dossier, par exemple dans mon cas un dossier **mercury** dans mon **$HOME**, puis je fais, dans mon répertoire utilisateur :

```
cd mercury
git init
```

pour un projet que j'appelle **mercury**.

2 Commandes universelles git

Ici, je note les commandes qui sont valables à la fois pour svn installé sur un serveur internet, ou sur une machine locale pour un usage personnel.

2.1 Commandes de base

2.1.1 Ajouter des fichiers ou dossiers au projet : add

Pour ajouter des fichiers il faut faire :

```
git add latex/ vim/
```

où **latex**/ et **vim**/ sont deux dossiers existant dans le dossier local de référence

Remarque : Au cas où ça serait pas clair. J'ai créé un dossier /**home**/**autiwa**/**Formulaires** grâce à [§ 1.3]. Dans ce dossier, j'ai créé et rempli à la main les sous-dossiers **latex**/ et **vim**/. Maintenant, grâce à la commande ci-dessus, je définis ces sous-dossiers comme étant rattachés au projet. En faisant ainsi le contenu est rajouté récursivement.

⚠ Cette commande n'agit que sur le répertoire local (la *working copy*). Il faut ensuite appliquer ces changements au dépot (voir [§ 2.1.2]) pour les valider.

Supposons que vous veniez d'ajouter un fichier à Git avec git add et que vous vous apprêtiez à le « commiter ». Cependant, vous vous rendez compte que ce fichier est une mauvaise idée et vous voudriez annuler votre git add.

Il est possible de retirer un fichier qui avait été ajouté pour être « commité » en procédant comme suit :

```
git reset HEAD -- fichier_a_supprimer
```

2.1.2 Enregistrer les modifications localement : commit

Pour mettre à jour les versions sur serveur à partir des modifications effectuées localement, il faut sélectionner les fichiers qu'on veut ajouter au prochain commit avec **git add** puis faire le commit :

```
git add FILE
git commit -m "initialisation"
```

où **"initialisation"** est le commentaire qui décrit la mise à jour et les modifications effectuées. **FILE** peut être un ou plusieurs fichiers, l'astérisque pouvant être utilisée.

À noter que l'option -a permet de se passer du **git add** et d'ajouter automatiquement tous les fichiers qui ont été modifiés :

```
git commit -a -m "initialisation"
```

Remarque : Si on a fait un 'add' par erreur et qu'on ne veut pas ajouter un certain fichier au prochain commit, il suffit de faire :

```
git reset FILE
```


Un commit avec git est local : à moins d'envoyer ce commit sur le serveur comme on apprendra à le faire plus loin, personne ne sait que vous avez fait ce commit pour le moment. Cela a un avantage : si vous vous rendez compte que vous avez fait une erreur dans votre dernier commit, vous avez la possibilité de l'annuler (ce qui n'est pas le cas avec SVN !).

2.1.3 Identifier un commit

Pour indiquer à quel commit on souhaite revenir, il existe plusieurs notations :

- `HEAD` : dernier commit ;
- `HEAD^` : avant-dernier commit ;
- `HEAD^^` : avant-avant-dernier commit ;
- `HEAD~2` : avant-avant-dernier commit (notation équivalente) ;
- `d6d98923868578a7f38dea79833b56d0326fcba1` : indique un numéro de commit précis ;
- `d6d9892` : indique un numéro de commit précis (notation équivalente à la précédente, bien souvent écrire les premiers chiffres est suffisant tant qu'aucun autre commit ne commence par les mêmes chiffres).

Exemple :

```
git diff HEAD^
```

permet de comparer la version courante avec la version juste avant.

