**ECOLE NATIONALE POLYTECHNIQUE DE YAOUNDE**

*NATIONAL POLYTECHNIC SCHOOL OF YAOUNDE*

**DEPARTEMENT DE GENIE INFORMATIQUE**

*DEPARTMENT OF COMPUTER ENGINEERING*

**UE DE PROGRAMMATION ORIENTE OBJET 2**

*OBJECT-ORIENTED PROGRAMMING UE 2*

**EXPOSE DE POO2 :**

# DESIGN PATTERN :

## CAS PARTICULIER DES DESIGN PATTERNS COMPORTEMENTAIX

RESUME

Le design pattern représente un tournant dans le monde du codage permettant de standardiser les bonnes pratiques. Entrons dans le monde particulier des design pattern et découvrons tout son apport.

**MEMBRES DU GROUPE :**

- FOTEDOU BILL JUNIOR 21P023
- ENYEGUE FREDDY 21P256
- DOEN DAVIS 21P157

**Sous la direction de :** Dr KEGNE

Année 2024/2025

**SOMMAIRE**


**I. Introduction ……1**

**II. Historique et Définitions (10 min)……2**

1. Origine (GoF, 1994 – inspiration architecture)……2
2. Définitions : Pattern, couplage faible, cohésion forte……2
3. Classification (Créationnels, Structurels, Comportementaux)……2

**III. Design Patterns en Général……3**

1. Utilité + exemples courts……3
2. Quand les utiliser ?……7
3. Critiques et bonnes pratiques……7

**IV. Patterns Comportementaux……7**

1. Observer, Strategy, Command……8
2. Cas réels : Notifications (Observer), Paiements (Strategy)……8

**V. Implémentation Java……11**

1. Code commenté……11
2. Démo exécutable……11

**VI. Références et Bibliographie……21**

**VII. Conclusion ……21**

# I. INTRODUCTION

La conception logicielle repose sur des défis récurrents que tout développeur rencontre immanquablement au cours de sa pratique. Qu'il s'agisse de : gérer efficacement les dépendances entre objets, adapter dynamiquement le comportement d'un système, ou garantir la maintenabilité d'une architecture logicielle. Ces problématiques transcendent les langages et les Framework. C'est précisément pour répondre à ces enjeux que les Design Patterns – ou patrons de conception – ont été formalisé et classés en trois catégories : les design patterns créationnels, structurels et comportementaux. Si les patterns créationnels et structurels sont essentiels, les patterns comportementaux occupent une place particulière : ils orchestrent les interactions entre objets et définissent des dynamiques de communication flexibles. Dès lors notre travail revêt un objectif triple à savoir premièrement Clarifier les principes fondamentaux des Design Patterns et leur classification ; puis deuxièmement Analyser en profondeur trois patterns comportementaux clés « Observer », « Strategy », et « Command » à travers leurs cas d'usage typiques ; et enfin Illustrer leur implémentation concrète en Java via une étude de cas modularisée.

## II. DEFINITIONS & HISTORIQUES

### 1. Origine des Design Patterns

- 1977 : L'architecte Christopher Alexander pose les bases du concept dans son livre "A Pattern Language", décrivant des solutions réutilisables pour l'architecture urbaine.
- Années 1990 : Adaptation au logiciel par Kent Beck et Ward Cunningham, puis formalisation par le "Gang of Four" (GoF) dans leur ouvrage fondateur (1994).
- Citation clé : "Un pattern décrit un problème qui survient souvent dans notre environnement, puis décrit la solution à ce problème." (GoF, 1994).

### 2. Définitions Clés

- **Design Pattern :** Modèle de conception éprouvé pour résoudre un problème récurrent dans un contexte donné.
- **Couplage faible** : Principe où les composants interagissent sans dépendre des implémentations internes (ex : via des interfaces).
- **Cohésion forte :** Une classe a une responsabilité unique et bien définie (ex : `Logger` ne gère que les logs).

