

Architectures Logicielles Modernes



Séance 1 — Du chaos aux couches : juger, diagnostiquer et refactorer une structure

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et
Techniques, Université Cheikh Anta Diop de Dakar

<https://ebtoure.sn/cours/architectures-logicielles-modernes/chapitres/01-du-chaos-aux-couches.html>

Version du 11 octobre 2026



Version en ligne

Architectures Logicielles — Séance 1

Du chaos aux couches : juger, diagnostiquer et refactorer une structure

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

Objectifs de la séance

Contenu

Cette séance pose le socle du cours : ce qu'est une **architecture logicielle**, comment on la **juge** (attributs de qualité), comment on la **diagnostique** (couplage, cohésion), et comment on l'**améliore en sécurité** (refactoring vers une architecture en couches). Le lab applique tout cela à SénResto, notre plateforme de livraison de repas à Dakar.

- Distinguer *fonctionner* et *être bien structuré* : la notion de **dette architecturale**.
- Juger une structure avec les **attributs de qualité** — et comprendre qu'ils se paient les uns avec les autres.
- Diagnostiquer avec le vocabulaire du métier : **couplage** et **cohésion**.
- Connaître le remède du jour, l'**architecture en couches**, et la méthode sûre pour y aller : le **refactoring** protégé par des tests.

Fonctionner ne suffit pas

Le patient du jour : SénResto

La situation

SénResto permet de commander un thiéboudienne chez Fatou à la Médina ou un mafé chez Aminata à Ouakam, d'assigner un livreur et de suivre la commande jusqu'à la livraison.

L'application **fonctionne parfaitement** : le frontend React est agréable, l'API répond, les livreurs sont assignés, les montants sont justes.

Et pourtant : tout le backend tient dans **une seule classe de 414 lignes**. La question de la séance — et du semestre :

La question centrale

Si l'application fonctionne, qu'est-ce qui ne va pas ? *Pour qui* est-elle mauvaise, et *quand* cela se paiera-t-il ?

Qu'est-ce qu'une architecture logicielle ?

Définition

L'**architecture** d'un logiciel est sa structure : les **parties** qui le composent, la **responsabilité** de chacune, et les **dépendances** entre elles. Ce sont les décisions structurantes — celles qui seraient *coûteuses à changer* une fois prises.

- Ajouter un bouton se défait en une heure : ce n'est pas de l'architecture.
- Décider que tout le code vit dans une classe — ou dans trois couches, ou dans cinq services — engage *tout* le développement futur : c'est de l'architecture.

À retenir

L'architecture ne se voit pas à l'écran. Deux applications identiques pour l'utilisateur peuvent être, pour l'équipe qui les maintient, un plaisir ou un cauchemar.

L'intuition du restaurant

Un restaurant bien organisé

En salle, le serveur prend les commandes et sert — il ne cuisine pas. En cuisine, le chef applique les recettes — il ne fait pas le service. Au cellier, le magasinier gère les stocks — il ne cuisine pas. Chacun a **un métier**, et la commande circule dans **un seul sens** : salle → cuisine → cellier.

Imaginez maintenant le restaurant où *une seule personne* prend les commandes, cuisine, gère le stock et encaisse. Ça marche... tant qu'il y a trois clients, et tant qu'elle ne tombe pas malade. C'est exactement l'état de SénResto aujourd'hui.

À retenir

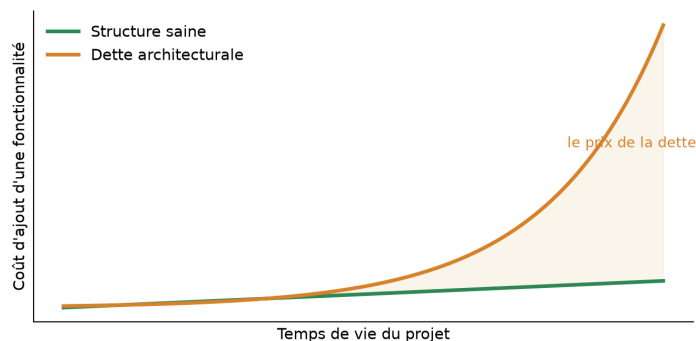
Une bonne structure permet de comprendre, modifier, tester et remplacer chaque partie **séparément**. C'est vrai d'un restaurant, c'est vrai d'un logiciel.