2.1.4 Mettre à jour sa copie locale : pull

```
git pull
```

Pour celà, il faut que le dossier dans lequel on se trouve ait déjà été défini comme un dossier git via un *clone* (voir [§ 1.3 page précédente])

2.1.5 Propager les modifications locales sur le dépot du serveur : push

On peut modifier ces commits là, tant qu'ils sont en local. Puis une fois vérifié. Il ne reste plus qu'à les envoyer sur le serveur. Pour celà on vérifie tout d'abord ce qu'on s'apprête à envoyer, et on envoie sur le serveur :

```
git log -p
git push
```

2.2 Annuler ou nettoyer

2.2.1 Annuler les modifications non commités : checkout/reset

Pour annuler les modifications locales et revenir à la dernière révision :

```
git checkout *.f90
```

va remettre en l'état tous les fichiers du dépot qui sont présents dans le dossier courant (on peut aussi ne préciser qu'un seul fichier).

```
git reset --hard
```

va nettoyer le dépot pour le remettre dans l'état du dernier commit, effaçant d'un coup toutes les modifications non sauvegardées.

2.2.2 Annuler un git add : git reset

Parfois, on peut avoir préparé des modifications à ajouter à un commit, via **git add**, soit sur des fichiers déjà suivis, soit pour suivre de nouveaux fichiers. Afin de retirer certains de ces fichiers de la liste des fichiers pour le prochain commit, il faut faire :

```
git reset HEAD <FILE>
```

2.2.3 Modifier le message du dernier commit

Valable si on n'a pas fait de *git push* :

```
git commit --amend
```

2.2.4 Annuler un commit pas encore publié

Valable si on n'a pas fait de *git push* :

```
git reset HEAD^
```

annule le dernier commit et revient à l'avant dernier. Pour autant les fichiers ne sont pas modifiés, seul le commit en lui même est annulé.

Pour annuler les modifications liées au commit, il faut faire un hard reset :

```
git reset --hard HEAD^
```

Annule le dernier commit et toutes les modifications qui s'y rapportent.

2.2.5 Annuler un commit publié

Si vous publiez un commit sur le serveur, mais que vous souhaitez l'annuler, c'est quand même possible, mais c'est un peu plus barbare qu'un commit non encore propagé (à l'aide de push).

Pour celà il faut créer un nouveau commit qui annule les modifications du commit que vous souhaitez annuler. Il faut alors connaître l'ID du commit visé et faire :

```
git revert 6261cc2
```

Il faut préciser l'ID du commit à « revert ». Il n'est pas obligatoire d'indiquer l'ID en entier (qui est un peu long), il suffit de mettre les premiers chiffres tant qu'ils sont uniques (les 4-5 premiers chiffres devraient suffire). On vous invite à entrer un message de commit. Un message par défaut est indiqué dans l'éditeur.

Une fois que c'est enregistré, le commit d'annulation est créé. Il ne vous reste plus qu'à vérifier que tout est bon et à le publier (avec un git push).

2.2.6 Mettre de coté et reprendre un travail en cours : stash

Si on est en train de faire une modification, sans faire de commit, mais qu'on doit faire des modifications plus urgentes, et surtout immédiate, on peut souhaiter ne pas perdre le travail déjà effectué. Au lieu de faire un commit on peut simplement faire une sauvegarde de l'état modifié, pour le restituer plus tard.

```
git stash save "ongoing work about something"
```

va sauvegarder les modifications effectuées depuis le dernier commit dans le **stash**, et remettre les fichiers dans leur état lors du dernier commit.

On peut ensuite faire les modifications urgente, dans cette branche ou dans une autre, faire des commit etc. Puis une fois qu'on veut réutiliser le stash, il suffit de faire :

```
git stash apply
```

2.2.7 Supprimer les fichiers inconnus de Git

Pour supprimer les fichiers **untracked**, il faut utiliser la commande *clean*.

D'abord on simule la suppression pour voir ce qu'il va supprimer (option **-n**) :

```
git clean -n
```

-d Pour supprimer aussi les dossiers

-x Pour supprimer les fichiers ignorés par Git (**.gitignore**)

-X Pour **ne** supprimer **que** les fichiers ignorés par Git (**.gitignore**)

-f Par défaut, il faut cette option pour supprimer les fichiers, ça évite de faire le nettoyage par erreur.

Pour supprimer les fichiers, une fois qu'on a vu ce qu'il allait faire :

```
git clean -f
```

ou

```
git clean -f <PATH>
```

si on souhaite faire cela dans un dossier particulier.

