ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR
Membre de HONORIS UNITED UNIVERSITIES

DevOps : Développement et Opération

Prof. Samya Bouhaddour

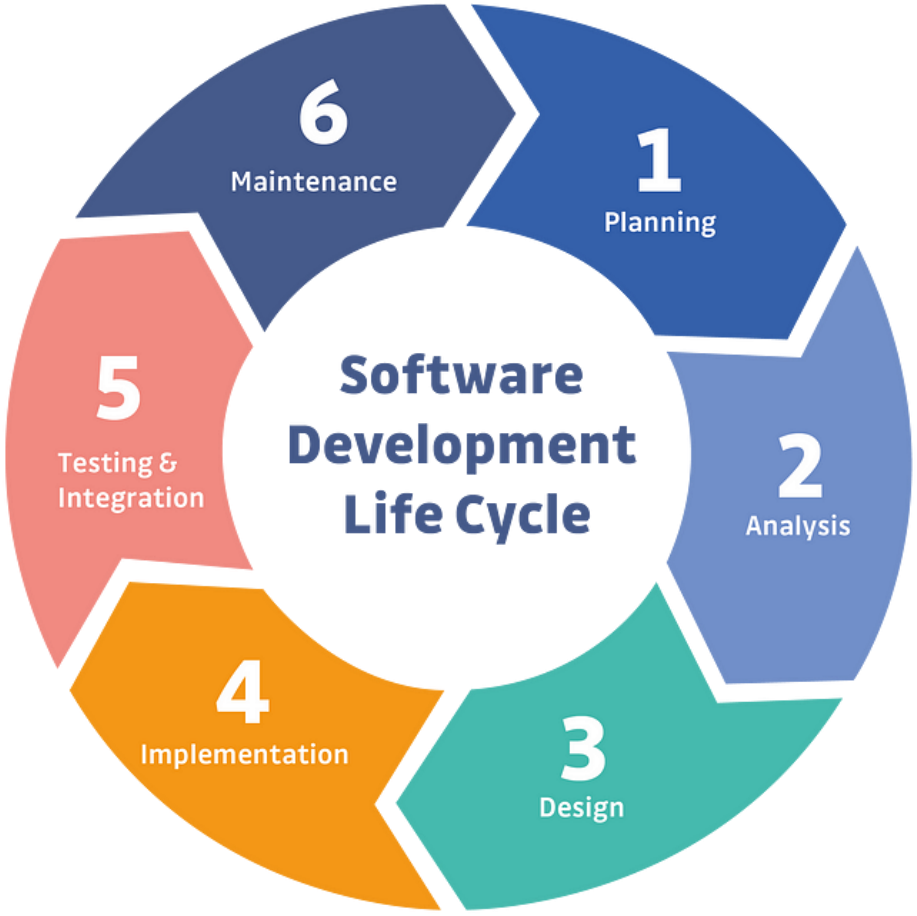
A.U. : 2023-2024

PLAN

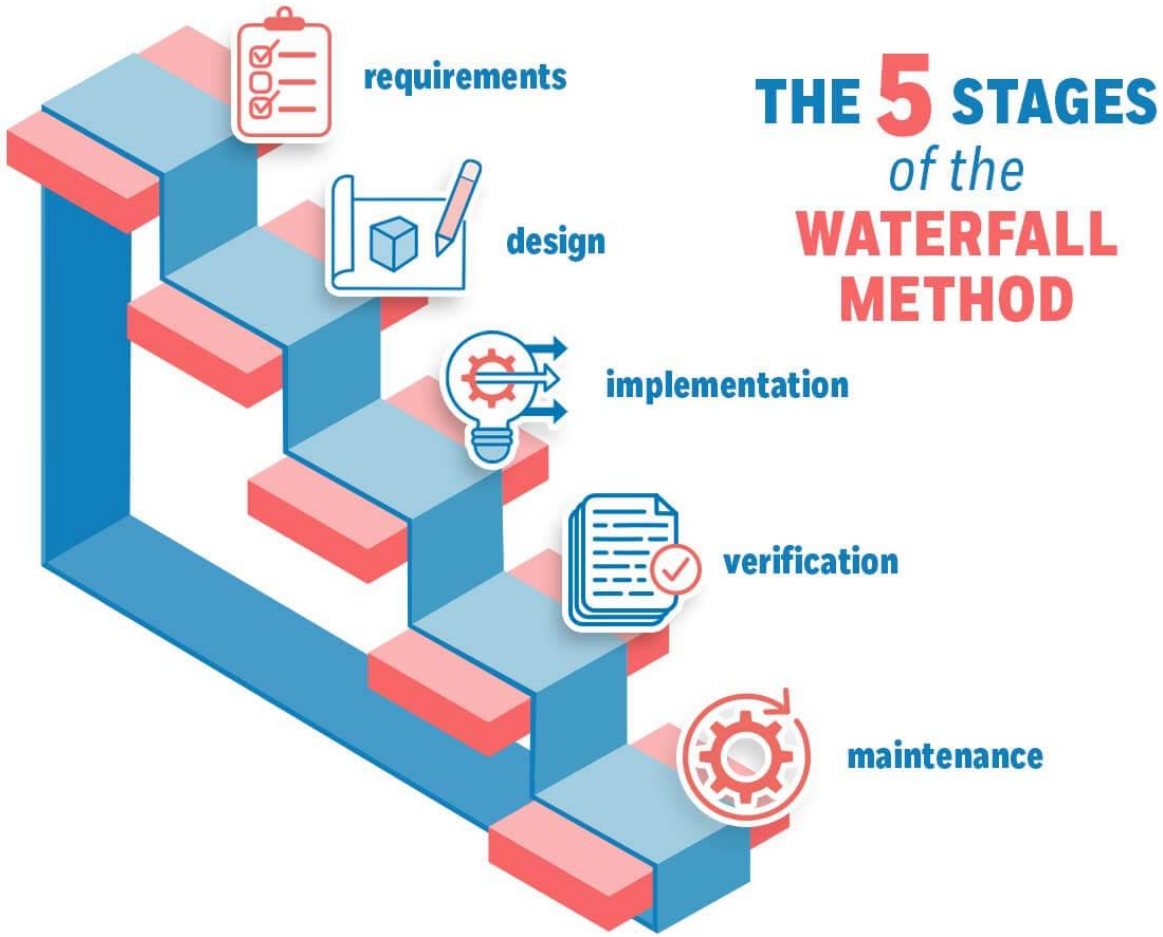
- Introduction au concept DevOps
  - Introduction à DevOps
  - Introduction à la plateforme Azure DevOps
- Contrôleur de code source GIT
  - Définition d'un contrôleur de code source
  - Application de commandes de base GIT
  - Partage d'un répertoire distanciel
  - Gestion des branches
  - Fast-Forward / No-Fast-Forward
  - Rebase / Rebase Interactif
- Planification
  - Utilisation d'Azure Board
- Outil de conteneurisation 'Docker '
  - Le concept de la virtualisation
  - Définition d'outil d'orchestration des conteneurs
  - Dockerfile
  - Docker-image
  - Conteneur Docker
  - Crétaion des images
  - Docker-Compose
  - Kubernetes
- Gestion de pipeline

Introduction à DevOps

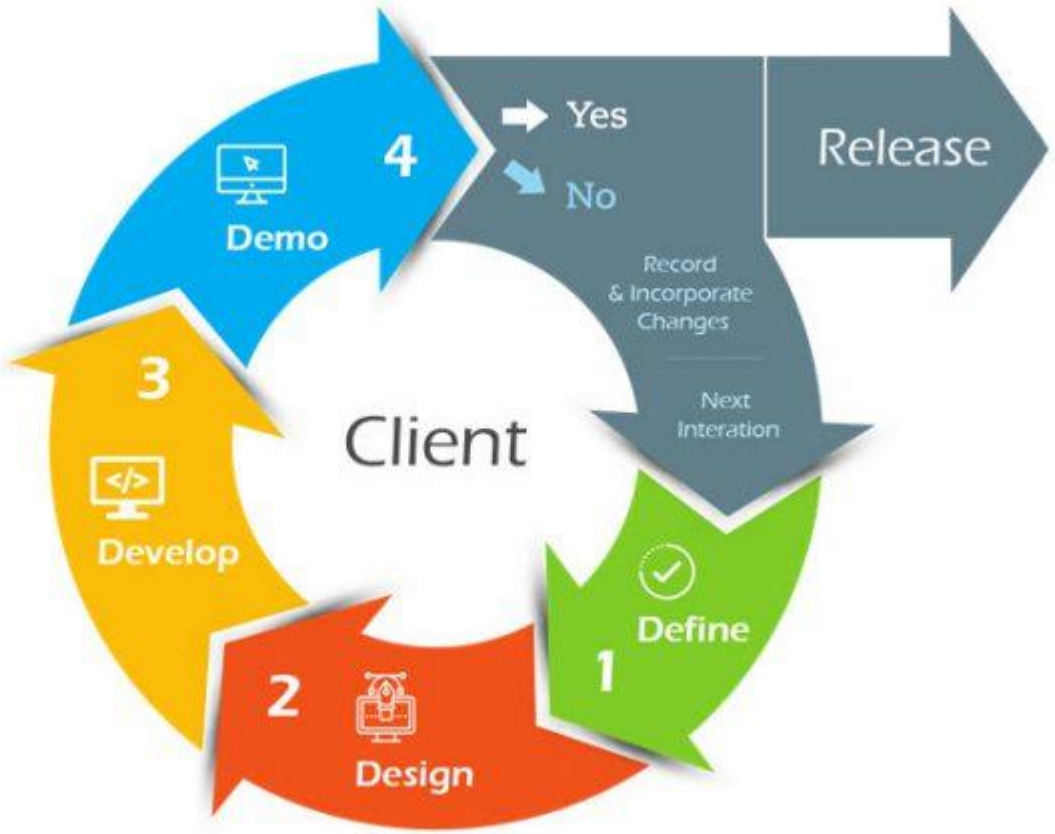
SDLC : Software Development Lice Cycle



WATERFULL : MODÈLE EN CASCADE



AGILE


4
Demo
Yes
No
Release
Record
& Incorporate
Changes
Next
Interation
3
Develop
Client
Define
1
2
Design

Le terme génie logiciel désigne l'ensemble des **méthodes***, des **techniques*** et **outils*** concourant à la production d'un logiciel, au delà de la seule activité de programmation.

Ensemble des méthodes, des techniques et outils * : Le génie logiciel ne se limite pas à une seule approche ou méthode. Il englobe un large éventail de méthodes, de techniques et d'outils utilisés pour développer un logiciel de manière efficace et de haute qualité. Cela comprend la planification, la conception, la construction, les tests, la maintenance et la gestion de projet.

LES DÉFIS

La taille des projets: pour certains, des millions de ligne de code (MLOC)

Contribution à des projets existants

- Peu de projets démarrés from scratch
- Plus de valeur d'ajouter 100 LOCs à un grand projet largement utilisé que d'écrire 10000 LOCs dans son coin. *Exemples de ces projets : systèmes d'exploitation, bases de données massives, ERP, plateformes de médias sociaux, etc.*
- La taille des projets impacte divers aspects : complexité du développement, ressources requises, planification, gestion du code et des versions, performances et maintenance.
- Gérer efficacement de grands projets exige une planification minutieuse, une coordination d'équipe efficace, ainsi que des outils et des processus appropriés.

LES DÉFIS

Réutilisation de code existant

Est ce que le concepteur d'une voiture commence par réinventer la roue?

- Pratique courante et bénéfique dans le développement logiciel.
- Évite de réinventer la roue à chaque projet.
- Recherche et utilisation de bibliothèques, frameworks et modules existants.
- Permet d'économiser du temps et des ressources.
- Favorise la cohérence, la fiabilité et la qualité du logiciel.
- Utilisation de composants éprouvés et souvent maintenus par une communauté plus large.

LES DÉFIS

Collaboration avec d’autres développeurs

Comment interagir?

- Utilisation d'outils de gestion de versions comme Git.
- Plateformes de gestion de projets telles que GitHub ou GitLab.
- Communication via des forums de discussion et des réunions virtuelles.

Comment fournir du code réutilisable?

- Assurer une documentation claire et complète du code.
- Adopter une conception modulaire et bien encapsulée.
- Publier le code sur des plateformes accessibles aux autres développeurs.
- Encourager une communication ouverte sur les normes de codage et les bonnes pratiques.

LES DÉFIS

Utilisateurs/Clients :

- Compréhension des besoins et des attentes des utilisateurs.
- Collecte de retours d'expérience et de feedbacks.
- Validation des fonctionnalités du logiciel en collaboration avec les utilisateurs.
- Développement de produits offrant une valeur réelle et une expérience utilisateur positive.

LES ENJEUX

L’industrie du logiciel, c’est 5% de projets “from scratch” et 95% de projets existants.

Le travail consiste alors à:

- ✓ **Réutiliser :** *Utiliser des composants existants pour éviter de recréer des fonctionnalités déjà développées.*
- ✓ **Faire évoluer :** *Ajouter de nouvelles fonctionnalités ou améliorer les existantes pour répondre aux besoins changeants.*
- ✓ **Etendre :** *Augmenter les capacités ou la portée d'un système pour inclure de nouveaux cas d'utilisation.*
- ✓ **Adapter :** *Modifier un logiciel pour qu'il fonctionne dans de nouveaux environnements ou avec de nouvelles technologies.*
- ✓ **Maintenir :** *Assurer le bon fonctionnement du logiciel en appliquant des correctifs et en fournissant un support continu.*
- ✓ **Réorganiser :** *Restructurer le code existant pour améliorer sa qualité et sa facilité de maintenance.*

DEVOPS

Définition simple :

Ensemble de techniques et d'outils facilitant le passage du développement à la production.

Bien plus que ça:

- Modèle de fonctionnement de l'entreprise
- Modèle d'interactions entre les équipes
- Intégration du retour sur expérience
- Une "culture"

Nous en resterons à la définition simple !

