Conduite de projet : Outils et processus pour le développement logiciel collaboratif

Mo Foughali
(IRIF, U-Paris) – foughali@irif.fr
(d'après transparents originaux de Yann Régis-Gianas et Aldric Degorre)

3 octobre 2021

Rappel

Ce que l'on a appris...

- La différence entre **programmer** et **développer**.
- La spécificité de la conduite d'un projet logiciel.
- Les méthodes Agile.
- Les outils pour le développement collaboratif, notamment git.

Les problèmes des comptes GitLab

Assurez-vous que :

- Vous êtes inscrits administrativement (rappel),
- Vous n'avez pas reçu vos identifiants lors des années précédentes,
- Vous n'avez pas reçu vos identifiants dans vos spams.

Si tous les items ci-dessus sont "checked", alors rendez-vous sur cette page (avec un lien de contact)
`http://www.informatique.univ-paris-diderot.fr/wiki/index`

Concentrés de cours

- Après chaque cours, un “concentré de cours” est mis sur Moodle.
- Chaque concentré sera plus concentré que le précédent.
- Vous pouvez prendre des notes[1], mais ne recopiez pas ce qu’il y a sur les slides !

1. de préférence sur vos cahiers pour adhérer au modèle anti-bruit

En travaux pratiques

Sur un total de 25 équipes :

- 1 équipe n'a pas créé de projet GitLab ou bien n'y a pas donné accès à son enseignant
- 15 équipes ont créé leur projet mais n'ont pas travaillé depuis [2]
- 9 équipes ont travaillé de façon visible, mais dont seulement 1 à 3 de façon suffisante. [3]

2. Pas de façon visible et, comme dit le proverbe, « Si ce n'est pas sur GitLab, ça ne s'est pas produit ! »

3. Il faut un backlog, des tâches assignées, des tâches en cours de résolution… et même des merge requests !

En travaux pratiques

En travaux pratiques

- Certains ne sont pas venus ou alors très en retard :

  2 absences injustifiées = cours non validé

  Trop de retards = des points en moins

- Tout membre de l'équipe doit travailler sur une tâche, et une "bonne" tâche.
- Une "bonne" tâche :
  - est clairement formulée ;
  - produit un résultat (document, code, ...) évaluable ;
  - est une activité dont on peut estimer la durée ;
  - est affectée à une personne mais est la responsabilité de toute l'équipe.

Pour la suite

- N'oubliez pas de rajouter vos chargés de TP (et idéalement moi-même) dans les membres du projet Gitlab.
- Chaque séquence de 2 semaines sera l'objet d'un **sprint**.
- La séance de TP type :
  - 1/4 d'heure : bilan du dernier sprint, comment s'améliorer ?
  - 1h45 : préparation du sprint suivant.
- Vous devrez produire une vidéo de présentation de votre projet.
- Rappel : l'évaluation est continue.

Séance de cours d'aujourd'hui

1. Retour sur les outils collaboratifs, Git et GitLab.
2. Retour sur Scrum.

Quelques rappels

- **blob** : Contenus de fichiers sauvegardés par Git.
- **tree** : Arborescence de fichiers.
- **commit** : Description d'un changement.
- **somme de contrôle "checksum" (sha1)** : Identificateur unique pour tout objet stocké par Git.
- **branch** : Séquence de commits caractérisée par un nom.

Notion de références

- Mémoriser des sommes de contrôles, c'est pénible !
- **référence** : nom associé à une arborescence.
- Exemples : `master`, `HEAD`, ...

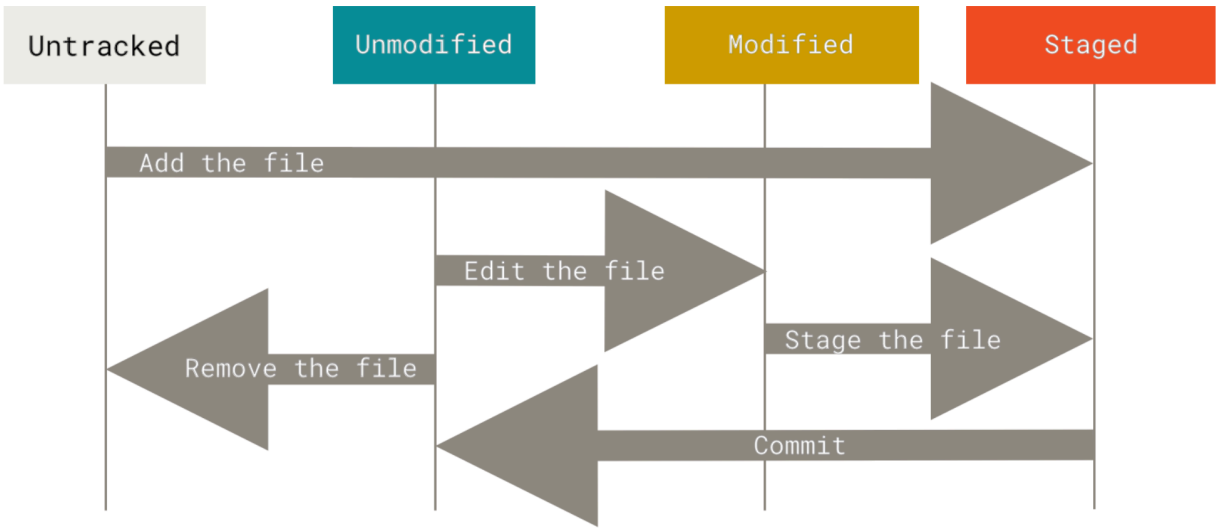
Index, HEAD et arborescence de travail

- Quand on se déplace dans un dépôt Git, en dehors du répertoire `.git`, on se trouve dans **l'arborescence de travail** (ou "working tree", appellée parfois "arborescence courante").
- Cette arborescence peut contenir des fichiers modifiés dont les changements n'ont pas encore été pris en compte par Git.
- Le rôle de l'**index** est de sauvegarder les changements à inclure dans le prochain commit : c'est une sorte de zone de préparation ("staging").
- **HEAD** est une référence qui pointe vers la **branche courante**, qui elle-même pointe vers son commit le plus récent.

