ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Travailler avec Git

Le GitHub flow avec GitLab

1. Introduction

Un workflow définit un plan d'organisation de la collaboration de plusieurs développeurs pour la production de logiciels. La version présentée ici est le *GitHub flow* qui est une version simplifiée du *GitFlow*. Ce workflow est adapté au développement de projets sans réelle gestion de distributions, tels que les projets étudiants. La collaboration se fait par l'intermédiaire d'un dépôt centralisé que partagent tous les développeurs et de dépôts locaux propres à chaque développeur.

Dans la description qui suit, le dépôt central du projet est supposé localisé sur https://gitlab.ecole.ensicaen.fr. Les commandes décrites sont les commandes de base de Git. Ces commandes sont aussi directement exécutables à partir de l'interface d'un IDE digne de ce nom.

2. Organisation du dépôt central

Le développement d'un projet s'organise typiquement autour d'une branche permanente de production nommée `master`, commune à tous les développeurs, et de plusieurs branches temporaires de travail, personnelles à chaque développeur. La Figure 1 donne un exemple d'une telle organisation avec deux branches temporaires.

2.1 Une branche permanente de production : *master*

La branche `master` contient la version courante du logiciel. C'est la branche de fusion des contributions des développeurs. Seul le responsable du dépôt peut modifier cette branche.

La version du logiciel dans `master` doit toujours être fonctionnelle. Dans l'idéal, la branche utilise l'intégration continue pour assurer que tout ce qui est ajouté à cette branche ne remet pas en cause le caractère fonctionnel du

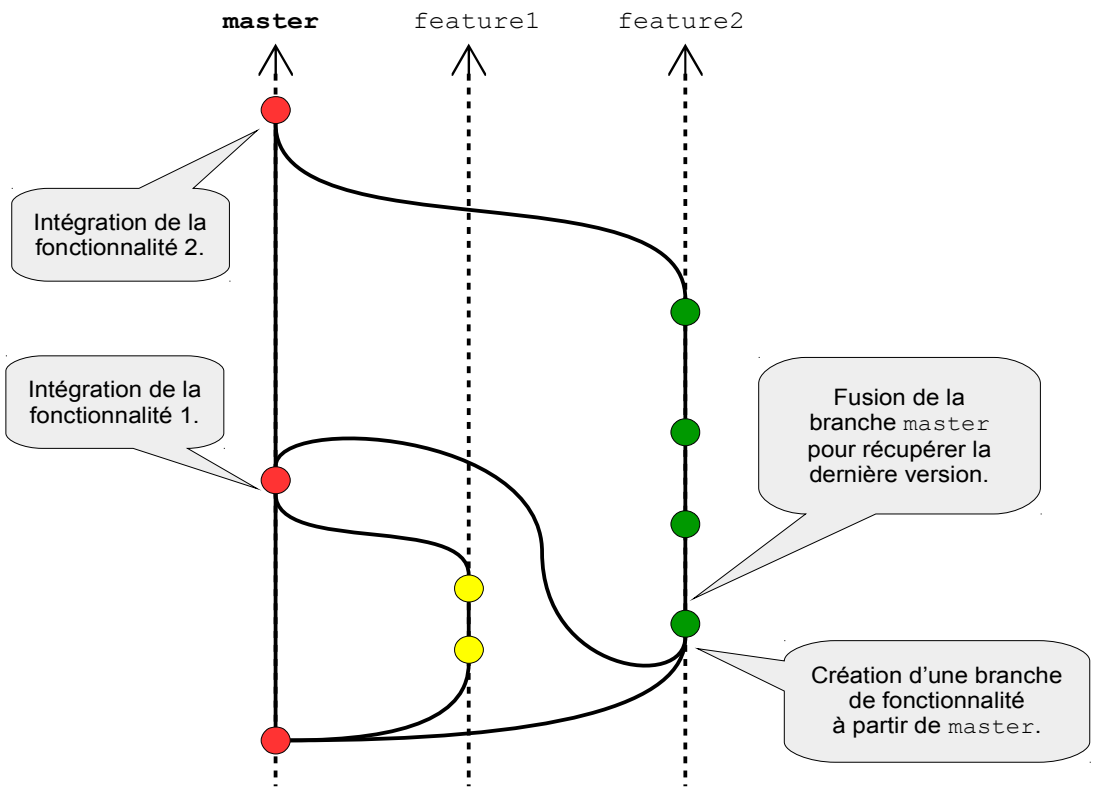


Figure 1. Organisation des branches sur le dépôt central.

logiciel.

