

Handout Buổi 14

Chủ đề: Giới thiệu một số mẫu design pattern cơ bản trong OOP

Mục tiêu của buổi học:

- Hiểu và phân biệt các mẫu design pattern cơ bản
- Vận dụng được một số mẫu design pattern

1. Giới thiệu chung về Design Patterns

1.1. Design Pattern là gì?





Design Pattern là những giải pháp đã được kiểm chứng cho các vấn đề phổ biến trong thiết kế phần mềm hướng đối tượng.


- Là **mẫu thiết kế (không phải mã nguồn)**, mô tả cách tổ chức **lớp, đối tượng** và **mối quan hệ** giữa chúng.
- Giúp **tái sử dụng kiến thức thiết kế**, thay vì giải quyết vấn đề từ đầu.

Ví dụ đơn giản:

Khi bạn cần đảm bảo chỉ có một đối tượng cấu hình hệ thống trong chương trình – bạn đang gặp vấn đề Singleton → Áp dụng **Singleton Pattern**.

1.2. Tại sao Design Pattern quan trọng?


Lý do	Ý nghĩa
 Tái sử dụng giải pháp đã kiểm chứng	Giảm thời gian thiết kế & phát triển
 Cải thiện khả năng bảo trì	Giải pháp có cấu trúc, dễ đọc, dễ mở rộng
 Tăng tính linh hoạt & mở rộng	Các mẫu thường khuyến khích sử dụng interface, abstract class
 Hỗ trợ giao tiếp nhóm phát triển	Các mẫu đều có tên gọi thống nhất , dễ trao đổi

 Khi mọi người nói "Dùng Strategy ở đây", ta biết ngay ý nghĩa mà không cần giải thích dài dòng.

1.3. Phân loại Design Patterns

Design Patterns thường được chia thành **3 nhóm chính** theo mục tiêu sử dụng:

Nhóm	Mục tiêu	Một số pattern tiêu biểu
Creational	Quản lý việc tạo đối tượng	Singleton, Factory Method, Builder
Structural	Tổ chức cấu trúc lớp/đối tượng	Adapter, Composite, Decorator
Behavioral	Quản lý giao tiếp & hành vi giữa đối tượng	Strategy, Template Method, State, Chain of Responsibility

 Mỗi pattern giải quyết **một loại vấn đề cụ thể** trong thiết kế phần mềm hướng đối tượng.

2. Các Design Pattern cơ bản được giới thiệu

2.1. Singleton Pattern

? Vấn đề đặt ra

Trong nhiều tình huống, ta cần đảm bảo rằng **chỉ có một thể hiện duy nhất** của một lớp tồn tại trong toàn bộ chương trình.

✦ Tình huống thực tế:

- Một **lớp cấu hình hệ thống** (AppConfig)
- Một **bộ ghi log hệ thống** (Logger)
- Một **kết nối cơ sở dữ liệu dùng chung** (DatabaseConnection)

Nếu tạo ra nhiều đối tượng, có thể dẫn đến **xung đột tài nguyên, tốn bộ nhớ**, hoặc hành vi **khó kiểm soát**.

🔧 Cách cài đặt cơ bản trong Java

```
public class Singleton {  
    private static Singleton instance; // Biến tĩnh duy nhất  
  
    // Constructor private để ngăn tạo mới từ bên ngoài  
    private Singleton() {  
        System.out.println("Singleton created!");  
    }  
  
    // Phương thức truy cập thể hiện duy nhất  
    public static Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton(); // Tạo nếu chưa có  
        }  
        return instance;  
    }  
}
```

🔧 Sử dụng:

```
Singleton s1 = Singleton.getInstance();  
Singleton s2 = Singleton.getInstance();  
  
System.out.println(s1 == s2); // true – cùng một đối tượng
```

✅ Ưu điểm

- ✅ Kiểm soát thể hiện: đảm bảo **chỉ có một đối tượng duy nhất**.
- ✅ Tiết kiệm tài nguyên: tránh tạo nhiều đối tượng cùng chức năng.
- ✅ Dễ truy cập: qua phương thức tĩnh (getInstance()).

⚠️ Nhược điểm

- ❌ Khó kiểm thử đơn vị (unit test): vì **khó thay thế Singleton bằng mock**.
- ❌ Có thể gây **xung đột trong môi trường đa luồng (multithreading)** nếu không xử lý đúng.

- ❌ Có thể bị lạm dụng → vi phạm nguyên tắc SRP (Single Responsibility Principle).

💡 **Ghi chú nâng cao:** Trong môi trường đa luồng, cần sử dụng các biến volatile hoặc kỹ thuật đồng bộ (synchronized) hoặc dùng **Initialization-on-demand holder** (sẽ học tiếp ở mục 2.2).

2.2. Initialization-on-Demand Holder (IoDH)

Một cách **cải tiến Singleton** giúp đảm bảo tính **lazy loading** và **an toàn trong môi trường đa luồng** mà không cần dùng synchronized.

? Vấn đề cần cải tiến

Phiên bản Singleton truyền thống có vấn đề trong môi trường **đa luồng (multithreading)**:

```
public static Singleton getInstance() {  
    if (instance == null) {  
        instance = new Singleton(); // Có thể tạo nhiều thể hiện nếu nhiều thread cùng  
        vào đây!  
    }  
    return instance;  
}
```

👉 Dùng synchronized để giải quyết, nhưng sẽ làm **giảm hiệu năng** do chi phí đồng bộ hóa.

💡 Giải pháp: Initialization-on-Demand Holder (IoDH)

Sử dụng **inner static class**, tận dụng đặc tính:

- Lớp tĩnh chỉ được tải vào bộ nhớ **khi được gọi lần đầu tiên**.
- Trình tải lớp (class loader) trong Java **luôn an toàn với đa luồng**.

🔧 Cài đặt mẫu IoDH

```
public class Singleton {  
    // Constructor vẫn private  
    private Singleton() {  
        System.out.println("IoDH Singleton created!");  
    }  
  
    // Inner static class chứa thể hiện Singleton duy nhất  
    private static class Holder {  
        private static final Singleton INSTANCE = new Singleton();  
    }  
  
    // Truy cập Singleton qua Holder  
    public static Singleton getInstance() {  
        return Holder.INSTANCE;  
    }  
}
```





```
}
```

Sử dụng:








```
Singleton s1 = Singleton.getInstance();  
Singleton s2 = Singleton.getInstance();
```

```
System.out.println(s1 == s2); // true
```

Ưu điểm của IoDH

Ưu điểm	Giải thích
 Thread-safe	Nhờ cơ chế class loading của JVM
 Lazy loading	Chỉ khởi tạo Singleton khi gọi getInstance()
 Hiệu năng cao	Không dùng synchronized, không gây chậm
 Đơn giản, dễ đọc	Không phức tạp như Double-Checked Locking

So sánh với Singleton truyền thống

Tiêu chí	Singleton truyền thống	IoDH
Thread-safe	 (cần synchronized)	
Lazy loading		
Hiệu năng	Trung bình hoặc thấp nếu dùng synchronized	 Cao
Dễ cài đặt	 Dễ	 Dễ
Dùng static?	Có, nhưng ít linh hoạt hơn	Dùng static inner class rất linh hoạt

Kết luận

IoDH là cách cài đặt Singleton tối ưu nhất trong Java thuần, đảm bảo:

- Dễ hiểu
- An toàn
- Tối ưu hiệu năng

2.3. Strategy Pattern

Mục tiêu

Cho phép thay đổi thuật toán (hành vi) của một đối tượng mà không cần sửa mã gốc.

Mỗi thuật toán được đóng gói riêng và có thể hoán đổi linh hoạt tại runtime.

Tình huống thực tế

Giả sử bạn đang viết một hệ thống tính phí vận chuyển. Phí có thể được tính:

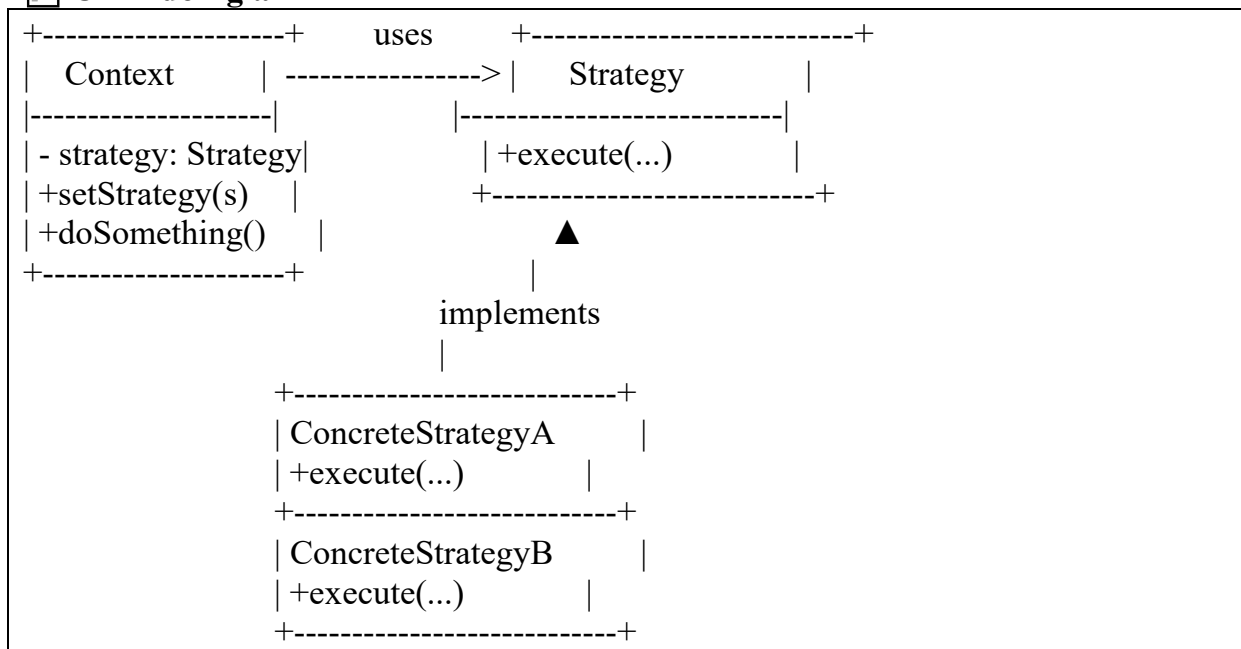
- Theo khoảng cách
- Theo trọng lượng
- Theo loại hàng hóa

