Git
. . . de survie

F. Langrognet

PLAN

PLAN

PLAN

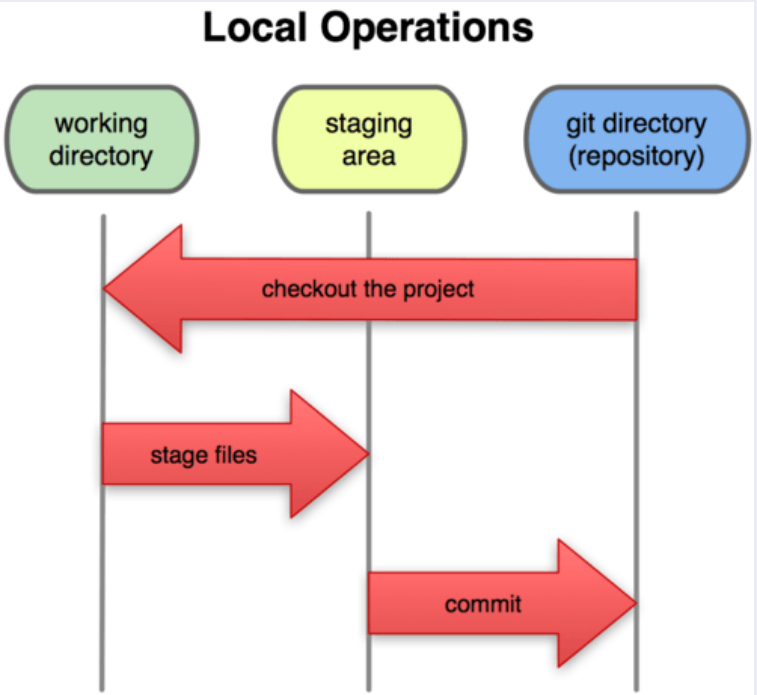
Git : les fichiers dans tous leurs états (3)

Modifié, indexé, validé

- Modifié : fichier modifié mais ni indexé ni validé
- Indexé : fichier marqué modifié pour qu'il fasse partie de la prochaine version (validée) du projet
- Validé : fichier stocké en sécurité dans la base de données locale

3 sections principales d'un projet Git

- le répertoire de travail
- la zone d'index
- le répertoire Git (dépôt)



Modifié n'est pas validé

Utilisation standard de Git

Avec Git, le fait de modifier un fichier ne suffit pas pour qu'il puisse passer au stade validé : il doit dabord être indexé

Utilisation standard de Git

- Modification des fichiers dans le répertoire de travail
- Indexation des fichiers modifiés : ils passent dans la zone d'index
- Validation des fichiers : ils passent de l'index dans la base de données du répertoire Git

PLAN

Démarrer avec Git

- Etat du répertoire de travail

```
date.cpp  date.h  personne.cpp  personne.h
```

- Initialisation d'un dépôt Git dans ce répertoire

```
$ git init .
 Initialized empty Git repository in ...
```

→ Création d'un nouveau sous-répertoire nommé .git ... vide

- Indexation des fichiers *.cpp

```
$ git add *.cpp
```

- Validation/commit

```
git commit -m "ajout des fichiers cpp dans le
   projet git - version initiale"
```

→ A ce stade, seuls les fichiers cpp sont dans le projet git

Modification, indexation, validation

- Modification du fichier date.cpp

```
$ echo "//ajout d'une ligne de commentaire"
  >> date.cpp
```

- Indexation du fichier date.cpp

```
$ git add date.cpp
```

- Validation du fichier date.cpp

```
$ git commit -m "ajout d'un commentaire dans
    le fichier date.cpp"
1 file changed, 1 insertion(+)
```

Indexation et validation

Pas indexé, un fichier n'est pas validé

- Modification du fichier personne.cpp

```
$ echo "//ajout d'un commentaire" >> personne.cpp
```

- Validation du fichier date.cpp

```
$ git commit -m "ajout d'un commentaire dans
   le fichier personne.cpp"
  no changes added to commit ...
```

La nouvelle version du fichier personne.cpp n'est pas dans le dépôt local

Ajout dans l'index, validation

```
$ git add personne.cpp

$ git commit -m "commentaire dans personne.cpp"
   1 file changed, 1 insertion(+)
```

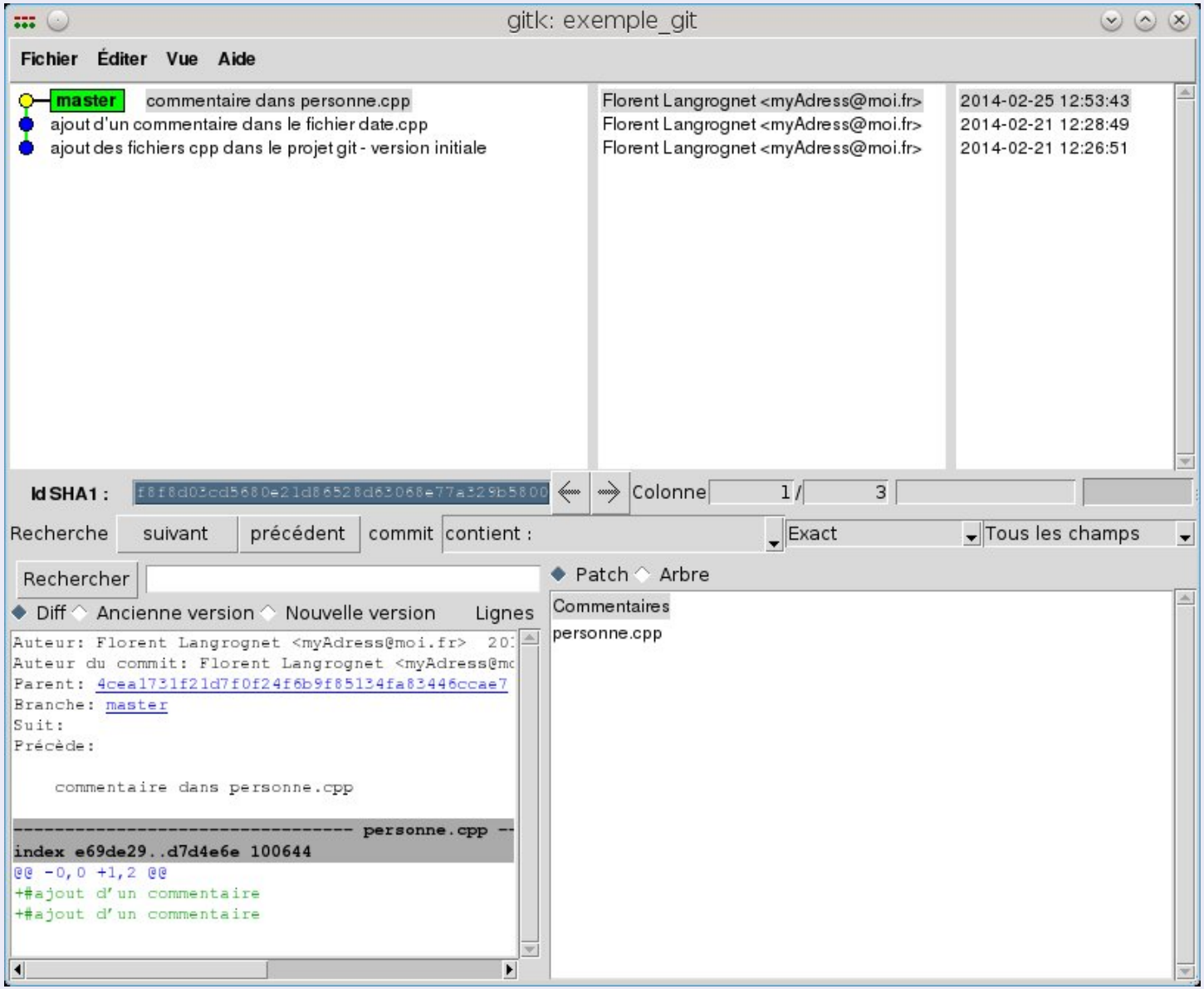
Historique

Accès à l'historique des validations

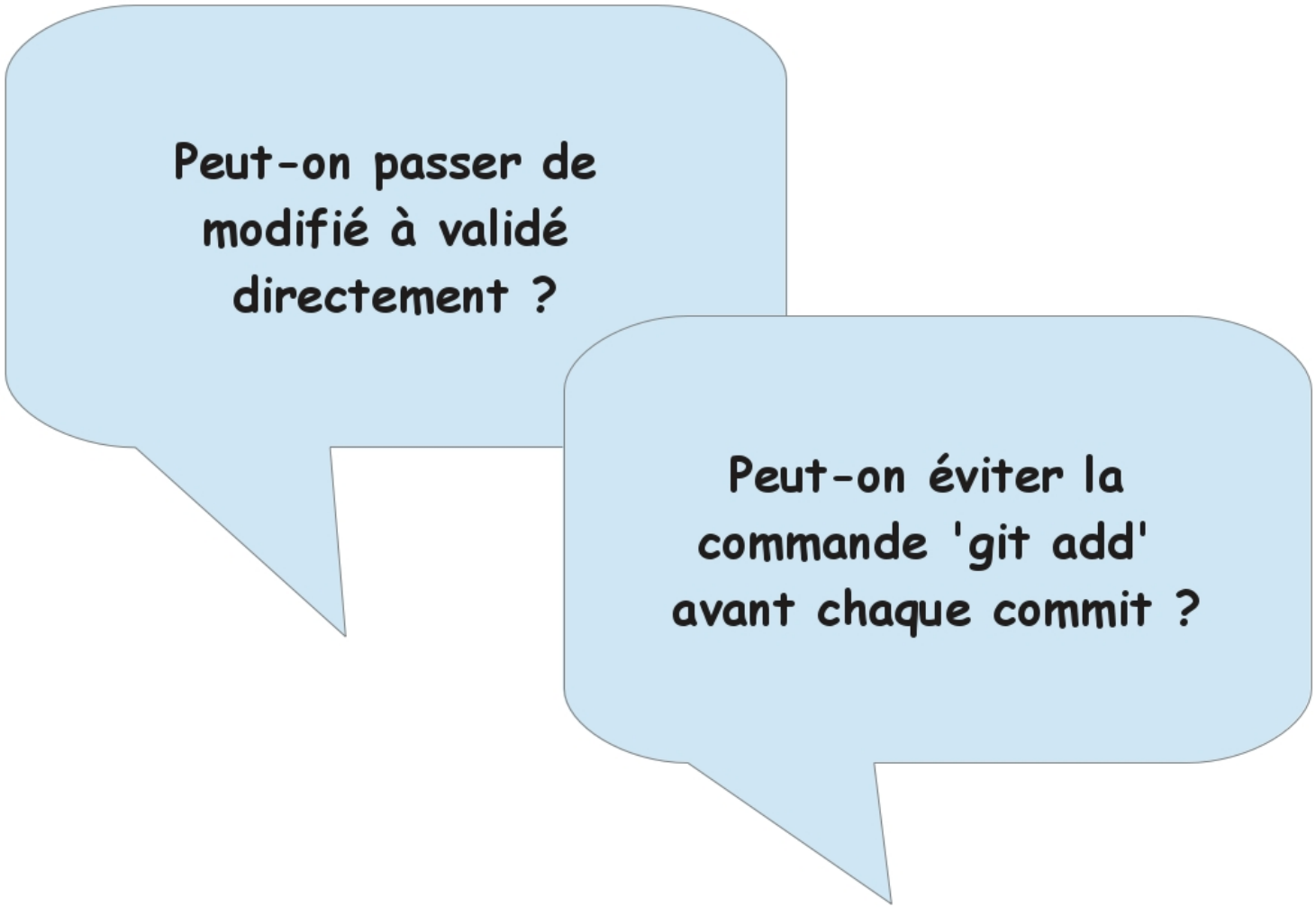
```
$ git log

   commit f8f8d03cd5680e21d86528d63068e77a329b5800
   Author: Florent Langrognet <myAdress@moi.fr>
   Date:   Tue Feb 25 12:53:43 2014 +0100
   commentaire dans personne.cpp

   commit 4cea1731f21d7f0f24f6b9f85134fa83446ccae7
   Author: Florent Langrognet <myAdress@moi.fr>
   Date:   Fri Feb 21 12:28:49 2014 +0100
   ajout d'un commentaire dans le fichier date.cpp

   commit 0462d45fc429a9c14ab85e723a1752814cc6e448
   Author: Florent Langrognet <myAdress@moi.fr>
   Date:   Fri Feb 21 12:26:51 2014 +0100
   ajout des fichiers cpp dans le projet git
     - version initiale
```

Visualisation

```
$ gitk
```



Des questions ?



