



Bonnes pratiques en programmation Java



Encapsulation et protection des invariants

Le problème

Accès direct aux structures internes
→ états incohérents et règles
métier brisées

L'enjeu

Les invariants garantissent la cohérence. Sans protection : bugs subtils et difficiles à tracer



Vues immuables : première ligne de défense



Collections.unmodifiableList() t()

Protège l'état interne sans risque de modification externe



Classe Commande

Empêche ajout/suppression d'articles sans validation métier



Garantie d'intégrité

Toutes les modifications passent par des méthodes contrôlées

API intentionnelle : exprimer le métier

01

addItem()

Vérification disponibilité, mise à jour total, notification observateurs

02

applyDiscount()

Validation conditions, calcul montant, enregistrement historique

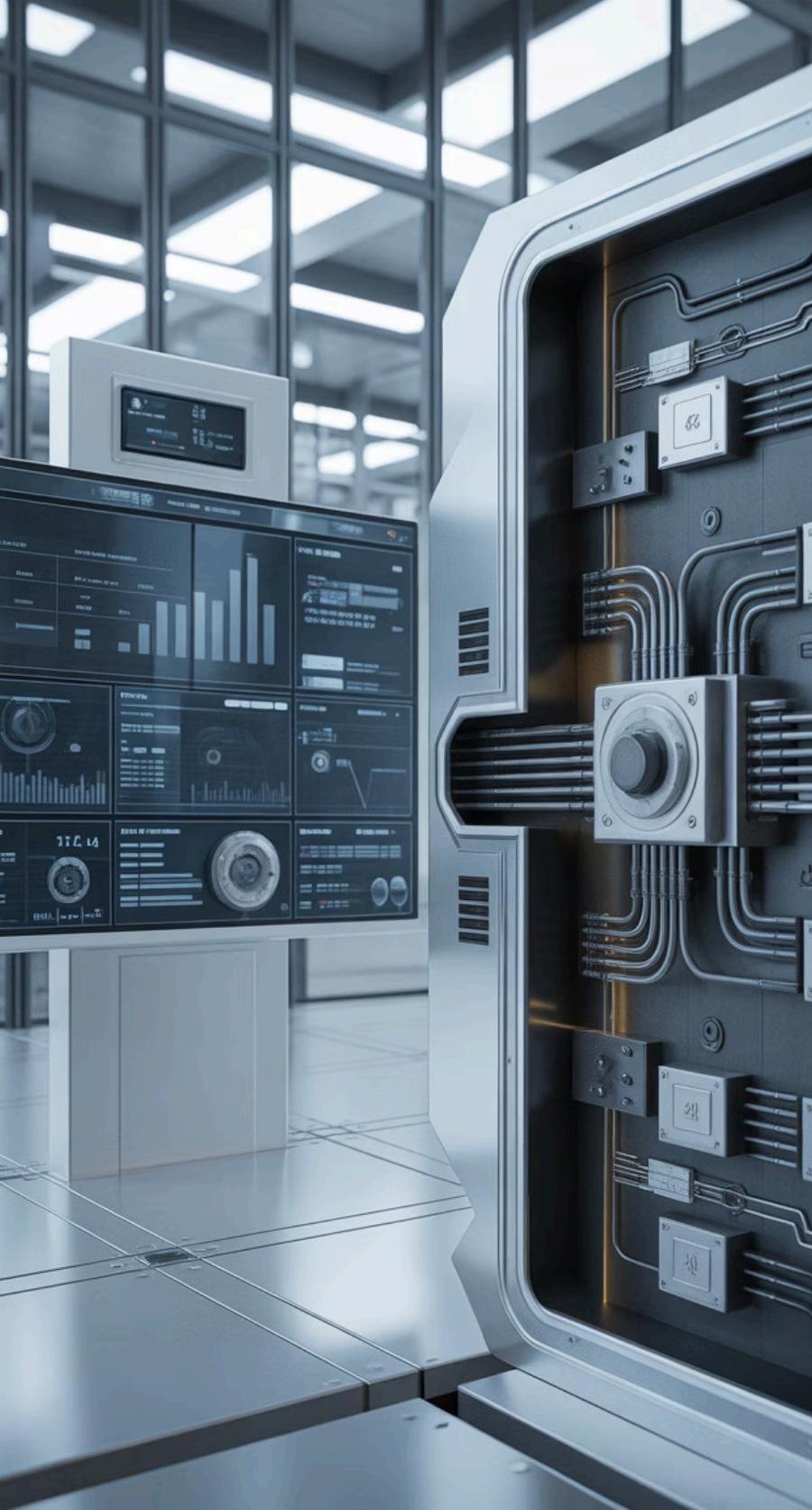
03

removeItem()

Suppression sécurisée, recalcul automatique, maintien cohérence

Ces méthodes forment une façade métier qui cache la complexité interne et garantit le respect des règles de gestion.





Validations précoce : fail-fast



Détection immédiate

Échouer au plus près de la source d'erreur

Débogage simplifié

L'erreur est identifiée instantanément

Code client allégé

Pas besoin de vérifier l'intégrité avant chaque utilisation

```
Objects.requireNonNull(items, "Items cannot be null");
if (items.isEmpty()) {
    throw new IllegalArgumentException(
        "Order must contain at least one item"
    );
}
```

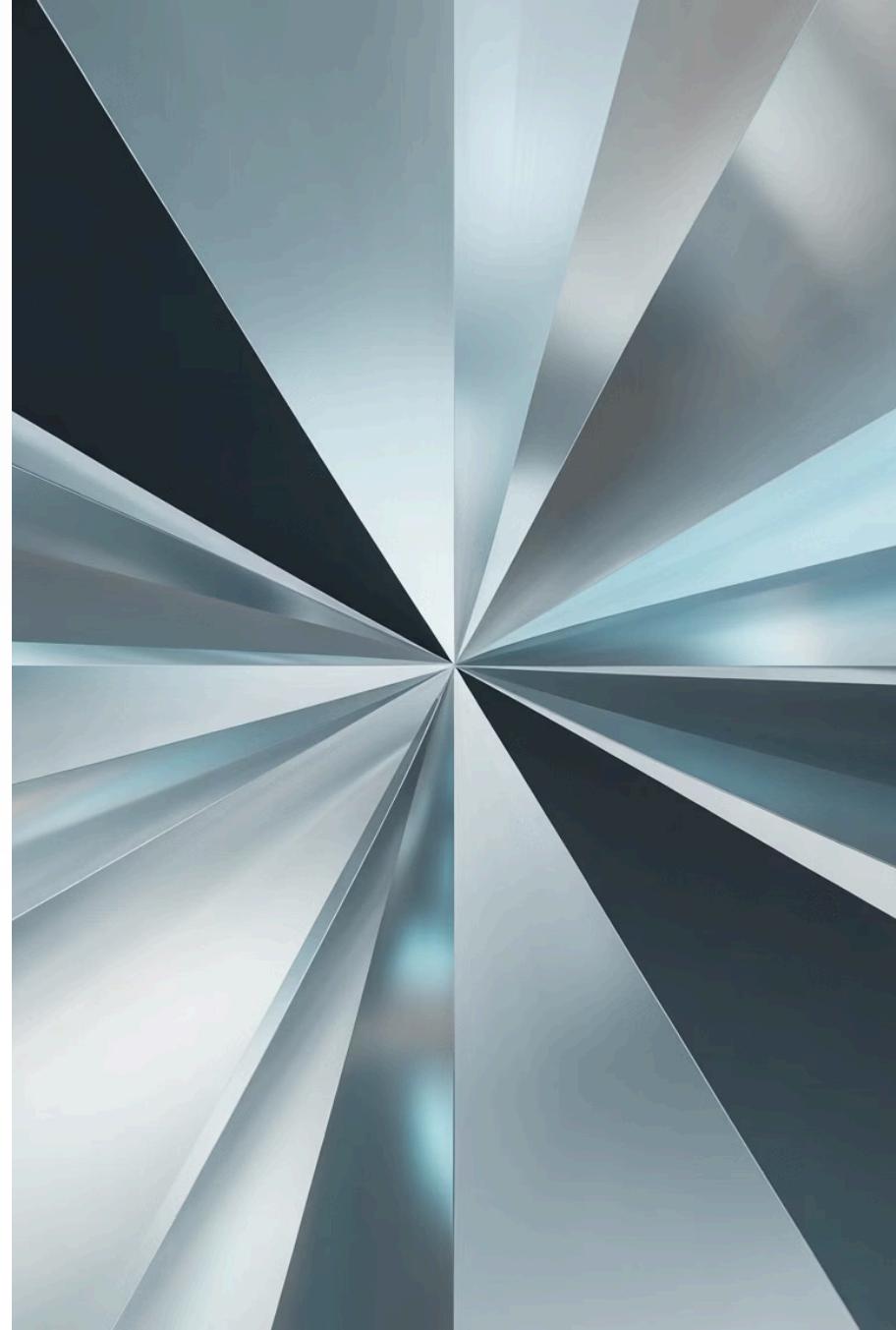
Immutabilité : fondations solides

Problématique

- Objets mutables : complexité en multi-threading
- Synchronisation nécessaire pour chaque modification
- Conditions de concurrence insidieuses
- Raisonnement difficile sur le code