👉 Bạn không muốn viết nhiều if-else dài dòng trong một hàm duy nhất. Strategy Pattern sẽ giúp tách các **thuật toán tính phí** thành các **class riêng biệt** và có thể **hoán đổi** linh hoạt.

🔄 Ý tưởng chính

- Tạo một **interface Strategy** chung cho các thuật toán.
- Các thuật toán cụ thể **cài đặt interface** này.
- Đối tượng sử dụng strategy giữ tham chiếu đến một **instance cụ thể** của strategy.
- Có thể thay đổi strategy ở **runtime**!

📄 UML đơn giản



🔧 Ví dụ minh họa bằng Java

📁 Bước 1: Định nghĩa interface Strategy

```
public interface FeeStrategy {
    double calculateFee(double distance, double weight);
}
```

📁 Bước 2: Các chiến lược cụ thể

```
public class DistanceFeeStrategy implements FeeStrategy {
    public double calculateFee(double distance, double weight) {
        return distance * 1.5;
    }
}

public class WeightFeeStrategy implements FeeStrategy {
    public double calculateFee(double distance, double weight) {
        return weight * 2.0;
    }
}
```

```
}
```



Bước 3: Lớp sử dụng Strategy

```
public class ShippingContext {  
    private FeeStrategy strategy;  
  
    public void setStrategy(FeeStrategy strategy) {  
        this.strategy = strategy;  
    }  
  
    public double calculate(double distance, double weight) {  
        return strategy.calculateFee(distance, weight);  
    }  
}
```



Sử dụng

```
public class Main {  
    public static void main(String[] args) {  
        ShippingContext context = new ShippingContext();  
  
        context.setStrategy(new DistanceFeeStrategy());  
        System.out.println("Distance Fee: " + context.calculate(100, 10)); // → 150.0  
  
        context.setStrategy(new WeightFeeStrategy());  
        System.out.println("Weight Fee: " + context.calculate(100, 10)); // → 20.0  
    }  
}
```



Lợi ích

Ưu điểm	Mô tả
Tách biệt thuật toán khỏi logic sử dụng	Giảm phức tạp, dễ bảo trì
Dễ mở rộng	Thêm chiến lược mới mà không cần sửa Context
Hỗ trợ runtime selection	Có thể thay đổi thuật toán khi chương trình đang chạy



Lưu ý khi sử dụng

- Sử dụng Strategy khi có **nhiều thuật toán tương tự nhau**, cần dễ thay đổi.
- Tránh khi các chiến lược quá đơn giản hoặc không có sự thay đổi thực sự.

2.4. Factory Pattern



Khái niệm

Factory Pattern là mẫu thiết kế **tạo đối tượng**, giúp **ẩn đi quá trình khởi tạo**, và **trả về đối tượng phù hợp** theo điều kiện cụ thể.

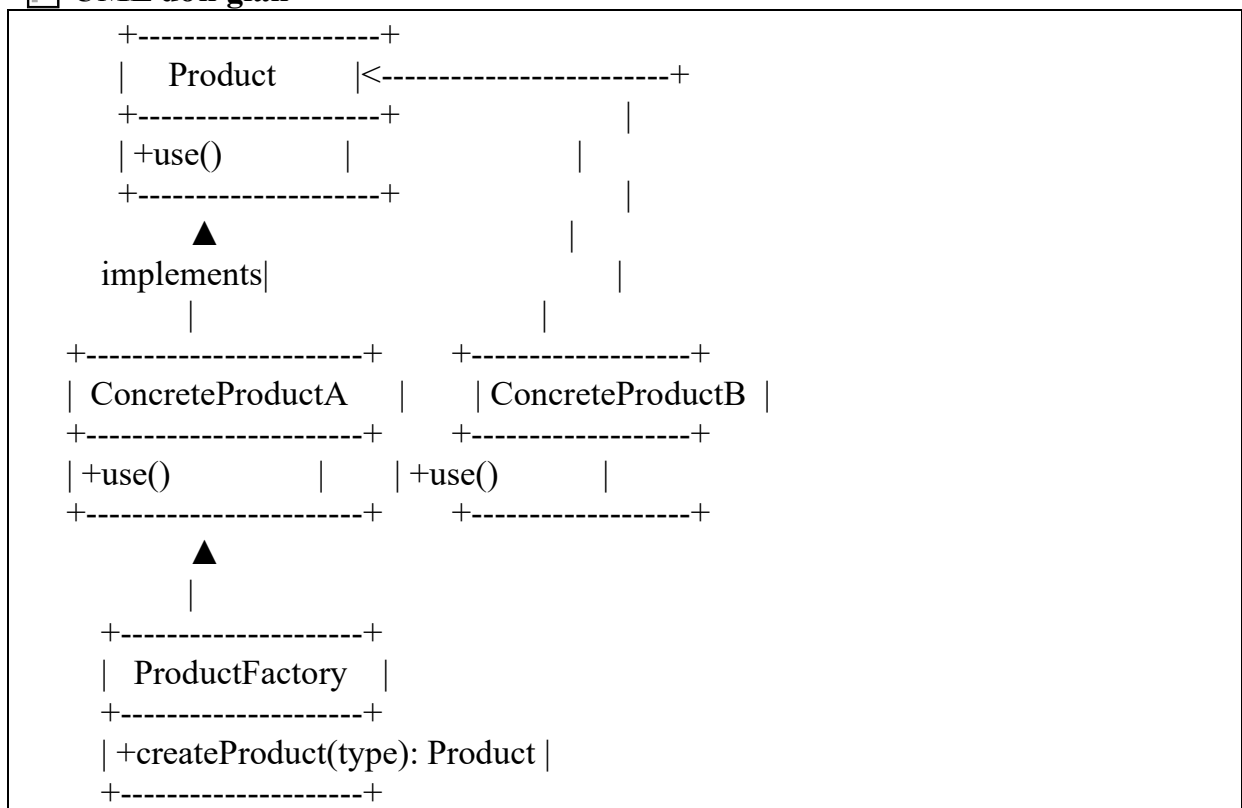
Thay vì dùng `new ClassA()` ở nhiều nơi, ta ủy thác việc khởi tạo cho một **Factory class** — từ đó giúp:

- Dễ mở rộng
- Dễ thay đổi cấu trúc lớp
- Giảm phụ thuộc giữa các phần của hệ thống

? Khi nào dùng Factory Pattern?

Tình huống	Mô tả
✓ Khi bạn muốn ẩn logic khởi tạo đối tượng	Không để new tản mát khắp chương trình
✓ Khi đối tượng cần tạo phụ thuộc vào tham số đầu vào	Ví dụ: người dùng chọn loại sản phẩm từ menu
✓ Khi có nhiều lớp con kế thừa cùng interface hoặc abstract class	Factory sẽ quyết định lớp nào phù hợp

📄 UML đơn giản



🔧 Ví dụ minh họa bằng Java

📦 Bước 1: Định nghĩa interface sản phẩm

```
public interface Animal {
    void speak();
}
```

📦 Bước 2: Các lớp cụ thể

```
public class Dog implements Animal {
    public void speak() {
        System.out.println("Gâu gâu!");
    }
}
```

```

}

public class Cat implements Animal {
    public void speak() {
        System.out.println("Meo meo!");
    }
}

```

Bước 3: Tạo lớp Factory

```

public class AnimalFactory {
    public static Animal createAnimal(String type) {
        if (type.equalsIgnoreCase("dog")) {
            return new Dog();
        } else if (type.equalsIgnoreCase("cat")) {
            return new Cat();
        } else {
            throw new IllegalArgumentException("Unknown animal type");
        }
    }
}

```

Sử dụng




```

public class Main {
    public static void main(String[] args) {
        Animal a1 = AnimalFactory.createAnimal("dog");
        Animal a2 = AnimalFactory.createAnimal("cat");



        a1.speak(); // Gâu gâu!
        a2.speak(); // Meo meo!
    }
}

```

Ưu điểm

Ưu điểm	Mô tả
 Ẩu đi chi tiết khởi tạo	Giúp client code ngắn gọn và không phụ thuộc
 Dễ mở rộng	Thêm loại mới mà không sửa client
 Tăng tính linh hoạt	Có thể kết hợp với Strategy hoặc Singleton

Nhược điểm

-  Logic phân loại (trong if-else) có thể trở nên phức tạp nếu không tổ chức tốt.
-  Không phù hợp với số lượng lớp ít, ít thay đổi.