DEVOPS

Relation entre Dev et Ops

Dev: Equipes de développeurs logiciels
Ops: Equipes en charge de la mise en production des produits

Antagonisme fort

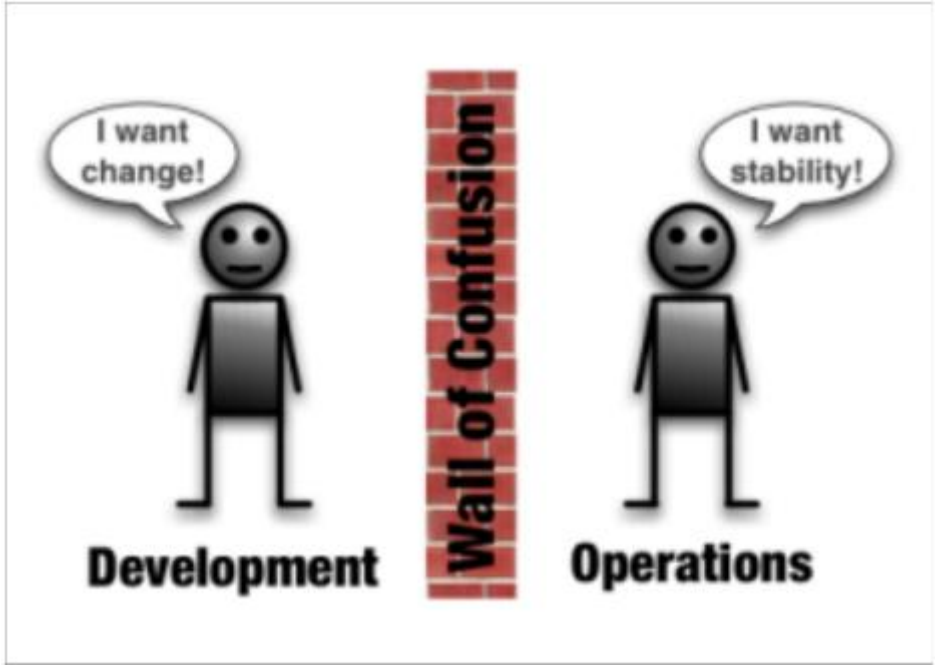
Dev: Modifications aux moindres coûts, le plus rapidement possible
Ops: Stabilité du système, qualité

L'automatisation est au cœur de l'approche DevOps !

Le DevOps est un **ensemble de pratiques** qui met l'emphase sur **l'automatisation des processus** entre les équipes de développement, et les équipes en charge du maintien en conditions opérationnelles de l'application développée.

- gain de confiance des équipes entre elles ;
- accélération des livraisons et des déploiements ;
- résolution des tickets plus rapide ;
- gestion plus efficace des tâches non planifiées…

Méthodes agiles



DevOps : Automatisation

Intégration continu

- Intégration automatique des modifications du code.
- Reconstruire et tester le logiciel à chaque modification.
- Détection rapide des problèmes.
- Amélioration de la qualité du code.
- Réduction des conflits d'intégration

Une méthode de développement logiciel dans laquelle le logiciel est reconstruit et testé à chaque modification apportée par un programmeur

Livraison continue

- Déploiement rapide et fiable des modifications validées.
- Livraison sans délai excessif ni risque.
- Réactivité accrue aux besoins des utilisateurs.
- Amélioration continue de l'application.
- Réduction des délais entre développement et mise en production

Une approche dans laquelle l'intégration continue associée à des techniques de déploiement automatiques assurent une mise en production rapide et fiable du logiciel.

Déploiement continue

- Automatisation totale du processus de mise en production.
- Mise à jour en temps réel de l'application.
- Pas de temps d'arrêt ni de risque d'erreur humaine.
- Agilité maximale dans les mises à jour.
- Capacité à répondre rapidement aux évolutions du marché

Une approche dans laquelle chaque modification apportée par un programmeur passe automatiquement toute la chaîne allant des tests à la mise en production. Il n'y a plus d'intervention humaine

L'intégration continue

L'intégration continue se réfère à plusieurs pratiques :

Construire une version fonctionnelle chaque jour :

Garantit la disponibilité d'une version testable du logiciel quotidiennement.

Assure une vision claire de l'état du prototype.

Exécuter les tests quotidiennement :

Permet de détecter rapidement les erreurs et les problèmes de compatibilité.

Réduit les risques d'introduire des défauts majeurs.

Commiter les changements quotidiennement :

Favorise une approche incrémentale du développement.

Facilite la collaboration et la gestion des versions.

Système d'observation des changements :

Automatise la vérification des modifications dans le dépôt.

Réalise des actions en réponse aux changements détectés.

La livraison continue

Les étapes principales d'une procédure de livraison continue sont:

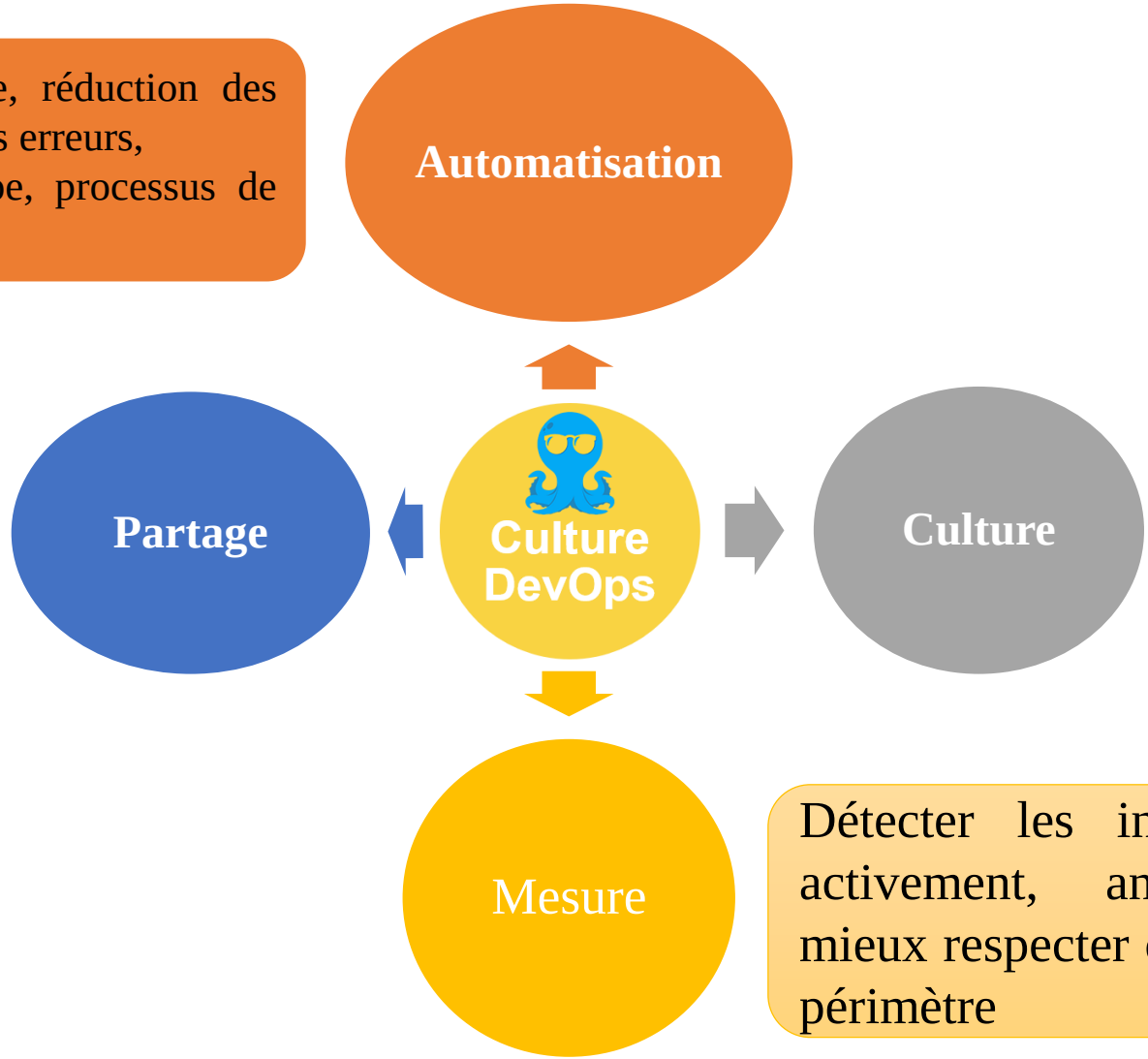
1. **Commit d'une modification**
2. **Exécution des tests unitaires (Intégration continue)**
3. **Tests de validation fonctionnelle (acceptance test)**
   - Tests black-box
   - Testent si le logiciel répond bien au besoin du client
   - Peuvent aussi être automatisés
4. **Tests de performances**
   - Testent si le logiciel peut répondre à la charge
   - Performance et passage à l'échelle
5. **Tests exploratoires**

Tests non-automatisés effectués par des experts

6. **Le déploiement en production**

Définition

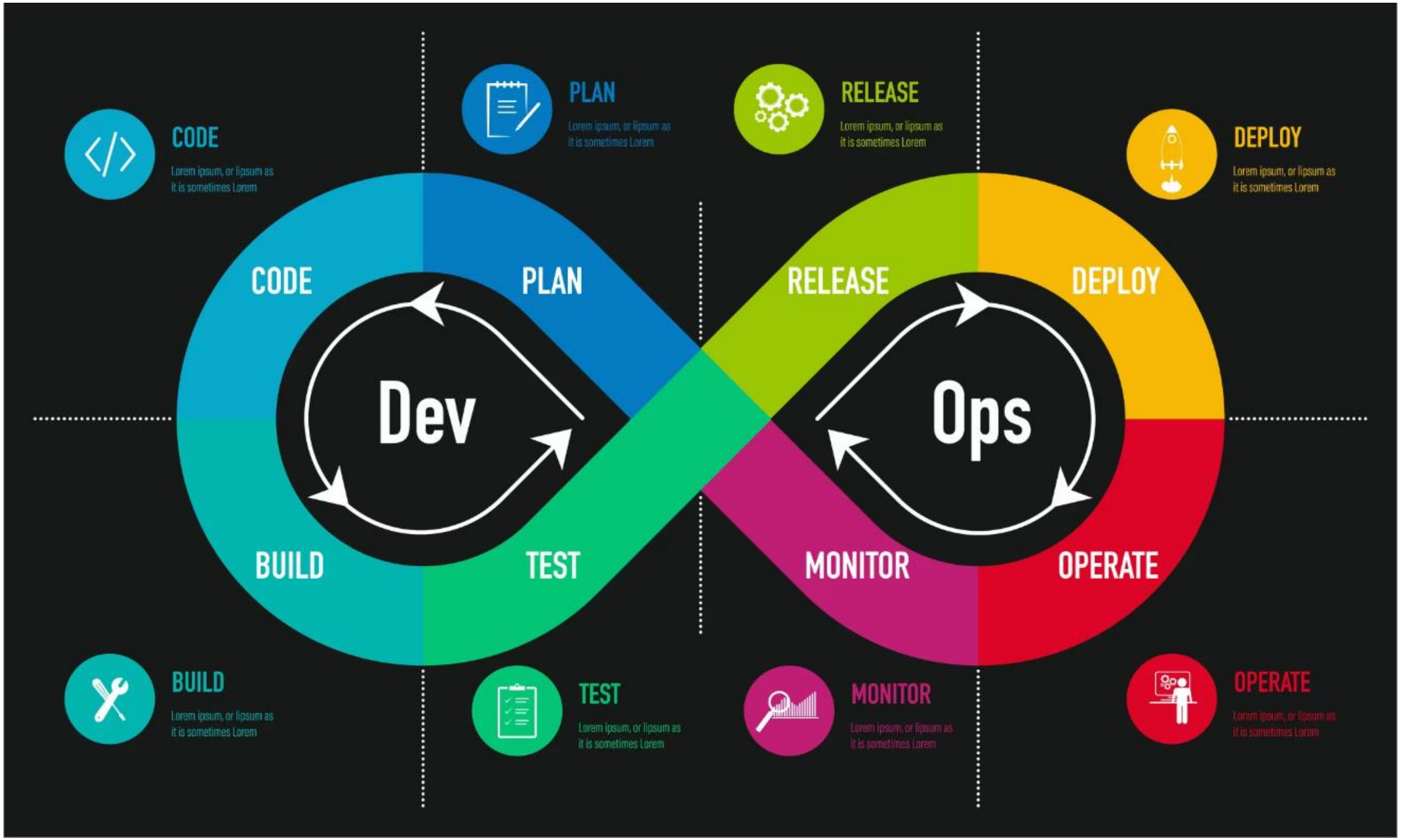
Diminution du temps de cycle, réduction des coûts de livraisons, réduction des erreurs, diminution du stress des équipe, processus de livraison fiable et rapide



Etre tous orientés vers une même cible, cible partagée entre business , développement et exploitation. Développer le «tous ensemble»

Favoriser le mode projet, l'agilité, la créativité et la mise en avant des compétences de chaque membres des équipes

Détecter les incidents pro activement, anticiper pour mieux respecter coûts, délais, périmètre


CODE
Lorem ipsum, or lipsum as it is sometimes Lorem
PLAN
Lorem ipsum, or lipsum as it is sometimes Lorem
RELEASE
Lorem ipsum, or lipsum as it is sometimes Lorem
DEPLOY
Lorem ipsum, or lipsum as it is sometimes Lorem
CODE
PLAN
RELEASE
DEPLOY
Dev
Ops
BUILD
TEST
MONITOR
OPERATE
BUILD
Lorem ipsum, or lipsum as it is sometimes Lorem
TEST
Lorem ipsum, or lipsum as it is sometimes Lorem
MONITOR
Lorem ipsum, or lipsum as it is sometimes Lorem
OPERATE
Lorem ipsum, or lipsum as it is sometimes Lorem

Avantages à adopter l’approche DevOps

- Accélération et amélioration de la fourniture des produits
- Résolution plus rapide des problèmes et complexité réduite
- Stabilité accrue des environnements d'exploitation
- Meilleure utilisation des ressources
- Automatisation accrue
- Meilleure visibilité sur les résultats du système
- Innovation renforcée