Solution immutable

- État ne change jamais après création
- Élimine toute une catégorie de bugs
- Champs final, aucun setter
- Méthodes retournent nouvelles instances





Objets valeur : anatomie d'une classe immuable

1

Classe finale

Empêche l'héritage mutable

2

Champs privés et finaux

État interne protégé et immuable

3

Absence de setters

Aucune modification après construction

4

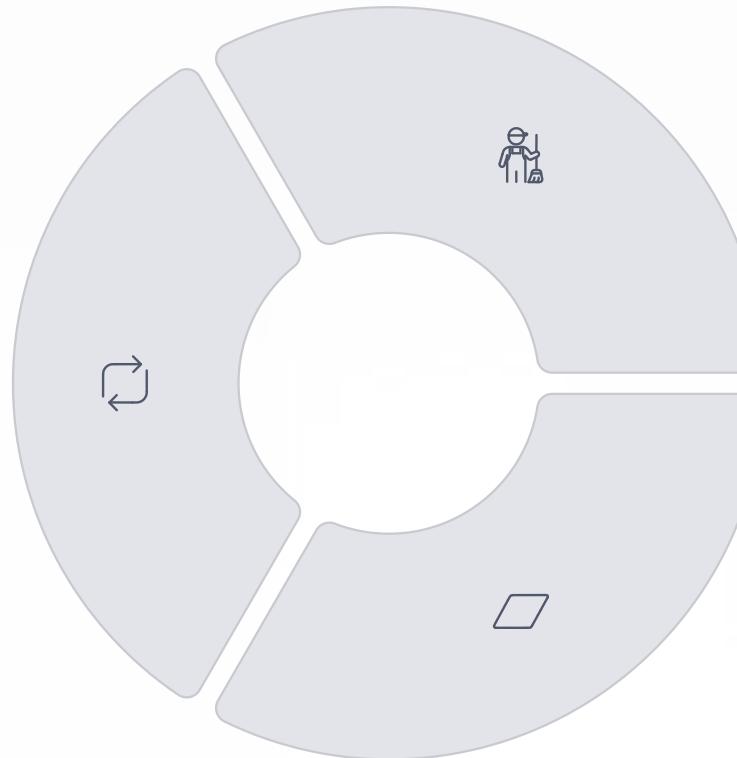
Méthodes créent nouvelles instances

plus() retourne nouveau Money sans modifier l'existant

Méthodes pures : prévisibilité et testabilité

Transparence référentielle

Même résultat pour mêmes arguments, toujours



Absence d'effets de bord

Ne modifie aucun état observable du système

Thread-safe intrinsèque

Appels simultanés sans risque de corruption



Immutabilité et tests simplifiés

Avec objet mutable

```
Account account = new  
Account(1000);  
account.withdraw(200);  
assertEquals(800,  
account.getBalance());  
// État modifié !  
account.withdraw(300);
```

Avec objet immuable

```
Money initial = new  
Money(1000, EUR);  
Money after1 =  
initial.minus(200);  
Money after2 =  
after1.minus(300);  
// initial inchangé !
```

L'approche immuable permet de tester chaque opération isolément et élimine les bugs liés à la mutation partagée.



Equals, HashCode, CompareTo

Le triangle de cohérence

Trois contrats doivent être respectés simultanément pour le bon fonctionnement des collections Java.

equals()

Définit l'égalité logique entre objets

hashCode()

Fournit empreinte pour tables de hachage

compareTo()

Établit un ordre total cohérent

Le contrat equals() : cinq propriétés



Réflexivité

`x.equals(x)` doit retourner true



Symétrie

Si `x.equals(y)` alors `y.equals(x)`



Transitivité

Si `x.equals(y)` et `y.equals(z)` alors `x.equals(z)`



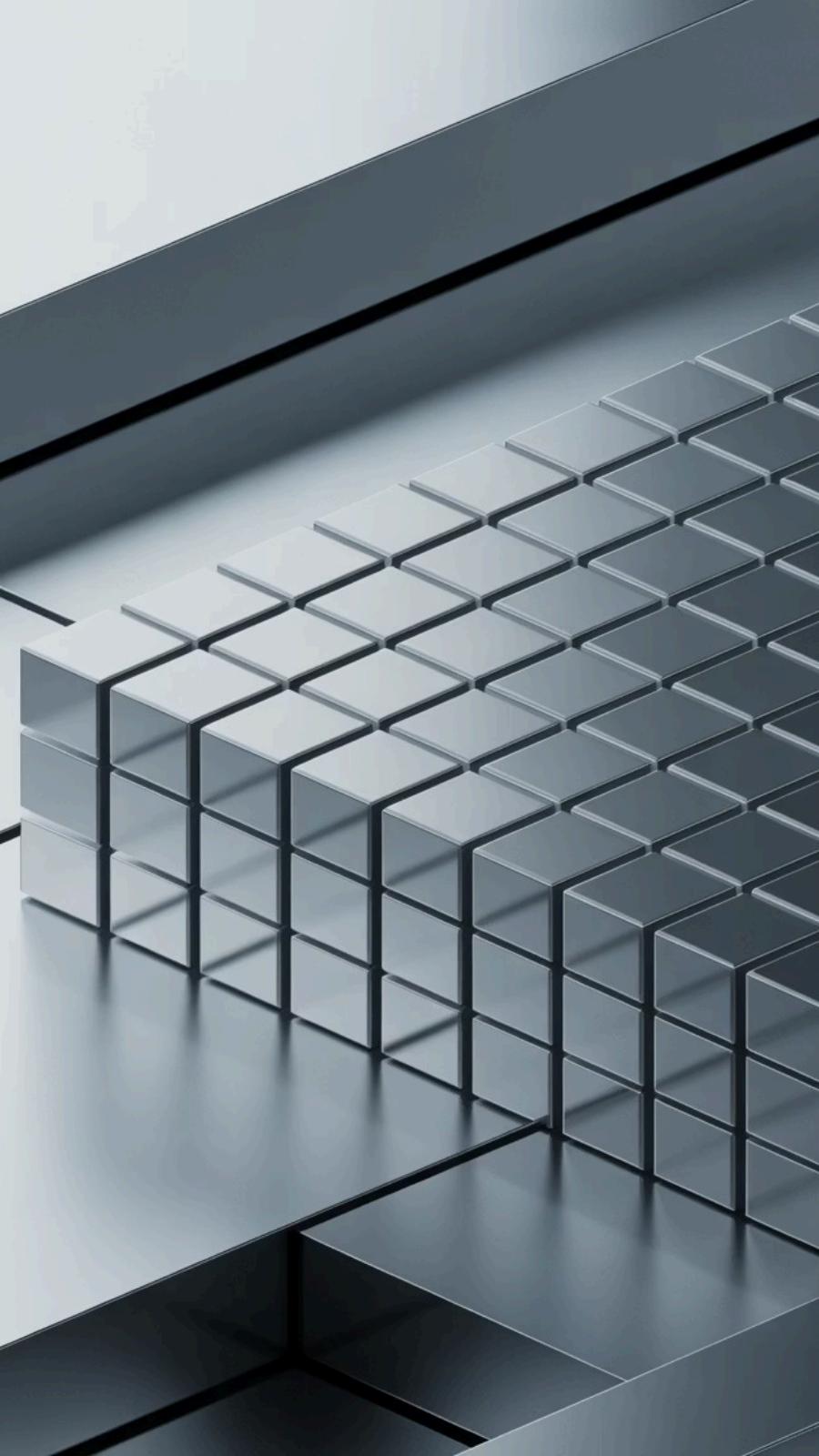
Cohérence

Appels répétés retournent même résultat



Non-nullité

`x.equals(null)` doit retourner false



HashCode : la clé des performances

```
@Override  
public int hashCode() {  
    return Objects.hash(email, username);  
}
```