Kết luận

Factory Pattern là "**trạm sản xuất**" đối tượng, giúp quản lý quá trình tạo ra các thể hiện phù hợp, **đảm bảo nguyên tắc đóng/mở** (Open/Closed Principle).




2.5. Chain of Responsibility Pattern

Khái niệm

Cho phép gửi yêu cầu **qua một chuỗi các đối tượng xử lý (handlers)** thay vì gắn chặt với một đối tượng cụ thể.

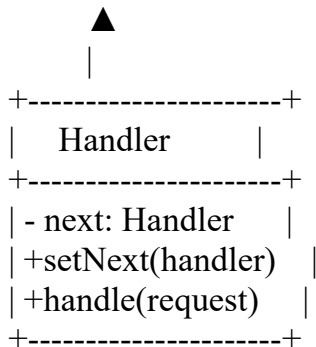
- Mỗi đối tượng trong chuỗi **có thể xử lý hoặc chuyển tiếp yêu cầu** đến handler tiếp theo.
- Giúp **giảm sự phụ thuộc** giữa đối tượng gửi yêu cầu (sender) và đối tượng xử lý (receiver).

Khi nào dùng?

Tình huống	Mô tả
 Khi có nhiều handler có thể xử lý yêu cầu	Nhưng bạn không biết trước handler nào sẽ xử lý
 Khi cần tách riêng logic xử lý theo cấp bậc	Ví dụ: xử lý lỗi, phê duyệt đơn, log hệ thống
 Khi muốn xây dựng quy trình xử lý linh hoạt, dễ mở rộng	Có thể thêm handler mới mà không ảnh hưởng code cũ

UML đơn giản

Client --> Handler1 --> Handler2 --> Handler3 --> ...



Ví dụ thực tế: xử lý yêu cầu hỗ trợ kỹ thuật

Giả sử hệ thống support có các cấp:

- **Level 1:** hỗ trợ cơ bản
- **Level 2:** hỗ trợ kỹ thuật chuyên sâu
- **Level 3:** chuyển lên quản lý

Cài đặt trong Java

Bước 1: Giao diện Handler

```
public abstract class SupportHandler {
    protected SupportHandler next;

    public void setNext(SupportHandler next) {
        this.next = next;
    }
}
```

```
}  
    public abstract void handleRequest(String issue);  
}
```

Bước 2: Các lớp Handler cụ thể

```
public class BasicSupport extends SupportHandler {  
    public void handleRequest(String issue) {  
        if (issue.contains("password")) {  
            System.out.println("BasicSupport: Resetting password...");  
        } else if (next != null) {  
            next.handleRequest(issue);  
        }  
    }  
}  
  
public class TechnicalSupport extends SupportHandler {  
    public void handleRequest(String issue) {  
        if (issue.contains("network")) {  
            System.out.println("TechnicalSupport: Fixing network...");  
        } else if (next != null) {  
            next.handleRequest(issue);  
        }  
    }  
}  
  
public class ManagerSupport extends SupportHandler {  
    public void handleRequest(String issue) {  
        System.out.println("ManagerSupport: Escalating issue – \"" + issue + "\" to  
management.");  
    }  
}
```

Bước 3: Sử dụng chuỗi xử lý

```
public class Main {  
    public static void main(String[] args) {  
        SupportHandler h1 = new BasicSupport();  
        SupportHandler h2 = new TechnicalSupport();  
        SupportHandler h3 = new ManagerSupport();  
  
        // Thiết lập chuỗi xử lý  
        h1.setNext(h2);  
        h2.setNext(h3);  
  
        // Gửi yêu cầu  
        h1.handleRequest("password reset");  
        h1.handleRequest("network failure");  
        h1.handleRequest("refund request");  
    }  
}
```

```
}  
}
```



Kết quả:

BasicSupport: Resetting password...

TechnicalSupport: Fixing network...

ManagerSupport: Escalating issue – "refund request" to management.



Lợi ích

Ưu điểm	Mô tả
<input checked="" type="checkbox"/> Giảm sự phụ thuộc giữa sender và receiver	Không cần biết cụ thể ai xử lý
<input checked="" type="checkbox"/> Dễ mở rộng	Thêm hoặc thay đổi handler không ảnh hưởng hệ thống
<input checked="" type="checkbox"/> Mỗi handler chỉ xử lý phần việc của mình	Đảm bảo nguyên tắc SRP (Single Responsibility Principle)



Nhược điểm

- ☒ Có thể khó theo dõi quá trình xử lý nếu chuỗi quá dài.
- ☒ Nếu không có handler phù hợp → có thể bỏ sót yêu cầu.



Kết luận

Chain of Responsibility thích hợp cho các tình huống xử lý phân cấp hoặc kiểm duyệt, giúp hệ thống **linh hoạt hơn, dễ bảo trì và mở rộng**.

2.6. Builder Pattern



Khái niệm

Builder Pattern là mẫu thiết kế hướng đến **việc tạo các đối tượng phức tạp**, tách rời quá trình **khởi tạo từng phần khỏi kết cấu cuối cùng** của đối tượng.

- Đặc biệt hữu ích khi đối tượng có nhiều **thuộc tính tùy chọn (optional fields)** hoặc khi quá trình tạo ra đối tượng **bao gồm nhiều bước**.



Khi nào nên dùng?

Tình huống	Mô tả
<input checked="" type="checkbox"/> Khi constructor có nhiều tham số , gây khó nhớ và dễ sai thứ tự	Đặc biệt với kiểu dữ liệu giống nhau
<input checked="" type="checkbox"/> Khi đối tượng có nhiều thuộc tính tùy chọn	Ví dụ: tạo User, Product, Report...
<input checked="" type="checkbox"/> Khi cần tạo đối tượng theo từng bước hoặc từ nhiều nguồn dữ liệu	



Vấn đề với constructor truyền thống

```
public class User {  
    public User(String name, int age, String email, String phone) {
```

```
    // quá nhiều tham số → dễ nhầm!  
    }  
}
```

Gọi constructor:

```
User u = new User("An", 20, null, null); // khó đọc, không rõ thông tin nào là gì
```

💡 Giải pháp: Builder Pattern với Fluent Interface

📦 Lớp User với Builder bên trong

```
public class User {  
    private String name;  
    private int age;  
    private String email;  
    private String phone;  
  
    private User(Builder builder) {  
        this.name = builder.name;  
        this.age = builder.age;  
        this.email = builder.email;  
        this.phone = builder.phone;  
    }  
  
    public static class Builder {  
        private String name;  
        private int age;  
        private String email;  
        private String phone;  
  
        public Builder(String name) { // required  
            this.name = name;  
        }  
  
        public Builder age(int age) {  
            this.age = age;  
            return this; // Fluent  
        }  
  
        public Builder email(String email) {  
            this.email = email;  
            return this;  
        }  
  
        public Builder phone(String phone) {  
            this.phone = phone;  
            return this;  
        }  
    }  
}
```

```

    public User build() {
        return new User(this);
    }
}

```

Sử dụng với Fluent Interface

```

User u = new User.Builder("An")
    .age(20)
    .email("an@example.com")
    .phone("0123456789")
    .build();

```

✅ Dễ đọc, dễ mở rộng, rõ ràng từng thông tin

So sánh với constructor truyền thống

Tiêu chí	Constructor truyền thống	Builder Pattern
Số lượng tham số	Có thể nhiều → khó nhớ	Chia nhỏ thành bước
Tính đọc hiểu	Thấp nếu nhiều tham số giống kiểu	Rất cao nhờ cú pháp tênPhươngThức(...)
Khả năng mở rộng	Khó khăn, phải tạo nhiều constructor	Rất dễ, chỉ cần thêm phương thức builder
Hỗ trợ optional field	Phải truyền null	Chỉ gọi khi cần
Immutable object?	Có thể	✅ Dễ hỗ trợ

✅ Ưu điểm

- ✅ Rõ ràng, dễ đọc (Fluent interface)
- ✅ Tránh constructor khổng lồ (telescoping constructor)
- ✅ Hỗ trợ tạo object bất biến (immutable)
- ✅ Dễ mở rộng thêm thuộc tính mà không phá vỡ code cũ

⚠️ Nhược điểm

- ❌ Phức tạp hơn khi dùng cho các object rất đơn giản
- ❌ Viết code nhiều hơn ban đầu

Kết luận

Builder Pattern giúp tạo ra các đối tượng phức tạp một cách linh hoạt, **đọc như ngôn ngữ tự nhiên**, đồng thời **đảm bảo tính nhất quán và dễ bảo trì** trong các hệ thống lớn.

2.7. Template Method Pattern

Khái niệm

Template Method Pattern định nghĩa **bộ khung (template)** của một thuật toán trong lớp cha (**abstract class**), nhưng cho phép **các bước cụ thể được định nghĩa lại (override)** bởi các lớp con.

- Thuật toán tổng quát được **cố định**, nhưng một số bước cụ thể thì **linh hoạt thay đổi**.
- Mục tiêu là **tái sử dụng logic cốt lõi** và **mở rộng hành vi từng phần** khi cần thiết.