2.3 Contrôler

2.3.1 État du dépot local (modifications par rapport au dernier commit) : status

La commande git status vous indique les fichiers que vous avez modifiés récemment :

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

2.3.2 Historique des modifications : log

La commande `log` permet de voir un historique des commentaires de révision, soit de l'ensemble des fichiers, soit d'un fichier en particulier.

```
git log
```

affiche l'historique de toutes les révisions.

 Chaque commit est numéroté grâce à un long numéro hexadécimal comme 12328a1bcbf231da-8eaf942f8d68c7dc0c7c4f38. Cela permet de les identifier.

Pour avoir les informations d'un commit particulier, il faut faire :

```
git show 99adb32062d66121f2ba056dc820b9529a9ea08c
```

2.3.3 Différences entre versions : diff

```
git diff
```

Remarque : Il est possible de regarder les différences sur un fichier en particulier.

Pour regarder les différences avec le commit précédent, il suffit de faire :

```
git diff HEAD^
```

On peut faire des diff entre des branches

```
git diff master branch_devel
```

ou entre deux versions en spécifiant leur hashtag :

```
git diff dqsfg54qsdf35dqs4f ze9r8az7er3azer2
```

J'utilise personnellement *difftool*, ça me permet d'utiliser *meld* que j'affectionne particulièrement.

2.3.4 Revenir à une révision antérieure : checkout

Pour changer la révision courante, il suffit de faire :

```
git checkout c23db4a1d05e
```

et la révision **c23db4a1d05e** devient la révision active.

Pour revenir à la dernière révision d'une branche donnée, il suffit de faire :

```
git checkout nom_de_branche
```

ainsi

```
git checkout master
```

active la dernière révision de la branche principale.

3 Travailler avec des branches : branch

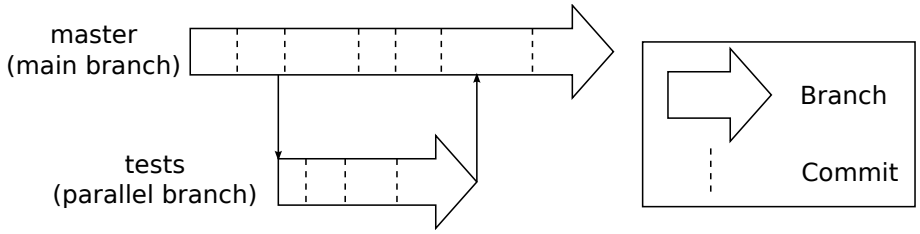

FIGURE 1 – Le principe des branches est de travailler à part sur des modifications et de ne les fusionner avec la branche principale que si c'est ok et ça donne quelque chose.

L'idée derrière le fait de créer des branches, c'est de bien séparer les idées de développement, et de pouvoir les tester et les séparer avant de les inclure (ou pas) dans la branche principale.

> En gros, Si vous avez une modification à faire, qui risque de prendre un peu de temps et qui va nécessiter plusieurs commit, faites une branche !

3.1 Lister les branches

Première chose utile, lister les branches existantes. Par défaut il n'y aura que la branche principale :

```
git branch
```

3.2 Créer une branche

Pour créer une branche **test_01**, il faut utiliser la commande :

```
git branch test_01
```

Si vous listez alors les branches, vous aurez :

```
$ git branch
* master
  test_01
```

où on voit que **master** est toujours la branche active. Il faut donc changer de branche pour pouvoir modifier les fichiers et faire des commits dans cette branche.

3.3 Gérer une branche distante

Pour que cette branche soit envoyée et synchronisée avec le dépot distant, si on veut en fair eune branche permanente, il faut lancer la commande :

```
git push origin test_01
```

Pour que quelqu'un puisse **suivre** les modifications de cette branche, il devra utiliser la commande suivante :

```
git checkout -b ma-branche origin/ma-branche
```

Ce qui revient à créer dans le nouveau dépôt créé une branche **ma-branche** qui suivra celle du dépôt distant **origin/ma-branche**.

