Module 4

Design Patterns

Génie Logiciel et Qualité — M1 MIAGE | 1h30 CM | 1h TD | 2h TP

Créationnels | Structurels | Comportementaux

Objectifs du module

1. **Reconnaître** les patterns classiques dans du code existant
2. **Appliquer** les patterns créationnels, structurels et comportementaux
3. **Justifier** le choix d'un pattern selon le contexte
4. **Refactorer** du code vers des patterns pour améliorer la conception

📘 **Référence :** "Design Patterns: Elements of Reusable Object-Oriented Software" — Gang of Four (1994)

1

Introduction

Qu'est-ce qu'un Design Pattern ?

Qu'est-ce qu'un Design Pattern ?

> *"Un design pattern décrit un problème récurrent et le cœur de sa solution, de telle manière que vous puissiez utiliser cette solution un million de fois sans jamais l'appliquer deux fois de la même façon."*
>
> — Christopher Alexander

Structure d'un pattern

1. Nom	Vocabulaire partagé
2. Problème	Contexte d'utilisation
3. Solution	Structure et interactions
4. Conséquences	Avantages / Inconvénients

Ce qu'un pattern N'EST PAS

- ❌ Du code à copier-coller
- ❌ Une bibliothèque toute faite
- ❌ Une règle absolue
- ❌ Une solution miracle

Classification des 23 patterns GoF

Créationnels (5)

Comment créer ?

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Structurels (7)

Comment composer ?

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Comportementaux (11)

Comment interagir ?

- Chain of Resp.
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor
- Interpreter

Quand utiliser un pattern ?

🚨 Signaux d'alerte

- ☐ Même code écrit 3× fois
- ☐ Switch/if sur des types d'objets
- ☐ Classe avec trop de responsabilités
- ☐ Code difficile à tester
- ☐ Feature = modifier beaucoup de classes

⚠️ Principe YAGNI

"You Aren't Gonna Need It"
N'introduisez pas un pattern "au cas où" !
Commencez simple, refactorez si nécessaire.

💊 Anti-pattern : Patternite

- Factory pour 1 seul type d'objet
- Singleton pour tout
- Observer pour 2 objets

`Code simple → Duplication détectée → Refactoring → Pattern émerge`

Évolution naturelle vers les patterns

Patterns et principes SOLID

S	**Single Responsibility**	Facade, Mediator, Command
O	**Open/Closed**	Strategy, Decorator, Template Method
L	**Liskov Substitution**	Factory Method, polymorphisme
I	**Interface Segregation**	Adapter, Facade
D	**Dependency Inversion**	Abstract Factory, Strategy, Observer

Exemple OCP + Strategy : Ajouter un nouveau type = créer une classe, pas modifier l'existant

2

Patterns Créationnels

Comment créer des objets ?

Singleton

Créationnel

Problème : Garantir qu'une classe n'a qu'UNE SEULE instance

Cas d'usage légitimes

- Configuration
- Pool de connexions
- Logger
- Cache partagé

```
// ✅ Singleton thread-safe avec enum
public enum AppConfig {
    INSTANCE;

    private String databaseUrl;

    public String getDatabaseUrl() {
        return databaseUrl;
    }
}

// Utilisation
AppConfig.INSTANCE.getDatabaseUrl();
```

⚠️ Inconvénients

- État global → couplage fort
- Difficile à tester (mocker)
- Thread-safety complexe

✅ Alternative : Injection de dépendances

Le container DI (Spring) gère
le scope singleton pour vous
→ Testable, découplé

Factory Method

Créationnel

Problème : Déléguer la création d'objets à des sous-classes

```
// Interface produit
public interface Notification {
    void send(String message);
}

// Factory Method dans la classe abstraite
public abstract class NotificationFactory {
    // Factory Method - implémenté par les sous-classes
    protected abstract Notification createNotification();

    public void notify(String message) {
        Notification notif = createNotification();
        notif.send(message);
    }
}

// Concrete Factories
public class EmailFactory extends NotificationFactory {
    protected Notification createNotification() {
        return new EmailNotification();
    }
}
```

Quand l'utiliser ?