Lire : https://git-scm.com/book/fr/v2/Utilitaires-Git-Reset-d%C3%A9mystifi%C3%A9#s_git_reset

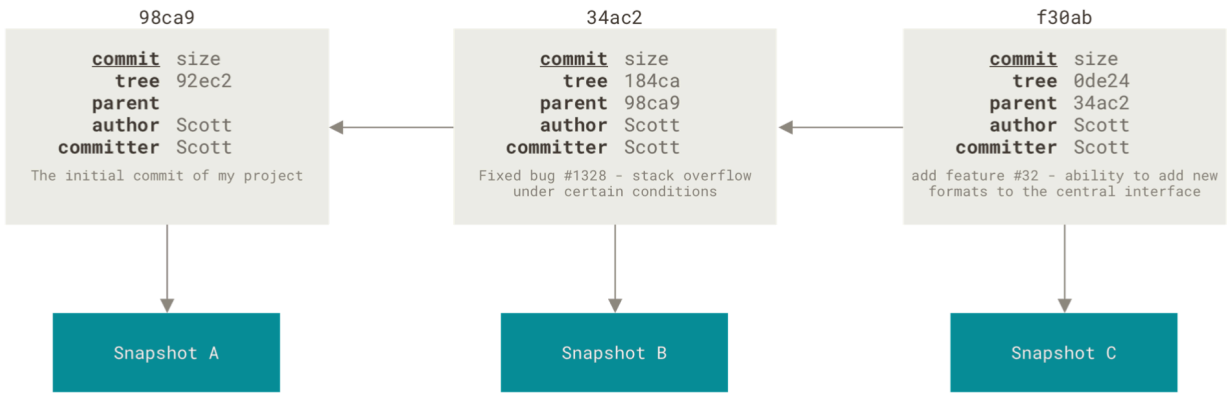
Comment ça marche

Staging



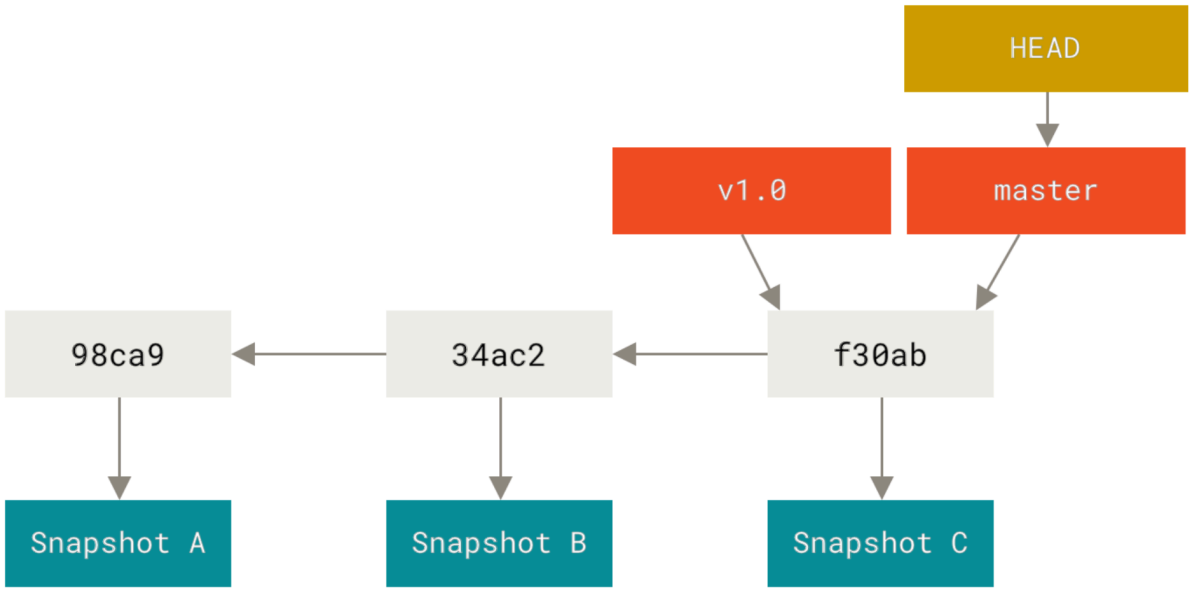
Comment ça marche

Committing (suite)



