Git

Introduction

Qui suis-je ?

- Développeur Java
- Très vite parti dans le monde du Web et du mobile
- Passionné par toutes les nouveautés dans le domaine du développement

La gestion de versions

- **Besoin**
  - → Pouvoir restaurer une version antérieure d'un fichier

- **Solution**
  - → Les systèmes de gestion de version (VCS)

Rôles des VCS

→ Ramener un fichier à un état précédent
→ Visualiser les changements au cours du temps
→ Voir qui a modifié quoi
→ Déterminer quel changement a causé des problèmes

Les VCS locaux

- **Historiquement les premiers VCS**
  → Utilisent une base de données simple
  → Stockent les modifications d'un fichier

- **Ne répondent pas aux besoins de collaboration**

Les VCS centralisés

Ou CVCS

- **Résolvent le problème de collaboration**
  - → Un serveur centralisé stocke tous les fichiers
  - → Les clients peuvent extraire une version et soumettre des modifications
  - → Tous les clients connaissent la dernière version officielle du projet

Les VCS centralisés

- **Point unique de panne**
  - → Perte du serveur = perte de l'historique
  - → Impossibilité de sauvegarder des changements

Les VCS distribués

Ou DVCS

- **Suppression du point unique de panne**
  - → Chaque client a une copie complète du dépôt
  - → Si le serveur est perdu, n'importe quel dépôt client peut servir de serveur

- **Possibilité de collaborer sur le même projet avec plusieurs équipes en simultané**

Git

- **Initiative de Linus Torvalds**
  → Remplacer BitKeeper
  → Gérer les sources de Linux
- **Construction autour des acquis**
- **Licence GNU GPL v2**

Particularités

- **Système décentralisé (DCVS)**
  - → Aucun serveur maître n'est requis
  - → Chaque développeur a son dépôt autonome
- **Fichiers identifiés par un hash sha-1**
  - → Aucun diff n'est stocké
  - → Si un fichier est modifié, deux versions sont enregistrées

Pourquoi Git ?

- Robustesse
- Architecture distribuée
- Conception simple
- Support de milliers de branches en parallèles
- Gestion efficace des projets d’envergure

Comprendre Git

Bien comprendre

- **Mettre de côté sa vision des autres VCS**
  - → Git gère les informations très différemment
  - → L'interface semble la même, mais les concepts sont différents
  - → Comprendre les différences évite les confusions

Les versions : méthode historique

Reprise de génération en génération

- **Gestion de l'information comme**
  - → Une liste de fichiers
  - → Une liste de modifications sur chaque fichier

Les versions : méthode historique

Reprise de génération en génération

- **Résultat**
  - → Pour chaque révision d'un fichier, un delta ou diff est enregistré par rapport à la révision précédente
- **Inconvénient**
  - → Pour récupérer une révision, il faut appliquer tous les diff depuis la révision de base

Les versions : méthode Git

- **Git ne stocke aucun diff**
  - → Génération d'un instantané à chaque changement dans le projet
  - → Référence vers la version antérieure si un fichier n'est pas modifié
  - → Chaque fichier est différencié par une empreinte sha-1

Les versions : méthode Git

- **Avantages**
  - → Passage de version en version très rapide
  - → Gestion de l'intégrité grâce aux hash sha-1
  - → Un nombre de possibilités accru

Local vs distant

- **Décentralisé**
  - → Aucun dépôt maître n’est requis pour créer un nouveau dépôt
- **Mais tout en local**
  - → Chaque développeur possède une copie complète du dépôt et de son historique
  - → la majorité des commandes s’exécute en local

Les trois états d'un fichier

- **Validé**
  - → Le fichier est versionné et en sécurité
- **Modifié**
  - → Le fichier a été modifié, aucune version n'est enregistrée
- **Indexé**
  - → Les changements sont enregistrés, le fichier peut être validé

Trois premières commandes “locales”