Introduction à AZURE DevOps

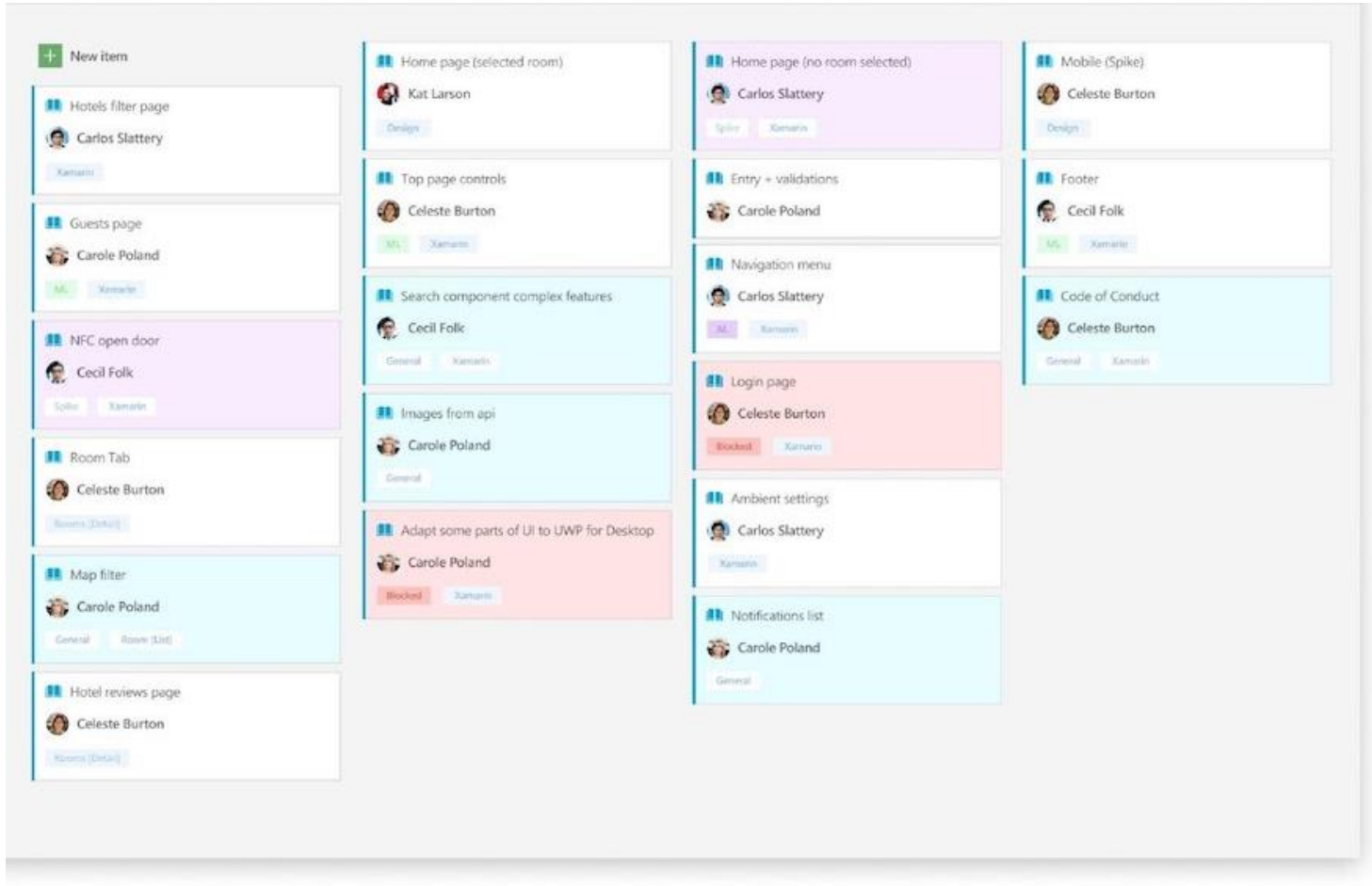
Définition

- Azure DevOps fournit des services de développement pour aider les équipes à planifier le travail, à collaborer au développement de code et à créer et déployer des applications.
- Azure DevOps fournit des fonctionnalités intégrées auxquelles vous pouvez accéder via votre navigateur Web (azure.microsoft.com) ou votre client IDE (VS).

- **Azure Repos :** Pour le partage du code source.
- **Azure Boards:** Outils de la planification.
- **Azure Pipelines :** Workflow de build et déploiement.
- **Azure Test Plans :** Automatisation et exécution des tests.
- **Azure Artifacts:** Création de package pour déploiement.

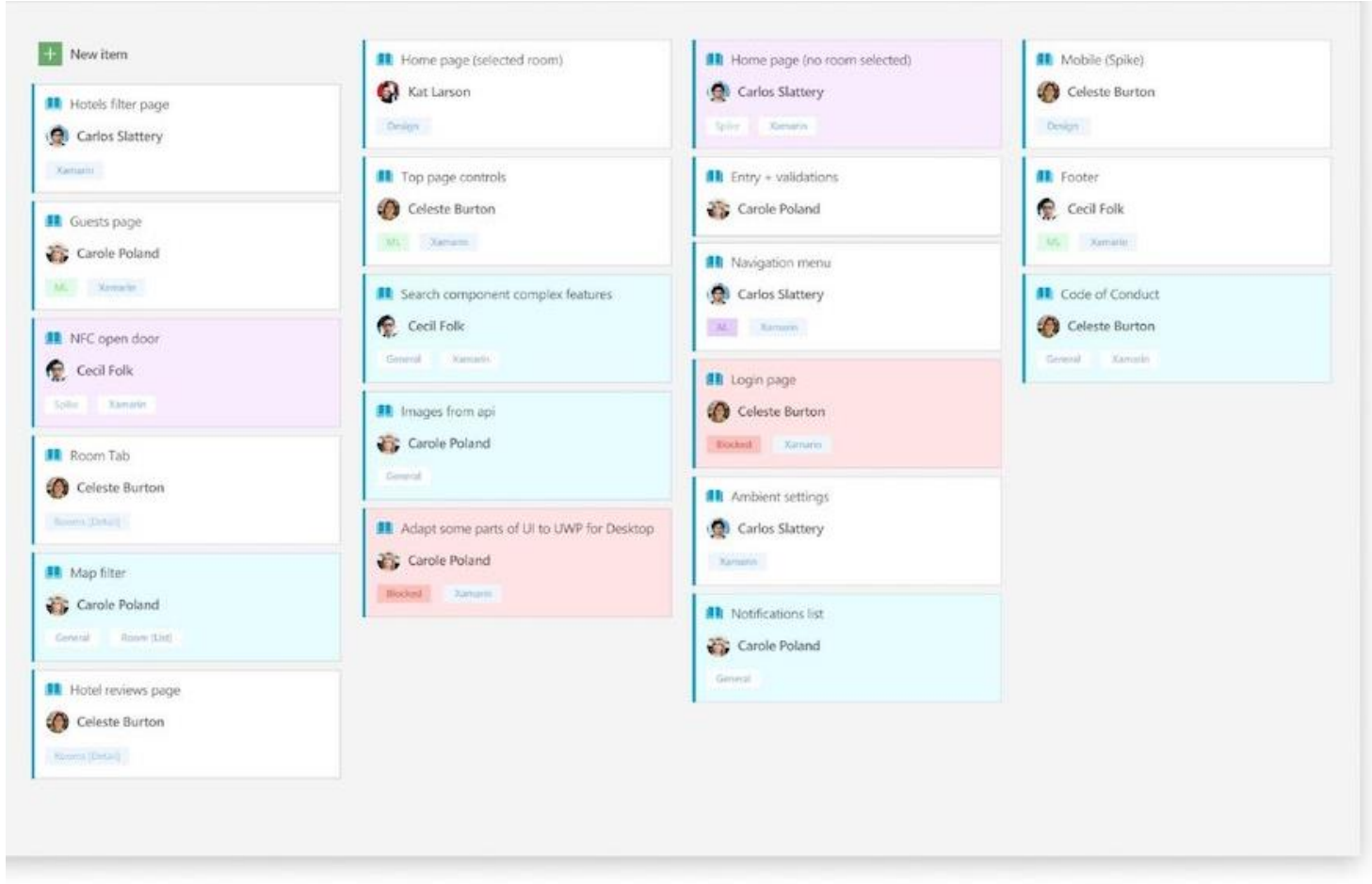
Azure Boards

Suivez le travail effectué avec des tableaux Kanban configurables, des backlogs interactifs et des outils de planification puissants.
Grâce à une traçabilité et à des rapports inégalés, Boards est la solution idéale pour toutes vos idées, petites ou grandes.



Azure Repos

Suivez le travail effectué avec des tableaux Kanban configurables, des backlogs interactifs et des outils de planification puissants.
Grâce à une traçabilité et à des rapports inégalés, Boards est la solution idéale pour toutes vos idées, petites ou grandes.



Contrôleur de code source

LES DÉFIS

Comment gérez-vous actuellement un projet ?

- L'envoyer à travers un message sur Facebook, ... (Très mauvaise idée)
- L'envoyer par mail (Un peu moins)
- Utiliser une Dropbox, Google Drive, ... (Déjà mieux mais toujours risqué ou manque de fonctionnalités)

SOLUTION

UTILISER UN SYSTÈME DE GESTION DE VERSION DÉCENTRALISÉ


geek & poke
YESTERDAY IT WORKED
YOU KNOW YOU'RE IN A SOFTWARE PROJECT

DÉFINITION

- Source contrôle est la pratique du suivi et de la gestion des modifications du code
- Les sources contrôles fournissent un historique de développement du code et aident à résoudre les conflits lors de la fusion de contributions provenant de plusieurs sources.
- Le contrôle des sources protège le code source des dégradations occasionnelles suite aux erreurs humaines.
- Les avantages incluent: la réutilisabilité, la traçabilité, portabilité, l'efficacité, la collaboration et l'apprentissage.

CE QUE PERMET DE FAIRE UN SYSTÈME DE CONTRÔLE DU CODE SOURCE

- Conserver tout le code source
- Prendre note de tous les changements effectués au code source et à sa documentation

  => permet de retourner à une version antérieure (qui, elle, fonctionnait ! !)
- Identifier quels fichiers ont été modifiés
- Déterminer qui a modifié un bout de code
- Comparer des versions
- Fusionner des versions développées de façon concurrente
- Identifier et gérer les releases*, les versions**, les branches de développement ***

Release* : des versions stables et officielles du logiciel, généralement destinées à être utilisées par les utilisateurs finaux

Versions** : Les versions correspondent à différentes itérations du logiciel, avec des fonctionnalités ajoutées, des bogues corrigés, etc.

Branches de développement ***: sont des versions du code source qui évoluent séparément les unes des autres.

Par exemple, il peut y avoir une branche de développement pour travailler sur de nouvelles fonctionnalités tandis qu'une autre branche est dédiée à la correction de bugs. Un système de contrôle de code source permet de gérer ces différentes entités de manière organisée, en les identifiant clairement et en facilitant leur gestion.

LES DIFFÉRENTS SCCS : SCCS CENTRALISÉ VS. SCCS DISTRIBUÉ

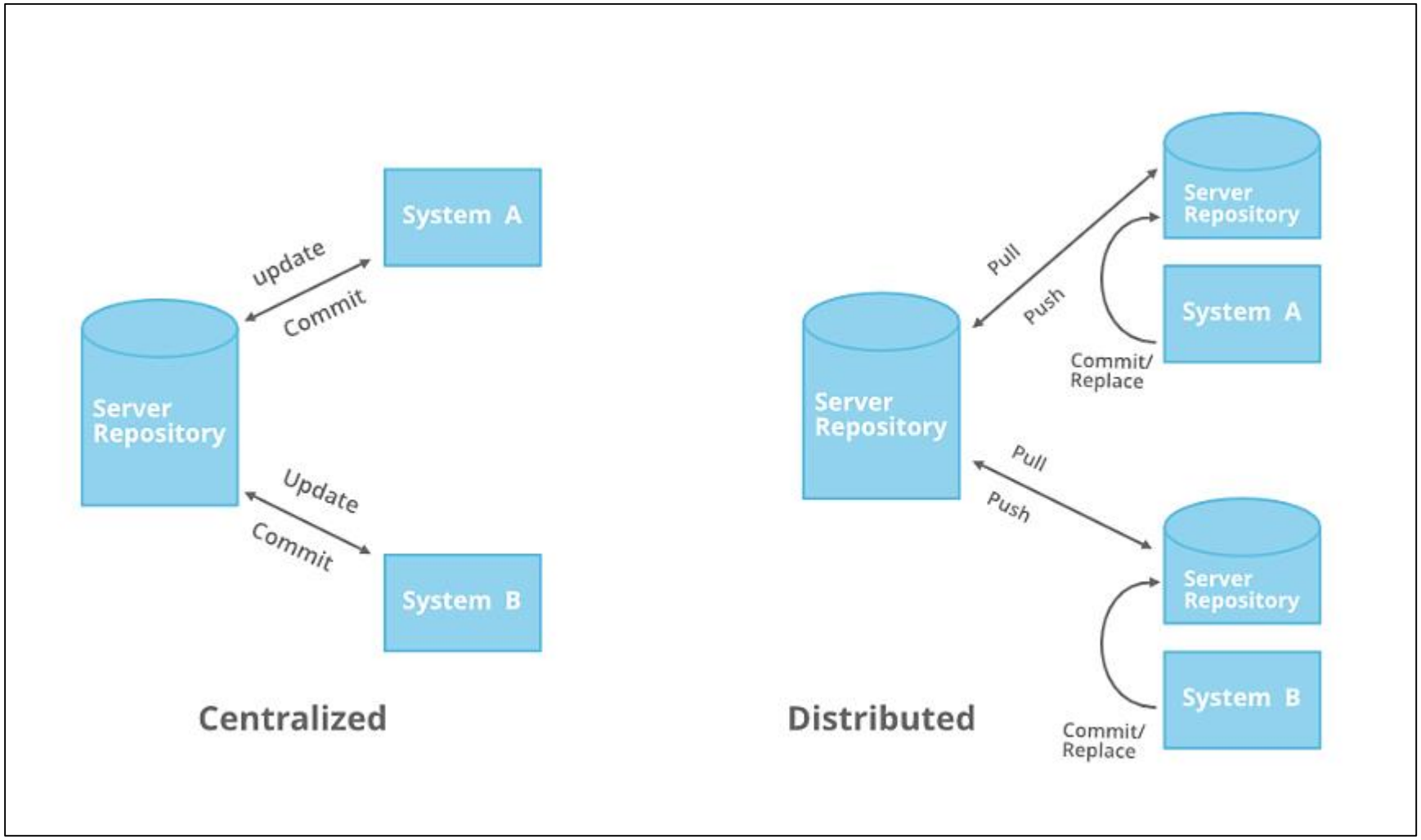
SCCS Centralisé

Tout l'historique des changements est conservé sur un serveur central (distant), duquel n'importe qui peut obtenir la version la plus récente ou envoyer les changements les plus récents.