On peut de plus supprimer une branche distante en faisant :

```
git push origin :test_01
```

3.4 Aller dans une branche et y faire des commits

Il faut déclarer cette branche comme active via :

```
git checkout test_01
```

La liste des branches donne alors :

```
$ git branch
  master
* test_01
```

qui montre que la branche **test_01** est maintenant active.

Remarque : **git checkout** est utilisé pour changer de branche mais aussi pour restaurer un fichier tel qu'il était lors du dernier commit. La commande a donc un double usage.

⚠ Vous pouvez changer d'une branche à l'autre, faire des commits (éventuellement des push) sur la branche principale, mais ceux ci ne seront pas propagés aux autres branches, c'est d'ailleurs le principe des branches.

3.5 Déplacer des modifications non commités dans une autre branche

Parfois, on peut se tromper de branche quand on fait des modifications. Deux solutions. Soit on change directement de branche, les modifications qui ne font pas partie d'un *commit* sont alors déplacées automatiquement, il ne reste plus qu'à faire un commit.

Soit on utilise *git stash* [§ 2.2.6 page 5].

3.6 Fusionner une branche avec la branche principale

Pour celà, il faut se placer dans la branche principale et demander d'intégrer les changements d'une autre branche en faisant :

```
git checkout master
git merge test_01
```

Une fois fusionné, vous pouvez supprimer la branche **test_01** (voir la section suivante).

3.7 Incorporer les nouvelles modifications de la branche principale dans une branche annexe : rebase

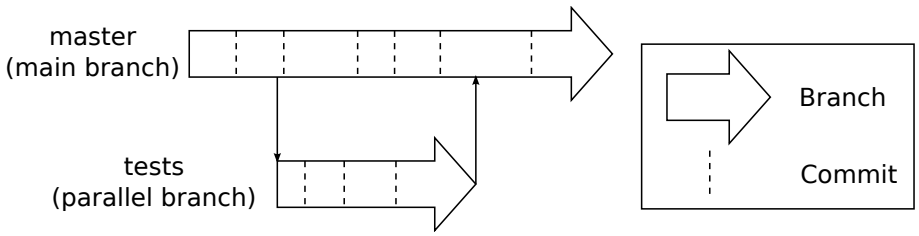

FIGURE 2 – Avec git rebase, on peut inclure les dernières modifications d'une branche dans une autre branche, afin de décaler la « base » de cette branche. En résumé, c'est comme si on faisait bifurquer la branche concernée à la fin de la branche de laquelle on veut inclure les modifications

⚠ Ceci n'est utile que si on souhaite plus tard fusionner cette branche avec la branche principale. Sinon, *merge* est suffisant, et beaucoup plus pratique à utiliser. *rebase* est loin d'être une opération anodine.

Pour inclure les dernières modifications de **master** dans **devel**, il faut se placer dans la branche **devel**, et faire :

```
git rebase master

git rebase master devel
```

permet d'inclure dans **devel** les dernières modifications de **master** quelle que soit la branche courante.

En cas de conflits, on peut modifier les fichiers affectés et continuer via la commande

```
git rebase --continue
```

ou annuler directement avec la commande

```
git rebase --abort
```

3.8 Supprimer une branche

Pour supprimer une branche dont vous avez déjà propagé les modifications dans la branche principale, faites :

```
git branch -d test_01
```

Si jamais vous souhaitez supprimer une branche sans avoir propagé les changements (si cet essai était une erreur par exemple) vous pouvez utiliser la commande suivante qui force la suppression :

```
git branch -D test_01
```

4 Gérer un conflit

Un conflit survient quand on veut mettre à jour le dépot :

```
git pull
remote: Counting objects: 7, done.
remote: Finding sources: 100% (7/7), done.
remote: Total 7 (delta 0)
Unpacking objects: 100% (7/7), done.
From https://code.google.com/p/blablabla
   93ad6a9..a10e8d1  master     -> origin/master
Auto-merging recettes/recettes.pdf
CONFLICT (content): Merge conflict in recettes/recettes.pdf
Automatic merge failed; fix conflicts and then commit the result.
```

L'idée pour résoudre le conflit est de modifier le fichier de conflit, soit en choisissant –**ours** ou –**theirs**, soit en utilisant **mergetool**, puis d'ajouter le fichier au dépot via un nouveau commit, et ensuite d'essayer de mettre à jour de nouveau.