Options de git add

- *git add -u*
  Permet d'ajouter dans la zone d'index tous les fichiers modifiés ou supprimés qui sont déjà gérés par git (/dans le dépôt local) du répertoire courant et des sous-répertoires.
  On doit avoir déjà fait un git add pour ces fichiers.

- *git add -A*
  Permet d'ajouter dans la zone d'index tous les fichiers du répertoire courant et des sous-répertoires.
  Y compris les fichiers qui n'avaient jamais été indexés.

Options de git commit

- *git commit -a*
  Raccouci pour git add -u et git commit
  Permet de d'ajouter dans la zone d'index tous les fichiers modifiés ou supprimés qui sont déjà dans le dépôt local (/gérés par git) du répertoire courant et des sous-répertoires et de les valider/propager.

PLAN

Modifier le dernier commit

Modifier le commentaire du dernier commit

```
$ git commit --amend -m "modification du commentaire"
```

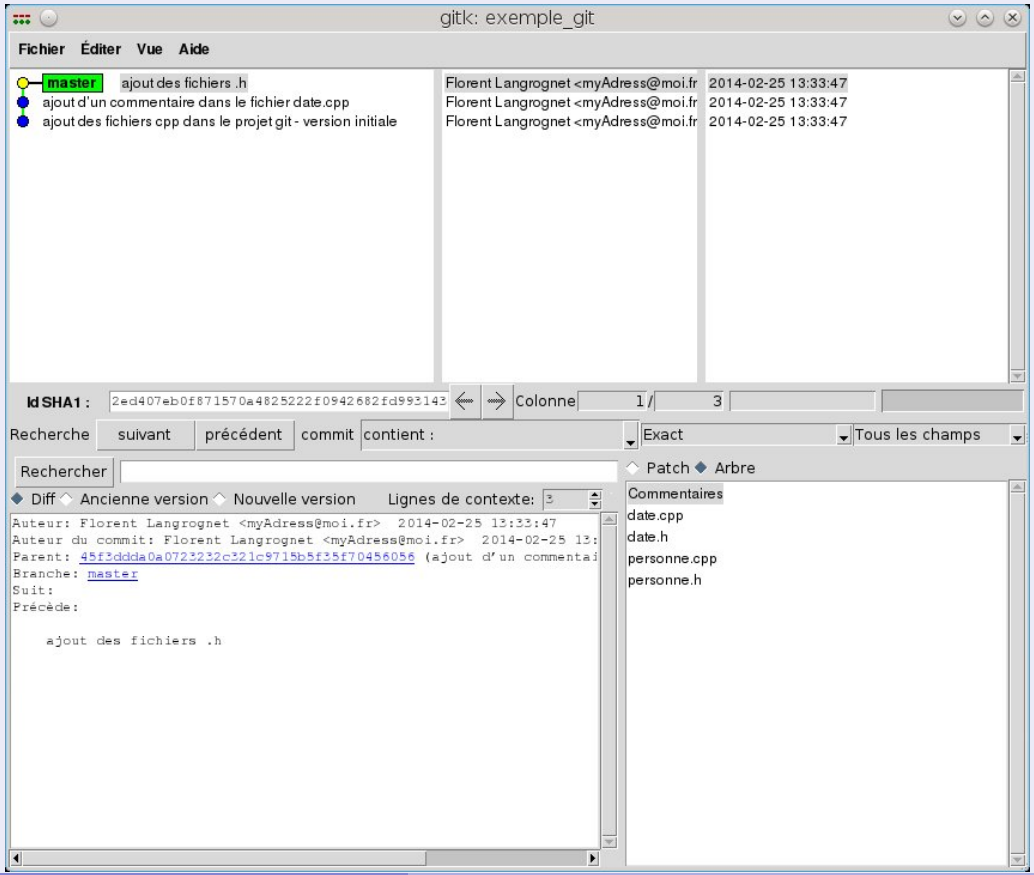
```
$ git log

  commit d291717e96ec3009fd21289766f036cd06bd63ca
  Author: Florent Langrognet <myAdress@moi.fr>
  Date:   Tue Feb 25 12:53:43 2014 +0100
  modification du commentaire

  commit 4cea1731f21d7f0f24f6b9f85134fa83446ccae7
  Author: Florent Langrognet <myAdress@moi.fr>
  Date:   Fri Feb 21 12:28:49 2014 +0100
  ajout d'un commentaire dans le fichier date.cpp

  ...
```

Le commentaire du dernier commit a été modifié

Ajout de fichiers qui n'avaient pas été indexés

```
$ git add *.h

$ git commit --amend -m "ajout des fichiers .h"

$ gitk
```

Le dernier commit a été modifié



Désindexer, modifier l'index

Modifier l'index

Désindexer un fichier (1)

- Modification de 2 fichiers

```
$ echo "//ajout d'un 2e commentaire" >> date.cpp

$ echo "//ajout d'un 2e commentaire" >> personne.cpp

$ git add date.cpp personne.cpp

$ git status

 # On branch master
 # Changes to be committed:
 #   (use "git reset HEAD <file>..." to unstage)
 #
 #       modified:   date.cpp
 #       modified:   personne.cpp
 #
```

Modifier l'index

Désindexer un fichier (2)

- Désindexer le fichier personne.cpp

```
$ git reset HEAD personne.cpp
 Unstaged changes after reset:
 M       personne.cpp

$ git status
 # On branch master
 # Changes to be committed:
 #   (use "git reset HEAD <file>..." to unstage)
 #       modified:   date.cpp
 #
 # Changes not staged for commit:
 #   (use "git add <file>..." to update what will
 #      be committed)
 #   (use "git checkout -- <file>..." to discard
 #      changes in working directory)
 #       modified:   personne.cpp
```

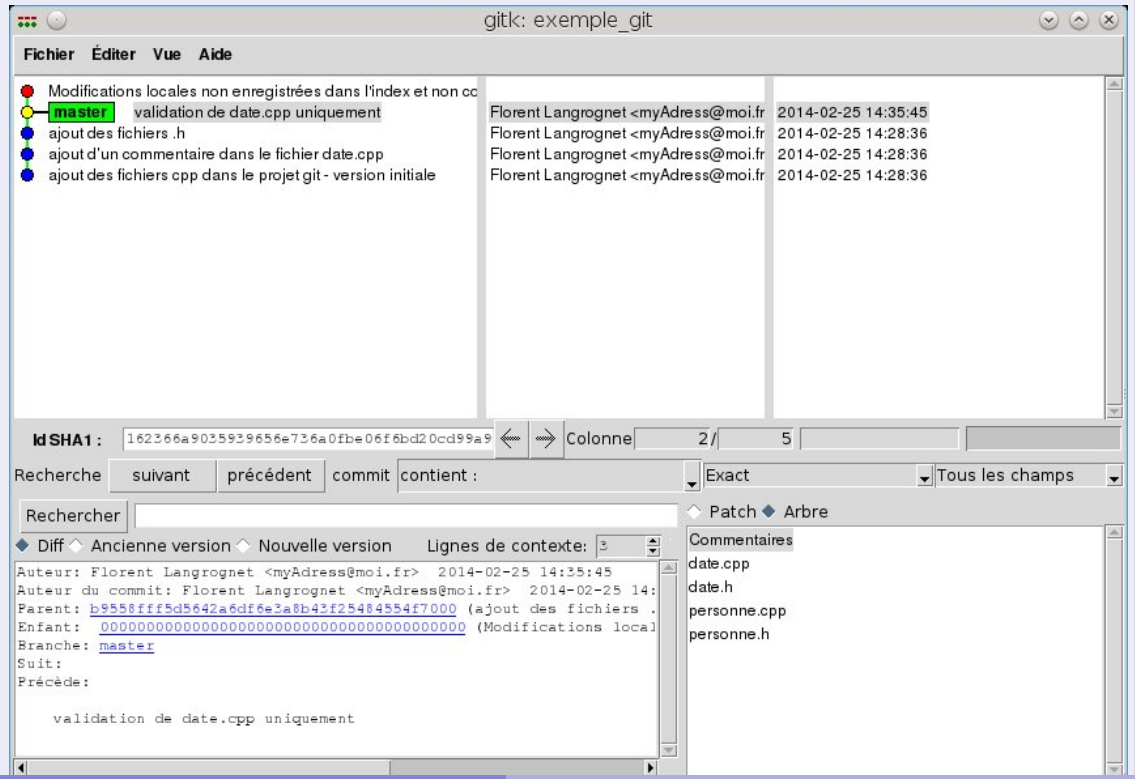
Modifier l'index

Désindexer un fichier (3)

- Valider les modifications pour date.cpp

```
$ git commit -m "validation de date.cpp uniquement"

[master 162366a] validation de date.cpp uniquement
 1 file changed, 1 insertion(+)
```

- Seul date.cpp a été validé
- Modifications locales non indexées : personne.cpp



Réinitialiser un fichier modifié

Réinitialiser un fichier modifié (1)

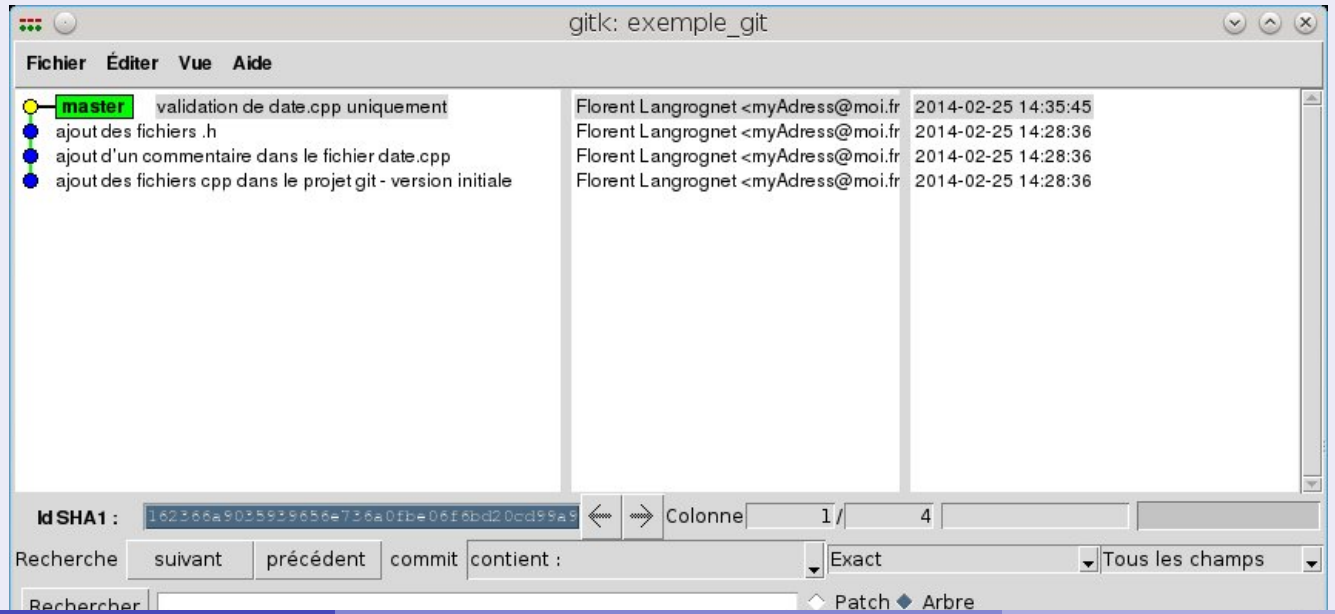
- Le fichier personne.cpp est modifié et non indexé

```
$ git status

 # On branch master
 # Changes not staged for commit:
 #   (use "git add <file>..." to update what will
 #     be committed)
 #   (use "git checkout -- <file>..." to discard
 #      changes in working directory)
 #
 #       modified:   personne.cpp
 #
 no changes added to commit
   (use "git add" and/or "git commit -a")
```

