

# Git

Présenté par Iheb Eljani

# Git

1.Partager ses sources

2.Fonctionnement de GIT

3.Cycle de vie d'un fichier

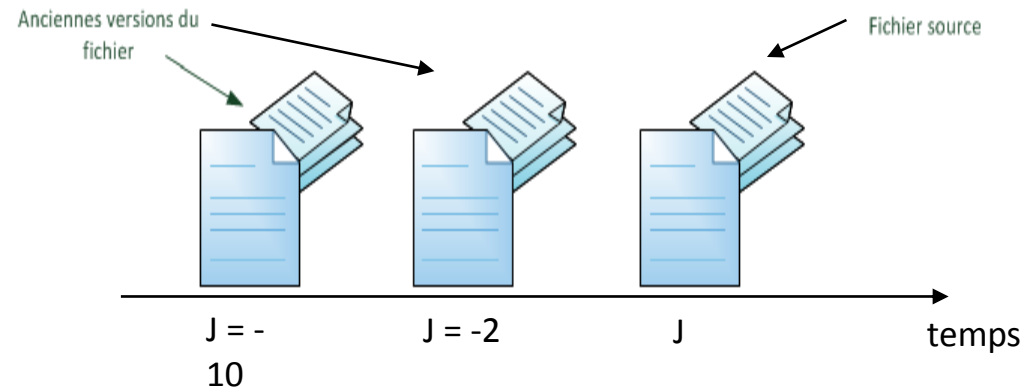
4.Commandes de bases

5.Notion de branches

# GIT

## Concepts généraux (1)

- Un logiciel de contrôle de versions (SVC) est un logiciel qui enregistre tous les états d'une arborescence au fil du temps **permettant de revenir sur une version précédente**.



- Un SVC permet de **tracer les modifications de code** sources mais également toutes sortes d'information telles que documentations ou encore images car tout est donnée.

# GIT

## Concepts généraux (2)

- Un SVC **photographie** (parfois de manière différentielle) l'arborescence au fil du temps et organise les données sous forme de **révisions** ou plan de révisions.
- Un SVC garde une trace de **toutes les modifications**
  - qui, quand, où, quoi, pourquoi
- Un SVC permet de développer du code de façon **collaborative** en autorisant plusieurs développeurs à travailler et **modifier un même fichier** (contrairement à Synergie : technique de lock).
  - Evite les « archives » dont on perd vite le compte
  - Permet un gain de place
  - Evite de devoir prévenir les utilisateurs à chaque modification
  - Evite l'échange de versions par mail...

# GIT

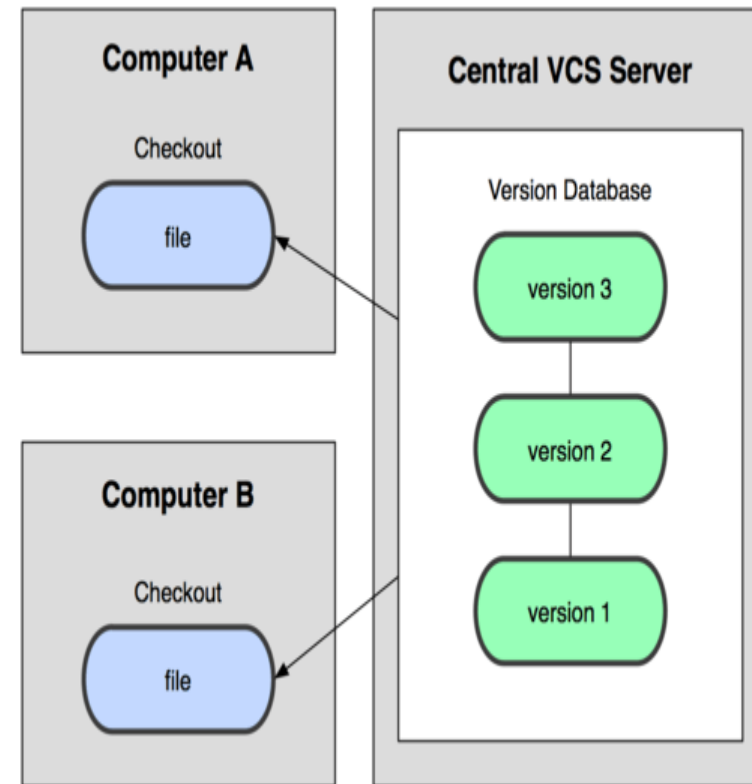
## Concepts généraux (3)

- **Logiciels de gestion centralisés (CVCS)**

- Référent **unique** sur serveur.
- Les postes **se connectent** pour se synchroniser.

**Permet un travail collaboratif**

Exemples : CVS (1990),  
ClearCase, Synergie, PVCS,  
SVN (2000)...

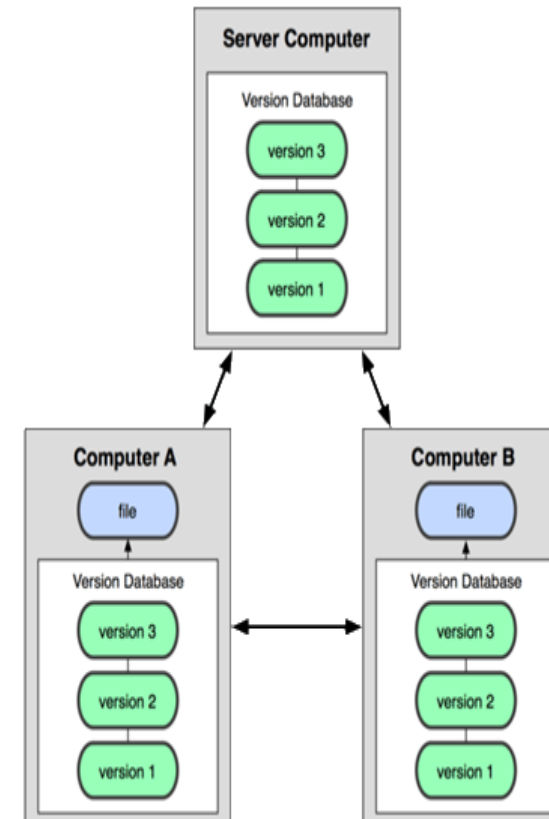


# GIT

## Concepts généraux (4)

- **Logiciels de gestion décentralisés ( DVCS)**
  - Pas de dépôt central **unique**, chaque développeur dispose **en local** d'un historique des versions
  - Le développeur est responsable de son dépôt et doit **intégrer** (merge) **le travail des autres** en ramenant dans le sien
  - Le travail peut être livré (commit) sans accès au réseau
- **Permet un travail collaboratif sans surcharge du serveur**

Exemples : Bazar, Mercurial, BitKeeper, GIT (2005),...