4.1 Écraser un fichier par un autre

On peut garder soit le fichier local, soit le fichier distant. C'est la partie la plus facile. Pour garder notre fichier :

```
git checkout --ours filename.c
git add filename.c
git commit -m "using theirs"
```

Pour garder leur version :

```
git checkout --theirs filename.c
git add filename.c
git commit -m "using theirs"
```

4.2 Regarder les différences et en tenir compte

Si on veut voir le contenu des fichiers et créer notre futur fichier à partir des deux fichiers du conflit, on a :

```
git mergetool
```

Mais ceci ne va fonctionner qu'avec des fichier ASCII. Dans mon cas, j'avais un conflit avec un fichier .pdf, et dans ce cas, mergetool ne fonctionne pas, il faut choisir si on veut le fichier distant ou le fichier local. Dans tous les cas ce sont normalement des fichiers qui sont aisément générable, le soucis majeur étant de ne plus être embêté par le conflit.

Remarque : J'utilise **meld**, que j'aime beaucoup, et qui marche autant pour les diff (**difftool**) que pour les conflits (**mergetool**).

5 Avancé

5.1 Tagger une version : tag

Il est possible d'associer un tag (un nom de code) à un commit particulier, afin de le mettre en valeur, pour lui donner un numéro de version par exemple, pour les versions stables que les gens devraient utiliser.

Pour ajouter un tag sur un commit :

```
git tag NOMTAG IDCOMMIT
```

ou avec un message explicatif :

```
git tag NOMTAG IDCOMMIT -m "message associe au tag"
```

Donc dans le cas présent, on écrirait :

```
git tag v1.3 2f7c8b3428aca535fdc970feeb4c09efa33d809e
```

Un tag n'est pas envoyé lors d'un push, il faut préciser l'option –**tags** pour que ce soit le cas :

```
git push --tags
```

Maintenant, tout le monde peut référencer ce commit par ce nom plutôt que par son numéro de révision.

Pour supprimer un tag créé :

```
git tag -d NOMTAG
```

Pour lister les tags existants :

```
git tag -l
```

On peut aussi lister les tags correspondant à un motif particulier :

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

Pour avoir les infos sur le tag **nomdutag** :

```
git show nomdutag
```

5.2 Ignorer des fichiers (.gitignore)

Pour ignorer un fichier dans git, créez un fichier **.gitignore** (à la racine) et indiquez-y le nom du fichier. Entrez un nom de fichier par ligne, comme ceci :

```
project.xml
dossier/temp.txt
*.tmp
cache/*
```

Aucun de ces fichiers n'apparaîtra dans git status, même s'il est modifié. Il ne paraîtra donc pas dans les commits. Utilisez cela sur les fichiers temporaires par exemple, qui n'ont aucune raison d'apparaître dans Git.

Il est possible d'utiliser une étoile (*) comme joker. Dans l'exemple précédent, tous les fichiers ayant l'extension **.tmp** seront ignorés, de même que tous les fichiers du dossier cache.

5.3 Rechercher dans les fichiers du dépot : git grep

On peut vouloir chercher quelque chose parmis les fichiers du dépot. Une commande très simple pour faire ça :

```
git grep "xmax"
```

qui va donner toutes les occurences par fichier. Mais si on veut aussi les numéros de lignes, il suffit alors de faire :

```
git grep -n "xmax"
```

Remarque : À noter que les expressions régulières sont possibles.

5.4 Rechercher un bug en utilisant la puissance de GIT : bisect

J'ai eu le cas tout à l'heure et j'ai découvert la puissance de **bisect**, je vais tâcher de l'expliquer du mieux possible. Vous avez un bug dans votre code, vous ne savez pas d'où vient le bug, **bisect** est fait pour vous.

