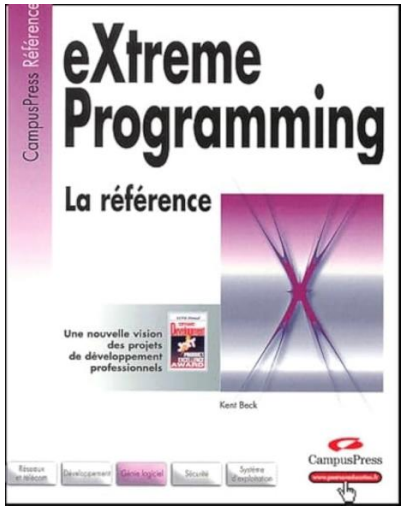
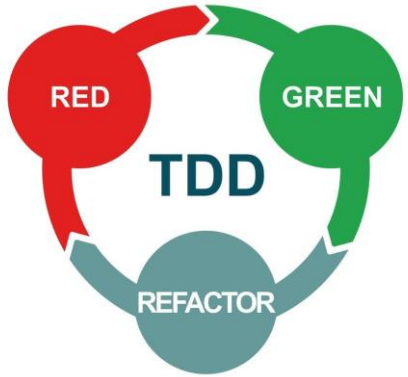
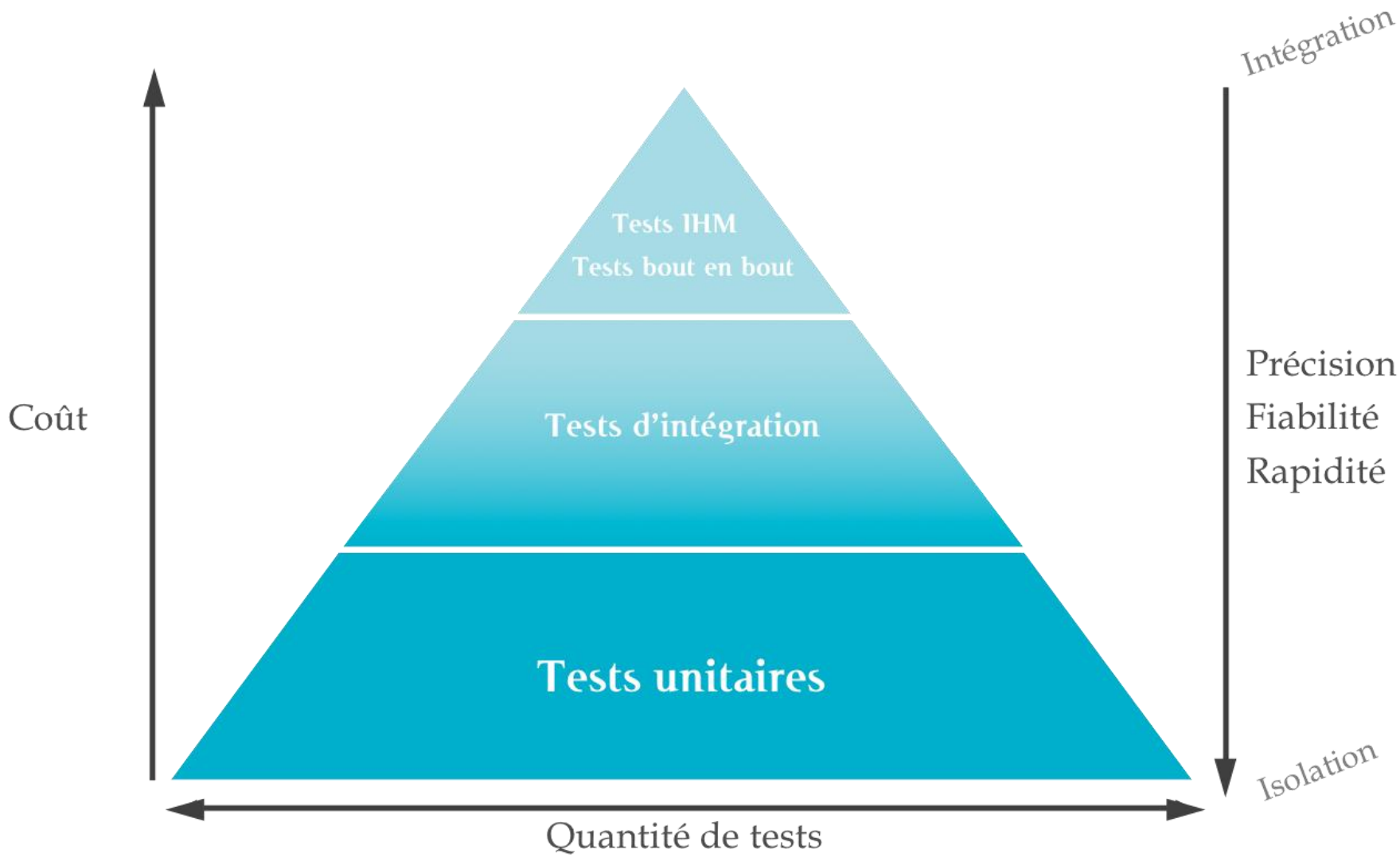
Test Driven Development (TDD)