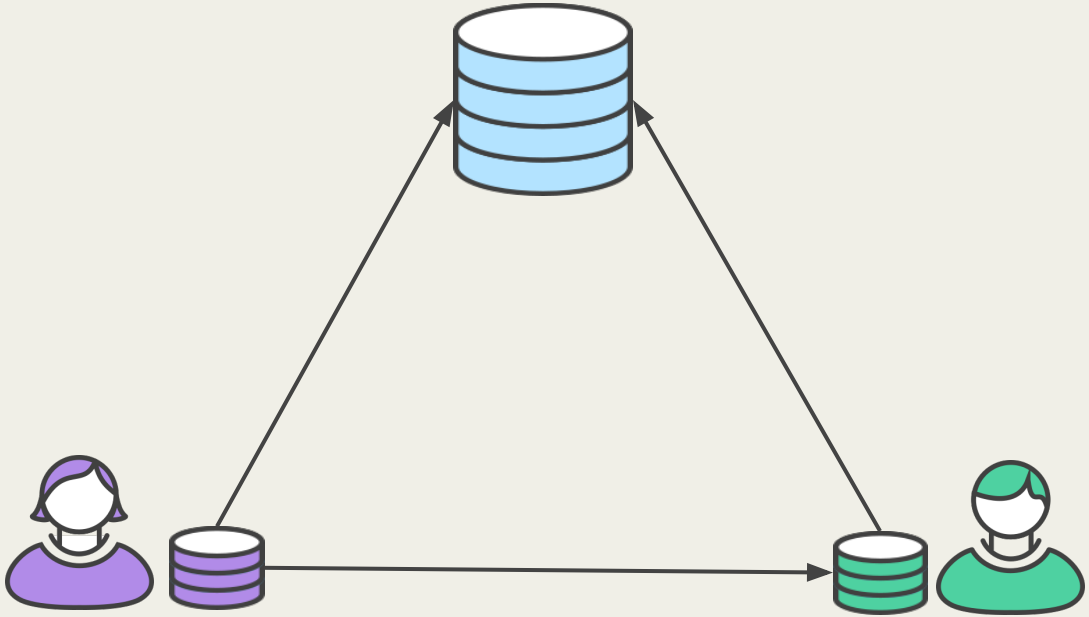
Ne pas faire de parallèle avec SVN

- **checkout**
  - → Récupère une révision précise du fichier/projet
- **add**
  - → Index des changements
- **commit**
  - → Valide les changements indexés
  - → Crée une nouvelle révision

Les dépôts distants

- **Un dépôt pointe vers un ou des dépôts distants**
  → Centraliser les sources
  → Collaborer
- **Le développeur choisi quand synchroniser son code avec le dépôt distant**

Local vs distant



Trois premières commandes “distantes”

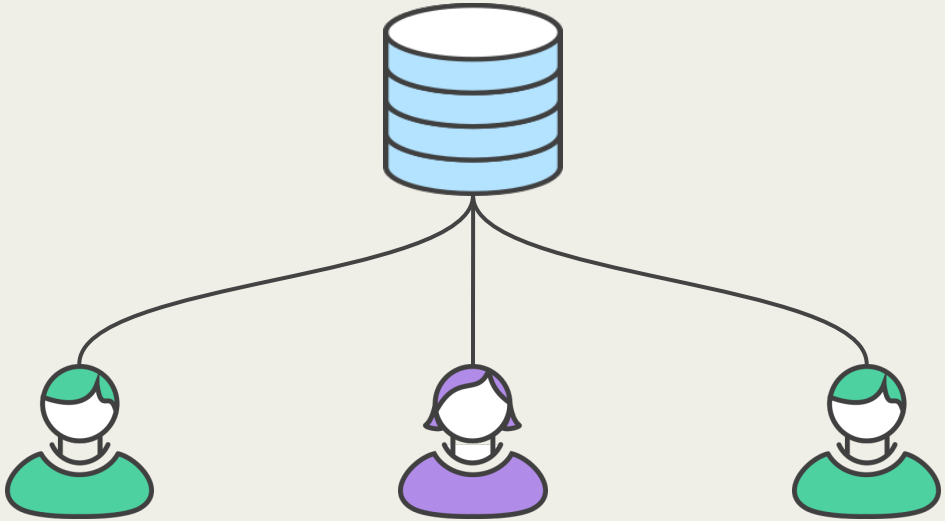
- **clone**
  - → Copie un projet depuis un dépôt distant
- **push**
  - → Envoie au dépôt distant les modifications indexées
- **fetch**
  - → Récupère les modifications présentes sur le dépôt distant

Git au quotidien

Architecture classic

Git as SVN

- **Un seul serveur sert de dépôt de référence**



Avantages

Garde le même workflow que SVN

Avec des bénéfices :

