amU Faculté des sciences Aix Marseille Université

Année 2024-25

Domaine Sciences et Technologies

Master informatique

Génie logiciel : Git

Codes UE : SINA09EL

Gestion de versions

Gestion de versions

Arnaud Labourel

1 Introduction

Comme on l'a déjà dit, il est essentiel lorsqu'on travaille sur un projet logiciel un peu conséquent d'utiliser un système de gestion de version. Les raisons sont les suivantes :

— Cela facilite le travail collaboratif sur un projet et c'est donc quasiment indispensable pour le travail en équipe sur un projet.
— Cela permet de documenter toutes les modifications effectuées.
— Cela permet de sauvegarder un travail sur un serveur distant, et ainsi de prévenir sa perte en cas de problème avec un ordinateur (vol ou panne).
— Cela permet de revenir en arrière, et donc donne un filet de sauvetage au programmeur peu confiant ; il ne prend pas de risque, car il pourra toujours revenir en arrière.

Les principes de base de la gestion de version sont les suivants :

— Le code d'un projet est stocké dans un serveur.
— Les développeurs soumettent des modifications avec des commentaires à chaque fois.
— Le serveur conserve l'historique des mises à jour

Dans cet enseignement, nous allons utiliser le gestionnaire de version git. Les raisons sont les suivantes :

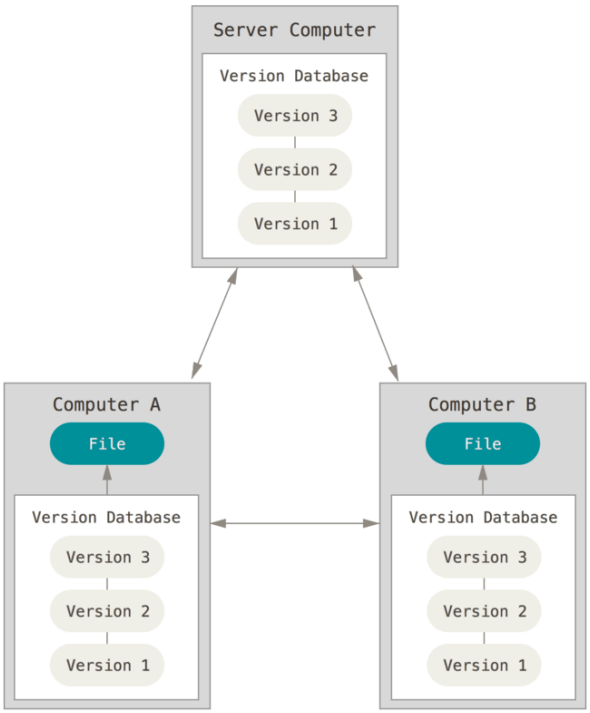
— c'est actuellement le logiciel de gestion de version le plus populaire et le plus utilisé ;
— il existe des serveurs gratuits : github ou gitlab
— il existe une version libre de logiciel serveur : gitlab auto-géré
— git permet la gestion de version décentralisée : la gestion de version se fait aussi en local ce qui permet de faire la gestion de versions sans accès au serveur.

On peut exécuter git via l'IDE (IntelliJ IDEA intègre la gestion de version dans ces menus) ou bien directement en ligne de commande.

De manière simple, l'utilisation que vous allez faire de git va suivre le déroulement suivant :

— À la première utilisation, on crée une copie locale du dépôt git (`clone` ou `init`).
— À chaque commit :
  — on récupère la version courante du dépôt sur le serveur (`pull`) ;
  — on ajoute les fichiers à modifier (`add`) ;
  — on finalise le commit en donnant un message résumant les modifications (`commit`).
— Après un ou plusieurs commits, on met à jour la version distante avec nos modifications (`push`).

Git est un système de gestion de version distribué. Dans un tel système et contrairement aux systèmes de gestion de version non-distribués, les clients n'extraient pas seulement la dernière version d'un fichier, mais ils dupliquent complètement le dépôt. Ainsi, si le serveur disparaît et si les systèmes collaboraient via ce serveur, n'importe quel dépôt d'un des clients peut être copié sur le serveur pour le restaurer. Chaque extraction devient une sauvegarde complète de toutes les données.



Les explications qui suivent sont tirées du Pro Git book.

2 Utilisation basique de Git

2.1 Première utilisation de Git

La première chose à faire est d'installer Git ce que vous pouvez faire en suivant les instructions au lien suivant : https://git-scm.com/book/fr/v2/D%C3%A9marrage-rapide-Installation-de-Git.

La première chose à faire après l'installation de Git est de renseigner votre nom et votre adresse de courriel. C'est une information importante car toutes les validations dans Git utilisent cette information.

```
$ git config --global user.name "Prénom Nom"
$ git config --global user.email votre.adresse@etu.uni-amu.fr
```

Vous pouvez obtenir de l'aide sur les commandes git à l'aide de la commande *help* :

```
$ git help nom_de_la_commande
```

2.2 Démarrer un dépôt Git

Vous pouvez démarrer un dépôt Git de deux manières.

— Vous pouvez prendre un répertoire existant et le transformer en dépôt Git.
— Vous pouvez cloner un dépôt Git existant sur un autre serveur.

Pour créer un dépôt local, il suffit d'appeler la commande `git init` dans le répertoire dans lequel vous voulez

démarrer votre dépôt. Cela crée un nouveau sous-répertoire nommé `.git` qui contient tous les fichiers nécessaires au dépôt. Pour l'instant, aucun fichier n'est encore versionné.

Si vous souhaitez démarrer le contrôle de version sur des fichiers existants (par opposition à un répertoire vide), vous devrez probablement suivre ces fichiers et faire un commit initial. Vous pouvez le réaliser avec quelques commandes `add` qui spécifient les fichiers que vous souhaitez suivre, suivies par un `git commit` :

```
$ git add *.java
$ git commit -m 'initial project version'
```

