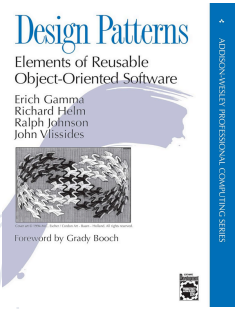
# AOP

## Introduction aux Design Patterns

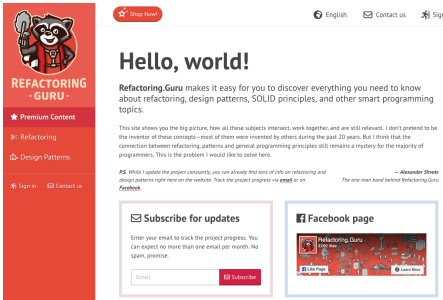

Loïc Rouquette
EPITA - Laboratoire de Recherche de l'EPITA (LRE)
12 novembre 2025

# Table des matières

# Références




**(a)** Design Patterns : Elements of Reusable Object-Oriented Software.




**(b)** Refactoring Guru.

# 1

# Introduction & Concepts Clés

# Pourquoi ?

```
62  public void attack(Monster target, Weapon weapon) {
63      String attackStat = "undefined";
64      switch(weapon.getDamageType().toLowerCase()) {
65          case "slashing", "bludgeoning":
66              attackStat = "strength";
67              break;
68          case "piercing":
69              if (weapon.getName().toLowerCase().contains("bow")) {
70                  attackStat = "dexterity";
71              } else {
72                  attackStat = "strength";
73              }
74              break;
75          case "fire", "ice", "light", "darkness":
76              attackStat = "intelligence";
77              break;
78          default:
79              attackStat = "strength";
80              break;
81      }
82
83      // ...
84  }
```

# Pourquoi ?

Le code est

# Pourquoi ?

Le code est **difficile** à maintenir,

# Pourquoi ?

Le code est **difficile** à maintenir, **impossible** à tester,

# Pourquoi ?

Le code est **difficile** à maintenir, **impossible** à tester, **fragile**.

# Pourquoi ?

Le code est **difficile** à maintenir, **impossible** à tester, **fragile**.

**Solution**

La ”boîte à outils” d’un bon concepteur.

# Qu'est ce qu'un Design Pattern ?

**Définition**

Une **solution éprouvée, réutilisable** et **formalisée** à un **problème de conception courant**.

# Qu'est ce qu'un Design Pattern ?

**Définition**

Une **solution éprouvée, réutilisable** et **formalisée** à un **problème de conception courant**.

Ce n'est pas un framework ou une librairie, c'est un "plan" ou une "recette".

# Qu'est ce qu'un Design Pattern ?

**Définition**

Une **solution éprouvée, réutilisable** et **formalisée** à un **problème de conception courant**.

Ce n'est pas un framework ou une librairie, c'est un "plan" ou une "recette".

L'importance du **"Gang of Four" (GoF)** et du vocabulaire commun.

# Les 3 Grandes Familles

**Créationnels**
Comment *instancier* les objets ?

**Structurels**
Comment *assembler* les objets ?

**Comportementaux**
Comment les objets *communiquent* et *collaborent* ?

# 2

# Les Patterns Créationnels

**Objectif :** Gagner en flexibilité sur qui crée quel objet.

# Focus 1 : Le Singleton

**Problème**

Avoir la garantie d'une **instance unique** pour une classe (*ex* : un gestionnaire de configuration, un pool de connexions BDD).

# Focus 1 : Le Singleton

**Problème**

Avoir la garantie d'une **instance unique** pour une classe (*ex* : un gestionnaire de configuration, un pool de connexions BDD).

**Solution**

Constructeur **privé** + méthode statique `getInstance()`.

# Focus 1 : Le Singleton

**Problème**

Avoir la garantie d'une **instance unique** pour une classe (*ex* : un gestionnaire de configuration, un pool de connexions BDD).

**Solution**

Constructeur **privé** + méthode statique `getInstance()`.

**Discussion**

Les avantages (accès global) vs. les risques (état globel, difficulté à tester). *À utiliser avec parcimonie.*

# Focus 1 : Le Singleton

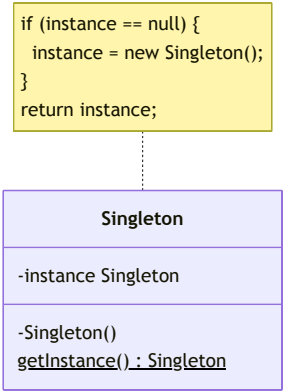



**Figure 1 –** Design Pattern Singleton

# Focus 1 : Le Singleton

```java
public class Singleton {
    private static Singleton instance = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println(s1 == s2); // true
    }
}
```

# Focus 2 : Factory Method (Fabrique)

**Problème**

Une classe ne peut pas anticiper la classe exacter des objets qu'elle doit créer (*ex* : un jeu qui crée des `Ennemi` mais on veut des `Zombie` ou `Squelette` sans que le jeu ne le sache).

