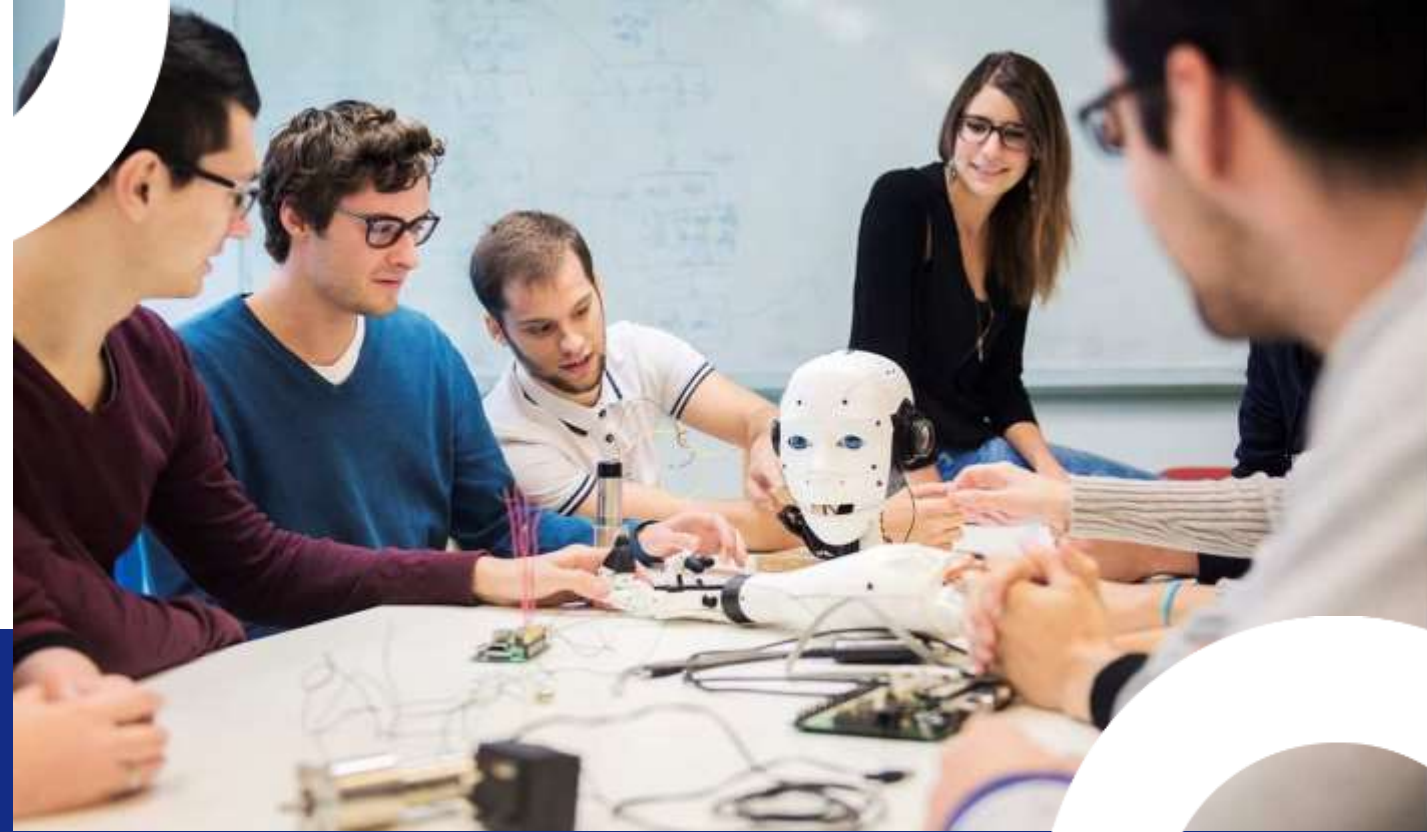




L'ÉCOLE DE L'INNOVATION
TECHNOLOGIQUE



Exercice Scientifique et technique

Maintenabilité logiciel et migration vers de nouvelles pratique de développement

Jovanovic Marko

Introduction

Problématique :

Comment garantir la maintenabilité d'un logiciel sur le long terme lors d'une migration vers de nouvelles pratiques de développement ?

