

Développement et contrôle de logiciel



Rédacteur : HAJIBA IFRAH

Objectifs pédagogiques

Introduction à l'Ingénierie Logicielle

- ✓ Comprendre les concepts fondamentaux de l'ingénierie logicielle et son importance dans le développement de logiciels.
- ✓ Identifier les principales méthodologies de développement (Cycle en V, Agile, DevOps).
- ✓ Expliquer les bonnes pratiques de conception et d'architecture logicielle (modularité, maintenabilité, principes SOLID).



Objectifs pédagogiques

Gestion du Code Source avec Git

- ✓ Comprendre le fonctionnement de Git et son utilité dans la gestion de versions.
- ✓ Maîtriser les commandes Git de base (commit, branch).
- ✓ Utiliser GitHub/GitLab pour collaborer en équipe et gérer des dépôts distants.



Objectifs pédagogiques

Automatisation des Tâches avec Apache Ant

- ✓ Comprendre le rôle de l'automatisation dans le développement logiciel.
- ✓ Installer et configurer Apache Ant pour la compilation et la gestion des tâches.
- ✓ Rédiger des scripts Ant pour automatiser le build, le test et le déploiement.



Objectifs pédagogiques

Gestion de Projet avec Maven

- ✓ Comprendre l'architecture et le fonctionnement de Maven.
- ✓ Utiliser le fichier pom.xml pour configurer les dépendances et les plugins.
- ✓ Gérer les phases du cycle de vie d'un projet avec Maven (clean, compile, test, package, install, deploy).



Objectifs pédagogiques

Intégration Continue et Déploiement Continu (CI/CD) avec Jenkins

- ✓ Expliquer les principes de l'Intégration Continue (CI) et du Déploiement Continu (CD).
- ✓ Installer, configurer et utiliser Jenkins pour automatiser les builds et les tests.
- ✓ Créer des pipelines CI/CD avec Jenkinsfile et Groovy.
- ✓ Intégrer Jenkins avec Git, Maven et d'autres outils DevOps.
- ✓ Superviser et maintenir une chaîne CI/CD pour optimiser le développement logiciel.

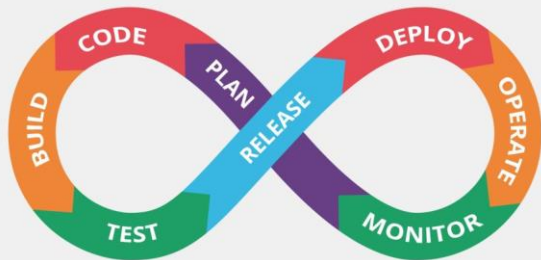


Jenkins

Objectifs pédagogiques

🔧 Compétences Attestées à la Fin du Cours

- 🎯 Capacité à gérer efficacement le code source avec **Git** et GitHub/GitLab.
- 🎯 Automatisation des processus de build avec **Apache Ant** et **Maven**.
- 🎯 Mise en place et gestion d'une **chaîne d'intégration et de déploiement continu (CI/CD)** avec **Jenkins**.
- 🎯 Compréhension approfondie des **bonnes pratiques d'ingénierie logicielle** et des outils DevOps.



Introduction à l'Ingénierie Logicielle



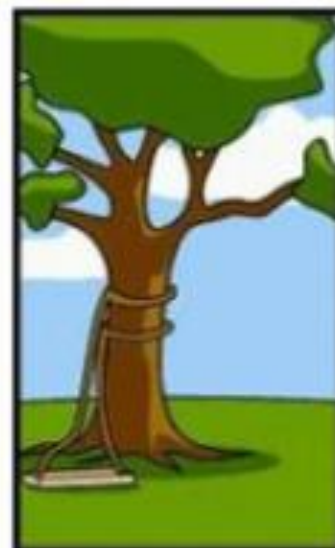
Comment le client
a exprimé son besoin



Comment le chef de
projet l'a compris



Comment l'ingénieur
l'a conçu



Comment le
programmeur l'a écrit



Comment le responsable
des ventes l'a décrit



Comment le projet
a été documenté



Ce qui a finalement
été installé



Comment le client
a été facturé



Comment la hotline
répond aux demandes



Ce dont le client avait
réellement besoin

Définition de l'ingénierie Logicielle

L'**ingénierie logicielle** est une **discipline informatique** qui applique des **principes d'ingénierie** au développement de logiciels. Son objectif est de concevoir, développer, tester, déployer et maintenir des logiciels de manière **efficace, fiable et évolutive**.

Définition formelle (IEEE - Institute of Electrical and Electronics Engineers) :

"L'ingénierie logicielle est l'application d'une approche systématique, disciplinée et quantifiable au développement, à l'exploitation et à la maintenance des logiciels."



Les Objectifs de l'Ingénierie Logicielle

L'ingénierie logicielle vise à produire des logiciels qui sont :

- ✓ **Fiables** : Peu de bugs, robustesse face aux erreurs.
- ✓ **Scalables** : Capables de s'adapter à une augmentation des utilisateurs.
- ✓ **Faciles à maintenir** : Code compréhensible et bien structuré.
- ✓ **Optimisés** : Performants en termes de vitesse et de consommation de ressources.
- ✓ **Sécurisés** : Protection contre les cyberattaques et les vulnérabilités.



Pourquoi l'Ingénierie Logicielle est-elle Importante ?

Avec la **complexité croissante** des logiciels modernes, une approche structurée est indispensable pour garantir **qualité, sécurité et performance**.

Elle permet de :

- ✓ Réduire les **coûts** et les **délais** de développement.
- ✓ Assurer une **meilleure organisation** et gestion des projets.
- ✓ Améliorer la **collaboration** entre développeurs, testeurs et chefs de projet.
- ✓ Favoriser l'**innovation** en intégrant de nouvelles technologies.



Exemples d'Applications

L'ingénierie logicielle s'applique à de nombreux domaines, comme :

- 🖥️ **Logiciels d'entreprise** (ERP, CRM, SaaS)
- 📱 **Applications mobiles et web** (réseaux sociaux, e-commerce)
- 📄 **Systemes embarqués** (voitures autonomes, objets connectés)
- 🎮 **Jeux vidéo** (moteurs graphiques, IA des jeux)
- 🔬 **Logiciels scientifiques et médicaux** (analyse d'images médicales, simulations)



Les Étapes Clés du Développement Logiciel

L'ingénierie logicielle suit généralement un processus appelé **SDLC** (**Software Development Life Cycle**) :

Analyse des besoins : Comprendre et définir ce que le logiciel doit faire.

Conception : Définir l'architecture et l'organisation du logiciel.

Implémentation : Écriture du code et intégration des fonctionnalités.

Tests : Vérification et validation du bon fonctionnement du logiciel.

Déploiement : Mise en production et mise à disposition des utilisateurs.

Maintenance et amélioration : Correction de bugs et ajout de nouvelles fonctionnalités.



Les Cycles de Vie Logiciel les Plus Utilisés

Le cycle de vie d'un logiciel définit **les étapes du développement, de la conception à la maintenance**. Plusieurs modèles existent, adaptés à des contextes et besoins différents. Voici les plus utilisés :