SCCS distribué

Chaque usager a une copie locale de tout l'historique des changements. Il n'est donc pas nécessaire d'être connecté à un réseau/serveur pour sauvegarder des changements. En outre, n'importe quel usager peut se synchroniser avec n'importe quel autre.

Les différents SCCS : SCCS Centralisé vs. SCCS Distribué



LES AVANTAGES ET LES INCONVÉNIENTS DU SCCS DISTRIBUÉ

AVANATGES

- Chaque développeur a son espace privé — son sandbox
- On peut travailler — et faire des commits ! — hors ligne
- Plusieurs opérations sont très rapides — exécution locale
- La création et fusion de branches est efficace

INCONVÉNIENTS

- Quand même préférable d'avoir un dépôt central (sauvegarde)
- Pas nécessairement «une» version «la plus récente»

DÉMARRER AVEC GIT

- Git est un système de contrôle de version distribué gratuit qui permet aux programmeurs de suivre les modifications du code, via des "instantanés" (commits), dans leur état actuel.
- L'utilisation de validations permet aux programmeurs de tester, déboguer et créer de nouvelles fonctionnalités en collaboration. Tous les commits sont conservés dans ce que l'on appelle un « référentiel Git » pouvant être hébergé sur votre ordinateur, des serveurs privés ou des sites Web open source, tels que Github.
- Git permet également aux utilisateurs de créer de nouvelles "branches" du code, ce qui permet aux différentes versions du code de cohabiter.

LES CONCEPTS DU GIT

- Espace de travail : C'est l'endroit où vous travaillez sur vos fichiers. Il s'agit simplement des fichiers et répertoires que vous voyez et manipulez sur votre ordinateur. Ils n'ont rien de spécial par rapport à d'autres dossiers sur votre ordinateur.
- Dépôt local: Il s'agit de votre espace de travail associé à l'historique des modifications. C'est là que Git enregistre toutes les modifications que vous avez apportées au fil du temps.
- Commit : Un commit représente une version spécifique de votre projet à un moment donné. Chaque commit est comme une "capture d'écran" de l'état de vos fichiers à un moment précis. Il contient les modifications que vous avez apportées ainsi qu'un message descriptif.
- Historique : C'est la "chaîne" de tous les commits, du plus ancien au plus récent. Cet historique vous permet de voir l'évolution de votre projet au fil du temps.
- Dépôt distant : C'est un dépôt Git qui se trouve sur un serveur distant, comme GitHub. Il est souvent utilisé pour collaborer avec d'autres personnes sur un projet ou pour sauvegarder votre code.

LES ZONES DU GIT

Git a 3 "zones" différentes pour votre code:

- Répertoire de travail : zone dans laquelle vous allez travailler (création, modification, suppression et organisation des fichiers)
- Zone de transit : la zone dans laquelle vous listerez les modifications apportées au répertoire de travail
- Référentiel : où Git stocke en permanence les modifications que vous avez apportées sous différentes versions du projet

CONTRÔLE WORKFLOW

Le contrôle de version a un flux de travail général que la plupart des développeurs utilisent pour écrire du code et le partager avec l'équipe

- Obtenez une copie locale du code s'ils n'en ont pas encore.
- Apportez des modifications au code pour corriger les bogues ou ajouter de nouvelles fonctionnalités.
- Une fois le code prêt, rendez-le disponible pour examen par votre équipe (Pull Request).
- Une fois le code révisé, fusionnez-le dans la base de code partagée de l'équipe.

ACTIONS AVEC GIT

- **Créer** un dépôt sur GitHub.
- **Cloner** (faire une copie d') un dépôt de GitHub sur son PC: Git clone URL_Repo
- **Ajouter un fichier modifié :** il sera pris en compte dans le prochain commit: Git add nom_fichier / . / -a / *
- **Faire un commit :** créer une nouvelle version, qui contient tous les fichiers ajoutés. On y ajoute un commentaire (qui décrit les changements): Git commit –m « message du commit »

ACTIONS AVEC GIT

- Consulter un historique: Git log
- Push : envoyer ses nouveaux commits sur GitHub: Git push
- Pull : récupérer des changements (qui ont été envoyés par quelqu'un d'autre) depuis GitHub: Git pull
- Branches :
  - Créer une branche: Git branch nom_branche
  - Basculer entre les branches: Git checkout nom_branche
  - Lister les branches: Git branch –a
  - Supprimer une branche: Git branch –d nom_branch
- Merge : quand on Pull et qu'on a aussi des nouveaux commits sur son PC. Git essaye de fusionner automatiquement ; s'il ne sait pas le faire, il demande à l'utilisateur: Git merge

Bonnes pratiques pour le contrôle des sources

- Faites de petits changements, avec des commit régulières.
- Ne pas comité des fichiers personnels.
- Mettre à jour souvent et juste avant de pousser pour éviter les conflits de fusion.
- Vérifiez votre changement de code avant de le pousser dans un référentiel, assurez-vous qu'il compile et que les tests réussissent.
- Portez une attention particulière à la validation des messages car ceux-ci vous indiqueront pourquoi un changement a été effectué.
- Lier les modifications de code aux éléments de travail (WorkItem).

Gestion des Branches

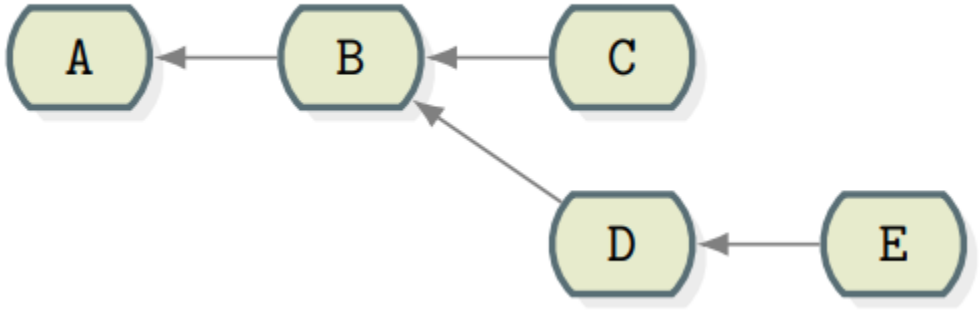
Motivations

Une équipe de développeurs participe à la réalisation d'une application:

- Comment conserver un historique?
- Comment revenir en arrière?
- Comment travailler à plusieurs en parallèle sur le même code?
- Comment gérer plusieurs versions du code à la fois?
- Comment savoir ce qui a modifié et par qui (et pourquoi)?

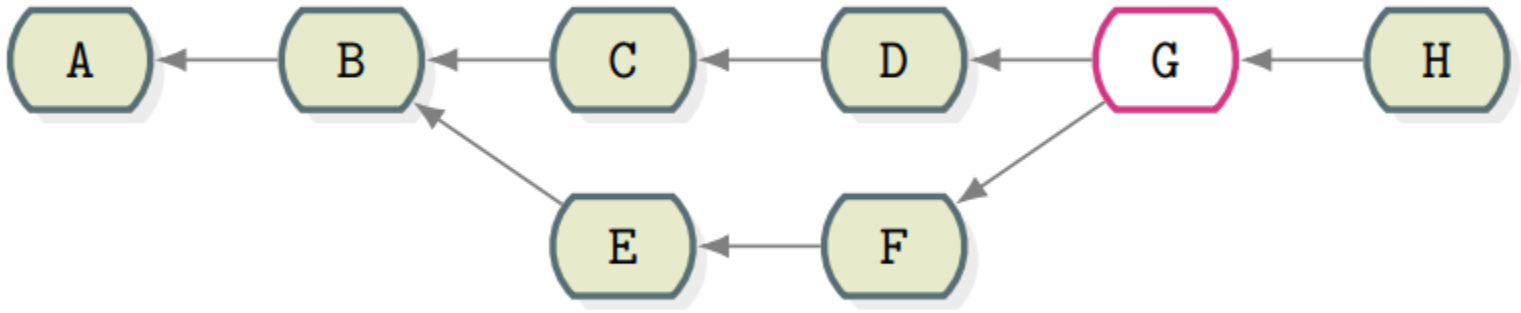
La notion d’historique

• L’historique est un graphe orienté composé d’un ensemble de versions pouvant être recalculées à partir des versions adjacentes en appliquant les patchs.
• L’historique peut inclure plusieurs branches, c’est-`a-dire des sous-graphes qui évoluent en parallèle.



Historique: les merges

On appelle merge toute version ayant un degré sortant strictement supérieur à 1.
Cette version correspond alors à la fusion des patchs de plusieurs branches.



Gestion des accès concurrents

Gestion pessimiste

- Un seul contributeur à accès en écriture à un fichier
- Pas de conflits
- Pas pratique

Gestion optimiste

- Chaque développeur peut modifier sa copie locale en parallèle
- Risques de conflits

► Modifications concurrentes de la même zone de texte

Les objets dans Git

- *Blobs*
- *Tree*
- *Commit*
- *Tag*

Blob « Binary Large Object »

On appelle Blob, l'élément de base qui permet de stocker le contenu d'un fichier.
C'est simplement le contenu d'un fichier, stocké de manière compressée dans la base de données Git. Les blobs sont identifiés par des clés SHA-1 qui sont calculées à partir de leur contenu.

- Chaque Blob est identifié de manière unique par sa clé
- A chaque révision du fichier correspond un nouveau Blob
  - Le blob stocke le contenu entier du fichier

Le Blob ne dépend pas du nom ou de l'emplacement

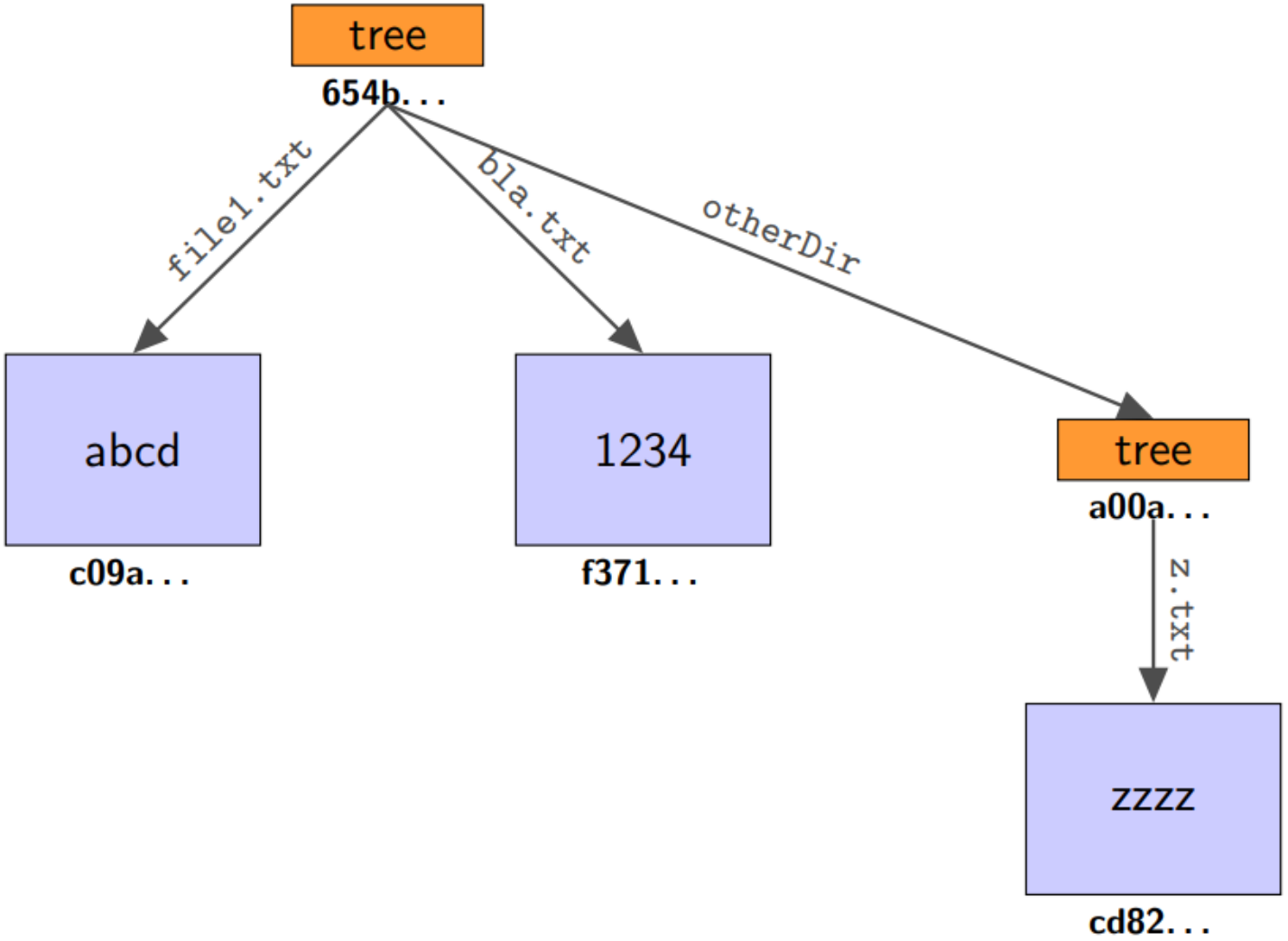
  - Si un fichier est renommé, pas de nouveau Blob
  - Si un fichier est déplacé, pas de nouveau Blob

- Le contenu du Blob est compressé avec zlib. Il contient:
  - Le type d'objet (blob)
  - La taille du fichier initial
  - Le contenu du fichier

Tree

- Un arbre représente un répertoire (ou dossier) dans un projet Git.
- Chaque arbre contient des entrées qui pointent vers des blobs correspondant aux fichiers présents dans ce répertoire, ainsi que vers d'autres sous-arbres pour les sous-répertoires.
- L'entrée d'un arbre contient des métadonnées telles que le nom du fichier ou du sous-répertoire, le mode (permissions), et l'identifiant SHA-1 du blob ou de l'arbre auquel elle pointe.