Gestion des branches avancée

Isolation de l'environnement du développeur

L'organisation

- **Le dépôt central est le projet officiel**
  - → Nommé *origin*
  - → Version "sacrée"
  - → Est le point d'entrée de tous développeurs
    - ⇀ Le clone initial
    - ⇀ Les push / pull
- **La branche principale**
  - → Nommée *master*
  - → Semblable au *trunk* de SVN

Une première notion de branche

- **Divergence dans le développement principal**

- **Souvent utilisées pour :**
  - → Développer une nouvelle fonctionnalité
  - → Corriger une version spécifique
  - → Séparer les versions stables /en test / en dev

Les tags

- **Deux types de tags**
  - → Tags légers
    - → Simple pointeur sur un commit
    - → Idéal pour un tag temporaire
  - → Tags annotés
    - → Un objet à part entière pour Git
    - → Contient une date, l'email du taggeur, son nom, …
    - → Peut être signé et vérifié par GPG

Un peu de concret

Objectifs

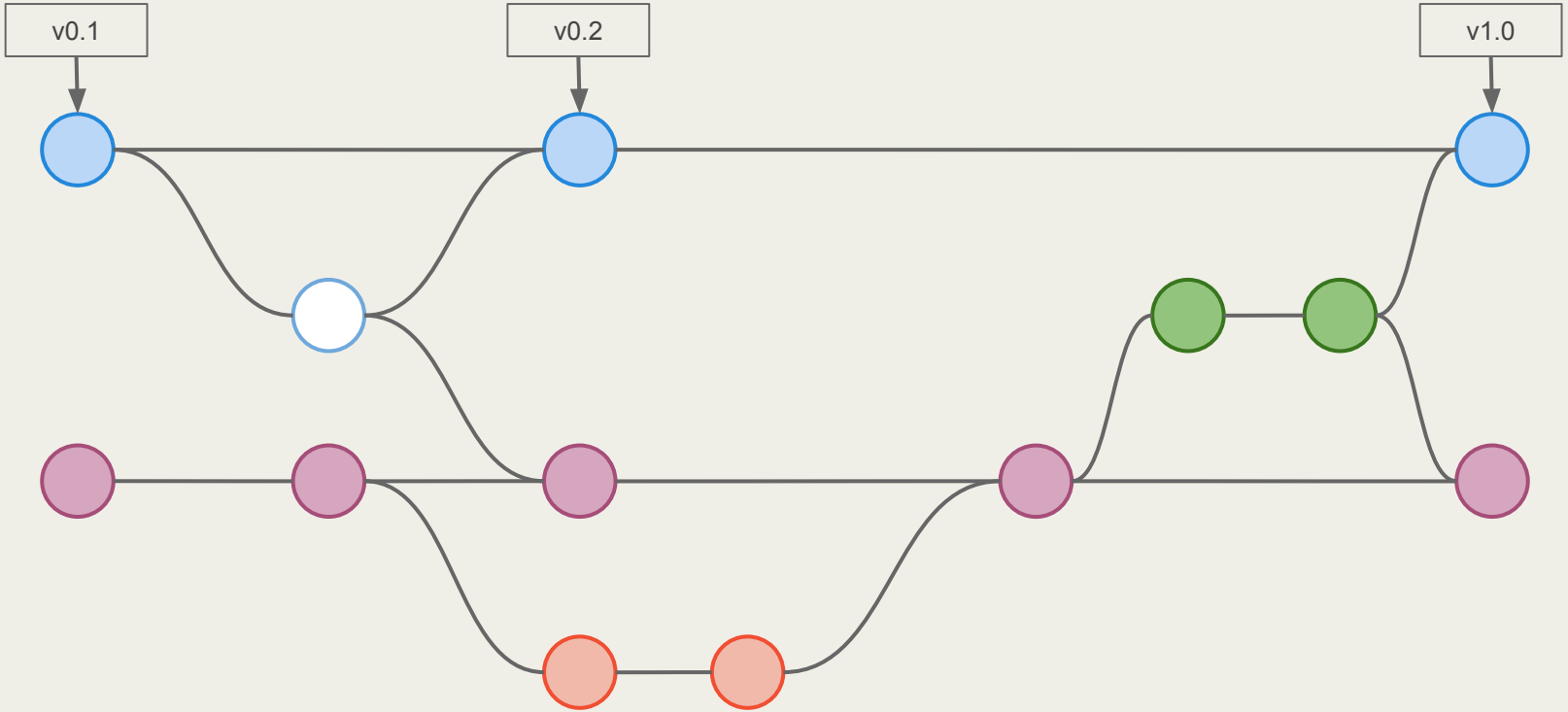
- Configurer Git
- Démarrer un dépôt Git
- Indexer des fichiers
- Ignorer des fichiers
- Supprimer des changements
- Comprendre les logs
- Ajouter un dépôt distant
- Collaborer avec un dépôt distant
- Gérer un conflit

