# Introduction aux Design Patterns

Rezak AZIZ
rezak.aziz@lecnam.net

CNAM Paris

# Qu'est-ce qu'un *Design Pattern* ?

## Définition

Un **Design Pattern** est une **solution classique** à un **problème récurrent** de conception logicielle.

## Principe fondamental

Un Pattern agit comme un *plan conceptuel*. Il décrit :

- les éléments en jeu ;
- leurs relations ;
- et la manière dont ils collaborent pour résoudre un problème donné.

## Utilisation en conception

- Les Patterns ne sont pas des bibliothèques toutes prêtes.
- Ils guident la **pensée architecturale** avant l'implémentation.
- Un même Pattern peut donner lieu à plusieurs implémentations concrètes.

## À retenir

Les Patterns ne sont pas du code à copier, mais des **schémas de conception réutilisables et adaptables**.

# Histoire des *Design Patterns*

## Origine du concept

Le terme **Design Pattern** ne désigne pas une invention soudaine, mais la formalisation de solutions déjà utilisées depuis longtemps.

## Première apparition

L'idée de "Pattern" vient de l'architecte **Christopher Alexander**, dans son ouvrage *A Pattern Language: Towns, Buildings, Construction* (1977). Il y décrit un **langage de conception urbaine** fondé sur des unités récurrentes : les *Patterns*.

## Transposition à l'informatique

En 1994, quatre auteurs : **Erich Gamma, John Vlissides, Ralph Johnson, et Richard Helm**, publient l'ouvrage fondateur : *Design Patterns: Elements of Reusable Object-Oriented Software*.
Ce livre, plus connu sous le nom de **"Gang of Four" (GoF)**, présente **23 Patterns** pour la conception orientée objet.

Depuis, de nombreux autres Patterns ont été découverts.

# Pourquoi apprendre les *Design Patterns* ?

## Une pratique implicite

- Il est possible de développer pendant des années sans avoir étudié les Design Patterns.
- la plupart des développeurs les utilisent déjà **sans le savoir**.

## Pourquoi alors les étudier ?

- Comprendre et **nommer** les schémas que vous appliquez déjà ;
- Pour améliorer votre capacité à **concevoir, refactoriser et discuter** du code.

## Une boîte à outils conceptuelle

Une véritable **boîte à outils de solutions fiables** face aux problèmes classiques de la conception orientée objet. Ils enseignent

- séparation des responsabilités,
- faible couplage,
- réutilisation et extensibilité.

## Un langage commun

Les Patterns offrent un vocabulaire partagé entre développeurs. Dire : « Utilisons un *Singleton* » suffit pour se comprendre, sans avoir à redécrire toute la structure du code.

# Patterns, algorithmes et structure d'un Pattern

## Pattern $\neq$ Algorithme

- Un **algorithme** décrit une suite d'étapes précises menant à un résultat.
- Un **Pattern** décrit une organisation de classes et d'objets, adaptable selon le besoin.

## Structure type d'un Pattern

Chaque Pattern est généralement décrit selon les sections suivantes :

- **Intention** - Objectif ou Role.
- **Problème** - Contexte et Motivations.
- **Solution**
- **Structure** — classes, rôles, relations ;
- **Exemple de code**
- **Possibilité d'application** – Quand ?
- **Avantages et Inconvénients**
- **Liens pratiques** — variantes et relations avec d'autres Patterns.

# Exemple de Design Pattern : Singleton

## Intention

Le **Singleton** est un Design Pattern de création qui garantit qu'une classe ne possède qu'une seule instance dans le système, tout en fournissant un **point d'accès global** à cette instance.

## Objectif

- Éviter la duplication d'un objet unique (ex. connexion à une base de données, logger).
- Offrir un accès centralisé et contrôlé à cette instance.

# Problème traité par le Singleton

## Problèmes

Le Singleton résout deux problèmes majeurs :

1. **Garantir l'unicité d'une instance** pour les classes gérant des ressources partagées.
2. **Offrir un accès global sécurisé** à cette instance.

## Difficulté

Un constructeur classique crée toujours un nouvel objet.