# Focus 2 : Factory Method (Fabrique)

**Problème**

Une classe ne peut pas anticiper la classe exacter des objets qu'elle doit créer (*ex* : un jeu qui crée des `Ennemi` mais on veut des `Zombie` ou `Squelette` sans que le jeu ne le sache).

**Solution**

On définit une *interface* pour créer un objet (`IFactoryEnnemi`), mais on laisse les sous-classes (`FactoryZombie`) décider quelle classe concrète instancier.

# Focus 2 : Factory Method (Fabrique)

**Problème**

Une classe ne peut pas anticiper la classe exacter des objets qu'elle doit créer (*ex* : un jeu qui crée des `Ennemi` mais on veut des `Zombie` ou `Squelette` sans que le jeu ne le sache).

**Solution**

On définit une *interface* pour créer un objet (`IFactoryEnnemi`), mais on laisse les sous-classes (`FactoryZombie`) décider quelle classe concrète instancier.

**Bénéfice**

Découple le clinet de l'implémentation concrète. Respecte le principe **Open/Closed**.

# Focus 2 : Factory Method (Fabrique)

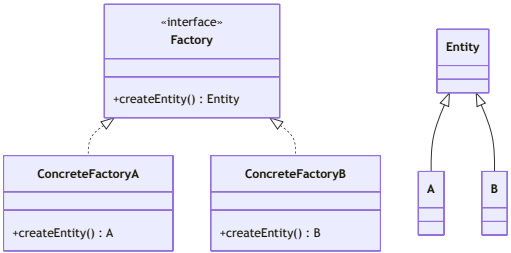



**Figure 2 –** Design Pattern Factory

# Focus 2 : Factory Method (Fabrique)

```java
public class Factory {
    public static class Enemy {}
    public static class Skeleton extends Enemy {}
    public static class Zombie extends Enemy {}

    public static interface IEnemyFactory {
        Enemy createEnemy();
    }

    public static class SkeletonFactory implements IEnemyFactory {
        public Enemy createEnemy() {
            return new Skeleton();
        }
    }

    public static class ZombieFacotry implements IEnemyFactory {
        public Enemy createEnemy() {
            return new Zombie();
        }
    }

    public static void main(String[] args) {
        IEnemyFactory enemyFactory = null;
        if (args[0] == "skeleton world") {
            enemyFactory = new SkeletonFactory();
        } else if (args[0] == "zombie world") {
            enemyFactory = new ZombieFacotry();
        } else {
            throw new RuntimeException("Unknown world");
        }
        Enemy enemy = enemyFactory.createEnemy();
        System.out.println(enemy.getClass().getName());
    }
}
```

# Mention Rapide

**Abstract Factory**

Une ”super-fabrique” pour créer des familles d’objets (*ex* : `GUIFactory` qui crée `Button` et `Checkbox` pour Windows, MacOS ou Linux).

# Mention Rapide

## Abstract Factory

Une ”super-fabrique” pour créer des familles d’objets (*ex* : `GUIFactory` qui crée `Button` et `Checkbox` pour Windows, MacOS ou Linux).

## Builder

Pour construire un objet complexe étape par étape. (*ex* : `new PizzaBuilder().addBase().addSauce().addCheese().build()`).

# 3

# Les Patterns Structurels

**Objectif :** Organiser les classes et objets en structures plus larges et flexibles.

# Focus 1 : L'adapter (Adaptateur)

**Problème**

Faire fonctionner ensemble deux interfaces incompatibles (*ex* : une nouvelle API doit utiliser du code d'une ancienne librairie).

# Focus 1 : L'adapter (Adaptateur)

## Problème

Faire fonctionner ensemble deux interfaces incompatibles (*ex* : une nouvelle API doit utiliser du code d'une ancienne librairie).

## Analogie

L'adaptateur de prise électrique (UK → FR).

# Focus 1 : L'adapter (Adaptateur)

## Problème

Faire fonctionner ensemble deux interfaces incompatibles (*ex* : une nouvelle API doit utiliser du code d'une ancienne librairie).

## Analogie

L'adaptateur de prise électrique (UK → FR).

## Solution

Une classe "wrapper" qui "traduit" les appels de l'interface attendue vers l'interface existante.