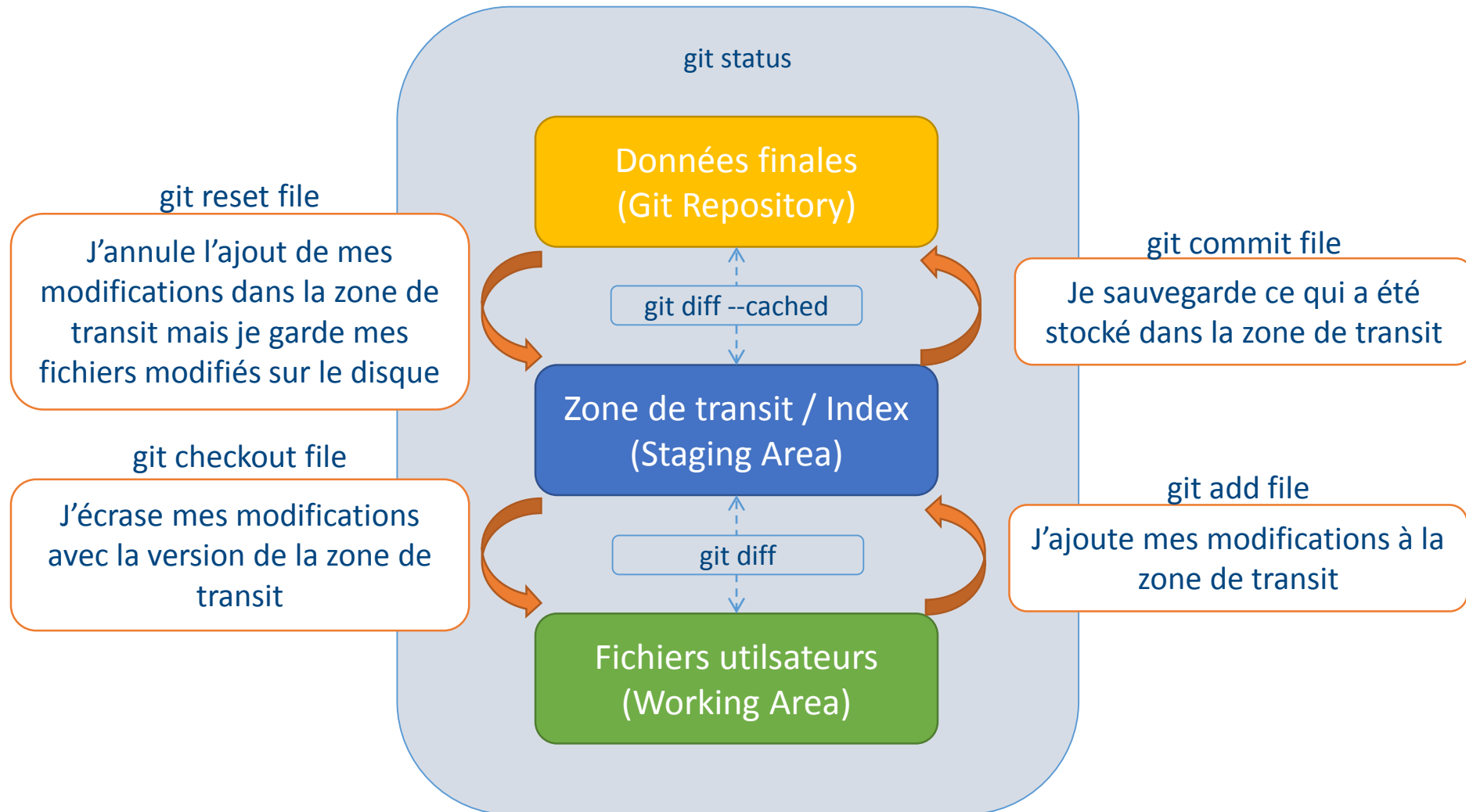
# MODIFICATION DES DONNÉES EN LOCAL (1/2)

## ➤ Repository local

- Les données de votre repository local sont modélisées par git suivant un modèle en 3 couches:



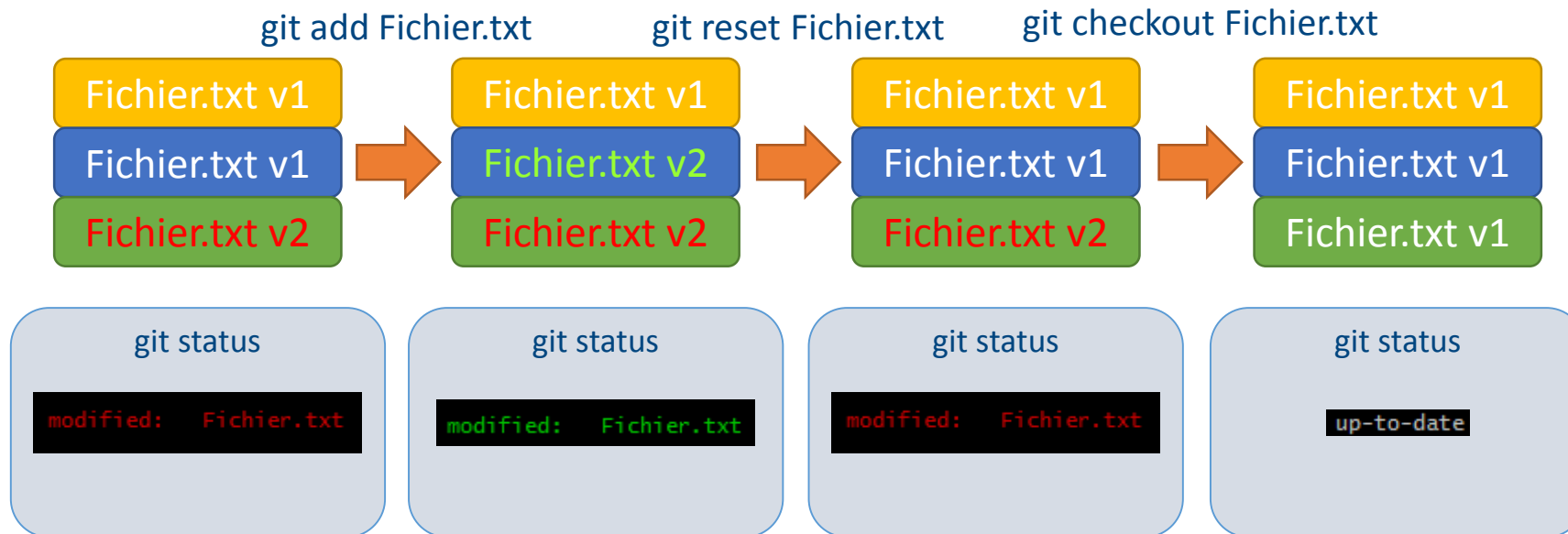
## MODIFICATION DES DONNÉES EN LOCAL (2/2)