Contexte professionnel

Contexte du projet :

- Application web en évolution constante
- Développement rapide (startup)
- Backend / Frontend
- Utilisation d'un ORM

Problème :

- Code difficile à maintenir
- Complexité croissante

La dette technique

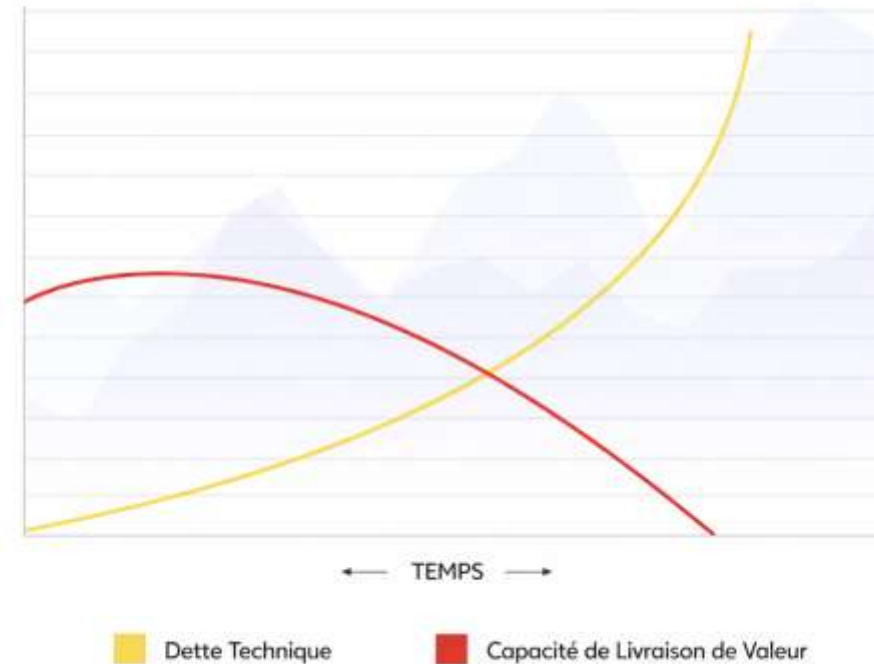
Concept introduit par Ward Cunningham :

- Compromis technique a court terme
- Accumulation de complexité
- Code difficile à modifier

Conséquences :

- Ralentissement du développement
- Augmentation de bugs

Corrélation entre Dette Technique et Capacité de Livraison de Valeur



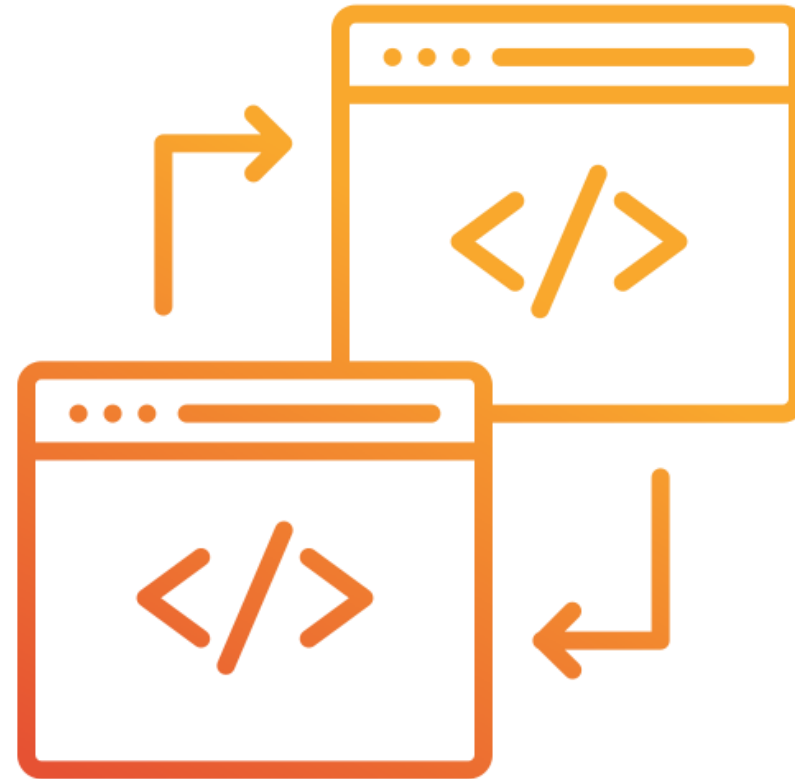
La refactorisation

Concept popularisé par Martin Fowler :

- Améliorer la structure du code
- Sans modifier le comportement
- Réduire la complexité
- Améliorer la lisibilité

Objectifs :

- Améliorer la maintenabilité



Principes SOLID



Single Responsibility Principle

C'est sans doute le principe le plus simple à comprendre dans SOLID. Une classe ne doit avoir qu'une seule et unique responsabilité.