Règles d’Ouverture

- Liberté de parole
  - Tu peux poser des questions, te tromper, challenger les idées.
- Pas de 'chef’
  - Je facilite, mais la valeur vient de l’échange collectif.
- Les deux pieds
  - Si ce n’est pas utile pour toi → tu es libre de partir.


Intégration
Tests IHM
Tests bout en bout
Tests d'intégration
Tests unitaires
Coût
Précision
Fiabilité
Rapidité
Isolation
Quantité de tests

Test After

Risques

- Code difficile à tester
- Zones non testés (couverture)
- Fausses excuses : « Je testerai plus tard »
- Pas de feedback
- Tests difficiles à écrire
- Risques dans l’architecture

On écrit le code et puis on voit après si on arrive à le tester…

Test First

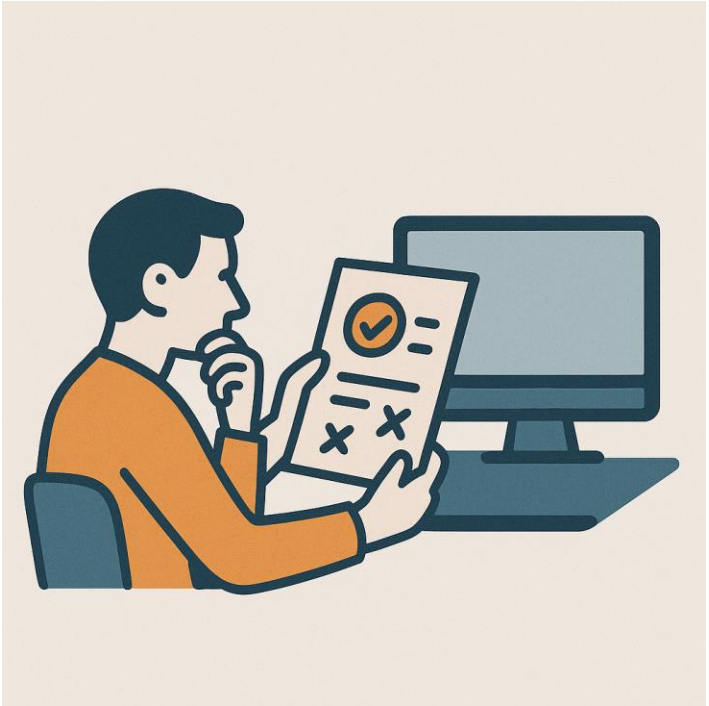
On écrit les tests avant le code, et on code dans l'objectifs de faire passer les tests

- Le test est un point de départ
- On peut tout coder d'un coup sans faire de petits pas

Les risques :