2.2 Des branches temporaires de développement : *features*

À chaque fois qu'un développeur veut modifier la version courante du logiciel dans `master` pour ajouter une fonctionnalité ou corriger un bug, il doit créer une branche spécifique qui émane de `master`.

Quand la fonctionnalité est complète, la branche est poussée sur le dépôt central et le responsable du dépôt la fusionne à la branche `master` courante, puis la branche de fonctionnalité est détruite. Cette branche réside essentiellement sur le dépôt local du développeur jusqu'à la fin du développement où elle sera alors poussée sur le dépôt commun pour être fusionnée à la branche `master`. Néanmoins, elle peut aussi être poussée régulièrement sur le dépôt central pour assurer des sauvegardes intermédiaires et échanger avec les autres développeurs.

3. Le workflow

Examinons plus précisément comment une petite équipe formée de trois développeurs, Estelle, Sébastien et Loïc collaborent selon le workflow

via un dépôt centralisé sur GitLab. Estelle est la responsable du dépôt centralisé.

3.1 Initialisation du dépôt

3.1.1 Estelle initialise le dépôt central sur GitLab

Estelle crée le dépôt central sur *gitlab.ecole.ensicaen.fr*. Elle ajoute Loïc et Sébastien comme développeurs. Elle protège la branche `master` contre la modification par les autres développeurs. Estelle sera la seule à gérer cette branche pour garantir l'unité du projet.

Estelle réalise ces opérations par l'intermédiaire de l'interface Web de GitLab. Mais, elle peut aussi créer ce dépôt à partir de la ligne de commandes :

```
mkdir depot
cd depot
git init
touch README.md
git add README.md
git commit -m 'Initialisation du dépôt'
git remote add origin https://estelle@gitlab.ecole.ensicaen.fr/estelle
  /depot.git
git push -u origin master
```

3.1.2 Tous les développeurs clonent le dépôt central

Les trois développeurs créent leur dépôt local par copie du dépôt central.

```
git clone https://estelle@gitlab.ecole.ensicaen.fr/estelle/depot.git
cd depot
```

Chacun est prêt à collaborer au développement du logiciel.

3.2 Développement d'une fonctionnalité

Loïc commence son travail de développement d'une fonctionnalité. La Figure 2 résume les activités de développement d'une fonctionnalité.

