

Annexe 1 — Maîtriser les dépendances

Cette annexe traite des dépendances sous deux angles complémentaires :

1. Comprendre ce qu'est vraiment une dépendance
2. Apprendre à les inverser pour protéger le métier

PARTIE 1 : Comprendre vraiment les dépendances

Pourquoi les dépendances sont souvent sous-estimées

Quand on parle de dépendances, les étudiants pensent souvent :

- bibliothèques externes
- frameworks
- bases de données

Mais en architecture logicielle, le vrai problème n'est pas :

« Est-ce que j'utilise une dépendance ? »

La vraie question est :

« À quel point mon code est-il lié à cette dépendance ? »

Dépendance \neq import

Une dépendance n'est pas seulement :

- un `import`
- une librairie externe

C'est tout ce dont ton code a besoin pour fonctionner :

- un framework
- une base de données
- une API externe
- une implémentation concrète
- une décision technique

Dépendance \neq import (suite)

👉 Ce qui compte, ce n'est pas leur existence, c'est l'impact de leur changement.

La vraie question à se poser

👉 Si cette dépendance change ou disparaît, que se passe-t-il ?

- Le code métier continue de fonctionner → dépendance maîtrisée
- Le code s'effondre → dépendance critique

Les dépendances définissent la **fragilité structurelle** du système.

Dépendances et inertie du code

Plus un code dépend fortement de :

- frameworks
- choix techniques
- services externes

Dépendances et inertie du code (suite)

Plus il devient :

- difficile à faire évoluer
- coûteux à tester
- lent à adapter à de nouveaux besoins

👉 Le code acquiert une **inertie** :
il devient difficile à déplacer, modifier ou refactorer.

Exemple conceptuel (sans code)

Imagine une règle métier simple :

« Un ticket ne peut pas être fermé s'il est déjà clos. »

Si pour tester cette règle tu dois :

- lancer une base de données
- démarrer une API
- configurer un framework

Exemple conceptuel (suite)

➡ La règle dépend trop de la technique.

La règle n'a pas besoin de ces éléments pour exister, elle devrait donc en être **indépendante**.

Dépendances visibles vs dépendances cachées

Dépendances visibles

- paramètres de constructeurs
- interfaces explicites
- configuration claire

➡ Elles sont contrôlables et testables.

Dépendances visibles vs dépendances cachées (suite)

Dépendances cachées

- singletons globaux
- appels statiques
- accès directs à l'infrastructure
- contexte implicite

➡ Elles sont dangereuses car :

- invisibles dans la signature
- difficiles à remplacer
- difficiles à tester

Dépendances et testabilité

Un test lent ou compliqué révèle souvent :

- trop de dépendances
- des dépendances trop concrètes
- des dépendances mal isolées

Pose-toi cette question :

« De quoi ai-je vraiment besoin pour tester cette règle ? »

Tout le reste est probablement :

- une dépendance inutile
- ou mal placée

Dépendances et responsabilités

Une dépendance mal placée est souvent le symptôme :

- d'une violation du SRP
- d'un mauvais découpage des responsabilités

Exemple :

- le domaine dépend de la base de données
- la règle métier dépend du framework

➡ La responsabilité n'est pas au bon endroit.

Exercice mental : détail ou métier ?

Pour chaque dépendance importante, demande-toi :

« Est-ce que cette dépendance est un détail
ou un élément central de mon métier ? »

- Détail → elle doit être remplaçable
- Central → elle doit être explicitement modélisée

PARTIE 2 : L'inversion de dépendances — La solution

Pourquoi l'inversion de dépendances est souvent mal comprise

Quand on parle d'inversion de dépendances, beaucoup d'étudiants pensent :

- injection de dépendances
- frameworks (Spring, FastAPI, etc.)
- technique avancée

Mais l'inversion de dépendances **n'est pas d'abord un outil technique.**

C'est avant tout une **règle de conception** qui répond à une question simple :

Qui décide de quoi dans le système ?