# Focus 1 : L'adapter (Adaptateur)

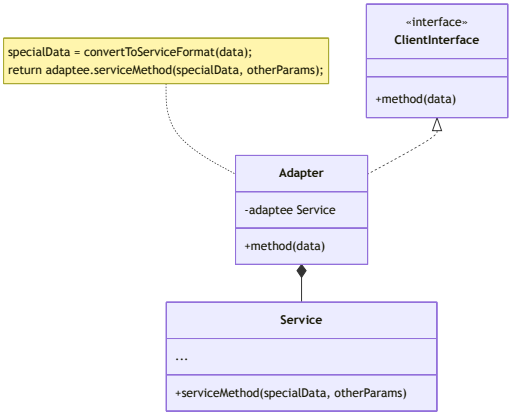



**Figure 3 –** Design Pattern Adapter

# Focus 1 : L'adapter (Adaptateur)

```java
public class Adapter {
    public static class JsonData {}
    public static class XMLData {}

    public interface NewAPI {
        void recordJson(JsonData data);
    }

    public static class OldLibraryClass {
        void recordXML(XMLData data) {
            System.out.println("Recording XML data...");
        }
    }

    public static class AdapterNewToOld implements NewAPI {
        private OldLibraryClass oldLibrary;

        public AdapterNewToOld(OldLibraryClass oldLibrary) {
            this.oldLibrary = oldLibrary;
        }

        @Override
        public void recordJson(JsonData data) {
            // Convert JsonData to XMLData if necessary, or simply call the old method
            System.out.println("Adapting JsonData to XMLData for old library...");
            oldLibrary.recordXML(new XMLData()); // Assuming a simple conversion or placeholder
        }
    }

    public static void main(String[] args) {
        // Using the old library directly
        OldLibraryClass oldLibrary = new OldLibraryClass();
        oldLibrary.recordXML(new XMLData());

        // Using the new API through the adapter
        NewAPI newApiAdapter = new AdapterNewToOld(oldLibrary);
        newApiAdapter.recordJson(new JsonData());
    }
}
```

# Focus 2 : Le Decorator (Décorateur)

**Problème**

Ajouter dynamiquement de nouvelles responsabilités (fonctionnalités) à un objet, sans modifier la classe.

# Focus 2 : Le Decorator (Décorateur)

**Problème**

Ajouter dynamiquement de nouvelles responsabilités (fonctionnalités) à un objet, sans modifier la classe.

**Analogie**

Commander une café (objet de base) et y ajouter des ”décorateurs” (lait, sucre, chantilly).

# Focus 2 : Le Decorator (Décorateur)

**Problème**

Ajouter dynamiquement de nouvelles responsabilités (fonctionnalités) à un objet, sans modifier la classe.

**Analogie**

Commander une café (objet de base) et y ajouter des ”décorateurs” (lait, sucre, chantilly).

**Solution**

Un ”wrapper” qui partage la *même* interface que l’objet décore et lui ajoute un comportement avant ou après l’appel.

# Focus 2 : Le Decorator (Décorateur)

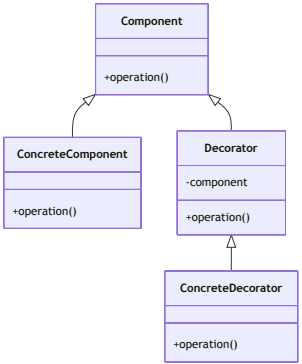



**Figure 4 –** Design Pattern Decorator

# Focus 2 : Le Decorator (Décorateur)

```java
public class Decorator {
    public interface Coffee {
        public void makeCoffee();
    }

    public static class SimpleCoffee implements Coffee {
        public void makeCoffee() {
            System.out.println("Simple coffee");
        }
    }

    public static class MilkDecorator implements Coffee {
        private Coffee coffee;
        public MilkDecorator(Coffee coffee) {
            this.coffee = coffee;
        }
        public void makeCoffee() {
            coffee.makeCoffee();
            System.out.println("Add milk");
        }
    }

    public static class SugarDecorator implements Coffee {
        private Coffee coffee;
        public SugarDecorator(Coffee coffee) {
            this.coffee = coffee;
        }
        public void makeCoffee() {
            coffee.makeCoffee();
            System.out.println("Add sugar");
        }
    }

    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();
        coffee = new MilkDecorator(coffee);
        coffee = new SugarDecorator(coffee);
        coffee.makeCoffee();
    }
}
```

# Focus 3 : La Facade (Façade)

**Problème**

Un sous-système est très complexe, avec de nombreures classes à coordonnée (*ex* : démarre un Home Cinéma = allumer TV + ampli + lecteur + baisser les lumières).

