

## Annexe 2 — Découpage et responsabilités

Cette annexe explore les trois facettes du bon découpage :

1. Le couplage — mesure la dépendance entre modules
2. La cohésion — mesure l'unité interne d'un module
3. Le SRP — règle de conception pour obtenir couplage faible et cohésion forte

# PARTIE 1 : Le couplage — Limiter la propagation du changement

## Pourquoi le couplage pose autant de problèmes

Le couplage mesure à quel point un module dépend d'autres modules.

Un couplage fort n'est pas toujours visible immédiatement :

- le code compile
- l'application fonctionne
- les fonctionnalités semblent correctes

👉 Les vrais problèmes apparaissent lors du changement.

## La vraie question à se poser

👉 Si je modifie ce module, qu'est-ce qui risque de casser ailleurs ?

- Réponse : *presque rien* → couplage faible
- Réponse : *beaucoup de choses* → couplage fort

Le couplage n'est donc pas un problème de syntaxe,  
mais un problème de **propagation du changement**.

## Couplage fort : illusion de simplicité

Erreur fréquente :

« Je vais appeler directement cette classe, c'est plus simple. »

À court terme :

- moins de fichiers
- moins d'abstraction
- code plus rapide à écrire

À moyen terme :

- tests difficiles à écrire
- dépendances en cascade
- peur de modifier le code

👉 Le couplage fort est souvent un gain immédiat

## Symptôme courant : le "train wreck"

On parle de **train wreck** lorsqu'on voit des appels en chaîne du type :

```
objetA.getObjetB()  
    .getObjetC()  
    .faireQuelqueChose();
```

### Pourquoi c'est un problème

Ce type de code signifie que :

- le module connaît **la structure interne** de plusieurs objets
- il dépend de **plusieurs niveaux de détails**
- un changement dans un objet intermédiaire peut tout casser

👉 Le module n'est plus seulement couplé à un objet,  
il est couplé à **toute une chaîne d'objets**.

## Conséquence pratique

Si :

- `getObjetB()` change
  - ou `getObjetC()` disparaît
  - ou la structure interne évolue
- ➡ le code appelant doit être modifié, même si son besoin métier n'a pas changé.

## Question mentale utile

« Pourquoi ai-je besoin de connaître autant de choses pour effectuer une action simple ? »

Si la réponse n'est pas évidente → alerte couplage.

## Couplage et héritage profond

Un autre couplage moins visible, mais très fréquent,  
apparaît avec des **hiérarchies d'héritage trop profondes**.

### Le problème

Dans une chaîne d'héritage longue :

- une classe dépend implicitement du comportement de ses parents
- les effets de bord ne sont pas visibles localement
- une modification en haut de la hiérarchie peut casser des classes très bas

👉 Ce couplage est **caché**, mais très fort.

## Pourquoi c'est dangereux

- Le comportement réel d'une classe n'est plus lisible dans son fichier
- Il faut comprendre toute la hiérarchie pour raisonner correctement
- Les impacts d'un changement deviennent difficiles à prédire

Cela crée :

- de l'incertitude
- de la peur de refactorer
- des bugs inattendus

## Message clé (sans dogme)

Il ne s'agit pas de dire :

« L'héritage est mauvais »

Mais de comprendre que :

**Un héritage profond augmente fortement le couplage.**

C'est pour cette raison que, dans beaucoup de cas :

- on préfère des objets qui **collaborent** (composition)
- plutôt que des objets qui **héritent** d'un comportement complexe

## Couplage et testabilité

Un test difficile à écrire révèle souvent :

- trop de dépendances
- des dépendances trop concrètes
- une connaissance excessive de l'environnement

Si pour tester une règle simple tu dois :

- instancier beaucoup d'objets
  - comprendre une hiérarchie complexe
  - configurer un contexte lourd
- ➔ le couplage est probablement trop fort.

## **Exercice mental : connaître COMMENT ?**

Avant d'ajouter une dépendance, demande-toi :

« Est-ce que ce module a vraiment besoin de connaître  
COMMENT l'autre fonctionne ? »

Si la réponse est non → alerte couplage.

## PARTIE 2 : La cohésion — Unité interne d'un module

### Pourquoi la cohésion est souvent mal comprise

Beaucoup d'étudiants pensent que la cohésion signifie :

« Une classe avec peu de méthodes »

ou

« Une classe qui fait un truc précis »

C'est insuffisant.

La cohésion ne concerne pas la taille d'un module,  
elle concerne la nature des raisons pour lesquelles il change.

## La vraie question à se poser

👉 Pourquoi ce module peut-il être modifié ?

- S'il n'a qu'une seule raison de changer, la cohésion est bonne.
- S'il a plusieurs raisons indépendantes, la cohésion est mauvaise.

Ce raisonnement est plus fiable que :

- le nombre de méthodes
- le nombre de lignes
- le nom de la classe

## Relation forte entre cohésion et SRP

Il existe une **relation directe et forte** entre la cohésion et le SRP — Single Responsibility Principle.

### SRP : une règle de conception

Le SRP dit :

« Un module ne doit avoir qu'une seule raison de changer. »

C'est une **règle** que l'on applique consciemment lors de la conception.