Le principe est le suivant : Vous avez une révision que vous savez ne pas fonctionner (la dernière par exemple) et une qui fonctionne (au pire la première si vous en avez peu, sinon vous en trouvez une assez ancienne et qui fonctionne. Personnellement j'ai fini par y aller franco, et je suis remonté six mois en arrière, une centaine de révisions à tester, mais avec **bisect** c'est rapide. Pour trouver cette révision, j'ai fait **git checkout** (qui permet de revenir à une révision antérieure [§ 2.3.4 page 6]) jusqu'à ce que je trouve une révision qui fonctionne.

Une fois fait, on peut exploiter la puissance de **bisect**, qui va nous aider à trouver par dichotomie la révision qui a introduit le bug. On part donc d'un encadrement en ayant une version bonne de référence et une version mauvaise, et on sait donc que la révision fautive se trouve au milieu.

La première étape est d'avoir une version de référence propre, il ne faut pas qu'il reste des modifications non encore sauvées dans un commit. Si vous voulez mettre des modifs de cotés, utilisez **git stash** :

```
git stash
```

Ensuite, on démarre l'opération de traque du bug via :

```
git bisect start
```

On déclare le **mauvais** commit :

```
git bisect bad # le commit actuel
```

On déclare le **bon** commit :

```
git bisect good 3776c939294f
```

où **3776c939294f** est la référence du commit en question.

Une fois fait, vous aurez un message qui ressemblera à ça :

```
Bisecting: 65 revisions left to test after this
```

bisect nous place automatiquement dans des commit différents, indique combien il reste de révisions à tester, et le nombre d'étape estimée pour arriver à trouver l'origine du bug.

Vous devez donc tester si cette version bugue ou pas. L'idéal est d'avoir un script qui permet de voir rapidement si ça bugue. J'avais pour ma part un script python de tests unitaires que je lançais à chaque fois.

Une fois que vous savez si la version courante bugue ou pas, il suffit de le dire à bisect via :

```
git bisect good
```

ou

```
git bisect bad
```

et en fonction de ça, bisect vous mettre une autre révision à tester. Une fois qu'il aura trouvé LA révision fautive, il le dira par :

```
d109d47732cb85652b79d679edd7bfe2379e5707 is first bad commit
```

 Si pour une raison ou pour une autre, un commit particulier vous pose problème (problème de compilation ou impossibilité de tester, il est possible de dire à bisect qu'on souhaite passer ce commit et en tester un autre à la place via :

```
git bisect skip
```

Pendant la bisection, git créé une branche spéciale dédiée. N'oubliez pas de repasser sur votre branche de développement quand vous aurez débusqué la régression. Il est possible de le faire simplement en tapant :

```
git bisect reset
```

Remarque : La première fois où j'ai utilisé bisect, j'ai oublié de faire **git bisect reset** et je me suis retrouvé sur une branche **(no branch)** sans m'en rendre compte. J'ai donc fait des modifications là, sans réaliser que je ne modifiais plus **master**. Pour plus d'infos quant à la solution à ce problème, voir [§ 5.6 page suivante].

Pour aller plus loin, il est possible de carrément automatiser la tâche si vous arrivez à faire un script qui peut tester le bug, et qui retourne 0 si le code est correct, 1 si le code est incorrect, et 125 si on ne peut pas tester le code (**git skip**). Pour le faire, il suffit de procéder ainsi :

```
git bisect start HEAD <bad_commit> -- # raccourci
git bisect run script
```