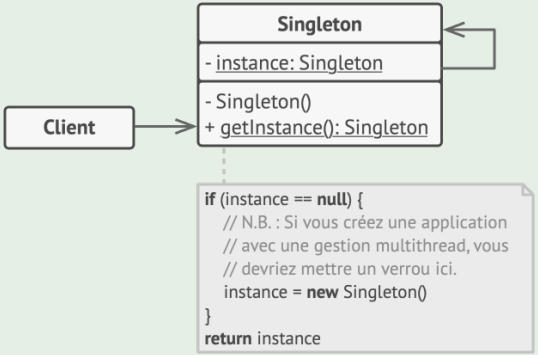
# Solution proposée

## Principe du Singleton

1. Rendre le constructeur **privé** pour bloquer l'instanciation externe.
2. Fournir une méthode **statique** `getInstance()` qui crée et retourne l'unique instance.
3. Conserver cette instance dans un attribut **statique**.

# Structure du Singleton

## Diagramme simplifié




Le code client ne peut pas créer d'objet directement : il accède à l'unique instance via la méthode `getInstance()`.

```
class Database {
    private static Database instance;

    private Database() {}
    public static Database getInstance() {
        if (instance == null) {
            synchronized(Database.class) {
                if (instance == null) {
                    instance = new Database();
                }
            }
        }
        return instance;
    }
}
```

# Possibilités d'application

- Lorsqu'une classe doit posséder une seule instance (ex. base de données, logger, gestionnaire de configuration).
- Lorsqu'un accès global contrôlé est nécessaire.
- Pour remplacer une variable globale par un mécanisme sûr.

# Avantages et inconvénients

## Avantages

- Garantit l'unicité de l'instance.
- Offre un point d'accès global.
- Permet une instanciation paresseuse.

## Inconvénients

- Ne respecte pas le principe de responsabilité unique (deux problèmes pour la base de données : unicité de l'accès global et logique métier (connexion, requete).
- Rend les tests unitaires plus difficiles.
- Doit être protégé contre le multithreading.
- Peut dissimuler une conception trop couplée.

# À surveiller : critiques des *Design Patterns*

## 1. Bidouilles pour de mauvais langages

Certains Patterns servent à compenser les limites d'un langage.

- Dans les langages modernes, plusieurs patterns deviennent superflus.

## 2. Solutions inefficaces

Appliquer un Pattern "à la lettre" sans adapter au contexte conduit à des architectures rigides et inutilement complexes.

## 3. Utilisation injustifiée

*« Si tout ce que vous avez est un marteau, tout ressemble à un clou. »* Les débutants veulent souvent appliquer des Patterns partout, même quand du code simple suffit.

## À retenir

Les Patterns sont des **outils**, pas des obligations. Ils doivent être compris, adaptés et utilisés avec discernement.

# Les grandes familles de Design Patterns

## Classification générale

Les Design Patterns sont classés selon **leur objectif principal** dans la structure d'un logiciel. On distingue traditionnellement trois grandes familles.

### 1. Patterns de création

Fournissent des mécanismes de création d'objets qui augmentent la flexibilité et la réutilisation du code. Examples : Singleton, Fabrique, Monteur, Prototype

### 2. Patterns structurels

Comment assembler des objets et des classes en de plus grandes structures, tout en les gardant flexibles et efficaces. Examples : Decorateur, Composite, Façade, Adaptateur, Pont, Procuration,...

### 3. Patterns comportementaux

Mettent en place une communication efficace et répartissent les responsabilités entre les objets. Examples : Visiteur, Template Method, Observer, Strategy, ...

# Les patterns de Structurels

# Les *Patterns structurels*

## Définition

Les **Patterns structurels** décrivent la manière de **composer des classes et des objets** afin de former des structures plus grandes et plus flexibles.

## Principaux Patterns structurels

- **Adapter** — permet de faire collaborer des objets ayant des interfaces incompatibles.
- **Bridge** — permet de séparer une grosse classe en deux hiérarchies.
- **Composite** — permet d'agencer les objets dans des arborescences.
- **Decorator** — permet d'affecter dynamiquement de nouveaux comportements.
- **Facade** — procure une interface offrant un accès simplifié à un sous système complexe.
- **Flyweight** — partage les objets pour économiser la mémoire.
- **Proxy** — permet d'utiliser un substitut pour un objet. (differer la création)

Section 1

# Composite

# Problème à résoudre

## Exemple concret

On souhaite calculer le coût total d'une commande contenant :

- des produits simples,
- des boîtes contenant d'autres produits,
- des boîtes imbriquées les unes dans les autres.