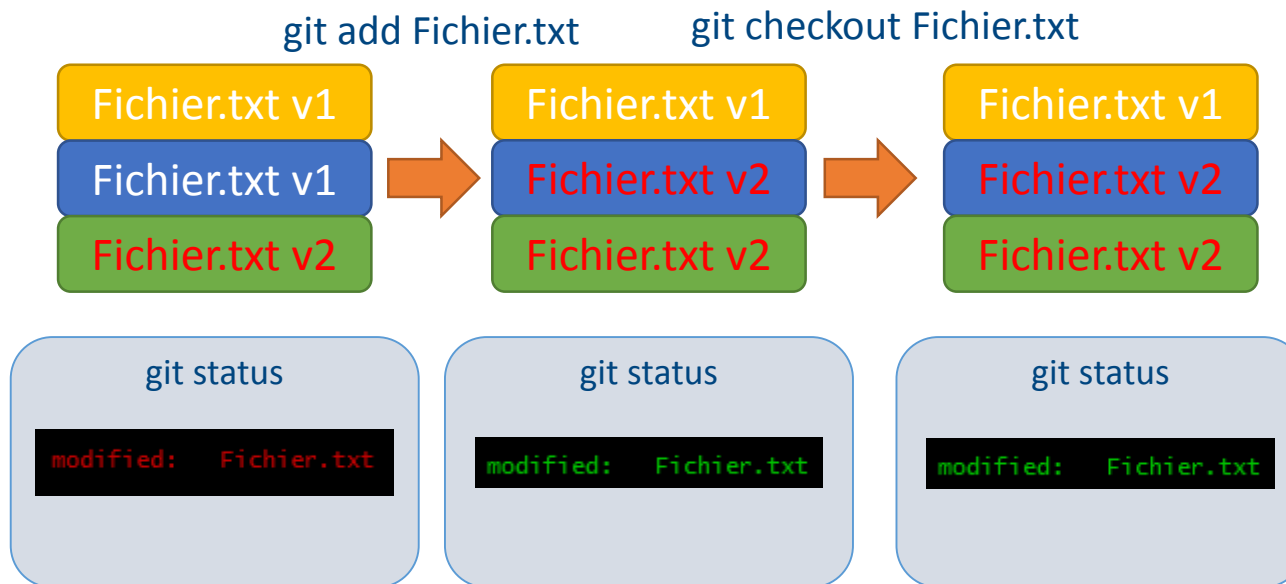
# EXEMPLES

- Que se passe-t-il si je modifie un fichier en local et que j'effectue les commandes suivantes?

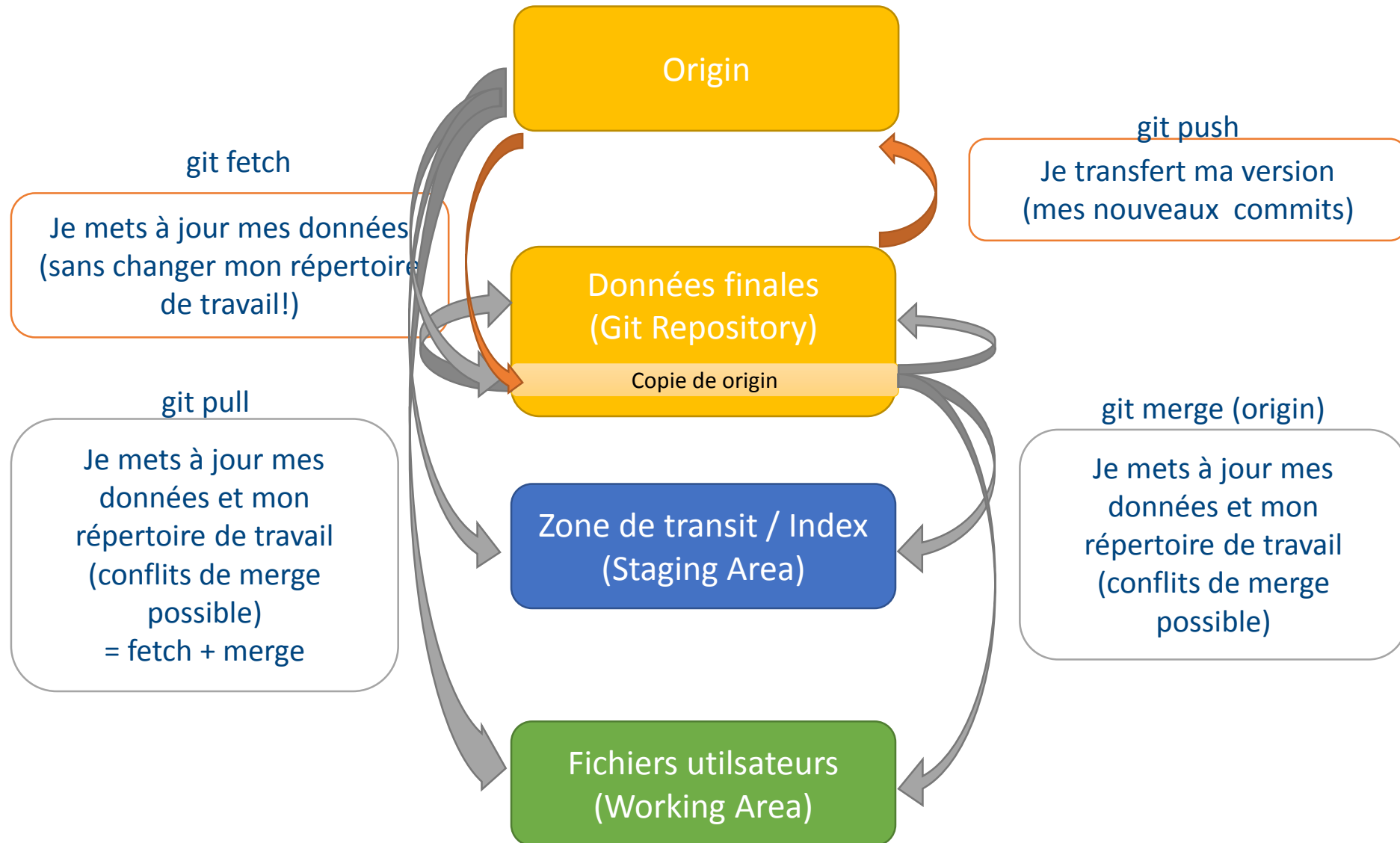


# EXEMPLES

- Que se passe-t-il si je modifie un fichier en local et que j'effectue les commandes suivantes?