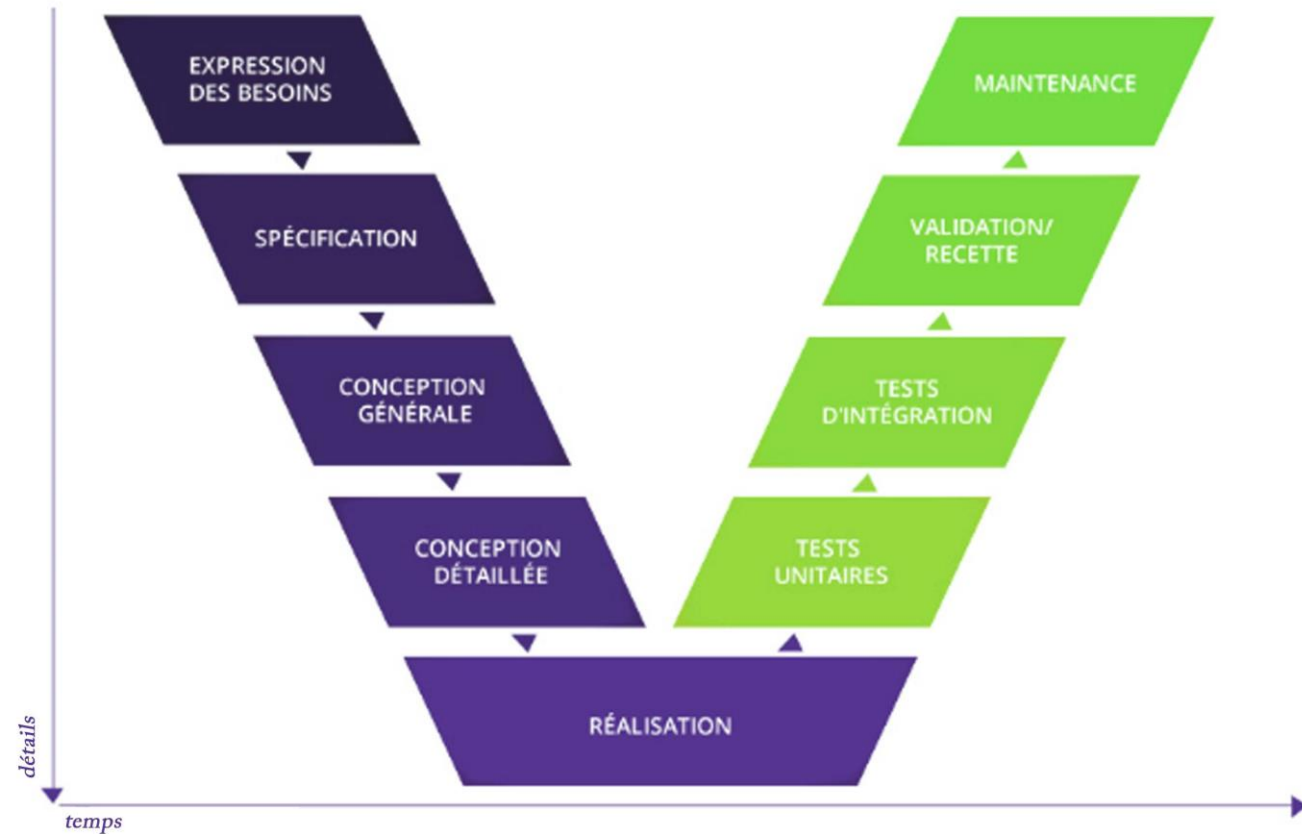


Cycle en V

Principe :

Ce modèle suit une approche **séquentielle** où chaque phase doit être terminée avant de passer à la suivante. Il met l'accent sur **la validation et la vérification**.

Étapes :



Cycle en V

✓ Avantages :

- ✓ Adapté aux **projets critiques** (ex : aérospatial, médical).
- ✓ Excellente documentation et suivi.

✗ Inconvénients :

- ✗ Rigidité : **peu de flexibilité en cas de changements.**
- ✗ Risque élevé d'erreurs non détectées avant les phases finales.

☞ Quand l'utiliser ?

- ✓ Pour des **projets bien définis et documentés.**
- ✓ Quand **les exigences sont stables** et peu sujettes aux modifications.



Cycle Agile (Scrum, Kanban...)

Principe :

Le cycle Agile repose sur une approche **itérative et incrémentale**, où le projet est découpé en **petits modules développés en sprints** (généralement de 2 à 4 semaines).

Étapes (Scrum - exemple d'Agile) :

Backlog produit 📋 (Liste des fonctionnalités)

Planification du sprint 📅 (Définition des tâches à réaliser)

Développement 💻 (Réalisation des fonctionnalités)

Tests et validation ☐ (Vérification et ajustements)

Démonstration 🗣️ (Présentation des résultats)

Rétrospective 🔄 (Amélioration continue)



Cycle Agile (Scrum, Kanban...)

✓ Avantages :

- ✓ **Flexibilité** : Adaptation rapide aux changements.
- ✓ **Livraison rapide** : Fonctionnalités utilisables dès les premiers sprints.
- ✓ **Meilleure collaboration** entre développeurs, testeurs et clients.

✗ Inconvénients :

- ✗ Exige une **forte implication des parties prenantes**.
- ✗ **Difficile à suivre** pour les grandes entreprises non familiarisées avec Agile.

☞ Quand l'utiliser ?

- ✓ Pour des **projets dynamiques** avec des exigences évolutives.
- ✓ Dans les **startups et entreprises tech** recherchant rapidité et flexibilité.



Cycle de Développement en Spirale

Principe :

Ce modèle combine le **cycle en V** et l'**approche itérative**. Chaque itération ajoute des fonctionnalités, avec une forte **prise en compte des risques**.

Étapes (répétées pour chaque itération) :

Identification des objectifs et des contraintes 🎯

Analyse et gestion des risques ⚠️

Développement et tests 💻

Évaluation et planification du prochain cycle 🔄



Cycle de Développement en Spirale

✓ Avantages :

- ✓ Approche flexible et incrémentale.
- ✓ Réduction des risques grâce à l'évaluation continue.
- ✓ Permet d'intégrer des changements en cours de projet.

✗ Inconvénients :

- ✗ Coût élevé en temps et en ressources.
- ✗ Complexité de gestion et de mise en œuvre.

🔑 Quand l'utiliser ?

- ✓ Pour les **projets à haut risque** (ex : sécurité informatique, fintech).
- ✓ Lorsqu'une **analyse approfondie des risques est nécessaire**.



Cycle de Vie DevOps (CI/CD)

Principe :

DevOps est une méthodologie qui met l'accent sur l'**automatisation**, l'**intégration continue** et la **collaboration** entre les équipes de développement et d'exploitation.

Étapes du cycle DevOps :

Planification 📅 (Définition des objectifs et des user stories)

Développement 💻 (Codage des fonctionnalités)

Build & Tests automatisés ↻ (Intégration continue - CI)

Déploiement automatisé 🚀 (Déploiement continu - CD)

Surveillance et Feedback 📊 (Logs, monitoring, correction des bugs)



Cycle de Vie DevOps (CI/CD)

✓ Avantages :

- ✓ **Déploiements rapides et fréquents.**
- ✓ **Moins d'erreurs** grâce à l'automatisation des tests.
- ✓ **Amélioration continue** grâce au monitoring et aux feedbacks.

✗ Inconvénients :

- ✗ **Exige des outils avancés** (Docker, Kubernetes, Jenkins, GitLab CI/CD).
- ✗ **Courbe d'apprentissage importante** pour les équipes non formées à DevOps.

☞ Quand l'utiliser ?

- ✓ Pour les **applications web & cloud** nécessitant **des mises à jour fréquentes.**
- ✓ Pour les **équipes DevOps** souhaitant optimiser leur workflow.



Cycle de Vie en Cascade

Principe :

Le modèle **en cascade (Waterfall)** est une approche **linéaire et séquentielle**, où chaque phase doit être complètement terminée avant de passer à la suivante.

Étapes :

Analyse des besoins

Conception

Implémentation Tests Déploiement

Maintenance



Cycle de Vie en Cascade

✓ Avantages :

- ✓ Clarté et documentation complète.
- ✓ Facile à gérer pour des projets bien définis.

✗ Inconvénients :

- ✗ Aucune flexibilité pour les modifications.
- ✗ Les erreurs ne sont découvertes qu'en fin de projet.

☞ Quand l'utiliser ?

- ✓ Pour les **petits projets stables et bien définis**.
- ✓ Pour les **applications où la documentation est primordiale**.



Comparaison des Cycles de Vie

Modèle	Adaptabilité	Documentation	Gestion des Risques	Coût
Cycle en V	Faible	Élevée	Moyenne	Moyenne
Agile	Très élevée	Faible	Faible	Bas
Spirale	Moyenne	Élevée	Très élevée	Élevée
DevOps	Très élevée	Moyenne	Élevée	Variable
Cascade	Très faible	Très élevée	Faible	Moyenne



Les Cycles de Vie Logiciel les Plus Utilisés (Conclusion)

Le choix du cycle de vie dépend **des objectifs, des contraintes et du type de projet.**

Besoin de flexibilité ? → Agile / DevOps

Projet critique avec peu de changements ? → Cycle en V / Waterfall

Gestion des risques importante ? → Spirale

Déploiement rapide et automatisé ? → DevOps



Les Acteurs dans l'Ingénierie Logicielle

Dans un projet logiciel, plusieurs acteurs interviennent à différentes étapes du cycle de vie. Voici les rôles les plus importants et leurs responsabilités.



Les Acteurs Métier

Ces acteurs sont **les bénéficiaires du logiciel** et définissent les besoins et exigences.

◆ **Client / Commanditaire :**

Définit les objectifs et exigences du projet.

Valide le produit final.

Finance le projet.

◆ **Utilisateur final :**

Utilise le logiciel dans son activité quotidienne.

Donne des retours pour améliorer l'expérience utilisateur.



Les Acteurs Métier

◆ Chef de Produit (Product Owner en Agile) :

Définit les priorités des fonctionnalités.

Gère le backlog produit.

S'assure que le logiciel répond aux besoins des utilisateurs.



Les Acteurs Techniques (Équipe de Développement)

Ces acteurs sont responsables de la **conception, du développement et de la maintenance** du logiciel.

◆ **Architecte Logiciel :**

Conçoit l'architecture du système (modularité, performance, sécurité).

S'assure de la cohérence technique du projet.

◆ **Développeurs (Front-end / Back-end / Full Stack / Mobile / DevOps) :**

Écrivent et maintiennent le code.

Implémentent les fonctionnalités demandées.

Corrigent les bugs et optimisent la performance.



Les Acteurs Techniques (Équipe de Développement)

◆ Testeur / QA (Quality Assurance) :

Vérifie que le logiciel fonctionne sans erreur.