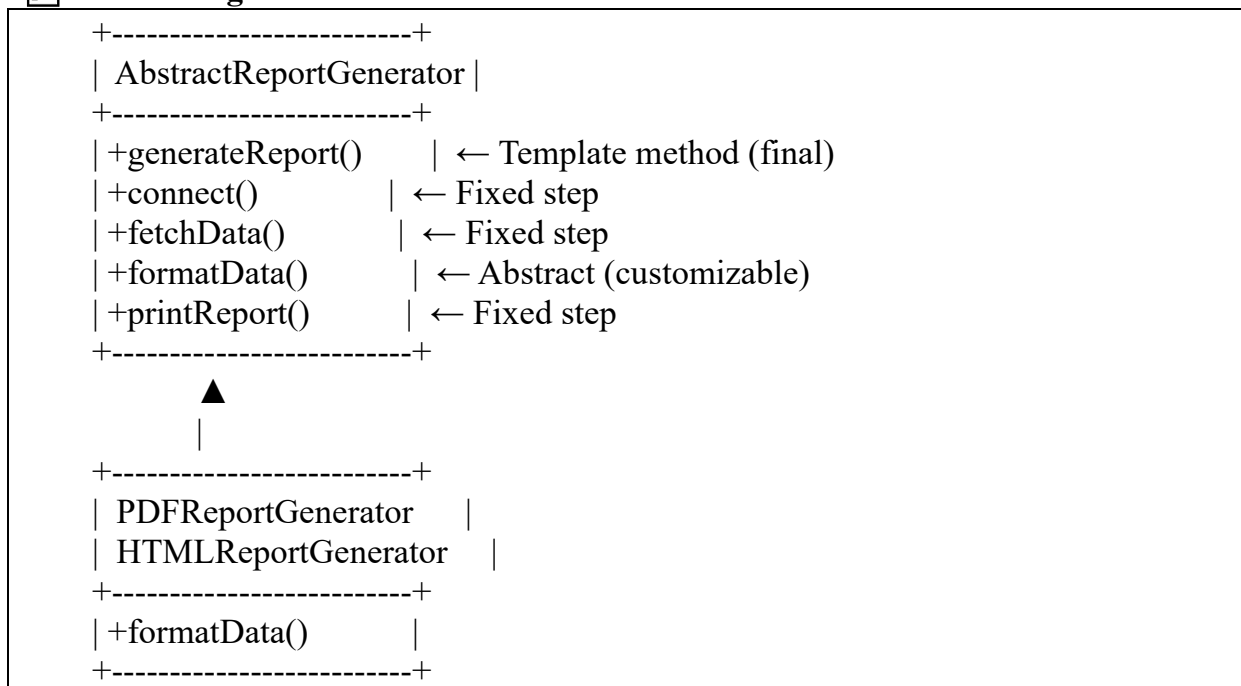
Ví dụ thực tế

Hệ thống **tạo báo cáo** luôn có các bước:

1. Kết nối dữ liệu
2. Trích xuất dữ liệu
3. Định dạng kết quả
4. In/hiển thị kết quả

 Nhưng mỗi loại báo cáo (báo cáo PDF, HTML, Excel...) có thể khác nhau ở bước **định dạng kết quả** → sử dụng Template Method Pattern.

UML đơn giản



Ví dụ minh họa bằng Java

Lớp cha định nghĩa khuôn mẫu

```
public abstract class ReportGenerator {

    // Template method (không cho override)
    public final void generateReport() {
        connect();
```

```

        fetchData();
        formatData(); // ← cho phép lớp con tùy chỉnh
        printReport();
    }

    protected void connect() {
        System.out.println("Connecting to database...");
    }

    protected void fetchData() {
        System.out.println("Fetching data...");
    }

    protected abstract void formatData(); // Lớp con sẽ định nghĩa

    protected void printReport() {
        System.out.println("Printing report...");
    }
}

```



Các lớp con override bước cụ thể

```

public class PDFReport extends ReportGenerator {
    protected void formatData() {
        System.out.println("Formatting data as PDF...");
    }
}

public class HTMLReport extends ReportGenerator {
    protected void formatData() {
        System.out.println("Formatting data as HTML...");
    }
}

```



Sử dụng

```

public class Main {
    public static void main(String[] args) {
        ReportGenerator pdf = new PDFReport();
        pdf.generateReport();
        // Output:
        // Connecting to database...
        // Fetching data...
        // Formatting data as PDF...
        // Printing report...

        ReportGenerator html = new HTMLReport();
        html.generateReport();
    }
}

```

}

✓ Ưu điểm

Ưu điểm	Mô tả
✓ Tái sử dụng code chung	Các bước cố định viết một lần trong abstract class
✓ Dễ mở rộng từng phần	Chỉ cần override các bước cụ thể
✓ Áp dụng nguyên tắc Hollywood: “Don’t call us, we’ll call you”	Lớp cha điều phối, lớp con chỉ định nghĩa phần cần thiết

⚠ Nhược điểm

- ✗ Cần hiểu kỹ kiến trúc kế thừa
- ✗ Hơi cứng nhắc nếu thuật toán có nhiều nhánh rẽ quá khác nhau
- ✗ Không phù hợp nếu yêu cầu runtime thay đổi linh hoạt (nên dùng Strategy)

✦ Kết luận

Template Method Pattern cho phép bạn thiết kế thuật toán theo kiểu “**bộ khung cố định, linh hoạt từng bước**”, giúp **quản lý sự phức tạp và tăng khả năng mở rộng** khi có nhiều biến thể của cùng một quy trình.

2.8. State Pattern

🔄 Khái niệm

State Pattern cho phép một đối tượng **thay đổi hành vi (behavior)** của nó **khi trạng thái nội tại thay đổi**.

Thay vì dùng các if-else hoặc switch phức tạp, mỗi trạng thái sẽ được đóng gói thành một lớp riêng biệt.

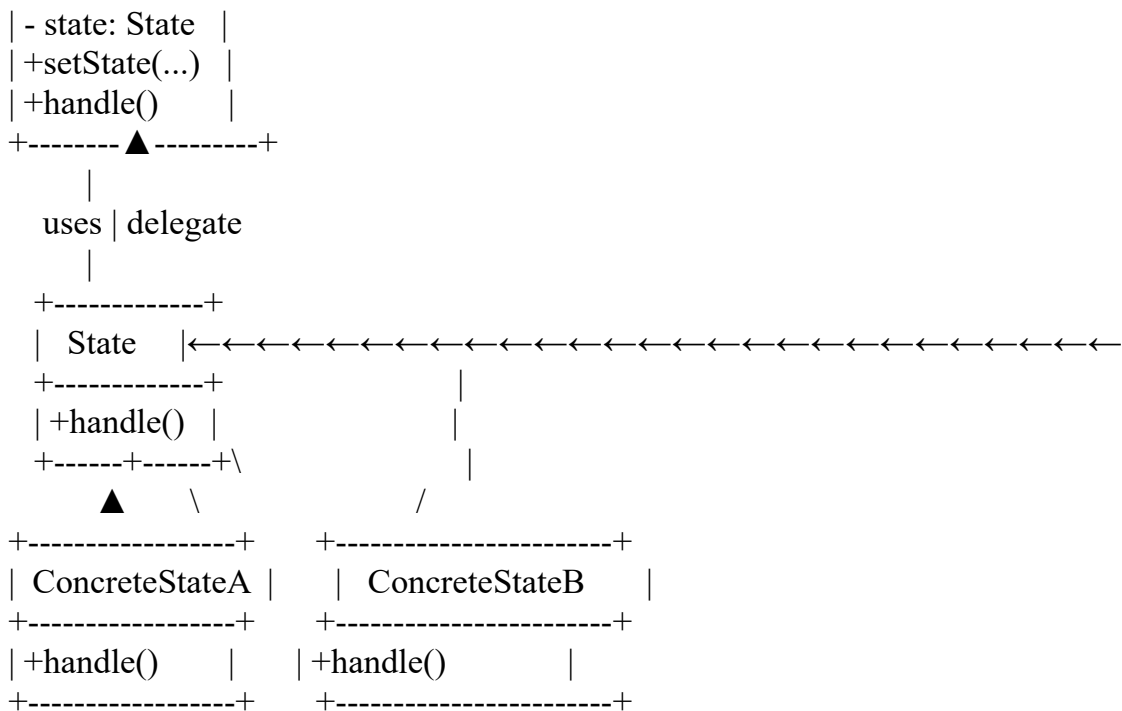
- Đối tượng chính **ủy quyền hành vi** cho đối tượng trạng thái hiện tại.
- Là một ví dụ điển hình của **biến đổi hành vi theo ngữ cảnh (context-aware behavior)**.

? Khi nào nên dùng?