# RÉCUPÉRATION DES DONNÉES DISTANTES



# Git

## Paramétrages

- Pour effectuer un commit, un username doit être fourni
  - Git permet de configurer votre profil.
    - **git config --global user.name "DOE John"**
    - **git config --global user.email [j.doe@gmail.com](mailto:j.doe@gmail.com)**

# Les commandes de base

## Initialisation

- Pour initialiser un projet (qui n'est pas dans Git)
  - Se placer à la racine du projet  
**\$ git init**
  - Attention : rien n'est commité, vous devez indiquer les fichiers à ajouter  
**\$ git add xxx yyy**  
**\$ git commit -m 'premier commit'**
- Pensez à ne pas inclure les dossiers / fichiers qui ne vous intéressent pas
  - Par exemple le dossier target de Maven n'a rien à faire dans votre Git

# Les commandes de base

## Récupération

- Pour récupérer un projet (qui est déjà dans Git)
  - Placez vous à l'endroit où vous voulez mettre le projet  
**\$ git clone git://github.com/xxx/yyy.git *nomDeDossier***
  - Si vous n'indiquez pas de nom de dossier, il prendra le nom du projet par défaut
  - L'URL du projet peut varier en fonctions des protocoles disponibles
- On **préfèrera** la méthode du **git clone** à partir du serveur central au git init pour s'assurer d'avoir la bonne « origin »

# Les commandes de base (1)

- A tout moment il est possible de connaître le statut de nos fichiers

**\$ git status**

- Modifié dans la zone de travail

```
modified: Fichier.txt
```

- Modifié dans la zone de transit

```
modified: Fichier.txt
```

- Pour ajouter tous les fichiers créés ou modifiés au prochain commit :

**\$ git add -A**

# Les commandes de base (2)

- Pour commiter sur le dépôt git local :  
**\$ git commit -m "un commentaire utile"**
- Pour pousser ses commits de sa branche master vers le dépôt distant origin :  
**\$ git push origin master**
- Pour modifier le commentaire joint au dernier commit si il n'est pas encore mis sur le serveur distant :  
**\$ git commit -- amend -m "nouveau commentaire"**



# Les commandes de base (3)

- Pour voir l'historique des commits :

**\$ git log**

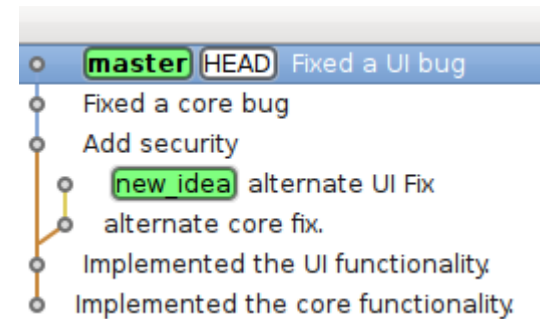
- <https://git-scm.com/docs/git-log>
- Pour comparer 2 commits
- Filtrer les informations affichées
- Mis en forme de graphe

`git log -p`

`git log --pretty=format:"%h:%s"`

`git log --graph`

```
* 47d8014 - (HEAD, feature) Merge b
| \
| * dca0784 - (origin/feature) Boulot
* | 34ae1ae - Navbar tooltips (Christ
* | e15d189 - Migrating HTML, CSS and
* | 186afc6 - README.md (Christophe P
| /
```




# Les commandes de base

## Ignorer

- Pour ignorer un dossier ou des fichiers
  - Créez un fichier `.gitignore` à la racine du dossier concerné
  - Editez le fichier

```
# Un commentaire, cette ligne est ignorée
# Pas de fichier .a
*.a
# Mais suivre lib.a malgré la règle précédente
!lib.a
# Ignorer uniquement le fichier TODO à la racine du projet
/TODO
# Ignorer tous les fichiers dans le répertoire build
build/
# Ignorer doc/notes.txt, mais pas doc/server/arch.txt
doc/*.txt
# Ignorer tous les fichiers .txt sous le répertoire doc/ (depuis 1.8.2)
doc/**/*.*txt
```

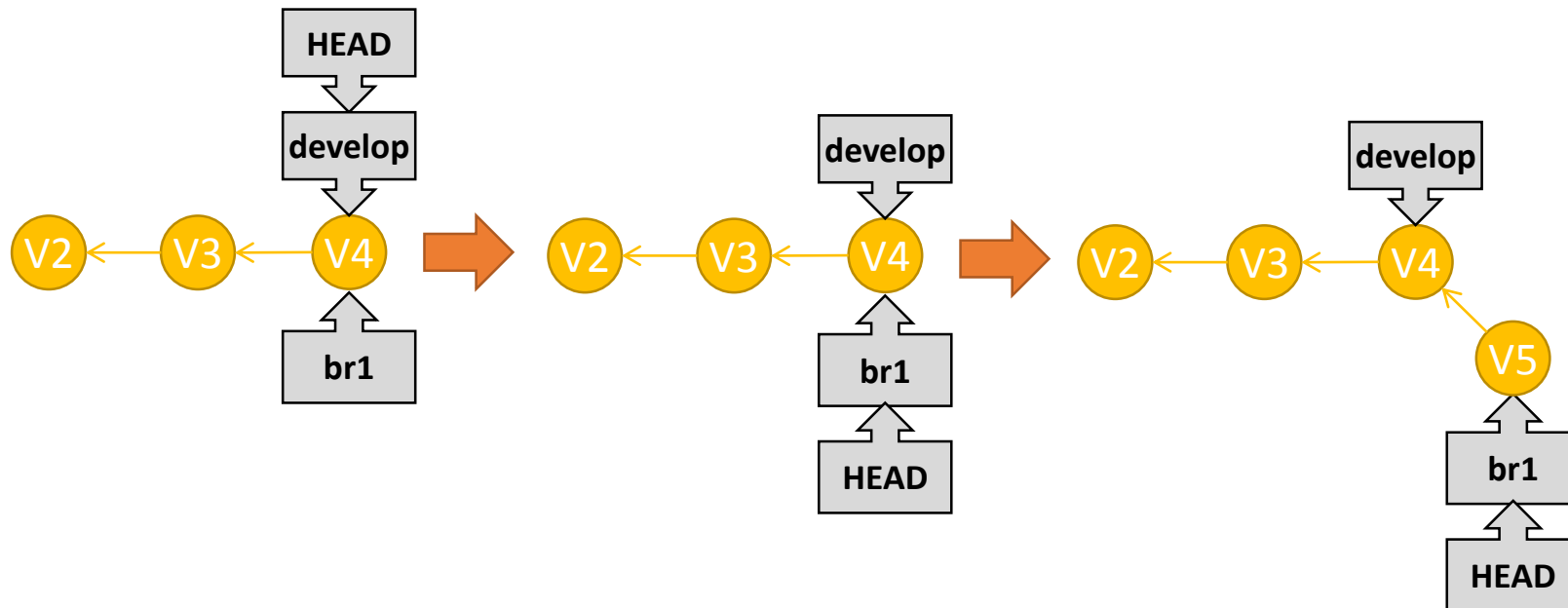
## HISTORIQUE DE COMMIT & BRANCHES (2/4)