Branches et organisation

GitWorkflow

- **Toujours un dépôt central**
  - → Communication entre les développeurs
- **Développement local et push sur le dépôt**
- **Différence :**
  - → La structure en branche du projet

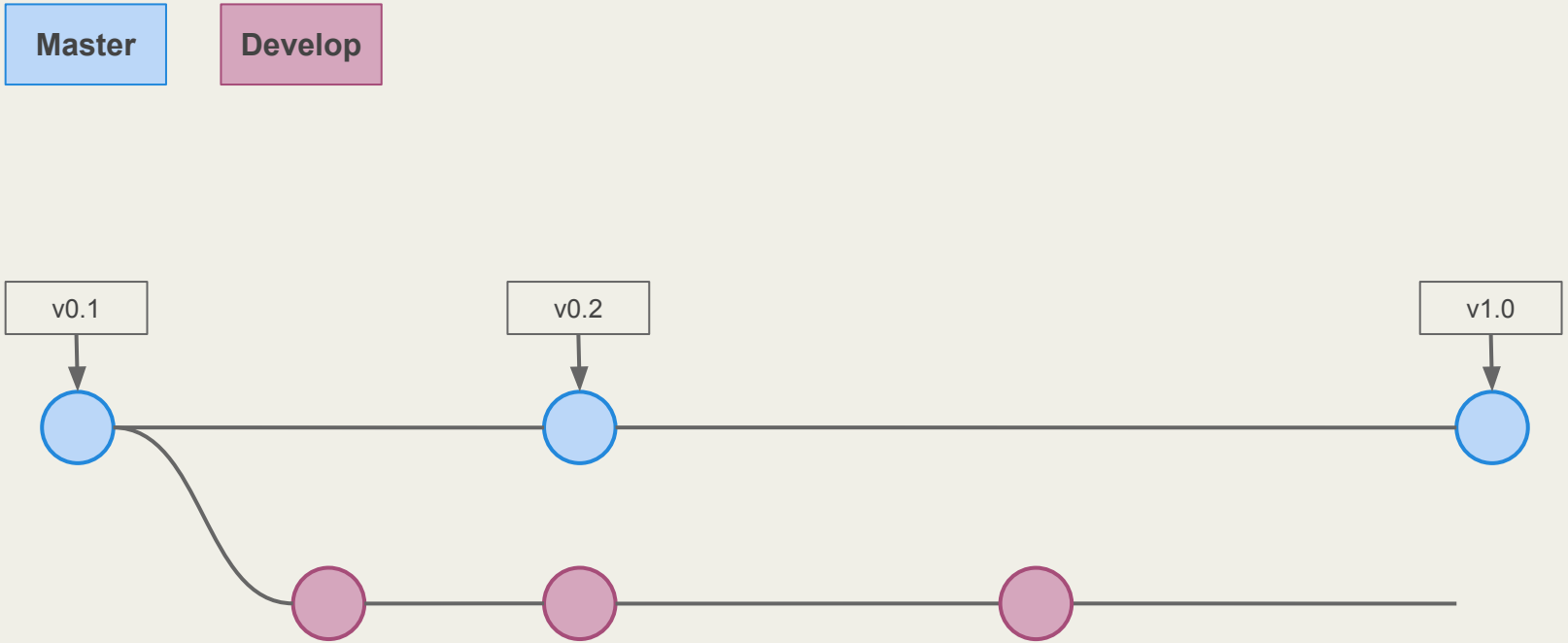
GitWorkflow



Master et Develop

- **Deux branches principales**
  - → Master
    - → Stocke les releases stables du projet
    - → Chaque commit est taggé
  - → Develop
    - → Point central du développement
    - → Sers à intégrer les modifications

