

# Annexe — Les tests comme révélateur architectural

## Pourquoi cette annexe existe

Dans beaucoup de cours, les tests sont présentés comme :

- un outil de validation
- un moyen d'éviter les bugs
- une obligation qualité

Dans ce module, ils jouent un rôle **plus fondamental** :

Les tests révèlent la qualité de l'architecture.

Cette annexe a pour objectif d'aider à comprendre **ce que les tests disent de votre code**,  
même avant de parler de couverture ou de frameworks.

## Le principe central

Un code difficile à tester est presque toujours mal architecturé.

Les tests ne créent pas les problèmes :

- ils les rendent visibles
- ils les rendent concrets
- ils empêchent de les ignorer

## Ce que les tests révèlent immédiatement

Quand un test est pénible à écrire, cela indique souvent :

- un **couplage trop fort**
- des **dépendances mal placées**
- une **Violation du SRP**
- une **mauvaise séparation des responsabilités**

👉 Les tests agissent comme un **scanner architectural**.

# La vraie question à se poser

Avant même d'écrire un test, demande-toi :

« De quoi ai-je vraiment besoin pour tester ce comportement ? »

Si la réponse inclut :

- une base de données
- un framework
- une API externe
- un contexte technique complexe

→ alors ce comportement dépend trop de la technique.

# Tests rapides vs tests lourds

## Tests rapides (signal positif)

- isolés
- sans infrastructure
- simples à lire
- exécutés très souvent

➡ Architecture saine.

## Tests lourds (signal d'alerte)

- lents
- dépendants de l'environnement
- difficiles à maintenir

# Tests et responsabilités

Un test avec :

- 30 lignes de setup
- 3 lignes d'assertions

révèle souvent :

- un module qui fait trop de choses
- des responsabilités mélangées

👉 Setup complexe = SRP probablement violé.

# Tests et dépendances

Les tests mettent immédiatement en évidence :

- les dépendances inutiles
- les dépendances cachées
- les dépendances concrètes difficiles à remplacer

Un bon test aide à distinguer :

- l'essentiel (métier)
- le détail (technique)

## Le rôle particulier du TDD

Le TDD (Test-Driven Development) est une pratique qui renforce naturellement les bons choix architecturaux.

### Le cycle

1. RED → écrire un test qui échoue
2. GREEN → écrire le code minimal pour passer
3. REFACTOR → améliorer la structure sans casser le test

Le test devient le **premier client du code**.

# Pourquoi le TDD améliore l'architecture

Le TDD ne garantit pas une bonne architecture,  
mais il exerce une pression constante vers :

- la simplicité
- le découplage
- la clarté des responsabilités

Une mauvaise architecture :

- rend le TDD pénible
- ou impossible à maintenir dans le temps

# TDD et dépendances

Si tu n'arrives pas à écrire un test sans :

- instancier une base de données
  - configurer un framework
- le design force des dépendances trop tôt.

Le TDD pousse naturellement vers :

- l'inversion de dépendances
- l'utilisation d'interfaces / ports
- des composants remplaçables

## Tests et architecture hexagonale

L'architecture hexagonale et le TDD se renforcent mutuellement :

- **le domaine** est testé sans dépendances
- **les use cases** sont testés avec des fakes
- **les adapters** sont testés séparément
- les tests E2E restent peu nombreux

👉 Une pyramide de tests équilibrée apparaît naturellement.

# Erreurs classiques chez les étudiants

« Je ferai les tests plus tard »

Plus l'architecture est mauvaise,  
plus écrire les tests plus tard sera coûteux.

## Over-mocking

Trop de mocks dans un test indique souvent :

- trop de responsabilités
- un couplage excessif

## Tester l'implémentation

Un test ne doit pas connaître :

- les détails internes

## Exercice mental recommandé

Quand un test te semble pénible, demande-toi :

« Qu'est-ce que ce test m'apprend sur la structure de mon code ? »

Très souvent, la réponse n'est pas :

« Je suis mauvais en tests »

Mais :

« Mon architecture a un problème. »

## Lien avec les autres principes

-  **Cohésion** : tests simples → modules cohésifs
-  **SRP** : tests complexes → responsabilités mélangées
-  **Couplage** : besoin de mocks excessifs → couplage fort
-  **Dépendances** : tests lents → dépendances mal placées
-  **Inversion de dépendances** : tests faciles → inversion réussie

## À retenir

Les tests ne servent pas seulement à vérifier le code,  
ils contraignent l'architecture.

Si les tests sont naturels :

- l'architecture est probablement saine

Si les tests sont pénibles :

- l'architecture demande à être repensée.