Comment ça marche

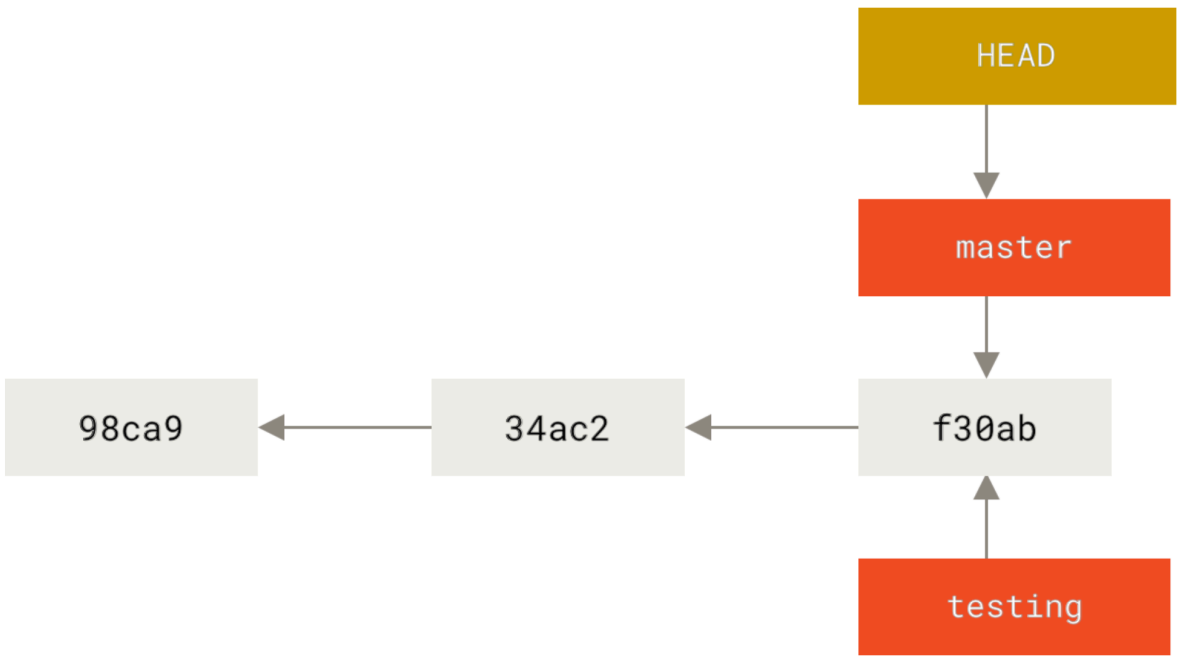
Références



Comment ça marche

Les branches dans Git

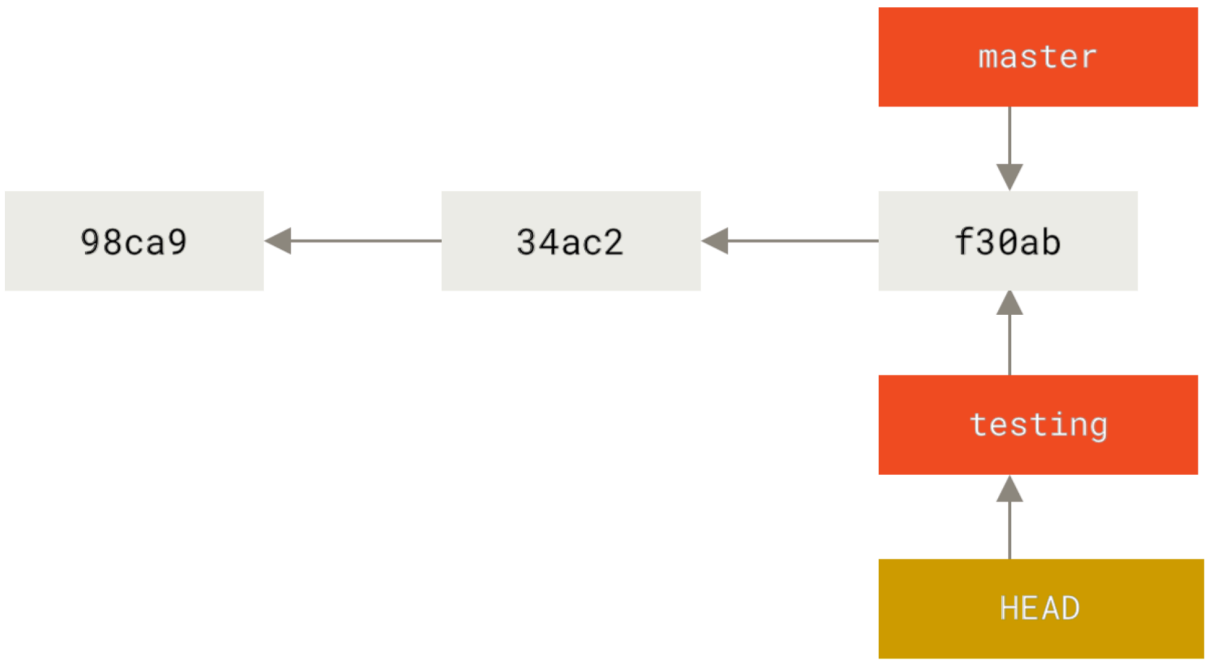
`git branch testing`



Comment ça marche

Les branches dans Git (suite)

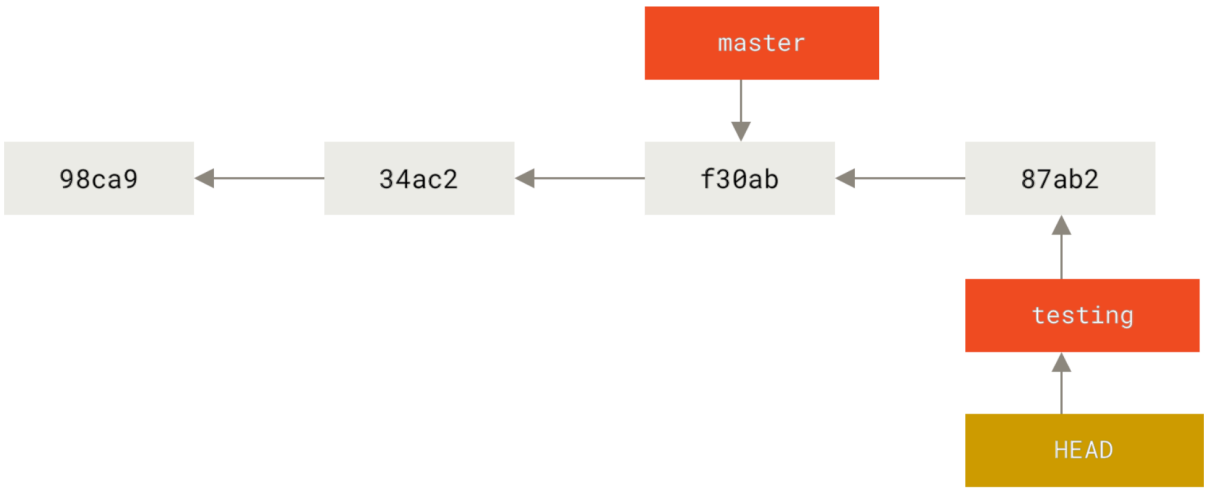
`git checkout testing`



Comment ça marche

Les branches dans Git (suite)

`git add, git commit`

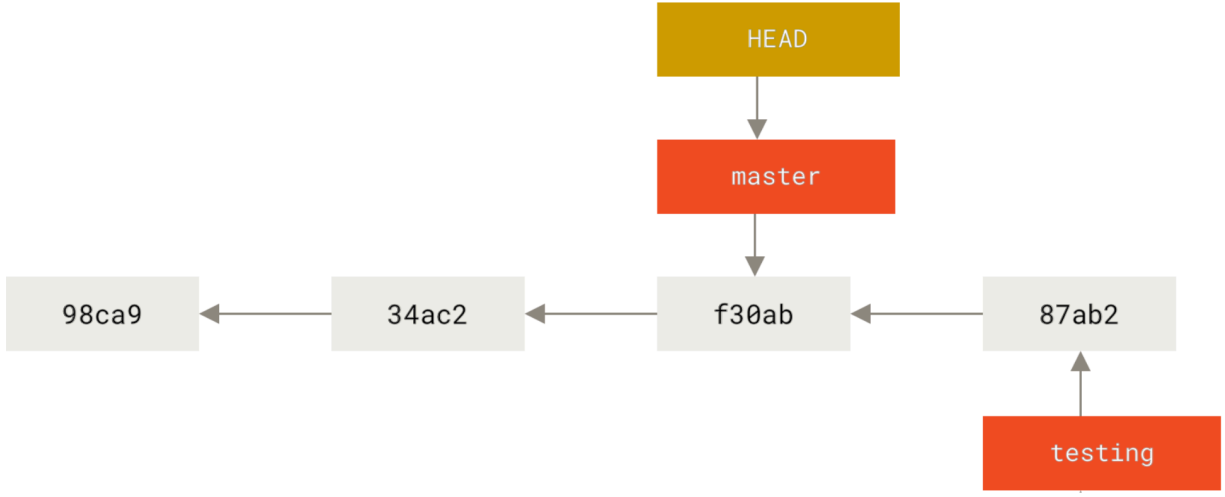


Un nouveau commit sur la nouvelle branche...