Réinitialiser un fichier modifié (2)

- Réinitialisation d'un fichier

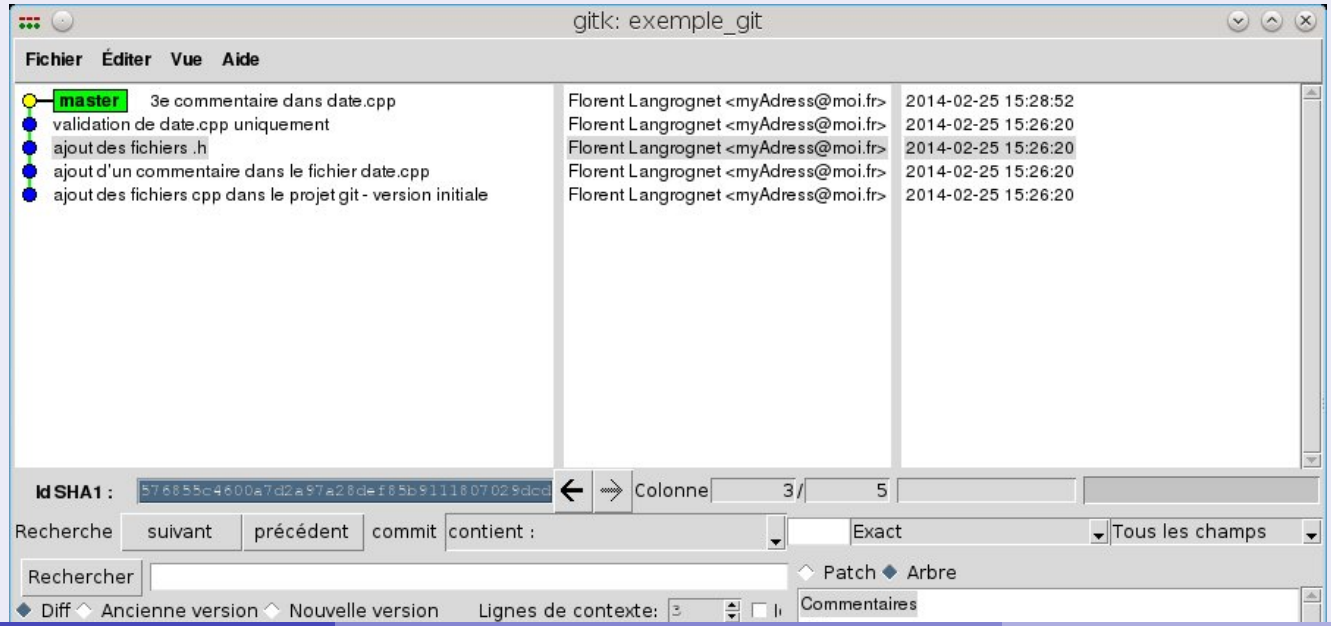
```
$ git checkout -- personne.cpp

$ git status

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



Revenir en arrière

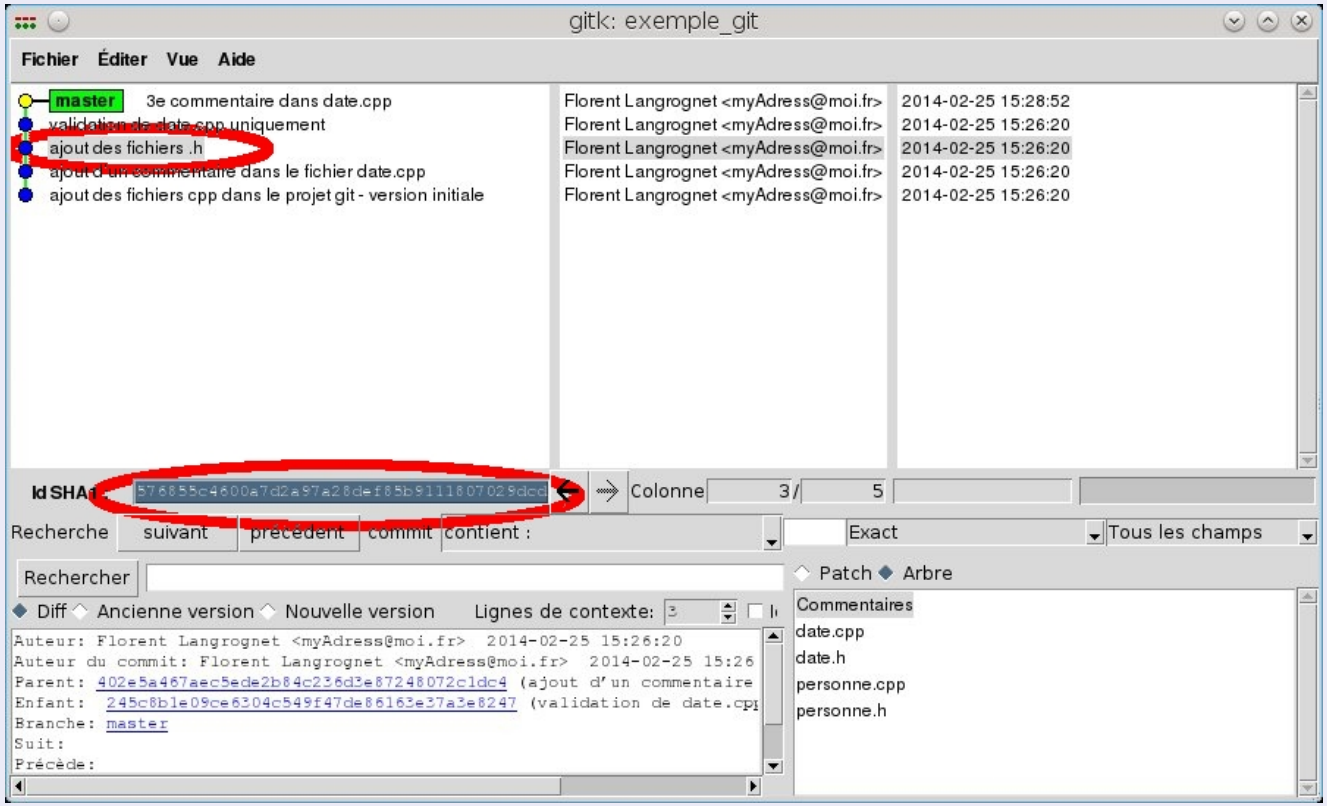
On poursuit le travail...

```
$ echo "//ajout d'un 3e commentaire" >> date.cpp

$ git add date.cpp

$ git commit -m "3e commentaire dans date.cpp"

 [master 7778ccc] 3e commentaire dans date.cpp
 1 file changed, 1 insertion(+)
```



Revenir à une version précédente (1)

On souhaite revenir à cette version

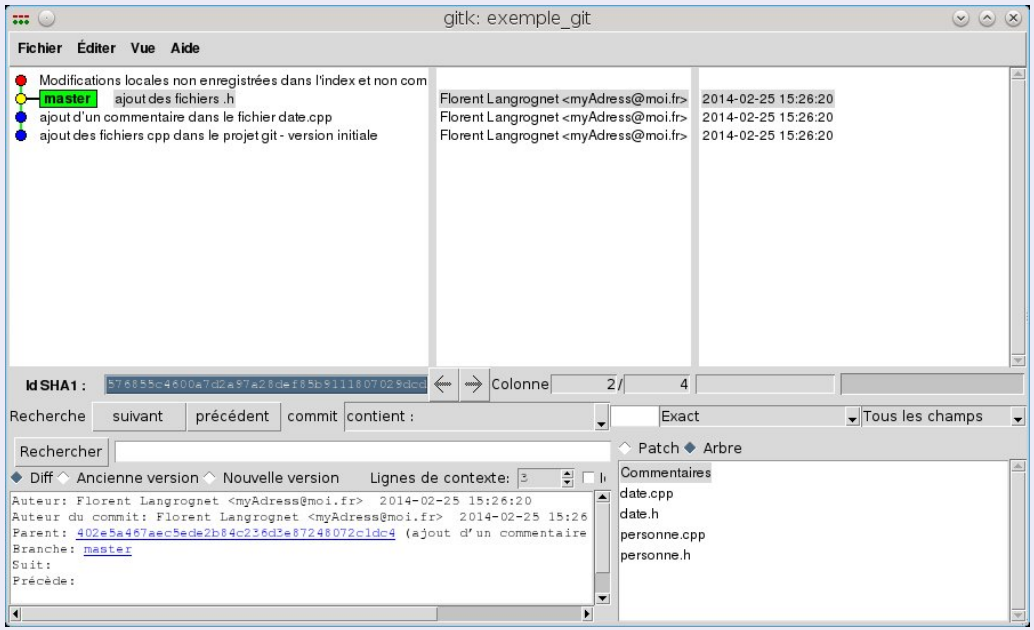


Revenir à une version précédente (2)

- Déplacer le HEAD

```
$ git reset 576855c4600a7d2a97a28def85b9111807029dcd

 Unstaged changes after reset:
 M       date.cpp
```

- Le HEAD a été modifié
- Aucun fichier n'a été modidié date.cpp conserve ses modifications locales (non indexées)



Revenir à une version précédente (3)

- Annuler les modifications locales

```
$ git checkout -- date.cpp

$ git status

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

PLAN

git et la gestion des branches

Les branches

Une branche est une version du projet qui

- suit son propre développement (historique via SGV)
- sera (ou non) fusionnée avec d'autres branches

Exemples :

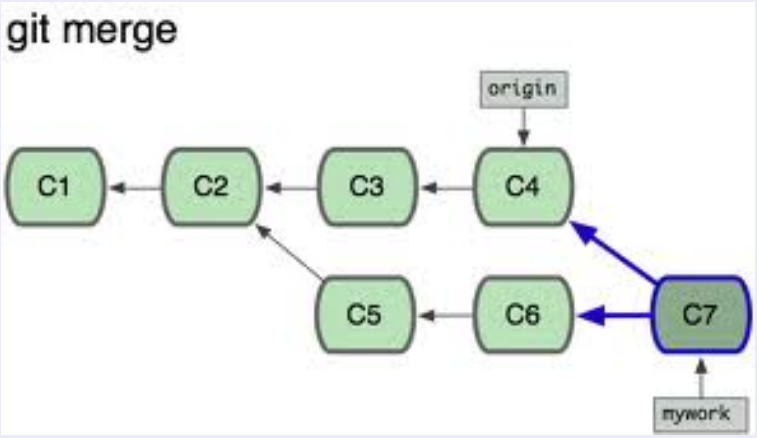
- Version testing menée en parallèle de la version officielle. Elle ne sera pas forcément conservée.
- Version "correction de bug" qui est destinée à être intégrée à la version officielle (qui aura poursuivi son évolution)