# Focus 3 : La Facade (Façade)

## Problème

Un sous-système est très complexe, avec de nombreures classes à coordonnée (*ex* : démarre un Home Cinéma = allumer TV + ampli + lecteur + baisser les lumières).

## Solution

Une classe unique (`FacadeHomeCinema`) qui fournit une méthode simple (ex : `RegaderFilm()`) et qui orchestre le sous-système en coulisses.

# Focus 3 : La Facade (Façade)

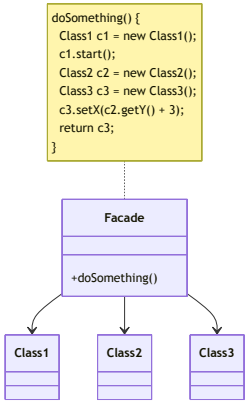



**Figure 5 –** Design Pattern Facade

# Focus 3 : La Facade (Façade)

```java
public class Facade {
    public static class TV {
        public void on() { System.out.println("TV is on"); }
        public void off() { System.out.println("TV is off"); }
        public void setInput(String input) { System.out.println("TV input set to " + input); }
    }

    public static class Amplifier {
        public void on() { System.out.println("Amplifier is on"); }
        public void off() { System.out.println("Amplifier is off"); }
        public void setVolume(int volume) { System.out.println("Amplifier volume set to " + volume); }
        public void setSource(String source) { System.out.println("Amplifier source set to " + source); }
    }

    public static class DVDPlayer {
        public void on() { System.out.println("DVD Player is on"); }
        public void off() { System.out.println("DVD Player is off"); }
        public void play(String movie) { System.out.println("Playing movie: " + movie); }
        public void stop() { System.out.println("DVD Player stopped"); }
    }

    public static class HomeTheaterFacade {
        private TV tv;
        private Amplifier amp;
        private DVDPlayer dvd;

        public HomeTheaterFacade(TV tv, Amplifier amp, DVDPlayer dvd) {
            this.tv = tv;
            this.amp = amp;
            this.dvd = dvd;
        }

        public void watchMovie(String movie) {
            System.out.println("Get ready to watch a movie ... ");
            tv.on();
            tv.setInput("DVD");
            amp.on();
            amp.setSource("DVD");
            amp.setVolume(10);
            dvd.on();
            dvd.play(movie);
        }
        // ...
    }
```

# 4

# Les Patterns Com-portementaux

**Objectif :** Gérer la communication, les responsabilités et les algorithmes.

# Focus 1 : L'Observer (Observateur)

**Problème**

Quand un objet (le ”Sujet”) change d’état, plusieurs autres objets (les ”Observateurs”) doivent être notifiés sans que le Sujet ne les connaisse.

# Focus 1 : L'Observer (Observateur)

**Problème**

Quand un objet (le ”Sujet”) change d’état, plusieurs autres objets (les ”Observateurs”) doivent être notifiés sans que le Sujet ne les connaisse.

**Exemple**

Une cellule dans un tableur (Sujet) et les graphiques (Observateurs) qui l’utilisent.

# Focus 1 : L'Observer (Observateur)

**Problème**

Quand un objet (le "Sujet") change d'état, plusieurs autres objets (les "Observateurs") doivent être notifiés sans que le Sujet ne les connaisse.

**Exemple**

Une cellule dans un tableur (Sujet) et les graphiques (Observateurs) qui l'utilisent.

**Solution**

Le Sujet maintient une liste d'Observateurs et appelle leur méthode `update()` lors d'un changement. Très utilisé en IHM (Interface Homme-Machine).

# Focus 1 : L'Observer (Observateur)

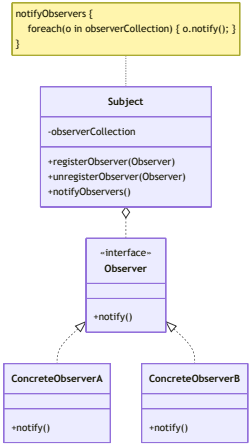



**Figure 6 –** Design Pattern Observer

# Focus 1 : L'Observer (Observateur)

```java
import java.util.List;
import java.util.ArrayList;

public class Observer {
    public interface SpreadsheetCellObserver {
        public void update(String content);
    }
    public static class Logger implements SpreadsheetCellObserver {
        public void update(String content) {
            System.out.println("Logging: " + content);
        }
    }
    public static class SpreadsheetCell {
        private List<SpreadsheetCellObserver> observers;
        private String content = "";
        public SpreadsheetCell() {
            observers = new ArrayList<>();
        }
        public void addObserver(SpreadsheetCellObserver observer) {
            observers.add(observer);
        }
        public void removeObserver(SpreadsheetCellObserver observer) {
            observers.remove(observer);
        }
        public void setContent(String content) {
            this.content = content;
            update();
        }
        public void update() {
            for (SpreadsheetCellObserver observer : observers) {
                observer.update(this.content);
            }
        }
    }
    public static void main(String[] args) {
        SpreadsheetCell cell = new SpreadsheetCell();
        cell.addObserver(new Logger());
        cell.setContent("New content");
    }
}
```