Automatise et exécute des tests (unitaires, d'intégration, de performance).

Rédige des rapports de tests et suit la correction des bugs.

◆ Administrateur Système / DevOps :

Gère les serveurs, l'hébergement et les bases de données.

Automatise le déploiement avec CI/CD (Jenkins, GitHub Actions...).

Supervise la sécurité et la performance du logiciel.



Les Acteurs Organisationnels

Ces acteurs assurent **la gestion et la coordination** du projet.

◆ **Chef de Projet :**

Gère les délais, le budget et les ressources.

Coordonne l'équipe et assure la communication avec les parties prenantes.

Suit l'avancement du projet et gère les risques.

◆ **Scrum Master (dans Agile/Scrum) :**

Facilite l'application de la méthodologie Agile.

Supprime les obstacles rencontrés par l'équipe.

Anime les réunions (daily stand-up, rétrospective...).

◆ **Responsable Sécurité (CISO - Chief Information Security Officer) :**

Met en place des politiques de sécurité pour protéger les données.

S'assure que le logiciel respecte les normes (RGPD, ISO 27001...).



Les Acteurs Externes

Ces acteurs apportent un support ou une expertise ponctuelle.

◆ **Consultants / Experts Techniques :**

Fournissent des conseils sur des technologies spécifiques.

Aident à résoudre des problèmes complexes.

◆ **Fournisseurs de Services Cloud (AWS, Azure, GCP...) :**

Hébergent et gèrent l'infrastructure du projet.

◆ **Communauté Open Source :**

Fournit des bibliothèques et frameworks utilisés dans le projet.

Participe aux mises à jour et corrections des outils open source.



Architecture Logicielle : Concepts et Principes Fondamentaux

L'**architecture logicielle** est l'ensemble des décisions structurantes qui définissent l'**organisation des composants d'un logiciel**, leurs interactions et leurs évolutions dans le temps.

Elle vise à garantir la **modularité**, la **maintenabilité**, la **scalabilité** et la **performance** du système.



Principes Clés de l'Architecture Logicielle

- ◆ **Modularité** : Séparation du code en composants indépendants pour faciliter la maintenance.
- ◆ **Scalabilité** : Capacité du système à gérer une augmentation de la charge.
- ◆ **Séparation des préoccupations** : Chaque module doit avoir une responsabilité claire (ex : MVC).
- ◆ **Interopérabilité** : Capacité à interagir avec d'autres systèmes via APIs.
- ◆ **Sécurité** : Protection des données et des accès au système.



Types d'Architecture Logicielle

Architecture Monolithique

Un seul **bloc de code** où toutes les fonctionnalités sont regroupées dans une même application.

✓ **Avantages :**

- ✓ Simple à développer et déployer.
- ✓ Facile à tester.

✗ **Inconvénients :**

- ✗ Peu évolutif et difficile à maintenir.
- ✗ Déploiement lourd (une modification = redéploiement total).

☞ **Utilisation :** Petites applications



Types d'Architecture Logicielle

Architecture Monolithique

Un seul **bloc de code** où toutes les fonctionnalités sont regroupées dans une même application.

✓ **Avantages :**

- ✓ Simple à développer et déployer.
- ✓ Facile à tester.

✗ **Inconvénients :**

- ✗ Peu évolutif et difficile à maintenir.
- ✗ Déploiement lourd (une modification = redéploiement total).

☞ **Utilisation :** Petites applications



Types d'Architecture Logicielle

Architecture Monolithique

Un seul **bloc de code** où toutes les fonctionnalités sont regroupées dans une même application.

✓ **Avantages :**

- ✓ Simple à développer et déployer.
- ✓ Facile à tester.

✗ **Inconvénients :**

- ✗ Peu évolutif et difficile à maintenir.
- ✗ Déploiement lourd (une modification = redéploiement total).

☞ **Utilisation :** Petites applications



Types d'Architecture Logicielle

Architecture en Couches (Layered Architecture - N-Tiers)

Le logiciel est divisé en **plusieurs couches**, chacune ayant une responsabilité bien définie.

Exemple : Architecture 3-Tiers (Modèle MVC)

Couche Présentation (Front-end) : Interface utilisateur (React, Angular, Vue.js).

Couche Métier (Back-end) : Logique de traitement (Java, Python, Node.js).

Couche Données (Base de données) : Stockage et accès aux données (MySQL, MongoDB).



Types d'Architecture Logicielle

Architecture en Couches (Layered Architecture - N-Tiers)

✓ Avantages :

- ✓ Séparation claire des responsabilités.
- ✓ Facilité de maintenance et d'évolutivité.

✗ Inconvénients :

- ✗ Peut devenir complexe avec le temps.
- ✗ Performances limitées si mal optimisée.

☞ **Utilisation** : Applications web, systèmes d'entreprise.



Types d'Architecture Logicielle

Architecture Microservices

Le logiciel est divisé en **petits services indépendants**, chacun ayant sa propre base de données et communiquant via API.

✓ Avantages :

- ✓ Haute scalabilité et flexibilité.
- ✓ Déploiement et maintenance indépendants.

✗ Inconvénients :

- ✗ Complexité de gestion des services.
- ✗ Coût élevé en ressources et infrastructure.

☞ **Utilisation** : Applications Cloud, plateformes SaaS, systèmes évolutifs.



Bonnes Pratiques en Architecture Logicielle

- ✓ Appliquer le principe **SOLID** pour un code maintenable.
- ✓ Utiliser des modèles de conception (**Design Patterns**) comme Singleton, Factory, Observer.
- ✓ Documenter l'architecture (diagrammes UML).
- ✓ Automatiser le déploiement avec CI/CD (Jenkins, GitHub Actions).
- ✓ Optimiser la sécurité (authentification, chiffrement).



Les Principes SOLID en Ingénierie Logicielle

Les principes **SOLID** sont des bonnes pratiques de conception logicielle visant à rendre le code **plus maintenable, modulaire et évolutif**. Ces principes ont été introduits par Robert C. Martin (*Uncle Bob*).

💡 **SOLID = 5 principes fondamentaux** pour une architecture bien structurée et évolutive.



1. S - Single Responsibility Principle (SRP)

Une classe doit avoir une seule responsabilité.

- ◆ **Problème** : Une classe qui gère plusieurs responsabilités devient difficile à maintenir.
- ◆ **Solution** : Chaque classe doit **faire une seule chose et bien le faire.**

.



2. O - Open/Closed Principle (OCP)

Une classe doit être ouverte à l'extension mais fermée à la modification.

◆ **Problème** : Modifier une classe existante pour ajouter de nouvelles fonctionnalités peut introduire des bugs.

◆ **Solution** : Permettre l'extension via l'héritage ou l'abstraction au lieu de modifier directement le code existant.

.



3. L - Liskov Substitution Principle (LSP)

Une classe dérivée doit pouvoir remplacer sa classe mère sans altérer le comportement du programme.

◆ **Problème** : Une sous-classe qui modifie fortement le comportement d'une classe mère peut casser l'application.

◆ **Solution** : Assurer que les classes filles respectent le contrat de la classe mère.



4. I - Interface Segregation Principle (ISP)

Une interface ne doit pas forcer une classe à implémenter des méthodes inutilisées.

◆ **Problème** : Une interface trop large impose aux classes filles d'implémenter des méthodes inutiles.

◆ **Solution** : Créer des interfaces plus spécifiques.



5. D - Dependency Inversion Principe (DIP)

Une classe ne doit pas dépendre directement d'une implémentation concrète, mais d'une abstraction.

◆ **Problème** : Une classe qui dépend directement d'une autre classe rend le code difficile à modifier.

◆ **Solution** : Utiliser des interfaces ou des abstractions pour découpler les dépendances.

.



Pourquoi Appliquer SOLID ?

- ✓ **Facilite la maintenance** (modifications plus simples).
 - ✓ **Améliore la réutilisabilité** (modularité accrue).
 - ✓ **Renforce la scalabilité** (le code évolue sans tout casser).
 - ✓ **Diminue le couplage** (dépendance réduite entre composants).
- ◆ **SOLID est une base essentielle pour écrire un code propre et évolutif !**

.



Gestion de version avec GIT



Problématiques

Perte de données : Écrasement accidentel des fichiers.

Difficulté à suivre les modifications : Pas de suivi clair des évolutions du projet.

Collaboration chaotique : Plusieurs personnes modifiant un fichier en même temps sans coordination.

Absence d'historique : Impossible de revenir à une version précédente.

Risques accrus d'erreurs : Mauvaises manipulations sans possibilité de correction facile.



Besoins

Suivi des modifications : Chaque modification est historiée.

Travail collaboratif facilité : Plusieurs développeurs peuvent travailler sur le même projet.

Possibilité de revenir en arrière : On peut restaurer une version précédente en cas d'erreur.

Meilleure organisation : Structuration des versions via des branches.

Amélioration de la qualité du code : Revue de code facilitée par l'historique et les branches dédiées.