- A chaque commande « git commit », un commit est créé dans Git. Ce commit contient:
  - Des méta données (commentaire du commit...)
  - Une référence vers un commit parent
  - Les données modifiées
- La représentation des commits à donc la forme 
- Une branche est une **étiquette** qui pointe sur un **commit**
  - Par défaut Git assigne l'étiquette master à la branche initiale (mais on travaille sur develop)
  - Une étiquette particulière HEAD est l'étiquette de référence **pour l'emplacement actuel**.
- Les branches **avancent avec les commits** sur celle-ci



# HISTORIQUE DE COMMIT & BRANCHES (3/4)

- Créer & Commit sur une branche
  - `git branch br1` Crée l'étiquette de branche
  - `git checkout br1` Déplace HEAD sur l'étiquette br1 (et met à jour la totalité du repo local en conséquence)
  - `git checkout -b br1` Equivalent aux 2 commandes précédentes
- `git commit` Nouvelle version. master ne bouge pas. HEAD et br1 avancent.



# HISTORIQUE DE COMMIT & BRANCHES (4/4)

## ➤ Passage de branche

➤ git checkout master

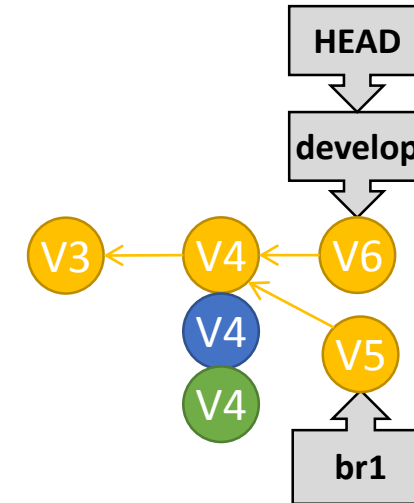
Se positionner sur master

Mise à jour du repo, HEAD se déplace

Nouvelle version

HEAD et master se déplacent

➤ git commit



## ➤ Fusion de branche

➤ git checkout develop

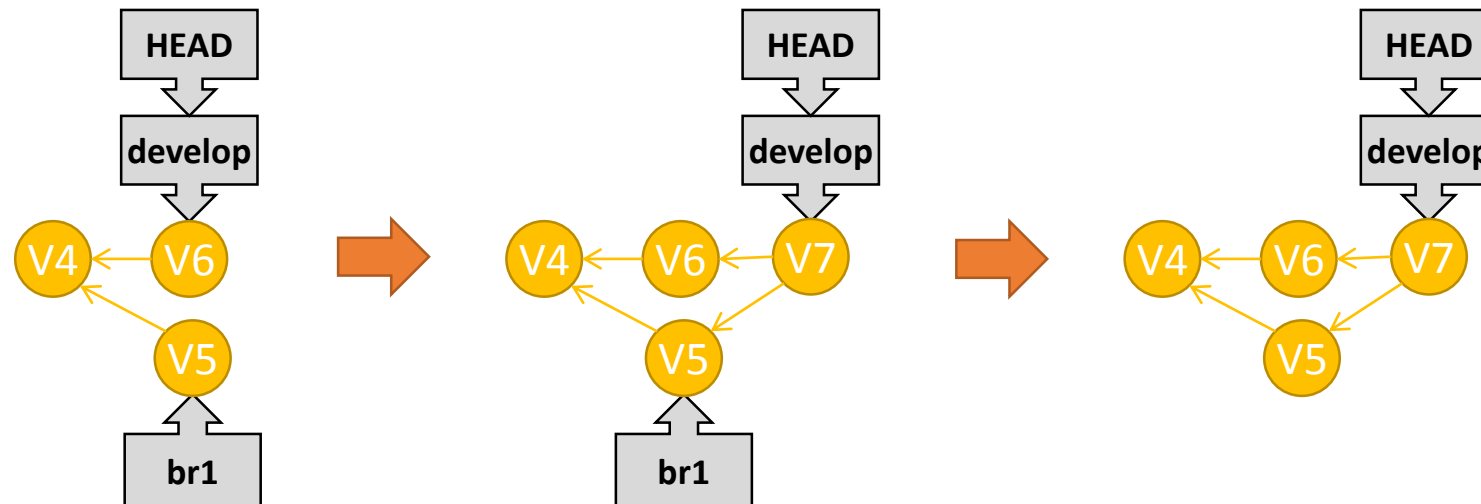
Se positionner sur develop

➤ git merge br1

Fusion sur master, br1 ne bouge pas

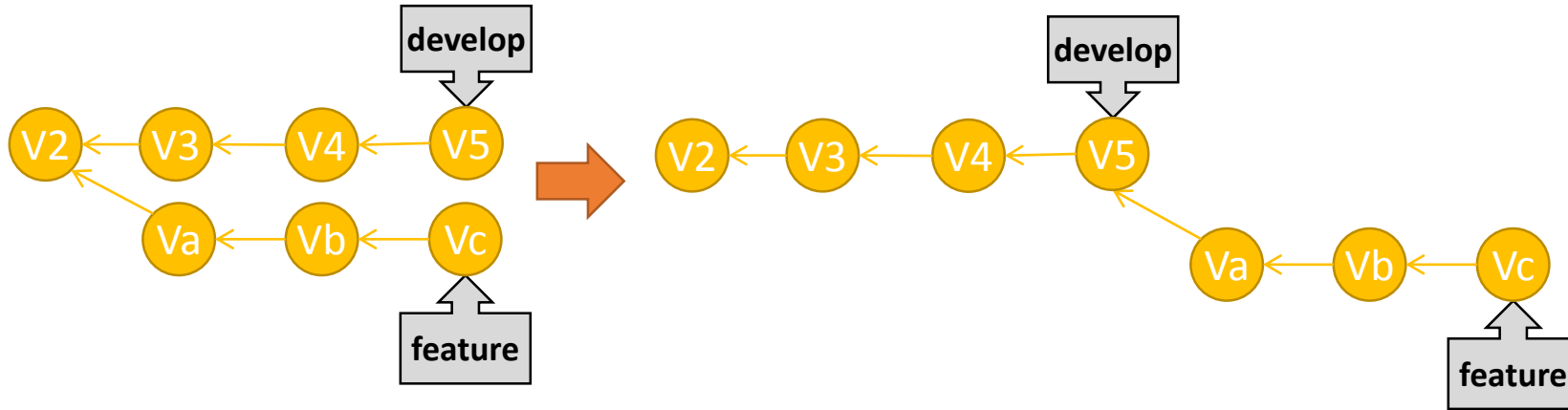
➤ (git branch -d br1)