Mêmes champs que equals()

email et username définissent l'égalité et le hash

Objects.hash() simplifie

Distribution raisonnable garantie

Mauvais hashCode = désastre

HashMap O(1) devient liste chaînée O(n)



CompareTo : établir un ordre total

Contrat Comparable

compareTo() doit être cohérent avec equals()

Si $x.compareTo(y) == 0$, alors $x.equals(y)$ devrait être true

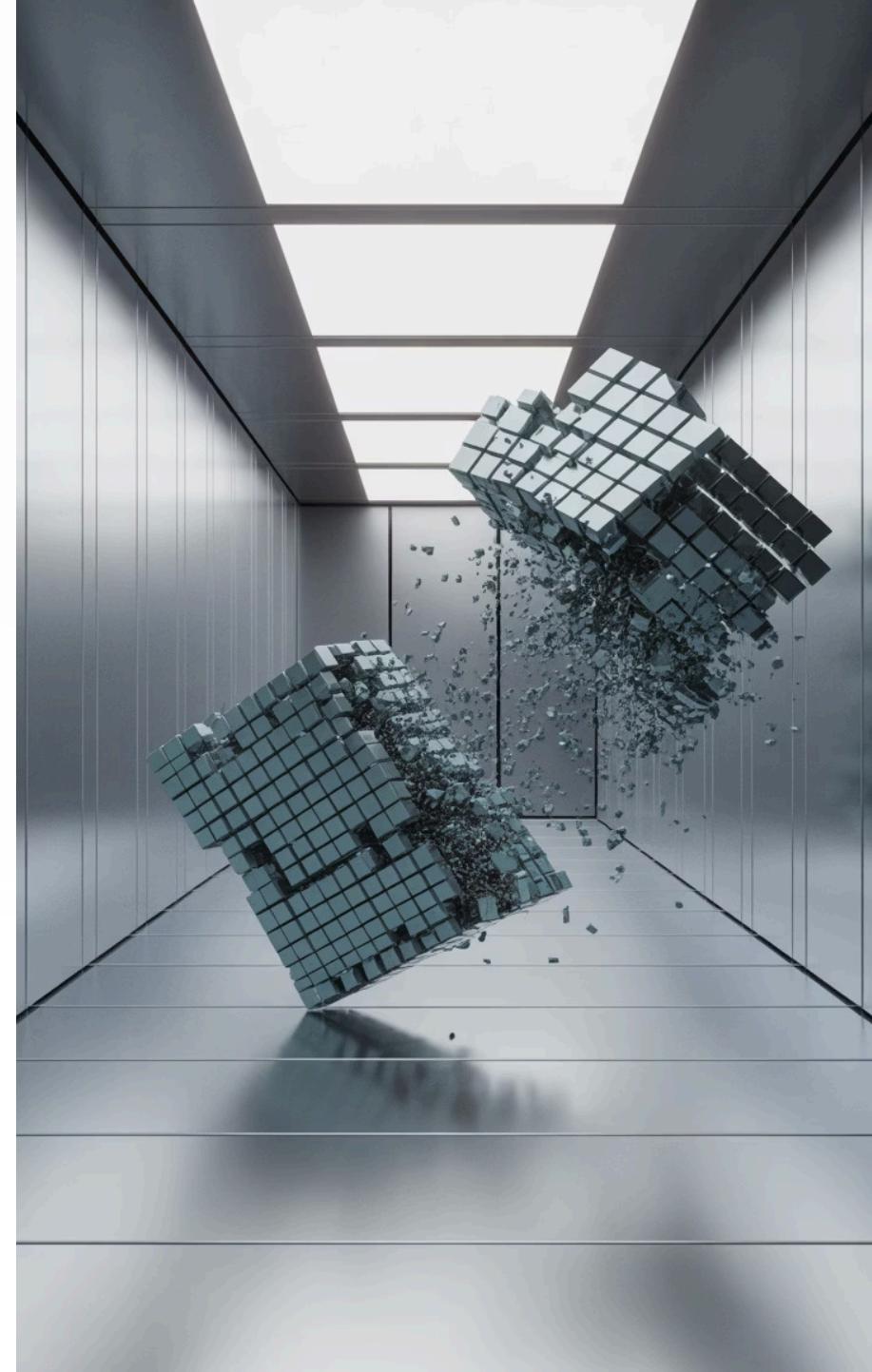
```
public int compareTo(Person o) {  
    int lastNameComp =  
        lastName.compareTo(o.lastName);  
    if (lastNameComp != 0) {  
        return lastNameComp;  
    }  
    return  
        firstName.compareTo(o.firstName);  
}
```

Incohérence equals/hashCode : cas d'école

- **Problème :** Product où equals() compare uniquement SKU, mais hashCode() utilise SKU et nom

```
Product p1 = new Product("SKU123", "Widget A");
Product p2 = new Product("SKU123", "Widget B");
p1.equals(p2); // true (même SKU)
p1.hashCode() == p2.hashCode(); // false !
set.contains(p2); // false ! Bug silencieux
```

Le HashSet cherche dans le bucket du hashCode. Comme p1 et p2 ont des hashCode différents, contains() échoue même si equals() retourne true.



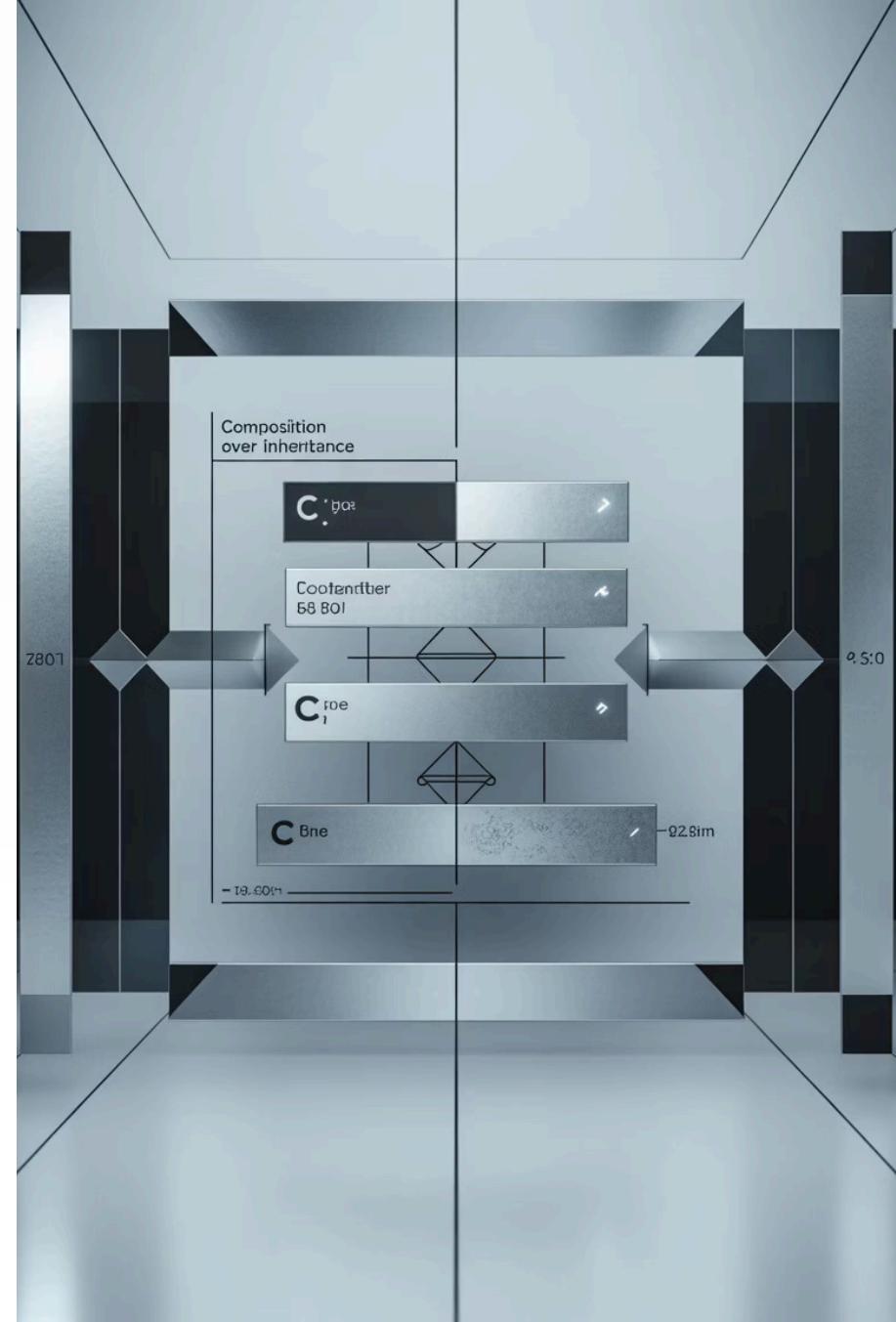
Composition plutôt qu'héritage

Problème de l'héritage

- Couplage fort parent-enfant
- Évolutions difficiles
- Une seule dimension de variation
- Explosion combinatoire de classes