### 3. Classification des Patterns (GoF)

| **Catégorie** | **Problème Résolu** | **Exemples** |
|---|---|---|
| **Créationnels** | Instanciation d'objets | Singleton, Factory, Abstract, Builder, Prototype |
| **Structurels** | Composition des objets | Adapter, Composite, Bridge, Decorator, Facade, Flyweight, Proxy |
| **Comportementaux** | Interaction entre objets | Observer, Strategy, Template Method, Visitor, Iterator, State, Memento, Command, |

**Tableau 1 : Classification des 23 patterns (GoF)**

**4. Avantages et Risques**

- **Avantages :**

- Réduction de la complexité : Solutions standardisées pour des problèmes connus.
- Maintenabilité : Code organisé et lisible.
- Communication : Vocabulaire commun entre développeurs.

- **Risques :**

- Surutilisation : Appliquer un pattern "parce qu'il faut le faire" → complexité inutile.
- Anti-pattern : Mauvais usage (ex : Singleton global dans une app multi-thread).

# III. DESIGN PATTERN EN GENERAL

## 1) Utilité des Design Patterns : Pourquoi sont-ils incontournables ?

### a. Résoudre des problèmes récurrents

**Exemple concret :**

- **Problème** : Gérer une unique instance d'une classe (ex: connexion DB), contrôler l'accès à une ressource partagée
- **Solution** : **Singleton** → Garantit une seule instance, avec un accès global contrôlé.

```
public class Singleton {
   // Instance unique (statique et privée)
   private static Singleton instance;

   // Constructeur privé
   private Singleton() {}

   // Méthode d'accès contrôlée
   public static Singleton getInstance() {
      if (instance == null) {
         instance = new Singleton(); // Création "paresseuse" (lazy)
      }
      return instance;
   }
   // Méthodes métier
   public void log(String message) {
      System.out.println("[LOG] " + message); }}
// Utilisation
Singleton logger = Singleton.getInstance();
logger.log("Test"); // "[LOG] Test"
```

**Exemple 2 :**

**Probleme** : la création d'un objet necessite une logique complexe, besoin de changer dynamiquement le type de l'objet créé

**Solution** : **Factory** -> Propose une interface/classe abstraite pour la création d'objet, delegue l'instanciation à une sous classe

```
// Interface commune
interface Voiture {
   void demarrer();
}

// Implémentations
class VoitureElectrique implements Voiture {
   public void demarrer() { System.out.println("Démarrage silencieux"); }
}

class VoitureThermique implements Voiture {
   public void demarrer() { System.out.println("Vroum vroum !"); }
}

// Factory
class VoitureFactory {
   public static Voiture creerVoiture(String type) {
      switch (type) {
         case "electrique": return new VoitureElectrique();
         case "thermique": return new VoitureThermique();
         default: throw new IllegalArgumentException("Type inconnu");
      }
   }
}

// Utilisation (découplée !)
Voiture maVoiture = VoitureFactory.creerVoiture("electrique");
maVoiture.demarrer(); // "Démarrage silencieux"
```

### b. Standardiser les bonnes pratiques

**Avantages :**

- Maintenabilité : Code structuré et prévisible.
- Collaboration : Vocabulaire commun (ex : "On utilise un Observer ici").

### c. Limiter les anti-patterns

Un **anti-pattern** est une solution courante mais **inefficace**, **contre-productive** ou même **nuisible** à un problème de conception logicielle. Contrairement aux *design patterns* (bonnes pratiques), les anti-patterns semblent fonctionner à court terme mais entraînent des complications à long terme (maintenance, performance, évolutivité).

**Exemple à éviter :**

- "Spaghetti Code" → Classes aux responsabilités entremêlées.
- Solution : Appliquer Strategy pour découpler les algorithmes.

## 2) Catégories de Patterns : Rappel et Applications

### a) Créationnels

- Objectif : Optimiser la création d'objets.
- Exemple : Factory Method :

```
public interface Document { / méthodes / }
public class PDFDocument implements Document {}
public class WordDocument implements Document {}

public class DocumentFactory {
    public Document createDocument(String type) {
        return switch (type) {
            case "PDF" -> new PDFDocument();
            case "Word" -> new WordDocument();
            default -> throw new IllegalArgumentException();
        };
    }
}
```

### b) Structurels

- Objectif : Simplifier les relations entre objets.