Suppression de l'étiquette

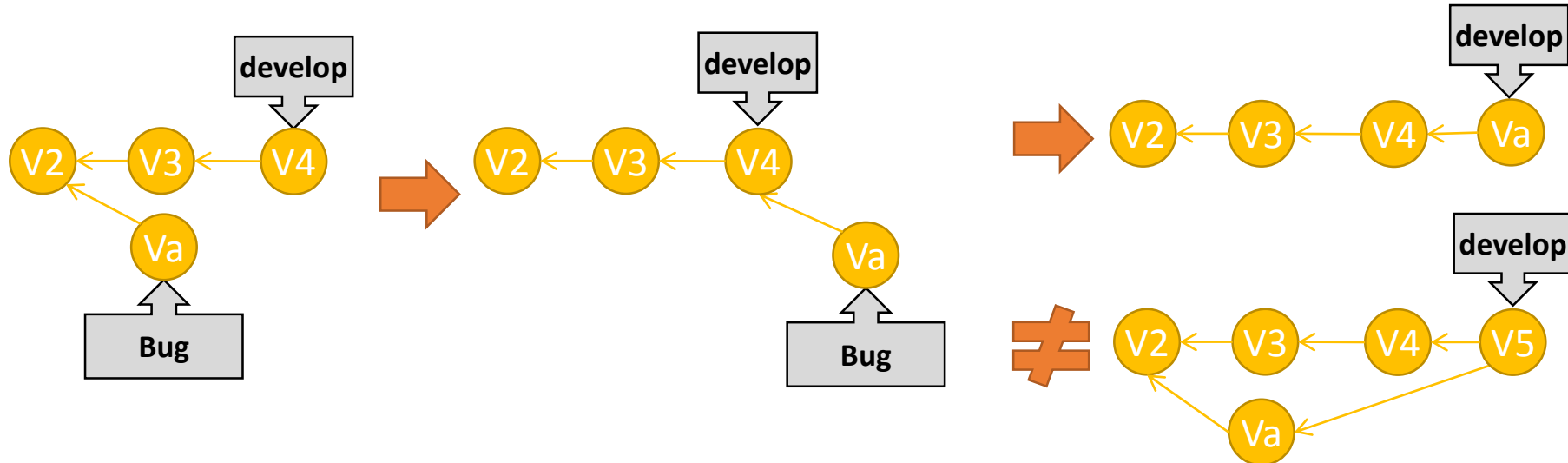


# REBASE

- Les principales utilisations de la commande `rebase`:
  - Mettre à jour une branche sur une **version plus récente**
    - `git rebase master`
    - `git rebase master feature`



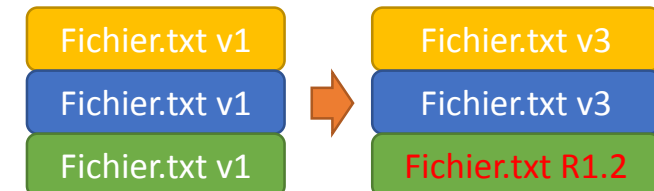
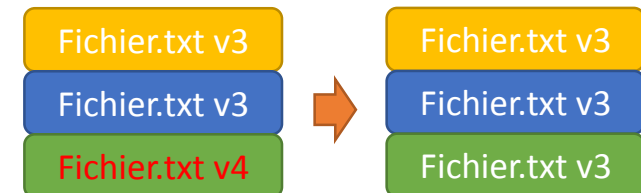
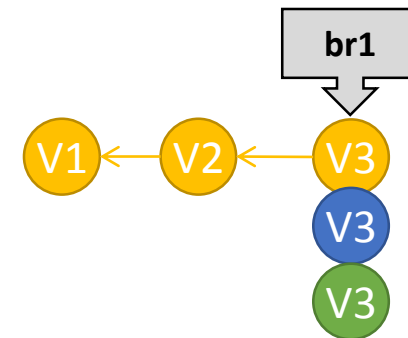
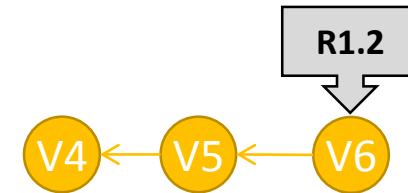
- Lisser l'historique des commits: git rebase master + git checkout master + git merge bug



# LES TAG & LES UTILISATIONS DE CHECKOUT

- Les **tag** permettent de figer des étiquettes sur des commit.
  - Identique aux branches mais **statique**
  - Servent à garder des marqueurs comme des **versions**
  - `git tag -a R1.2`

- Checkout à 3 grandes utilisations différentes:
  - Passer sur une **branche** et mettre à jour la totalité de la zone de transit et le répertoire de travail (avec merge!)
  - `git checkout br1`
  - Remettre un fichier/dossier du répertoire de travail au niveau de la **version de la zone de transit**
  - `git checkout Fichier.txt`
  - Remettre un fichier/dossier du répertoire de travail à une **version spécifique**
  - `git checkout R1.2 Fichier.txt`



# GIT

## INTRODUCTION AU GITFLOW

- 2 branches principales
  - **master** = Production
  - **develop** = Intégration
- Si tout le monde travaille sur la même branche, cela devient vite compliqué.
- Par conséquent on crée un second niveau de branche :
  - **feature** : pour les évolutions
  - **release** : pour préparer une nouvelle version de production
  - **hotfix** : pour publier rapidement une correction à partir de la branche master