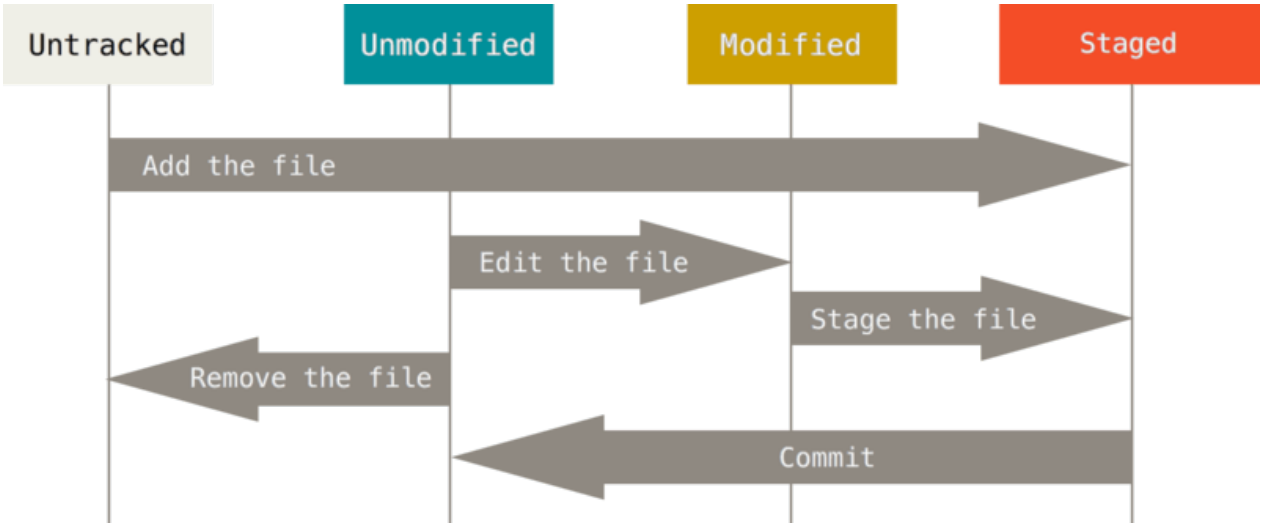
Pour obtenir une copie d'un dépôt Git existant, il faut utiliser la commande `git clone`. Vous clonez un dépôt avec `git clone url` avec url l'adresse serveur du dépôt.

2.3 Enregistrer des modifications dans le dépôt

Une fois que vous avez un dépôt Git valide et une extraction ou copie de travail du projet. Vous devez faire quelques modifications et valider des instantanés de ces modifications dans votre dépôt à chaque fois que votre projet atteint un état que vous souhaitez enregistrer.

Chaque fichier de votre copie de travail peut avoir deux états : sous suivi de version ou non suivi (*untracked*). Les fichiers suivis sont les fichiers qui appartenaient déjà au dernier instantané ; ils peuvent être inchangés (*unmodified*), modifiés (*modified*) ou indexés (*staged*). En résumé, les fichiers suivis sont ceux que Git connaît. Tous les autres fichiers sont non suivis. Quand vous clonez un dépôt pour la première fois, tous les fichiers seront sous suivi de version et inchangés, car Git vient tout juste de les extraire et vous ne les avez pas encore édités.

Au fur et à mesure que vous éditez des fichiers, Git les considère comme modifiés, car vous les avez modifiés depuis le dernier instantané. Vous indexez ces fichiers modifiés et vous enregistrez toutes les modifications indexées, puis ce cycle se répète.



L'outil principal pour déterminer quels fichiers sont dans quel état est la commande `git status`. Supposons que vous souhaitez ajouter un nouveau fichier `README.md` que vous venez de créer. Ce fichier n'est pas en suivi

de version. Pour commencer à suivre ce nouveau fichier, il faut utiliser la commande `git add` suivi du nom de fichier. Vous pouvez entrer ceci :

```
git add README.md
```

Si vous lancez à nouveau la commande git status, vous pouvez constater que votre fichier `README.md` est maintenant suivi et indexé. La commande `git add` accepte en paramètre un chemin qui correspond à un fichier ou un répertoire ; dans le cas d'un répertoire, la commande ajoute récursivement tous les fichiers de ce répertoire.

Il est aussi possible d'indexer (*stage*) des fichiers déjà suivis afin d'enregistrer par la suite dans le dépôt la modification du fichier. La commande `git add` est multi-usage : elle peut être utilisée pour placer un fichier sous suivi de version, pour indexer un fichier ou pour d'autres actions telles que marquer comme résolus des conflits de fusion de fichiers. Sa signification s'approche plus d'ajouter ce contenu pour la prochaine validation que d'ajouter ce contenu au projet.

2.4 Valider vos modifications

Maintenant que vous avez choisis les fichiers indexés, c'est-à-dire les fichiers dont les ajouts ou modifications seront stocké dans le dépôt, vous pouvez valider votre mise-à-jour. Souvenez-vous que tout ce qui est encore non indexé — tous les fichiers qui ont été créés ou modifiés, mais n'ont pas subi de `git add` depuis que vous les avez modifiés — ne feront pas partie de la prochaine validation. Ils resteront en tant que fichiers modifiés sur votre disque. Dans notre cas, la dernière fois que vous avez lancé `git status`, vous avez vérifié que tout était indexé, et vous êtes donc prêt à valider vos modifications. La manière la plus simple de valider est de taper `git commit`.

Vous constatez que le message de validation par défaut contient une ligne vide suivie en commentaire par le résultat de la commande `git status`. Vous pouvez effacer ces lignes de commentaire et saisir votre propre message de validation, ou vous pouvez les laisser en place pour vous aider à vous rappeler ce que vous êtes en train de valider.

Autrement, vous pouvez spécifier votre message de validation en ligne avec la commande git commit en le saisissant après l'option -m, comme ceci :

```
git commit -m "Story 182: Fix benchmarks for speed"
```

Souvenez-vous que la validation enregistre l'instantané que vous avez préparé dans la zone d'index. À chaque validation, vous enregistrez un instantané du projet en forme de jalon auquel vous pourrez revenir ou avec lequel comparer votre travail ultérieur.

2.5 Travailler avec des dépôts distants

Il y a plusieurs manières de travailler avec un dépôt distant. Si vous avez cloné un dépôt avec la commande `git clone`, votre dépôt est automatiquement lié au dépôt distant que vous avez cloné. Si vous avez besoin d'ajouter un nouveau dépôt distant Git, il faut exécuter la commande `git remote add url` avec `url` l'adresse du dépôt.

Lorsque votre dépôt vous semble prêt à être partagé, il faut le pousser en amont, c'est-à-dire envoyer vos

commits dans le dépôt distant. La commande pour le faire est simple : `git push`.

Cette commande ne fonctionne que si vous avez cloné depuis un serveur sur lequel vous avez des droits d'accès en écriture et si personne n'a poussé dans l'intervalle. Si vous et quelqu'un d'autre clonez un dépôt au même moment et que cette autre personne pousse ses modifications et qu'après vous tentez de pousser les vôtres, votre poussée sera rejetée. Vous devrez tout d'abord tirer (commande `git pull`) les modifications de l'autre personne et les fusionner avec les vôtres avant de pouvoir pousser.