### Cohésion : un indicateur de qualité

La cohésion est :

- une **mesure**
- un **symptôme**

## Exemple conceptuel (sans code)

Imagine un module appelé :

UserManager

Il est modifié :

1. Quand les règles d'inscription changent
2. Quand le format d'email change
3. Quand la politique de mot de passe change
4. Quand le stockage en base évolue

- ➔ 4 raisons différentes de changer
- ➔ Violation du SRP
- ➔ Cohésion faible

Même si le code est :

## Bonne cohésion ≠ découpage excessif

Erreur fréquente :

« Si je découpe tout en petites classes, j'aurai une bonne cohésion »

✗ Faux.

Un découpage excessif peut :

- augmenter la complexité
- nuire à la lisibilité
- créer du couplage inutile

👉 Bonne cohésion = découpage pertinent, pas découpage maximal.

## Cohésion et lisibilité mentale

Un bon test simple :

« Est-ce que je peux expliquer ce module en une phrase courte ? »

- Oui → cohésion probablement bonne
- Non → responsabilité floue

## **Exercice mental : logique de changer CE fichier ?**

Avant de coder, pose-toi cette question :

« Si cette règle change demain,  
est-ce que c'est logique que je modifie CE fichier ? »

Si la réponse est hésitante → alerte cohésion.

## PARTIE 3 : Le SRP — La règle de conception

### Pourquoi le SRP est souvent mal appliqué

Beaucoup d'étudiants pensent respecter le SRP parce que :

« Ma classe ne fait qu'une seule chose : gérer les commandes »

Le problème est que :

- "gérer X" n'est pas une responsabilité
- c'est un **thème fonctionnel**, pas un critère de conception

Le SRP ne parle pas de *quoi* fait une classe,  
mais de **pourquoi elle change**.

## Responsabilité ≠ rôle fonctionnel

Une **responsabilité**, en architecture logicielle, correspond à :

- une source de décision
- une raison de modification
- un type de règle

Une classe peut sembler avoir un seul rôle fonctionnel,  
tout en ayant **plusieurs responsabilités structurelles**.

## Exemple conceptuel (sans code)

Une classe `OrderService` qui :

- applique des règles métier
- orchestre des appels à d'autres services
- gère des erreurs techniques
- écrit des logs
- appelle une API externe

Fonctionnellement :

« Elle gère les commandes »

Architecturalement :

- règles métier
- orchestration applicative

## SRP et couches architecturales

Le SRP est l'un des fondements implicites des architectures bien structurées.

Dans une architecture en couches ou hexagonale :

- **le domaine** porte les règles métier
- **l'application** orchestre les cas d'usage
- **les adapters** gèrent la technique

👉 Mélanger ces rôles dans un même module viole le SRP,  
même si le thème métier est unique.

## SRP et évolution du code

Un bon indicateur SRP est l'historique des changements.

Pose-toi ces questions :

- Ce fichier change-t-il pour des raisons très différentes ?
- Qui demande ce changement ? (métier, technique, infrastructure)
- Est-ce que ces changements sont liés entre eux ?

Si plusieurs raisons indépendantes existent → SRP violé.

## Erreur classique : "c'est plus pratique ici"

Très souvent, le SRP est violé par pragmatisme à court terme :

« Je mets ça ici, c'est plus simple. »

Ce choix :

- accélère aujourd'hui
- ralentit tout le projet demain

Le SRP est un principe de **protection contre l'accumulation de décisions** dans un même endroit.

## SRP et lisibilité du code

Un module qui respecte le SRP :

- est plus simple à lire
- a un comportement prévisible
- peut être modifié sans crainte excessive

Un module qui viole le SRP :

- devient un point central du projet
- attire toujours plus de logique
- devient difficile à refactorer

## Exercice mental : qui serait mécontent ?

Pour chaque classe importante, demande-toi :

« Qui serait mécontent si je modifie ce fichier ? »

- une seule réponse claire → SRP respecté
- plusieurs réponses différentes → SRP probablement violé

# Synthèse : Les trois facettes du bon découpage

## Couplage (Partie 1)

Le couplage détermine la fragilité de ton code face au changement.

Moins un module connaît de détails sur les autres,  
plus le système est **robuste, testable et évolutif**.

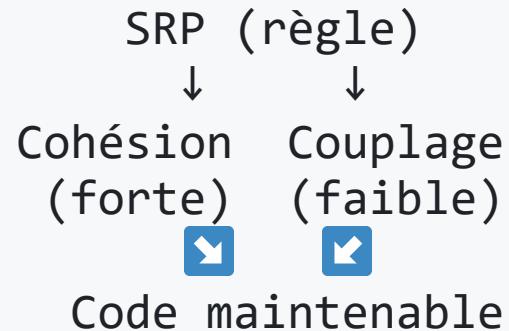
## Cohésion (Partie 2)

Le SRP est une règle à appliquer,  
la cohésion est un signal à observer.

Les deux parlent de la même chose,  
mais sous des angles différents et complémentaires.

## SRP (Partie 3)

# Lien entre les trois concepts



- **Appliquer le SRP** → augmente la cohésion, réduit le couplage
- **Cohésion forte** → limite le couplage (modules bien délimités)
- **Couplage faible** → facilite la cohésion (modules indépendants)

Ces trois concepts sont **indissociables** :  
ils forment ensemble la base du bon découpage.

## À retenir

Un bon découpage, c'est :

1. **SRP** : une seule raison de changer par module
2. **Cohésion forte** : tout ce qui va ensemble est ensemble
3. **Couplage faible** : les modules ne savent pas comment les autres fonctionnent

Ces trois principes travaillent ensemble pour créer un code :

- facile à comprendre
- facile à tester
- facile à faire évoluer