# GIT

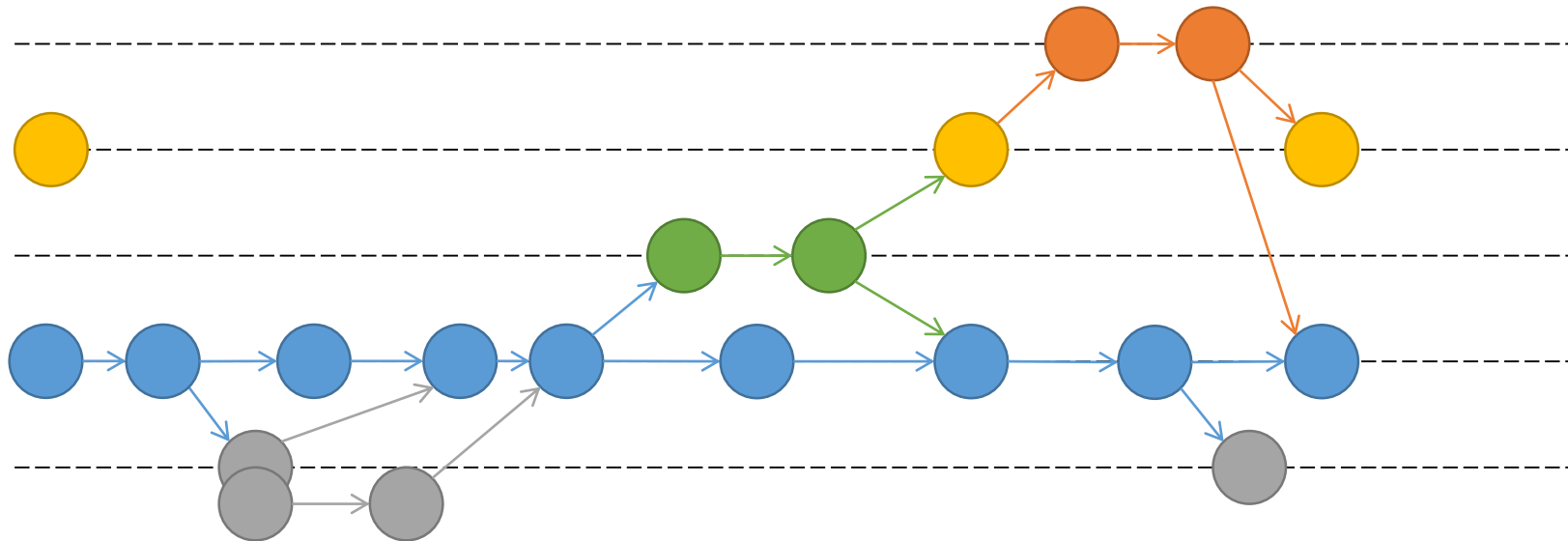
## GIT FLOW

- Le GitFlow est la base de travail pour son utilisation
- Pas une norme obligatoire mais fortement recommandé
  - **Simplification** de l'utilisation de git
  - Cohérent avec les différents environnements
  - **Interaction** avec Jenkins
  - **Automatisation** de tâches
- Le GitFlow peut être customisable si besoin et dans des cas exceptionnels.

# GIT FLOW

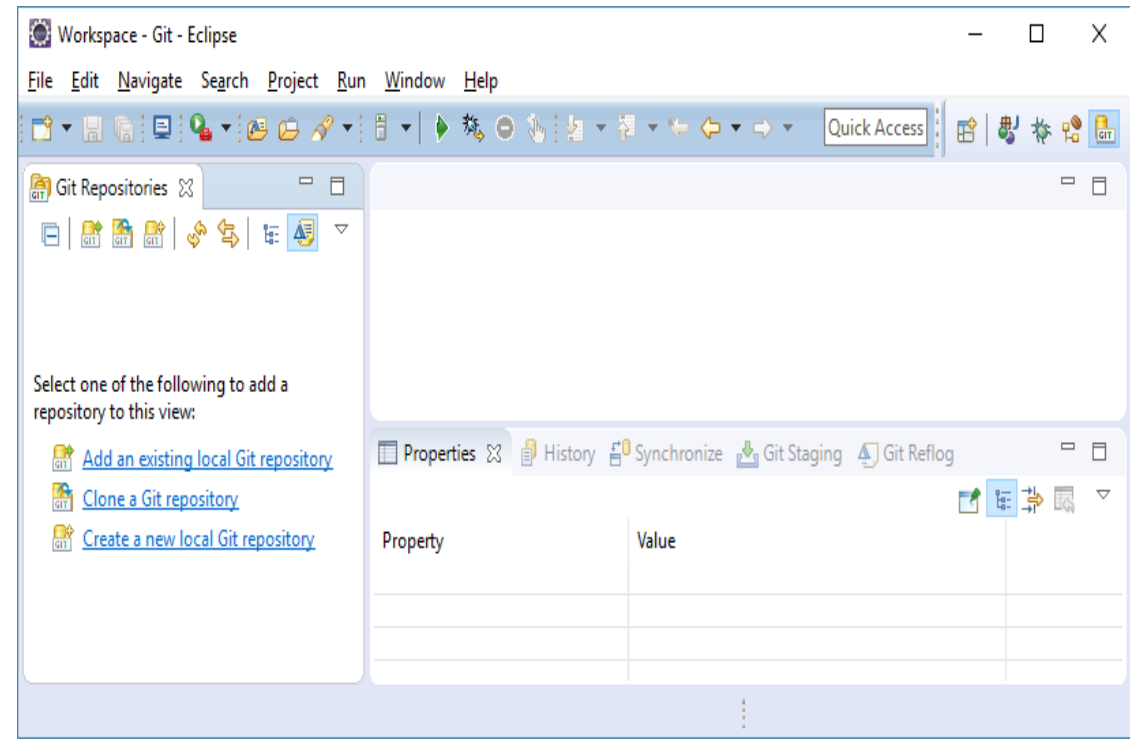
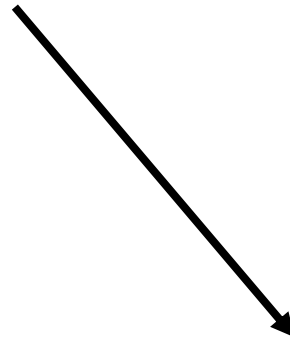
- **Le GitFlow:** 5 familles de branches
  - **feature:** Développement et gros correctifs
  - **develop:** Centralisation des développements
  - **release:** Préparation d'une version
  - **master:** Version en production
  - **hotfix:** Correctif de production

- 1) master & dev sont au même niveau
- 2) dev évolue
- 3) Des features sont créées depuis dev
- 4) Les développements convergent sur dev
- 5) Une version est créée
- 6) dev peut continuer à évoluer
- 7) La version est publiée sur dev et master
- 8) Des hotfix sont réalisés sur la prod
- 9) Dev continue d'évoluer
- 10) Les hotfix sont publiés sur master et dev