- Exemple : Adapter (convertisseur d'interfaces) :

Cas réel : Adapter une API de paiement legacy à une interface moderne.

### c) Comportementaux (Focus principal)

Ces patrons s'occupent des algorithmes et de la répartition des responsabilités entre les objets. Ces designs particuliers feront l'objet principal de notre travail et seront présentés dans le grand point suivant.

## 3. Critique et Bonnes Pratiques

### a) Quand utiliser un pattern ?

- Le problème correspond à un cas classique (ex : notification → Observer).
- Le code devient difficile à étendre/maintenir.

### b) Pièges à éviter

- Over-engineering :
- Mauvais : Implémenter un Composite pour une structure simple.
- Bon : Commencer par une solution naïve, puis refactoriser si besoin.

# IV. LES PATTERNS COMPORTEMENTAUX : THÉORIE & PRATIQUE

## 1. Introduction aux Patterns Comportementaux

- **Définition :**

Solutions pour gérer les interactions entre objets et répartir les responsabilités de manière flexible.

- **Pourquoi sont-ils cruciaux ?**

Ils permettent d'apporter des solutions aux problèmes

- Communication (ex : notifications) → Observer
- Comportements dynamiques (ex : algorithmes interchangeables) → Strategy
- Traitements encapsulés (ex : undo/redo) → Command

## 2. Pattern Observer : Notifications Événementielles

### a) Problème

- Comment notifier automatiquement des objets lorsqu'un état change ?
- Exemple réel : Système d'abonnement (YouTube, newsletters).

### b) Solution

Acteurs :

- Subject (observable) : Gère la liste des observateurs + notifie.
- Observer (interface) : Définit la méthode update().

d) **Implémentation Java**

```
// Interface Observer
public interface Observer {
   void update(String message);
}

// Subject (Observable)
public class NewsAgency {
   private List<Observer> observers = new ArrayList<>();

   public void addObserver(Observer o) {
      observers.add(o);
   }

   public void notifyObservers(String news) {
      for (Observer o : observers) {
         o.update(news); // Notification
      }
   }
}
```

### c) Avantages/Inconvénients

- Avantages :
  - Découplage sujet
  - Extensible (ajout facile)
- Inconvénients :
  - Notifications non contrôlées
  - Observateur  Peut causer des fuites mémoire

## b) Pattern Strategy : Algorithmes Interchangeables

### a. Problème

- Comment changer dynamiquement un algorithme sans modifier le code client ?
- Exemple : Tri (QuickSort, BubbleSort), validation de données.

### b. Solution

Structure :

- `Strategy` (interface) : Définit la méthode `execute()`.
- `ConcreteStrategy` : Implémentations spécifiques.
- `Context` : Utilise une stratégie.

### c. Exemple Java

```
// Interface Strategy
public interface SortingStrategy {
   void sort(int[] data);}

// Implémentations
public class QuickSort implements SortingStrategy {
   @Override
   public void sort(int[] data) { / Implémentation QuickSort / }}

public class BubbleSort implements SortingStrategy {
   @Override
   public void sort(int[] data) { / Implémentation BubbleSort / }}

// Context
public class Sorter {
   private SortingStrategy strategy;
   public void setStrategy(SortingStrategy strategy) {
      this.strategy = strategy;
   }
   public void sortData(int[] data) {
      strategy.sort(data); }}
```

### c) Pattern Command : Encapsuler les Requêtes

#### a. Problème

- Comment paramétrer des actions, les annuler ou les journaliser ?
- Exemple : Boutons "Undo", files d'attente de tâches.

#### b. Solution

- Acteurs :
  - Command(interface) : Méthodes `execute()`, `undo()`.
  - Invoker : Exécute la commande.