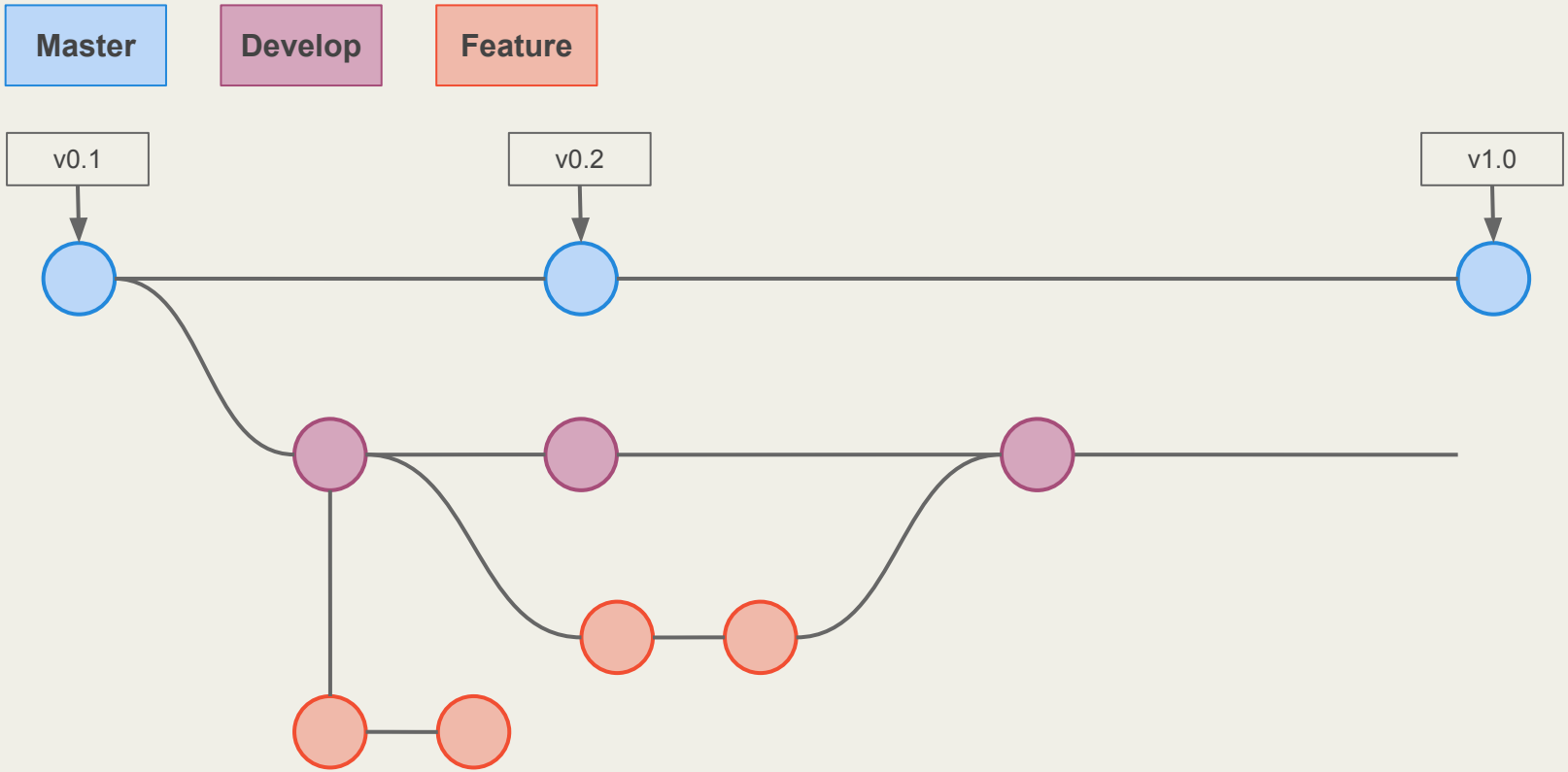
GitWorkflow



Feature

- Branche à partir de develop
- Contient une nouvelle fonctionnalité
- Mergée avec develop une fois le développement terminé
- Ne dois jamais être mergée avec master