Comment ça marche

Les branches dans Git (suite)

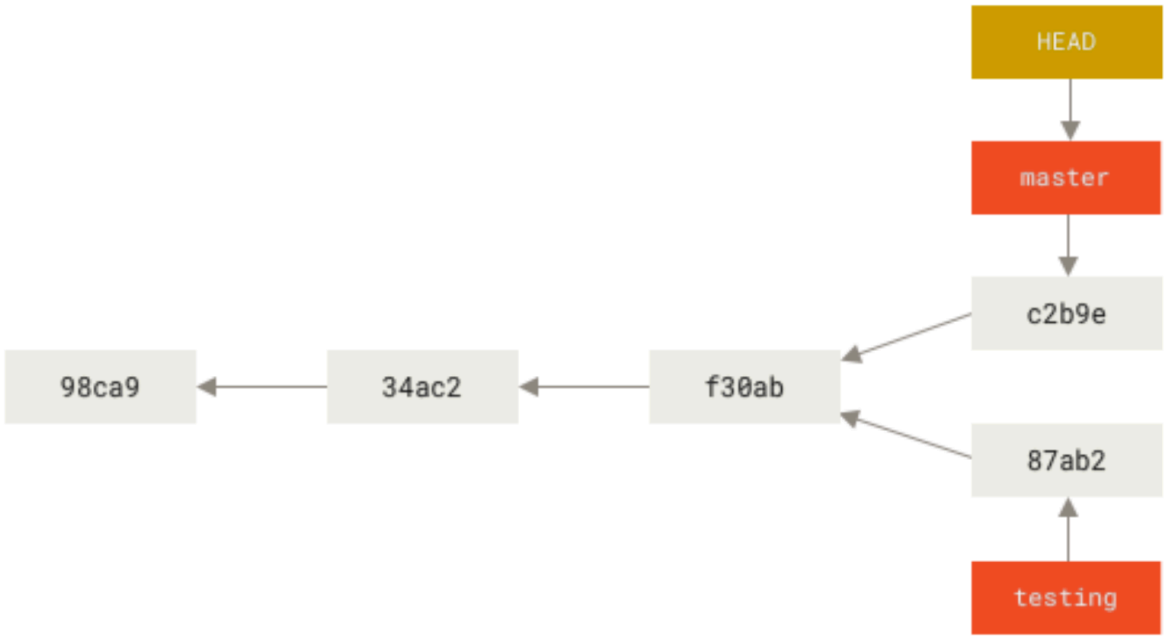
```
git checkout master
```



Revenir à la branche master...

Comment ça marche

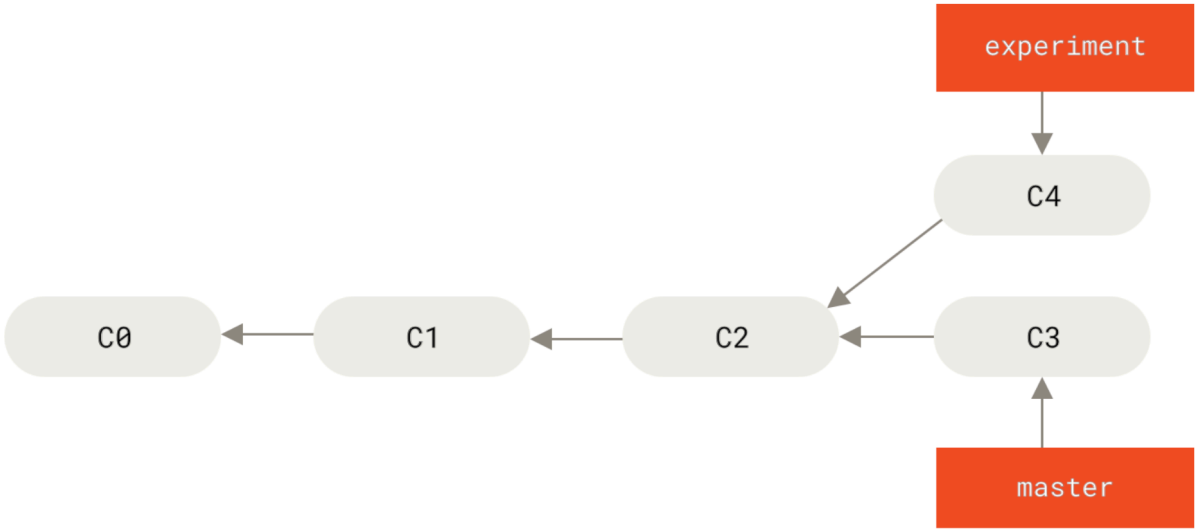
Les branches dans Git (suite)



Après un nouveau commit sur master, les branches ont **divergé**

Comment ça marche

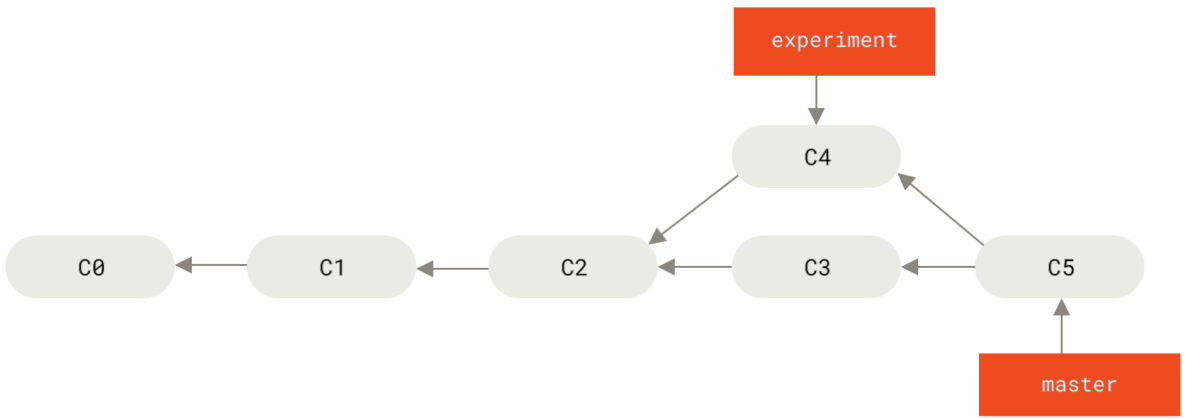
Les branches dans Git (suite)



On merge les deux branches sur master...