Avantages composition

- Comportements injectés via interfaces
- Combinaisons multiples possibles
- Changement dynamique à l'exécution
- Flexibilité maximale



Le problème de l'explosion combinatoire

3

Formats

PDF, Excel, HTML

2

Niveaux détail

Sommaire, Complet

3

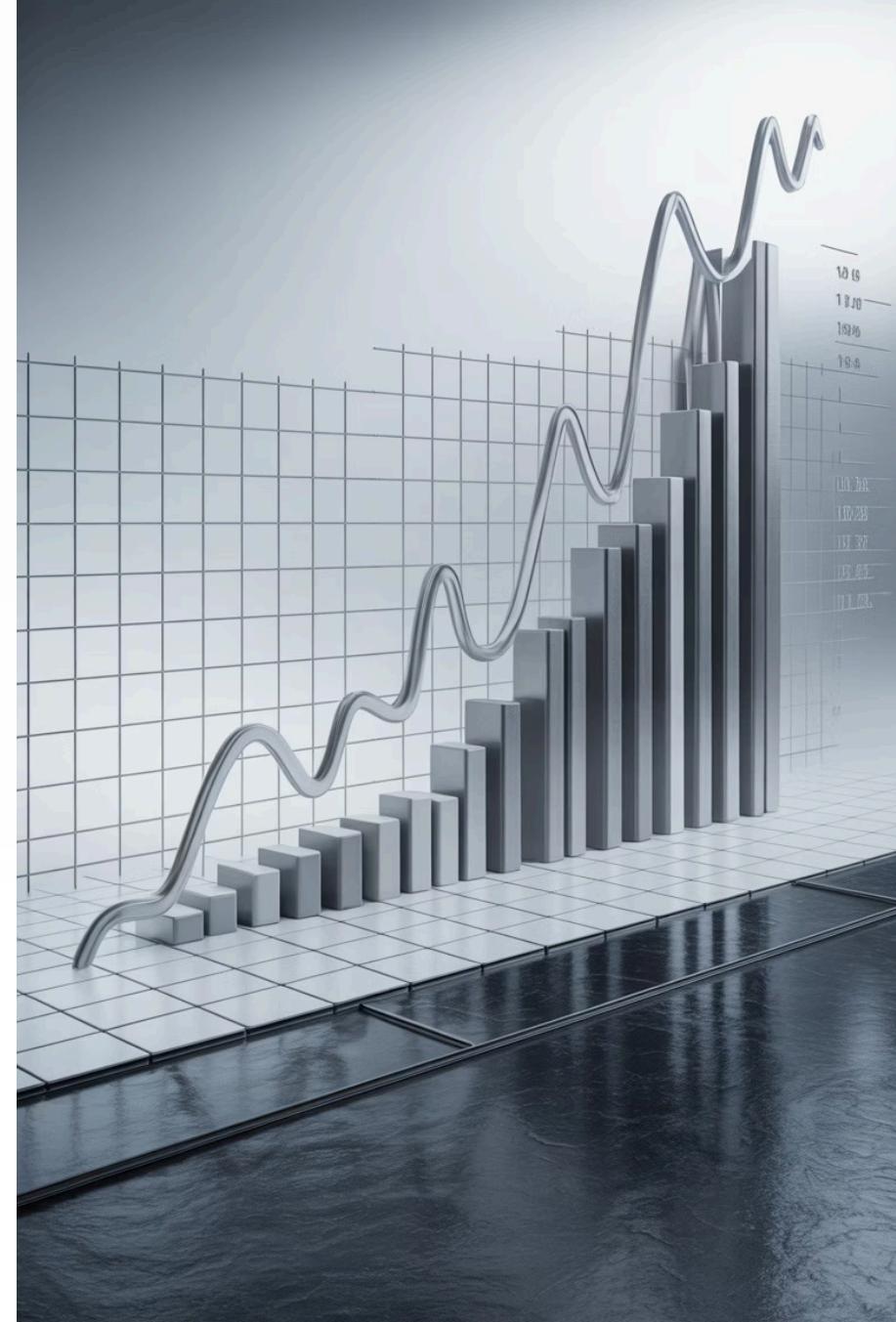
Sources données

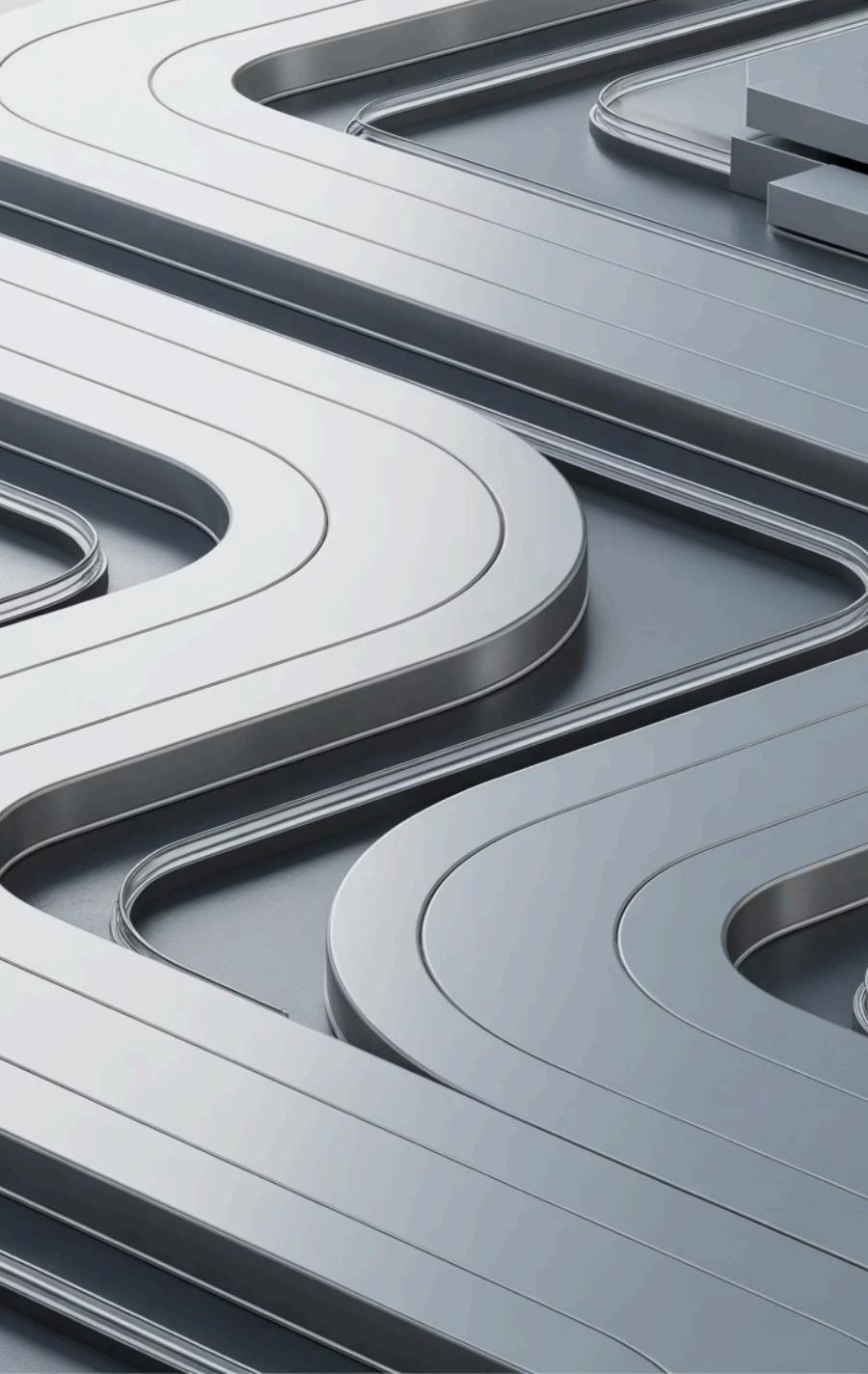
DB, File, API

18

Classes
nécessaires

$3 \times 2 \times 3 = \text{ingérable !}$





Composition : stratégie Renderer

```
public interface Renderer {  
    void render(ReportData data, OutputStream out);  
}  
  
public class Report {  
    private final Renderer renderer;  
  
    public Report(Renderer renderer) {  
        this.renderer = renderer;  
    }  
}
```

Nouveaux formats = nouvelle implémentation Renderer, sans modifier le code existant. Stratégie injectable et testable.



