1. **FactoryMethod**

**1. Ứng dụng thực tế trong hệ thống thanh toán**

* Khi bạn có nhiều phương thức thanh toán (thẻ tín dụng, PayPal, Momo, VNPay), bạn có thể sử dụng Factory Method để tạo các đối tượng xử lý thanh toán phù hợp.

2. **Quản lý Logger trong ứng dụng**

* Khi ứng dụng cần hỗ trợ nhiều loại logger (ConsoleLogger, FileLogger, DatabaseLogger), Factory Method giúp tạo logger phù hợp.

3. **Tạo đối tượng Database Connection**

* Khi bạn muốn kết nối với nhiều loại cơ sở dữ liệu như MySQL, PostgreSQL, MongoDB.

4. **Ứng dụng trong hệ thống gửi thông báo (Notification System)**

* Khi bạn cần gửi thông báo bằng Email, SMS hoặc Push Notification.

**Khi nào nên sử dụng Factory Method?**

* Khi hệ thống có nhiều lớp con với cách tạo đối tượng khác nhau.
* Khi muốn giấu logic khởi tạo đối tượng khỏi người dùng.
* Khi cần tạo đối tượng mà không làm thay đổi code của lớp client.

1. **Singleton**
2. **Quản lý kết nối cơ sở dữ liệu (Database Connection)**

* Đảm bảo chỉ có một kết nối tới cơ sở dữ liệu duy nhất trong toàn bộ ứng dụng, tránh tạo nhiều kết nối dư thừa gây tốn tài nguyên.

1. **Quản lý cấu hình hệ thống (Configuration)**

* Trong ứng dụng, có thể có một lớp lưu trữ thông tin cấu hình như API keys, database URLs, settings,... chỉ cần tải một lần duy nhất.

1. Quản lý Logger (Ghi log hệ thống)

* Đảm bảo rằng tất cả các phần của ứng dụng đều sử dụng cùng một logger duy nhất để ghi log.

1. Quản lý Bộ nhớ Cache

* Một ứng dụng có thể có một lớp lưu trữ dữ liệu cache (ví dụ: dữ liệu tạm thời từ database) để truy xuất nhanh hơn.

1. **Quản lý Luồng xử lý (Thread Pool)**

* Trong các ứng dụng đa luồng, có thể sử dụng Singleton để đảm bảo chỉ có một Thread Pool duy nhất.
* **🔥 Khi nào nên sử dụng Singleton?**
* ✔ Khi một lớp cần có **duy nhất một thể hiện** trong suốt vòng đời ứng dụng.  
  ✔ Khi muốn **tiết kiệm bộ nhớ** bằng cách sử dụng chung một đối tượng thay vì tạo nhiều đối tượng mới.  
  ✔ Khi muốn **đảm bảo tính nhất quán** trong quản lý dữ liệu hoặc tài nguyên (vd: database connection, logger).
* **🚨 Khi nào KHÔNG nên sử dụng Singleton?**
* ❌ Khi ứng dụng cần mở rộng dễ dàng, vì Singleton có thể gây khó khăn khi viết test và mở rộng code.  
  ❌ Khi cần sử dụng đa luồng, Singleton có thể gây lỗi nếu không được thiết kế thread-safe.

1. **State Pattern - Mẫu thiết kế trạng thái**

State Pattern là một mẫu thiết kế hành vi (Behavioral Pattern) cho phép một đối tượng thay đổi hành vi của nó khi trạng thái bên trong thay đổi. Điều này giúp mã dễ mở rộng và tránh việc sử dụng quá nhiều điều kiện if-else hoặc switch-case.

**📌 Các trường hợp nên sử dụng State Pattern**

* Khi một đối tượng có nhiều trạng thái khác nhau và mỗi trạng thái có hành vi riêng biệt.
* Khi muốn tránh sử dụng quá nhiều if-else hoặc switch-case trong mã nguồn.
* Khi trạng thái của đối tượng thay đổi theo thời gian và các hành vi tương ứng cũng thay đổi.