Tình huống	Mô tả
✓ Khi một đối tượng có nhiều trạng thái logic và hành vi thay đổi theo từng trạng thái	Ví dụ: máy bán hàng, cửa xoay, trình soạn thảo
✓ Khi có quá nhiều câu lệnh if-else/switch-case lặp đi lặp lại	Gây khó bảo trì, dễ bug
✓ Khi cần phân tách rõ ràng logic cho từng trạng thái	

📄 UML đơn giản

```
+-----+
| Context |
+-----+
```

 Ví dụ minh họa: Máy bán hàng (Vending Machine)

Máy có các trạng thái:

- NoCoinState: chưa có tiền
- HasCoinState: đã nhét tiền
- SoldOutState: hết hàng

Cài đặt mẫu bằng Java

 Giao diện State

```
public interface State {
    void insertCoin();
    void pressButton();
    void dispense();
}
```

Các trạng thái cụ thể

```
public class NoCoinState implements State {
    public void insertCoin() {
        System.out.println("Coin inserted.");
    }
    public void pressButton() {
        System.out.println("Insert coin first.");
    }
    public void dispense() {
        System.out.println("No coin inserted.");
    }
}
```

```

public class HasCoinState implements State {
    public void insertCoin() {
        System.out.println("Already has coin.");
    }
    public void pressButton() {
        System.out.println("Product dispensed.");
    }
    public void dispense() {
        System.out.println("Thank you!");
    }
}

```



Lớp Context: VendingMachine

```

public class VendingMachine {
    private State state;

    public void setState(State state) {
        this.state = state;
    }

    public void insertCoin() {
        state.insertCoin();
    }

    public void pressButton() {
        state.pressButton();
        state.dispense();
    }
}

```



Sử dụng

```

public class Main {
    public static void main(String[] args) {
        VendingMachine vm = new VendingMachine();

        vm.setState(new NoCoinState());
        vm.pressButton();    // → Insert coin first.
        vm.insertCoin();     // → Coin inserted.

        vm.setState(new HasCoinState());
        vm.insertCoin();     // → Already has coin.
        vm.pressButton();    // → Product dispensed. Thank you!
    }
}

```



So sánh với if-else dài dòng

Cách cũ:

```

if (state == "NO_COIN") {
    // xử lý
} else if (state == "HAS_COIN") {
    // xử lý khác
} else if (...) {
    ...
}

```

➔ **Khó mở rộng, dễ sai, khó test, vi phạm nguyên tắc SRP.**

Cách mới với State Pattern:

- Tách riêng mỗi trạng thái thành **một class chuyên trách**
- **Không cần if-else**, chỉ cần gọi `context.setState(...)` và `state.handle()`

✅ Ưu điểm

Ưu điểm	Mô tả
✅ Tách biệt hành vi theo trạng thái	Mỗi lớp xử lý đúng phần việc của mình
✅ Dễ mở rộng	Thêm trạng thái mới mà không phá vỡ code cũ
✅ Loại bỏ if-else phức tạp	Clean code, dễ bảo trì

⚠️ Nhược điểm

- ❌ Số lượng lớp tăng lên đáng kể
- ❌ Có thể gây phức tạp hóa nếu trạng thái đơn giản

📌 Kết luận

State Pattern là giải pháp tuyệt vời khi đối tượng có hành vi thay đổi theo trạng thái — giúp mã rõ ràng hơn, dễ mở rộng và dễ kiểm thử hơn nhiều so với if-else.






3. Tổng kết và So sánh các Design Pattern đã học

📌 3.1. Mục tiêu và cách dùng của từng pattern

Pattern	Nhóm	Mục tiêu chính	Tình huống điển hình
Singleton	Creational	Đảm bảo chỉ có một thể hiện duy nhất	Cấu hình hệ thống, Logger, DB connection
IoDH	Creational	Cải tiến Singleton cho đa luồng + hiệu suất	Singleton thread-safe, hiệu năng cao
Strategy	Behavioral	Thay đổi thuật toán tại runtime	Sắp xếp, tính phí, định giá
Factory	Creational	Ẩn việc tạo đối tượng , chọn loại phù hợp	Tạo sản phẩm tùy chọn người dùng
Chain of Responsibility	Behavioral	Tạo chuỗi xử lý linh hoạt , giảm phụ thuộc	Hệ thống support, phê duyệt đơn
Builder	Creational	Tạo đối tượng phức tạp , dễ đọc, dễ mở rộng	Tạo đối tượng có nhiều field tùy chọn

Template Method	Behavioral	Xây dựng thuật toán khung , cho lớp con tùy biến	Tạo báo cáo, xử lý dữ liệu nhiều bước
State	Behavioral	Hành vi thay đổi theo trạng thái nội tại	Máy bán hàng, game, editor, quy trình duyệt






🔍 3.2. Phân biệt Strategy – State – Template

So sánh	Strategy	State	Template Method
 Thay đổi hành vi	✅ Có	✅ Có	✅ Có
 Mục tiêu chính	Chọn thuật toán từ bên ngoài	Đổi hành vi khi trạng thái thay đổi	Cố định khung , cho phép tùy biến bước nhỏ
 Cách mở rộng	Thêm strategy class	Thêm state class	Override bước cụ thể trong abstract class
 Thay đổi runtime	✅ Có thể thay đổi linh hoạt	✅ Có (thường tự động theo trạng thái)	❌ Thường cố định
 Sử dụng	Context có setStrategy(...)	Context có setState(...)	Lớp cha gọi templateMethod()
Ví dụ	Giảm giá, tính phí	Máy bán hàng, cửa tự động	Tạo báo cáo HTML/PDF

💡 Ghi nhớ:

- Strategy = chọn *cách làm*
- State = *hành vi thay đổi theo trạng thái*
- Template = *giữ khung, thay thế chi tiết*

📦 3.3. Khi nào dùng Builder vs Factory?

So sánh	Factory Pattern	Builder Pattern
 Mục tiêu	Ẩn logic khởi tạo, chọn lớp phù hợp	Xây dựng đối tượng phức tạp theo từng bước
 Đối tượng tạo ra	Có thể là nhiều loại con (subclass/interface)	Một loại duy nhất với nhiều field
 Cấu trúc	Dựa vào if-else, trả về subclass	Dựa vào builder, có build()
 Sử dụng khi	Không biết chính xác class cần khởi tạo	Có nhiều tham số, hoặc tham số tùy chọn
 Ví dụ	Tạo Animal: Dog, Cat...	Tạo User, Product, Pizza...

✅ Gợi ý ôn tập & luyện tập

- Hãy thử phân tích một ứng dụng bạn biết (ví dụ: Facebook, Shopee, máy ATM...)
 ➤ Có thể áp dụng pattern nào trong số trên?

- Tập cài đặt lại từng pattern với một ví dụ **khác biệt** (không trùng handout).

BÀI TẬP CÓ HƯỚNG DẪN

Bài tập 1: Hệ thống đặt món ăn theo chiến lược khuyến mãi

Đề bài:

Bạn được giao xây dựng một **hệ thống đặt món ăn nhanh**, trong đó có thể áp dụng **nhiều loại khuyến mãi khác nhau** tùy từng thời điểm. Hệ thống yêu cầu:

- Người dùng chọn món ăn và số lượng
- Áp dụng **chiến lược khuyến mãi phù hợp** (ví dụ: giảm 10%, mua 2 tặng 1, miễn phí ship)
- Tính tổng tiền phải thanh toán sau khi áp dụng khuyến mãi

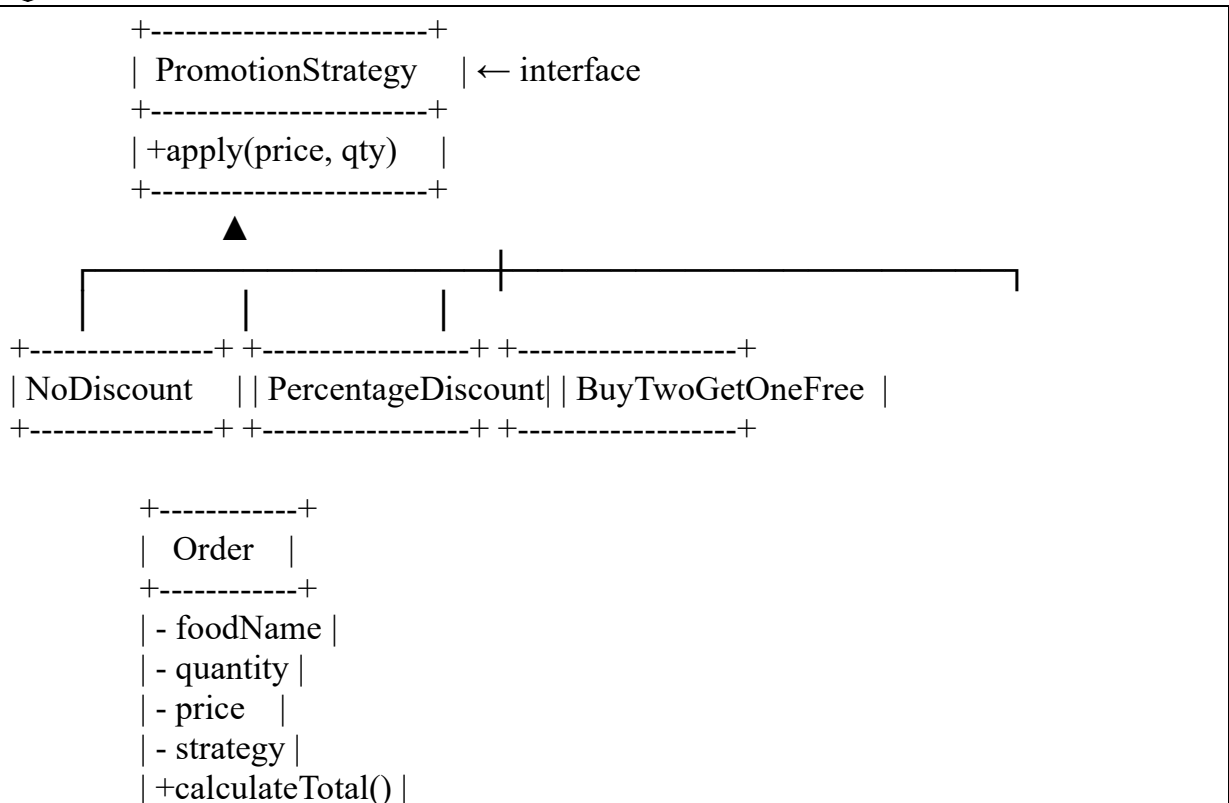
Hãy **thiết kế và cài đặt hệ thống này sử dụng Strategy Pattern**, sao cho:

