La conception Objet

Rezak AZIZ
rezak.aziz@lecnam.net

CNAM Paris

Pourquoi une architecture orientée objet ?

- Organiser le code pour qu'il soit **simple, clair et réutilisable**.
- Concevoir des classes qui représentent bien les **éléments du monde réel**.
- Apprendre comment les classes **coopèrent entre elles**.
- Découvrir les **règles de conception** (SOLID, délégation, découplage, etc.).

Objectifs du cours

- Organiser le code pour qu'il soit simple et réutilisable.
- Concevoir des classes représentant fidèlement le monde réel.
- Comprendre comment les classes coopèrent entre elles.
- Découvrir les principes de conception (SOLID, découplage, délégation...).

Attention

La POO n'est pas qu'une question de syntaxe : c'est une **façon de penser la structure du logiciel**, en termes de rôles, de responsabilités et d'interactions.

Références utiles

- Eric Evans, *Domain Driven Design*, 2003
- Martin Fowler, *UML Distilled*, 2003 ;
- Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* 1994
- Barbara Liskov, *Program Development in Java,* 2000
- Robert C. Martin *Clean Architecture*, 2018.
- Bertrand Meyer, *Object Oriented Software Construction,* 1997
- Emmanuel Puybaret, *Les Cahiers du Programmeur Swing,* 2006
- https://martinfowler.com/
- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Les qualités d'un bon logiciel

Un logiciel bien conçu doit être :

- **Fiable** — il fait ce qu'on attend de lui.
- **Facile à développer** — les classes sont testables indépendamment.
- **Facile à modifier** — une petite évolution ne casse pas tout.
- **Réutilisable** — une partie du code peut être réutilisée ailleurs.

Principes d'architecture orientée objet

Qu'est-ce qu'une architecture logicielle ?

Définition

L'**architecture logicielle** définit la structure du système :

- comment les composants sont organisés ;
- comment ils interagissent ;
- comment les responsabilités sont réparties.

Objectif

Favoriser la **clarté**, la **testabilité** et la **maintenabilité**.

Analogie

L'architecture d'un logiciel est à son code ce que les plans sont à un bâtiment : elle guide la construction, les extensions et les réparations.

Attention

Une mauvaise architecture se paie cher : les correctifs deviennent risqués, les tests impossibles, et les évolutions prennent un temps exponentiel.

Les objectifs d'une architecture orientée objet

Principaux buts

- Organiser le code de manière **cohérente et modulaire**.
- Séparer les **rôles et responsabilités**.
- Favoriser la **réutilisation** et la **testabilité**.
- Minimiser les dépendances entre les classes.

En pratique

- Un module = un concept métier clair.
- Chaque classe a un rôle bien identifié.
- Les interactions se font via des interfaces.

Une classe bien définie

Caractéristiques

Une bonne classe :

- représente un seul concept ;
- a une **responsabilité unique** ;
- est **faiblement couplée** aux autres ;
- est **hautement cohésive**.

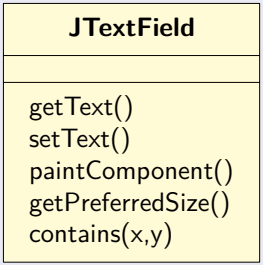
Analogie base de données

Une classe bien conçue est comme une table bien normalisée : chaque entité stocke une information cohérente et indépendante.

Mauvais signe

Si une classe fait « trop de choses » (gestion, affichage, logique, stockage), c'est probablement un *God Object* qu'il faut refactorer.

Exemple : une classe qui fait trop de choses

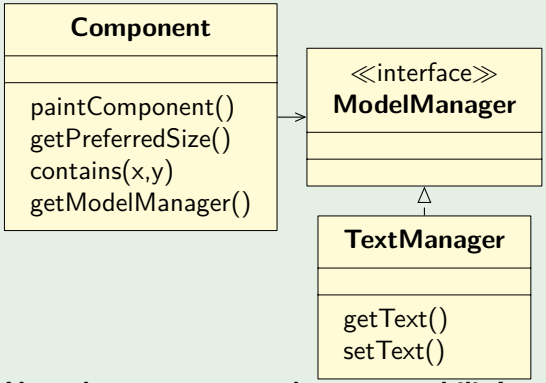


Problème : gère à la fois

- le contenu du texte ;
- l'affichage graphique ;
- la gestion des événements.

→ Une seule classe, trois responsabilités !