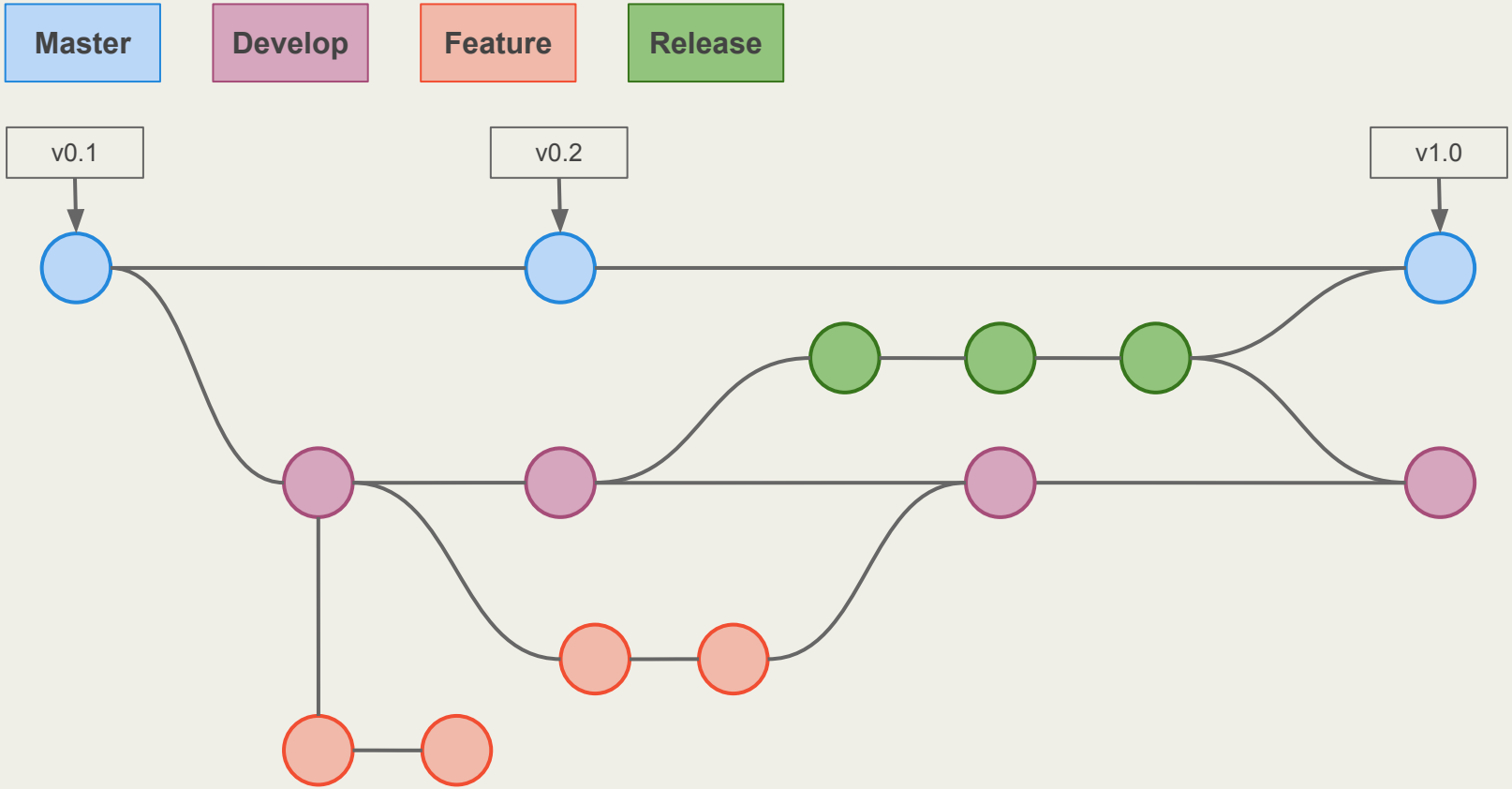
GitWorkflow



Release

- **Branche à partir de develop**
- **Aucune feature ne doit être mergée**
- **Seulement des patch de stabilité / sécurité**
- **Une fois stable**
  - → Commit sur master pour créer une release
  - → Merge avec develop pour propager les correctifs