Comment ça marche

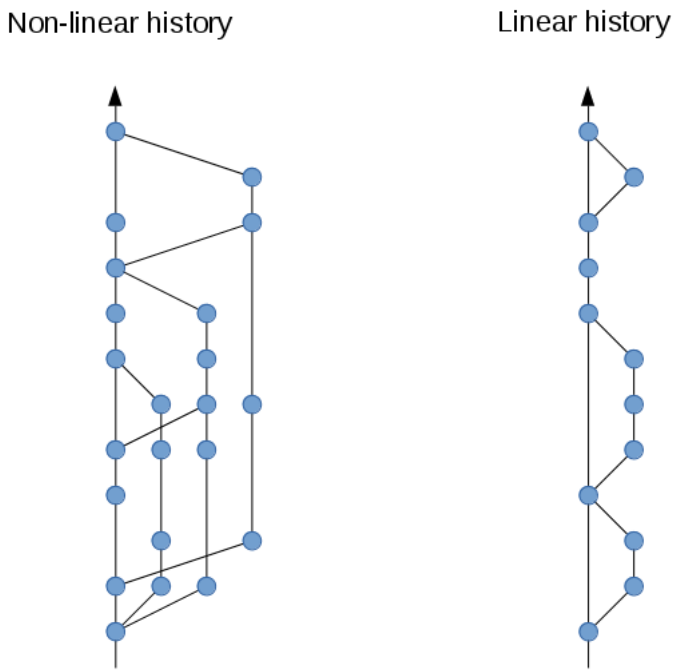
Les branches dans Git (suite)



Les deux branches sont mergées sur master...

git rebase

```
1  $ git rebase [ref]
```



Voir https://git-scm.com/book/fr/v2/Les-branches-avec-Git-Rebaser-Rebasing

git rebase

```
1   $ git rebase [options]
```

Cette commande permet de réécrire le sommet de l'historique d'une branche (source) sur une autre (cible).

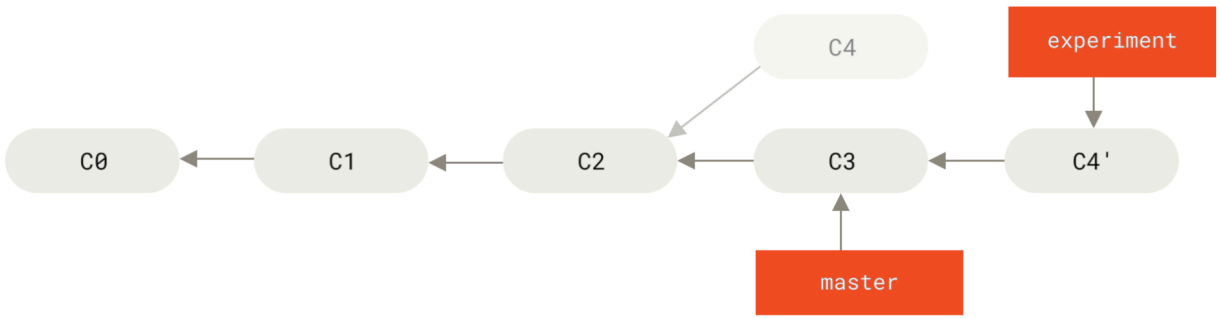
- les commits de la source inconnus de la cible [4] sont mis dans une zone d'attente
- la branche courante est déplacée vers la cible
- les commits en attente sont réappliqués par dessus, un à un, dans l'ordre (cela est fait automatiquement si possible, sinon, stratégies de fusion. Par exemple : `git rebase -i` demande **interactivement** à l'utilisateur).

Bien lire l'aide pour comprendre les options !

4. C'est-à-dire ultérieurs à leur point de divergence.

git rebase

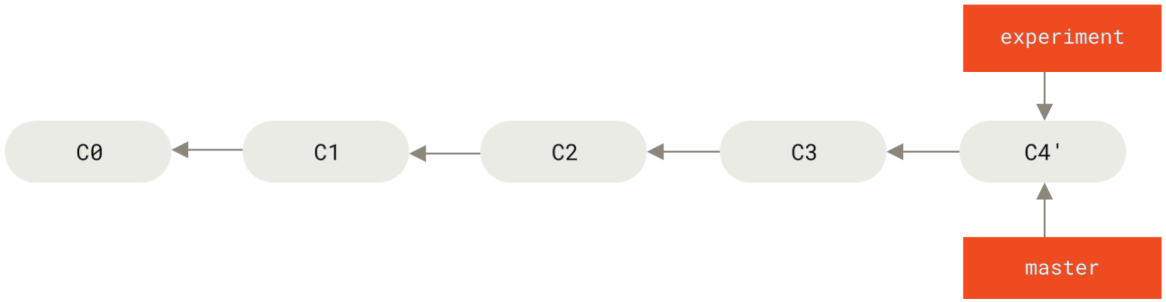
```
1  $ git rebase [ref]
```



Voir https://git-scm.com/book/fr/v2/Les-branches-avec-Git-Rebaser-Rebasing

git rebase (suite)

```
1  $ git rebase [ref]
```



Voir https://git-scm.com/book/fr/v2/Les-branches-avec-Git-Rebaser-Rebasing

`git pull --rebase`

```
1  $ git pull --rebase
```

- Par défaut, **`git pull`** = **`git fetch`**; **`git merge`**
  Résultat : historique = 2 séquences parallèles (une avec les commits du **`fetch`**, l'autre avec nos commits) qui divergent puis convergent (lors du commit de **`merge`**).
- Il faut plutôt lui préférer **`git pull --rebase`** pour éviter des historiques non linéaires.
  En effet : **`git pull --rebase`** = **`git fetch`** ; **`git rebase`**.
  Résultat : historique linéaire contenant les commits distants (de **`fetch`**), **suivis de** nos commits (remaniés).

`git reset` (vu en amphi)

Gitlab

GitLab est une application web pour la collaboration autour d'un dépôt git.

Notion de projet Gitlab

L'aspect décentralisé de Git a de nombreux avantages en termes de flexibilité et de résilience. Cependant, si plusieurs développeurs souhaitent collaborer en travaillant sur un même dépôt, il faut leur donner accès à un dépôt de référence. C'est ce problème que résout Gitlab.