Les branches avec git

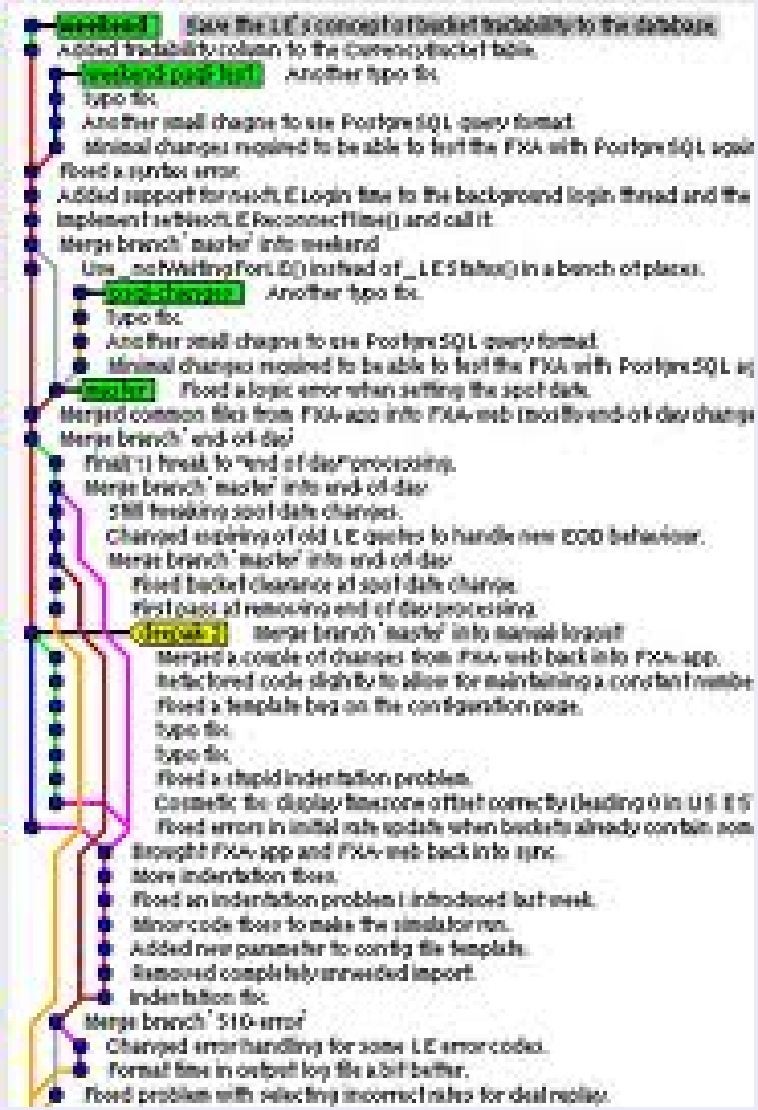
- La gestion des branches est souvent délicate avec les autres SGV (en particulier avec SVN)
- git permet de gérer facilement et efficacement les branches

Gestion des branches avec git

Un cas d'école



La réalité



PLAN

Création de branches

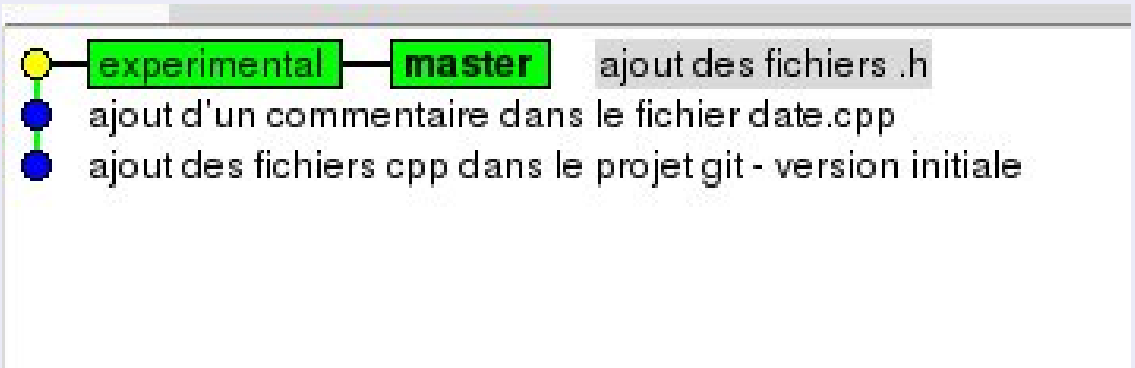
- Création d'une nouvelle branche (experimental)

```
$ git branch experimental
```

- Liste des branches

```
$ git branch

   experimental
 * master
```

  - 2 branches (master et experimental)
  - Branche courante : master



Changer de branche

- On passe sur la branche experimental

```
$ git checkout experimental

  Switched to branch 'experimental'
```

- Liste des branches

```
$ git branch

 *  experimental
    master
```

  - 2 branches (master et experimental)
  - Branche courante : experimental

On travaille dans la branche experimental (1)

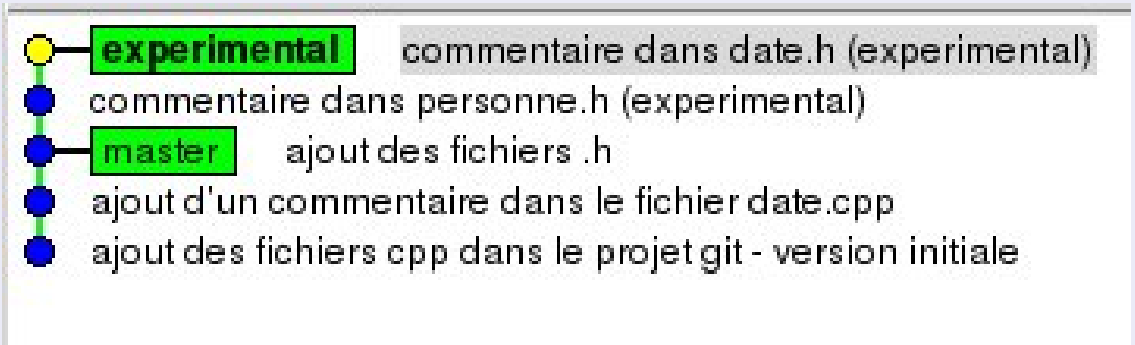
- 1er commit

```
$ echo "//ajout d'un commentaire" >> personne.h

$ git add personne.h

$ git commit -m "commentaire dans personne.h
   (experimental)"

  [experimental ba5a787] commentaire dans
    personne.h (experimental)
  1 file changed, 1 insertion(+)
```

On travaille dans la branche experimental (2)

- 2e commit

```
$ echo "//ajout d'un commentaire" >> date.h

$ git add date.h

$ git commit -m "commentaire dans date.h
   (experimental)"

  [experimental c74f3a6] commentaire dans date.h
    (experimental)
  1 file changed, 1 insertion(+)
```



On repasse dans branche master

- On change de branche

```
$ git checkout master

  Switched to branch 'master'

$ git branch

    experimental
 *  master
```

On travaille dans la branche master (1)

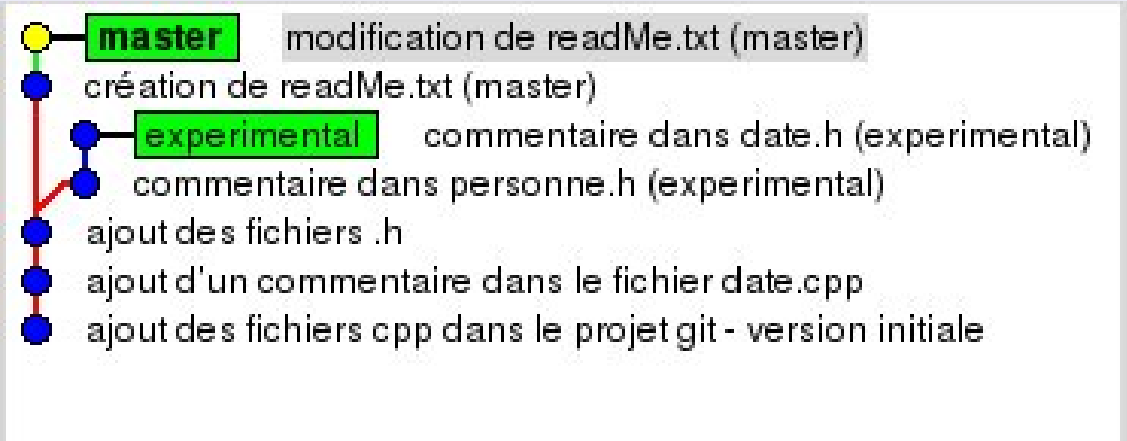
- Création d'un nouveau fichier

```
$ echo "Fichier readMe.txt" > readMe.txt

$ git add readMe.txt

$ git commit -m "création de readMe.txt (master)"

 [master 3372440] création de readMe.txt (master)
 1 file changed, 1 insertion(+)
 create mode 100644 readMe.txt
```

On travaille dans la branche master (2)

- Modification de ce fichier

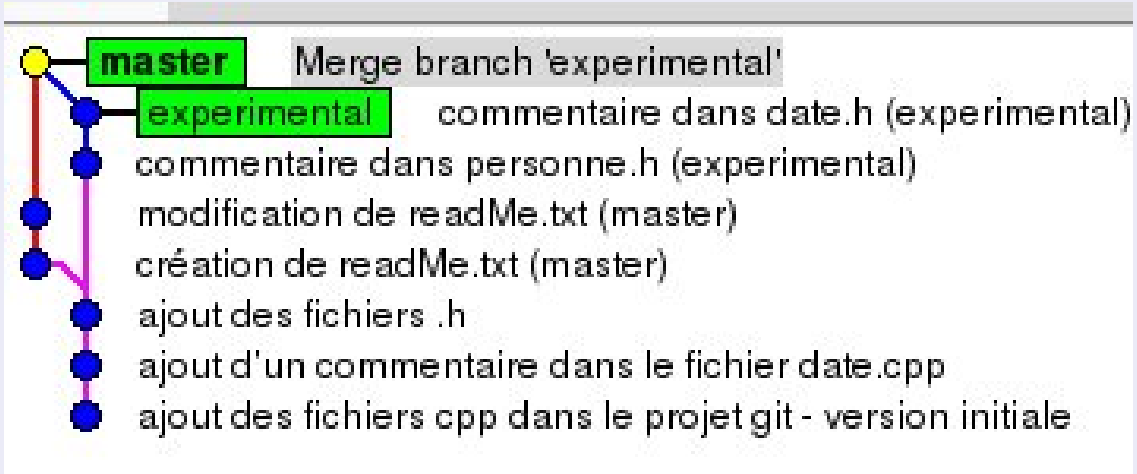
```
$ echo "nouvelle ligne`` >> readMe.txt

$ git add readMe.txt

$ git commit -m "modification de readMe.txt (master)"

  [master fab5767] modification de readMe.txt (master)
  1 file changed, 1 insertion(+)
```