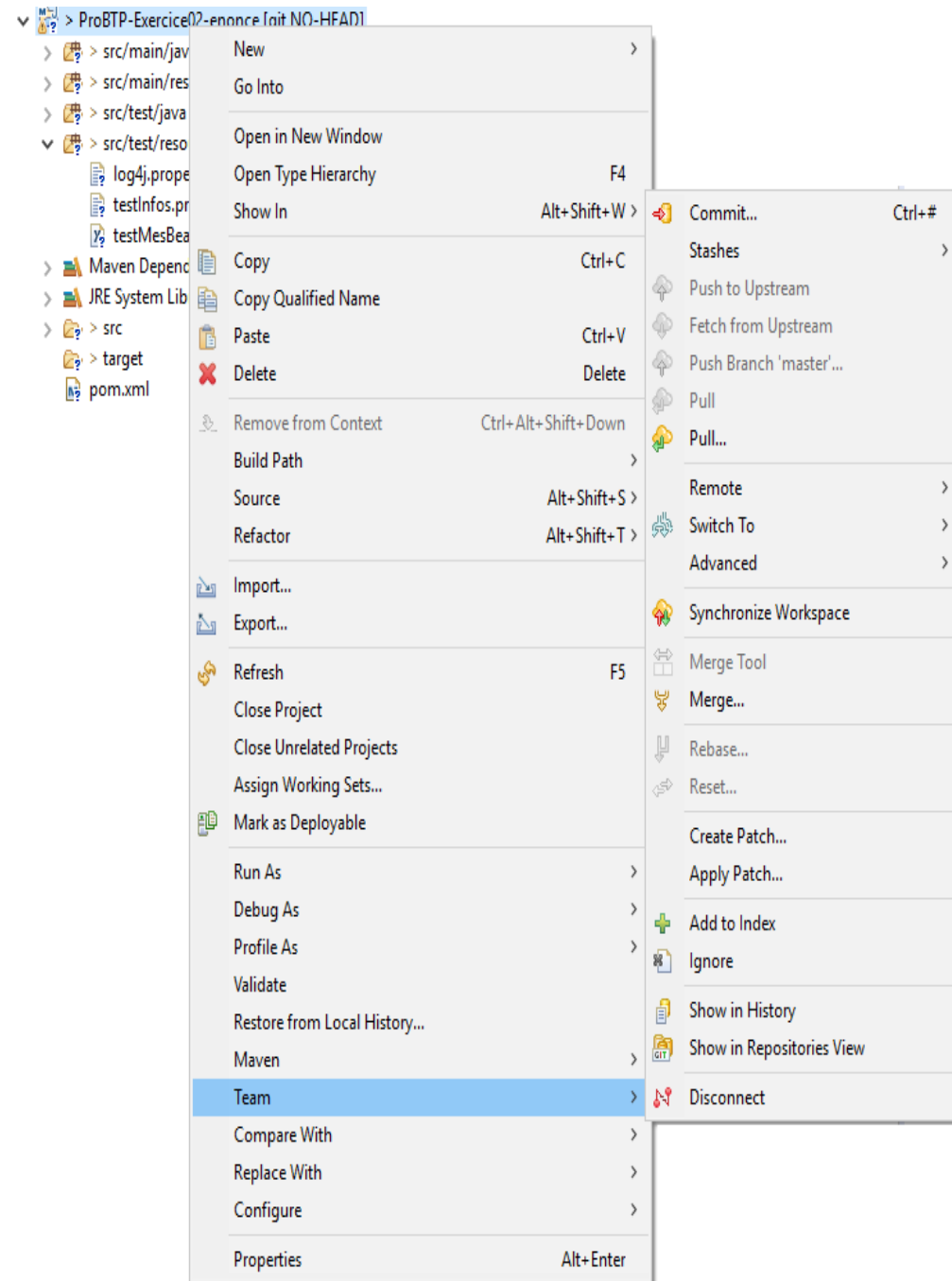
# GIT Eclipse

- Un client Git est **inclu** dans Eclipse
- Pour l'utiliser passez en perspective Git
  - Window – Perspective – Open Perspective – Other ... - Git
- De là vous pouvez cloner un Repository Git



# GIT Eclipse

- Dès que votre projet est lié à un repository Git, dans votre perspective de développement vous pourrez enclencher toutes les actions Git à travers le menu **Team**
- Plus d'informations sur <http://www.vogella.com/tutorials/EclipseGit/article.html>



# Les Bonnes pratiques

- Les bonnes pratiques :
  - Gitflow
    - Ne **jamais** toucher à la branche **master**
    - **Toujours** développer sur des branches **feature**/**<nom de la feature>**
  - Git
    - Utiliser la **zone de transit** pour **organiser** les commits selon leur impact **fonctionnel**
    - **Ne pas mettre de binaire sous Git**
    - Toujours penser à remplir le fichier **.gitignore**
    - **Respecter les nomenclatures** de projets / branches
      - **CamelCase** pour les nom de projet
      - **feature/xxx, hotfix/xxx** pour les noms de branches

# TP – Mise en place dans Git

- Se connecter à GitHub
  - Clonner un projet créé
  - Créer un projet et ajouter un fichier README via l'interface graphique
- Récupérer et mettre à jour le projet
  - Cloner le projet git sur votre poste (C:/Work)
  - Créer une branche develop puis travailler sur celle-ci
  - Créer un fichier .gitignore qui exclue les fichiers Eclipse & Maven (.project, target, etc)
  - Faites un commit & un push du projet (bien vérifier qu'aucun fichier parasite n'est présent avant de commit grâce à le git status)
  - Vérifier que le projet est bien présent sous la branche develop sous GitHub

# TP – Mise en place dans Git

- Depuis develop, créer et se positionner sur la branche feature/lion
- Ajouter un fichier test.txt avec le contenu suivant + Commit

**Le lion est un félidé. Il est considéré comme le roi des animaux.  
Il vit principalement dans des zones arides telles que la savane.**

- Depuis feature/lion, créer et se positionner sur la branche feature/chat
- Modifier le fichier test.txt + Commit

**Le chat est un félidé. Il est considéré comme le roi d'internet.  
Il vit principalement dans des zones arides telles que la savane.  
Ou alors la télévision ne nous transmet pas toujours la vérité.**

- Depuis feature/lion, créer et se positionner sur la branche feature/elephant
- Modifier le fichier test.txt + Commit

**L'éléphant est un pachyderme. Il est considéré comme le roi des animaux.  
Il aime l'eau.  
Il vit principalement dans des zones arides telles que la savane.  
Il se fait chasser pour son ivoire.**

# TP – Mise en place dans Git

- Depuis la branche git/chat, effectuer un git merge
  - Que va-t-il se passer?
- Depuis Eclipse néon, ouvrir le fichier test.txt en conflit
- Analyser & merger selon votre choix. Sauvegarde & Commit