Attributs de qualité

La dette architecturale

Dette architecturale

La **dette architecturale** est le coût futur créé par les raccourcis structurels d'aujourd'hui. Comme une dette d'argent, elle porte intérêt : chaque fonctionnalité coûte plus cher que la précédente.



Au début, les deux courbes se confondent — la mauvaise structure est même souvent *plus rapide*. C'est pour cela qu'on la choisit... et qu'on la regrette.

► Voir le code de la figure

Les attributs de qualité

Attributs de qualité

Les **attributs de qualité** sont les propriétés non fonctionnelles par lesquelles on juge une structure. Chacun se traduit par une question concrète et vérifiable.

Attribut	La question concrète
Maintenabilité	Combien de fichiers faut-il toucher pour ajouter un quartier livrable ?
Testabilité	Peut-on vérifier la formule des frais <i>sans</i> démarrer une base de données ?
Évolutivité	Ajouter le paiement mobile money : chantier local ou refonte générale ?
Lisibilité	Un nouveau membre du groupe comprend-il où vit chaque règle ?
Résilience	Si une partie tombe, le reste survit-il ? (rendez-vous au Lab 5)

Les trade-offs

Trade-off

Un **trade-off** est un échange : améliorer un attribut se paie presque toujours en dégradant un autre. Il n'existe pas de « meilleure architecture » dans l'absolu — seulement des architectures *adaptées à un contexte*.

- Séparer en couches améliore la testabilité... au prix de *plus de fichiers* et d'indirection.
- Les microservices améliorent le déploiement indépendant... au prix de la complexité du réseau (nous le *vivrons* au Lab 4).

À retenir — la devise du semestre

Un architecte ne demande jamais « quelle est la meilleure solution ? » mais « **qu'est-ce que j'achète, et avec quoi je le paie ?** ». Chaque ADR que vous rédigerez devra nommer ce qui est payé.

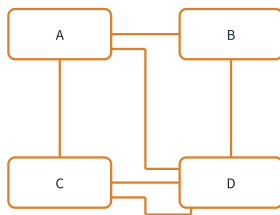
Couplage et cohésion

Le couplage

Définition

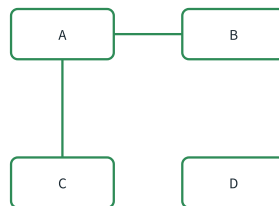
Le **couplage** mesure à quel point les parties d'un système dépendent les unes des autres. Il se lit dans une question : *si je modifie ceci, qu'est-ce que je risque de casser ?*

Couplage fort



tout dépend de tout :
toucher A, c'est risquer B, C et D

Couplage faible



des interfaces minimales :
toucher A ne concerne que ses voisins déclarés

La cohésion

Définition

La **cohésion** mesure à quel point les éléments regroupés ensemble le sont *pour une bonne raison* — parce qu'ils participent à la même responsabilité.

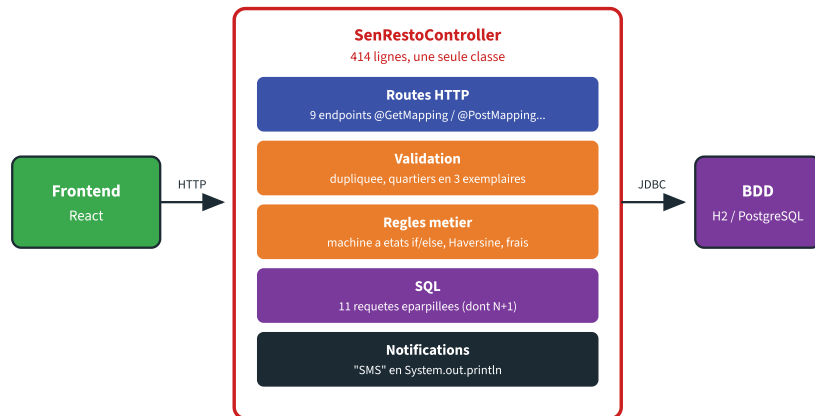
Test rapide sur SénResto

Décrivez `SenRestoController` en une phrase sans dire « et » : « *il reçoit les requêtes HTTP **et** valide les données **et** calcule les frais **et** écrit le SQL **et** assigne les livreurs **et** envoie les SMS...* »
Chaque « et » est un aveu : la classe a plusieurs métiers, sa cohésion est faible.