# Focus 2 : Le Strategy (Stratégie)

**Problème**

Avoir plusieurs variantes d'un algorithme (*ex* : modes de paiement `CarteBleue`, `Paypal`, `Virement`) et vouloir les rendre interchangeables.

# Focus 2 : Le Strategy (Stratégie)

**Problème**

Avoir plusieurs variantes d'un algorithme (*ex* : modes de paiement `CarteBleue`, `Paypal`, `Virement`) et vouloir les rendre interchangeables.

**Solution**

Encapsuler chaque algorithme (stratégie) dans sa propre classe (implémentant `IPaiementStrategy`). Le "Contexte" (la commande) possède une stratégie et l'appelle.

# Focus 2 : Le Strategy (Stratégie)

**Problème**

Avoir plusieurs variantes d'un algorithme (*ex* : modes de paiement `CarteBleue`, `Paypal`, `Virement`) et vouloir les rendre interchangeables.

**Solution**

Encapsuler chaque algorithme (stratégie) dans sa propre classe (implémentant `IPaiementStrategy`). Le "Contexte" (la commande) possède une stratégie et l'appelle.

**Bénéfice**

Permet de changer d'algorithme à la volée. Respecte le principe **Open/Closed**.

# Focus 2 : Le Strategy (Stratégie)

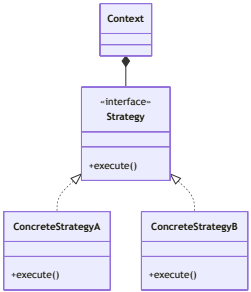



**Figure 7 –** Design Pattern Strategy

# Focus 2 : Le Strategy (Stratégie)

```java
public class Strategy {
    public interface IAttackStrategy {
        void execute(Character attacker, Character target, Weapon weapon);
    }

    public static class WarriorAttackStrategy implements IAttackStrategy {
        @Override
        public void execute(Character attacker, Character target, Weapon weapon) {
            int damage = weapon.getDamageDie() + attacker.getStats().getStrength();
            target.getDamage(damage);
        }
    }

    public static class ThiefAttackStrategy implements IAttackStrategy {
        @Override
        public void execute(Character attacker, Character target, Weapon weapon) {
            int damage = weapon.getDamageDie() + attacker.getStats().getDexterity();
            target.getDamage(damage);
        }
    }

    public static Character {
        // ...
        private IAttackStrategy attackStrategy;

        public void attack(Enemy enemy, Weapon weapon) {
            attackStrategy.execute(this, enemy, weapon);
        }
    }

    public static void main(String[] args) {
        Player p = new Hero();
        p.setStrategy(new WarriorAttackStrategy());
        p.attack(enemy, weapon)
    }
}
```

# Mention Rapide

**Command**

Encapsuler une requête dans un objet (*ex* : pour faire un système d'Undo,Redo, ou des boutons de menu).

# Mention Rapide

## Command

Encapsuler une requête dans un objet (*ex* : pour faire un système d'Undo,Redo, ou des boutons de menu).

## Iterator

Fournir un moyen standard de parcours une collection sans exposer sa structure interne (le `foreach` en est l'aboutissement).

# Synthèse & Conclusion

# Quand & Pourquoi

- Ne pas apprendre les patterns par coeur pour les ”caser” partout (le syndrome du marteau-clou).
- Un pattern doit *résoudre* un problème. Si le problème n’existe pas, le pattern ajoute de la complexité inutilement.

# Les Anti-Patterns

## Définition

Des solutions couramment utilisées mais qui sont en réalité mauvaises.

## Exemples

### God Object

Une classe qui fait tout.

### Spaghetti Code

Flux de contrôle impossible à suivre (`goto`, asynchrone, etc.).

### Copier-Coller

Duplication de code au lieu d'abstraction.

# Synthèse

- Les Design Patterns sont des outils de **communication** et de **conception**.
- Ils rendent le code :
  - Plus flexible ;
  - Plus maintenable ;
  - Plus compréhensible pour les autres développeurs.
- Pour aller plus loin : pratiquer le **Refactoring**.

# Des Questions ?