## Difficulté

- Vous pouvez penser à déballer toutes les boites puis prendre tous les objets et faire la somme.
- Mais,
  - La structure est **hiérarchique et variable**.
  - Impossible d'écrire une boucle simple : il faut connaître la nature de chaque objet et son niveau d'imbrication.

# Intention générale

## Intention

Le Pattern **Composite** permet d'agencer des objets dans une **structure arborescente**, afin de traiter de la même façon :

- les objets simples (feuilles),
- et les objets composés (conteneurs).

# Solution du Pattern *Composite*

## Principe

Définir une **interface commune** pour les objets simples et composés :

- **Feuille** : retourne son propre résultat (ex. prix).
- **Conteneur (Composite)** : délègue aux enfants, additionne ou combine leurs résultats.

## Approche récursive

Chaque composite parcourt ses enfants, appelle leur méthode commune, et agrège les résultats.

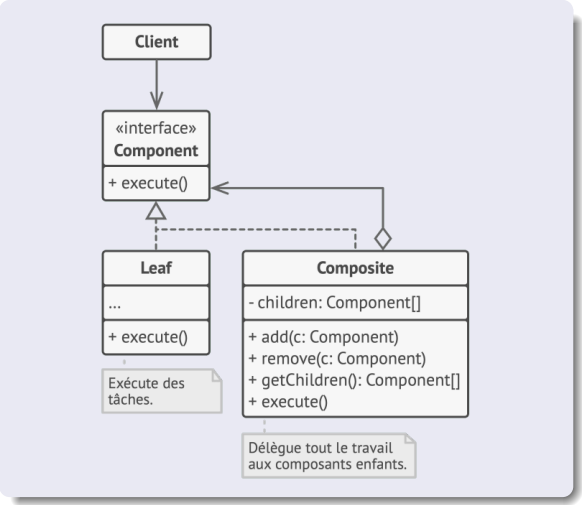
## Avantage clé

Le client ne distingue plus :

- une feuille d'un composite ;
- un élément simple d'un ensemble complexe.

Tout se manipule via une même interface.

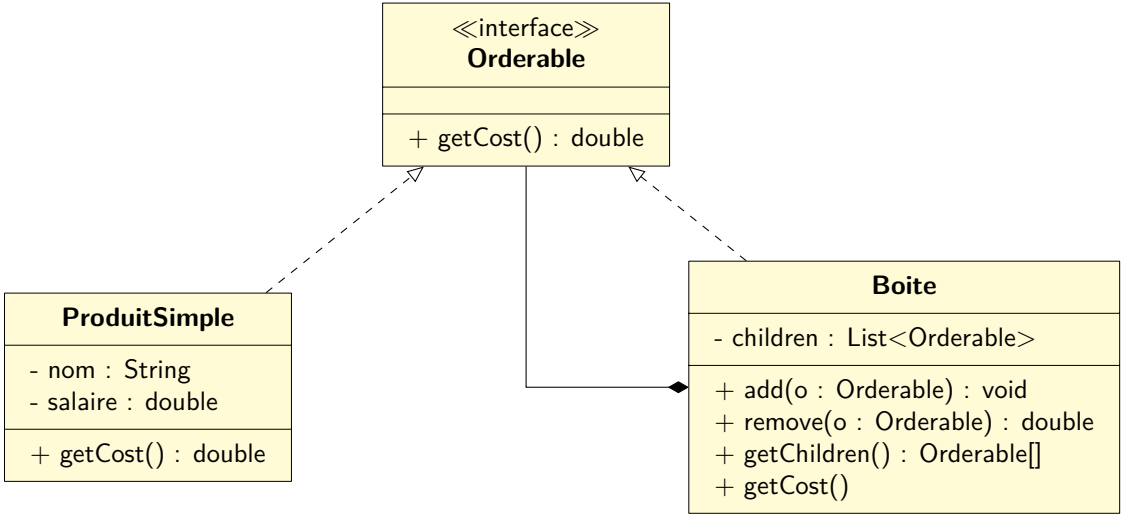
# Structure du *Composite*




## Principe

- L'interface Component décrit les opérations communes aux objets simples et complexes de l'arborescence.
- Une Leaf est un élément de base d'une branche qui n'a pas de sous-élément.
- Le Composite est un élément composé de sous-éléments.
- Chaque composite délègue les appels à ses enfants et agrège leurs résultats.