On merge la branche experimental vers la branche master

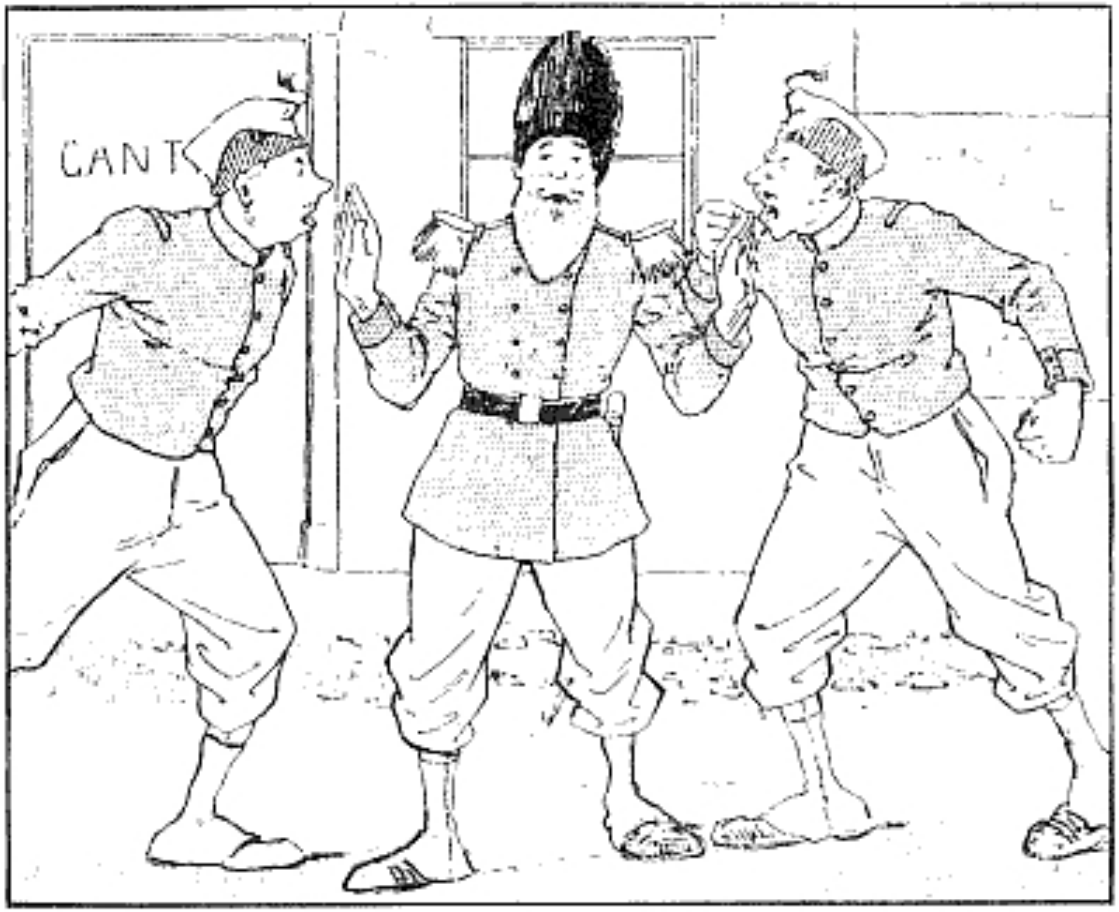
- Merge

```
$ git merge experimental

 Merge made by the 'recursive' strategy.
 date.h     |    1 +
 personne.h |    1 +
 2 files changed, 2 insertions(+)
```



PLAN

Gestion des conflits



On travaille dans la branche experimental

- Modification du fichier readMe.txt

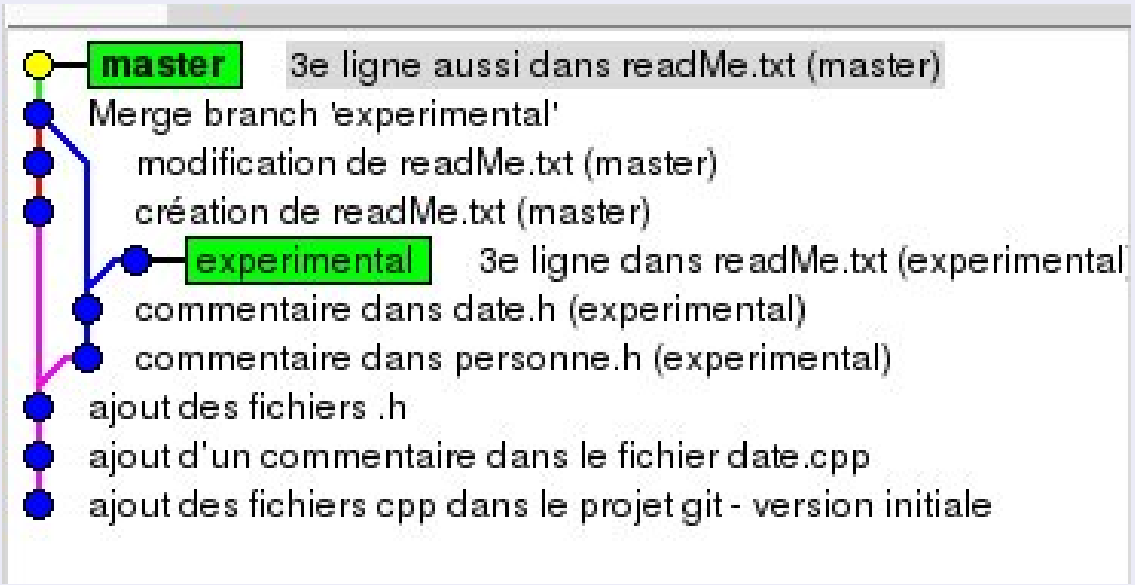
```
$ git checkout experimental
 Switched to branch 'experimental'

$ echo "3e ligne dans readMe.txt" > readMe.txt

$ git add readMe.txt

$ git commit -m "3e ligne dans readMe.txt
  (experimental)"

 [experimental 36786fa] 3e ligne dans readMe.txt
   (experimental)
 1 file changed, 1 insertion(+)
 create mode 100644 readMe.txt
```

On travaille dans la branche master

- Modification du fichier readMe.txt (le même)

```
$ git checkout master
 Switched to branch 'master'

$ echo "3e ligne aussi dans readMe.txt" > readMe.txt

$ git add readMe.txt

$ git commit -m "3e ligne aussi dans readMe.txt
    (master)"
 [master 22828c0] 3e ligne aussi dans readMe.txt
    (master)
 1 file changed, 1 insertion(+), 2 deletions(-)
```

Et si on ”mergeait” ?



- Tentative de merge

```
$ git merge experimental
 Auto-merging readMe.txt
 CONFLICT (add/add): Merge conflict in readMe.txt
 Automatic merge failed; fix conflicts and then
  commit the result.
```

Le merge automatique n’est pas possible
Il faut réoudre les conflits

Où est le conflit ?

- Status

```
$ git status

 # On branch master
 # Unmerged paths:
 #   (use "git add/rm <file>..." as appropriate
 #  to mark resolution)
 #
 #       both added:          readMe.txt
 #
 no changes added to commit (use "git add"
   and/or "git commit -a")
```

Le conflit porte bien sur le fichier readMe.txt

Gérer le conflit

Etat du fichier

```
$ more readMe.txt

<<<<<<< HEAD
3e ligne aussi dans readMe.txt
=======
3e ligne dans readMe.txt
>>>>>>> experimental
```

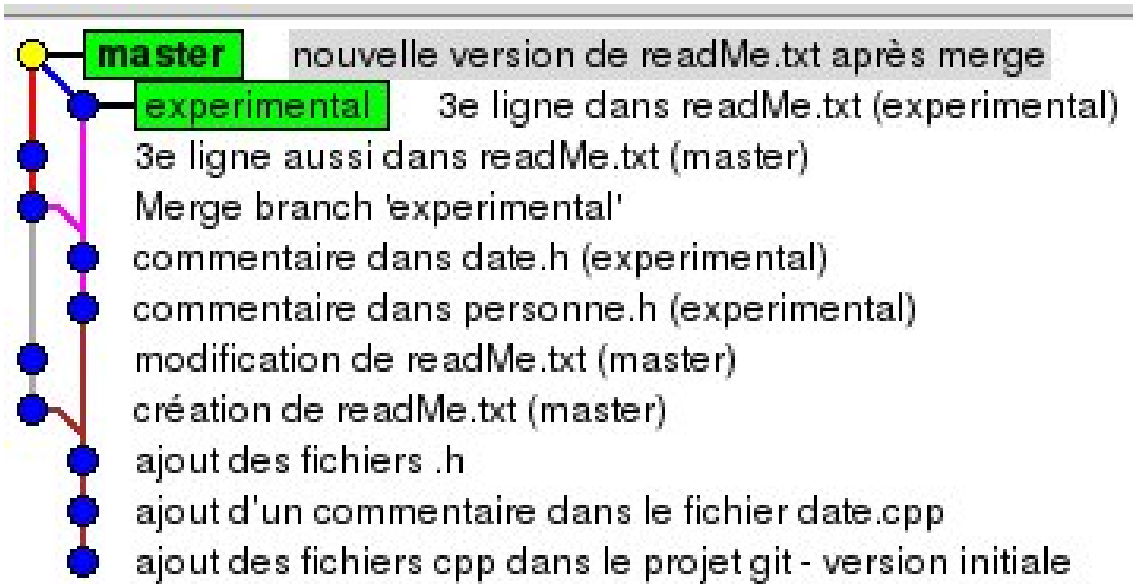
Résoudre le conflit

- Outil graphique : git mergetool
- A la main : choisir une version

```
$ more readMe.txt

 J'ai choisi de remplacer la ligne par une nouvelle
```

Puis valider

```
$ git add readMe.txt

$ git commit -m "nouvelle version de readMe.txt
  après merge"

 [master 95feb13] nouvelle version de readMe.txt
  après merge
```

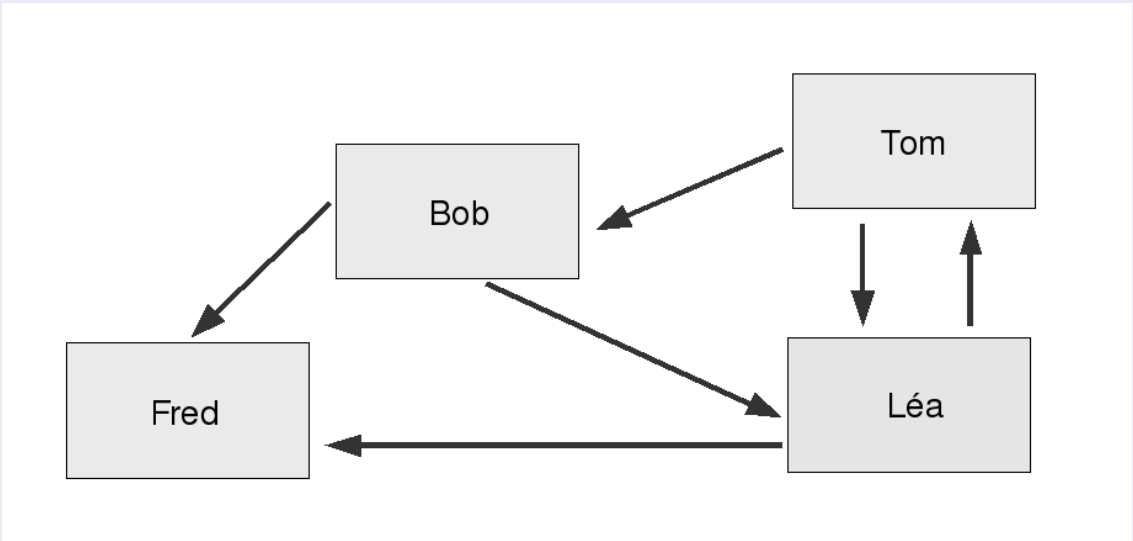


PLAN

Péambule

Droits lecture/écriture

- Travailler avec d'autres dépôts, c'est échanger avec eux
  - Attention à la gestion des droits en lecture/écriture
  - On peut autoriser simplement la lecture, ou l'écriture, ou les 2