Open / Closed Principle

On commence à rentrer dans le vif du sujet. Les entités doivent être ouvertes à l'extension et fermées à la modification.



Liskov Substitution Principle

Les objets dans un programme doivent être remplaçables par des instances de leur sous-type sans pour autant altérer le bon fonctionnement du programme.



Interface Segregation Principle

Aucun client ne devrait être forcé d'implémenter des méthodes / fonctions qu'il n'utilise pas.



Dependency Inversion Principle

Une classe doit dépendre de son abstraction, pas de son implémentation.

Objectifs : 5 principes de conception orientée objet visant à produire un code modulaire, maintenable et facile à évoluer

Risques de la migration

Modifier un logiciel peut entraîner :

- Des bugs
- Régressions
- Comportements inattendus

Problème :

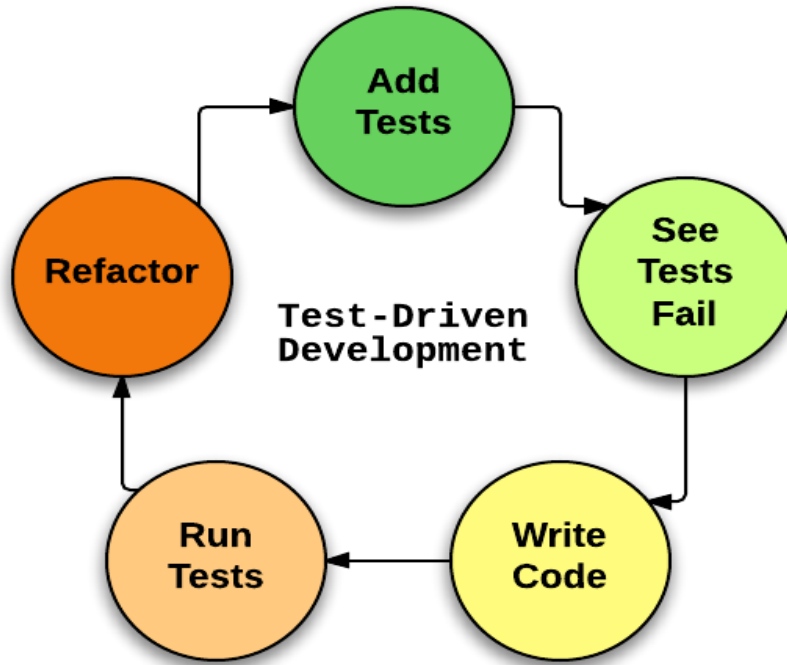
Comment modifier le code sans casser le logiciel ?



Ne pas se précipiter

Tests automatisés / TDD (Méthode introduite par Kent Beck)

Cycle TDD:



Avantages :

- Sécuriser les refactorisations
- Eviter les régressions

Mesure de la qualité 1/2



Présentation :

Plateforme open-source d'inspection automatique du code source, + de 30 langages pris en charge (Java, C#, Python, JavaScript, PHP, etc.)

Ce que ça fait :

- Détecte les bugs, vulnérabilités et "code smells"
- Analyse statique sur 30+ langages
- Intégration dans la CI/CD (Jenkins, GitHub Actions, GitLab...)

Mesure de la qualité 2/2



Les 3 axes clés :

- Fiabilité — bugs critiques détectés avant la prod
- Sécurité — failles OWASP identifiées automatiquement
- Maintenabilité — dette technique mesurée et suivie

Bénéfices :

- Feedback rapide aux développeurs
- Historique et tendances de qualité
- Standardisation des pratiques d'équipe

Conclusion

Pour garantir la maintenabilité :

- Réduire la dette technique
- Appliquer des principes de conception
- Sécuriser avec des tests
- Mesurer la qualité du code

Afin de garantir une migration progressive et contrôlée.