# Exemple de *Composite*



# Avantages et limites du *Composite*

## Avantages

- Simplifie la gestion des structures hiérarchiques ;
- Exploite le **polymorphisme** ;
- Ouvre la voie au principe **OCP**.

## Inconvénients

- Difficile de définir une interface commune quand les éléments sont très différents ;
- Risque d'interface trop générique, peu lisible.

## Quand l'utiliser ?

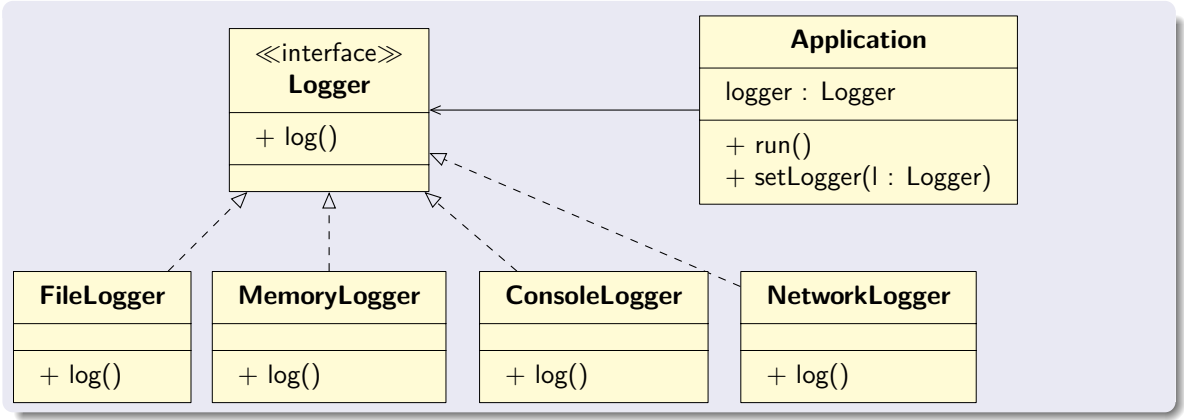
Le *Composite* est idéal pour modéliser des objets imbriqués/arborescence et uniformiser leur manipulation :

- fichiers et dossiers,
- boîtes contenant d'autres boîtes,
- éléments graphiques imbriqués, etc.

Section 2

# Decorateur

# Problème à résoudre




Supposons qu'on veuille parfois chiffrer les logs, ajouter un timestamp, et les enregistrer à la fois sur la console et dans un fichier, sans modifier la classe `Application`.

**Quelle solution proposez-vous ?**

# Intention générale

## Intention

Le Pattern **Décorateur** (aussi appelé *Wrapper* ou *Emballeur*) permet d'**ajouter dynamiquement de nouveaux comportements** à un objet existant, sans modifier son code source.

## Idée clé

On "emballe" un objet dans un autre objet qui implémente la même interface, et qui peut enrichir, filtrer ou transformer les appels avant ou après délégation.

# Solution du Pattern *Décorateur*

## Principe de composition

Plutôt que d'étendre une classe :

- On crée un **décorateur** qui référence un autre objet ;
- Il implémente la même interface et délègue les appels à cet objet ;
- Il peut exécuter du code avant ou après cette délégation.

## Avantage clé

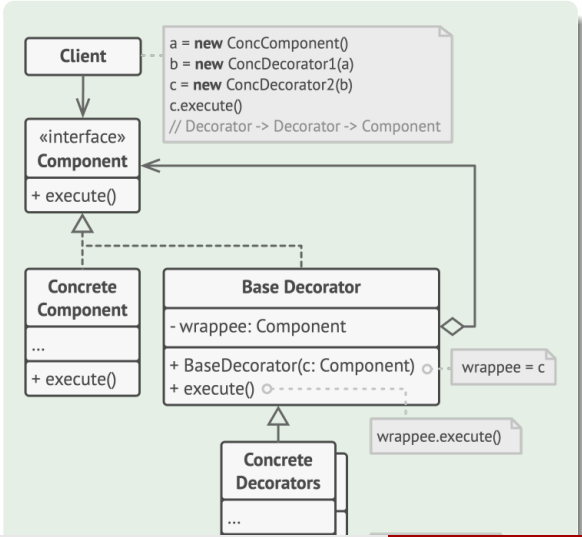
Le comportement d'un objet peut être modifié ou enrichi **dynamiquement**, au moment de l'exécution.