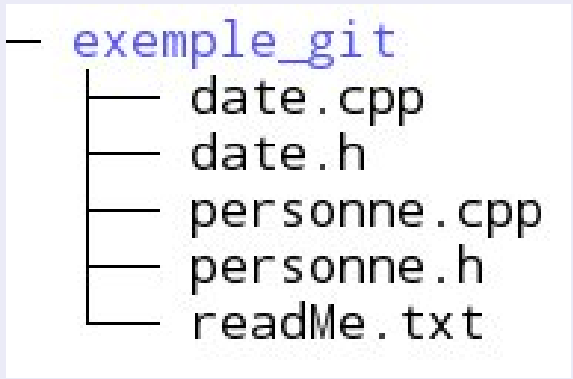
PLAN

Cas pratique

Pour plus de simplicité, on se placera sur le même PC

- Dépôt existant : le répertoire ”exemple_git” (utilisé précédemment)
- Nouveau dépôt : un répertoire (”exemple2_git”) au même niveau de l’arborescence

```
$ tree
```



Dépôts distants

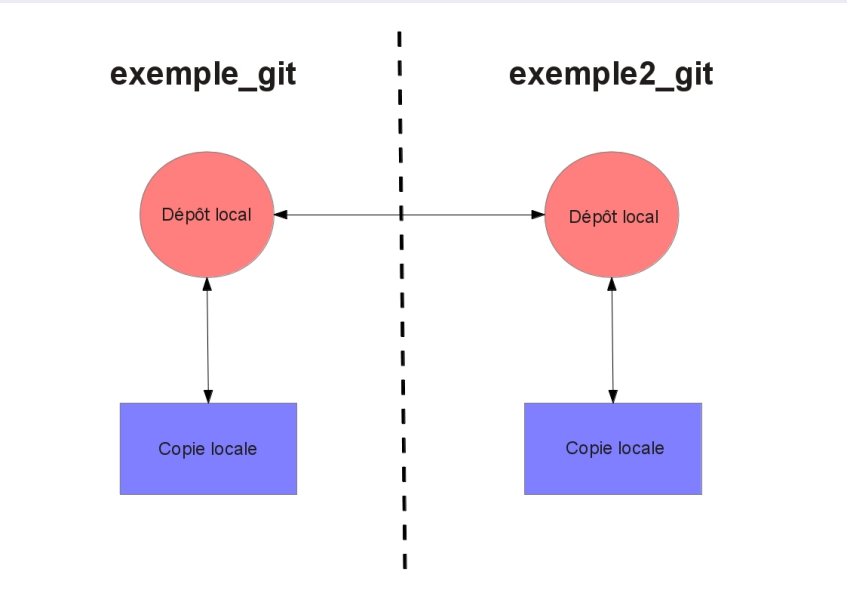
Travailler avec des dépôts distants ne présente pas de difficulté supplémentaire sauf éventuellement les accès en lecture/écriture.

On clone le dépôt

- clone

```
$ git clone exemple_git/ exemple2_git

 Cloning into 'exemple2_git'...
 done.
```

```
exemple2_git
    date.cpp
    date.h
    personne.cpp
    personne.h
    readMe.txt
exemple_git
    date.cpp
    date.h
    personne.cpp
    personne.h
    readMe.txt
```



Informations sur les autres dépôts

Dans le répertoire exemple2_git :

- git remote : donne la liste des autres dépôts

```
$ git remote
 origin
```

Ici il n'y a que le dépôt ayant servi pour le clone (origin)

- git remote -v : des détails

```
$ git remote -v
origin  /home/.../exemple_git/ (fetch)
origin  /home/.../exemple_git/ (push)
```

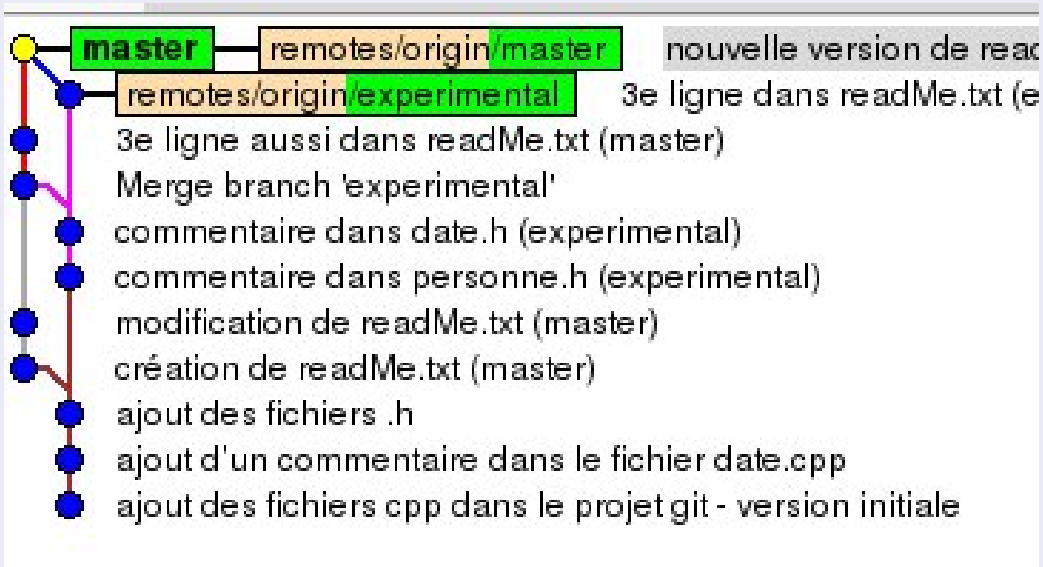
- git remote add : alias

```
$ git remote add LeDepotDeFlorent origin
$ git remote -v
 LeDepotDeFlorent        origin (fetch)
 LeDepotDeFlorent        origin (push)
 origin  /home/.../exemple_git/ (fetch)
```

Branches et origin

Vu de exemple2_git

- Les branches (y compris le master) du dépôt d'origine sont gérées comme des branches distantes (branches d'un autre dépôt que l'on ne peut pas modifier).



PLAN

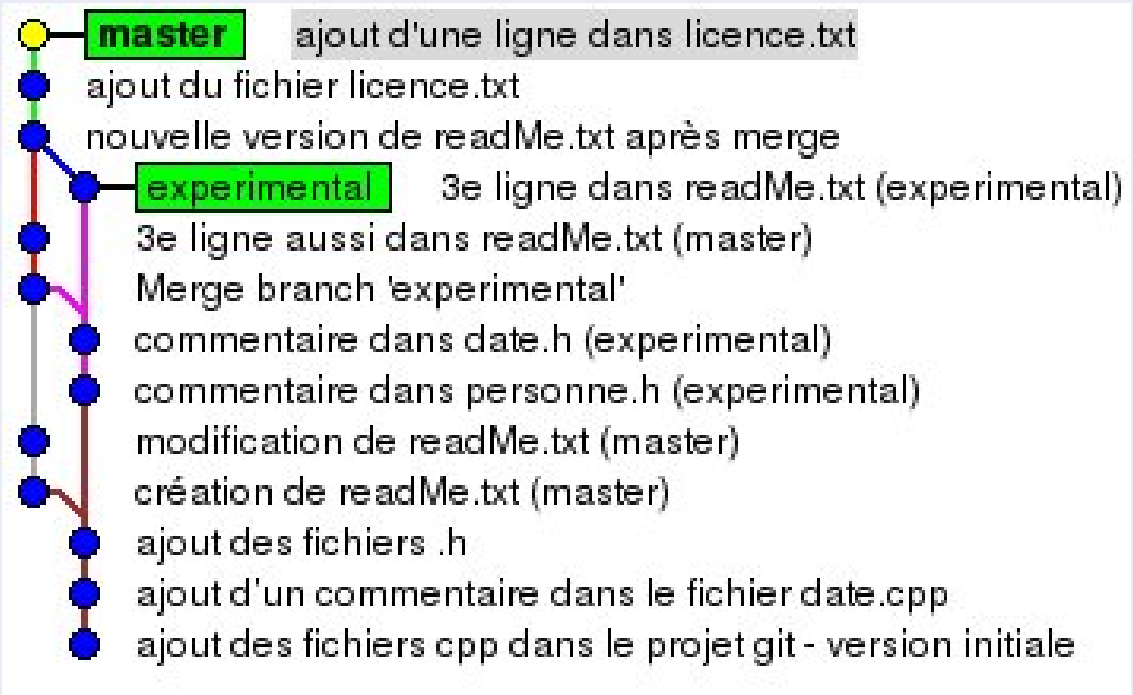
On poursuit le travail dans exemple_git (1)

- On poursuit le travail dans exemple_git

```
$ echo "Ma licence" >> licence.txt

$ git add licence.txt

$ git commit -m "ajout du fichier licence.txt"
  [master 4464ee7] ajout du fichier licence.txt
  1 file changed, 1 insertion(+)
  create mode 100644 licence.txt

$ echo "encore une ligne" >> licence.txt

$ git add licence.txt

$ git commit -m "ajout d'une ligne dans licence.txt"
  [master 4442640] ajout d'une ligne dans licence.txt
  1 file changed, 1 insertion(+)
```

On poursuit le travail dans exemple_git (2)

gitk



Note : les rôles ne sont pas symétriques entre les 2 dépôts : exemple_git ne connaît pas exemple2_git

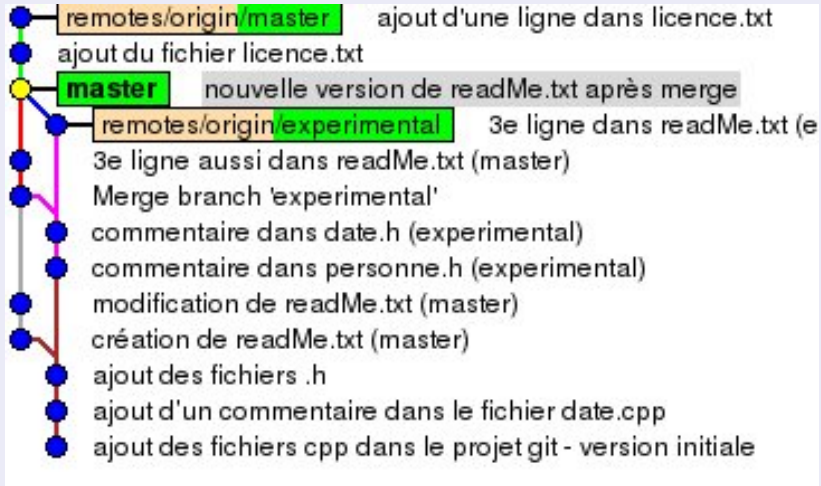
On tire...

...dans exemple2_git

- git fetch

```
$ git fetch origin

 remote: Counting objects: 7, done.
 remote: Compressing objects: 100% (4/4), done.
 remote: Total 6 (delta 2), reused 0 (delta 0)
 Unpacking objects: 100% (6/6), done.
 From /home/.../exemple_git
   95feb13..4442640  master     -> origin/master
```

Mise à jour des branches distantes (celle de origin) sans merge.