La devise

Couplage faible, cohésion forte. Chaque module fait une chose (cohésion), et dépend du minimum d'autres modules (couplage). C'est le principe de responsabilité unique, à toutes les échelles.

Les symptômes au microscope : SénResto



Une seule classe fait tout : chaque modification traverse tout, rien ne se teste isolément.

Un *God controller* : toutes les responsabilités dans une classe. Au lab, vous cartographierez ligne par ligne les sept défauts de cette structure.

Le symptôme qui les résume tous

Rien ne se teste isolément

Pour vérifier la simple formule « $\text{prix} \times \text{quantité} + \text{frais}$ », il faut aujourd'hui : démarrer Spring, initialiser une base de données, construire une requête HTTP, et analyser du JSON. La règle métier est **prisonnière** de la technique qui l'entoure.

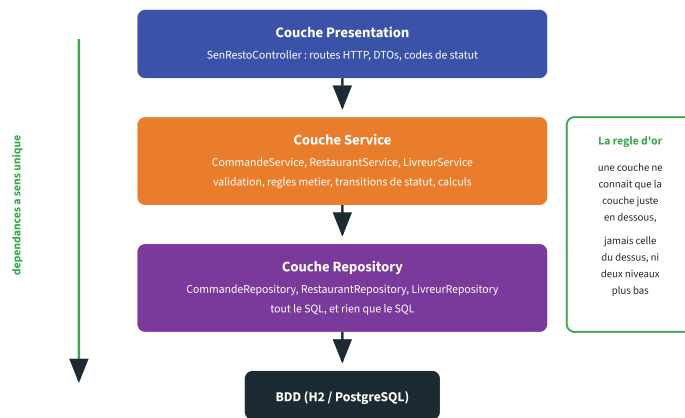
- C'est pour cela que le projet ne contient *que* des tests d'intégration : le test unitaire y est **impossible** — non pas difficile, impossible.
- La testabilité est le **thermomètre** de l'architecture : un code difficile à tester est un code mal structuré. L'inverse est (presque) toujours vrai.

Vérité de terrain

La duplication aggrave tout : la liste des quartiers de Dakar existe en **trois** exemplaires (frais, GPS, formulaire React). Ouvrir la livraison aux Parcelles Assainies = trois modifications à ne pas oublier. La probabilité d'oubli croît avec le temps... et se découvre en production.

L'architecture en couches

Trois couches, trois métiers



Chaque couche a une seule responsabilité : on peut la comprendre, la modifier et la tester seule.

La responsabilité de chaque couche

Couche	Son métier	Ce qu'elle s'interdit
Présentation (controller)	parler HTTP : routes, codes de statut, sérialisation	toute règle métier, tout SQL
Service (service)	le métier : validation, règles, calculs, transitions d'état	parler HTTP, parler SQL
Accès aux données (repository)	tout le SQL, et rien que le SQL	valider, décider, calculer

La règle d'or

Les dépendances vont dans **un seul sens**, du haut vers le bas, et chaque couche ne connaît que sa voisine immédiate. Le controller ignore la base ; le repository ignore HTTP.

Ce que la séparation achète

Avant : la validation vit *dans* l'endpoint, testable uniquement via HTTP + BDD.

Après : elle vit dans le service, et lève une exception métier — testable en une ligne, sans rien démarrer :

```
// Dans CommandeService : du metier pur, sans HTTP ni SQL
private String validerTelephone(Object tel) {
    if (tel == null || !tel.toString().matches("^7[05678][0-9]{7}$")) {
        throw new ValidationException("Telephone invalide : format 7XXXXXXXX");
    }
    return tel.toString();
}
```

- Un seul `@RestControllerAdvice` traduit les exceptions métier en codes HTTP — là où le God controller répétait neuf fois le même bloc d'erreur.
- La règle des frais, la machine à états, le calcul d'ETA deviennent des méthodes **nommées**, à **une** adresse, testables **unitairement**.