3 Utilisation avancée de *git*

3.1 Utilisation des branches

Presque tous les outils de gestion de versions proposent une certaine forme de gestion de branches. Créer une branche signifie diverger de la ligne principale de développement et continuer à travailler sans impacter la ligne principale. Une bonne pratique de développement pour des projets est de créer des branches pour introduire toutes modifications du code. Grâce aux branches, les équipes de développement logiciel peuvent donc apporter des modifications sans affecter la branche principale (*main*). L'historique des *commits* est enregistrée dans une branche créée pour l'ajout de la fonctionnalité, et lorsque le code est prêt, il est fusionné dans la branche *main*. Les branches permettent donc d'organiser le développement et de séparer le travail en cours du code stable et testé de la branche *main*. Cela permet d'éviter que certains bugs et vulnérabilités ne se glissent dans le code principal et n'affectent les utilisateurs, car il est plus facile de les tester et de les trouver dans une branche séparée.

La manière dont Git gère les branches est très légère et permet de réaliser les opérations sur les branches de manière rapidement. À la différence d'autres outils, Git encourage la création et la fusion fréquentes de branches, jusqu'à plusieurs fois par jour. Bien comprendre et maîtriser cette fonctionnalité vous permettra de faire de Git un outil puissant et unique et peut totalement changer votre manière de développer. Pour avoir davantage de détails sur la création de branches, on vous conseille de consulter la partie consacrée aux branches dans le *Git book*.

3.1.1 Stockage des données sous *git*

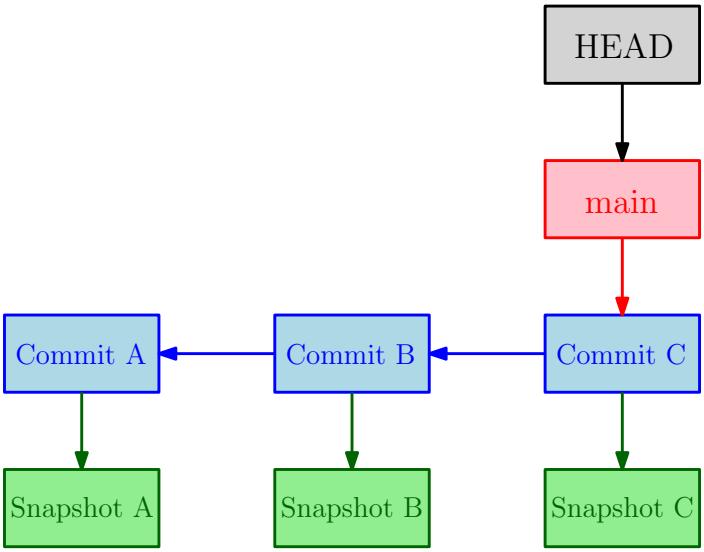
Avant de détailler la manière dont les branches fonctionnent, il faut détailler la manière dont git stocke les données sous forme de *commits*. Lorsque vous faites un commit, *Git* stocke un objet *commit* qui contient un pointeur vers l'instantané (*snapshot*) du contenu que vous avez indexé. L'instantané correspond à une arborescence sous forme d'un objet *tree*. Un *tree* peut contenir récursivement d'autres *trees* pour représenter les sous-répertoires, mais aussi des *blobs* contenant les données brutes des fichiers du dépôt. L'objet correspondant au *commit* contient aussi des informations sur le *commit* (le nom et prénom de l'auteur ainsi que le message associé) mais surtout des pointeurs vers le ou les *commits* qui précèdent directement ce *commit* :

- aucun parent pour le *commit* initial,
- un parent pour un commit normal et
- de multiples parents pour un commit qui résulte de la fusion d'une ou plusieurs branches.

Une branche dans *Git* est simplement un pointeur léger et déplaçable vers un de ces *commits*. La branche par

défaut dans *Git* s'appelle `main`, mais elle n'a pas véritablement une branche spéciale. Elle est identique à toutes les autres branches. La seule raison pour laquelle chaque dépôt en a une est que la commande `git init` la crée par défaut. Au fur et à mesure des validations, la branche `main` pointe vers le dernier des `commits` réalisés. À chaque validation, le pointeur de la branche `main` avance automatiquement.

L'exemple ci-dessous illustre la structure de donnée d'un dépôt après trois *commits* A, B et C. La branche *main* pointe sur le troisième *commit* C qui a un pointeur ver le commit B qui a lui-même un pointeur vers le commit A. Afin de pouvoir déterminer en tout tant quelle est la branche courante, *Git* conserve un pointeur spécial appelé `HEAD` pointant sur celle-ci.

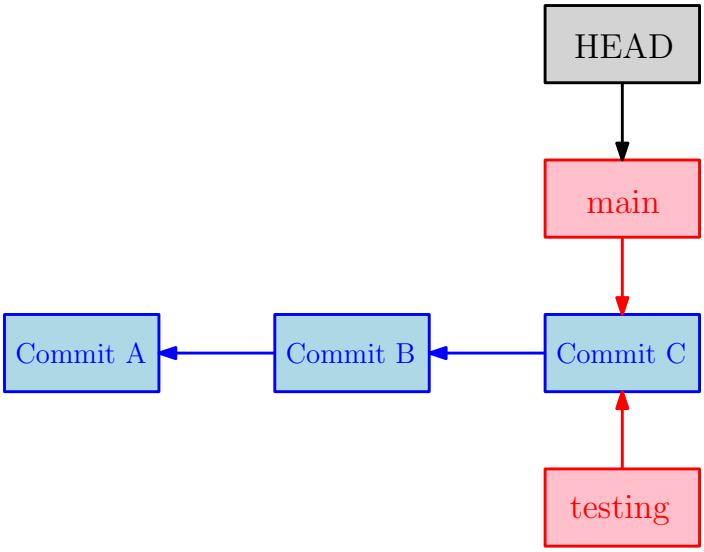


3.1.2 Création de branche

Lors de la création d'une branche, *Git* construit simplement un nouveau pointeur vers le *commit* courant. Supposons que vous créez une nouvelle branche nommée `testing` à l'aide de la commande suivante :

```
git branch testing
```

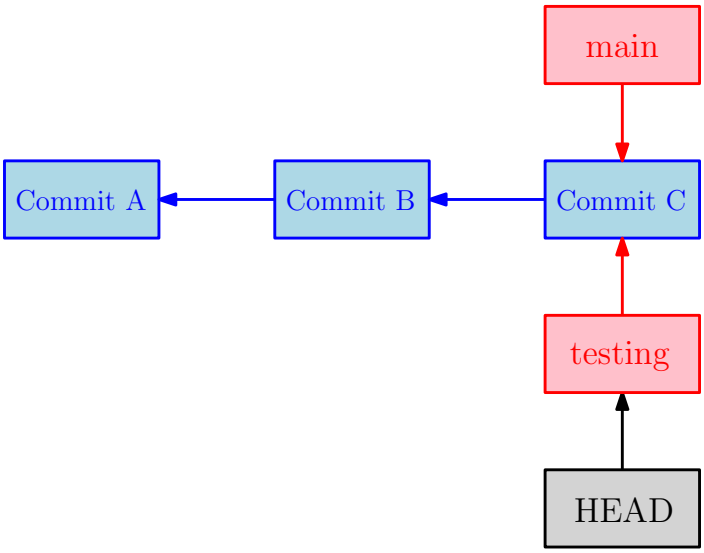
Dans *Git*, la création de branche ne vous fait pas changer de branche et donc la branche courante et donc la branche pointée par `HEAD` reste `main`.