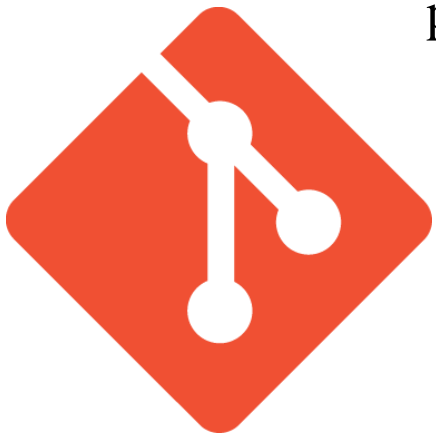
Introduction à la Gestion de Version

Qu'est-ce que la gestion de version ?

La gestion de version est un système qui permet de suivre les modifications apportées à un fichier ou un ensemble de fichiers.

Elle est essentielle dans le développement logiciel pour conserver un historique des changements et permettre le travail collaboratif.

Exemple : Sauvegarder différentes versions d'un document pour revenir à une version précédente en cas d'erreur.



Types de systèmes de gestion de version

1. Système de gestion de version centralisé

Un serveur central stocke toutes les versions des fichiers.

Les utilisateurs obtiennent une copie de travail, mais doivent se connecter au serveur pour enregistrer les modifications.

Exemple : Subversion (SVN).

Avantages :

Contrôle centralisé facilitant la gestion.

Facile à comprendre pour les petites équipes.

Inconvénients :

Dépendance à un serveur central (si le serveur tombe, plus d'accès aux versions).

Risque de perte de données en cas de panne du serveur.



Types de systèmes de gestion de version

2. Système de gestion de version distribué

Chaque utilisateur possède une copie complète du dépôt avec tout l'historique.

Les modifications peuvent être faites localement et synchronisées plus tard avec d'autres dépôts.

Exemples : Git, Mercurial.

Avantages :

Pas de point de défaillance unique (chaque copie contient l'historique complet).

Travail hors ligne possible.

Fusion des modifications plus flexible.

Inconvénients :

Peut être plus complexe à prendre en main.

Consommation de stockage plus importante (chaque utilisateur a une copie complète du dépôt).



Types de systèmes de gestion de version

2. Système de gestion de version distribué

Chaque utilisateur possède une copie complète du dépôt avec tout l'historique.

Les modifications peuvent être faites localement et synchronisées plus tard avec d'autres dépôts.

Exemples : Git, Mercurial.

Avantages :

Pas de point de défaillance unique (chaque copie contient l'historique complet).

Travail hors ligne possible.

Fusion des modifications plus flexible.

Inconvénients :

Peut être plus complexe à prendre en main.

Consommation de stockage plus importante (chaque utilisateur a une copie complète du dépôt).



Types de systèmes de gestion de version

Comparaison entre VCS et DVCS

Critère	VCS (Centralisé)	DVCS (Distribué)
Serveur requis	Oui	Non (local possible)
Travail hors ligne	Non	Oui
Performance	Moins rapide	Plus rapide
Gestion des branches	Limitée	Puissante
Serveur requis	Oui	Non (local possible)



Introduction à Git

Git est un **système de gestion de versions** distribué permettant de suivre l'historique des modifications d'un projet.

Ce système est créé par Linus Torvalds en 2005.

Utilisé pour la collaboration, il permet à plusieurs développeurs de travailler simultanément.



Fonctionnalités Clés

Suivi des Versions

Git permet de suivre chaque modification effectuée sur les fichiers d'un projet sous forme de **commits**, qui représentent des instantanés de l'état du code à un moment donné.

Fonctionnement :

Chaque commit contient un identifiant unique (SHA), un message descriptif, une date, et les informations de l'auteur.

Les changements entre les **commits** peuvent être comparés et restaurés si nécessaire.



Les développeurs peuvent naviguer dans l'historique pour retrouver une version spécifique du projet.



Fonctionnalités Clés

Commandes principales :

bash

 Copier  Modifier

```
git add <fichier>      # Ajouter un fichier à la zone de staging
git commit -m "Message" # Valider les modifications avec un message
git log                 # Afficher l'historique des commits
git diff                # Voir les différences entre versions
```



Fonctionnalités Clés

Commandes principales :

- **HEAD** est un pointeur vers le dernier commit de la branche active.
- **git checkout HEAD~1** : se déplacer vers le commit précédent.
- **git reset --soft HEAD~1** : annuler le dernier commit sans perdre les modifications.
- **git reset --hard HEAD~1** : annuler et supprimer complètement le dernier commit.



Fonctionnalités Clés

Suivi des Versions

Avantages :

- Historique clair des modifications.
- Revenir à une version antérieure en cas d'erreur.
- Traçabilité des contributions.



Fonctionnalités Clés

Développement en Branches

Les branches permettent de travailler en parallèle sur plusieurs fonctionnalités ou corrections sans affecter le code principal.

Fonctionnement :

- La branche principale (main ou master) représente la version stable du projet.
- Les développeurs créent des branches secondaires pour implémenter de nouvelles fonctionnalités, expérimenter, ou corriger des bugs.
- Une fois les modifications finalisées et testées, les branches secondaires peuvent être fusionnées avec la branche principale.





Fonctionnalités Clés

Développement en Branches

Commandes principales :

bash

 Copier  Modifier

```
git branch <nom>      # Créer une nouvelle branche
git checkout <nom>     # Passer à une branche
git branch              # Lister les branches disponibles
```



Fonctionnalités Clés

Développement en Branches

Avantages :

- Développement isolé pour chaque fonctionnalité.
- Facilite les contributions collaboratives.
- Prévention des bugs sur la branche principale.



Fonctionnalités Clés

Supprimer une branche

En local : `git branch -d nom_branche` (ou `-D` pour forcer la suppression).

Sur un dépôt distant : `git push origin --delete nom_branche`



Fonctionnalités Clés

Historique Complet

Git conserve un historique détaillé et permanent de toutes les modifications apportées au projet.

Fonctionnement :

Chaque commit est stocké avec ses métadonnées (auteur, date, message, modifications). L'historique peut être consulté sous différentes formes : graphique, simple liste, etc.




Fonctionnalités Clés

Historique Complet

Commandes principales :

bash

 Copier  Modifier

```
git log           # Historique détaillé  
git log --oneline # Affichage condensé  
git log --graph   # Vue graphique des branches
```



Fonctionnalités Clés

Historique Complet

Avantages :

- Transparence complète sur les changements apportés au projet.
- Diagnostic rapide des erreurs.
- Suivi de l'évolution du projet au fil du temps.



Fonctionnalités Clés

Fusion et Résolution de Conflits

La fusion permet d'intégrer les modifications d'une branche dans une autre. Cependant, des conflits peuvent survenir lorsque les mêmes lignes de code sont modifiées dans les deux branches.

Fonctionnement :

Git détecte automatiquement les conflits.

Les conflits doivent être résolus manuellement.

Une fois résolus, il faut valider les modifications fusionnées.




Fonctionnalités Clés

Fusion et Résolution de Conflits

Commandes principales :

bash

 Copier  Modifier

```
git merge <branche>      # Fusion d'une branche
git status                 # Vérification des conflits
git add <fichier>          # Ajout des résolutions de conflits
git commit                 # Validation après résolution
```



Fonctionnalités Clés

Fusion et Résolution de Conflits

Avantages :

- Permet la collaboration entre plusieurs développeurs.
- Conservation d'une version cohérente du code.
- Résolution fine des conflits.



Fonctionnalités Clés

Travail Local et Distant

Git permet de travailler à la fois localement sur votre machine et de synchroniser les changements avec un dépôt distant hébergé sur des plateformes comme GitHub, GitLab ou Bitbucket.

Fonctionnement :

- Les modifications sont effectuées localement et validées avec commit.
- Les changements peuvent être synchronisés avec le dépôt distant via push.
- Les mises à jour du dépôt distant sont récupérées avec pull.





Fonctionnalités Clés

Travail Local et Distant

Commandes principales :

bash

 Copier  Modifier

```
git remote add origin <url> # Associer un dépôt distant
git push                      # Envoyer les modifications
git pull                      # Récupérer les mises à jour
git clone <url>               # Cloner un dépôt distant
```



Fonctionnalités Clés

Travail Local et Distant

Avantages :

- **Marquage des Versions Importantes** : Identifier rapidement les versions stables ou milestones (v1.0).
- **Traçabilité** : Retrouver facilement l'état exact d'une version pour déploiement ou correction.
- **Documentation** : Les tags annotés permettent d'ajouter des messages descriptifs.
- **Isolation** : Facilite la création de branches à partir d'une version spécifique.
- **CI/CD** : Déclenchement automatisé de pipelines sur création de tags.
- **Partage** : Les tags sont synchronisables avec des dépôts distants (git push origin <tag>).
- **Versionnement Sémantique** : Pratique pour structurer les releases (v1.0.0).