Combinaison flexible des stratégies

```
DataSource source = new DatabaseDataSource(conn);
Renderer renderer = new PDFRenderer();
Report report = new Report(source, renderer);

// Changement à l'exécution
report.setRenderer(new ExcelRenderer());
```

Isolation

Chaque dimension dans sa propre interface

Extension

Nouvelles sources/renderer sans toucher aux autres

Avantages de la composition

Flexibilité

Comportements combinés librement, changement à l'exécution

Découplage

Dépendances inversées vers abstractions, pas implémentations

Testabilité

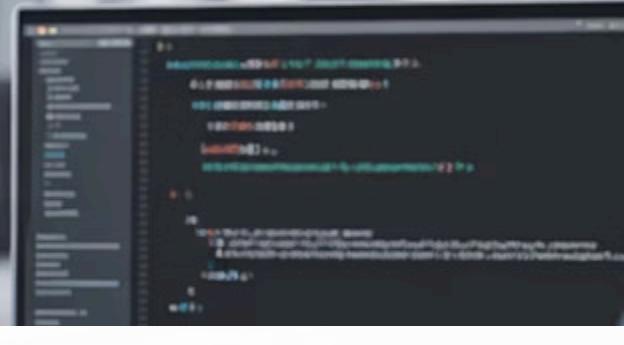
Chaque stratégie testée isolément, mocks faciles

Évolutivité

Nouveaux comportements par ajout, non modification



```
    null = safeEv->::  
    >>  
    moreRootsToAsk->  
    create->  
    checkForState("root does not have a good state", true));  
}  
create->  
(OS::  
>>  
    PointeeMethodPointee beta1->  
    pointeeMethodImplementation(beta1);
```



Optional : gérer l'absence de valeur

Problème avec null

- Aucune information sur l'absence
- NullPointerException différée
- Vérifications répétitives partout
- Ambiguïté sémantique

Solution Optional

- Absence explicite dans le contrat
- Visible à la compilation
- Force gestion des deux cas
- Intention claire

Quand utiliser Optional



Recherches

findUserById(), searchProduct() - absence normale, pas exceptionnelle



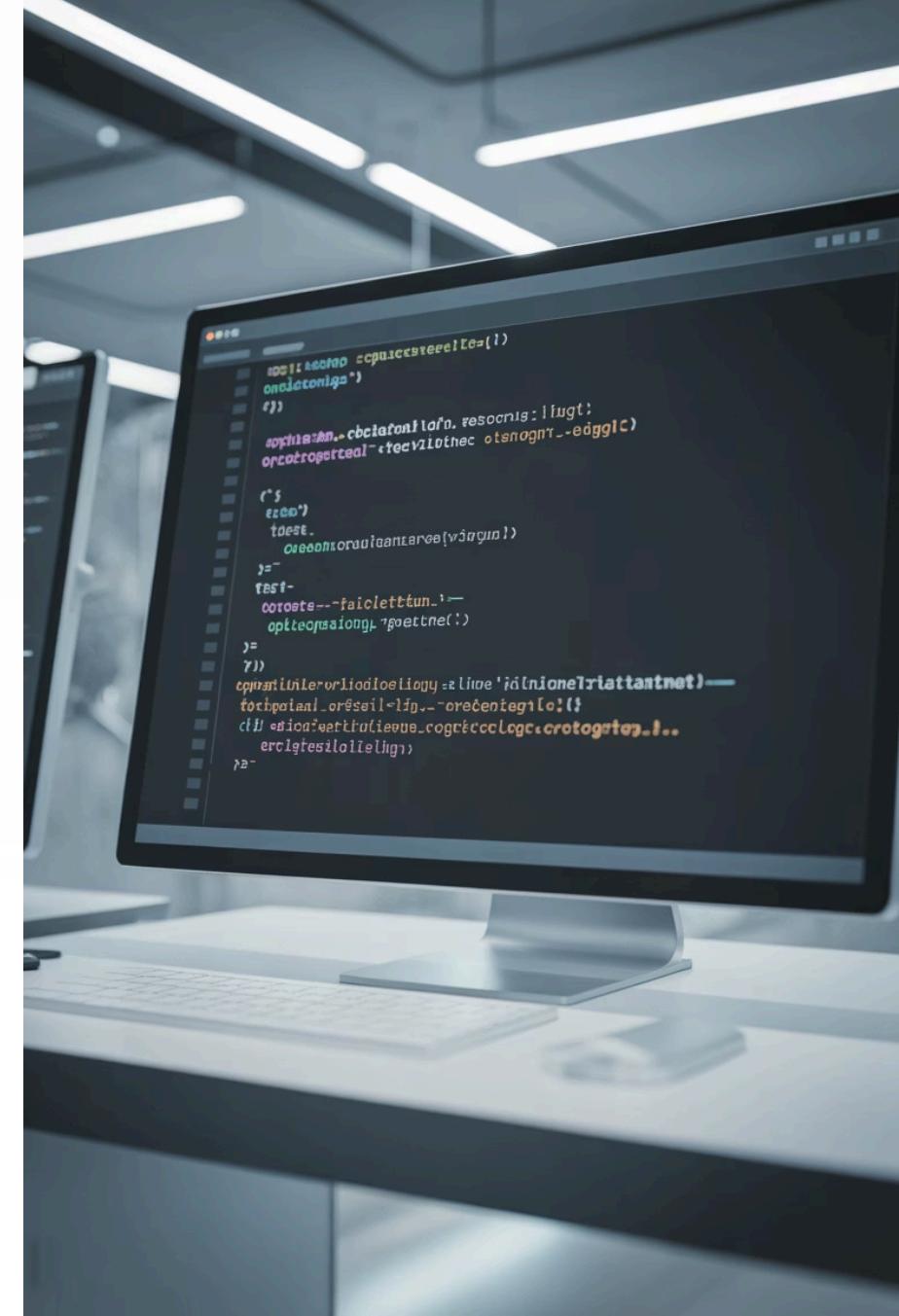
Calculs optionnels

calculateDiscount() retourne Optional si aucune promotion applicable



Configuration

getTimeout() retourne Optional, valeur standard si non configuré



Patterns avec Optional

orElse()

Fournit valeur par défaut si vide

orElseThrow()

Lance exception si vide

map()

Transforme valeur si présente

orElseGet()

Valeur par défaut paresseuse (lazy)

ifPresent()

Exécute action si valeur présente

flatMap()

Transforme en autre Optional, évite Optional<Optional>



Anti-patterns avec Optional

À éviter

```
// Optional comme paramètre  
public void process(Optional  
user)  
  
// Optional.get() sans  
vérification  
User user = opt.get(); // NPE !  
  
// Champs Optional  
private Optional email;
```