- Có thể dễ dàng thêm chiến lược mới mà không sửa code cũ
- Có thể thay đổi khuyến mãi tại runtime

Phân tích yêu cầu:

- **Biến đổi chính là "chiến lược khuyến mãi" → phù hợp với Strategy Pattern**
- Mỗi chiến lược nên được đóng gói trong một class riêng
- Lớp Order sẽ chứa thông tin món ăn, số lượng, và giữ PromotionStrategy
- PromotionStrategy là interface với phương thức apply(double originalPrice, int quantity)

Thiết kế class





Cài đặt Java mẫu



Interface PromotionStrategy

```
public interface PromotionStrategy {  
    double apply(double pricePerUnit, int quantity);  
}
```



Các chiến lược cụ thể

```
public class NoDiscount implements PromotionStrategy {  
    public double apply(double price, int qty) {  
        return price * qty;  
    }  
}  
  
public class PercentageDiscount implements PromotionStrategy {  
    private double percent;  
  
    public PercentageDiscount(double percent) {  
        this.percent = percent;  
    }  
  
    public double apply(double price, int qty) {  
        return price * qty * (1 - percent);  
    }  
}  
  
public class BuyTwoGetOneFree implements PromotionStrategy {  
    public double apply(double price, int qty) {  
        int payableQty = qty - (qty / 3); // Mua 3 tính tiền 2  
        return price * payableQty;  
    }  
}
```



Lớp Order

```
public class Order {  
    private String foodName;  
    private int quantity;  
    private double unitPrice;  
    private PromotionStrategy strategy;  
  
    public Order(String foodName, int quantity, double price, PromotionStrategy  
strategy) {  
        this.foodName = foodName;  
        this.quantity = quantity;  
        this.unitPrice = price;  
        this.strategy = strategy;  
    }  
}
```

```

    }

    public double calculateTotal() {
        return strategy.apply(unitPrice, quantity);
    }

    public void printReceipt() {
        System.out.printf("Món: %s | SL: %d | Giá/sp: %.2f\n", foodName, quantity,
            unitPrice);
        System.out.printf("Tổng thanh toán sau khuyến mãi: %.2f\n", calculateTotal());
    }
}

```

✅ Demo sử dụng

```

public class Main {
    public static void main(String[] args) {
        Order o1 = new Order("Gà rán", 3, 50000, new BuyTwoGetOneFree());
        Order o2 = new Order("Pizza", 2, 120000, new PercentageDiscount(0.1));
        Order o3 = new Order("Khoai tây", 1, 30000, new NoDiscount());

        o1.printReceipt();
        o2.printReceipt();
        o3.printReceipt();
    }
}

```

✅ Kết quả mẫu (console)

```

Món: Gà rán | SL: 3 | Giá/sp: 50000.00
Tổng thanh toán sau khuyến mãi: 100000.00

Món: Pizza | SL: 2 | Giá/sp: 120000.00
Tổng thanh toán sau khuyến mãi: 216000.00

Món: Khoai tây | SL: 1 | Giá/sp: 30000.00
Tổng thanh toán sau khuyến mãi: 30000.00

```

💡 Mở rộng gợi ý

- Thêm chiến lược FreeshipOverAmount
- Cho phép thay đổi strategy bằng order.setStrategy(...)
- Cho phép nhập từ bàn phím hoặc giao diện console

🔗 Bài tập 2: Hệ thống khởi tạo Pizza linh hoạt

📄 Đề bài:

Một chuỗi cửa hàng Pizza cần xây dựng hệ thống tạo đơn hàng Pizza theo yêu cầu khách hàng. Mỗi loại Pizza có thể thuộc một trong các **kiểu phổ biến (Classic, Hawaiian, Seafood)**, với các **tùy chọn linh hoạt** như:

- Cỡ pizza: Small, Medium, Large
- Thêm phô mai, viên xúc xích, thêm topping đặc biệt
- Ghi chú thêm (ghi trên hóa đơn)

Yêu cầu:

1. Sử dụng **Factory Pattern** để khởi tạo **đúng loại Pizza cơ bản** (Classic, Hawaiian, Seafood)
2. Sử dụng **Builder Pattern** để tùy chỉnh cấu hình chi tiết của Pizza (cỡ, phô mai, topping...)



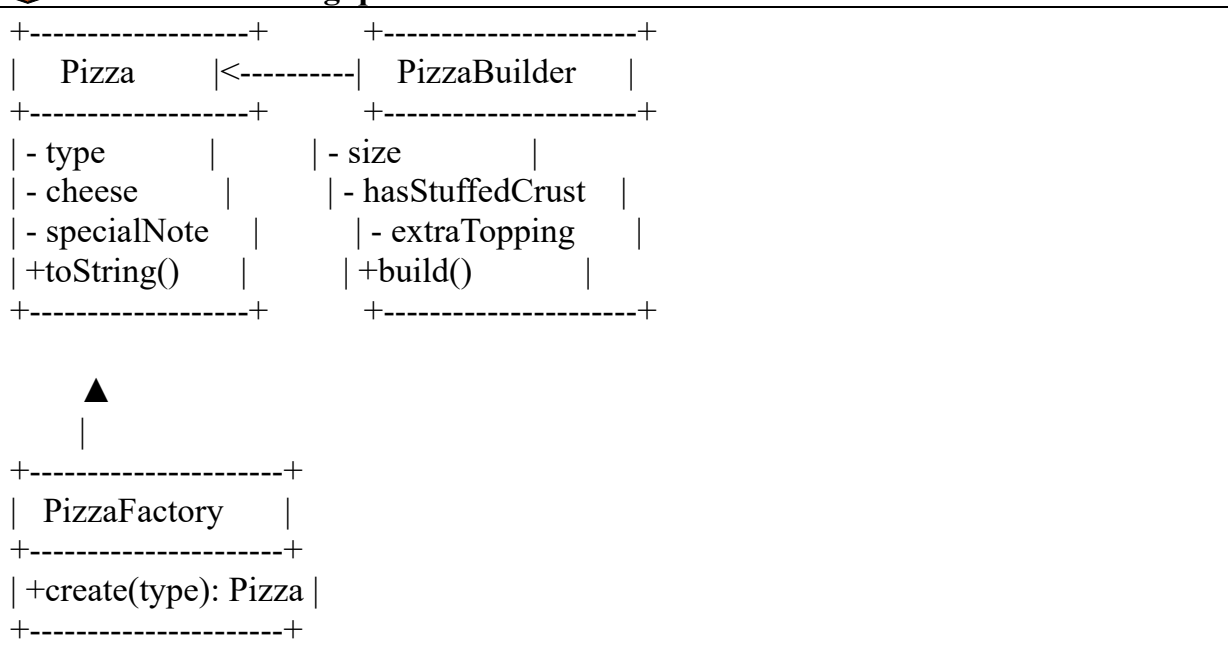
Phân tích yêu cầu thiết kế

- Việc chọn loại Pizza (Classic, Hawaiian, ...) → dùng **Factory Pattern**
- Việc thêm tùy chọn linh hoạt → dùng **Builder Pattern**

→ Kết hợp cả hai pattern: **Factory tạo Pizza base**, rồi dùng **Builder để “đắp” thêm các tùy chọn!**



Thiết kế class tổng quát



Lời giải bằng Java



Class Pizza

```

public class Pizza {
    String type;
    String size;
    boolean cheese;
    boolean stuffedCrust;
    String extraTopping;
    String note;
}
  
```



```

    public Pizza(String type, String size, boolean cheese, boolean stuffedCrust, String
    topping, String note) {
        this.type = type;
        this.size = size;
        this.cheese = cheese;
        this.stuffedCrust = stuffedCrust;
        this.extraTopping = topping;
        this.note = note;
    }

    public String toString() {
        return String.format("Pizza: %s [%s] %s %s\nTopping: %s\nNote: %s",
            type, size,
            cheese ? "+ Cheese" : "",
            stuffedCrust ? "+ Stuffed Crust" : "",
            extraTopping != null ? extraTopping : "None",
            note != null ? note : "N/A");
    }
}

```

Builder

```

public class PizzaBuilder {
    private String type;
    private String size = "Medium"; // default
    private boolean cheese = false;
    private boolean stuffedCrust = false;
    private String extraTopping;
    private String note;

    public PizzaBuilder(String type) {
        this.type = type;
    }

    public PizzaBuilder size(String size) {
        this.size = size;
        return this;
    }

    public PizzaBuilder addCheese() {
        this.cheese = true;
        return this;
    }

    public PizzaBuilder stuffedCrust() {
        this.stuffedCrust = true;
        return this;
    }
}