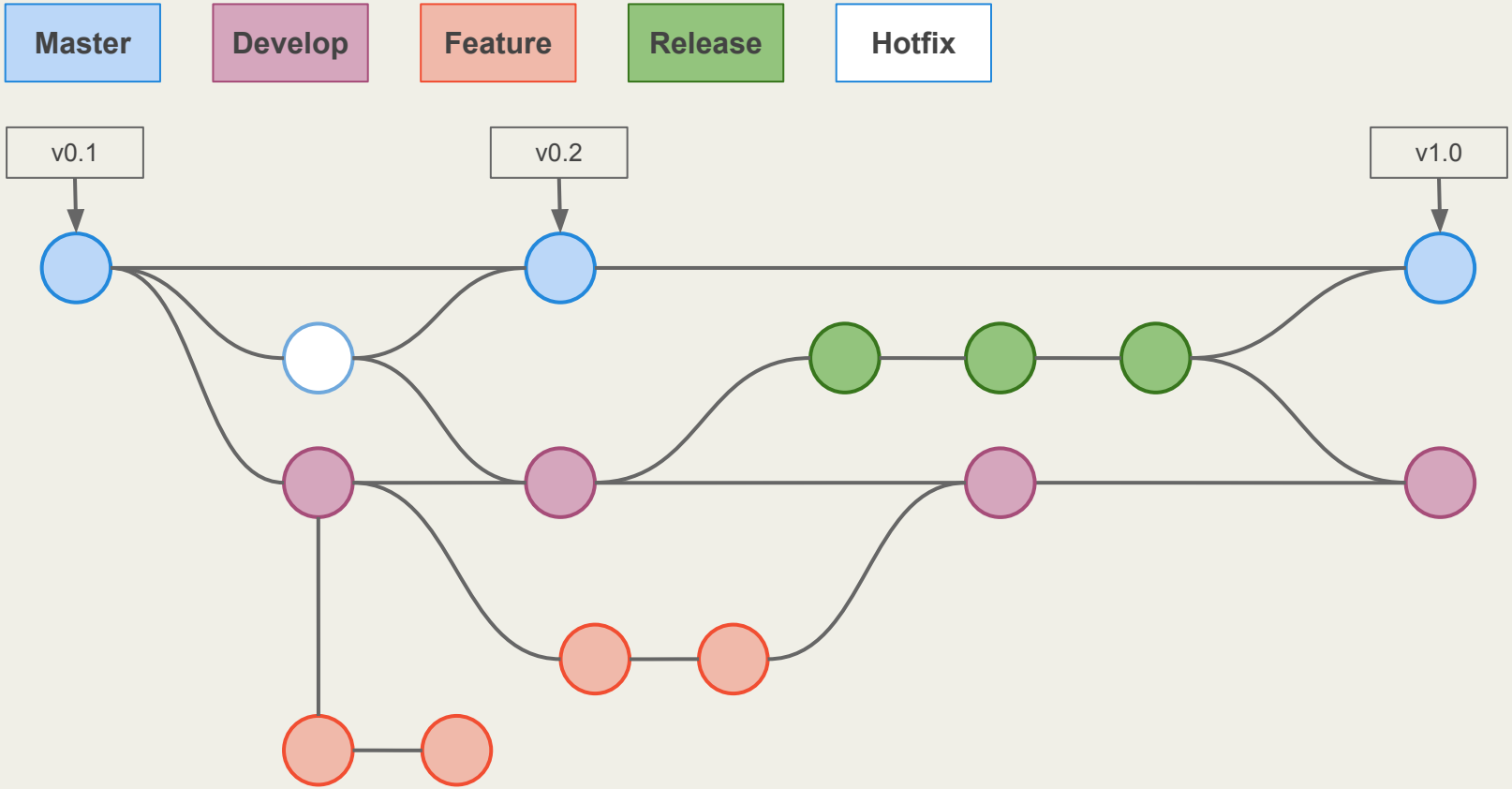
GitWorkflow



Hotfix

- **Branche à partir de master**
- **Une fois terminé**
  - → Commit sur master et incrément de la version
  - → Merge avec develop pour propager le patch

GitWorkflow



La pratique !