5.5 Afficher la branche dans le prompt de la console

Avec Bash, il est possible d'afficher la branche active quand on arrive dans une partie sous gestionnaire de version GIT avec la commande suivante, rajoutée dans le **.bash_profile** ou le **.bashrc** :

```
1  git_branch_name_prompt() {
2      git_status_output=$(git status 2> /dev/null) || return
3
4      branch_name() {
5          sed -n 's/# On branch //p' <<< "$git_status_output"
6      }
7
8      echo -e "($(branch_name))"
9  }
10
11 git_branch_colour_prompt() {
```

```
12      git_status_output=$(git status 2> /dev/null) || return
13
14      find_pattern_in_status() {
15          local pattern="$1"
16          [[ "$git_status_output" =~ ${pattern} ]]
17      }
18
19      is_clean() {
20          find_pattern_in_status '(working directory clean)'
21      }
22
23      is_local_changes() {
24          local added='# Changes to be committed'
25          local not_added='# Changes not staged for commit'
26          find_pattern_in_status "($added|$not_added)"
27      }
28
29      is_untracked() {
30          find_pattern_in_status '# Untracked files'
31      }
32
33      # local bold="\033[1m"
34      local no_colour="\033[0m"
35
36      local red="\033[31m"
37      local green="\033[32m"
38      local yellow="\033[33m"
39      local branch_colour=""
40
41      if is_untracked
42      then
43          branch_colour=$red
44      elif is_local_changes
45      then
46          branch_colour=$yellow
47      elif is_clean
48      then
49          branch_colour=$green
50      fi
51      # echo -e "$branch_colour$(branch_name)$no_colour"
52      echo -e "$branch_colour"
53  }
54
55  PS1="\h.\u\[\$(git_branch_colour_prompt)\]\$(git_branch_name_prompt)\[\033[0m\]> "
```

Le résultat sera alors :

`arguin.login(master)>`

si on est dans un dossier GIT, **master** étant le nom de la branche active, et :

`arguin.login>`

si le dossier courant n'est pas un dossier GIT.

La couleur sera fonction du status du dépot. Si le dépot est à jour, ce sera vert, s'il y a eu des changements locaux, jaune, et s'il y a eu des commits locaux qui n'ont pas encore été propagés (push), ce sera orange. S'il y a des fichiers qui ne sont pas suivis, ce sera rouge (pour ignorer des fichiers il faut se servir du fichier .gitignore, voir [§ 5.2 page 11]).

Remarque : La difficulté est ici d'afficher les couleurs. Il faut échapper ces caractères pour que le retour à la ligne et l'effacement de la ligne se fasse de manière correcte. On est ainsi obligé d'afficher et d'échapper les couleurs au tout dernier moment, dans PS1, et on ne peut donc pas encapsuler ces fonctions dans une grosse fonction qu'on appelerait par la suite.

5.6 branche (no branch), récupérer les commits

Par exemple avec bisect, on peut sans faire exprès faire des commits dans une branche **(no branch)** et se retrouver dans **master** sans rien de nouveau. Passé le coup de stress, voici ce qu'on peut faire pour facilement récupérer les données.

La commande **reflog** permet d'avoir un historique de ce qu'on a fait et donc de récupérer le **hash-ID** de la branche **(no branch)** qui n'apparait plus une fois qu'on est repassé dans **master** :

```
$ git reflog
934101e HEAD@{0}: commit: improving some scripts
c23db4a HEAD@{1}: checkout: moving from 99adb32062d66121f2ba056dc820b9529a9ea08c to master
99adb32 HEAD@{2}: commit: improving some scripts
70ca613 HEAD@{3}: commit: Modification of the formula for the feedback of the eccentricity
```

La ligne qui nous intéresse est celle-ci :

```
checkout: moving from 99adb32062d66121f2ba056dc820b9529a9ea08c to master
```

C'est la ligne qui montre où on est passé depuis **(no branch)** jusqu'à **master**, en faisant

```
git checkout master
```

Remarque : Si on est encore dans **(no branch)** on peut directement noter de **hash-ID** en fait.

Une fois dans master, on peut alors faire :

```
git merge 99adb32062d66121f2ba056dc820b9529a9ea08c
```

pour récupérer toutes les modifications de cette branche.

5.7 Créer des patchs et les appliquer

Il est possible de créer des patchs, qui vont contenir différentes modifications à appliquer où on veut. Dans la pratique, je me suis servi de ça pour appliquer des modifications sur certains fichiers (sur plusieurs centaines de révisions) afin de déporter ces modifications sur une autre branche.