Refactorisation proposée



Une classe = une seule responsabilité.

Cohésion et couplage

Cohésion

Mesure la **solidarité interne** d'une classe. Plus ses méthodes sont liées à un même but, plus elle est cohésive.

Couplage

Mesure la **dépendance externe** entre classes. Moins il est fort, plus la classe est réutilisable et testable.

Bon équilibre

- Cohésion élevée : chaque classe a un rôle clair.
- Couplage faible : les classes communiquent via interfaces.

Mauvais signe

Une classe très couplée et peu cohésive est difficile à isoler, tester ou réutiliser. Elle devient un point de fragilité dans l'architecture.

Bonnes pratiques de conception

Principes essentiels

- **Simplicité :** éviter les abstractions inutiles.
- **Clarté :** nommer les classes selon leur rôle métier.
- **Découplage :** limiter les dépendances directes.
- **Testabilité :** concevoir pour faciliter les tests unitaires.

Bon réflexe

Avant d'écrire une nouvelle méthode : se demander "appartient-elle vraiment à cette classe ?"

Conseil

Favoriser la **composition** plutôt que l'héritage : elle réduit le couplage et améliore la flexibilité du code.

Types Abstraits de Données (TAD)

Définition d'un Type Abstrait de Données

Qu'est-ce qu'un TAD ?

Un **Type Abstrait de Données (TAD)** décrit un type par **ce qu'il fait**, pas par **comment il le fait**.

- Définit les opérations disponibles.
- Spécifie leurs comportements (pré/postconditions).
- Cache les détails d'implémentation.

Exemples de TAD

- `Pile`, `File`, `Tableau`, `Liste`.
- En POO : les `interfaces` jouent le même rôle.

Attention

Le TAD ne s'intéresse pas à la structure interne : c'est une **abstraction comportementale**, pas une implémentation.

Objectifs du TAD

Pourquoi utiliser un TAD ?

- Améliorer la **réutilisabilité** du code.
- Faciliter la **preuve de correction**.
- Favoriser la **stabilité du code client**.
- Permettre plusieurs implémentations pour une même interface.

Avantage majeur

On peut changer le code interne sans casser les programmes qui utilisent le TAD. $\rightarrow$ *Le contrat reste inchangé.*

Principe clé

En POO, on manipule des objets via leur **interface**, pas via leur implémentation concrète.

Spécification formelle d'un TAD

Spécifications formelles

Chaque opération est décrite par :

- une **précondition** — ce qui doit être vrai avant l'appel ;
- une **postcondition** — ce qui est garanti après l'appel ;
- un **invariant** — ce qui reste toujours vrai.

Langage de description

On utilise souvent la logique mathématique ou la notation pseudo-code.

Exemple : Tableau de réels

Opérations :

- `creer(taille : entier)`
- `get(i : entier)`
- `set(i : entier, v : double)`
- `taille()`

Spécifications :

- `creer(a)` : $a \geq 0$, crée un tableau de taille a rempli de 0.
- `get(i)` : $0 \leq i < taille()$.
- `set(i, v)` : $0 \leq i < taille()$ et après appel, $get(i) = v$.

Abstraction vs Implémentation

Séparer le "quoi" du "comment"

- L'abstraction décrit **ce que fait** le type.
- L'implémentation décrit **comment il le fait**.
- Cette séparation garantit la flexibilité et la stabilité.

Conséquence

On peut changer l'implémentation interne sans modifier le code utilisateur.

Exemple : TAD Pile

- Interface : `empiler()`, `depiler()`, `estVide()`.
- Implémentation 1 : tableau fixe.
- Implémentation 2 : liste chaînée.

L'utilisateur n'a pas besoin de savoir laquelle est utilisée.

Principe de conception

Toujours **programmer contre une abstraction**, jamais contre une implémentation concrète.

Les TAD et la Programmation Orientée Objet

Correspondance conceptuelle

En POO :

- Un TAD correspond à une **classe abstraite** ou une **interface**.
- Les implémentations concrètes sont des **sous-classes**.

Exemple en Java

- Interface : `List<E>`
- Implémentations : `ArrayList<E>`, `LinkedList<E>`

Les méthodes sont définies dans l'interface et implémentées différemment.

Avantage

La POO fournit naturellement les mécanismes d'encapsulation et de polymorphisme nécessaires aux TAD.