Fonctionnalités Clés

Supprimer les modifications non enregistrées

git checkout -- nom_fichier : annule les modifications d'un fichier spécifique.

git reset --hard : annule toutes les modifications non commit.

git clean -df : supprime les fichiers non suivis.



Fonctionnalités Clés

SHA dans Git

SHA (Secure Hash Algorithm) est un identifiant unique de 40 caractères généré pour chaque commit.

Il garantit l'intégrité des données et permet de référencer précisément un commit.

Exemple : `git log --oneline` affiche les SHA des commits sous forme abrégée.



Fonctionnalités Clés

Git Tagging (Gestion des Versions)

Les **tags** marquent des points spécifiques dans l'historique, souvent pour signaler une version stable (release).

Deux types de tags :

- **Légers (lightweight)** : Simple pointeur vers un commit.
- **Annotés (annotated)** : Stockent des métadonnées (message, auteur, date).





Fonctionnalités Clés

Git Tagging (Gestion des Versions)

Commandes principales :

bash

 Copier  Modifier

```
git tag v1.0 # Créer un tag simple  
git tag -a v1.0 -m "Version stable 1.0" # Tag annoté  
git push origin v1.0 # Envoyer le tag distant
```



Fonctionnalités Clés

Envoyer un tag vers un dépôt distant

- **git tag -a v1.0 -m "Version 1.0"** : créer un tag annoté.
- **git push origin v1.0** : envoyer un tag spécifique.
- **git push origin --tags** : envoyer tous les tags locaux.



Fonctionnalités Clés

Revenir à une version précédente

git checkout commit_SHA : naviguer vers un commit précis en mode détaché.

git reset --hard commit_SHA : restaurer un commit en supprimant les modifications ultérieures.

git revert commit_SHA : créer un commit annulant les changements d'un commit précédent.



Fonctionnalités Clés

Afficher les dépôts distants associés

- **git remote -v** : affiche les dépôts distants liés.
- **git remote add origin URL** : ajoute un dépôt distant.
- **git remote remove origin** : supprime un dépôt distant.



Fonctionnalités Clés

Utilisation de git stash

git stash : stocke temporairement les modifications non commit.

git stash list : affiche la liste des stashes.

git stash pop : applique et supprime le dernier stash.

git stash apply stash@{n} : applique un stash spécifique.



Fonctionnalités Clés

Cherry-Picking dans Git

Le cherry-picking est une technique qui permet d'appliquer un commit spécifique d'une branche à une autre, sans fusionner toute la branche.

Utilisation de git cherry-pick

1. Identifier le commit à appliquer avec git log :

```
git log --oneline
```

Exemple de sortie :

```
a1b2c3d Correction d'un bug  
e4f5g6 Ajout d'une nouvelle fonctionnalité
```

2

2. Appliquer un commit spécifique :

```
git cherry-pick a1b2c3d
```

3. Gérer les conflits si nécessaire et finaliser avec :

```
git commit
```



Fonctionnalités Clés

Historique graphique

- **git log --graph --oneline --all** : affiche un historique en mode graphique simplifié.
- **gitk** : outil graphique pour visualiser l'historique.
- **git log --pretty=format:"%h - %an, %ar : %s"** : personnaliser l'affichage de l'historique.



Fonctionnalités Clés

Hooks dans Git

- Scripts exécutés automatiquement lors d'événements spécifiques (pre-commit, pre-push, etc.).
- Situés dans **.git/hooks/**, ils peuvent être personnalisés pour exécuter des tests, valider du code, etc.
- Exemple de hook pre-commit : empêcher le commit si du code non formaté est détecté.



Fonctionnalités Clés

Fichier .gitignore

Qu'est-ce que .gitignore ?

Le fichier .gitignore contient une liste de fichiers et dossiers que Git doit ignorer. Ces fichiers ne seront ni suivis ni inclus dans les commits.

Utilisation Principale :

Empêcher le suivi des fichiers sensibles, volumineux ou spécifiques à une machine locale.

•Exemples :

- Fichiers de configuration (config.json)
- Données temporaires (*.log, *.tmp)
- Dossiers spécifiques (node_modules, __pycache__)





Fonctionnalités Clés

Fichier .gitignore

Exemple de Fichier .gitignore

gitignore

 Copier  Modifier

Ignorer les fichiers Python compilés

*.pyc

__pycache__/

Ignorer les fichiers de logs

*.log

Ignorer les dossiers de dépendances Node.js

node_modules/

Ignorer les fichiers de configuration locaux

.env



Bonnes Pratiques

- Utiliser des branches pour chaque fonctionnalité (Feature Branches).
- Rédiger des messages de commit explicites.
- Faire des commits atomiques (un seul changement par commit).
- Tirer (pull) régulièrement pour éviter les conflits.
- Utiliser .gitignore pour exclure les fichiers inutiles du suivi.



Automatisation des Tâches avec Apache Ant



Définition de l'automatisation des tâches

L'**automatisation des tâches** consiste à utiliser des outils ou des logiciels pour exécuter des actions de manière automatique, sans intervention humaine. Elle permet de gagner du temps, d'éviter les erreurs et d'améliorer l'efficacité.

Exemples concrets :

Développement logiciel : compilation du code, exécution de tests, déploiement automatique.

Bureau : envoi d'e-mails programmés, génération de rapports.

Industrie : robots sur les chaînes de production.

Marketing : publications automatiques sur les réseaux sociaux.



Définition de l'automatisation des tâches

Outils courants :

Apache Ant : automatise la compilation et le déploiement d'applications.

Scripts (Python, Bash, PowerShell) : exécutent des tâches répétitives.

Logiciels spécialisés (Zapier, UiPath, Power Automate) : gèrent des flux de travail automatisés.



Définition de l'automatisation des tâches

Dans ce cours, nous allons nous concentrer sur Apache Ant, un outil puissant et largement utilisé pour l'automatisation des processus de construction (build) dans les projets Java. Nous explorerons ses fonctionnalités clés, sa syntaxe basée sur XML, et son utilisation pour compiler, tester et déployer des applications.

L'objectif est de vous permettre de maîtriser cet outil afin de gagner en efficacité dans la gestion de vos projets logiciels. Nous aborderons également des cas pratiques pour illustrer comment Apache Ant peut s'intégrer dans un workflow de développement moderne



Avantages de l'automatisation dans le développement logiciel

L'automatisation dans le développement logiciel offre de nombreux **avantages** qui améliorent la productivité, la qualité et l'efficacité des projets.

1. Gain de temps et augmentation de la productivité

- ✓ Les tâches répétitives comme la compilation, les tests et le déploiement sont exécutées automatiquement, libérant du temps pour les développeurs.
- ✓ Réduction du **temps de mise en production** grâce aux pipelines CI/CD.



Avantages de l'automatisation dans le développement logiciel

L'automatisation dans le développement logiciel offre de nombreux **avantages** qui améliorent la productivité, la qualité et l'efficacité des projets.

2. Réduction des erreurs humaines

- ✓ Moins d'erreurs dues à l'oubli ou aux fautes de frappe dans les étapes manuelles.
- ✓ Standardisation des processus, garantissant une **exécution fiable et reproductible**.

3. Amélioration de la qualité du code

- ✓ Intégration de **tests automatisés** pour détecter rapidement les bugs.
- ✓ Analyse statique du code et respect des bonnes pratiques dès le développement.



Avantages de l'automatisation dans le développement logiciel

L'automatisation dans le développement logiciel offre de nombreux **avantages** qui améliorent la productivité, la qualité et l'efficacité des projets.

4. Déploiement rapide et fiable

- ✓ Avec des outils comme **Ant**, **Maven**, **Jenkins** ou **GitHub Actions**, le build et le déploiement deviennent automatiques et reproductibles.
- ✓ Possibilité de **rollback** rapide en cas d'erreur.



Avantages de l'automatisation dans le développement logiciel

L'automatisation dans le développement logiciel offre de nombreux **avantages** qui améliorent la productivité, la qualité et l'efficacité des projets.

5. Collaboration et cohérence dans les équipes

- ✓ Un processus automatisé garantit que tous les développeurs suivent les mêmes étapes.
- ✓ Moins de dépendance aux connaissances spécifiques d'un individu, facilitant l'intégration des nouveaux arrivants.



Avantages de l'automatisation dans le développement logiciel

L'automatisation dans le développement logiciel offre de nombreux **avantages** qui améliorent la productivité, la qualité et l'efficacité des projets.

6. Meilleure gestion des versions et des dépendances

- ✓ Automatisation du contrôle des versions pour éviter les conflits.
- ✓ Outils comme **Ant**, **Maven** ou **Gradle** gèrent automatiquement les bibliothèques et mises à jour.



Avantages de l'automatisation dans le développement logiciel

L'automatisation dans le développement logiciel offre de nombreux **avantages** qui améliorent la productivité, la qualité et l'efficacité des projets.

7. Scalabilité et adaptabilité

- ✓ Les pipelines d'automatisation s'adaptent facilement à la croissance d'un projet.
- ✓ Facilite l'intégration de nouvelles technologies et outils sans perturber le workflow.