Tree et Blob: Exemple

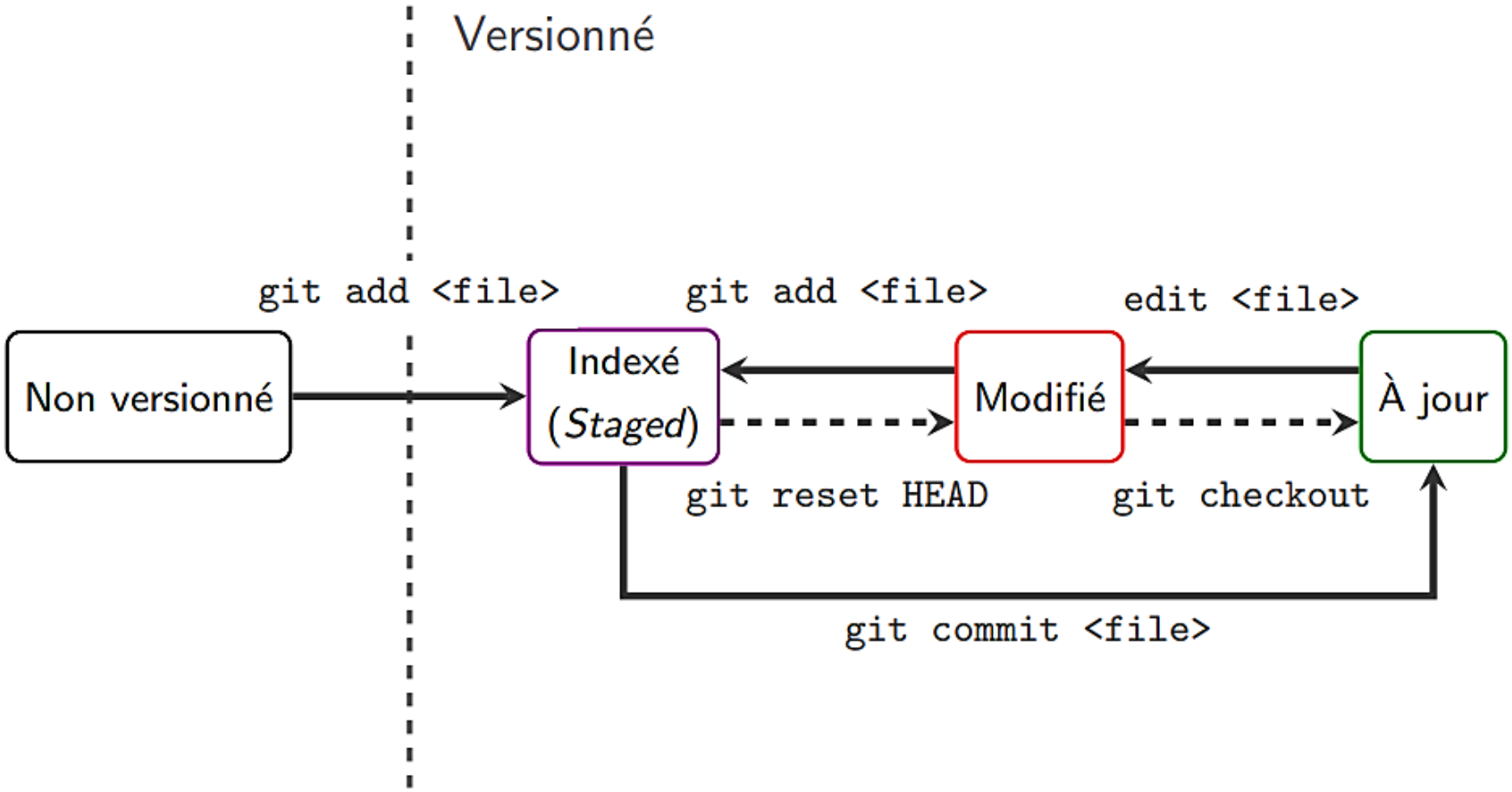


Commit

Un Commit stocke l'état d'une partie du dépôt à un instant donné.
Il contient :

- Un pointeur vers un Tree (arbre racine) dont on souhaite sauver l'état.
- Un pointeur vers un ou plusieurs autres Commits pour constituer un historique.
- Les informations sur l'auteur du Commit.
- Une description sous forme d'une chaîne de caractères.

Le cycle de vie d'un fichier



Les branches

Dans Git

- Une branche est un pointeur sur un commit
- Chaque commit pointe vers son prédécesseur
- La variable HEAD pointe sur la branche sur laquelle on travaille actuellement.

Les branches – Les commandes

branch : liste les branches avec une * pour la branche active.
branch <nom> : crée une nouvelle branche <nom>.
branch -m : permet de renommer une branche.
branch -d : permet de supprimer une branche.
checkout : change (ou/et crée) de branche active.
show-branch : affiche les branches et leurs commits.

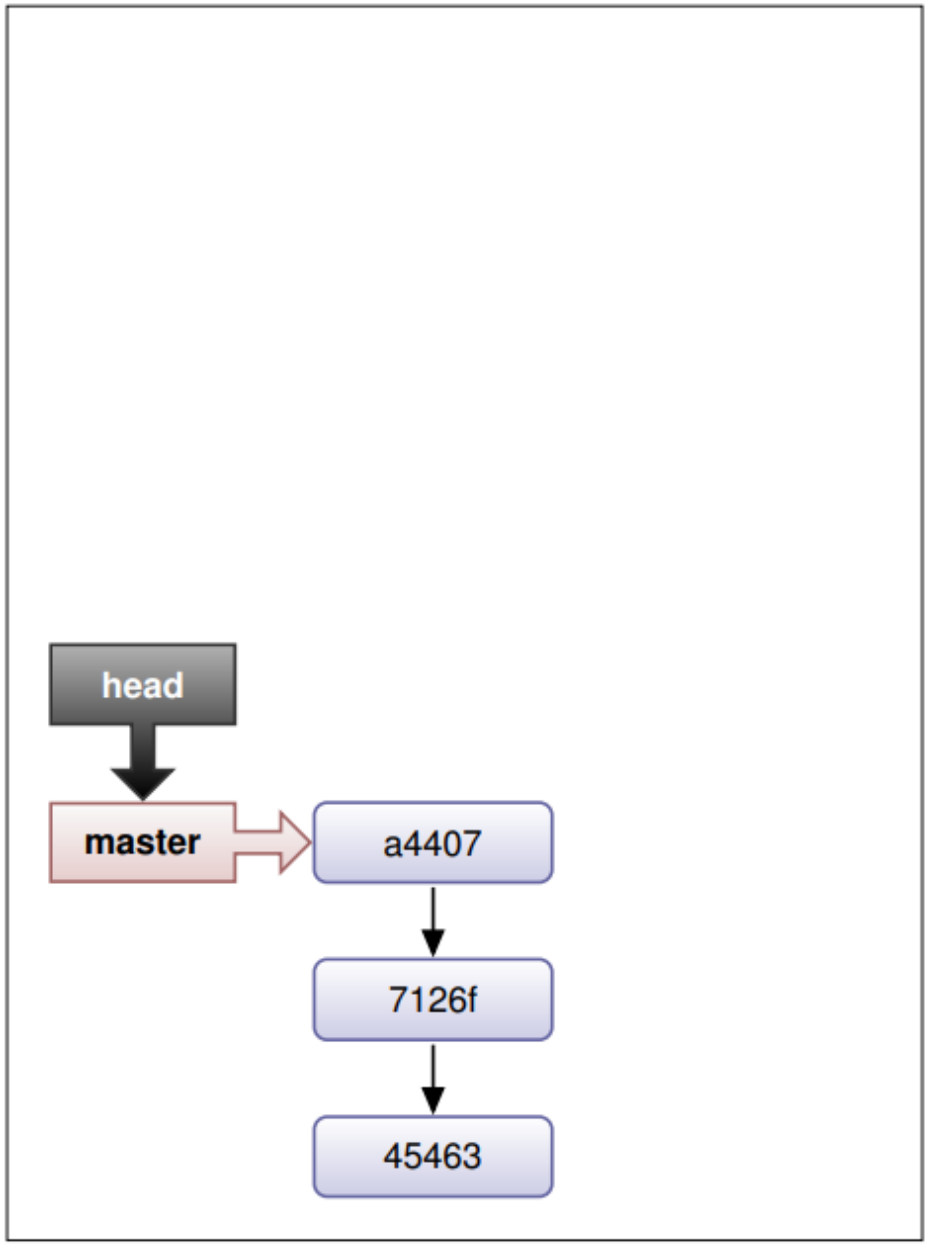
Exemple

```
$ git branch
    * master
$ git branch maBranche
$ git branch
      maBranche
    * master
$ git checkout maBranche
$ git branch
    * maBranche
      master
```

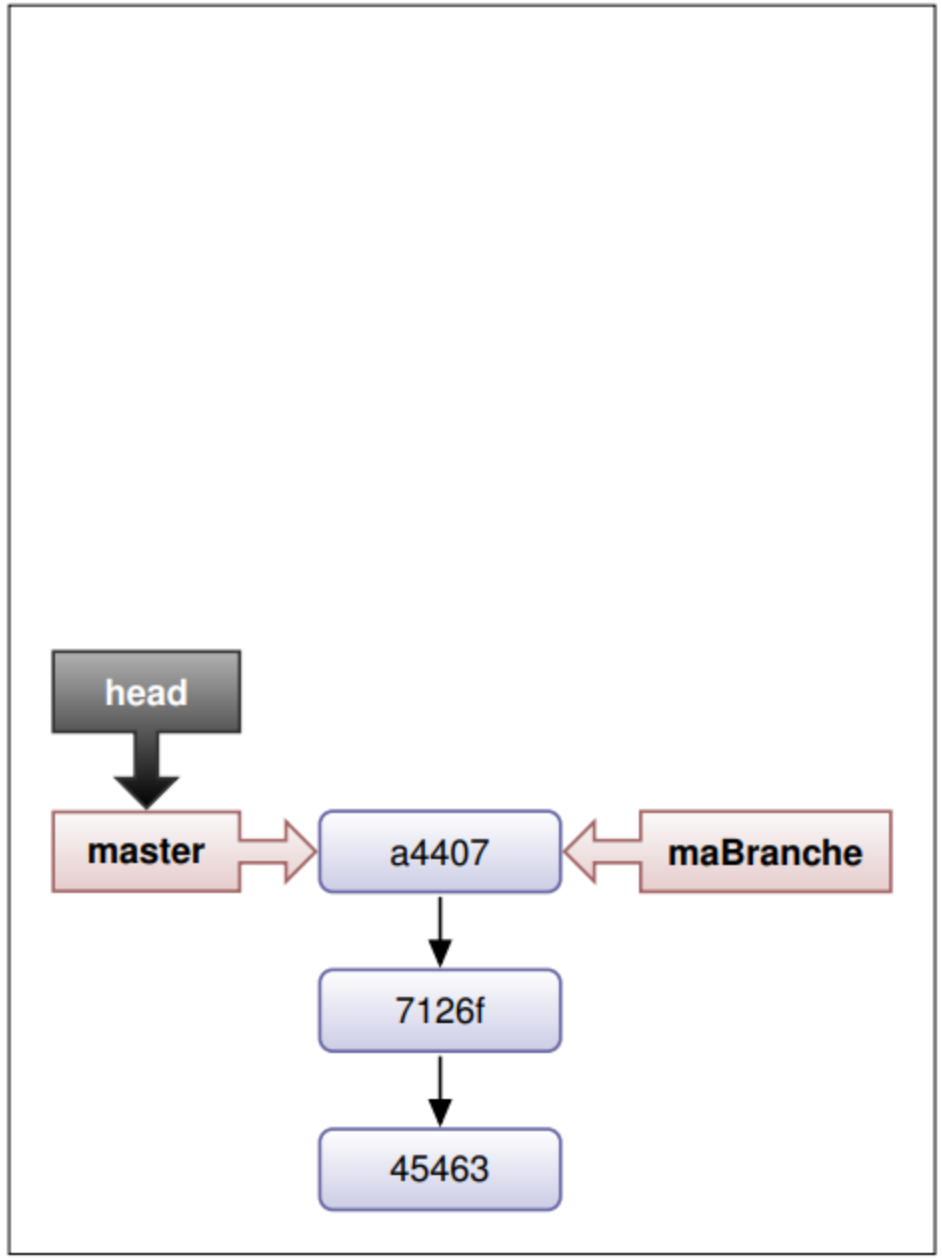
Les Merges

$ git checkout brancheDestination
$ git merge brangeSource
• Crée un commit qui a pour parent les deux branches
• La branche courante avance à ce commit
• La source ne bouge pas, mais devient un fils du nouveau commit