Refactorer en sécurité

Le refactoring : définition et contrat

Définition

Le **refactoring** consiste à améliorer la structure interne d'un code **sans changer son comportement observable**. Mêmes routes, mêmes JSON, mêmes codes d'erreur : le frontend ne doit rien remarquer du chantier.

Le piège classique

« Tant qu'on y est, corrigeons aussi ce petit bug... » — Non. Changer le comportement *pendant* un refactoring, c'est perdre son juge de paix : un test casse, et on ne sait plus si c'est la structure ou le comportement. **Une intention à la fois.**

La méthode des petits pas

Extraire *une* responsabilité → relancer les tests → committer → recommencer. Le refactoring avance en pas de fourmi, mais il n'échoue jamais.

Le filet : les tests de comportement

Le projet fournit **quatre tests d'intégration** qui décrivent le comportement de l'API — et ignorent tout de sa structure interne. C'est précisément pour cela qu'ils survivront au refactoring :

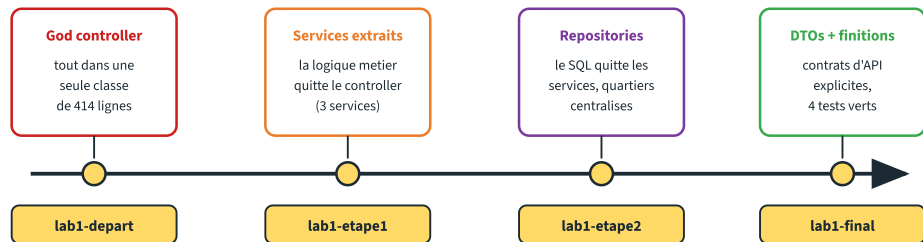
```
./mvnw test  
# Tests run: 4, Failures: 0, Errors: 0    <- l'etat obligatoire, en permanence
```

- Lister les restaurants, créer une commande, rejeter un téléphone invalide, confirmer et assigner un livreur : les quatre parcours vitaux.
- **Un test rouge = le comportement a changé = ce n'est plus du refactoring.** On revient en arrière ([git](#)), on comprend, on recommence plus petit.

À retenir

On ne refactorise jamais sans filet. Les tests transforment une opération à cœur ouvert en une promenade balisée.

Le chemin du lab, tag par tag



Bloques ? git checkout du tag suivant et on repart : personne ne reste au bord de la route.

Le code de départ de chaque lab du semestre sera l'état final du lab précédent :
rester dans le rythme, c'est capitaliser.

Votre mission d'aujourd'hui

Rôle	Pilotage au Lab 1
Le Doyen	branches <code>feature/</code> , merges, tags — et l'ADR avec l'Architecte du jour
L'Huissier	<code>CommandeService</code> , exceptions métier, <code>GestionnaireErreurs</code>
Le Rédacteur	<code>RestaurantService</code> , puis les DTOs (le contrat de son frontend)
Le Notaire	l'inventaire SQL, puis les trois repositories
L'Appariteur	<code>LivreurService</code> , <code>QuartiersDakar</code> , vérification H2 + PostgreSQL

Le livrable individuel de la séance

L'**Architecte du jour** signe l'**ADR n°1** : « **Pourquoi séparer en couches ?** » — une page, deux alternatives sérieuses, deux conséquences négatives assumées (gabarit distribué).

L'essentiel de la séance

À retenir

- **Fonctionner ne suffit pas** : la dette architecturale se contracte en silence et se paie avec intérêts.
- On juge une structure par ses **attributs de qualité**, et on sait qu'ils s'achètent les uns avec les autres : **trade-offs**.
- Le diagnostic tient en une devise : **couplage faible, cohésion forte** — et la testabilité en est le thermomètre.
- Le remède du jour : **trois couches**, des dépendances à sens unique — atteintes par **petits pas**, sous la protection des tests, jalonnées par les tags.

La question qui ouvre la séance 2

Votre code sera propre ce soir... mais vos règles métier dépendront toujours, trois étages plus bas, d'une base de données. Peut-on *retourner* cette dépendance comme un gant ?
Réponse au Lab 2 : l'architecture hexagonale.