Présentation d'Apache Ant

Apache Ant (Another Neat Tool) est un outil open-source conçu pour **automatiser le processus de build des applications**, principalement en Java. Il exécute des tâches répétitives comme :

- ✓ **Compilation du code source**
- ✓ **Exécution de tests unitaires**
- ✓ **Génération de fichiers JAR, WAR, ZIP**
- ✓ **Déploiement d'applications**

Ant utilise un **fichier XML (build.xml)** pour décrire les tâches à effectuer, offrant ainsi une approche **déclarative et modulaire**.



Présentation d'Apache Ant

Pourquoi utiliser Apache Ant ?

- ✓ **Automatisation des tâches récurrentes** : plus besoin de taper des commandes manuelles.
- ✓ **Indépendance de la plateforme** : fonctionne sous Windows, Linux et macOS.
- ✓ **Flexibilité** : possibilité d'adapter les tâches en fonction des besoins spécifiques du projet.
- ✓ **Intégration facile** : compatible avec des outils comme **Jenkins, Maven et Gradle**.



Installation et Configuration d'Apache Ant

1. Prérequis

Avant d'installer Ant, assurez-vous d'avoir :

- ✓ **Java (JDK) installé** : Ant fonctionne avec Java, donc il est indispensable d'avoir un JDK installé.
- ✓ **Variable d'environnement JAVA_HOME configurée** : Ant doit savoir où trouver Java.

Vérifiez votre installation de Java avec :

```
sh

java -version
```

Si Java est installé, vous verrez un message affichant la version.



Installation et Configuration d'Apache Ant

2. Téléchargement et installation d'Ant

2.1 Télécharger Ant

Rendez-vous sur le site officiel d'Apache Ant :

↪ <https://ant.apache.org>

Téléchargez la version la plus récente sous forme d'archive **ZIP** (Windows) ou **tar.gz** (Linux/macOS).



Installation et Configuration d'Apache Ant

2. Téléchargement et installation d'Ant

2.2 Installer Ant

☞ **Windows :**

- Extrayez le fichier ZIP dans un répertoire (ex: C:\apache-ant).
- Notez le chemin d'installation (ex: C:\apache-ant\bin).



Installation et Configuration d'Apache Ant

2. Téléchargement et installation d'Ant

2.2 Installer Ant

👉 **Linux/macOS :**

- Ouvrez un terminal et extrayez l'archive

```
sh
tar -xvzf apache-ant-*.tar.gz -C /opt
```

- Renommez le dossier pour plus de simplicité

```
sh
sudo mv /opt/apache-ant-* /opt/ant
```



Installation et Configuration d'Apache Ant

2. Téléchargement et installation d'Ant

2.2 Installer Ant

👉 **Linux/macOS :**

- Ouvrez un terminal et extrayez l'archive

```
sh
tar -xvzf apache-ant-*.tar.gz -C /opt
```

- Renommez le dossier pour plus de simplicité

```
sh
sudo mv /opt/apache-ant-* /opt/ant
```



Installation et Configuration d'Apache Ant

3. Configuration des variables d'environnement

3.1 Définir ANT_HOME et ajouter Ant au PATH

☞ **Windows :**

1. Ouvrez **Paramètres système avancés > Variables d'environnement**.

2. Ajoutez une nouvelle variable système :

- **Nom** : ANT_HOME

- **Valeur** : C:\apache-ant

3. Modifiez la variable Path et ajoutez :

- C:\apache-ant\bin



Installation et Configuration d'Apache Ant

3. Configuration des variables d'environnement

3.1 Définir ANT_HOME et ajouter Ant au PATH

☞ **Linux/macOS :**

Ajoutez ces lignes à votre fichier ~/.bashrc ou ~/.zshrc :

```
sh
export ANT_HOME=/opt/ant
export PATH=$ANT_HOME/bin:$PATH
```

Puis rechargez le fichier avec :

```
sh
source ~/.bashrc # ou source ~/.zshrc
```



Installation et Configuration d'Apache Ant

4. Vérification de l'installation

Ouvrez un terminal ou un **Invite de commandes** et tapez :

```
sh
```

[Copier](#) [Modifier](#)

```
ant -version
```

Si tout est bien configuré, vous verrez un message indiquant la version d'Ant installée, par exemple :

```
scss
```

[Copier](#) [Modifier](#)

```
Apache Ant(TM) version 1.10.12 compiled on July 10 2023
```



Installation et Configuration d'Apache Ant

5. Test rapide avec un projet Ant

Ouvrez un terminal ou un **Invite de commandes** et tapez :

Créez un fichier **build.xml** dans un dossier de test et ajoutez :

```
xml
<project name="TestAnt" default="hello">
  <target name="hello">
    <echo message="Ant est bien installé !" />
  </target>
</project>
```

Exécutez la commande :

```
sh
ant
```

Si Ant fonctionne correctement, il affichera :
makefile





Installation et Configuration d'Apache Ant

5. Test rapide avec un projet Ant

Si Ant fonctionne correctement, il affichera :

makefile

 Copier  Modifier

```
Buildfile: build.xml
```

```
hello:
```

```
    [echo] Ant est bien installé !
```

```
BUILD SUCCESSFUL
```



Les Bases de build.xml dans Apache Ant

Le fichier **build.xml** est au cœur du système d'automatisation d'Apache Ant. C'est un fichier de configuration écrit en XML qui permet de définir les différentes **tâches** ou **cibles** (targets) à exécuter lors du processus de construction du projet. Ces tâches peuvent inclure la **compilation**, le **nettoyage**, le **packaging**, l'**exécution**, etc.



Structure de Base du fichier build.xml

```
xml
Copier Modifier

<?xml version="1.0" encoding="UTF-8"?>
<project name="NomDuProjet" default="targetParDefaut" basedir=".">
  <!-- Définition des propriétés -->
  <property name="source.dir" value="src"/>
  <property name="bin.dir" value="bin"/>

  <!-- Cibles ou Tâches -->
  <target name="clean">
    <!-- Action de nettoyage -->
  </target>

  <target name="compile" depends="clean">
    <!-- Action de compilation -->
  </target>

  <target name="package" depends="compile">
    <!-- Action de création de JAR -->
  </target>

  <target name="run" depends="package">
    <!-- Action d'exécution -->
  </target>
</project>
```

<?xml version="1.0" encoding="UTF-8"?> :

La déclaration XML qui définit la version du XML et l'encodage du fichier.

<project> : C'est l'élément racine qui contient toutes les informations du projet.

name : Le nom du projet (par exemple, "MonProjet").

default : Le nom de la cible par défaut qui sera exécutée si aucune cible n'est spécifiée.

basedir : Le répertoire de base du projet. Par défaut, il est défini à "." (le répertoire courant), mais il peut être personnalisé.



Les Bases de build.xml dans Apache Ant

Définition des Propriétés (<property>)

Les **propriétés** dans Ant permettent de définir des variables utilisées dans tout le fichier build.xml.

Elles sont utiles pour rendre les chemins et les configurations plus flexibles. Par exemple : xml



xml

 Copier  Modifier

```
<property name="source.dir" value="src"/>
<property name="bin.dir" value="bin"/>
<property name="dist.dir" value="dist"/>
<property name="lib.dir" value="lib"/>
```

Les propriétés peuvent être utilisées dans des tâches et des cibles :

xml

 Copier  Modifier

```
<javac srcdir="${source.dir}" destdir="${bin.dir}"/>
```



Les Bases de build.xml dans Apache Ant

Cibles (<target>)

Les **cibles** dans Ant définissent les actions spécifiques que le projet doit accomplir. Chaque cible a un **nom** et peut avoir des **dépendances** (cibles à exécuter avant cette cible). Voici un exemple de la structure d'une cible :

```
xml
<target name="nomDeCible" depends="cible1,cible2">
  <!-- Actions à exécuter -->
</target>
```

Copier Modifier

Exemple de cibles :

- **clean** : Nettoie les fichiers temporaires générés.
- **compile** : Compile le code source.
- **package** : Crée le fichier JAR.
- **run** : Exécute l'application Java.