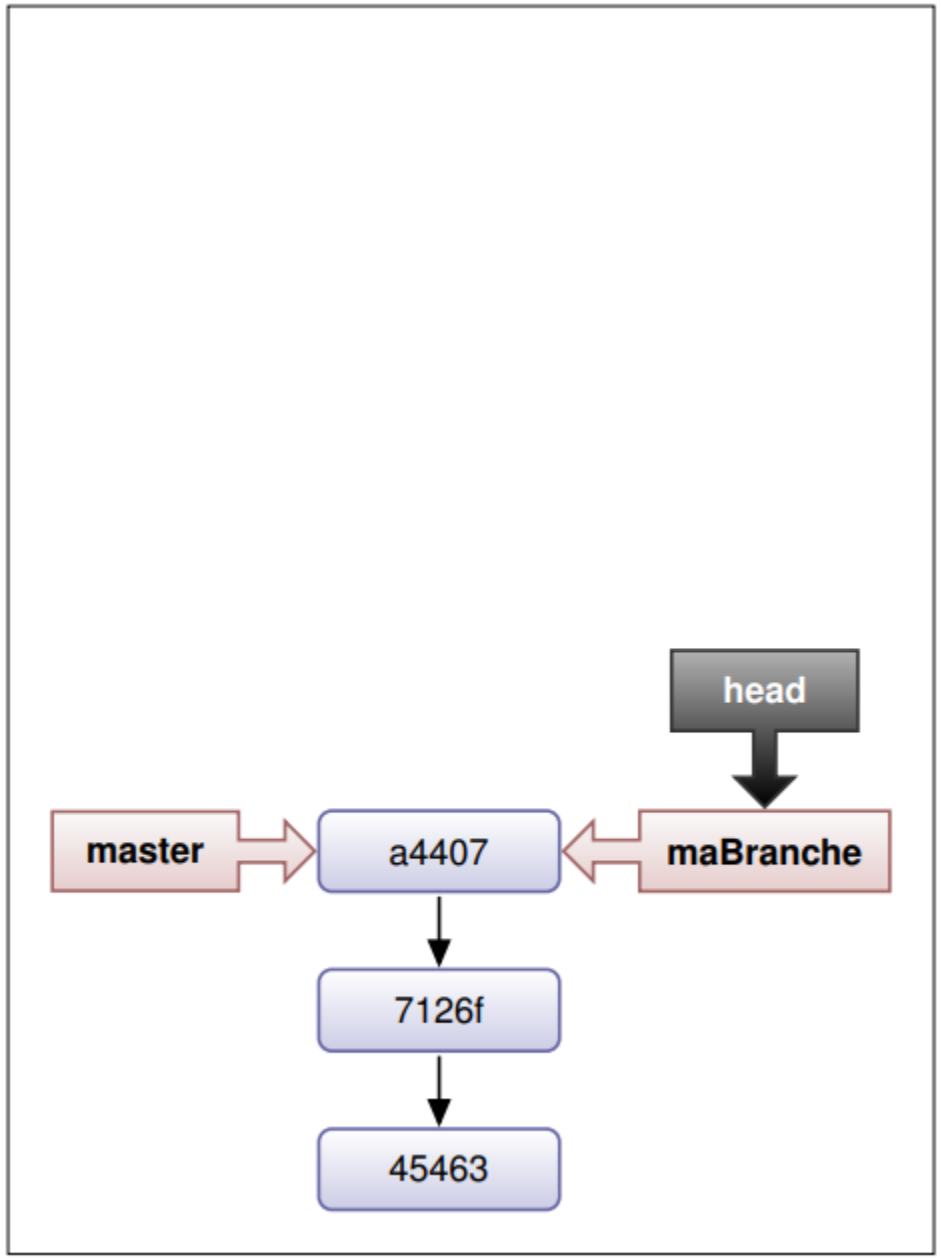
```
ls
 foo.txt dir
```



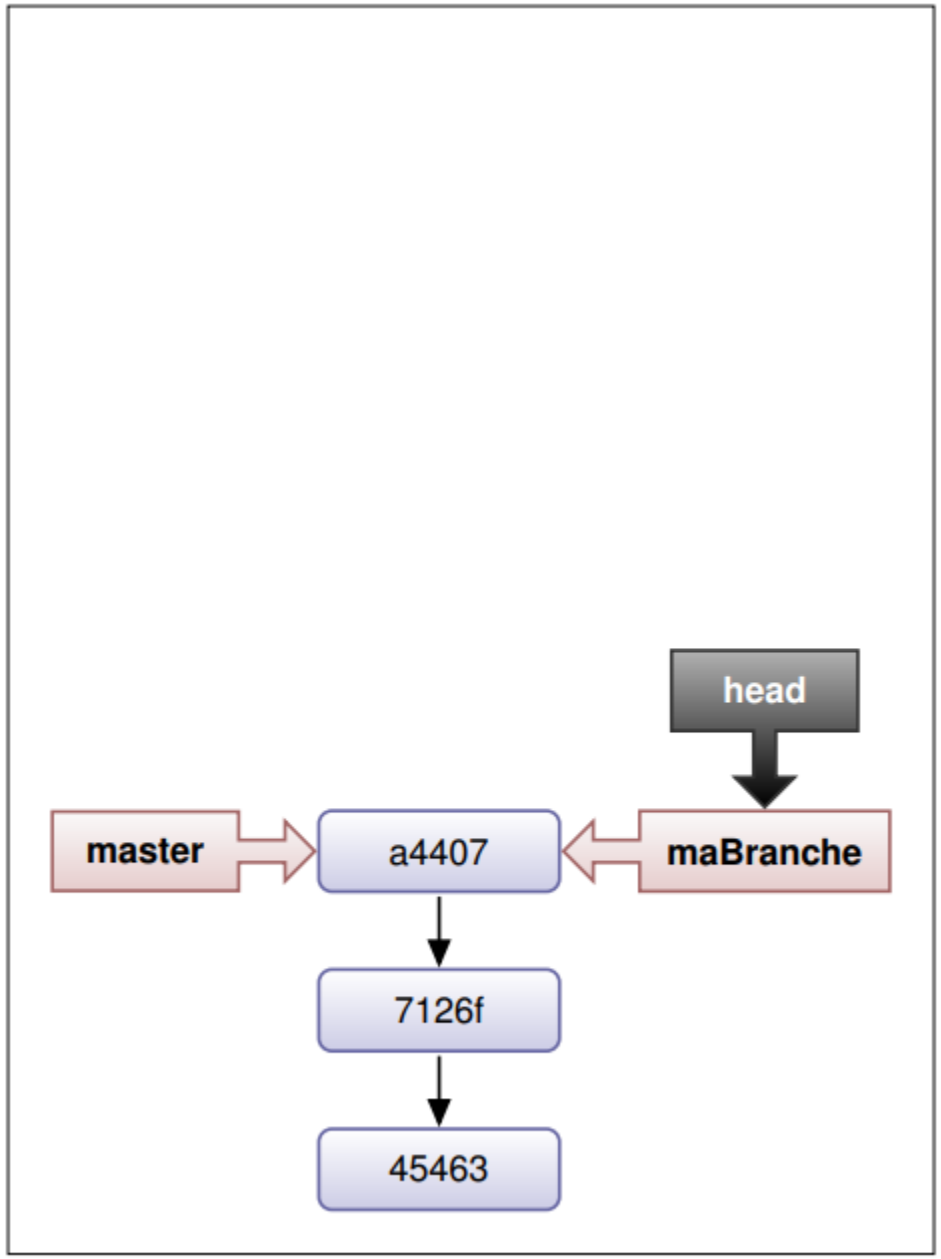
```
ls
 foo.txt dir

git branch maBranche
```



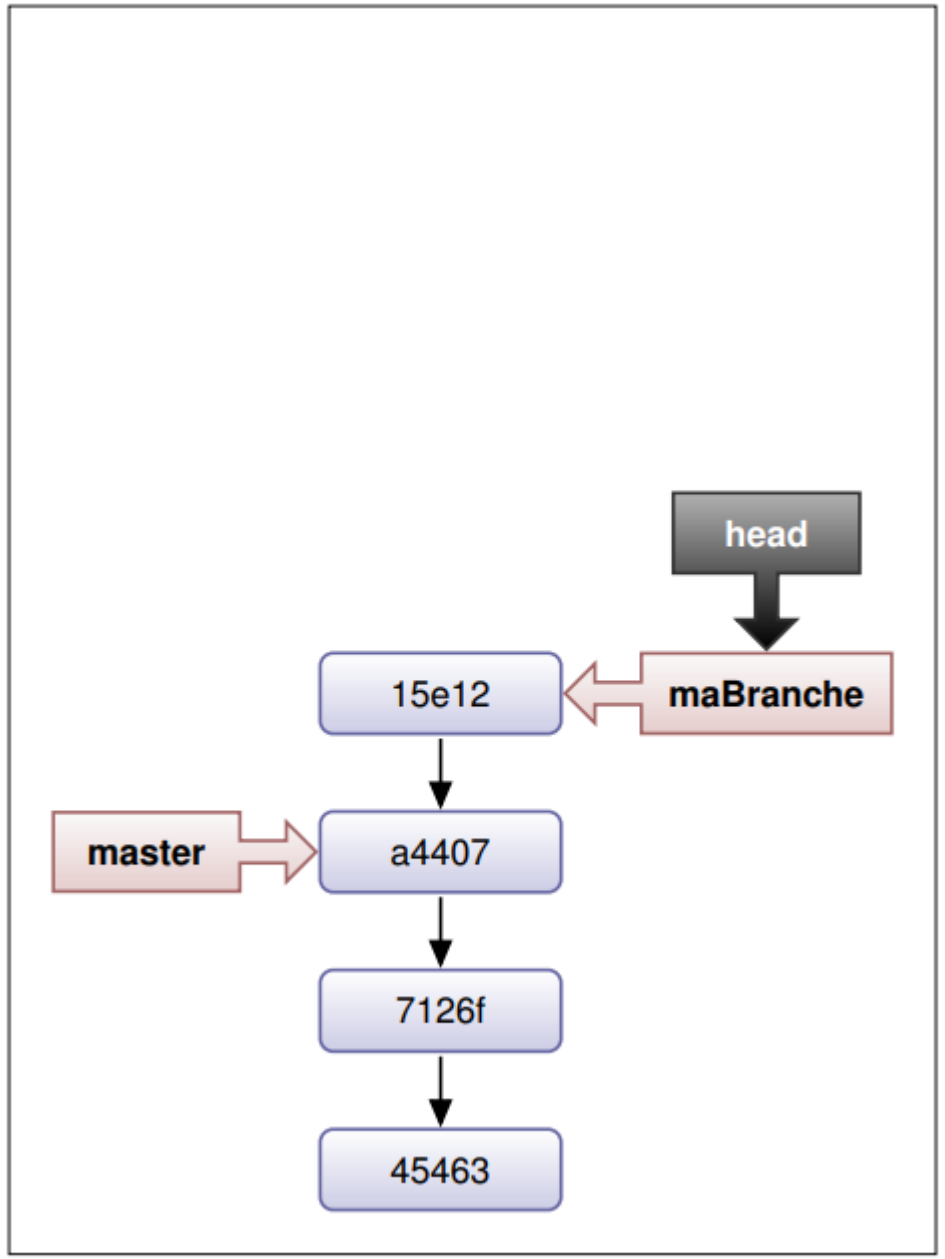
```
ls
 foo.txt dir

git branch maBranche

git checkout maBranche
```



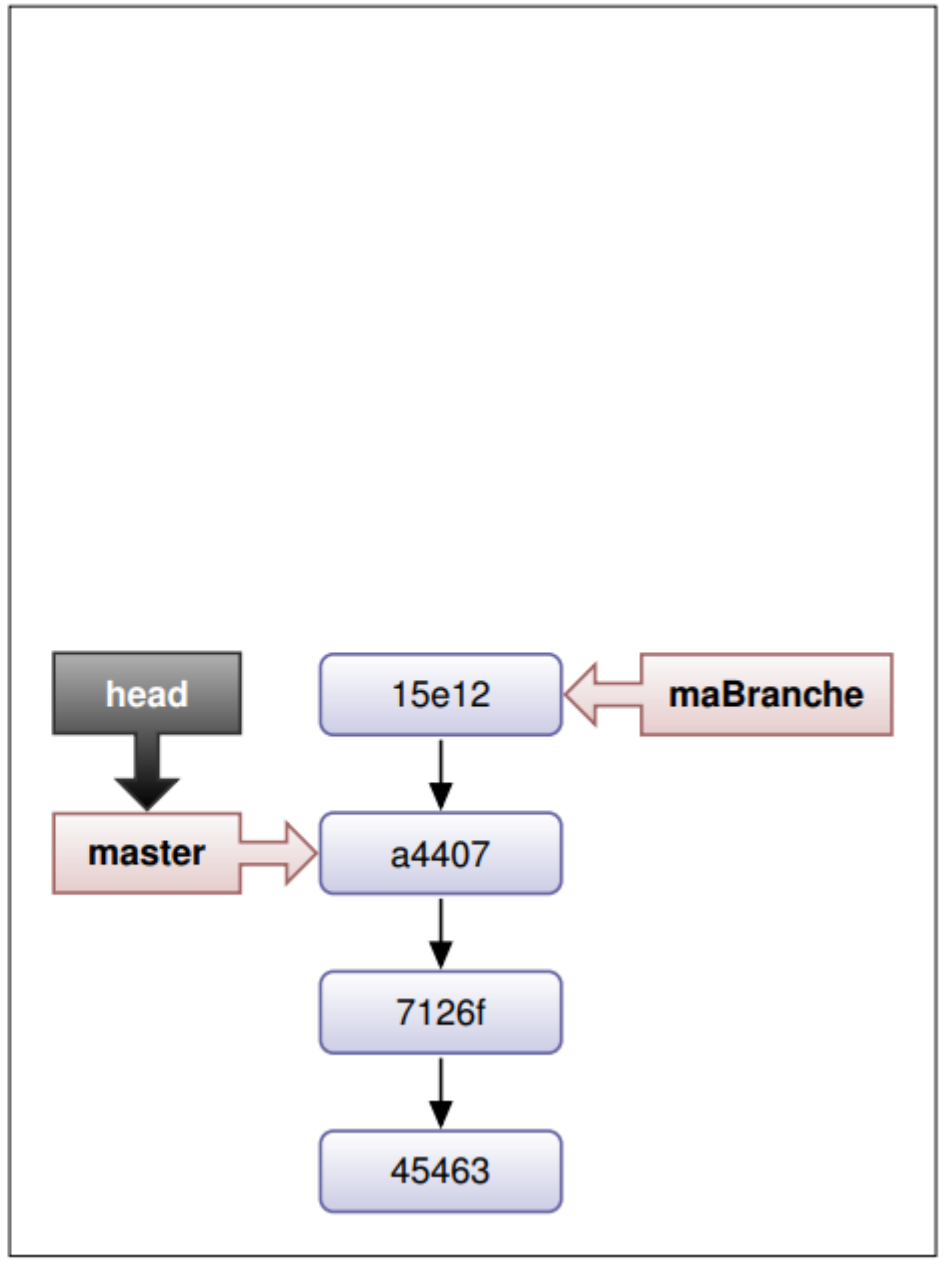
```
ls
 foo.txt dir

git branch maBranche

git checkout maBranche
```