Pour basculer sur une branche existante, il suffit de lancer la commande `git checkout`. Il est donc possible de basculer sur la nouvelle branche `testing` avec la commande suivante :

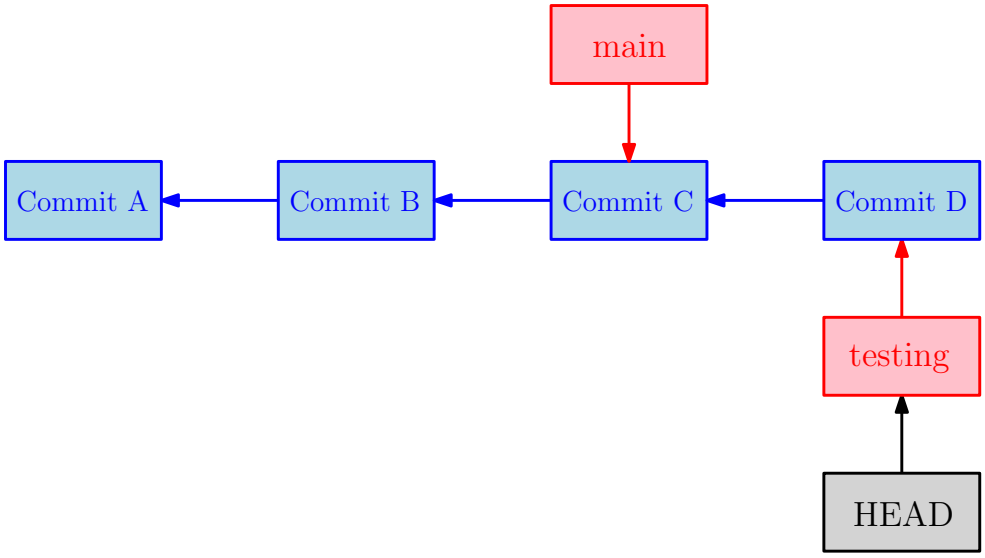
```
git checkout testing
```

Cela déplace le pointeur `HEAD` sur la branche `testing` comme l'illustre la figure ci-dessous :



Les commits que vous effectuez par la suite impacteront la nouvelle branche courante pointée par `HEAD`. Afin de gagner du temps, il est possible de créer une branche et directement changer la branche en une seule commande. Pour cela, il faut utiliser la commande `checkout` avec l'option `-b`. La commande `git checkout -b testing` est donc équivalente à `git branch testing` suivi de `git checkout testing`.

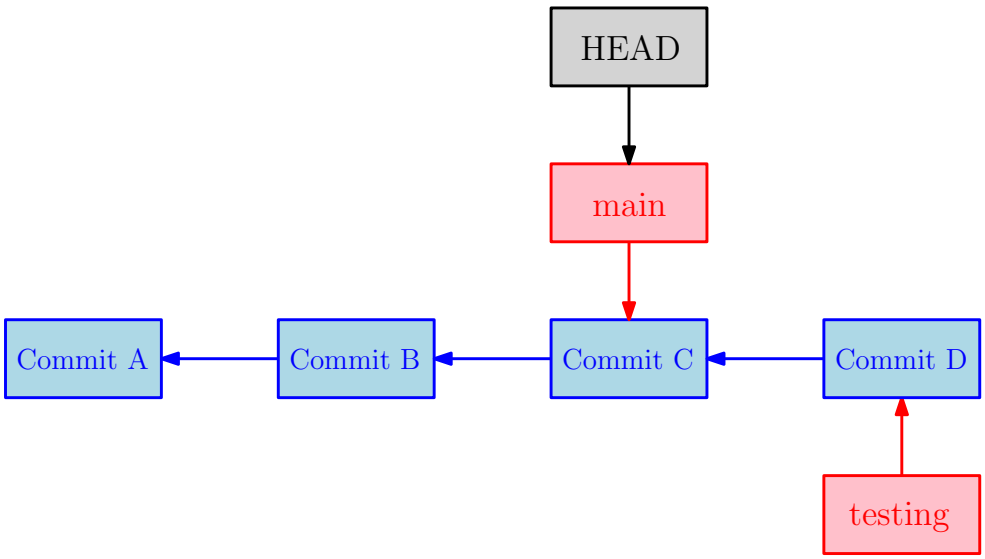
Si on fait un quatrième *commit* alors qu'on est dans la branche `testing`, le pointeur de la branche `testing` est automatiquement changé pour pointer vers ce nouveau *commit*, mais par contre le pointeur de la branche `main` n'est pas modifié.