- Frameworks / plugins
- Type exact inconnu à compile
- Déléguer la création aux sous-classes

Simple Factory (idiom)

Méthode statique avec switch
→ pas un vrai pattern GoF
mais très utilisé

Builder

Créationnel

Problème : Construire des objets complexes étape par étape

❌ **Constructeur télescopique**

```
new Pizza("L", true, true, false,
          true, false, "Extra crispy");
```

✅ **API fluide avec Builder**

```
Pizza.builder("Large")
    .cheese(true).pepperoni(true)
    .specialInstructions("Extra crispy")
    .build();
```

```
// Builder avec Lombok (génération automatique)
@Builder
@Getter
public class Pizza {
    private final String size;
    @Builder.Default
    private final boolean cheese = true;
    private final boolean pepperoni;
    private final String specialInstructions;
}

// Utilisation identique
Pizza pizza = Pizza.builder()
    .size("Large")
    .pepperoni(true)
    .build();
```

3

Patterns Structurels

Comment composer les objets ?

Adapter

Structurel

Problème : Faire collaborer des classes avec des interfaces incompatibles

🔌 Analogie : Prise européenne → Adaptateur → Appareil US

```
// Interface moderne attendue
public interface PaymentProcessor {
    PaymentResult process(PaymentRequest request);
}

// Système legacy incompatible
public class LegacyPaymentGateway {
    public int makePayment(String cardNumber, int amountCents) { ... }
}

// Adapter : pont entre les deux interfaces
public class LegacyPaymentAdapter implements PaymentProcessor {
    private final LegacyPaymentGateway legacy;

    @Override
    public PaymentResult process(PaymentRequest request) {
        int cents = (int)(request.amount() * 100);
        int code = legacy.makePayment(request.cardNumber(), cents);
        return code == 0 ? PaymentResult.success() : PaymentResult.failure();
    }
}
```

✅ Quand l'utiliser ?

- Intégrer une lib tierce
- Moderniser un legacy
- Réutiliser une classe existante

Decorator

Structurel

Problème : Ajouter des responsabilités dynamiquement sans modifier la classe

❌ **Héritage : explosion combinatoire**

```
CoffeeWithMilk
CoffeeWithMilkAndSugar
CoffeeWithMilkSugarCream
... 🤯
```

✅ **Decorator : composition flexible**

```
new WhippedCream(
  new Sugar(
    new Milk(new Coffee())))
```

```
public interface Notifier { void send(String msg); }

public class SMSDecorator implements Notifier {
    private final Notifier wrappee;

    public SMSDecorator(Notifier wrappee) { this.wrappee = wrappee; }

    public void send(String msg) {
        wrappee.send(msg);  // Déléguer d'abord
        sendSMS(msg);       // Puis ajouter le comportement
    }
}

// Composition à runtime
Notifier notifier = new LoggingDecorator(
    new SMSDecorator(new EmailNotifier()));
```

Facade

Structurel

Problème : Fournir une interface simplifiée à un sous-système complexe

Sous-systèmes

Inventory | Payment | Shipping | Notification | Loyalty


```
public class OrderFacade {
    private final InventoryService inventory;
    private final PaymentService payment;
    private final ShippingService shipping;
    // ... autres services

    // UNE méthode simple pour le client
    public OrderResult placeOrder(OrderRequest request) {
        inventory.checkStock(request.productId());
        payment.processPayment(request.amount());
        shipping.createShipment(request.address());
        // ... orchestration complète
        return OrderResult.success(orderId);
```

4

Patterns Comportementaux

Comment les objets interagissent ?

Strategy

Comportemental

🔁 Rappel Module 3 : refactoring de calculatePenalty() avec PenaltyPolicy

Problème : Définir une famille d'algorithmes interchangeables

```
public interface CompressionStrategy {
    byte[] compress(byte[] data);
}

public class ZipCompression implements CompressionStrategy { ... }
public class GzipCompression implements CompressionStrategy { ... }

// Context
public class FileCompressor {
    private CompressionStrategy strategy;

    public void setStrategy(CompressionStrategy s) { this.strategy = s; }

    public void compress(String path) {
        byte[] result = strategy.compress(readFile(path));
    }
}

// Changement à runtime
compressor.setStrategy(new GzipCompression());
```

💡 Java 8+ : Lambdas

```
// Interface fonctionnelle
data -> compress(data)

// Méthode reference
Utils::gzipCompress
```

Observer

Comportemental

Problème : Notifier automatiquement plusieurs objets quand un état change

```
public interface StockObserver {
    void onPriceChange(String symbol, double price);
}

public class StockMarket {
    private final List<StockObserver> observers = new ArrayList<>();
    private final Map<String, Double> prices = new HashMap<>();

    public void subscribe(StockObserver o) { observers.add(o); }
    public void unsubscribe(StockObserver o) { observers.remove(o); }

    public void updatePrice(String symbol, double price) {
        prices.put(symbol, price);
        // Notifier tous les observateurs
        observers.forEach(o -> o.onPriceChange(symbol, price));
    }
}

// Concrete Observer
public class TradingBot implements StockObserver {
    public void onPriceChange(String symbol, double price) {
        if (price < threshold) buy(symbol);
    }
}
```