Les Bases de build.xml dans Apache Ant

Exemple de définition de cibles :

Ici, la cible compile dépend de clean

(on nettoie d'abord), et la cible run dépend de package

(on crée le fichier JAR avant d'exécuter l'application).



xml

[Copier](#) [Modifier](#)

```
<target name="clean">
  <delete dir="${bin.dir}"/>
  <delete dir="${dist.dir}"/>
</target>

<target name="compile" depends="clean">
  <javac srcdir="${source.dir}" destdir="${bin.dir}"/>
</target>

<target name="package" depends="compile">
  <mkdir dir="${dist.dir}"/>
  <jar destfile="${dist.dir}/MonProjet.jar" basedir="${bin.dir}"/>
</target>

<target name="run" depends="package">
  <java classname="com.exemple.Main" fork="true">
    <classpath>
      <pathelement location="${bin.dir}"/>
      <pathelement location="${dist.dir}/MonProjet.jar"/>
    </classpath>
  </java>
</target>
```

Les Bases de build.xml dans Apache Ant

Tâches (<Task>)

Les tâches sont les actions spécifiques à réaliser dans une cible. Elles sont définies dans le corps de la cible. Voici quelques tâches courantes dans un projet Ant :

Quelques tâches courantes :

- **<javac>** : Compile le code Java.

xml

 Copier  Modifier

```
<javac srcdir="${source.dir}" destdir="${bin.dir}"/>
```



Les Bases de build.xml dans Apache Ant

Tâches (<Task>)

- **<delete>** : Supprime des fichiers ou des répertoires.



xml

 Copier  Modifier

```
<delete dir="${bin.dir}"/>
```

- **<mkdir>** : Crée un répertoire.

xml

 Copier  Modifier

```
<mkdir dir="${bin.dir}"/>
```



Les Bases de build.xml dans Apache Ant

Tâches (<Task>)

- **<jar>** : Crée un fichier JAR.

```
xml Copier Modifier
<jar destfile="${dist.dir}/MonProjet.jar" basedir="${bin.dir}"/>
```

- **<java>** : Exécute une classe Java.

```
xml Copier Modifier
<java classname="com.exemple.Main" fork="true">
  <classpath>
    <pathelement location="${bin.dir}"/>
  </classpath>
</java>
```



Les Bases de build.xml dans Apache Ant

Tâches (<Task>)

- **<copy>** : utiliser pour copier des fichiers ou des répertoires dans un projet Ant.

```
xml Copier Modifier  
  
<copy file="source.txt" tofile="destination.txt"/>
```

☞ Copie source.txt vers destination.txt dans le même répertoire.

Copier un répertoire entier

```
xml Copier Modifier  
  
<copy todir="backup/">  
  <fileset dir="data/" />  
</copy>
```



☞ Copie tout le contenu du dossier data/ vers backup/.

Les Bases de build.xml dans Apache Ant

Tâches (<Task>)

- **<copy>** : utiliser pour copier des fichiers ou des répertoires dans un projet Ant.

Copier des fichiers avec un filtre

```
xml
<copy todir="backup/">
  <fileset dir="data/">
    <include name="*.txt"/>
    <exclude name="secret.txt"/>
  </fileset>
</copy>
```

Copier Modifier

☞ Copie tous les fichiers .txt sauf secret.txt du dossier data/ vers backup/.



Les Bases de build.xml dans Apache Ant

Tâches (<Task>)

- **<copy>** : utiliser pour copier des fichiers ou des répertoires dans un projet Ant.

Copier et renommer un fichier

```
xml
```

```
<copy file="data/report.txt" tofile="backup/report_backup.txt"/>
```

Copier Modifier

☞ Copie report.txt et le renomme report_backup.txt dans backup/.



Les Bases de build.xml dans Apache Ant

Tâches (<Task>)

- **<import>** : Permet de centraliser des tâches communes (ex : compilation, tests)..

■ Fichier build.xml (principal) :

```
xml Copier Modifier

<project name="MainProject" default="compile" basedir=".">
  <import file="common.xml"/>

  <target name="compile">
    <echo message="Compilation du projet principal..."/>
  </target>
</project>
```

■ Fichier common.xml (inclus) :

```
xml Copier Modifier

<project name="CommonTasks">
  <target name="clean">
    <echo message="Nettoyage des fichiers..."/>
  </target>
</project>
```



Les Bases de build.xml dans Apache Ant

Tâches (<Task>)

- **<include>** :  Utilisé pour inclure une partie de fichier XML (ex. un fragment contenant des tâches)
 - ◆ Utile pour structurer des tâches réutilisables.

■ Fichier build.xml (principal) :

```
xml
Copier Modifier

<project name="MainProject" default="compile" basedir=".">
  <include file="tasks.xml"/>

  <target name="compile">
    <echo message="Compilation du projet..."/>
  </target>
</project>
```

■ Fichier tasks.xml (inclus partiellement) :

```
xml
Copier Modifier

<target name="backup">
  <echo message="Sauvegarde des fichiers..."/>
</target>
```



Les Bases de build.xml dans Apache Ant

<classpath>

L'élément <classpath> est utilisé dans Apache Ant pour **spécifier les fichiers et dossiers nécessaires à l'exécution ou à la compilation** d'un programme Java. Il définit où Ant doit chercher les bibliothèques et les classes utilisées dans un projet.

```
xml
Copier Modifier

<classpath>
  <pathelement location="libs/library1.jar"/>
  <pathelement location="libs/library2.jar"/>
</classpath>
```

◆ Ce classpath contient **deux fichiers .jar** situés dans le dossier libs/.





Les Bases de build.xml dans Apache Ant

<Path>

☞ Un <path> est une liste de fichiers ou de dossiers (souvent des .jar) que l'on peut réutiliser dans différentes tâches (compilation, exécution, tests, etc.).

xml

 Copier  Modifier

```
<path id="mon.classpath">
  <pathelement location="libs/library1.jar"/>
  <pathelement location="libs/library2.jar"/>
</path>
```

◆ Ici, on définit un chemin contenant deux fichiers .jar.



Les Bases de build.xml dans Apache Ant

<pathelement>

Le tag **<pathelement>** est utilisé dans **Apache Ant** pour ajouter un **chemin spécifique** (fichier ou dossier) à une définition de **<path>** ou **<classpath>**.

◆ Il est souvent utilisé pour inclure des **bibliothèques .jar**, des **répertoires contenant du code compilé**, ou des **emplacements spécifiques** nécessaires à l'exécution.





Les Bases de build.xml dans Apache Ant

<fileset>

L'élément <fileset> est utilisé pour **sélectionner un ensemble de fichiers** selon des critères définis (chemin, motif d'inclusion/exclusion, etc.). Il est souvent utilisé avec <copy>, <delete>, <javac>, <jar>, etc.

xml

 Copier  Modifier

```
<fileset dir="libs" includes="**/*.jar"/>
```

◆ Ici, **tous les fichiers .jar** du dossier libs (y compris ses sous-dossiers) sont sélectionnés.



Les Bases de build.xml dans Apache Ant

<fileset> vs <pathelement>

Élément	Utilisation
<fileset>	Sélectionne plusieurs fichiers selon un motif (**/*.jar, *.java, etc.).
<pathelement>	Ajoute un seul fichier ou dossier au classpath.



Les Bases de build.xml dans Apache Ant

<exec>

L'élément <exec> permet d'exécuter **une commande système ou un programme externe** depuis un script Ant. Il est utile pour lancer des scripts, des outils en ligne de commande ou d'autres programmes externes.

```
xml
<exec executable="cmd">
  <arg value="/c"/>
  <arg value="echo Hello, Ant!"/>
</exec>
```

Copier Modifier

- cmd : Lance l'invite de commande Windows.
- /c : Indique à cmd d'exécuter la commande donnée (echo Hello, Ant!) **puis de se fermer immédiatement**.
- echo Hello, Ant! : Affiche le texte "Hello, Ant!" dans la console.



Les Bases de build.xml dans Apache Ant

`<macrodef>`

Une **macro** dans Apache Ant permet de **créer des tâches personnalisées** réutilisables, semblables à des fonctions en programmation.

On utilise l'élément `<macrodef>` pour définir une macro et ses arguments.

```
xml Copier Modifier  
  
<macrodef name="hello">  
  <sequential>  
    <echo message="Bonjour, ceci est une macro Ant !"/>  
  </sequential>  
</macrodef>  
  
<target name="run">  
  <hello/>  
</target>
```



◆ La macro hello affiche un message lorsqu'elle est appelée dans la cible run.

Les Bases de build.xml dans Apache Ant

<macrodef>

Un fichier macrodefs.xml contient des macros définies avec <macrodef>. Il permet de **modulariser** un script Ant en séparant les définitions de macros du fichier build.xml.

```
xml
Copier Modifier

<?xml version="1.0" encoding="UTF-8"?>
<project name="Macros" default="default">

  <!-- Définition d'une macro pour afficher un message -->
  <macrodef name="print-message">
    <attribute name="msg"/>
    <sequential>
      <echo message="@{msg}"/>
    </sequential>
  </macrodef>

</project>
```



◆ Cette macro print-message permet d'afficher un message personnalisé.

Les Bases de build.xml dans Apache Ant

<macrodef >

Dans le fichier build.xml, on peut **importer** macrodefs.xml et utiliser ses macros :

```
xml
Copier Modifier

<?xml version="1.0" encoding="UTF-8"?>
<project name="TestProject" default="run">

  <!-- Importer le fichier macrodefs.xml -->
  <import file="macrodefs.xml"/>

  <target name="run">
    <print-message msg="Bonjour depuis build.xml !"/>
  </target>

</project>
```



◆ **L'importation de macrodefs.xml** permet d'utiliser print-message comme une tâche standard.