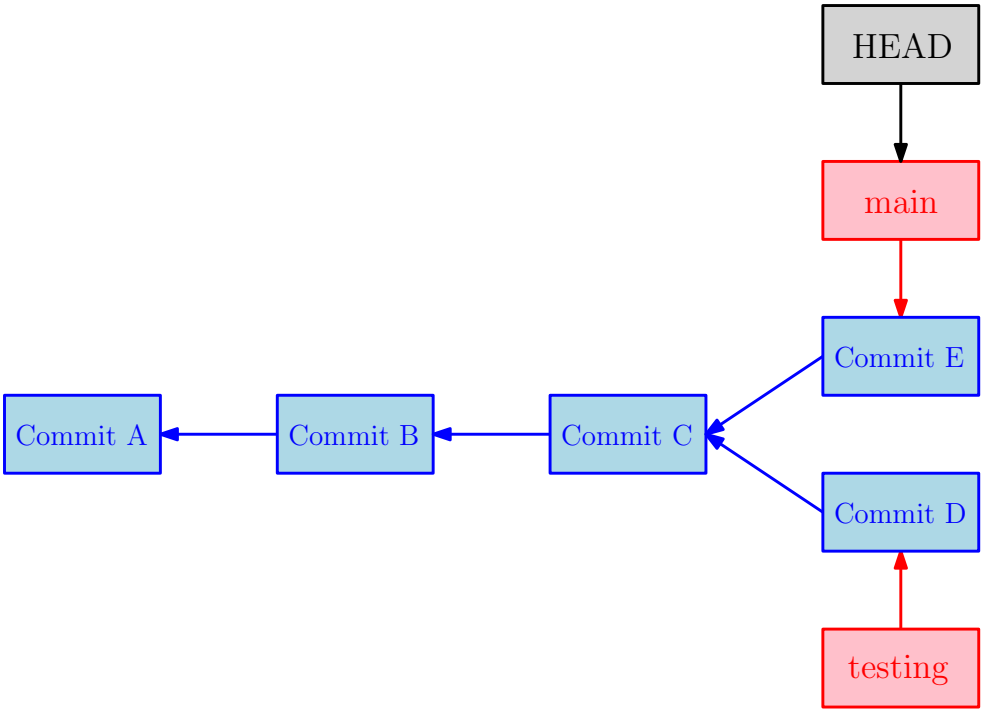
Il est possible de retourner à la branche `main` grâce à la commande suivante :

```
git checkout main
```

Cela nous donne la configuration suivante pour laquelle le pointeur `HEAD` pointe sur la branche `main` correspondant au *commit* C :



Si on réalise un nouveau *commit* E dans cette configuration, cela crée une divergence.



Des fois il est utile de visualiser l'arborescence git de votre dépôt. Vous pouvez utiliser pour cela la commande `log` de `git` avec un certain nombre d'options :

```
git log --oneline --decorate --graph --all
```

Pour notre exemple, le résultat obtenu est le suivant (il suffit de taper `q` pour quitter la visualisation) :

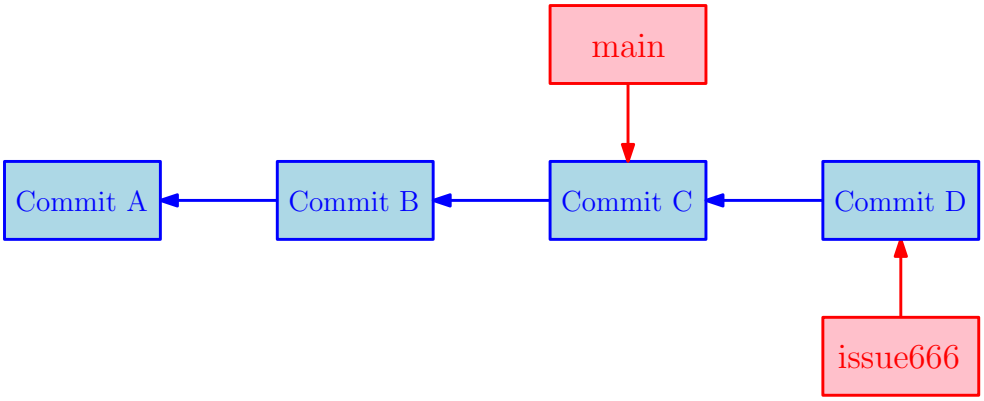
```
* e0a5214 (HEAD -> main) Commit E
| * 675776b (testing) Commit D
|/
* c568bb1 Commit C
* 2d8cf5f Commit B
* 3563128 Commit A
```

3.1.3 Branches et fusions

Afin d'illustrer l'utilisation de fusion de branches, on va considère un exemple simple faisant intervenir des branches et des fusions (*merges*) qu'il est possible de trouver dans le monde réel. Supposez que vous effectuiez les tâches suivantes :

— vous travaillez sur une application mobile ;
— vous créez une branche `issue666` pour prendre en compte un problème (*issue*) dans l'application qui est numéroté 666 dans l'outil de gestion des tâches que votre entreprise utilise ;
— vous commencez à travailler sur cette branche `issue666` et vous avez déjà fait un commit sur cette nouvelle branche.

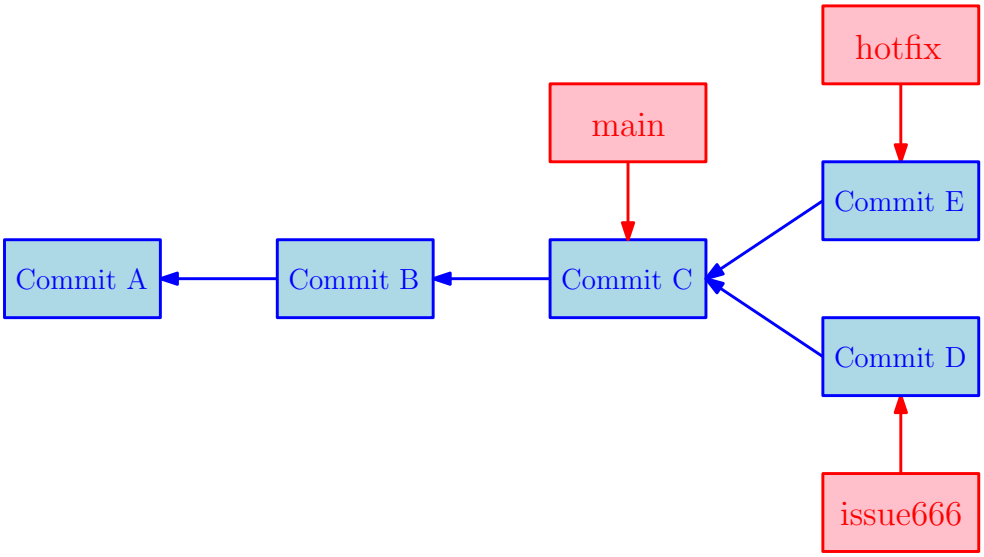
On peut imaginer que la configuration du dépôt git est la suivante :



À ce moment, vous recevez un appel de votre supérieur (le *product owner* de l'application) pour vous dire qu'un problème critique a été découvert dans l'application et qu'il faut le régler au plus tôt. Vous faites donc ce qui suit :

— vous basculez sur la branche `main` de production (avec `git checkout main`) ;
— vous créez une branche `hotfix` pour y ajouter le correctif et vous faites un commit sur cette branche ;

On peut imaginer que la configuration du dépôt git est alors la suivante :



Vous testez le correctif et il semble correct. La prochaine étape est donc de fusionner la branche du correctif avec la branche `main` et de pousser le résultat en production.