Pour faire court, si on veut suivre les modifications d'un ou plusieurs fichiers bien précis (et ignorer les autres) pour appliquer ces modifications ailleurs, il faut faire :

```
git format-patch --output-directory myPatch 7552b3a2fb57..dabfc11cce3884fa1e
 -- user_module.f90 bessel.f90
```

Ici, toutes les modifications des fichiers **user_module.f90** et **bessel.f90** contenues entre les révisions **7552b3a2fb57** et **dabfc11cce3884fa1e** seront enregistrées dans des patchs différents. Ces patchs seront ensuite stockés dans le sous-dossier **myPatch**.

Pour appliquer les patchs, il suffit de faire :

```
git apply myPatch/*.patch
```

Remarque : Comme je voulais des choses un peu propre, j'ai fait un script python qui applique les patchs un par un et qui fait un commit pour chacun d'entre eux. En effet, le patch va appliquer des modifications, mais il ne va pas faire de commit (comme pourrait le faire un **cherry-pick**).

Vous trouverez ci-dessous le code du script python (même si vous n'avez pas le module **autiwa** qui contient la fonction **lancer_commande**, il suffit d'avoir une fonction équivalente qui lance une commande système) :

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # will apply a list of patch separately, and create one commit for each single patch
4
5  import autiwa
6  import pdb
7
8  (process_stdout, process_stderr, return_code) = autiwa.lancer_commande("ls ../myPatch
       /*.patch")
9  if (return_code != 0):
10   print("the command return an error "+str(return_code))
11   print(process_stderr)
12   exit()
13
14 patches = process_stdout.split("\n")
15 patches.remove('') # we remove an extra element that doesn't mean anything
16 nb_patchs = len(patches)
17
```

```
18 patches.sort()
19
20 for patch in patches:
21   # Apply patch
22   cmd = "git apply %s" % patch
23   autiwa.lancer_commande(cmd)
24
25   # Get corresponding hashID
26   patchText = open(patch, 'r')
27   tmp = patchText.readline()
28   patchText.close()
29   hashID = tmp.split()[1]
30
31   # Get log text
32   cmd = 'git log --pretty=format:"%%s" %s -1' % hashID
33   (logText, process_stderr, return_code) = autiwa.lancer_commande(cmd)
34
35   # Get untracked files
36   cmd = 'git ls-files --other --exclude-standard'
37   (process_stdout, process_stderr, return_code) = autiwa.lancer_commande(cmd)
38
39   if (return_code != 0):
40     print("the command return an error "+str(return_code))
41     print(process_stderr)
42     exit()
43
44   untracked_files = process_stdout.split("\n")
45   untracked_files.remove('') # we remove an extra element that doesn't mean anything
46
47   # Add untracked files
48   cmd = 'git add %s' % (' '.join(untracked_files))
49   (process_stdout, process_stderr, return_code) = autiwa.lancer_commande(cmd)
50
51   # Commit the modif
52   cmd = 'git commit -a -m "%s"' % logText
53   autiwa.lancer_commande(cmd)
```

6 FAQ

6.1 fatal : git checkout : updating paths is incompatible

```
fatal: git checkout: updating paths is incompatible with switching branches/forcing
Did you intend to checkout ‘origin/‘ which can not be resolved as commit?’
```

If you try to synchronize a distant branch and get this error, try

```
git pull
```

first, to retrieve the distant branch before trying to associate it with a local branch.

6.2 git demande le mot de passe à chaque fois

Il est possible de rentrer les mots de passe dans un fichier .netrc dans le répertoire utilisateur. Le fichier ressemble à ceci :

```
machine code.google.com login usernamea@gmail.com password blablabla
```

Mais si lors de la création du dépot, l'url utilisée contenait l'identifiant, git demandera le mot de passe à chaque fois. Afin de résoudre le problème, il faut se placer dans le répertoire du dépot. Puis aller dans le sous-dossier **.git** et modifier dans le fichier **config** pour enlever la partie **login@** qu'il pourrait y avoir.

Index