Les Bases de build.xml dans Apache Ant

`<macrodef >`

Avantages d'un fichier macrodefs.xml

- ✓ **Modularité** : Séparation des macros du build.xml, rendant le projet plus lisible.
- ✓ **Réutilisation** : Définition d'un ensemble de macros réutilisables dans plusieurs builds.
- ✓ **Maintenance facile** : Les modifications des macros n'impactent pas directement le build.xml.



Les Bases de build.xml dans Apache Ant

Exemple Complet de build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MonProjet" default="build" basedir=". ">

  <!-- Définition des propriétés -->
  <property name="source.dir" value="src"/>
  <property name="bin.dir" value="bin"/>
  <property name="dist.dir" value="dist"/>
  <property name="lib.dir" value="lib"/>
  <property name="main.class" value="com.exemple.Main"/>

  <!-- Cible de nettoyage -->
  <target name="clean">
    <delete dir="${bin.dir}"/>
    <delete dir="${dist.dir}"/>
  </target>
```


Les Bases de build.xml dans Apache Ant

Exemple Complet de build.xml

```
<!-- Cible de compilation -->  
<target name="compile" depends="clean">  
  <mkdir dir="${bin.dir}"/>  
  <javac srcdir="${source.dir}" destdir="${bin.dir}">  
    <classpath>  
      <pathelement location="${lib.dir}"/>  
    </classpath>  
  </javac>  
</target>
```

Les Bases de build.xml dans Apache Ant

Exemple Complet de build.xml

```
<!-- Cible de packaging (création du fichier JAR) -->
<target name="package" depends="compile">
  <mkdir dir="${dist.dir}"/>
  <jar destfile="${dist.dir}/MonProjet.jar" basedir="${bin.dir}">
    <manifest>
      <attribute name="Main-Class" value="${main.class}"/>
    </manifest>
  </jar>
</target>
```

Les Bases de build.xml dans Apache Ant

Exemple Complet de build.xml

```
<!-- Cible d'exécution -->
<target name="run" depends="package">
  <java classname="${main.class}" fork="true">
    <classpath>
      <pathelement location="${bin.dir}"/>
      <pathelement location="${dist.dir}/MonProjet.jar"/>
    </classpath>
  </java>
</target>

</project>
```



Fichier build.properties dans Apache Ant

Le fichier **build.properties** est utilisé pour **définir des variables** qui seront réutilisées dans build.xml.

Cela permet de rendre un projet plus **dynamique, flexible et facile à modifier**.

Exemple de fichier build.properties

properties

 Copier  Modifier

```
# Définition des variables
```

```
src.dir=src
```

```
bin.dir=bin
```

```
lib.dir=lib
```

```
app.name=MonApplication
```



Fichier build.properties dans Apache Ant

build.xml utilisant build.properties

xml

Copier Modifier

```
<project name="${app.name}" default="compile" basedir=".">

  <!-- Chargement du fichier build.properties -->
  <property file="build.properties"/>

  <!-- Compilation -->
  <target name="compile">
    <mkdir dir="${bin.dir}"/>
    <javac srcdir="${src.dir}" destdir="${bin.dir}"/>
    <echo message="Compilation terminée dans ${bin.dir}"/>
  </target>

</project>
```



Fichier build.properties dans Apache Ant

Utilisation des variables

Dans build.xml, on utilise `${nom_de_variable}` pour accéder aux valeurs définies dans build.properties.

Variable	Valeur	Utilisation dans build.xml
src.dir	src	Dossier source du code
bin.dir	bin	Dossier de sortie des fichiers compilés
lib.dir	lib	Dossier contenant les bibliothèques
app.name	MonApplication	Nom du projet





Fichier build.properties dans Apache Ant

Chargement du fichier build.properties

◆ **Manière automatique** (comme dans l'exemple ci-dessus) :



xml

 Copier  Modifier

```
<property file="build.properties"/>
```

◆ **Manière manuelle** (passer une variable à la commande Ant) :

sh

 Copier  Modifier

```
ant -Dsrc.dir=source -Dbin.dir=output
```



Passer un argument lors de l'exécution d'un build Ant

Apache Ant permet de **passer des arguments dynamiques** à l'exécution via l'option -D. Cela permet de **modifier des valeurs sans changer build.xml**.

Utilité des arguments dynamiques

- ✓ Permet de personnaliser l'exécution sans modifier build.xml.
- ✓ Utile pour changer des chemins, des noms de fichiers, des options de compilation.
- ✓ Facilite l'automatisation des builds avec des paramètres flexibles.




Passer un argument lors de l'exécution d'un build Ant

Exemple

a) Passer un argument en ligne de commande

Commande Ant avec un argument env (exemple pour un environnement de build) :

```
sh  
  
ant -Denv=production
```

 Copier  Modifier

b) Récupération de l'argument dans build.xml

```
xml  
  
<project name="Demo" default="showEnv" basedir=".">  
  <target name="showEnv">  
    <echo message="Environnement actuel : ${env}"/>  
  </target>  
</project>
```

 Copier  Modifier



Tests Unitaires avec Apache Ant

Apache Ant permet d'automatiser l'exécution des **tests unitaires** en intégrant **JUnit**, un framework de test très utilisé en Java. Cela permet de **valider le bon fonctionnement du code**, d'**identifier les bugs** et d'**assurer la non-régression** après chaque modification.

Pré-requis pour Exécuter des Tests avec Ant

Avant de configurer Ant pour exécuter des tests unitaires, assurez-vous d'avoir :



- ✓ **Apache Ant installé**
- ✓ **JUnit (junit.jar) dans le projet**
- ✓ **Un projet Java structuré avec des classes de test**

Pré-requis pour Exécuter des Tests avec Ant

Structure du Projet

Voici une structure de projet classique :

bash

 Copier  Modifier

/MonProjet

```
|— /src          # Code source Java
|   |— Main.java
|— /test         # Fichiers de test JUnit
|   |— MainTest.java
|— /lib          # Bibliothèques externes
|   |— junit.jar
|— /build        # Répertoire de compilation
|— build.xml     # Script Ant
```

Pré-requis pour Exécuter des Tests avec Ant

Fichier build.xml pour Ant et JUnit

```
xmlCopier Modifier  
  
<project name="TestProject" basedir="." default="test">  
  
    <!-- Définition des répertoires -->  
    <property name="src.dir" value="src"/>  
    <property name="test.dir" value="test"/>  
    <property name="build.dir" value="build"/>  
    <property name="report.dir" value="${build.dir}/reports"/>  
  
    <!-- Définition du classpath -->  
    <path id="classpath">  
        <fileset dir="lib">  
            <include name="**/*.jar"/>  
        </fileset>  
        <pathelement path="${build.dir}"/>  
    </path>  
  
    <!-- Compilation du code source -->  
    <target name="compile">  
        <mkdir dir="${build.dir}"/>  
        <javac srcdir="${src.dir}" destdir="${build.dir}" classpathref="classpath"/>  
    </target>  
  
    <img alt="down arrow icon" data-bbox="468 950 488 980"/>
```

Pré-requis pour Exécuter des Tests avec Ant

Fichier build.xml pour Ant et JUnit

```
<!-- Compilation des tests -->
<target name="compile-test" depends="compile">
    <javac srcdir="${test.dir}" destdir="${build.dir}" classpathref="classpath"/>
</target>

<!-- Exécution des tests JUnit -->
<target name="test" depends="compile-test">
    <mkdir dir="${report.dir}"/>
    <junit printsummary="yes" haltonfailure="yes">
        <classpath refid="classpath"/>
        <formatter type="xml"/>
        <batchtest fork="yes" todir="${report.dir}">
            <fileset dir="${test.dir}">
                <include name="**/*Test.java"/>
            </fileset>
        </batchtest>
    </junit>
</target>

</project>
```



Pré-requis pour Exécuter des Tests avec Ant

Exécution des Tests avec Ant

Dans un terminal, placez-vous dans le dossier du projet et lancez la commande :

```
sh
```

[Copier](#) [Modifier](#)

```
ant test
```

Ant exécutera les tests et affichera le résultat dans la console.

Pré-requis pour Exécuter des Tests avec Ant

Génération des Rapports de Test

Les résultats des tests sont enregistrés dans build/reports.

Pour **afficher un rapport HTML**, utilisez **JUnitReport** :

Ajoutez cette cible dans build.xml :

```
xmlCopier Modifier  
  
<target name="report" depends="test">  
  <mkdir dir="${report.dir}/html"/>  
  <junitreport todir="${report.dir}/html">  
    <fileset dir="${report.dir}">  
      <include name="TEST-*.xml"/>  
    </fileset>  
    <report format="frames" todir="${report.dir}/html"/>  
  </junitreport>  
</target>
```


Pré-requis pour Exécuter des Tests avec Ant

Génération des Rapports de Test

Puis exécutez :

```
sh  
  
ant report
```

Copier Modifier

Les rapports seront disponibles dans `build/reports/html/index.html`.

Merci pour votre attention