- Implémenter des choses inutiles ou prématurés
- Manque de feedback
- On code pour satisfaire les tests, tests fragiles
- Trop gros découpage, trop grosse fonctionnalité

Test First ≠ TDD

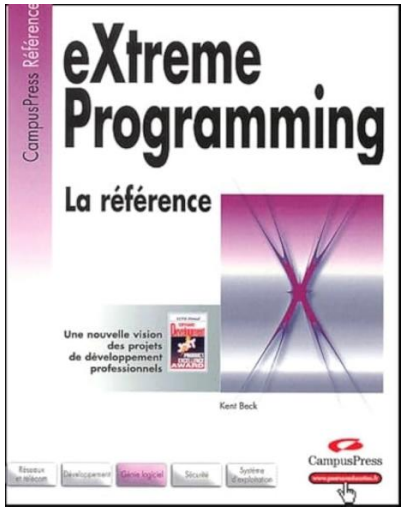


TDD

- Technique de développement qui impose l'écriture **d'un test** avant même l'écriture de la première ligne de code
- On progresse de manière itérative, petits pas (steps), à toi de trouver le bon rythme

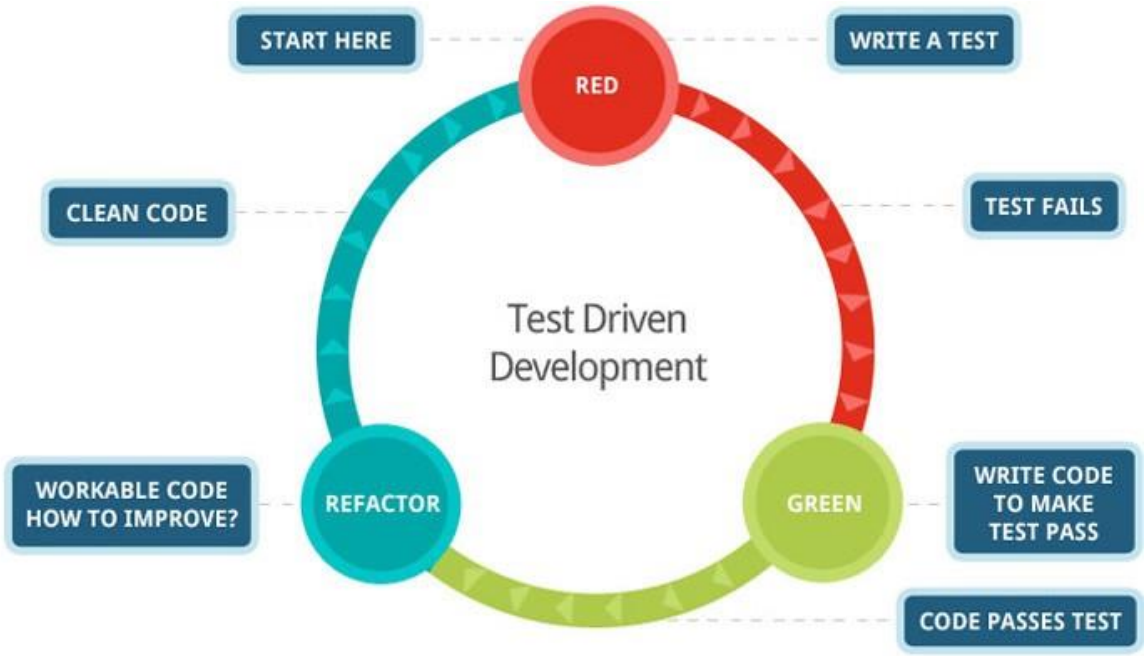
Un peu d'histoire
Octobre 1999 : Kent Beck présente une nouvelle méthode de programmation agile : l'eXtreme Programming (XP)

- Pair programming
- Revues de code
- ...
- TDD : piloter le développement par les tests



TDD ≠ juste tester
C'est une technique de conception guidée par les tests

Méthode



Red : Ecrire un test qui échoue, pour de bonnes raisons
Green : Ecrire du code le plus direct et suffisant pour faire passer le test
Refactor : Améliorer le code (simplicité, lisibilité, maintenabilité,...)