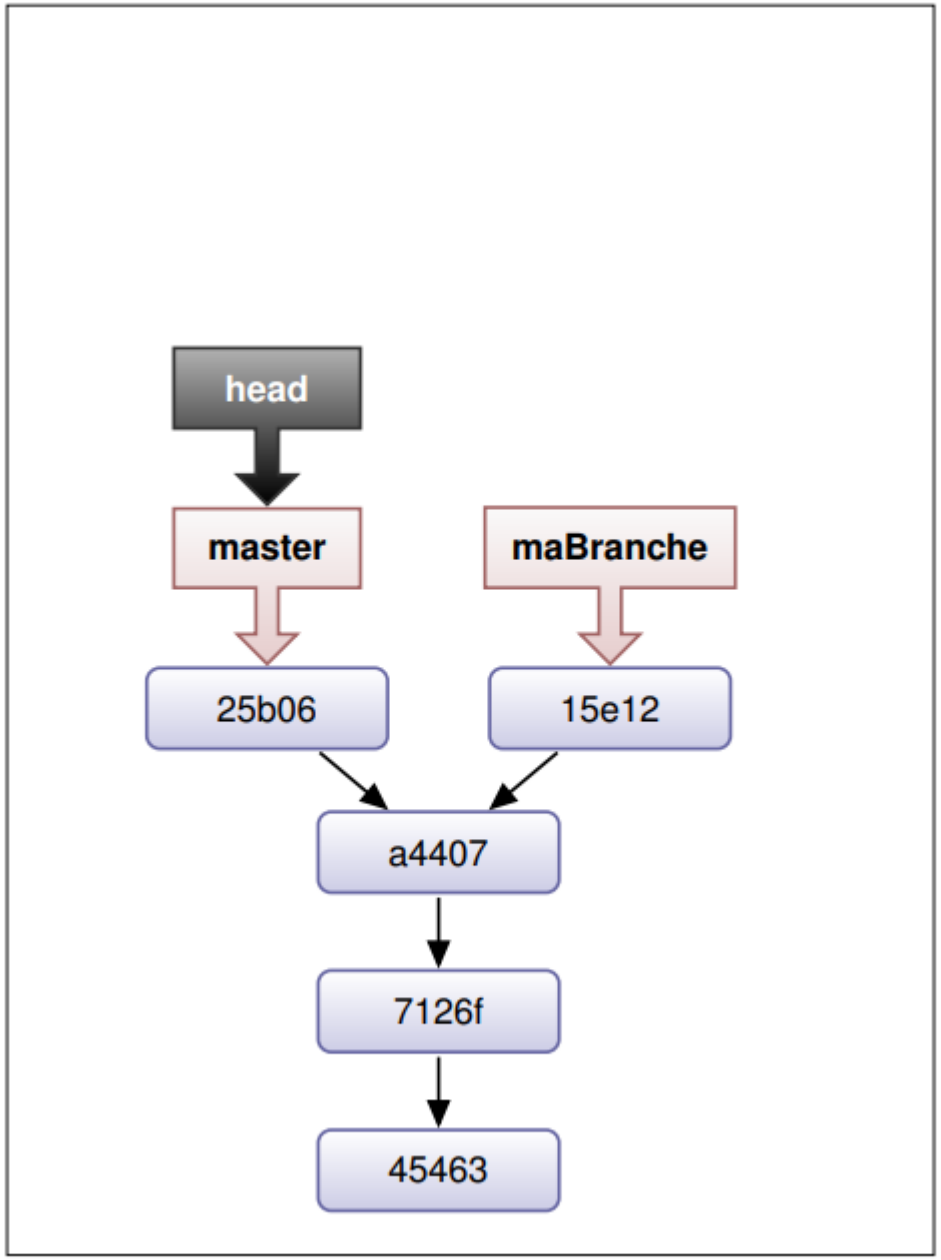
```
ls
 foo.txt dir

git branch maBranche

git checkout maBranche

touch fichier1.txt
ls
 dir fichier1.txt foo.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"
```



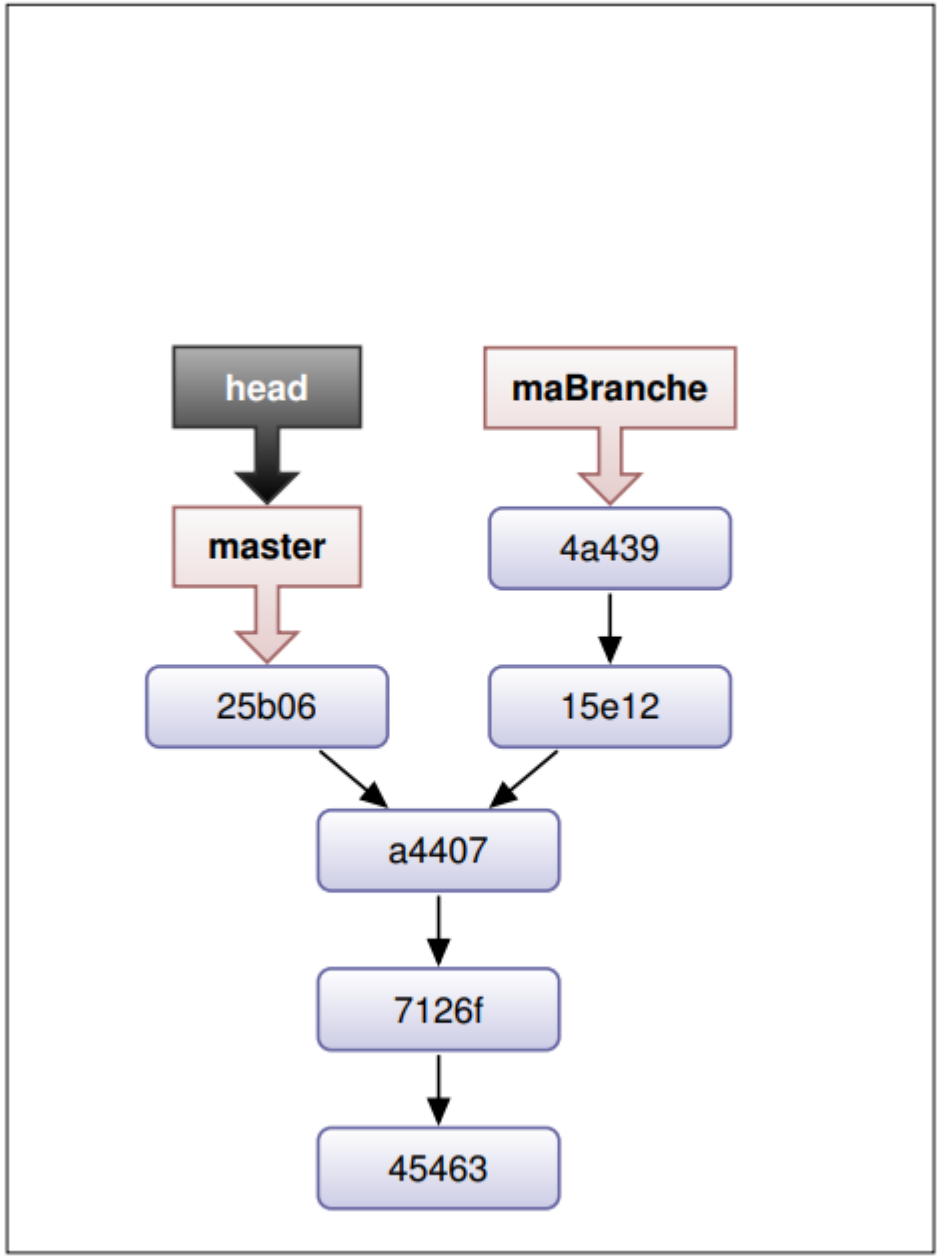
```
ls
 foo.txt dir

git branch maBranche

git checkout maBranche

touch fichier1.txt
ls
 dir fichier1.txt foo.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

git checkout master
ls
 dir foo.txt
```



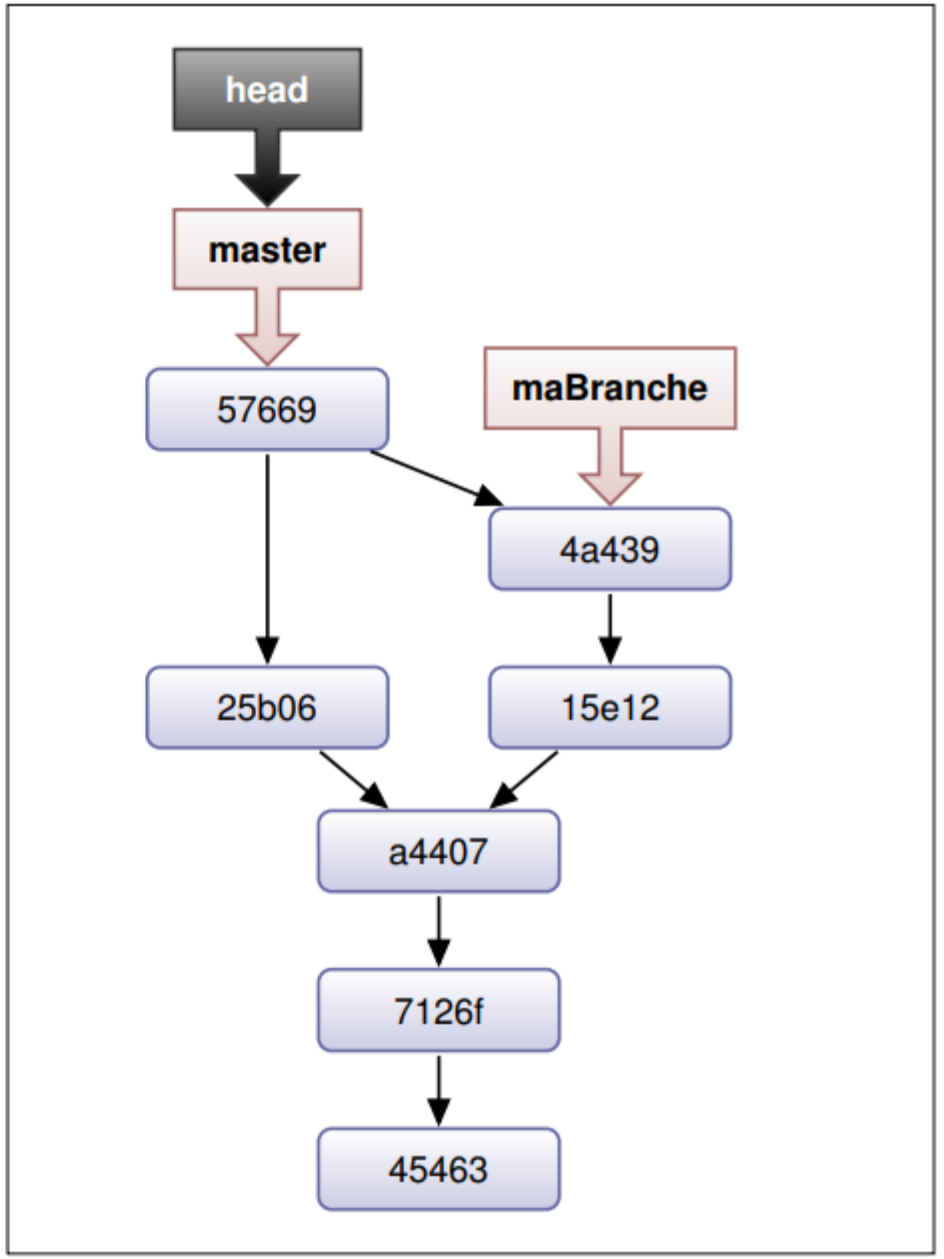
```
ls
 foo.txt dir

git branch maBranche

git checkout maBranche

touch fichier1.txt
ls
 dir fichier1.txt foo.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

git checkout master
ls
 dir foo.txt

touch fichier2.txt
git add fichier2.txt
git commit -m "Add fichier2.txt"
```



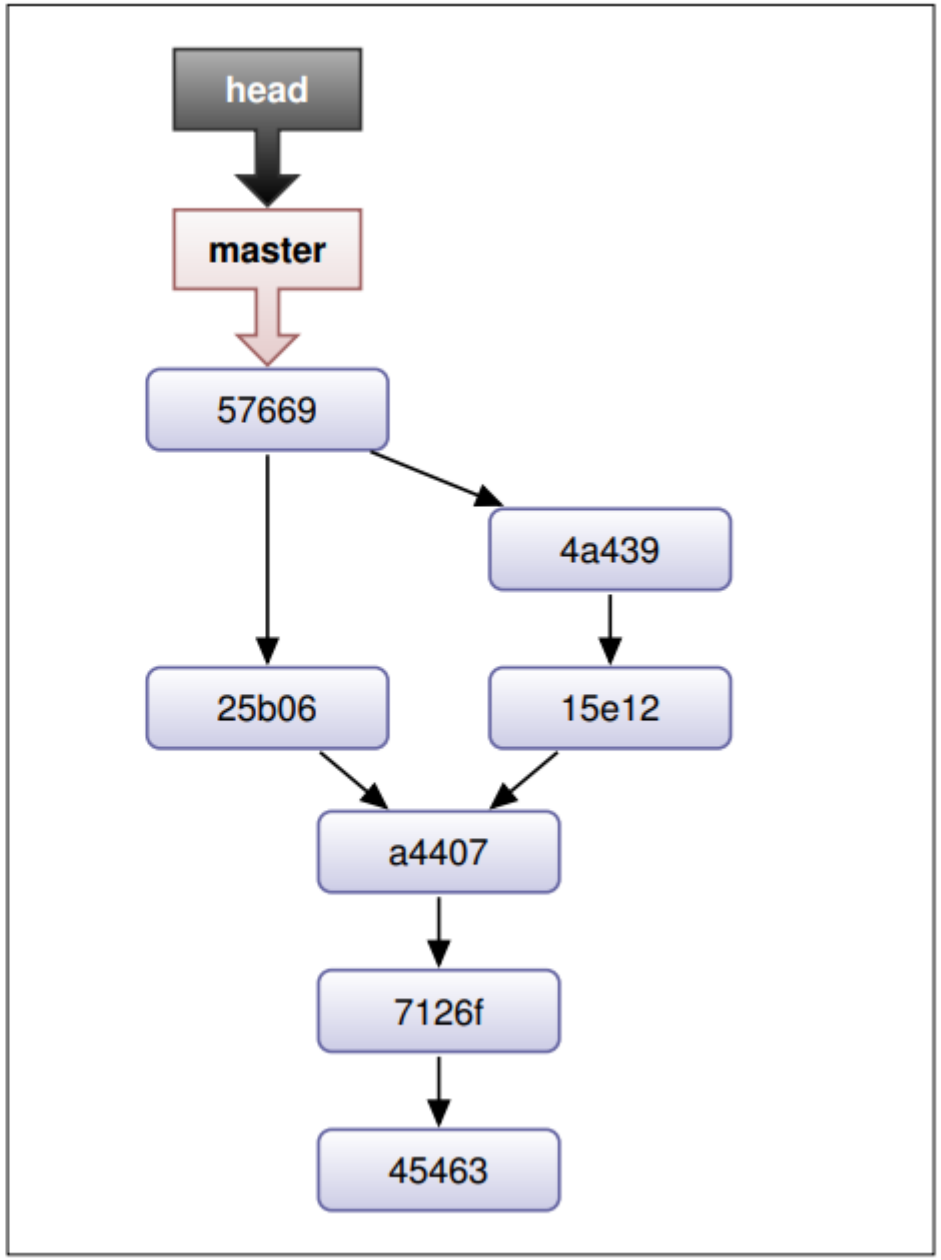
```
ls
 dir fichier2.txt foo.txt

git checkout maBranche
ls
 dir fichier1.txt foo.txt

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"

git checkout master
```



```
ls
 dir fichier2.txt foo.txt

git checkout maBranche
ls
 dir fichier1.txt foo.txt

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"

git checkout master
git merge maBranche
ls
 dir fichier1.txt fichier2.txt
foo.txt
```