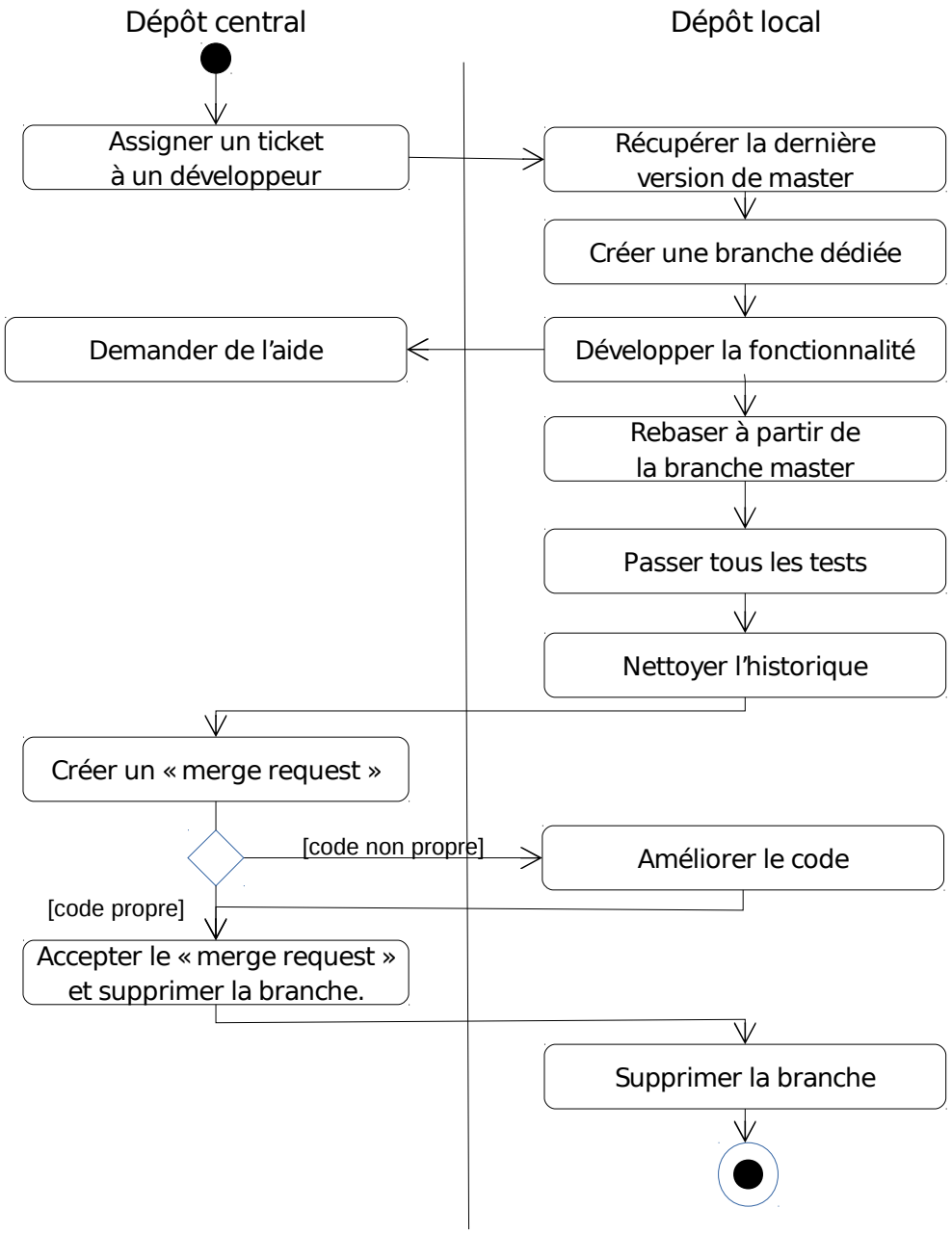


Figure 2. Diagramme d'activités de développement d'une fonctionnalité.

3.2.1 Loïc prend un ticket dans GitLab

Il faut toujours qu'il y ait un ticket (nommé *issue* dans GitLab) correspondant à tout changement dans le code. La plupart des tickets sont créés au début d'un sprint à partir du backlog. D'autres tickets sont créés à la volée par remontée de bug ou identification d'étapes intermédiaires apparues au cours du développement.

Le ticket doit porter un nom très explicite, par exemple : « *query database* » pour une fonctionnalité ou « *bug on signal processing* » pour un bug.

Pour commencer son travail, Loïc s'assigne le ticket en utilisant l'interface de GitLab.

3.2.2 Loïc crée sa branche de travail

Loïc crée une branche pour ce ticket à partir de la branche `master`. La branche est d'abord une copie de la version courante de `master` du dépôt. Le nom de la branche devrait être construit avec le numéro et le nom du ticket, par exemple « *15_query_database* ».

```
git checkout master
git pull origin master # récupérer la dernière version sur le dépôt central
git checkout -b 15_query_database # création de la branche
```

3.2.3 Loïc développe sa fonctionnalité

Sur la branche `15_query_database`, Loïc édite, crée, modifie et supprime des fichiers du projet avec son IDE pour ajouter sa fonctionnalité. En suivant le principe des petits pas, il fait autant de *commits* que nécessaire pour garder trace de ses différentes modifications et pouvoir revenir en arrière en cas de nécessité. Cela peut représenter plusieurs *commits* par heure, en fait à chaque fois qu'une étape est franchie dans le développement de la fonctionnalité.