**1️. Ví dụ: Máy bán hàng tự động (Vending Machine)**

**Bài toán:**

Một máy bán hàng có các trạng thái:

* **Không có tiền (No Coin)**
* **Có tiền (Has Coin)**
* **Đang bán hàng (Dispensing)**
* **Hết hàng (Out of Stock)**

Mỗi trạng thái sẽ có các hành vi khác nhau khi người dùng thao tác.

2. **Ví dụ: Trạng thái tài khoản ngân hàng**

**Bài toán:**

Một tài khoản ngân hàng có thể có 3 trạng thái:

* **Tài khoản hoạt động (Active)**
* **Bị đóng băng (Frozen)**
* **Đóng tài khoản (Closed)**

Mỗi trạng thái có quy tắc riêng khi gửi tiền, rút tiền.

1. **Strategy Pattern - Mẫu thiết kế chiến lược**

Strategy Pattern là một mẫu thiết kế **hành vi (Behavioral Pattern)**, cho phép chọn thuật toán hoặc phương thức thực thi tại **thời điểm chạy (runtime)** mà không cần sửa đổi mã nguồn chính.

📌 **Khi nào nên sử dụng Strategy Pattern?**  
✔ Khi có nhiều thuật toán hoặc cách xử lý khác nhau cho cùng một tác vụ.  
✔ Khi muốn **giảm số lượng if-else hoặc switch-case** trong mã nguồn.  
✔ Khi muốn mở rộng hoặc thay đổi thuật toán dễ dàng mà **không ảnh hưởng đến phần còn lại của hệ thống**.

**1️. Ví dụ: Tính toán phí vận chuyển**

**Bài toán**

Một công ty vận chuyển có nhiều cách tính phí khác nhau:

* **Giao hàng tiêu chuẩn (Standard Shipping)**
* **Giao hàng nhanh (Express Shipping)**
* **Giao hàng siêu tốc (Same-Day Delivery)**

**2. Ví dụ: Thanh toán (Payment)**

**Bài toán**

**Một ứng dụng thương mại điện tử hỗ trợ nhiều phương thức thanh toán:**

* **Thanh toán bằng thẻ tín dụng**
* **Thanh toán bằng ví điện tử (PayPal, MoMo, ZaloPay)**
* **Thanh toán bằng tiền mặt khi nhận hàng (COD)**

**3. Ví dụ: Sắp xếp danh sách (Sorting)**

**Bài toán**

**Một ứng dụng hỗ trợ nhiều cách sắp xếp danh sách:**

* **Sắp xếp theo Bubble Sort**
* **Sắp xếp theo Quick Sort**
* **Sắp xếp theo Merge Sort**

1. **Decorator Pattern - Mẫu thiết kế trang trí**

**Decorator Pattern** là một mẫu thiết kế thuộc nhóm **cấu trúc (Structural Pattern)**, cho phép **thêm hành vi mới vào một đối tượng mà không làm thay đổi mã nguồn của nó**.

**📌 Khi nào nên sử dụng Decorator Pattern?**

✔ Khi muốn mở rộng tính năng cho một đối tượng mà **không cần sửa đổi mã nguồn gốc**.  
✔ Khi cần **thêm/xóa linh hoạt** các chức năng trong runtime.  
✔ Khi muốn **tránh kế thừa lớp (class inheritance)** quá sâu.

**1️. Ví dụ: Trang trí đồ uống (Beverage)**

**Bài toán**

Một quán cà phê bán nhiều loại đồ uống như:

* Cà phê đen
* Cà phê sữa
* Cà phê thêm topping (đường, sữa, trân châu, v.v.)

Nếu dùng kế thừa (extends), ta sẽ có nhiều lớp như BlackCoffeeWithMilk, BlackCoffeeWithSugar, BlackCoffeeWithMilkAndSugar, … **gây rắc rối khi mở rộng**.  
Thay vào đó, ta dùng **Decorator Pattern** để linh hoạt thêm các topping.