À retenir

Un bon design objet commence toujours par une réflexion sur les abstractions disponibles.

Les principes SOLID

Introduction aux principes SOLID

Origine

Les principes **SOLID** ont été formalisés par **Robert C. Martin** (“Uncle Bob”) pour guider la conception objet.

Objectif

Créer des logiciels :

- plus **maintenables**,
- plus **évolutifs**,
- plus **testables**.

Acronyme

S — Single Responsibility Principle
O — Open/Closed Principle
L — Liskov Substitution Principle
I — Interface Segregation Principle
D — Dependency Inversion Principle

S — Single Responsibility Principle (SRP)

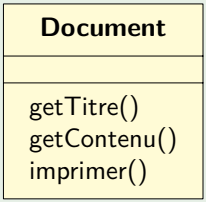
Principe

Une classe ne doit avoir **qu'une seule raison de changer**. → Elle doit assumer une seule responsabilité.

Motivation

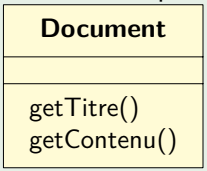
- Limite l'impact des modifications.
- Facilite la compréhension du code.
- Encourage la réutilisation.

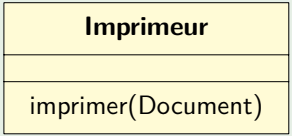
Exemple



`Deux Résponsabilités`

Devrait être plutôt séparer en deux classes:





O — Open/Closed Principle (OCP)

Principe

Les entités logicielles doivent être :

- **ouvertes à l'extension**,
- **fermées à la modification**.

But

Permettre d'ajouter de nouvelles fonctionnalités sans casser le code existant.

Exemple

Mauvais : Une méthode `calculerSalaire()` avec 10 `if` selon le type d'employé.

Bon : Définir une interface `Employe` avec `calculerSalaire()`, et spécialiser pour chaque type d'employé.

Bonnes pratiques

- Définir clairement l'interface publique d'une classe.
- Éviter d'exposer les détails d'implémentation dans la partie publique.
- Prévoir, lorsque c'est pertinent, des mécanismes d'extension.

L — Liskov Substitution Principle (LSP)

Principe

Une sous-classe doit pouvoir remplacer sa super-classe sans altérer le comportement attendu du programme.

Formulation

Si S est un sous-type de T, alors tout objet de type T peut être remplacé par un objet de type S sans erreur.

Exemple

Classe `Oiseau` avec méthode `voler()`.
Classe `Autruche` hérite de `Oiseau` mais ne vole pas.

$\rightarrow$ Violation du LSP.

Solution : introduire une interface `Volant`.

REVOIR COURS PRÉCÉDENT

Règles pratiques

- Une méthode redéfinie ne doit pas restreindre ses arguments.
- Elle peut limiter le type des valeurs retournées (**covariance**).

I — Interface Segregation Principle (ISP)

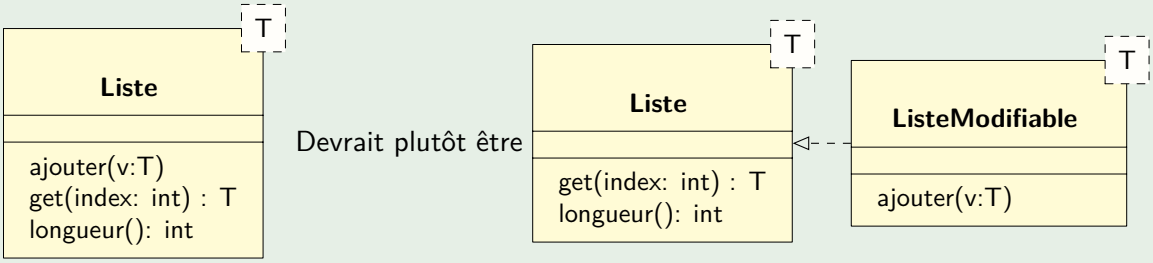
Principe

Il vaut mieux avoir plusieurs interfaces spécifiques qu'une seule interface générale et surchargée.

Conséquence

Les clients peuvent voir uniquement des méthodes dont ils ont besoin. Un client qui ont le droit de lire une liste n'a pas besoin d'une méthode ajouter.

Exemple



D — Dependency Inversion Principle (DIP)

Principe

- Les modules de haut niveau ne doivent pas dépendre de modules bas niveau.
- Les deux doivent dépendre d'abstractions.