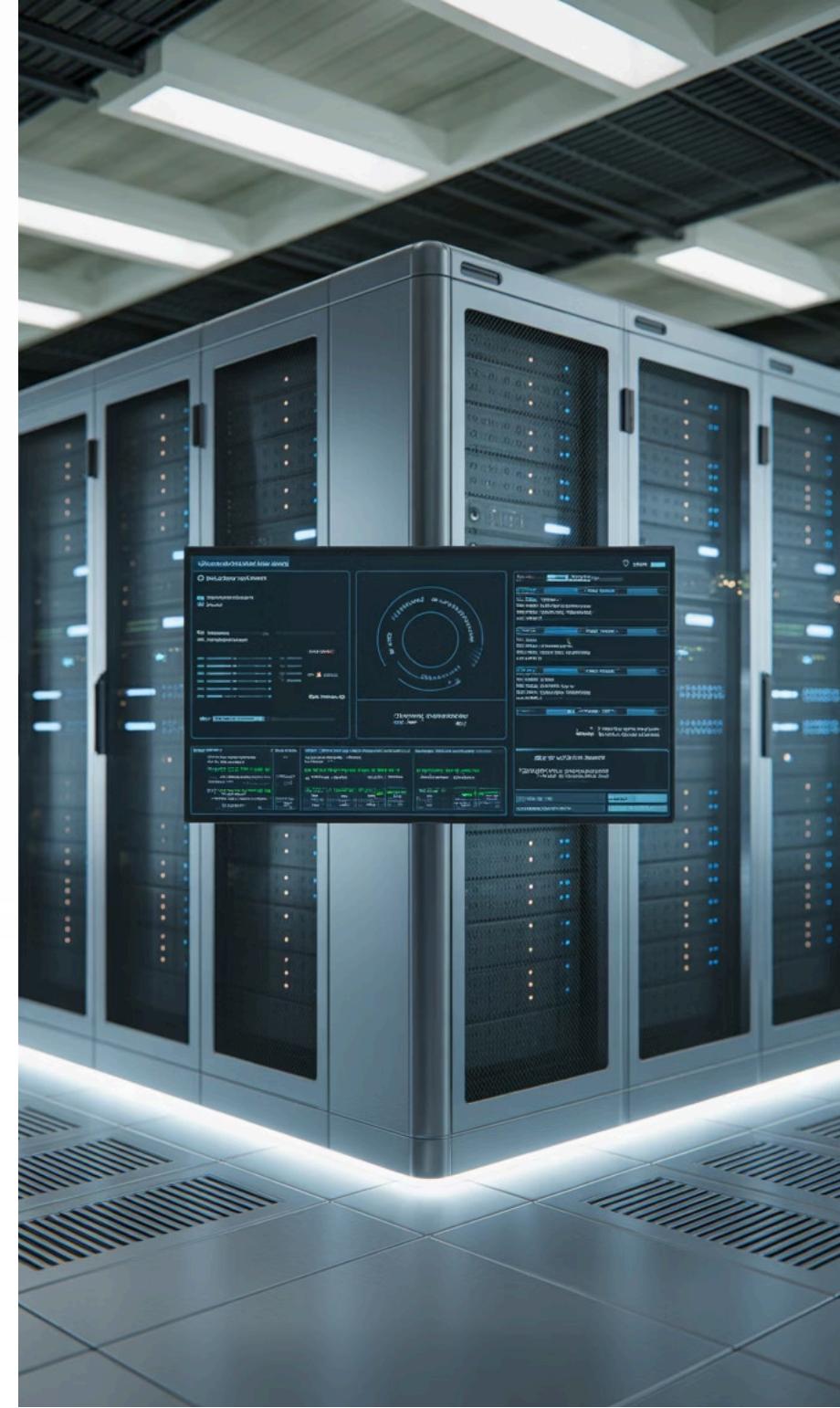
Alternatives

```
// Surcharge ou null  
public void process(User user)  
  
// Toujours vérifier  
opt.ifPresent(this::process);  
  
// Null classique pour champs  
private String email; // null OK
```

Exceptions dédiées : quand l'absence est exceptionnelle

```
public class UserNotFoundException extends RuntimeException {  
    private final UserId userId;  
  
    public UserNotFoundException(UserId userId) {  
        super("User not found: " + userId);  
        this.userId = userId;  
    }  
}  
  
public User getUserId(UserId id) {  
    return userRepository.findById(id)  
        .orElseThrow(() -> new UserNotFoundException(id));  
}
```

Exception dédiée plus informative qu'une générique. Porte données contextuelles et peut être interceptée spécifiquement.



Null : l'ennemi silencieux

Absence d'intention

Null ne communique aucune information sur le pourquoi

NPE différée

Se propage silencieusement, explose loin de l'origine

Vérifications répétitives

Code boilerplate pollue la logique métier

Ambiguïté sémantique

Null = non initialisé ? invalide ? non trouvé ? impossible ?





Command-Query Separation (CQS)

Séparer les questions des ordres

Requêtes (Query)

Observent sans modifier

Retournent une information

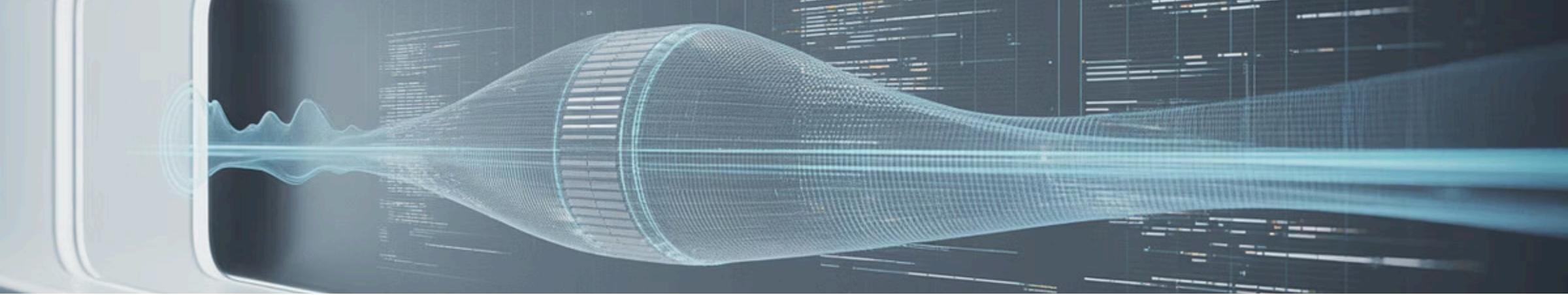
Pas d'effets de bord

Commandes (Command)

Agissent sans retourner

Modifient l'état du système

Retournent void ou statut



Requêtes pures : observer sans perturber



Caractéristiques

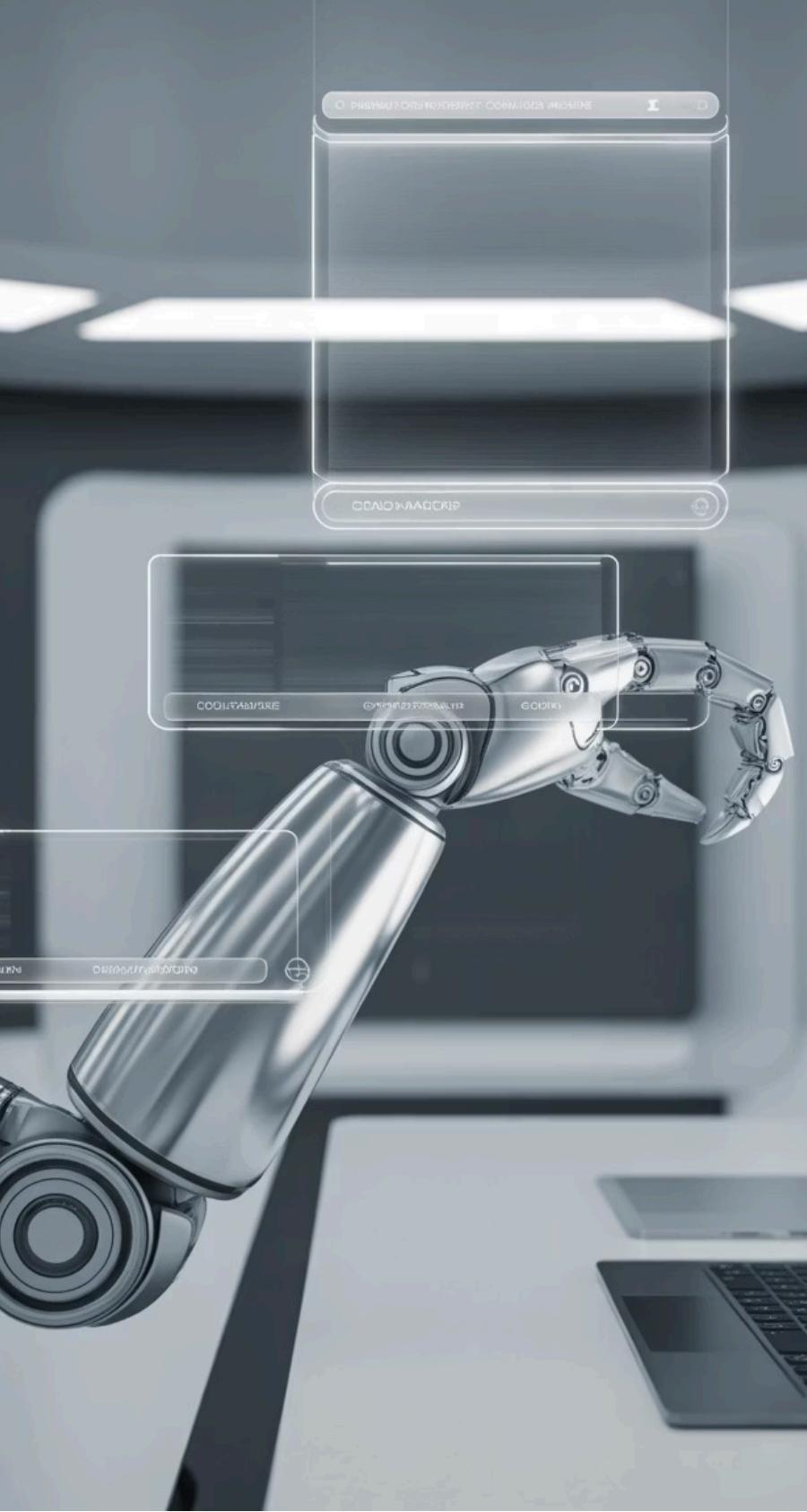
Retournent valeur, ne modifient rien, déterministes, sans effets de bord