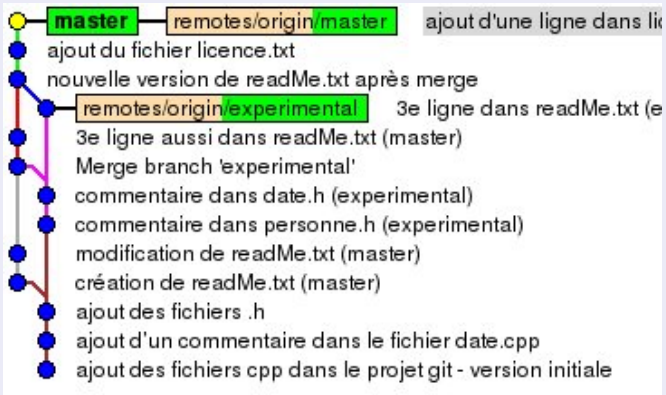
On merge

...dans exemple2_git

Il reste donc un merge à faire...

- git merge

```
$ git merge origin/master

 Updating 95feb13..4442640
 Fast-forward
 licence.txt |    2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 licence.txt
```



2 étapes en une

pull = fetch + merge

Les lignes

```
$ git fetch origin

$ git merge origin/master
```

Peuvent être remplacées par

```
$ git pull origin
```

→

Les modifications effectuées sur le dépôt origin sont intégrées :

- sur le dépôt local
- dans la copie de travail

PLAN

On poursuit le travail dans exemple2_git

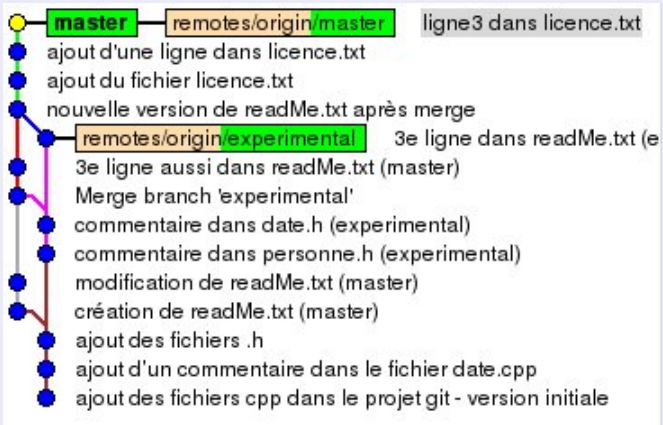
- On poursuit le travail dans exemple2_git

```
$ echo "ligne3" >> licence.txt

$ git add licence.txt

$ git commit -m "ligne3 dans licence.txt"
 [master f0c761d] ligne3 dans licence.txt
 1 file changed, 1 insertion(+)
```

On pousse

- git push

```
$ git push origin master
 Counting objects: 5, done.
 Delta compression using up to 4 threads.
 Compressing objects: 100% (2/2), done.
 Writing objects: 100% (3/3), 299 bytes, done.
 Total 3 (delta 1), reused 0 (delta 0)
 Unpacking objects: 100% (3/3), done.
 To /home/.../exemple_git/
   4442640..f0c761d  master -> master
```

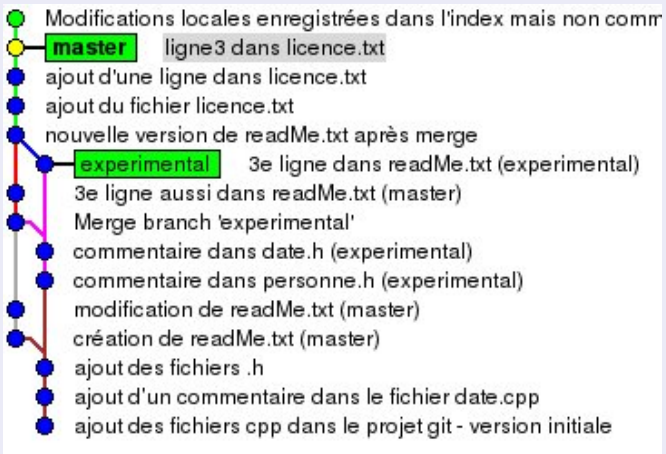


Que s'est-il passé sur origin (exemple_git) ?

- Statut ?

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   licence.txt
#
```

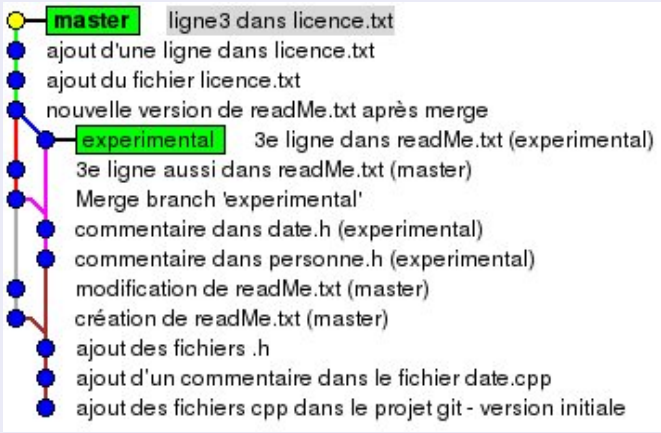
Le fichier licence.txt a bien été modifié sur le dépôt local
Il apparaît donc comme modifié dans la copie de travail



On met à jour

- On récupère la version du dépôt local (effacera les modifications locales)

```
$ git checkout licence.txt
```



Push est une opération entre dépôts.
Les copies locales ne sont pas touchées lors d'un push.

PLAN

PLAN

Etiquetage

Étiquette / Tag : nom d'un état de l'historique (un *pointeur vers un commit).*

- *Connaître la liste des tags*

```
$ git tag
 V1.0
 V1.1
```

- *Création d'une étiquette*

```
 $ git tag V2.0
 $ git tag
  V1.0
  V1.1
  V2.0
```

- *Information sur une étiquette*

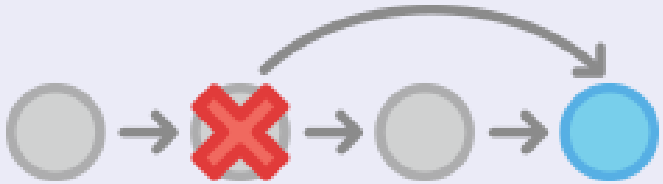
```
 $ git show V2.0
commit 4ac60d5897507ae24e55f99a5dbce6c16ace04f8
Author: Florent Langrognet <myAdress@moi.fr>
Date:   Fri Feb 28 15:33:03 2014 +0100
```

PLAN

Revert (1)

Annuler l'effet d'un commit

En créant un nouveau commit dont l'action est l'inverse du commit dont on veut annuler l'effet.



- Commande

```
$ git revert <identifiant_commit>
```

- Utilisations

```
$ git revert HEAD^ : l'avant dernière.
$ git revert HEAD~5 : la cinquième moins récente.
$ git revert e6337879 : la soumission e6337879.
```

Revert (2)

- Exemple : nouveau commit qui annule le dernier

```
$ git revert HEAD
[master 32b21e6] Revert "ajout d'une ligne dans
  licence.txt"
 1 file changed, 1 deletion(-)

$ git log
commit 32b21e687285c5d5edb5c73bed49bc847e898f37
...
  This reverts commit 4ac60d5897507ae24e55f99...
```

PLAN

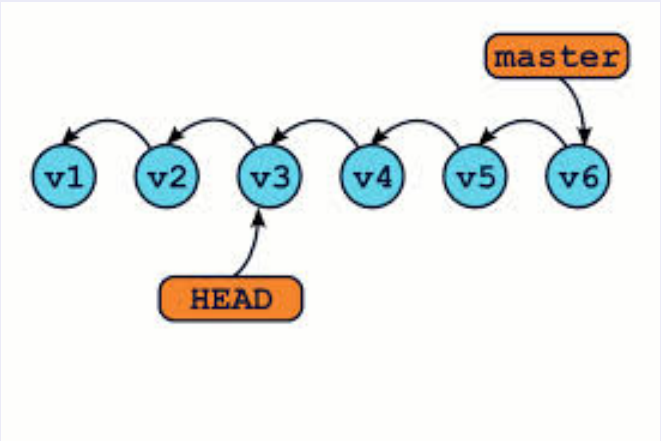
Définitions

HEAD - MASTER

L'historique *git est une succession de commits (qui définissent des états).*

- *HEAD : pointeur vers un commit qui sera le parent du prochain commit*
- *MASTER : pointeur vers le dernier commit de la branche principale*

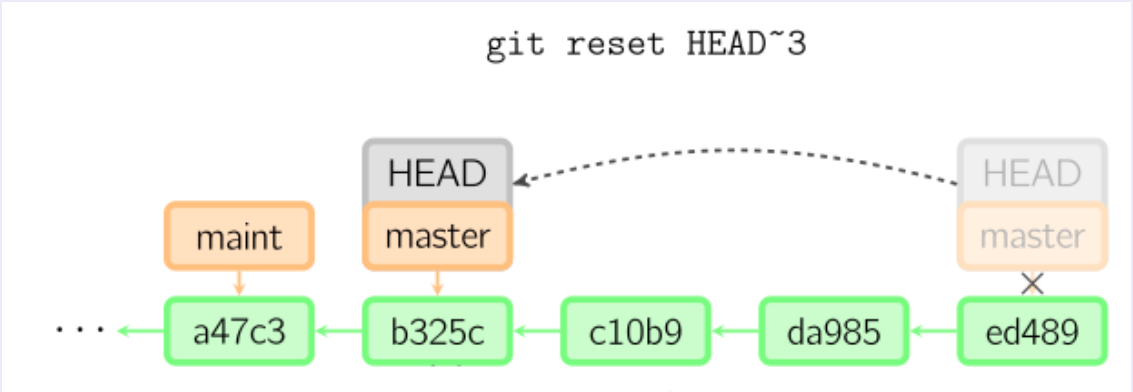
Généralement le HEAD pointe vers le MASTER, mais pas toujours...



Reset

Reset

- Objectif : Revenir en arrière
- Déplace le HEAD et le MASTER (sauf cas particulier)
- Ne modifie pas les fichiers (dont le contenu est celui de l'ancien HEAD)
  - Ils sont marqués modifiés mais non indexés



Reset

Utilisations

- Changements en référence au HEAD : revenir en arrière ... mais sur le même commit

```
$ git reset <fichier>
$ git reset
```

  - Seul l'index sera impacté (vidé)
  - Utilisation peu dangereuse : utilisation fréquente pour désindexer les fichiers modifés

- Changements en référence à un autre commit

```
$ git reset <commit>
```

  - L'index sera impacté
  - le MASTER et le HEAD seront modifiés 
    Tout nouveau commit se fait à la suite du commit ayant servi au reset.
    Les commits entre ce commit et l'ancien HEAD disparaissent

Reset

Exemple

- Quelques commandes

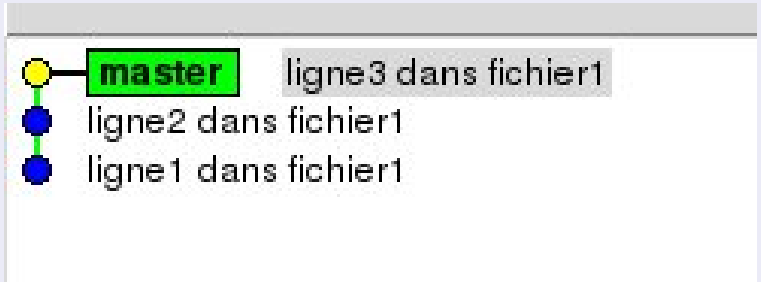
```
$ git init .
$ echo ligne1 > fichier1.txt
$ git add fichier1.txt
$ git commit ``ligne1 dans fichier1.txt''

$ echo ligne2 >> fichier1.txt
$ git add fichier1.txt
$ git commit ``ligne2 dans fichier1.txt''