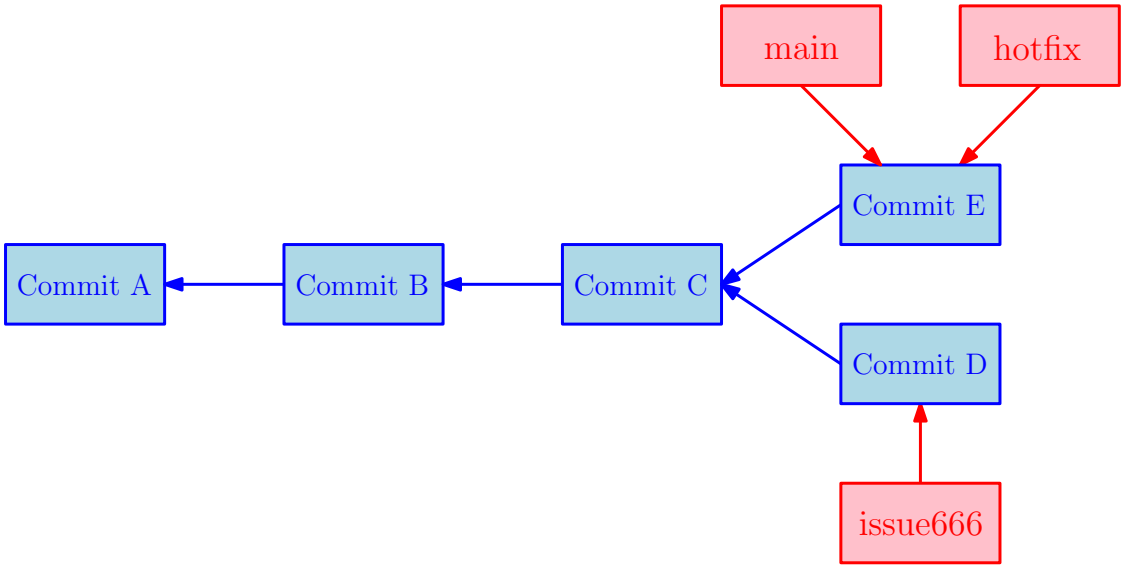
Pour cela vous devez d'abord passer dans la branche `main` avec la commande :

```
git checkout main
```

Ensuite, vous pouvez fusionner la branche `hotfix` dans la branche `main` avec la commande `merge` de *Git* :

```
git merge hotfix
```

On obtient la configuration suivante dans laquelle les branches `main` et `hotfix` pointent vers le même *commit* E :

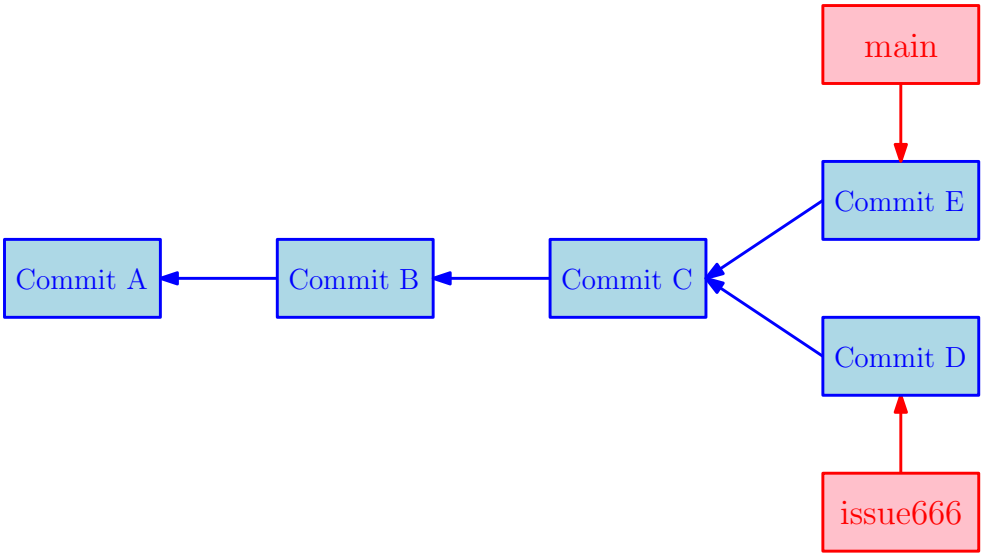


Ce type de fusion est simple, car le *commit* E pointé par la branche `hotfix` qu'on a fusionnée était directement après le *commit* C pointé initialement par la branche `main`. Lorsque l'on cherche à fusionner un *commit* qui peut être atteint en parcourant l'historique depuis le *commit* d'origine, *Git* se contente d'avancer le pointeur, car il n'y a pas de travaux divergents à fusionner. Ceci s'appelle un *fast-forward* (avance rapide).

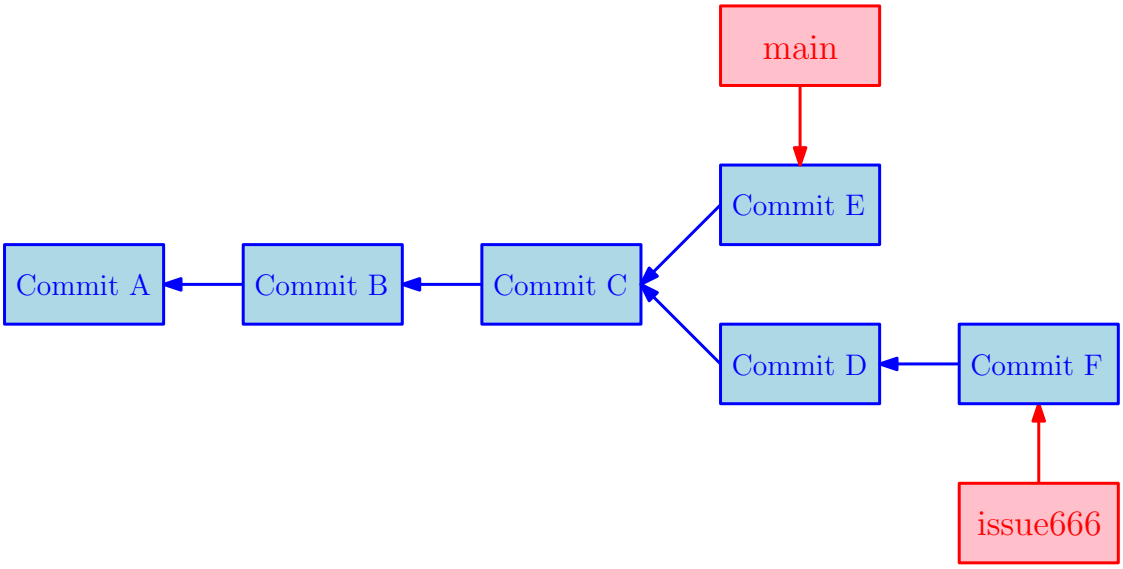
Une fois que la branche `hotfix` est fusionnée, elle n'est plus utile. On peut donc la supprimer en utilisant la commande `branch` et l'option `-d` nous donnant la commande suivante :

```
git branch -d hotfix
```