Exemples

getBalance(), getName(), size(), isEmpty(), contains()

```
public double getBalance() {  
    return this.solde; // Ne modifie pas l'état  
}
```



Commandes : agir avec intention



Validation

Vérifier préconditions et contraintes métier



Modification état

Appliquer les changements au système



Notification

Informer observateurs si nécessaire

```
public void deposit(double amount) {  
    if (amount <= 0) throw new IllegalArgumentException();  
    this.solde += amount;  
    // Pas de valeur de retour  
}
```

Violation du CQS : un anti-pattern

Problème

```
public User registerUser(  
    String email,  
    String password  
{  
    // Vérifie, crée, sauvegarde  
    // Envoie email  
    // ET retourne données  
    return user;  
}
```

Solution CQS

```
// Commande  
public void registerUser(...)  
  
// Requête séparée  
public User getUserByEmail(  
    String email  
)
```



Bénéfices du CQS



Clarté d'intention

Signature révèle nature : void = commande, type retour = requête



Testabilité améliorée

Requêtes testées par assertions, commandes par vérification d'état



Optimisations possibles

Requêtes pures : cache, mémoïzation, parallélisation

Collections et contrats : éviter les fuites d'état

Collections mutables exposées = brèche dans l'encapsulation. Code externe peut modifier sans validation.

- 1 Copies défensives**
Duplication complète pour sécurité maximale

- 2 Vues non modifiables**
`Collections.unmodifiableList()` protection efficace

- 3 API métier uniquement**
Ne jamais exposer collection directement





Vues non modifiables : protection efficace

```
public class Library {  
    private final List books = new ArrayList<>();  
  
    public List getBooks() {  
        return Collections.unmodifiableList(books);  
    }  
}
```

- ❑ Toute tentative de modification directe entraîne `UnsupportedOperationException`, protégeant l'état interne.



Fail-fast : détecter modifications concurrentes

Itérateurs Java détectent si collection modifiée pendant itération et lèvent ConcurrentModificationException.

modCount

Compteur incrémenté à chaque modification structurelle

expectedModCount

Copie locale dans l'itérateur au moment de création

Vérification

À chaque next(), compare les deux compteurs

Collections immuables : sécurité ultime

Java 9+ : List.of(), Set.of()

```
List immutable =  
    List.of("A", "B", "C");  
// Vraiment immuable
```

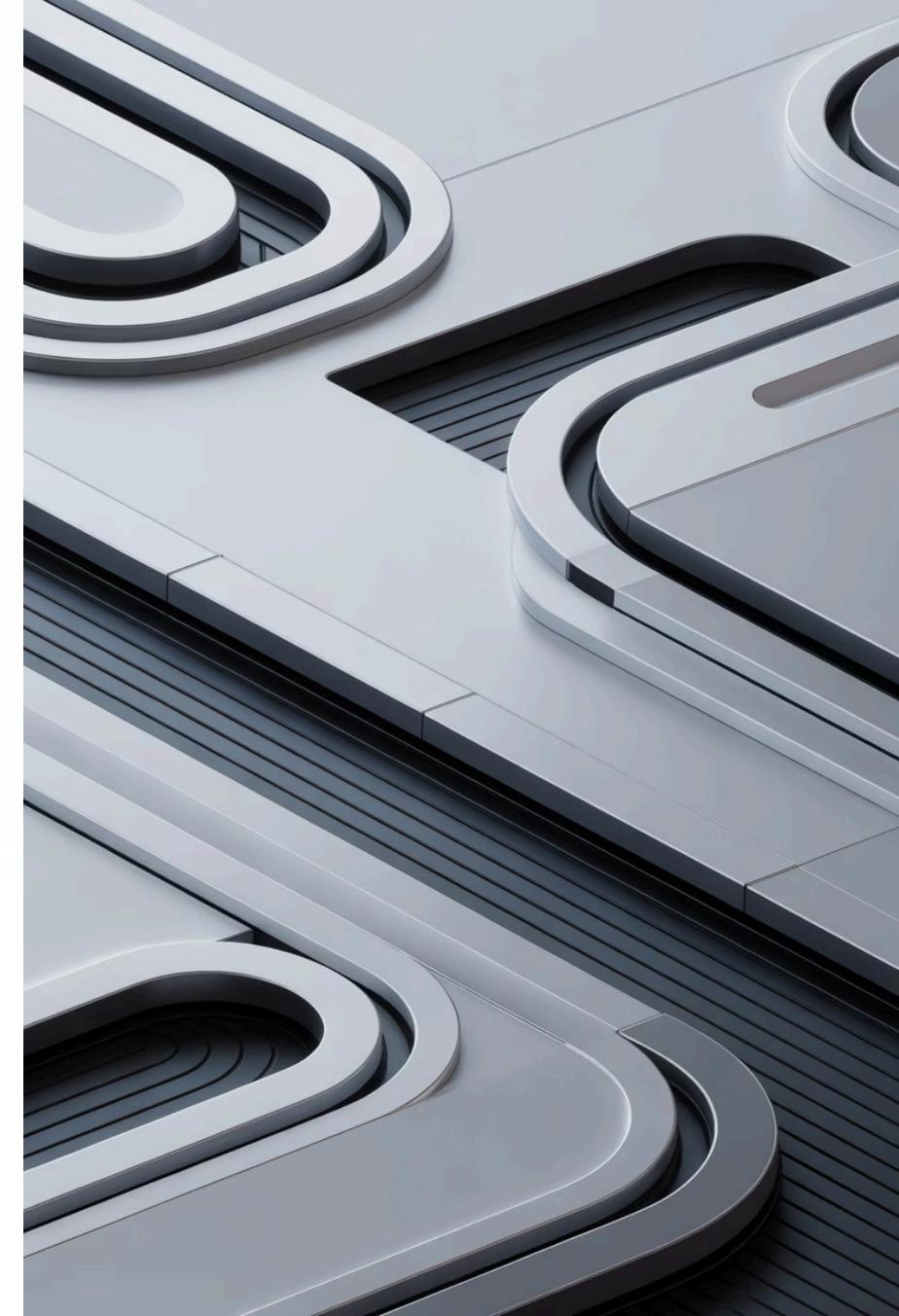
Aucune référence vers structure modifiable