$ echo ligne3 >> fichier1.txt
$ git add fichier1.txt
$ git commit ``ligne3 dans fichier1.txt''
```

Reset

Exemple

- Historique

```
$ git log
commit f437b7dbf020a856f909ad0ac00488049454dbe4
    ligne3 dans fichier1

commit a33bd3e3c1176f520183cc89cf7e79fe61608771
    ligne2 dans fichier1

commit a56d8aae8e2f6319de53e4bc07fb35e602d382af
    ligne1 dans fichier1
```



Reset

Exemple

- On revient 2 commits en arrière

```
$ git reset a56d8aae8e2f6319de53e4bc07fb35e602d382af

Unstaged changes after reset:
M       fichier1.txt
```

- Le fichier file1.txt n'est pas indexé mais modifié

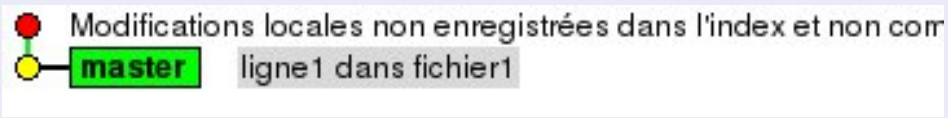
```
$ more fichier1.txt
ligne1.txt
ligne2.txt
ligne3.txt
```

Reset

Exemple

- Historique après le reset

```
$ git log
commit a56d8aae8e2f6319de53e4bc07fb35e602d382af
Author: Florent Langrognet <myAdress@moi.fr>
Date:   Thu Mar 20 10:25:38 2014 +0100

    ligne1 dans fichier1
```



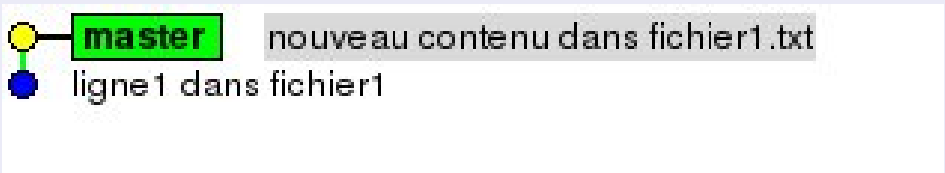
Les 2 autres commits ont disparus

Reset

Exemple

- Un nouveau commit

```
$ echo "nouveau contenu" >fichier1.txt
$ git add fichier1.txt
$ git commit -m "nouveau contenu dans fichier1.txt"

$ git log
commit 2527209aad5585473abfa1f42ba1757099dad1fb
    nouveau contenu dans fichier1.txt

commit a56d8aae8e2f6319de53e4bc07fb35e602d382af
    ligne1 dans fichier1
```



Les 2 autres commits ont disparus

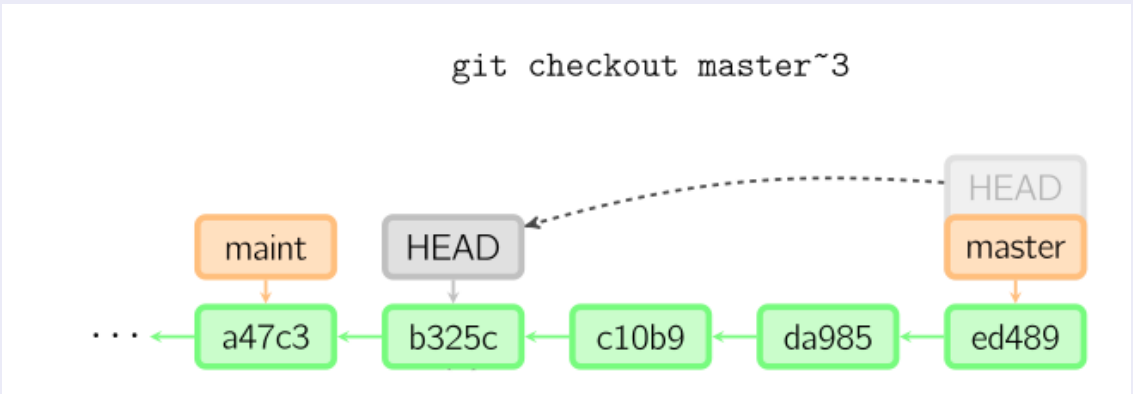
PLAN

Checkout

Ckeckout

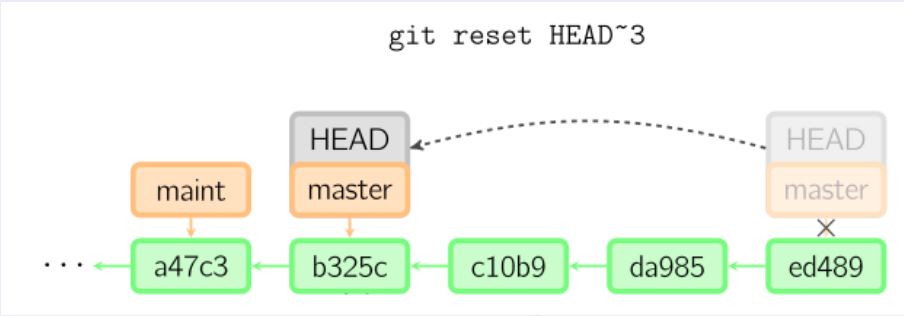
- Objectif : Revenir en arrière
- Déplace le HEAD mais pas le MASTER
- Modifie les fichiers (dont le contenu est celui du commit ayant servi au checkout)
  - Ils sont marqués non modifiés et non indexés

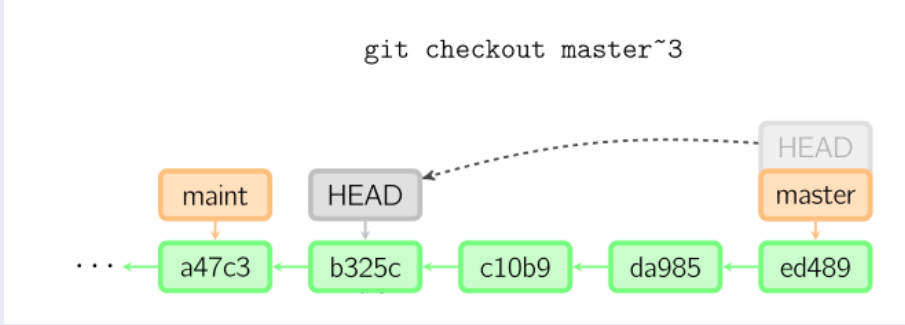
Comme le MASTER n'a pas été modifié, on peut revenir sur le MASTER à tout moment.
C'est un moyen sûr de revenir en arrière pour consulter, travailler à partir d'un certain commit.



Checkout - Reset

Différences entre checkout et reset





La différence essentielle est le déplacement du MASTER dans le cas du reset.

checkout

checkout et HEAD détaché

- Supposons la situation suivante (3 commits)

```
            HEAD (refers to branch 'master')
             |
             v
a---b---c  branch 'master' (refers to commit 'c')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

- Commit dans cet état

```
$ edit; git add; git commit

               HEAD (refers to branch 'master')
                |
                v
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

checkout

checkout et HEAD détaché

- Revenons en arrière de 2 commits avec checkout
  Le HEAD est détaché du MASTER

```
$ git checkout v2.0  # or
$ git checkout master^^

   HEAD (refers to commit 'b')
    |
    v
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

- Commit dans cet état

```
$ edit; git add; git commit

     HEAD (refers to commit 'e')
      |
      v
      e
     /
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

checkout

checkout et HEAD détaché

- Et un autre commit

```
$ edit; git add; git commit

         HEAD (refers to commit 'f')
          |
          v
      e---f
     /
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

checkout

checkout et HEAD détaché

- Revenons au MASTER

```
$ git checkout master

                HEAD (refers to branch 'master')
      e---f      |
     /           v
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

Il n'y a plus de pointeur sur l'état f ...

Sauf si l'on créé une nouvelle branche :

```
Warning: you are leaving 1 commit behind,
not connected to any of your branches:   5dab203 ..
 If you want to keep them by creating a new branch,
 this may be a good time to do so with:
 git branch new_branch_name 5dab203...
 Switched to branch 'master'
```

PLAN

Rebase

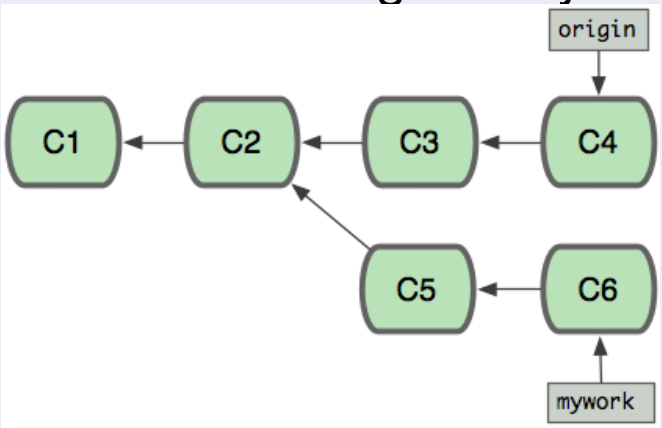
Rebase

- Objectif : Répercuter les commits d'une branche sans passer par la commande *merge*
- *Avantage : permet d'obtenir un historique plus clair (/linéaire)*

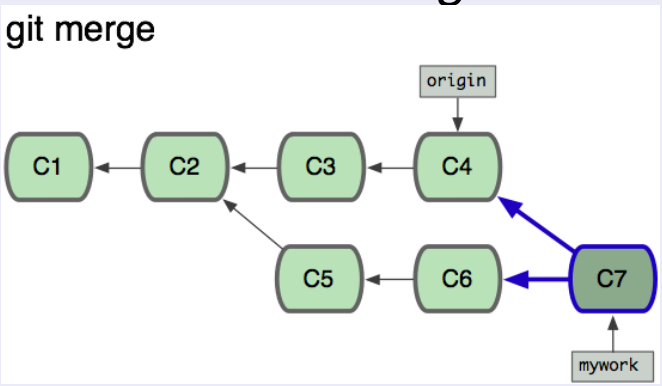
rebase

Exemple

- Situation initiale (avec 2 branches : origin + mywork)



- Récupérer les commits de la branche *origin avec merge*



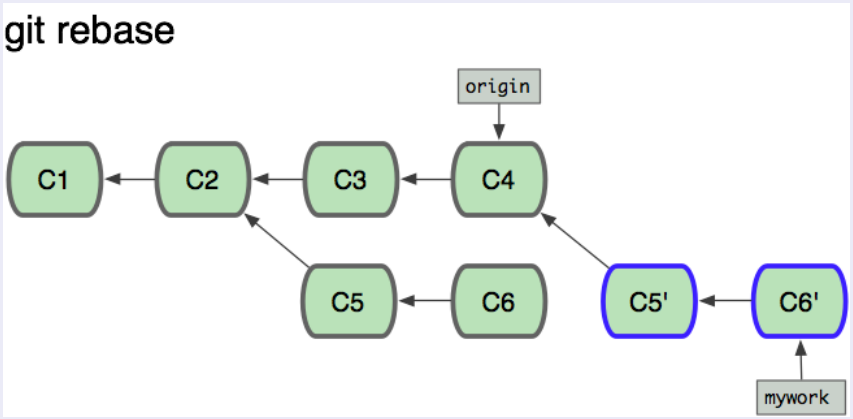
On récupère les commits de la branche origin en un seul nouveau commit (c7).

rebase

Exemple

- Récupérer les commits de la branche *origin avec rebase nouvelle branche :*

```
$ git checkout mywork
$ git rebase origin
```



On récupère les commits de la branche origin avec tout l'historique.

Git... de survie

F. Langrognet