Le vrai problème à résoudre

Sans inversion de dépendances :

- le code métier dépend de la technique
- les choix techniques deviennent structurants
- changer d'outil revient à réécrire le métier

Le vrai problème à résoudre (suite)

👉 Le cœur du système devient **prisonnier de ses dépendances**.

La vraie question à se poser

👉 Qui définit le contrat ?

- Si la technique définit le contrat → dépendance mal orientée
- Si le métier définit le contrat → dépendance maîtrisée

L'inversion de dépendances consiste à faire en sorte que :

les modules importants imposent leurs besoins,
et les modules techniques s'y adaptent.

Inversion de dépendances \neq injection de dépendances

Erreur très fréquente :

« J'utilise un framework d'injection, donc j'ai fait l'inversion. »

✗ Faux.

- L'injection est un **mécanisme**
- L'inversion est une **décision architecturale**

On peut :

- injecter des dépendances
- tout en gardant un couplage fort au technique

Exemple conceptuel (sans code)

Cas classique :

- le métier appelle directement une base de données
- le métier dépend d'une API externe

Si la base change :

➡ le métier change

Exemple conceptuel (suite)

Avec inversion :

- le métier définit ce dont il a besoin
- la technique fournit une implémentation

Si la technique change :

➡ le métier ne change pas

Inversion de dépendances et pouvoir de décision

L'inversion de dépendances est une question de **pouvoir** dans le système.

Sans inversion :

- la technique impose ses contraintes
- le métier s'adapte

Inversion de dépendances et pouvoir de décision (suite)

Avec inversion :

- le métier impose ses besoins
- la technique s'adapte

👉 Une bonne architecture protège ce pouvoir de décision.

Lien avec les ports et interfaces

Les **interfaces** utilisées dans l'inversion de dépendances :

- ne sont pas des abstractions "pour faire joli"
- représentent des **besoins métiers**

Lien avec les ports et interfaces (suite)

On ne crée pas une interface parce que :

- « C'est plus propre »

On la crée parce que :

- le métier refuse de dépendre d'un détail technique.

Inversion de dépendances et testabilité

Grâce à l'inversion :

- le métier peut être testé sans infrastructure
- les tests deviennent rapides et ciblés
- les règles métier sont isolées

Inversion de dépendances et testabilité (suite)

Si tester une règle métier nécessite :

- une base de données
- un framework
- un service externe

➡ l'inversion n'est pas correctement appliquée.

Erreur classique chez les étudiants

« Je verrai plus tard pour l'inversion,
pour l'instant je veux que ça marche. »

Ce "plus tard" arrive rarement sans douleur.

L'inversion de dépendances est **beaucoup plus coûteuse à ajouter après coup** que de la prévoir dès le départ.

Exercice mental : qui devrait changer ?

Pour chaque dépendance importante, demande-toi :

« Si cette dépendance change demain,
est-ce que le métier devrait vraiment changer ? »

- Non → inversion nécessaire
- Oui → dépendance probablement légitime

Synthèse : De la compréhension à la maîtrise

Les dépendances (Partie 1)

Les dépendances déterminent la capacité de ton code à résister au changement.

Une bonne architecture :

- accepte les dépendances
- mais refuse d'en être prisonnière

Synthèse (suite)





L'inversion (Partie 2)

L'inversion de dépendances n'est pas un détail technique, c'est une décision politique dans l'architecture.

Elle détermine :

- qui contrôle le système
- ce qui peut évoluer sans douleur
- ce qui est réellement central dans ton code

Lien avec les autres principes

-  **Couplage** : l'inversion réduit le couplage fort
-  **SRP** : une dépendance mal placée viole souvent le SRP
-  **Tests** : l'inversion rend le métier testable
-  **Architecture hexagonale** : l'inversion en est le cœur

À retenir

Les deux faces d'une même pièce :

1. Comprendre les dépendances → identifier les problèmes
2. Inverser les dépendances → les résoudre

L'inversion n'a de sens que si tu comprends d'abord
ce qu'est vraiment une dépendance et pourquoi elle pose problème.