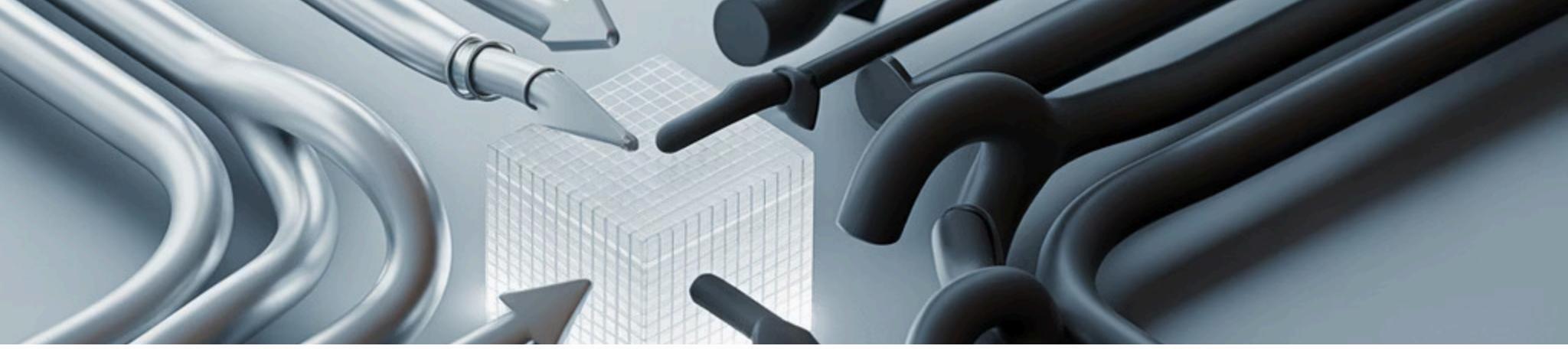
List.of() offre sécurité supérieure et meilleures performances : optimisées pour immuabilité dès création.

Collections.unmodifiableList()

```
List view =  
    Collections.unmodifiableList(  
        mutableList);  
// Vue seulement
```

Modification source affecte la vue





Concurrence : les pièges de la mutabilité

count++ n'est pas atomique !



Lecture

Thread A lit count = 0



Incrémantation

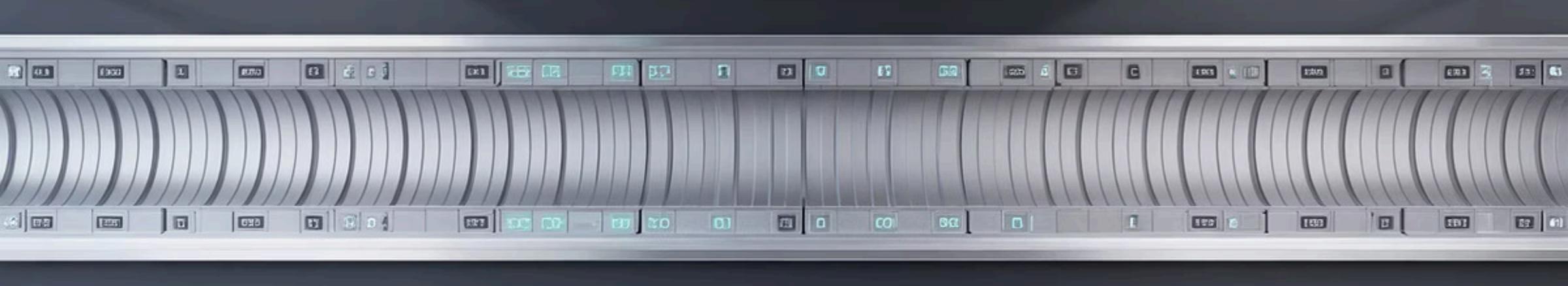
Thread A calcule $0 + 1 = 1$



Écriture

Thread A écrit 1 (mais Thread B a peut-être lu 0 aussi !)

Race condition : mises à jour perdues, résultats imprévisibles



AtomicInteger : atomicité garantie

```
private final AtomicInteger count = new AtomicInteger(0);

public void increment() {
    count.incrementAndGet(); // Opération atomique
}

public boolean compareAndSet(int expected, int newValue) {
    return count.compareAndSet(expected, newValue);
}
```

Toutes opérations effectuées comme unité indivisible. Garantit absence de race conditions.

Encapsuler la politique de thread-safety

1

Décision architecturale

Thread-safe ou non : choix conscient et documenté

2

Encapsulation complète

Pas de fuite des détails de synchronisation

3

Immutabilité préférée

Meilleure stratégie : automatiquement thread-safe

4

Synchronisation minimale

Sections critiques strictement nécessaires



Stratégies de synchronisation



Variables atomiques

AtomicInteger, AtomicBoolean : opérations simples, performances optimales via CAS



ReentrantLock

Besoins avancés : tryLock, timeout, fairness. Plus flexible que synchronized



Synchronized

Sections critiques complexes, plusieurs variables. Simple mais attention contentions



ReadWriteLock

Lectures fréquentes, écritures rares. Plusieurs lecteurs simultanés, un seul écrivain





Collections concurrentes : le bon outil



ConcurrentHashMap

Verrous granulaires par segment. Excellentes performances lectures/écritures concurrentes



CopyOnWriteArrayList

Copie complète à chaque modification. Idéale pour listes rarement modifiées



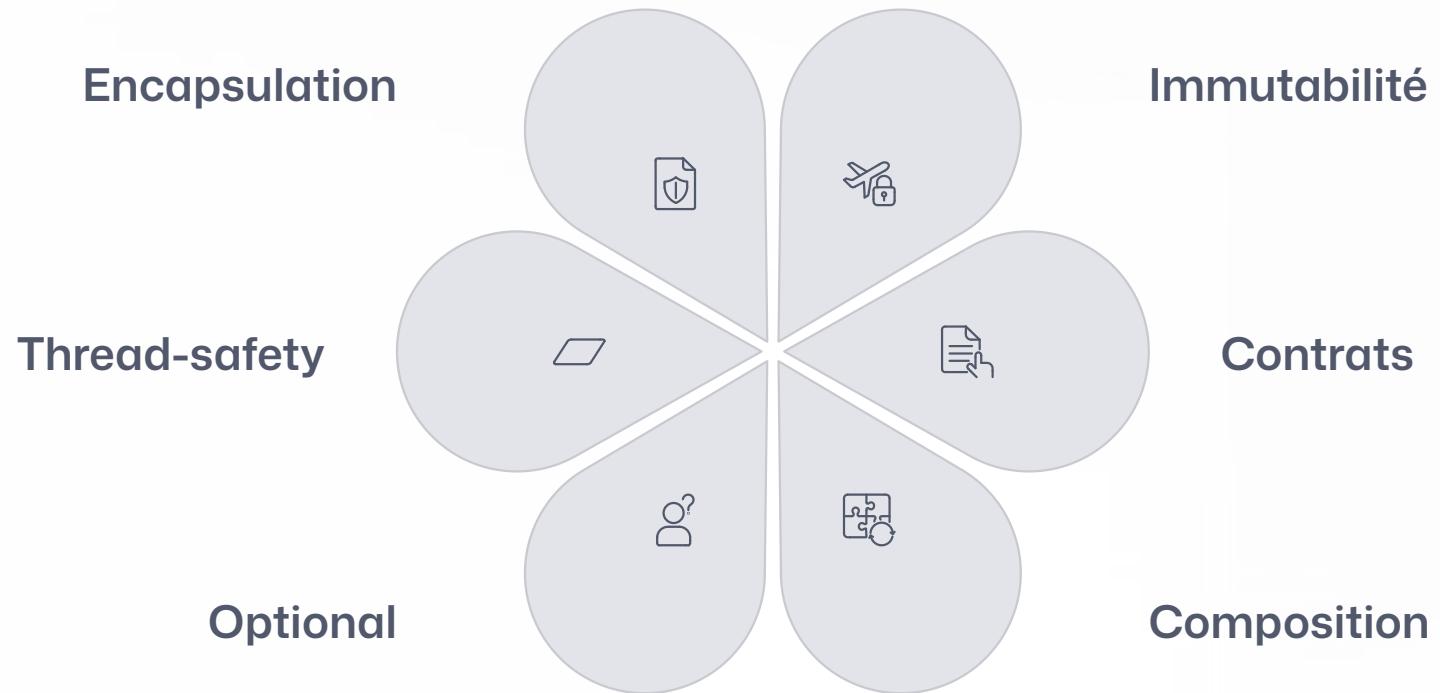
BlockingQueue

Opérations bloquantes. Parfaite pour producteur-consommateur

Conclusion : l'excellence est une pratique

"La qualité n'est jamais un accident ; c'est toujours le résultat d'un effort intelligent."

— John Ruskin



Appliquez ces principes progressivement. Revisitez votre code. Partagez avec votre équipe. Continuez d'apprendre. Bon code à tous !