Push vs Pull

Push : données envoyées avec la notification

Pull : observer récupère les données lui-même

🚀 Alternatives modernes

- Spring @EventListener
- RxJava / Reactor
- PropertyChangeListener

Command

Comportemental

Problème : Encapsuler une requête comme un objet (Undo/Redo, queues)

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

public class AddTextCommand implements Command {
    private final Document doc;
    private final String text;
    private final int position;

    public void execute() { doc.insertAt(position, text); }
    public void undo() { doc.deleteAt(position, text.length()); }
}

// Invoker avec historique
public class CommandHistory {
    private final Deque<Command> history = new ArrayDeque<>();

    public void execute(Command cmd) {
        cmd.execute();
        history.push(cmd);
    }

    public void undo() {
        if (!history.isEmpty()) history.pop().undo();
    }
}
```

Cas d'usage

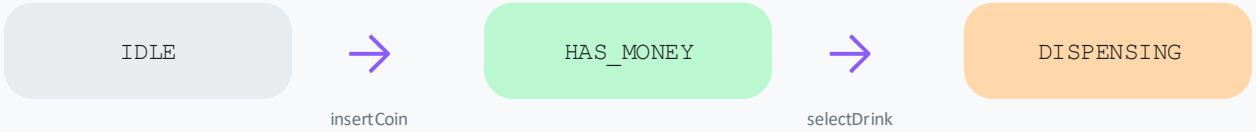
- Undo/Redo
- File de tâches
- Transactions
- Macro-commandes
- Logging des actions

State

Comportemental

Problème : Changer le comportement d'un objet selon son état interne

Machine à café : états et transitions




```
public interface CoffeeState {
    void insertCoin(CoffeeMachine m, double amount);
    void selectDrink(CoffeeMachine m, String drink);
}

public class IdleState implements CoffeeState {
    public void insertCoin(CoffeeMachine m, double amount) {
        m.addBalance(amount);
        m.setState(new HasMoneyState());
    }
    public void selectDrink(CoffeeMachine m, String drink) {
        System.out.println("Insert coin first!");
    }
}

// Context délègue à l'état courant
public class CoffeeMachine {
    private CoffeeState state = new IdleState();
```

5

Patterns en Pratique

Combinaisons et frameworks modernes

Patterns dans les frameworks modernes

Spring Framework

`@Bean, @Configuration`	→ **Factory**
`@Scope("singleton")`	→ **Singleton**
`ApplicationContext`	→ **Facade**
`@Transactional (AOP)`	→ **Proxy**
`RestTemplate`	→ **Template Method**
`@EventListener`	→ **Observer**
`HandlerInterceptor`	→ **Chain of Resp.**

Java Streams

```
items.stream()
  .filter(...)     // Strategy (Predicate)
  .map(...)        // Strategy (Function)
  .sorted(...)     // Strategy (Comparator)
  .collect(...)    // Strategy (Collector)
```

Reactive Streams

```
Flux.just("A", "B")
  .map(...)       // Decorator
  .filter(...)    // Decorator
  .subscribe(...) // Observer
```

Checklist du développeur

✅ Avant d'implémenter un pattern

- ☐ Le problème est-il vraiment récurrent ?
- ☐ La solution simple ne suffit-elle pas ?
- ☐ L'équipe comprend-elle le pattern ?
- ☐ Le pattern est-il adapté au contexte ?

Signaux → Patterns

new partout dans le code	→ **Factory**
Constructeur 7+ params	→ **Builder**
Switch sur type d'objet	→ **Strategy / State**
Beaucoup de if/else état	→ **State**
Notifications entre objets	→ **Observer**
Interface incompatible	→ **Adapter**
Features optionnelles	→ **Decorator**
Besoin d'undo/redo	→ **Command**
Sous-système complexe	→ **Facade**

Récapitulatif

Créationnels	Factory, Builder, Singleton	*Création d'objets*
Structurels	Adapter, Decorator, Facade, Proxy	*Composition*
Comportementaux	Strategy, Observer, Command, State	*Interactions*

🏆 **Règle d'or**

"Un pattern est une solution à un problème dans un contexte. Sans problème, pas de pattern."

🧱 **Prochaine étape : Module 5 — Architecture Hexagonale & DDD**

Patterns tactiques DDD • Ports & Adapters • Clean Architecture