En plus de cela, Gitlab fournit des outils de **gestion de projet** :

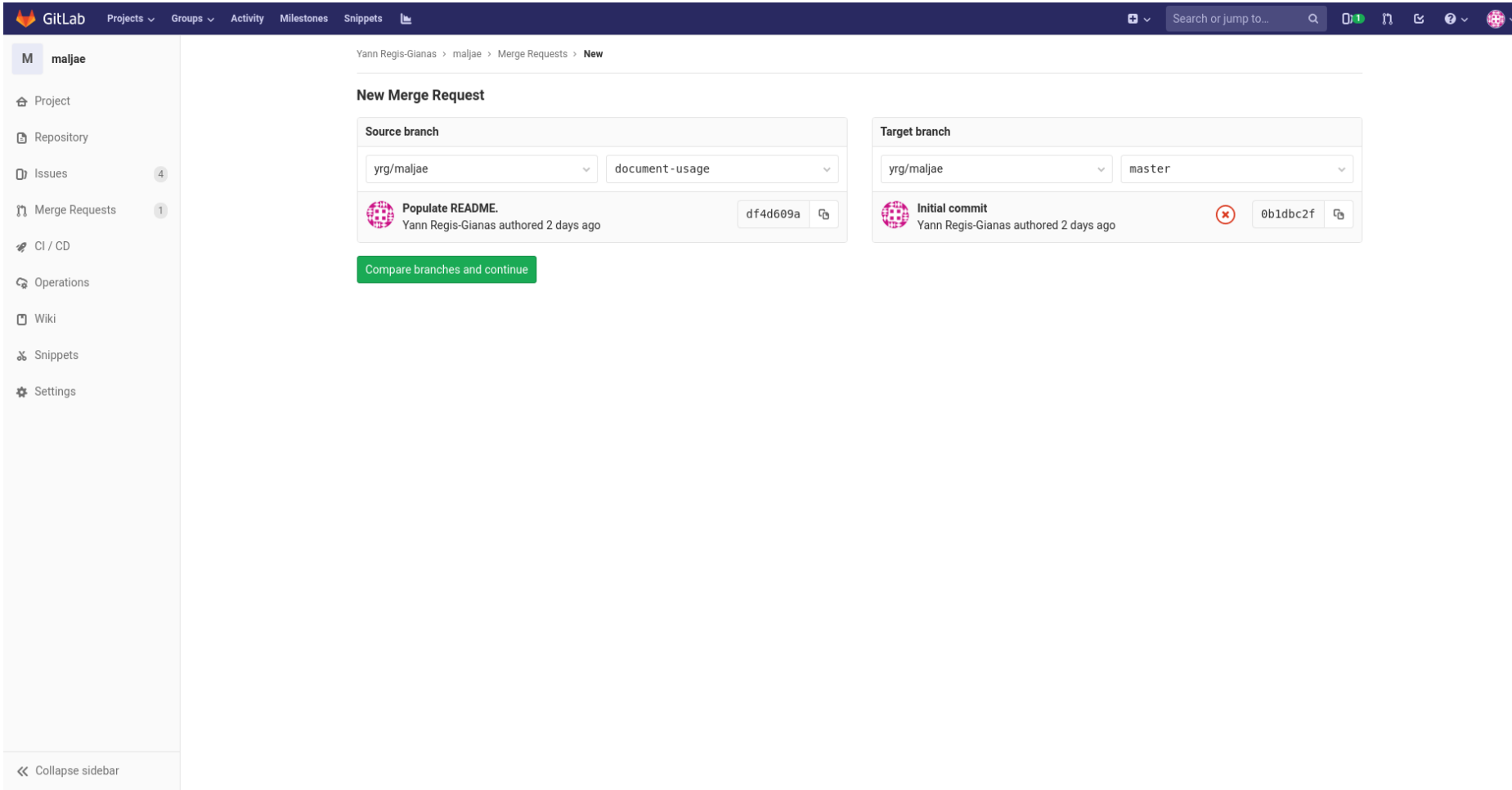
1. Contrôle d'accès sur le dépôt.
2. Système de tickets, pour documenter les bugs, les évolutions, ...
3. Système de contrôle et de revue de "merge requests" (voir transparent suivant).
4. Système d'intégration continue.
5. Système de notifications pour être alerté des événements d'un projet.

“Demandes de fusion” ou “Merge requests” (ou “Pull requests” [5])

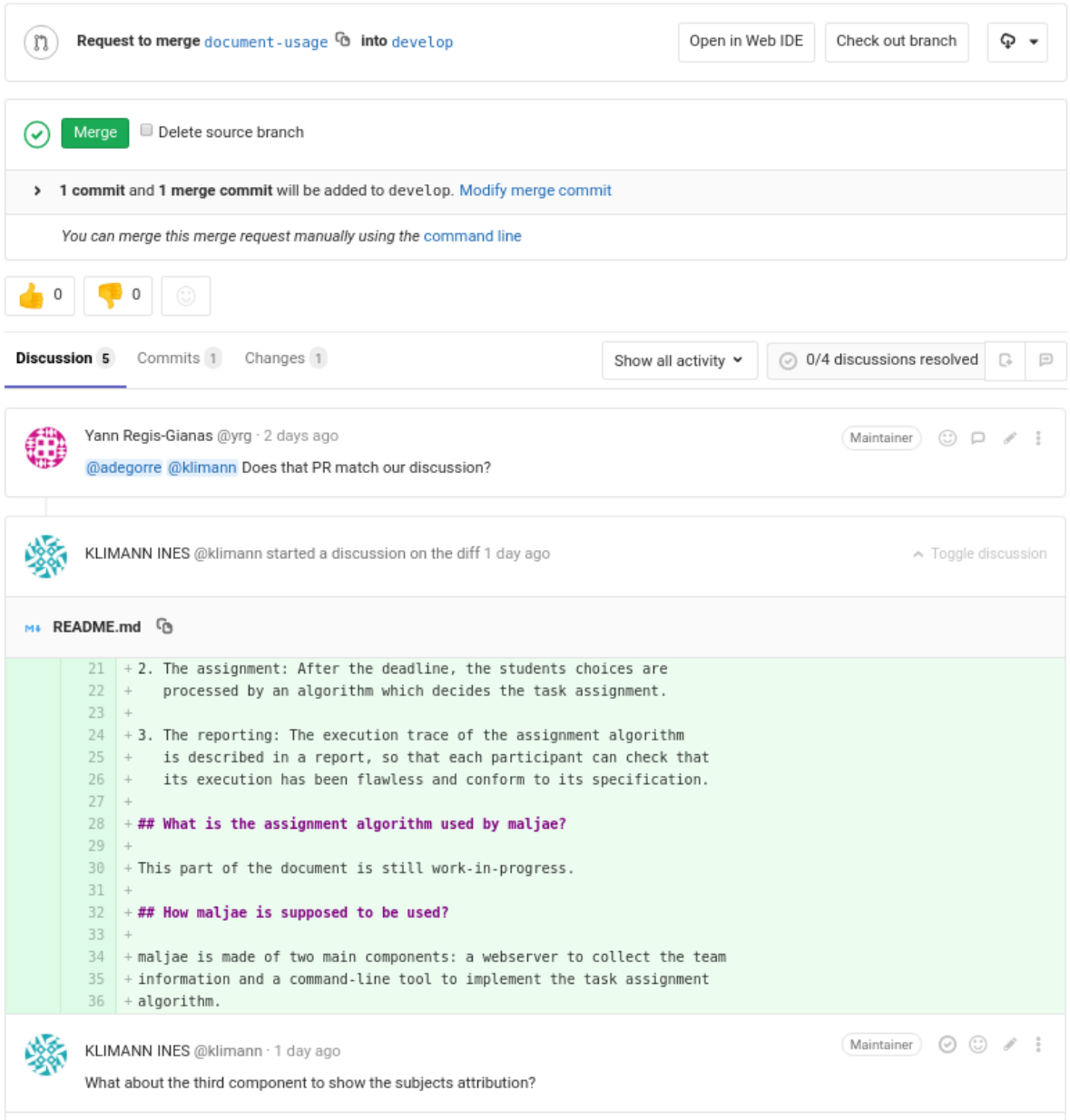
- Tout changement fait sur le code source d’un projet doit être documenté et chaque développeur doit en avoir conscience et le comprendre.
- À éviter : faire une multitude de petits commits dans son coin et les pousser directement sur le dépôt du projet (sans que personne ne contrôle la qualité ni ne comprenne l’objectif de ces changements).
- → **bonne pratique : ne jamais pousser directement des changements** sur un dépôt de référence d’un projet.
  → étape intermédiaire : soumission d’une “merge request” (MR).
- Soumettre une MR permet aux autres membres de l’équipe de **relire** les changements pour les comprendre et faire des suggestions d’améliorations.
  → C’est seulement lorsque la MR a été validée par suffisamment de membres de l’équipe qu’elle est effectivement intégrée au dépôt.