```

```

    }

    public PizzaBuilder topping(String topping) {
        this.extraTopping = topping;
        return this;
    }

    public PizzaBuilder note(String note) {
        this.note = note;
        return this;
    }

    public Pizza build() {
        return new Pizza(type, size, cheese, stuffedCrust, extraTopping, note);
    }
}

```

✓ PizzaFactory

```

public class PizzaFactory {
    public static PizzaBuilder createPizza(String type) {
        switch (type.toLowerCase()) {
            case "classic":
            case "hawaiian":
            case "seafood":
                return new PizzaBuilder(capitalize(type));
            default:
                throw new IllegalArgumentException("Unknown pizza type: " + type);
        }
    }

    private static String capitalize(String str) {
        return str.substring(0, 1).toUpperCase() + str.substring(1);
    }
}

```

Sử dụng

```

public class Main {
    public static void main(String[] args) {
        Pizza p1 = PizzaFactory.createPizza("classic")
            .size("Large")
            .addCheese()
            .stuffedCrust()
            .topping("Mushroom")
            .note("Không cắt sẵn")
            .build();

        Pizza p2 = PizzaFactory.createPizza("seafood")

```

```
        .topping("Tôm sú")
        .build();
```

```
System.out.println(p1);
System.out.println("-----");
System.out.println(p2);
    }
}
```

✅ Kết quả mẫu

Pizza: Classic [Large] + Cheese + Stuffed Crust

Topping: Mushroom

Note: Không cắt sẵn

Pizza: Seafood [Medium]

Topping: Tôm sú

Note: N/A

📖 Kiến thức được vận dụng

- ✅ Phân biệt Factory (chọn loại Pizza) và Builder (tùy chỉnh chi tiết)
- ✅ Fluent Interface trong Builder
- ✅ Khả năng mở rộng linh hoạt mà không sửa code cũ

💡 Gợi ý mở rộng

- Cho phép đọc thông tin từ giao diện nhập console
- Thêm đơn hàng vào List<Pizza> để tính tổng giá
- Thêm giá tiền dựa theo kích cỡ và topping

TRẮC NGHIỆM

🔹 Mức độ DỄ (Câu 1–7)

Câu 1:

Mục tiêu chính của Singleton Pattern là gì?

- A. Tăng tốc độ xử lý
- B. Đảm bảo chỉ có một thể hiện của lớp
- C. Cho phép mở rộng lớp dễ dàng
- D. Tăng khả năng kế thừa

Câu 2:

Cách nào sau đây thường dùng để tạo Singleton trong Java?

- A. Constructor public
- B. Dùng lớp con

- C. Dùng static instance và private constructor
- D. Gọi super() từ ngoài lớp

Câu 3:

Strategy Pattern chủ yếu được dùng để:

- A. Khởi tạo đối tượng
- B. Cố định thuật toán
- C. Hoán đổi thuật toán tại runtime
- D. Tạo cây đối tượng phức tạp

Câu 4:

Trong Factory Pattern, client code sẽ gọi trực tiếp:

- A. Constructor của lớp con
- B. Interface
- C. Factory method
- D. Abstract method

Câu 5:

Trong Builder Pattern, tại sao nên dùng fluent interface?

- A. Để tối ưu runtime
- B. Để dễ đọc và gọi lệnh theo chuỗi
- C. Để giảm số lượng class
- D. Để tránh dùng constructor

Câu 6:

Lớp nào thường định nghĩa “thuật toán khung” trong Template Method Pattern?

- A. Interface
- B. Subclass
- C. Abstract class
- D. Enum

Câu 7:

Chain of Responsibility giúp đạt được điều gì?

- A. Tối ưu hóa cấu trúc dữ liệu
- B. Gộp nhiều thuật toán thành một
- C. Tạo chuỗi các handler có thể xử lý yêu cầu
- D. Truy cập trực tiếp vào private fields

◆ Mức độ TRUNG BÌNH (Câu 8–14)

Câu 8:

Điểm khác biệt giữa Factory và Builder là gì?

- A. Builder trả về nhiều loại đối tượng hơn Factory
- B. Factory xử lý đối tượng phức tạp tốt hơn

- C. Builder giúp cấu hình đối tượng từng bước
- D. Không có khác biệt

Câu 9:

Trong Strategy Pattern, Context lớp có nhiệm vụ gì?

- A. Chứa nhiều Strategy kế thừa
- B. Chọn và sử dụng Strategy phù hợp
- C. Thực hiện thuật toán mặc định
- D. Không có nhiệm vụ gì

Câu 10:

IoDH (Initialization-on-Demand Holder) cải tiến Singleton bằng cách nào?

- A. Dùng synchronized toàn bộ lớp
- B. Cho phép kế thừa Singleton
- C. Đảm bảo lazy-loading và thread-safe
- D. Tự động tạo lại khi lỗi

Câu 11:

Trong Template Method Pattern, phương pháp templateMethod() nên:

- A. Là abstract
- B. Là interface method
- C. Là final
- D. Là private

Câu 12:

Trong State Pattern, mỗi trạng thái được cài đặt:

- A. Như một hàm static
- B. Bằng một class cài đặt interface chung
- C. Bằng một enum
- D. Trong context trực tiếp

Câu 13:

Dấu hiệu bạn nên dùng Chain of Responsibility là khi:

- A. Bạn muốn tính toán song song
- B. Bạn có nhiều bước xử lý nối tiếp, độc lập
- C. Mỗi xử lý cần truy cập toàn bộ dữ liệu
- D. Bạn chỉ có 1 xử lý duy nhất

Câu 14:

Builder Pattern giúp tạo object bất biến (immutable) nhờ:

- A. Gán trực tiếp vào field public
- B. Khởi tạo thông qua final constructor

- C. Tách quá trình build ra khỏi object
- D. Không cho phép kế thừa class

◆ Mức độ KHÓ (Câu 15–20)

Câu 15:

Trong Factory Pattern, nếu thêm loại mới (VD: VeganPizza), bạn cần:

- A. Sửa tất cả lớp con
- B. Thêm vào factory method
- C. Sửa constructor của Pizza
- D. Không làm gì cả

Câu 16:

Lỗi phổ biến khi viết State Pattern là gì?

- A. Không override đủ method
- B. Đặt logic chuyển trạng thái trong lớp sai
- C. Dùng static thay vì object
- D. Kết hợp nhầm với Strategy

Câu 17:

Cách tốt nhất để thêm logic khuyến mãi trong hệ thống đặt hàng là:

- A. Viết if-else trong Order
- B. Dùng State Pattern
- C. Dùng Strategy Pattern
- D. Tạo nhiều class con của Order

Câu 18:

Đây là lợi ích lớn nhất của Template Method Pattern?

- A. Giúp code chạy nhanh hơn
- B. Loại bỏ tất cả kế thừa
- C. Tái sử dụng phần khung thuật toán
- D. Giảm số lượng class cần viết

Câu 19:

Bạn đang xử lý một editor văn bản với trạng thái: "Chế độ xem", "Chế độ chỉnh sửa", "Chế độ highlight". Pattern nào phù hợp nhất?

- A. Strategy
- B. Builder
- C. Factory
- D. State

Câu 20:

Giả sử có đoạn code `Pizza p = new Pizza("Seafood", true, false, "Large", "Extra cheese")`. Bạn được yêu cầu mở rộng linh hoạt cách gọi. Pattern nào phù hợp nhất?

- A. Strategy
- B. Factory
- C. Template
- D. Builder

BÀI LUYỆN TẬP

✓ Bài 1 – Singleton cơ bản (dễ)

Đề bài: Viết một lớp Logger đảm bảo chỉ có duy nhất một đối tượng được tạo trong toàn chương trình. Mỗi lần gọi log(String message), nó sẽ in ra màn hình với timestamp.

Hướng dẫn:

- Dùng Singleton Pattern với constructor private
- Cung cấp phương thức getInstance() kiểu static
- Gợi ý dùng SimpleDateFormat để định dạng thời gian

✓ Bài 2 – Singleton nâng cao với IoDH (dễ)

Đề bài: Viết lại bài Logger phía trên bằng cách sử dụng **Initialization-on-demand holder** để đảm bảo thread-safe và lazy-loading.

Hướng dẫn:

- Sử dụng inner static class LoggerHolder
- getInstance() trả về LoggerHolder.INSTANCE

✓ Bài 3 – Strategy: tính điểm môn học (dễ)

Đề bài: Một hệ thống quản lý điểm cho các môn học có cách tính điểm khác nhau:

- Môn A: 50% giữa kỳ + 50% cuối kỳ
- Môn B: 30% giữa kỳ + 70% cuối kỳ
- Môn C: chỉ tính cuối kỳ

Hướng dẫn:

- Tạo interface GradingStrategy với double calculate(mid, finalScore)
- Tạo các lớp con cho từng kiểu tính điểm
- Lớp Subject chứa strategy tương ứng

✓ Bài 4 – Factory Pattern: tạo hình học (dễ)

Đề bài: Viết một factory cho phép tạo các đối tượng hình học như Circle, Square, Triangle, tất cả cài đặt interface Shape.