Remarque

En général, Loïc fait un *commit* directement avec tous les fichiers modifiés dans son dossier de travail :

```
git add -A
git commit -m "Descriptif du commit"
```

Mais, il arrive que lorsque Loïc travaille sur une fonctionnalité, il corrige une petite erreur qui n'a rien à voir avec la fonctionnalité en cours, par exemple un bug ou une faute d'orthographe dans un commentaire. Dans ce cas, il pousse uniquement le fichier modifié dans la zone d'index (*staging area*) et il fait un *commit* distinct :

```
git add src/main/java/fr/ensicaen/ecole/projet/DataAccessObject.java
git commit -m "Correction du bug d'accès aux données"
```

Ainsi, la correction est prise en compte mais n'est pas associée à des évolutions de la fonctionnalité.

3.2.4 Loïc demande de l'aide

Pour partager sa version, Loïc pousse sa branche de fonctionnalité sur le GitLab et fait une demande de merge-request, en commençant le titre

de la demande par « WIP ou [WIP] : », c'est-à-dire Work In Progress. Ceci signale à GitLab de ne pas fusionner cette branche à la branche `master`.

Tous les développeurs sont avisés de la demande. Sébastien rapatrie la branche sur son dépôt local et étudie le code. Ensuite, le dialogue passe par les messages associés au merge request sur GitLab. Quand la discussion est close, le merge request est fermé, sans qu'il y ait eu de fusion dans la branche `master` et Loïc poursuit son développement.

3.3 Publication d'une fonctionnalité

Loïc termine sa fonctionnalité et souhaite l'intégrer au dépôt central.

3.3.1 Loïc synchronise sa branche avec la branche *master*

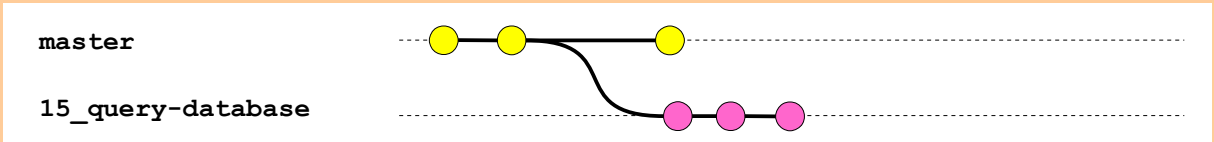
Avant d'envoyer son travail pour intégration au projet, Loïc doit synchroniser sa branche avec la branche `master` courante du dépôt central qui a pu évoluer depuis qu'il en a dérivé. Cela nécessite de résoudre tous les problèmes de conflit entre les deux versions. L'objectif est de faciliter le travail de fusion ultérieur d'Estelle afin qu'elle n'est pas à lire le code en profondeur pour résoudre les conflits ou les bugs résultants.

La synchronisation se fait en changeant la base de la branche de fonctionnalité de Loïc avec la tête courante de la branche `master` comme si la branche de fonctionnalité venait juste de dériver de la branche `master`. Pour cela, Loïc utilise la commande `rebase` :

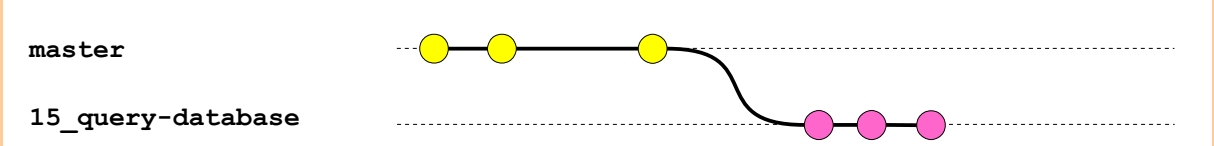
```
git checkout 15_query_database
git rebase master
```

Remarque

La commande `rebase` a pour effet de modifier le départ de la branche :



Après le `rebase` :



3.3.2 Loïc repasse tous les tests

Il est nécessaire de repasser tous les tests pour s'assurer que le *rebase* n'a pas introduit de régression.

