



Architectures Logicielles Modernes

Séance 2 — L'hexagone : mettre le domaine au centre, la technique en périphérie

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/02-l-hexagone.html>

Version du 11 octobre 2026



Version en ligne

Architectures Logicielles — Séance 2

L'hexagone : mettre le domaine au centre, la technique en périphérie

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

Le Lab 1 a rangé la maison : couches propres, responsabilités séparées. Cette séance s'attaque au problème qui reste : le **métier dépend toujours de la technique**. Nous allons **inverser** cette dépendance — ports & adaptateurs, l'architecture dite **hexagonale** — et en apporter la preuve par le **swap** : remplacer la base de données par une simple Map, sans toucher une ligne du domaine.

- Comprendre *pourquoi* des couches propres ne suffisent pas : le sens des dépendances.
- Maîtriser le **principe d'inversion de dépendances** — et sa question clé : *qui possède l'interface ?*
- Savoir lire et construire une architecture **ports & adaptateurs**.
- En vérifier le bénéfice au chronomètre : des tests du domaine **en 2 secondes, sans base de données**.

Le problème restant

Ce que le Lab 1 a réglé — et pas réglé

L'acquis du Lab 1

Trois couches, des dépendances à sens unique, zéro SQL hors des repositories, les quartiers de Dakar à une seule adresse. Le God controller est mort : chaque responsabilité a une maison.

L'exigence nouvelle

La direction de SénResto exige que les règles métier — frais, transitions de statut, calcul d'ETA — soient testées **à chaque commit**. Les tests d'intégration (Spring + BDD + HTTP) prennent plusieurs secondes chacun : trop lents. Il faut des tests **unitaires** du métier. Essayez d'en écrire un sur [CommandeService](#)... vous n'y arriverez pas proprement.

La chaîne qui emprisonne le métier

Suivez les flèches de dépendance de `CommandeService` :

`CommandeService` → `CommandeRepository` → `JdbcTemplate` → **BDD**

- Pour instancier le service, il faut un repository ; pour le repository, un `JdbcTemplate` ; pour le `JdbcTemplate`, une base qui tourne.
- Conclusion : impossible de tester la formule des frais sans démarrer toute la pile. Le métier est **propre**, mais **prisonnier** — trois étages au-dessus de la technique.

À retenir

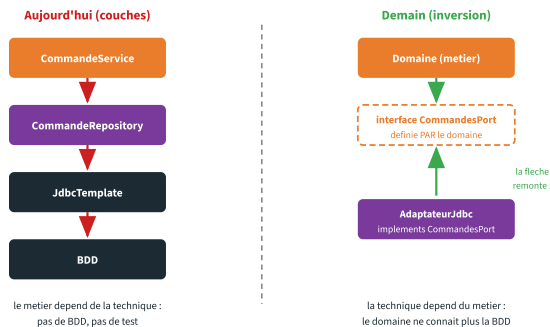
Le problème n'est plus le *désordre* (Lab 1 l'a réglé) : c'est le **sens des flèches**. Tant que le métier pointe vers la technique, la technique dicte ses conditions au métier.

Inverser les dépendances

Le principe d'inversion de dépendances

Définition

Principe d'inversion de dépendances (DIP) : le code de haut niveau (le métier) ne doit pas dépendre du code de bas niveau (la technique). Les deux doivent dépendre d'**abstractions** — des interfaces. Et surtout : ces interfaces sont **définies par le métier**, selon ses besoins.



La question qui change tout

Qui possède l'interface ?

Une interface ne suffit pas à inverser quoi que ce soit : si `CommandeRepository` définit lui-même son interface dans le paquetage `repository`, le domaine qui l'importe dépend *toujours* de la persistance. L'inversion se joue sur la **propriété** : l'interface habite **chez le domaine**, écrite dans **son** vocabulaire.

- Le domaine déclare : « *j'ai besoin qu'on me sauvegarde des commandes* » — il écrit `CommandesPort` chez lui.
- La technique répond : « *je sais faire, en JDBC* » — l'adaptateur **implémente** le port, et c'est donc *lui* qui importe le domaine. La flèche a remonté.