## V. IMPLÉMENTATION DU MINI-PROJET JAVA (Observer + Strategy + Command) + DIAGRAMMES UML

### 1. Contexte du Mini-Projet

Le but de notre mini projet est de coder un <<système de commande en ligne>> le but n'étant pas de coder une application complète mais de construire une API dont les fonctionnalités principales nous permettront de mettre facilement en avant l'utilisation des design patterns comportementaux. A cet effet nous coderons un système dont l'objectif est de permettre à des utilisateurs de commander et d'être notifiés du statut de leur paiement.

C'est ainsi qu'on implémentera les designs :

- Observer : Notifications utilisateur après paiement.
- Strategy : Choix du mode de paiement (Carte, PayPal).
- Command : Historique des transactions (undo, redo).

### 2. Implémentation des patterns

- **Pattern Observer** utile pour permettre de faciliter la notification au près des users

**Observer/Observer.java**

```
package payment.patterns.observer;
public interface Observer {
   void update(String message);
}
```

**observer/Observable.java:**

```
package payment.patterns.observer;

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

public class Observable {
   private List<Observer> observers = new ArrayList<>();

   public void addObserver(Observer observer) {
      observers.add(observer);
   }

   public void removeObserver(Observer observer) {
      observers.remove(observer);
   }

   public void notifyObservers(String message) {
      for (Observer observer : observers) {
         observer.update(message);
      }
   }
}
```

- **Pattern Strategy**

  **Stratégie** est un patron de conception comportemental qui permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables

  Il permettra aisément de créer les systèmes de paiement qui juste se subdivisent en deux catégorie ici. Ainsi on peut créer une méthode pour le paiement qui prendra les paiements (de façon général) comme objet ce qui aura pour effet de permettre l'interchangeabilité des objets (paiement par carte ou par PayPal).

Strategy/CreditCardPayment.java

```
package payment.patterns.strategy;

public class CreditCardPayment implements PaymentStrategy {
   private String cardNumber;
   private String expiryDate;
   private String cvv;

   public CreditCardPayment(String cardNumber, String expiryDate, String cvv) {
      this.cardNumber = cardNumber;
      this.expiryDate = expiryDate;
      this.cvv = cvv;
   }

   @Override
   public boolean pay(double amount) {
      System.out.printf("Paiement de %.2f€ effectué par carte %s%n",
            amount, cardNumber.substring(cardNumber.length() - 4));
      return true;
   }
}
```

strategy/PayPalPayment.java:

```
package payment.patterns.strategy;

public class PayPalPayment implements PaymentStrategy {
   private String email;
   private String password;

   public PayPalPayment(String email, String password) {
      this.email = email;
      this.password = password;
   }

   @Override
   public boolean pay(double amount) {
      System.out.printf("Paiement de %.2f€ effectué via PayPal (%s)%n",
            amount, email);
      return true;
   }
}
```

- **Pattern Command**

**Commande** est un patron de conception comportemental qui prend une action à effectuer et la transforme en un objet autonome qui contient tous les détails de cette action. Cette transformation permet de paramétrer des méthodes avec différentes actions, planifier leur exécution, les mettre dans une file d'attente ou d'annuler des opérations effectuées

Dans notre cas il s'agira d'encapsuler les actions d'execution(execute) et d'annulation (undo) dans une classe **paymentCommand**.

Command/Command.java

```
package payment.patterns.command;

public interface Command {
   boolean execute();
   boolean undo();
}
```

Command/CommandPayment.java

```
package payment.patterns.command;

import payment.patterns.strategy.PaymentStrategy;
import payment.model.TransactionManager;

public class PaymentCommand implements Command {
   private PaymentStrategy paymentStrategy;
   private double amount;
   private TransactionManager transactionManager;
   private boolean executed;

   public PaymentCommand(PaymentStrategy paymentStrategy, double amount,
                  TransactionManager transactionManager) {
      this.paymentStrategy = paymentStrategy;
      this.amount = amount;
      this.transactionManager = transactionManager;
      this.executed = false;
   }
```