2. **Ví dụ: Thêm chức năng cho Notifier (Thông báo)**

**Bài toán**

Hệ thống gửi thông báo **có thể nâng cấp linh hoạt**:

* Gửi **qua email**
* Gửi **qua SMS**
* Gửi **qua Facebook Messenger**

Người dùng có thể nhận thông báo **qua nhiều kênh cùng lúc**.

3. **Ví dụ: Thêm chức năng bảo mật cho file**

**Bài toán**

Ta có một hệ thống **lưu file**, cần thêm các tính năng như:

* **Mã hóa file**
* **Nén file**
* **Ghi log khi lưu file**

1. **Observer Pattern - Mẫu thiết kế quan sát**

**Observer Pattern** là một mẫu thiết kế thuộc nhóm **hành vi (Behavioral Pattern)**, cho phép **một đối tượng (Subject) thông báo đến nhiều đối tượng khác (Observers) khi có sự thay đổi trạng thái**.

**📌 Khi nào nên sử dụng Observer Pattern?**

✔ Khi **một đối tượng thay đổi trạng thái**, các đối tượng liên quan cần được thông báo **tự động**.  
✔ Khi **có nhiều đối tượng quan sát**, nhưng không muốn chúng bị ràng buộc chặt chẽ vào đối tượng chủ.  
✔ Khi cần **giảm sự phụ thuộc giữa các thành phần trong hệ thống** (**loose coupling**).

**1️. Ví dụ: Hệ thống thông báo thời tiết**

**Bài toán**

Hệ thống dự báo thời tiết cần cập nhật thông tin **nhiệt độ, độ ẩm, áp suất** và thông báo đến nhiều **bảng hiển thị** khác nhau như:

* **Màn hình hiển thị hiện tại**
* **Màn hình thống kê**
* **Màn hình dự báo**

Khi nhiệt độ thay đổi, tất cả màn hình phải cập nhật **tự động**.

2. **Ví dụ: Hệ thống thông báo khi có đơn hàng mới**

**Bài toán**

Trong hệ thống bán hàng online, khi có đơn hàng mới, hệ thống cần thông báo đến:

* **Bộ phận kho hàng**
* **Bộ phận giao hàng**
* **Khách hàng**

Mỗi đối tượng có thể tự động nhận thông báo khi đơn hàng được tạo.

1. **Abstract Factory Pattern - Mẫu thiết kế nhà máy trừu tượng**

**Abstract Factory** là một mẫu thiết kế thuộc nhóm **Creational Pattern (mẫu khởi tạo)**, giúp **tạo ra một nhóm đối tượng liên quan mà không cần chỉ rõ lớp cụ thể của chúng**.

**📌 Khi nào nên sử dụng Abstract Factory?**

✔ Khi hệ thống cần tạo ra **một nhóm đối tượng liên quan với nhau**.  
✔ Khi muốn **giữ tính đóng mở (Open/Closed Principle)** để **dễ dàng mở rộng mà không sửa đổi code hiện có**.  
✔ Khi muốn **giảm sự phụ thuộc giữa các lớp**, giúp code linh hoạt hơn.

**1️. Ví dụ: Hệ thống giao diện UI đa nền tảng (Windows vs MacOS)**

**Bài toán**

Ứng dụng cần hỗ trợ **giao diện cho cả Windows và MacOS**, mỗi hệ điều hành có **các thành phần UI riêng biệt**:

* **Windows**: Button & Checkbox có phong cách Windows.
* **MacOS**: Button & Checkbox có phong cách MacOS.

Ứng dụng cần **tạo các thành phần UI phù hợp** mà không cần kiểm tra hệ điều hành trực tiếp.

2. **Ví dụ: Hệ thống quản lý cơ sở dữ liệu (SQL vs NoSQL)**

**Bài toán**

Ứng dụng cần hỗ trợ **nhiều loại cơ sở dữ liệu** (SQL & NoSQL), mỗi loại có cách thao tác khác nhau:

* **SQL Database** (MySQL, PostgreSQL) sử dụng **câu lệnh SQL**.
* **NoSQL Database** (MongoDB) sử dụng **document-based API**.

Ứng dụng cần **tạo kết nối phù hợp** mà không cần kiểm tra loại database trực tiếp.