À retenir

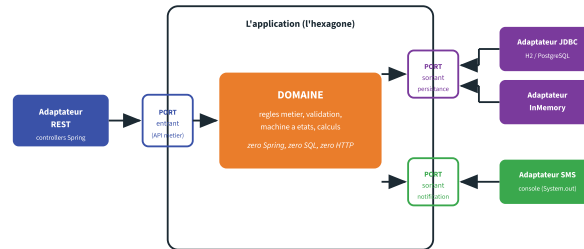
Inverser une dépendance, c'est déplacer la **propriété du contrat** : le métier ne demande plus la permission à la technique — il publie ses exigences, la technique s'y plie.

Ports et adaptateurs

L'architecture hexagonale

Définition (Alistair Cockburn, 2005)

L'architecture **hexagonale** (*ports & adaptateurs*) place le domaine au centre, entouré de **ports** — les interfaces qu'il définit. Les **adaptateurs** traduisent le monde extérieur vers les ports ou les implémentent. Règle unique : **toutes les dépendances pointent vers le centre**.



Les ports sont des interfaces définies PAR le domaine ; les adaptateurs les implémentent ou les appellent.
Toutes les flèches de dépendance pointent vers le centre : le domaine ne dépend de rien.

Ports entrants, ports sortants

	Port entrant	Port sortant
Qui l'appelle ?	le monde extérieur (REST, plus tard : messages)	le domaine
Qui l'implémente ?	le domaine	un adaptateur technique (JDBC, SMS...)
Chez SénResto	« créer une commande, la confirmer... »	« sauvegarder, chercher un livreur, notifier »

Le test du vocabulaire

Lisez un port à voix haute : s'il parle de `SELECT`, de JSON ou de HTTP, il est mal placé.
`CommandesPort.sauvegarder(commande)` — du métier. `executerRequete(sql)` — de la technique déguisée.

À quoi ressemble un port

Le domaine écrit son besoin, dans son vocabulaire, chez lui :

```
package sn.ucad.senresto.domaine.port;

/** Ce dont le métier des commandes a besoin – rien de plus. */
public interface LivreursPort {
    Optional<Livreur> premierDisponible();
    void reserver(long livreurId);
    void liberer(long livreurId);
}
```

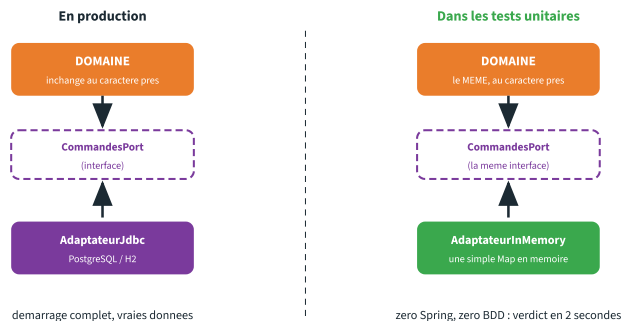
- Aucune annotation Spring, aucun mot technique : ce fichier compilerait en 2005 comme en 2035.
- L'adaptateur JDBC du Lab 1 devient : `class AdaptateurLivreursJdbc implements LivreursPort` — trois méthodes déjà écrites, seule l'*appartenance* change.

À retenir

Au Lab 2, vous n'écrirez presque pas de code nouveau : vous déplacerez la **propriété** du code existant. C'est un refactoring de frontières, pas une réécriture.

Le swap, la preuve

Le moment magique : changer la technique



On change la technique sans toucher une ligne du metier : c'est la preuve de l'inversion.