Hướng dẫn:

- Tạo interface Shape với draw()
- Cài đặt các lớp Circle, Square, Triangle
- Tạo lớp ShapeFactory với Shape create(String type)

✓ Bài 5 – Builder Pattern: cấu hình máy tính (dễ-trung)

Đề bài: Viết class Computer có các thông tin sau:

- CPU (bắt buộc)
- RAM, ổ cứng, card đồ họa, hệ điều hành (tùy chọn)

Hãy sử dụng Builder Pattern để tạo đối tượng Computer linh hoạt.

Hướng dẫn:

- Class ComputerBuilder nhận cpu trong constructor
- Cung cấp các method kiểu .ram(...).os(...).build()
- Đối tượng Computer được tạo từ builder

✓ **Bài 6 – Chain of Responsibility: xử lý hỗ trợ kỹ thuật (trung)**

Đề bài: Mô phỏng hệ thống xử lý yêu cầu hỗ trợ kỹ thuật gồm 3 cấp:

- Level 1: xử lý các vấn đề đơn giản (quên mật khẩu)
- Level 2: xử lý vấn đề mạng, hệ điều hành
- Level 3: chuyển đến quản lý

Hướng dẫn:

- Tạo interface SupportHandler với method handle(String issue)
- Tạo 3 class Level1Support, Level2Support, ManagerSupport
- Sử dụng setNext() để tạo chuỗi xử lý

✓ **Bài 7 – Template Method: xuất báo cáo (trung)**

Đề bài: Viết hệ thống xuất báo cáo với khung quy trình sau:

1. Kết nối dữ liệu
2. Trích xuất dữ liệu
3. Định dạng
4. Xuất ra console

Tạo lớp cha ReportTemplate, lớp con PDFReport, HTMLReport định nghĩa bước (3).

Hướng dẫn:

- Viết method generateReport() trong lớp cha (final)
- Các bước 1, 2, 4 viết sẵn trong lớp cha
- Bước 3 là abstract để lớp con override

✓ **Bài 8 – State Pattern: máy bán hàng tự động (trung)**

Đề bài: Mô phỏng máy bán hàng có 3 trạng thái:

- NoCoin: chưa cho tiền
- HasCoin: đã nhận tiền
- SoldOut: hết hàng

Mỗi trạng thái có hành vi khác nhau với insertCoin(), pressButton(), dispense().

Hướng dẫn:

- Tạo interface VendingState
- Viết các class NoCoinState, HasCoinState, SoldOutState
- Lớp VendingMachine giữ một VendingState hiện tại

✓ **Bài 9 – So sánh Strategy vs State (trung)**

Đề bài: Hãy viết hai phiên bản cho hệ thống nhân vật trong game:

- **Phiên bản 1 (Strategy):** mỗi nhân vật có thể thay đổi vũ khí linh hoạt
- **Phiên bản 2 (State):** nhân vật thay đổi hành vi theo trạng thái “tấn công”, “phòng thủ”, “ẩn nấp”

Hướng dẫn:

- Với Strategy: mỗi vũ khí là một class WeaponStrategy
- Với State: mỗi trạng thái là một class CharacterState

✓ Bài 10 – Builder + Factory kết hợp: hệ thống đặt hàng Pizza (trung)

Đề bài: Xây dựng hệ thống đặt hàng Pizza. Mỗi Pizza có loại (Classic, Hawaiian...) và các tùy chọn: kích cỡ, phô mai, topping, viền xúc xích.

Hướng dẫn:

- Dùng Factory để tạo Pizza cơ bản
- Dùng Builder để thêm tùy chọn vào Pizza
- Viết phương thức build() trả về đối tượng Pizza hoàn chỉnh

BÀI DỰ ÁN

📁 Dự án Mini 1 (mức DỄ): Trình phát nhạc đơn giản

🎯 Đề bài:

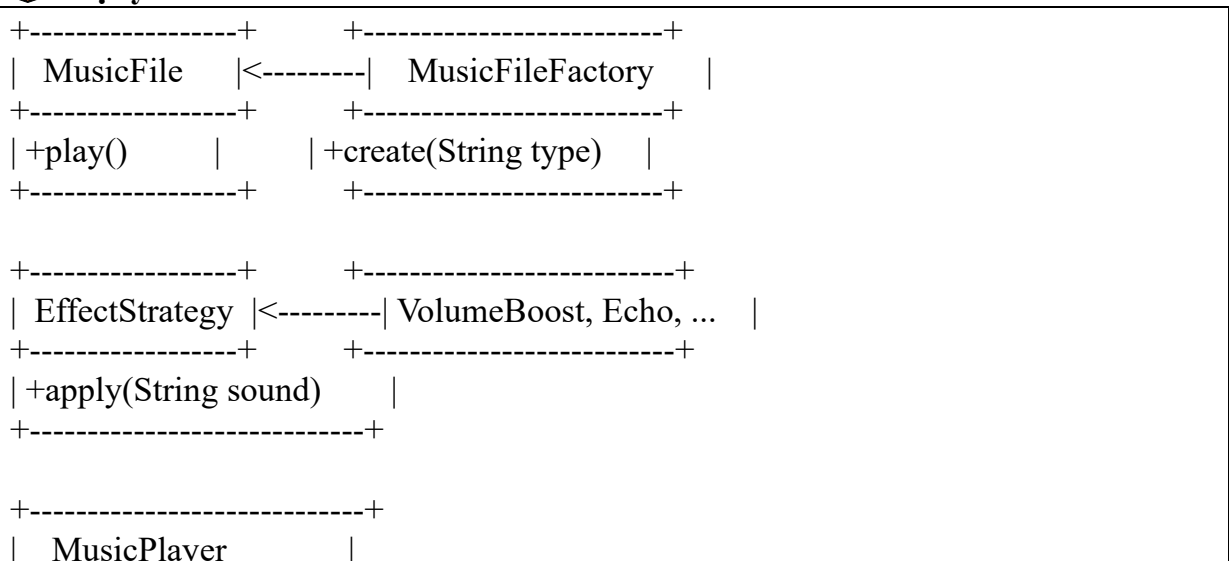
Hãy xây dựng một **Music Player Console App** đơn giản cho phép người dùng:

- Chọn định dạng file nhạc: MP3, WAV, FLAC
- Áp dụng các hiệu ứng: tăng âm lượng, echo, bass boost (có thể thay đổi trong lúc chạy)
- Phát bài nhạc đã chọn kèm theo hiệu ứng

🔧 Yêu cầu kỹ thuật:

- **Factory Pattern** để tạo đối tượng MusicFile phù hợp với định dạng (MP3/WAV/FLAC)
- **Strategy Pattern** để áp dụng các hiệu ứng (VolumeBoost, Echo, BassBoost...)

📦 Gợi ý cấu trúc:



```

+-----+
| - MusicFile          |
| - EffectStrategy     |
| +playWithEffect()    |
+-----+

```

Hướng dẫn triển khai:

- Mỗi file nhạc (MP3, WAV...) cài đặt interface MusicFile và method play()
- Mỗi hiệu ứng là một EffectStrategy → được gán vào MusicPlayer
- MusicPlayer có thể thay đổi hiệu ứng tại runtime

Mở rộng thêm (tuỳ chọn):

- Cho phép chọn hiệu ứng từ bàn phím
- Ghi log mỗi lần phát nhạc (gợi ý dùng Singleton Logger)
- Kết hợp với Builder để cấu hình playlist

Dự án Mini 2 (mức TRUNG): Hệ thống đăng ký khóa học OOP

Đề bài:

Xây dựng hệ thống đăng ký khóa học gồm các chức năng:

- Hiển thị danh sách các khóa học: Java, Python, C++
- Mỗi khóa học có cấu hình riêng (thời lượng, cấp độ, học phí)
- Học viên có thể chọn khóa học, chọn phương thức thanh toán (mỗi cách có cách tính phí khác nhau)
- In hóa đơn cuối cùng với đầy đủ thông tin

Yêu cầu sử dụng Design Pattern:

- **Factory Pattern:** để khởi tạo khóa học phù hợp
- **Builder Pattern:** để cấu hình chi tiết khóa học (tuỳ chọn: online/offline, ghi chú, hỗ trợ mentor...)
- **Strategy Pattern:** để tính phí theo phương thức thanh toán
 - Ví dụ: Trả thẳng, Trả góp, Trả qua ví điện tử (mỗi cái có phí khác nhau)

Gợi ý cấu trúc UML:

```

+-----+      +-----+
| Course      |<-----| CourseFactory      |
+-----+      +-----+
| +getFee()   |      | +create(type)        |
+-----+      +-----+

+-----+      +-----+
| CourseBuilder |----->| OnlineCourse, ... |
+-----+      +-----+

+-----+      +-----+
| PaymentStrategy |<-----| CreditCard, Installment...|
+-----+      +-----+

```

```

+-----+      +-----+
| +calculateFee() |
+-----+

+-----+
| StudentRegistration |
+-----+
| - Course           |
| - PaymentStrategy  |
| +generateInvoice() |
+-----+

```

Hướng dẫn triển khai:

- Factory chọn course theo tên ("Java", "Python")
- Builder thêm option: withMentor(), online(), note()...
- PaymentStrategy: calculateFee(baseFee) → cộng thêm phí tương ứng
- Hóa đơn thể hiện đầy đủ thông tin và giá cuối cùng

Gợi ý mở rộng:

- Cho phép lưu danh sách học viên đăng ký
- Thêm Template Method để định nghĩa khung xuất hóa đơn (PDF, HTML)
- Giao diện console để chọn khóa học, thanh toán