L'arborescence du dépôt a donc maintenant la configuration suivante :



Par la suite, vous repassez sur la branche `issue666` et vous ajoutez un *commit* à cette branche afin de finaliser le code réglant le problème de l'application. On est donc dans la situation suivante :



Supposons que vous ayez décidé que le travail sur le problème 666 était terminé et prêt à être fusionné dans la branche `main`. Pour ce faire, vous allez fusionner votre branche `issue666` de la même manière que vous l'avez fait plus tôt pour la branche `hotfix`. il suffit pour cela de passer à la branche `main` avec la commande :

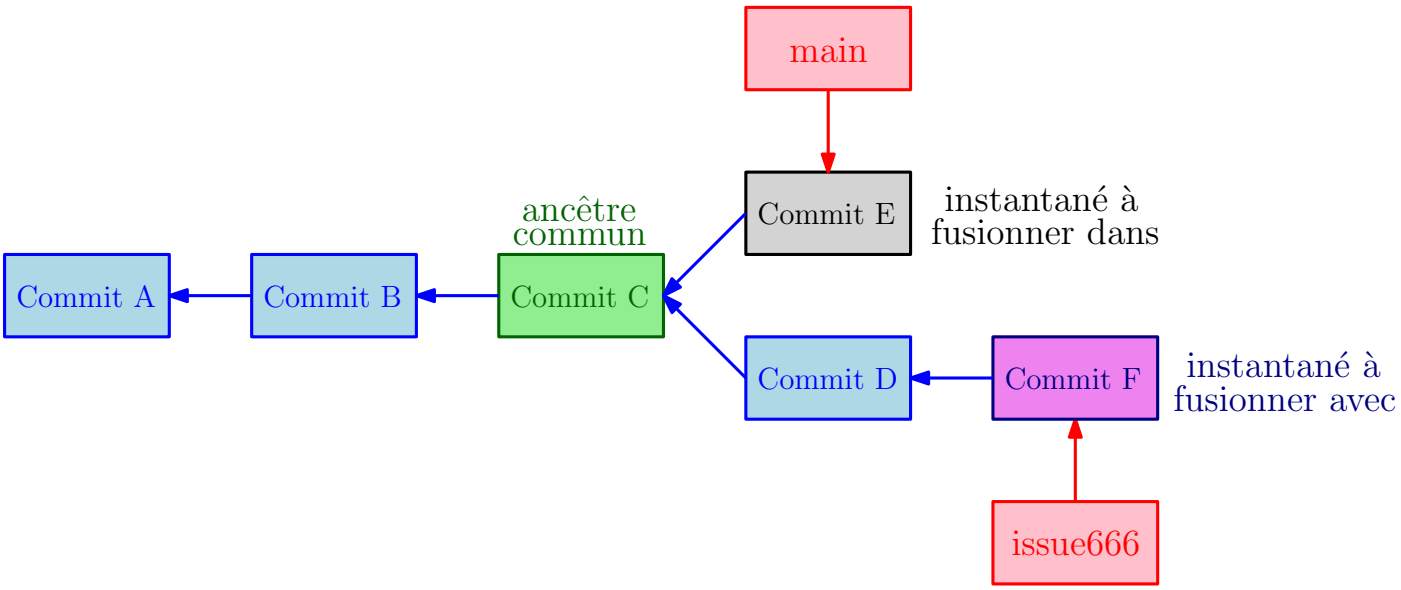
```
git checkout main
```

Ensuite vous pouvez réaliser la fusion de `issue666` dans `main` avec la commande :

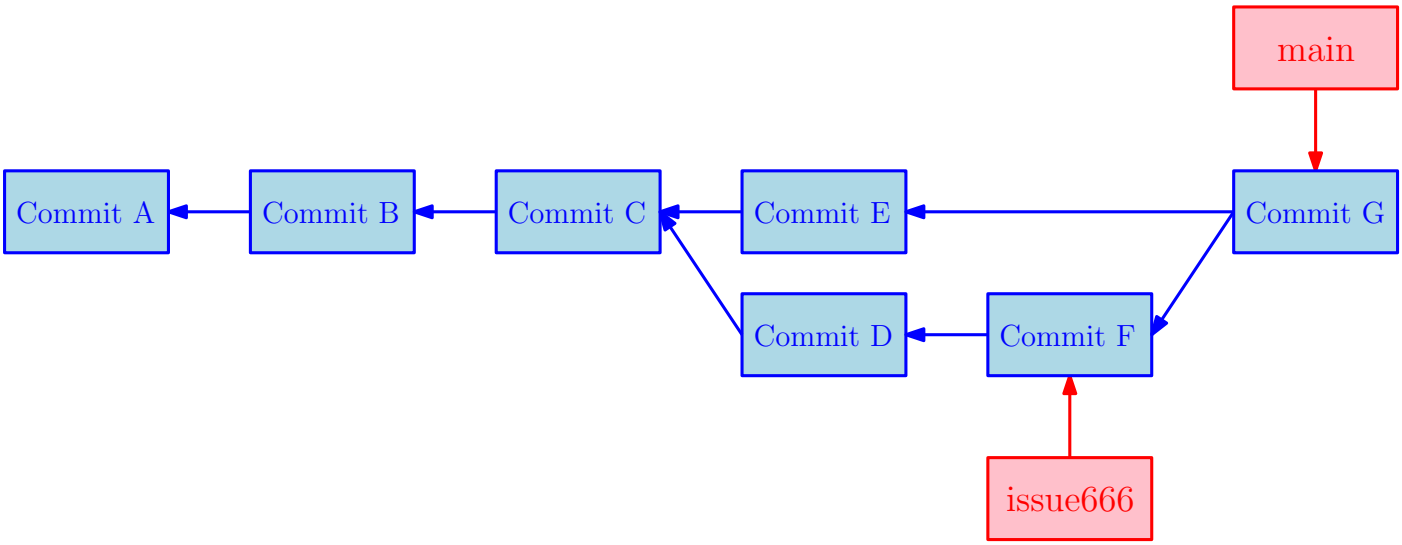
```
git merge issue666
```

Le comportement de cette commande est légèrement différent de celui observé pour la fusion précédente de la

branche `hotfix`. Dans ce cas, à un certain moment, l'historique de développement a divergé. Comme le *commit* sur la branche sur laquelle vous vous trouvez n'est pas un ancêtre direct de la branche que vous cherchez à fusionner, *Git* doit effectuer quelques actions. Dans ce cas, *Git* réalise une fusion à trois sources (*three-way merge*), en utilisant les deux instantanés pointés par les sommets des branches (correspondant aux *commits* E et F dans notre exemple) ainsi que leur plus proche ancêtre commun (correspondant au *commit* C dans notre exemple).