Le métier testé en deux secondes

L'adaptateur *in-memory* — une Map — implémente les mêmes ports. Le test unitaire devient trivial :

```
@Test
void lesFraisDeLivraisonSuiventLeQuartier() {
    var domaine = new CommandeMetier(new CommandesEnMemoire(),
        new RestaurantsEnMemoire(), new LivreursEnMemoire(),
        new NotificationsEnMemoire());
    var commande = domaine.creer(new CommandeRequete(
        "Awa Ba", "7700000000", "Ouakam", 1L, 2));
    assertEquals(2500 * 2 + 1500, commande.montantTotal()); // verdict : ~2 s
}
```

- Zéro Spring, zéro BDD, zéro HTTP : on instancie le domaine **au constructeur**, comme n'importe quel objet Java.
- Les 4 tests d'intégration restent en poste : ils vérifient le *branchement* complet. Les deux familles de tests sont complémentaires — rapides et nombreux au centre, lents et rares en périphérie.

Ce que l'hexagone achète — et ce qu'il coûte

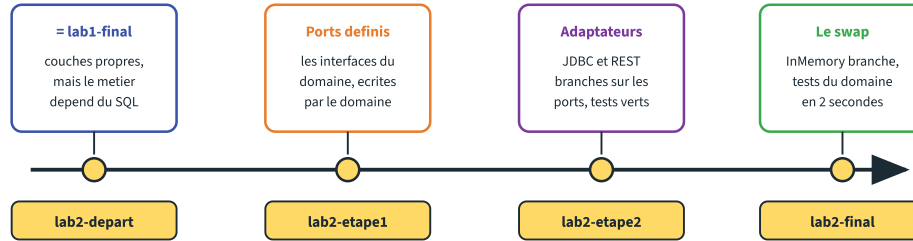
- **Achète** : le métier testable au centre ; la BDD, le framework, le canal de notification remplaçables sans toucher au domaine ; et une frontière prête pour la suite — au Lab 4, *extraire* un module sera d'autant plus simple que ses dépendances passent déjà par des ports.
- **Coûte** : des interfaces en plus, une indirection en plus, une discipline de vocabulaire permanente. Pour un script jetable, ce serait du zèle ; pour un système qui doit vivre et évoluer, c'est un investissement.

À retenir — le réflexe de l'architecte

Toujours la même question : *qu'est-ce que j'achète, avec quoi je le paie ?* Votre ADR n°2 devra répondre pour SénResto — pas en général.

Le lab

Le chemin du Lab 2



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

Point de départ : votre **lab1-final**. Le comportement de l'API ne change pas d'un octet — les 4 tests d'intégration restent les juges de paix.

Votre mission d'aujourd'hui

Rôle	Pilotage au Lab 2
Le Doyen	branches, merges, tags — et l'ADR avec l'Architecte du jour
L'Huissier	le domaine des commandes : <code>CommandeMetier</code> + ports sortants
Le Rédacteur	l'adaptateur REST (controllers) branché sur les ports entrants
Le Notaire	les adaptateurs JDBC : <code>implements</code> des ports, SQL inchangé
L'Appariteur	les adaptateurs in-memory + les nouveaux tests unitaires du domaine

Le livrable individuel de la séance

L'Architecte du jour signe l'ADR n°2 : « **Pourquoi inverser les dépendances ?** » — une page, deux alternatives sérieuses, deux conséquences négatives assumées.

L'essentiel de la séance

À retenir

- Des couches propres ne suffisent pas : tant que les flèches descendent vers la technique, **le métier reste prisonnier**.
- L'inversion se joue sur la **propriété de l'interface** : le port appartient au domaine, écrit dans son vocabulaire.
- **Hexagone** = domaine au centre, ports à la frontière, adaptateurs autour — toutes les dépendances pointent vers le centre.
- La preuve par le **swap** : même domaine, adaptateur JDBC en production, Map en mémoire dans les tests — verdict en 2 secondes.

La question qui ouvre la séance 3

Votre hexagone protège le métier... mais tout SénResto vit dans *un seul* hexagone : Catalogue, Commandes, Paiement, Livraison s'y mélangent. Où sont les frontières *entre les métiers* ? Réponse au Lab 3 : le monolithe modulaire.

Ressources de la séance

- **TP** · [Lab 2 — L'hexagone](#) (509 Ko)