Objectifs

- ➔ Créer un projet
- ➔ Récupérer un projet
- ➔ Mettre en place la branche develop sur le dépôt distant
- ➔ Brancher develop pour développer une nouvelle feature
- ➔ Merger la nouvelle feature
- ➔ Créer une branche Release
- ➔ Créer une nouvelle release de l’application
- ➔ Développer un hotfix sur une version antérieure

Dépendances entre projets

L'intérêt

- **Rendre ses projets plus clairs**
  - → Chaque sous projet a un but précis
- **Réutiliser du code**
  - → Entre différents projets
- **Travailler sur des sous projets en parallèle**
  - → Développement plus souple

Les submodules

- **Permet la dépendance entre projets**
  - → Un pluggin
  - → Une librairie tierce
- **Projets tiers vus comme des répertoires**
- **Les projets restent isolés dans leurs dépôts**

Les submodules

Attention !

- Les submodules ne sont pas récupérés lors d'un clone
- Les changements dans un submodule ne sont pas répercutés automatiquement
- Demande une plus grande discipline

Les subtrees

- **Plus complexe à mettre en oeuvre**
- **Plus simple à maintenir**
- **Projets tiers intégrés dans le projet**
- **Les projets ne sont pas isolés**

→ Ne convient pas aux gros projets

Conclusions

- Ne gérez les dépendances avec Git que si vous comprenez les intérêts et les contraintes

- Dans le cas contraire, l'utilisation d'un système de gestion des dépendances est préférable (Maven, NPM, Rubygem …)

Les clients graphiques

TortoiseGit

- **Gérer ses projets sans être dépendant de l'IDE**
- **Interface s'intégrant dans Windows**
  - Menu contextuel
  - Icônes dans l'explorer
- **Reprend l'interface de TortoiseSVN**

Plugin Eclipse

- **EGit**
  - → Implémenté au dessus de JGit
- **Intégré à Eclipse**
  - → Utilisation au quotidien simplifiée
  - → Merges interactif
  - → Push/Pull en quelques clicks

Les services en ligne

Liste non exhaustive

- **Github**
  - → Le plus répandu
  - → Possède son propre client graphique
- **Gitlab**
  - → Propose une offre d'hébergement
- **BitBucket**
  - → Propose des offres Git et Mercurial

GitHub

- **Axé sur l'aspect social du développement**
  - → Possibilité de suivre des personnes/projets
  - → Gestion de Wiki
- **Repose sur le principe du fork**
  - → La personne forkant devient le leader de son propre projet

GitHub “à la maison” : GitLab

Présentation

- **Gestion**
  - → Des dépôts Git
  - → Des utilisateurs et des droits
  - → Des tickets
  - → Des wiki pour chaque projet
- **Installation et maintenance simplifiées**
  - → Utilitaire gitlab-rake

Atelier

- ➔ Création d’un compte
- ➔ Création d’un groupe
- ➔ Création d’un dépôt
- ➔ Gestion des droits
- ➔ Travailler sur son dépôt
- ➔ Importer un dépôt existant

Intégration continue

Le principe

- **Tester le projet en permanence**
  - → Détecter les régressions
  - → Détecter les problèmes d'intégration
  - → Suivre l'évolution du développement
  - → Rendre disponible l'application fonctionnelle à tout moment

En pratique

- ➔ Automatiser la compilation / le déploiement
- ➔ Les développeurs push régulièrement leur code
- ➔ Rédaction de tests
- ➔ Rendre les résultats disponibles à toute l'équipe

Outils

- Jenkins / Hudson
- Apache Continuum
- Gitlab CI
- Team Foundation Server
- Strider-CD

Gitlab CI

- Intégration simplifiée des dépôts GitLab
- Interface claire et structurée
- Exécution des tests distribuée sur plusieurs machines
- Open Source

Git avancé

Debug avec bisect

- **Permet d'identifier le commit à l'origine d'un bug par recherche dichotomique**
  - → Désigne la dernière version stable connue
  - → Désigne une version buguées
  - → Parcours les commit entre les deux

Retour en arrière

Les stratégies

- **checkout**
  - → Un commit : pas de changement d'état
  - → Un fichier : passage à l'état modifié
- **revert**
  - → Suppression d'un ancien commit
- **reset**
  - → Suppression de changement
  - → Peut ne pas être réversible

Mise en forme des logs

- Mise en forme
- Filtrage
- Résumé

Hooks

- Scripts qui s’exécutent après certains évènements
- Présents en local et sur le dépôt distant

Cas pratique : traquer un bug

Objectifs

- Trouver le commit responsable avec bisect
- Création d’une branche pour le correctif
- Revert d’un commit
- Création d’un hook