Command/Command.java

```
package payment.patterns.command;


    @Override
    public boolean execute() {
        boolean success = paymentStrategy.pay(amount);
        if (success) {
            transactionManager.addTransaction(this);
            executed = true;
            return true;
        }
        return false;
    }

    @Override
    public boolean undo() {
        if (executed) {
            System.out.printf("Annulation du paiement de %.2f€%n", amount);
            executed = false;
            return true;
        }
        return false;
    }
}
```

model/TransactionManager.java:

```
package payment.model;

import payment.patterns.command.Command;
import java.util.Stack;

public class TransactionManager {
   private Stack<Command> transactions = new Stack<>();
   private Stack<Command> undoneTransactions = new Stack<>();

   public void addTransaction(Command transaction) {
      transactions.push(transaction);
      undoneTransactions.clear();
   }

   public boolean undoLastTransaction() {
      if (transactions.isEmpty()) {
         return false;
      }

      Command transaction = transactions.pop();
      if (transaction.undo()) {
         undoneTransactions.push(transaction);
         return true;
      }
      return false;
   }

   public boolean redoLastTransaction() {
      if (undoneTransactions.isEmpty()) {
         return false;
      }

      Command transaction = undoneTransactions.pop();
      if (transaction.execute()) {
         transactions.push(transaction);
         return true;
      }
      return false;
   }
}
```

model/User.java:

```
package payment.model;

import payment.patterns.observer.Observer;

public class User implements Observer {
   private String name;
   private String email;

   public User(String name, String email) {
      this.name = name;
      this.email = email;
   }

   @Override
   public void update(String message) {
      System.out.printf("Notification pour %s (%s): %s%n", name, email, message);
   }
}
```

model/PaymentProcessor.java:

```
package payment.model;

import payment.patterns.observer.Observable;
import payment.patterns.strategy.PaymentStrategy;

public class PaymentProcessor extends Observable {
   private PaymentStrategy paymentStrategy;

   public void setPaymentStrategy(PaymentStrategy strategy) {
      this.paymentStrategy = strategy;
   }

   public boolean processPayment(double amount) {
      if (paymentStrategy == null) {
         throw new IllegalStateException("Aucune méthode de paiement sélectionnée");
      }

      boolean success = paymentStrategy.pay(amount);
      if (success) {
         notifyObservers(String.format("Paiement réussi de %.2f€", amount));
      } else {
         notifyObservers(String.format("Échec du paiement de %.2f€", amount));
      }
      return success;
   }
}
```

**Explications**

1. **Pattern Observer**:
    - Les utilisateurs implémentent l'interface Observer et reçoivent des notifications
    - Le PaymentProcessor hérite de Observable et notifie les observateurs
2. **Pattern Strategy**:
    - Les différentes méthodes de paiement implémentent l'interface PaymentStrategy
    - On peut facilement changer la stratégie avec setPaymentStrategy()
3. **Pattern Command**:
    - Chaque transaction est encapsulée dans un objet PaymentCommand
    - Le TransactionManager gère l'historique avec des piles (undo/redo)

## IV. CONCLUSION

Ce projet nous a permis d'approfondir notre compréhension des design patterns comportementaux et leur utilité dans le développement logiciel. Nous avons pu constater concrètement comment les patterns Observer, Strategy et Command résolvent des problèmes courants de manière élégante.

L'Observer nous a montré comment gérer efficacement les notifications entre composants sans créer de couplage fort.

La Strategy nous a aidés à rendre nos algorithmes interchangeables, ce qui s'est avéré très pratique pour implémenter différents modes de paiement.

Le Command nous a permis de mieux structurer nos traitements en les encapsulant dans des objets, facilitant ainsi les fonctionnalités d'annulation et de rejeu.

Nous avons aussi réalisé qu'il faut éviter d'appliquer ces patterns systématiquement. Dans certains cas, une solution plus simple peut suffire. Cette expérience nous a confirmé l'importance de bien analyser les besoins avant de choisir une solution technique.

En conclusion, ce travail nous a fait progresser dans notre maîtrise des bonnes pratiques de conception. Nous sommes maintenant mieux armés pour concevoir des architectures logicielles plus propres et plus flexibles.