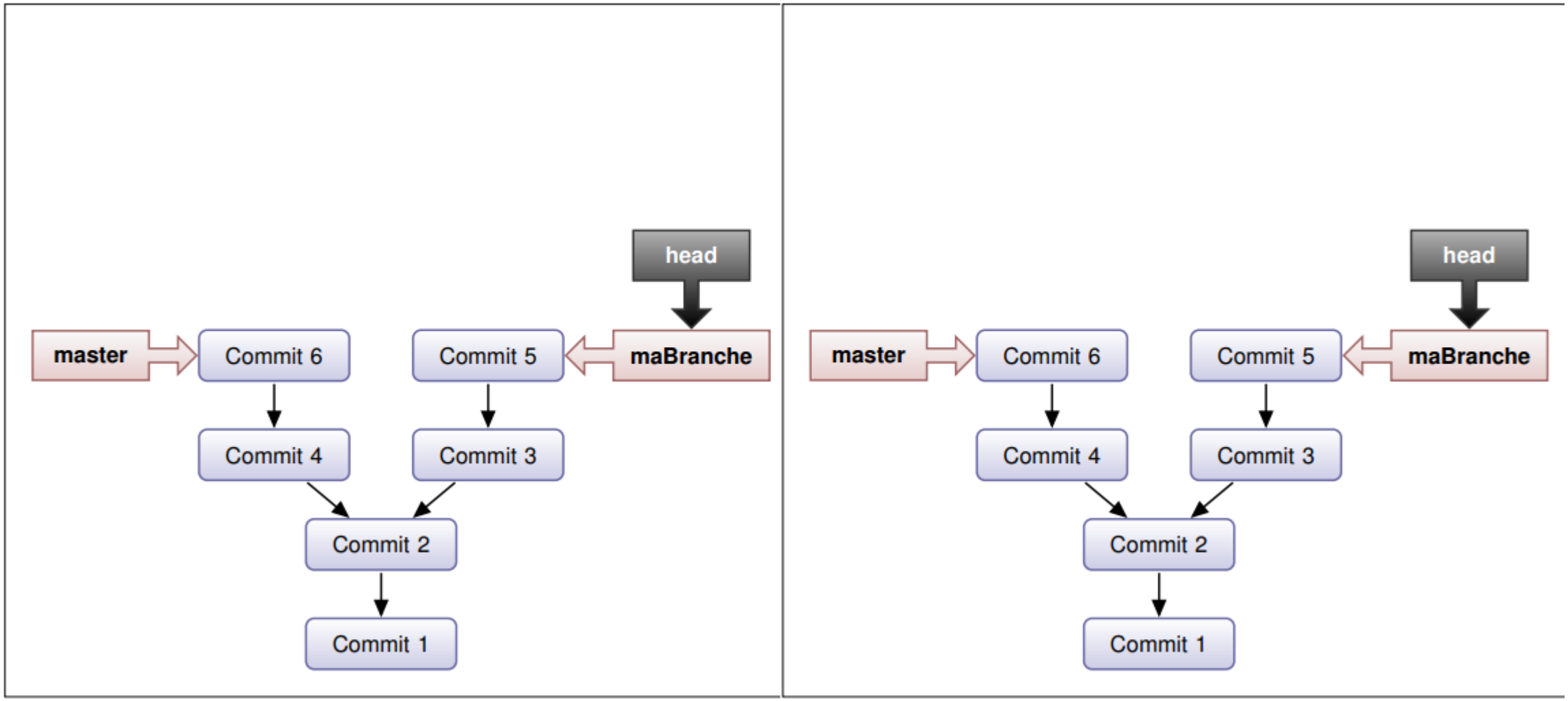
```
ls
 dir fichier2.txt foo.txt

git checkout maBranche
ls
 dir fichier1.txt foo.txt

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"

git checkout master
git merge maBranche
ls
 dir fichier1.txt fichier2.txt
foo.txt

git branch -d maBranche
```



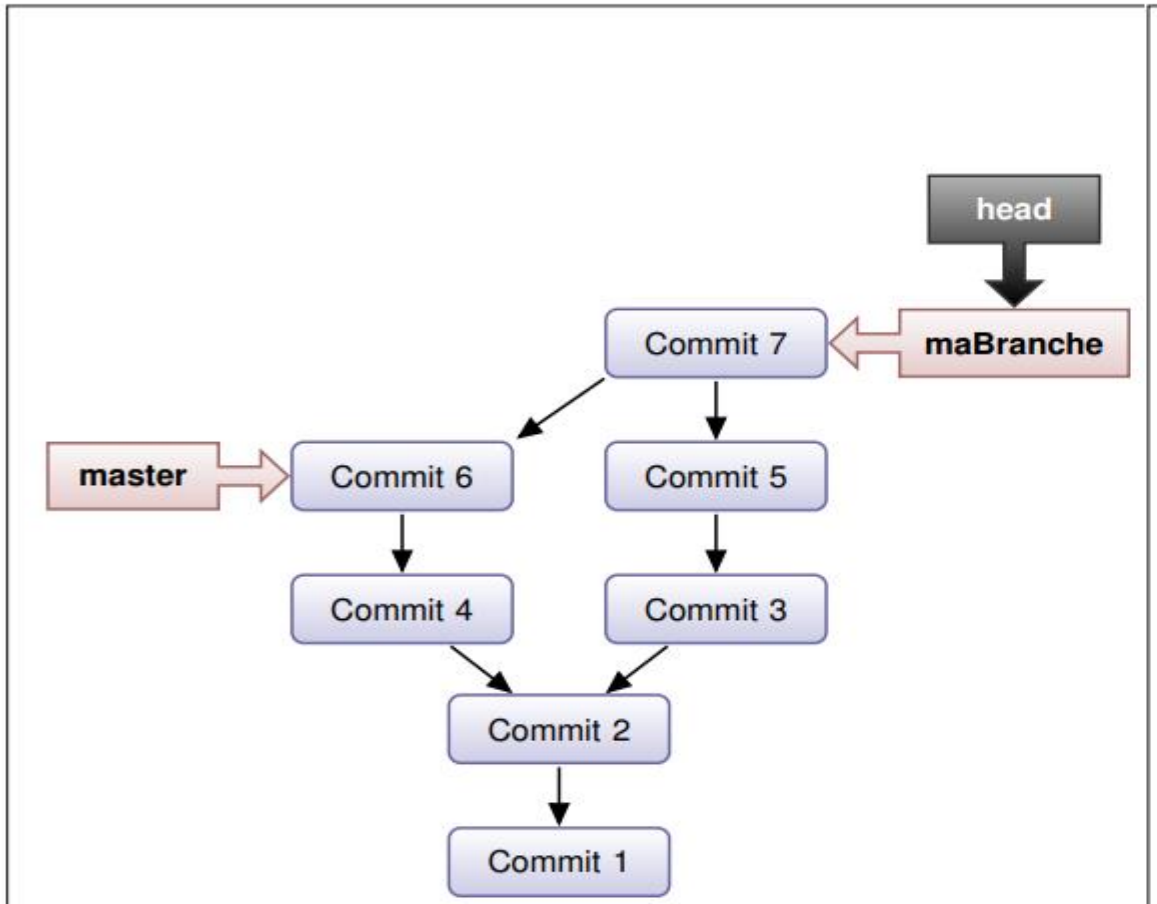
Rebase

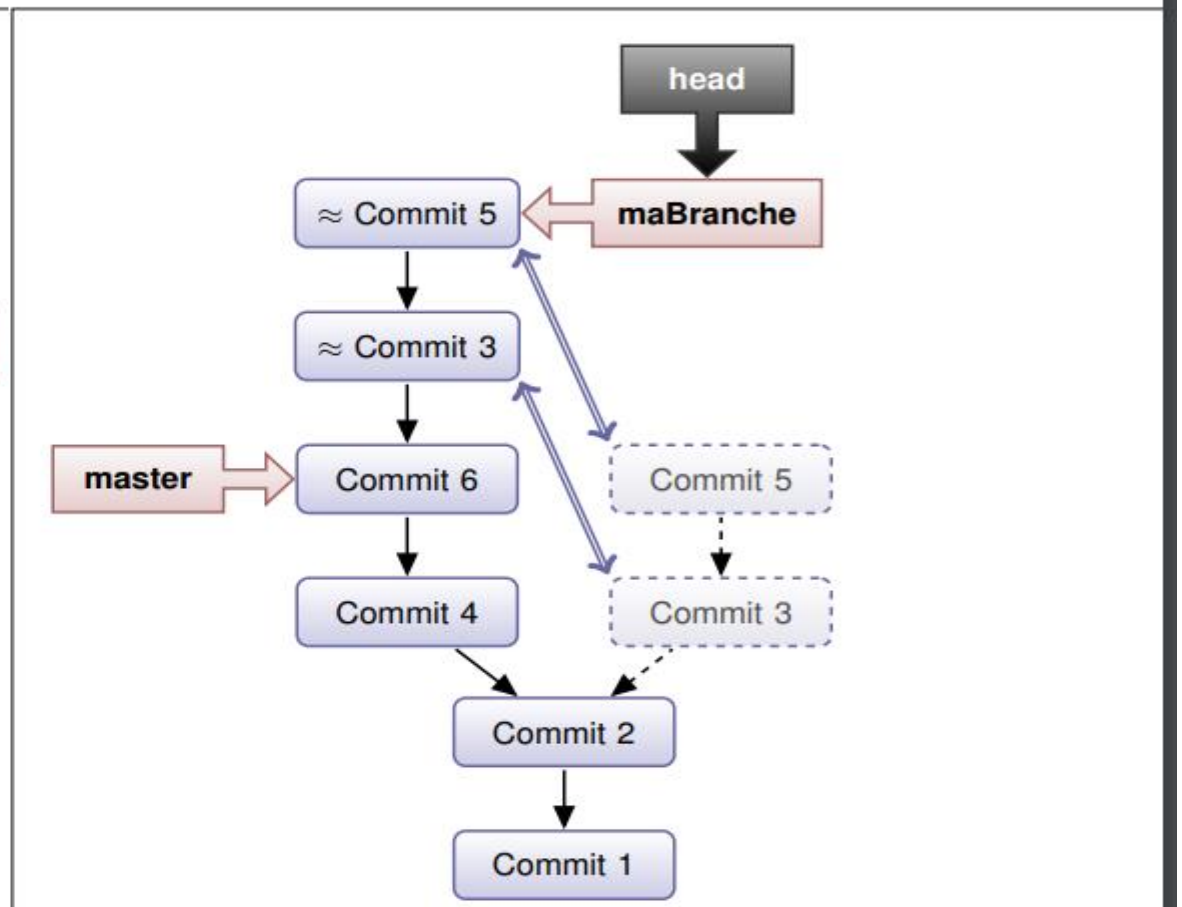
- Autre manière de fusionner 2 branches
- Permet de simplifier l’historique
- **Fusion (merge)** : Lorsque vous fusionnez deux branches avec **git merge**, Git combine l'historique des deux branches pour **créer un nouveau commit de fusion**. Ce nouveau commit de fusion a deux parents, représentant les commits les plus récents de chaque branche. En d'autres termes, Git prend tous les changements effectués dans les deux branches et les intègre dans un nouveau commit de fusion. Cela signifie que l'historique de développement de chaque branche est préservé.
- **Réalignement (rebase)** : Lorsque vous réalignez une branche avec une autre branche à l'aide de **git rebase**, **Git déplace les commits de la branche en cours de réalignement pour qu'ils apparaissent au-dessus des commits de l'autre branche**. Cela récrit essentiellement l'historique de la branche en cours sur le dessus de l'autre branche. Contrairement à la fusion, le réalignement n'ajoute pas de commit de fusion supplémentaire. Au lieu de cela, il réorganise l'historique de manière à ce que l'historique semble linéaire, ce qui peut rendre l'historique du projet plus clair.

GIT MERGE VS GIT REBASE



GIT MERGE VS GIT REBASE






```
git checkout maBranche
git merge master
```

```
git checkout maBranche
git rebase master
```

Rebase interactif

- Permet de réécrire l'historique en partant de CommitID
- Principales actions possibles
  - reword: editer le message du commit
  - squash: fondre le commit dans le commit précédent
  - drop: supprimer le commit

Message de commit

Le plus important

- Décrire quoi et pourquoi et pas comment
- Ne pas décrire les modifications qui sont faites (informations disponibles avec un diff)
- Décrire les fonctionnalités ajoutées

Exemple

- Bad: Modifie la fonction f pour tester la variable a
- Good: Vérifie les droits de l'utilisateur avant d'exécuter l'action X