5. “Pull request” est la terminologie du site GitHub, concurrent de GitLab. Elle est très couramment uilisée. Ainsi on dira souvent PR au lieu de MR sans y faire attention...

MR dans GitLab



MR dans GitLab



Fork de projet Gitlab

Gitlab favorise le développement collaboratif en autorisant n'importe qui à créer une copie [6] d'un projet. Cette opération s'appelle un fork dans le jargon de Gitlab ("divergence", dans l'interface en français).

Un fork permet à toute personne extérieure aux membres d'un projet Gitlab de proposer des contributions. Pour cela, il lui suffit, après avoir créé de fork et y avoir ajouté sa contribution, de créer une MR pour la proposer aux membres du projet d'origine ("upstream project") [7].

6. Je n'utilise volontairement pas le mot "clone", que l'on réservera à l'opération consistant à copier un dépôt git. Quand on fait un fork, le dépôt du nouveau projet est, certes, cloné depuis le projet d'origine, mais on crée aussi un projet complet, pas seulement un dépôt.
7. Accepter la fusion consiste alors, pour le projet upsteam à faire un "pull" du dépôt de son fork, d'où la terminologie "pull request" utilisée sur GitHub.

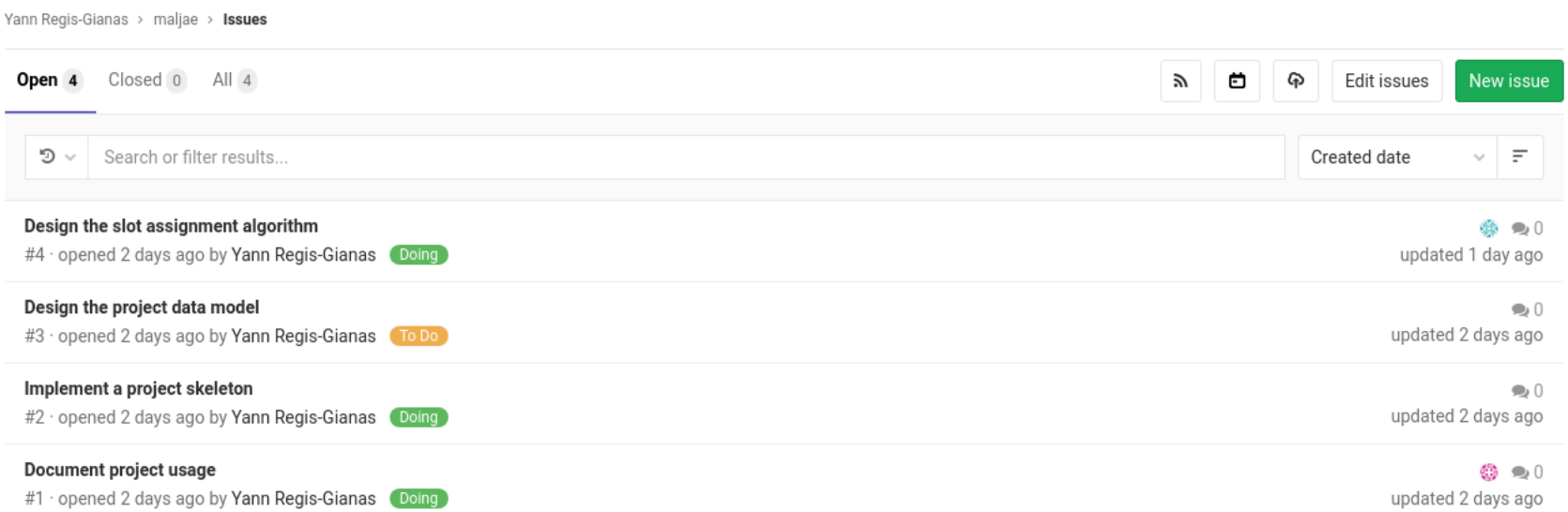
Gestion de tickets avec Gitlab

Gitlab permet de créer des **tickets**. Un ticket documente un problème à résoudre sur le projet : cela peut être une erreur à corriger dans le code, une fonctionnalité à rajouter, ...

Les tickets peuvent **être référencés** par les MR, dans les discussions ou dans les outils de gestion de projet. Cela améliore la traçabilité du projet.

Les tickets peuvent être **ouverts** (à résoudre) ou **fermés** (résolus).

Les tickets peuvent aussi être catégorisés par des étiquettes.