Au lieu d'avancer simplement le pointeur de branche, Git crée donc un nouvel instantané qui résulte de la fusion à trois sources et crée automatiquement un nouveau *commit* qui pointe dessus. On appelle ceci un commit de fusion (*merge commit*) qui est spécial en cela qu'il a plus d'un parent. On obtient donc la configuration suivante :



À présent que votre travail a été fusionné, vous n'avez plus besoin de la branche `issue666`. Vous pouvez donc supprimer la branche avec la commande suivante :

```
git branch -d issue666
```

3.1.3.1 Gestion de conflits de fusion Quelques fois, le processus de fusion ne se déroule pas sans problèmes. Si vous avez modifié différemment la même partie du même fichier dans les deux branches que vous souhaitez fusionner, *Git* ne sera pas capable de réaliser proprement la fusion. Si votre résolution du problème 666 a modifié la même section de fichier que le `hotfix`, vous obtiendrez un conflit qui ressemblera à ceci :

```
$ git merge iss53
Auto-merging Main.java
CONFLICT (content): Merge conflict in Main.java
Automatic merge failed; fix conflicts and then commit the result.
```

Git n'a pas automatiquement créé le *commit* de fusion. Il a arrêté le processus le temps que vous résolviez le conflit. Si vous voulez vérifier, à tout moment après l'apparition du conflit, quels fichiers n'ont pas été fusionnés, vous pouvez lancer la commande `git status` :

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      Main.java

no changes added to commit (use "git add" and/or "git commit -a")
```

Tout ce qui comporte des conflits et n'a pas été résolu est listé comme *unmerged*. *Git* ajoute des marques de résolution de conflit standards dans les fichiers qui comportent des conflits, pour que vous puissiez les ouvrir et résoudre les conflits manuellement. Votre fichier devrait contenir des sections qui ressemblent à ceci :

```
<<<<<<< HEAD:Main.java
System.out.println("Hello world !")
=======
System.out.println("Hello Arnaud !")
>>>>>>> issue66:Main.java
```

Cela signifie que la version dans `HEAD` (votre branche `main`, parce que c'est celle que vous aviez extraite quand vous avez lancé votre commande de fusion) est la partie supérieure de ce bloc (tout ce qui se trouve au-dessus de la ligne `=======`), tandis que la version de votre branche `issue666` se trouve en dessous. Pour résoudre le conflit, vous devez choisir une partie ou l'autre ou bien fusionner leurs contenus vous-même. Par exemple, vous pourriez choisir de résoudre ce conflit en remplaçant tout le bloc par ceci :

```
System.out.println("Hello world !\nHello Arnaud !")
```

Cette résolution comporte des éléments de chaque section et les lignes `<<<<<<<`, `=======` et `>>>>>>>` ont été complètement effacées. Après avoir résolu chacune de ces sections dans chaque fichier comportant un conflit, lancez `git add` sur chaque fichier pour le marquer comme résolu. Placer le fichier dans l'index marque le conflit comme résolu pour *Git*.

Si vous souhaitez utiliser un outil graphique pour résoudre ces conflits, vous pouvez lancer la commande `mergetool` qui démarre l'outil graphique de fusion approprié et vous permet de naviguer dans les conflits :

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis
↪  bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'Main.java':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Si vous souhaitez utiliser un outil de fusion autre que celui par défaut (ici `opendiff` a été ouvert), vous pouvez voir tous les outils supportés après l'indication « of the following tools: ». Les outils plus populaires sont les suivants :

- meld
- p4merge
- kdiff3
- tortoisemerge

Par exemple, vous pouvez changer l'outil par défaut pour `meld` (après l'avoir installé) avec la commande suivante :

```
git config --global merge.tool p4merge
```

Après avoir quitté l'outil de fusion, *Git* vous demande si la fusion a été réussie. Si vous répondez par la positive à l'outil, il ajoute le fichier dans l'index pour le marquer comme résolu. Vous pouvez lancer à nouveau la commande `git status` pour vérifier que tous les conflits ont été résolus :

```
$ git status
On branch main
All conflicts fixed but you are still merging.
```

```
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   Main.java
```

Si cela vous convient et que vous avez vérifié que tout ce qui comportait des conflits a été ajouté à l'index, vous pouvez entrer la commande `git commit` pour finaliser le commit de fusion. Le message de validation par défaut ressemble à ceci :

```
Merge branch 'issue666'

Conflicts:
    Main.java
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch main
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#
```

Vous pouvez modifier ce message pour inclure les détails sur la manière dont le conflit a été résolu si vous pensez que cela peut être utile lors d'une revue ultérieure. Indiquez pourquoi vous avez fait ces choix, si ce n'est pas clair.

3.1.4 Dépôt distant et branches

3.1.4.1 Utilisation de push Si vous souhaitez *push* votre branche sur le dépôt distant du serveur, il vous faut spécifier la première fois la branche correspondante sur le serveur. Par exemple, si vous souhaitez avoir sur le serveur une branche `new_branch` correspondant au contenu de la branche courante, vous pouvez utiliser la commande suivante :

```
git push --set-upstream origin new_branch
```

3.1.5 Merge requests

Une fois qu'une fonctionnalité est finalisée au sein d'une branche il est souvent utile de ne pas faire directement la fusion, mais plutôt de demander au préalable sa validation par une ou plusieurs personnes (comme le *product owner*). Pour cela, il faut, sur *gitlab*, passer par une *merge request*. Vous pouvez faire cette demande lors d'un `push` en spécifiant l'option `merge_request.create`. Vous pouvez choisir la branche cible avec l'option `merge_request.target`. Par exemple, la commande suivante réalise un `push` de la branche courante demandant la création d'un *merge request* vers la branche `main`.

```
push -o merge_request.create -o merge_request.target=main
```

Une fois la demande envoyée, les personnes devant valider la fusion peuvent accéder dans l'interface web de l'instance *gitlab* aux requêtes à valider. Pour cela, on peut passer par l'option *merge requests* dans le menu utilisateur ou bien l'option *code* → *merge requests* dans le menu du projet. Il est possible de valider les *merge requests* en cliquant sur *merge*.