3.3.3 Loïc nettoie l'historique de sa branche

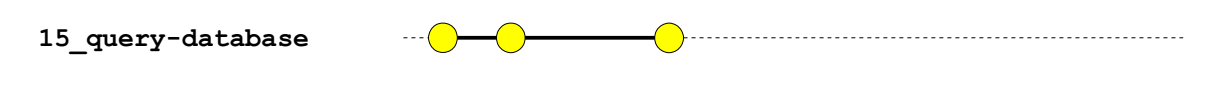
Quand Loïc développe, il fait beaucoup de commits intermédiaires selon le principe des petits pas. Il se retrouve donc avec un historique comportant beaucoup de commits intermédiaires non utiles, par exemple, « *Fix bug on window closing* », puis « *really fix bug on window closing* » puis « *really really fix bug on window closing* ». L'option `-i` de `git rebase` permet de nettoyer l'historique interactivement en fusionnant les commits intermédiaires.

Remarque

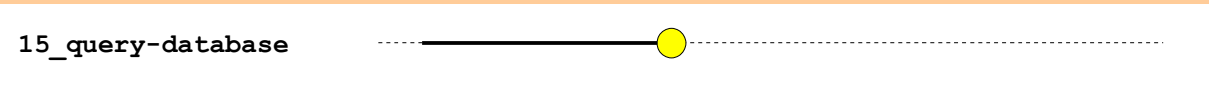
La commande suivante :

```
git rebase -i HEAD~3
```

permet de réécrire l'historique interactivement des trois derniers *commits* et par exemple de les fusionner en un seul :



Le résultat d'une fusion des *commits* donne :



Attention : Il ne faut pas rebaser les commits de la partie d'un historique déjà poussée sur le dépôt central. D'autres développeurs peuvent avoir fait des pointeurs sur des parties de cet historique.

3.3.4 Loïc fait une demande d'intégration de son travail

Une fois la résolution des conflits effectuée, les tests passés et l'historique nettoyé, Loïc pousse sa branche sur le dépôt central :

```
git add -A
git commit -m "Descriptif du commit"
git push origin 15_query_database
```

Puis il remplit un « *merge request* » dans GitLab pour demander à Estelle de fusionner sa branche à `master`.

Dans le message de *commit*, Loïc ajoute les mots « *fixes #15* » ou « *closes #15* » dans la description du *merge request* pour signifier que

cela conclut sa branche et ferme le ticket correspondant.

3.3.5 Estelle reçoit le « merge request » de Loïc

Quand Estelle reçoit la demande d'intégration, elle contrôle la qualité de la contribution, notamment le respect des conventions de codage et la présence des tests. Elle peut décider de modifier ou de faire modifier la contribution par Loïc avant l'intégration dans le projet officiel.

3.3.6 Loïc termine le travail de révision de sa fonctionnalité

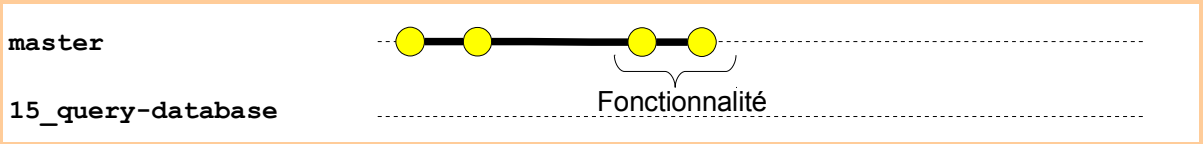
Une fois que la contribution de Loïc est jugée acceptable par Estelle, elle fusionne la contribution sur la branche `master` en utilisant simplement le bouton "Accepter" de l'interface de GitLab, ou bien les lignes de commandes :

```
git pull origin master
git checkout master
git merge -no-ff 15_query_database
```

La fusion doit utiliser l'option `--no-ff` de manière à garder la trace de la branche `15_query_database` dans l'historique de la branche `master`.

Remarque

En *fast-forward*, la branche est ajoutée dans `master` comme s'il s'agissait de commits classiques. Il n'y a plus trace de la branche. Il est impossible de voir à partir de l'historique quels commits font partie de la fonctionnalité ajoutée.



L'option `--no-ff` (*no fast-foward*) force la fusion à toujours créer un nouvel objet commit, même quand le merge peut se faire avec fast-forward. Cela permet de ne pas perdre l'information sur l'existence historique d'une branche de fonctionnalité.