## Différence avec l'héritage

- L'héritage est statique : le comportement est fixé à la compilation ;
- La composition est dynamique : on change de comportement à l'exécution.

→ *Le Décorateur repose sur la composition, pas sur l'héritage.*

# Structure du *Décorateur*

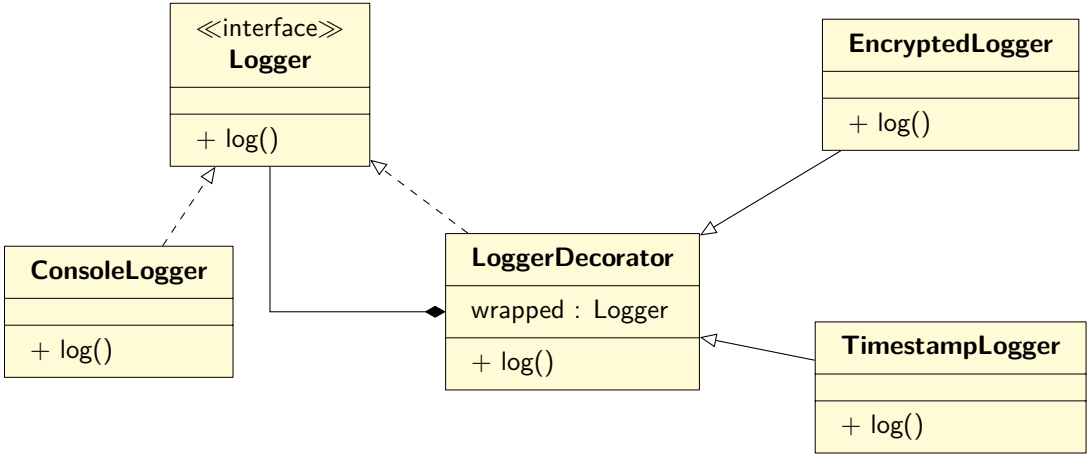



## Principe

Le client manipule l'objet décoré via l'interface commune, sans savoir combien de décorateurs sont empilés.

- **Component** définit l'interface commune à tous ;
- **Concret component** définit le comportement de base ;
- **Base Decorator** référence vers un component (qui peut être déjà décoré ou non) ;
- **Concrete Decorators** ajoutent des comportements spécifiques à execute.

# Exemple de *Décorateur*



# Avantages et limites du *Décorateur*

## Avantages

- Ajout dynamique de comportements sans hériter ;
- Combinaison flexible de fonctionnalités ;
- Respect du principe de **responsabilité unique**.

## Inconvénients

- Configuration des couches parfois complexe ;
- Difficile de retirer un décorateur spécifique ;
- Empilements multiples peu lisibles.

## Quand l'utiliser ?

- Pour étendre le comportement d'un objet au moment de l'exécution ;
- Quand l'héritage devient trop rigide ou interdit ;
- Quand plusieurs extensions doivent être combinées librement.

# Utilisation des Patterns

# Les patterns comme langage commun

## Idée clé

Les **patterns offrent un vocabulaire partagé** entre concepteurs/développeurs :

- « C'est un Décorateur »
- « C'est une Façade »
- « C'est un Composite »

## Avantage

Communication fluide entre développeurs :
*On se comprend sans avoir à lire tout le code.*

# Bonnes pratiques d'utilisation des patterns

## À faire

- Utiliser les patterns pour **clarifier le design** ;
- Refactoriser vers un pattern quand une structure récurrente émerge ;
- Documenter les choix de patterns dans le code ou la conception.

## À éviter

- La "**patternite**" : vouloir caser un pattern partout ;
- Le surdesign inutile ;
- L'emploi d'un pattern juste "pour faire pro".

# En résumé — Utilisation des Patterns

## Essentiel à retenir

- Les patterns ne sont pas des recettes toutes faites, mais des **modèles de réflexion**.
- Ils facilitent la communication et la maintenance du code.
- Ils doivent être utilisés avec discernement et compréhension.

## Mise en pratique

Lors de vos conceptions : identifiez les structures récurrentes, donnez-leur un nom (le pattern), et partagez ce vocabulaire avec votre équipe.