Idée clé

Coder pour des **interfaces**, pas pour des classes concrètes.

Conséquence

Ce principe favorise :

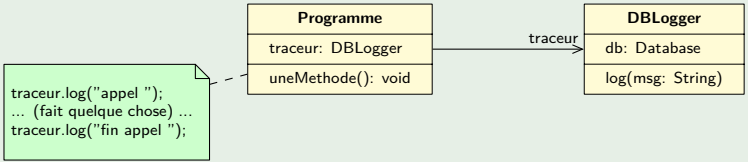
- le **découplage** des composants,
- la **testabilité** (injection de mocks),
- la **flexibilité** (changement d'implémentation).

D — Dependency Inversion Principle (DIP)

Dépendre des abstractions, pas des implémentations

Exemple : conception naïve

La classe `Programme` enregistre les actions de l'utilisateur dans une base via `DBLogger`.

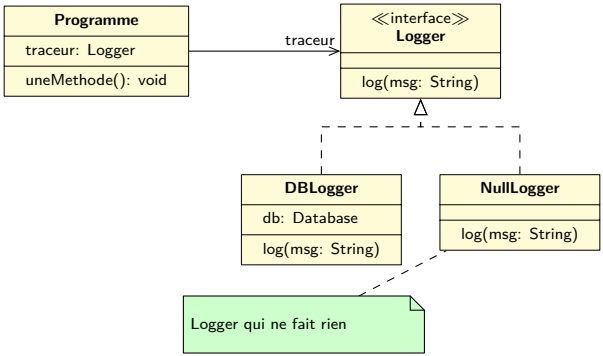


Problèmes

- Impossible de tester sans une vraie `Database`.
- Difficile de changer la destination des logs.

D — Dependency Inversion Principle (DIP)

Solution : dépendre d'une interface



- **Avantage :** conception plus souple et facilement testable.
- La classe `Programme` ne dépend plus d'une implémentation concrète (`DBLogger`).
- **Inconvénient :** légère complexité supplémentaire.

Synthèse des principes SOLID

En résumé

- **S** — Une classe = une responsabilité.
- **O** — Étendre sans Casser/Modifier.
- **L** — Les sous-classes doivent être substituables.
- **I** — Interfaces fines et spécifiques.
- **D** — Dépendre d'abstractions.

Impact global

L'application devient :

- plus **robuste**,
- plus **évolutive**,
- plus **testable**,
- et plus **facile à comprendre**.

Transition

Dans la partie suivante, nous verrons comment appliquer ces principes via la **délégation** et le **découplage**.

Délégation, découplage

Définition de la délégation

Principe

La **délégation** consiste à confier à un autre objet la responsabilité d'exécuter une tâche spécifique. → Un objet ne fait pas tout lui-même, il *délègue*.

Objectif

- Réduire la complexité d'une classe.
- Favoriser la spécialisation et la réutilisation.
- Limiter le couplage entre les composants.

Exemple simple

Une classe `Voiture` ne calcule pas directement la vitesse : elle délègue cette responsabilité à un `Moteur`. `Voiture.getVitesse()` appelle `moteur.calculerVitesse()`.

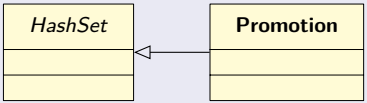
Avantage clé

La délégation permet de **remplacer un composant sans impacter les autres**. → Base du principe de substitution et du découplage.

Délégation vs Héritage

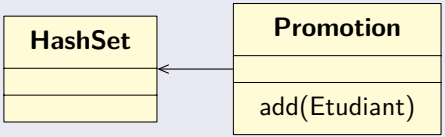
Héritage

- Relation “est un” (*is a*).
- Réutilisation de code via la hiérarchie de classes.
- Risque de couplage fort et rigidité.



Délégation

- Relation “a un” (*has a*).
- Un objet utilise un autre pour une tâche.
- Plus souple et modulaire.



À retenir

Favorisez la composition à l’héritage. L’héritage exprime un lien fort ; la délégation, un lien flexible.

Réduction du couplage

Définition

Le **couplage** mesure la dépendance entre deux classes :

- Fort couplage → modification contagieuse.
- Faible couplage → indépendance et modularité.

But du découplage

- Faciliter la maintenance et les tests.
- Permettre des évolutions sans réécriture globale.

Exemple UML

Si `Application` dépend directement de `Affichage`, toute modification du GUI affecte la logique métier.