AVANTAGES

- Améliore la qualité et la compréhension du code
- Le test documente l'application
- Encourage une architecture claire et découplée
- Donne un feedback immédiat
- Réduit les bugs et les régressions
- On ne code que le nécessaire YAGNI (you ain't gonna need it)

LIMITES

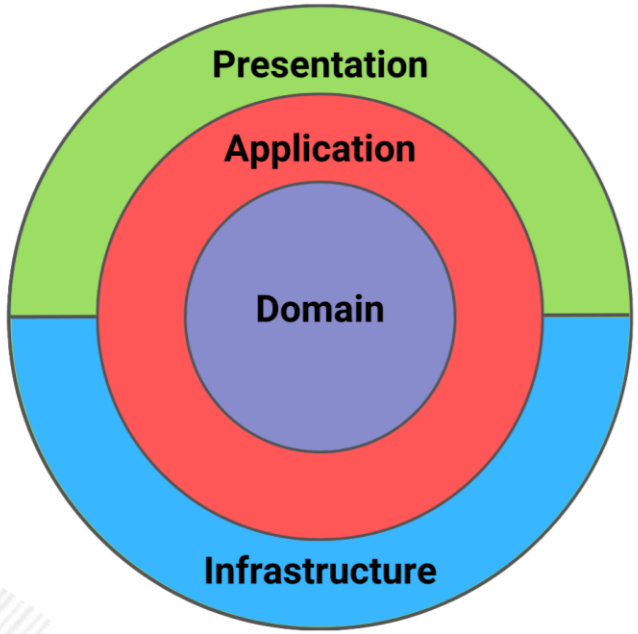
- Requiert une architecture découplée
- Demande de la discipline
- Plus long au départ
- Impossible sur le code spaghetti
- Nécessite une bonne maitrise du test