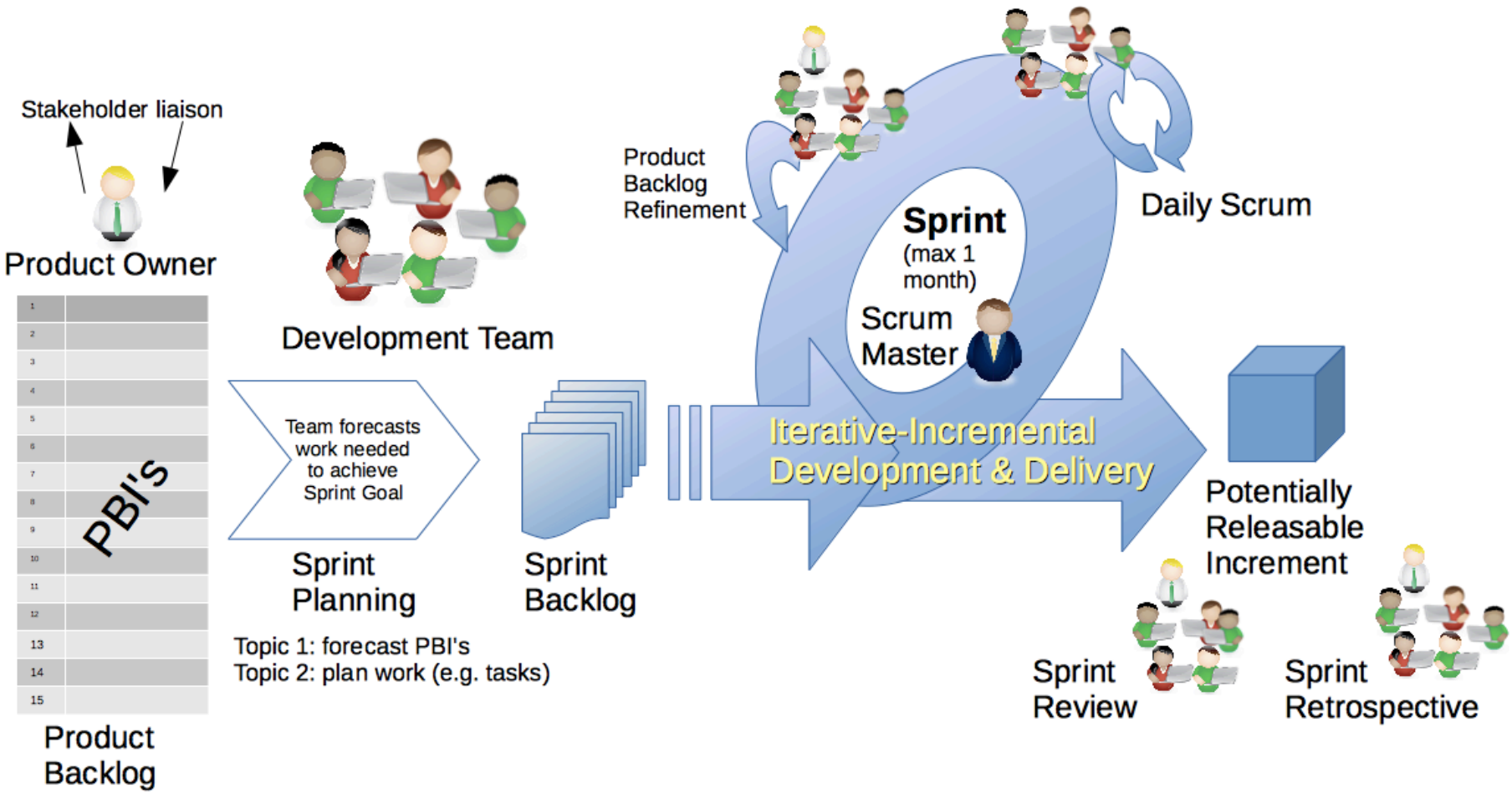
Les méthodes Agile

- Méthodes Agile = processus de développement de logiciel.
- Retour sur les principes des méthodes de cette famille.
- Scrum est l'une de ces méthodes. Remarque : ces méthodes ne se consacrent pas forcément aux mêmes aspects du cycle de développement et peuvent parfois se compléter.

Scrum

- Processus incrémental et itératif.
- Organisés autour de **sprints**.
- Chaque sprint est :
  - limité dans le temps (de 2 à 4 semaines) ;
  - décomposé en une phase de **planification**, une phase de **développement**, une **revue** et une **rétrospective** ;
- focalisé sur un **ensemble fixé d'objectifs** mais **intègre le raffinement** de ces objectifs (pour un sprint ultérieur).
- mené par une équipe **pluridisciplinaire** qui va **chercher le travail où il est** ;
- engendre un **logiciel fonctionnel** (intégré, testé, documenté et déployable).

Scrum


Stakeholder liaison
Product Owner
PBI's
Product Backlog
Development Team
Team forecasts work needed to achieve Sprint Goal
Sprint Planning
Topic 1: forecast PBI's
Topic 2: plan work (e.g. tasks)
Sprint Backlog
Product Backlog Refinement
Sprint
(max 1 month)
Scrum Master
Daily Scrum
Iterative-Incremental Development & Delivery
Potentially Releasable Increment
Sprint Review
Sprint Retrospective

Utilisation d'un kanban

- Voir le module "board" de Gitlab.

Daily scrum

- **Daily scrum** : Réunion courte de 15 minutes maximum.
- Objectif : un point d'information par chaque membre, pour l'équipe.
- Chaque membre répond à :
  1. Qu'ai-je fait depuis la dernière réunion ?
  2. Que vais-je faire jusqu'à la prochaine ?
  3. Quels sont mes difficultés ?
- C'est un point d'information, pas de discussion.
- On peut organiser des réunions en petit groupe pour résoudre un problème mais sans bloquer l'équipe entière.

Revue de Sprint

- Réunion de l'équipe pour faire le tour de ce qui a été intégré lors du sprint. (Par exemple, en regardant les MRs intégrées.)
- Lors de cette réunion, les membres de l'équipe :
  - prennent connaissance du travail effectué ;
  - doivent évaluer la qualité de ce travail.
- On peut alors décider une mise en production / release du logiciel.

Rétrospective de Sprint

- Prise de recul de l'équipe sur elle-même :
  - Qu'est-ce qui a posé problème pour son bon fonctionnement ?
  - Qu'est-ce qui a bien fonctionné ?

Planification du prochain Sprint

- L'équipe met à jour le "product backlog".
- Le sommet de ce dernier fournit en général le "sprint backlog".

Suggestions pour de thèmes pour les sprints à venir

Sprint 1	Refactoring. (Changer la façon dont est structuré et écrite le code, sans en changer le comportement.)
Sprint 2	Corrections de bugs.
Sprint 3	Implémenter la fonctionnalité principale manquante : génération de page HTML avec graphiques et tester rigoureusement le système.
Sprint 4	Implémenter des fonctionnalités ”co-définies” avec le “product owner”.
Sprint 5	Implémenter des fonctionnalités que vous jugez importantes.

Conclusion

- Vous devez avoir lu le livre “The Scrum Primer”.
- Bon sprints à vous !
- Si vous souhaitez approfondir :
  - The Pragmatic Programmer : From Journeyman to Master – Andrew Hunt
  - Agile Project Management with Scrum – Ken Schwaber
  - Les pratiques de l’équipe agile – Thomas Thiry