Solution : introduire une interface `IAffichage`.

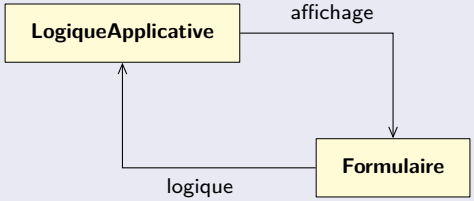
Principe

Découpler les composants via des **interfaces** ou des **classes abstraites** plutôt que des dépendances concrètes.

Suppression technique du couplage

Situation typique

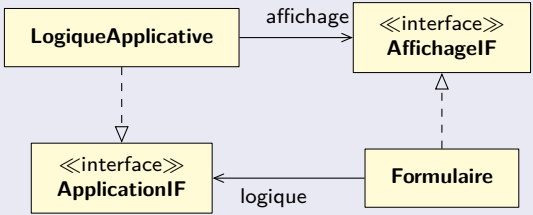
Deux classes dépendent l'une de l'autre :



→ Problème : cycle de dépendance.

Solution

Introduire des **interfaces croisées** :



→ Chaque classe ne dépend plus que d'une abstraction.

Résultat

Découplage total : chaque composant devient interchangeable, favorisant les tests et la modularité.

Avantages du découplage par interfaces

Bénéfices

- Réduction de la dépendance entre modules.
- Meilleure testabilité (injection de mocks).
- Facilite la maintenance et l'évolution.
- Prépare à des architectures multi-couches (MVC, hexagonale...).

Exemple concret

Dans une architecture métier :
`ServicePatient` dépend de l'interface `PatientRepository`.
→ On peut injecter :

- une version `PostgresPatientRepo`,
- une version `MockPatientRepo` pour les tests.

Bon réflexe

Toujours **programmer contre une abstraction**, jamais contre une implémentation.

Principes avancés de conception

Introduction aux principes avancés

Objectif

Après les principes SOLID, d'autres règles guident la conception d'un logiciel **pragmatique et durable**.

Trois principes clés

- **Loi de Déméter** — limiter les dépendances indirectes.
- **YAGNI** — éviter l'over-engineering.
- **Leaky Abstraction** — gérer les abstractions imparfaites.

Enjeux pratiques

Ces principes permettent de :

- Simplifier les architectures.
- Réduire les effets de bord.
- Concevoir des systèmes plus robustes et évolutifs.

La Loi de Déméter

Principe

« Ne parlez pas aux inconnus ». Une méthode M d'un objet O ne doit interagir qu'avec :

- d'autres méthodes de O;
- des méthodes des paramètres de M ;
- des méthodes des objets qu'elle crée directement.

Objectif

Réduire les dépendances en chaîne et renforcer l'encapsulation.

Conséquence

Ce principe améliore la lisibilité et protège le code contre les changements structurels internes.

Exemple

Mauvais : `client.getCompte().getBanque().getAdresse()`
Bon : `client.getAdresseBanque()` (la méthode interne délègue la requête au bon objet).

YAGNI – You Aren't Gonna Need It

Principe

Ne pas implémenter une fonctionnalité tant qu'elle n'est pas nécessaire. (Avoir une méthodologie Agile)

Objectifs

- Réduire la complexité inutile ;
- Favoriser l'adaptabilité ;
- Prévenir la sur-ingénierie et la dette technique.

Exemple

Mauvais : prévoir une architecture de plug-ins avant d'avoir un seul module fonctionnel.
Bon : implémenter le besoin réel, puis généraliser si cela devient nécessaire.

À retenir

Faire simple tant que possible : le code inutile est une source potentielle de bugs.

Leaky Abstraction – L'abstraction qui fuit

Principe

Une **Leaky abstraction** est une abstraction dont des détails d'implémentation deviennent visibles à travers son comportement.

Symptômes

- Nécessité de connaître la structure interne ;
- Effets de bord imprévus ;
- Dépendance du client à des détails techniques.

Exemples

Classe `Personne` exposant un objet mutable :

```
class Personne {
  private Date dateNaissance;
  public Date getDateNaissance() {
    return dateNaissance; // fuite !
  }
}
```

L'appelant peut modifier la date interne :

```
p.getDateNaissance().setYear(3000);
```

Recommandation

Faire un copie défensive

```
return new Date(dateNaissance.getTime());
```