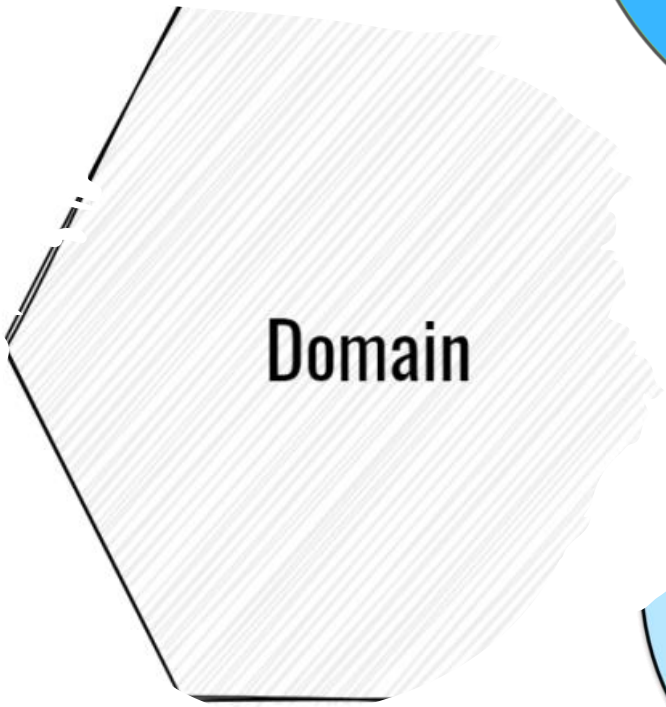
Il n'y a pas d'inconvénients

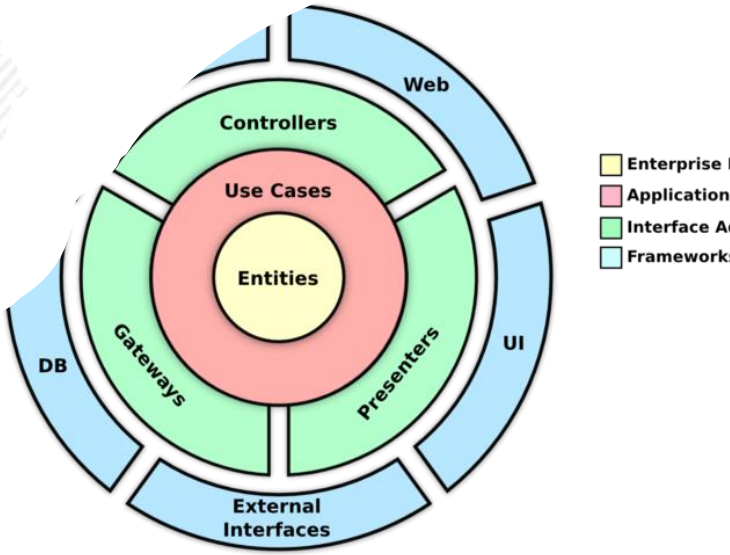
Architecture découplée

- Les tests forcent à séparer le domaine, les dépendances et les effets de bord.
- Clean / Hexa / Onion facilitent l'écriture de tests unitaires rapides et fiables.
- TDD fait émerger une architecture propre ; l'architecture propre simplifie le TDD
- C'est par le **refactoring** qu'on fait émerger une structure testable dans le legacy.

⚠️ Nécessite de savoir refactorer pour mettre en place dans le Legacy







Doublures de test

Pourquoi ?

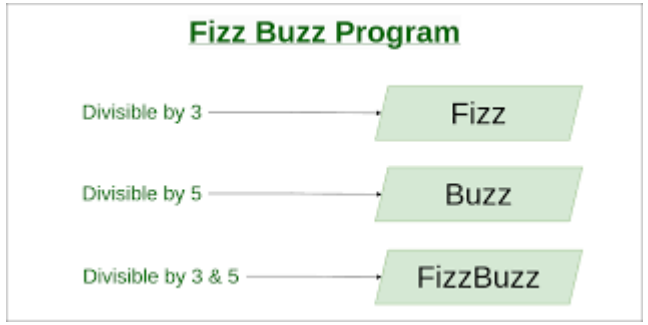
Pour remplacer une dépendance réelle dans un test et maitriser son comportement

Stub	Fournit une réponse prédéfinie
Mock	Vérifie les interactions (méthodes appelées)
Fake	Implémentation simple mais fonctionnelle
Spy	Enregistre ce qui a été appelé
Dummy	Objet factice passé juste pour remplir une signature de méthode

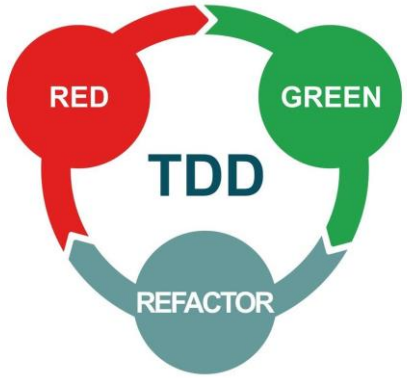
Utiliser une doublure **quand la vraie dépendance complique le test** (instable, lente, externe ou non déterministe).

KATA : FizzBuzz

- Mob programming
- Timebox : 1h30 Max
- Mieux vaut arrêter en plein milieu que dépasser le temps.
- L’objectif = apprendre ensemble, pas produire du “beau code” parfait



Bonus : tester ses tests



Next Steps

- 📌 Approfondir le **refactoring avec I/O** :
Isoler lectures/écritures (fichiers, DB, API) pour rendre le code testable.
- 📌 Clean : exemple d’une architecture découplée
- 📌 Aller plus loin dans le TDD avec le BDD (Behaviour Driven Developement)

Biblio

- https://www.dunod.com/sciences-techniques/software-craft-tdd-clean-code-et-autres-pratiques-essentielles-0
- XP: Kent Beck