

# Design Patterns trong Phát triển Phần mềm

Phân tích và vận dụng các  
Behavioral, Structural và Creational Patterns

## Nhóm 11

23127006 – Trần Nguyễn Khải Luân

23127179 – Nguyễn Bảo Duy

23127404 – Lê Tuấn Lộc

Trường Đại học Khoa học Tự nhiên, ĐHQG-HCM  
*Học phần:* Thiết kế phần mềm

Ngày 14 tháng 2 năm 2026

# Nội dung trình bày

- 1 Tổng quan về Design Patterns
- 2 Factory Method (Creational)
- 3 Abstract Factory (Creational)
- 4 Decorator (Structural)
- 5 Adapter (Structural)
- 6 Observer (Behavioral)
- 7 Strategy (Behavioral)
- 8 Kết luận

# Khi không có Design Patterns...

- **Tight coupling** — thay đổi 1 class ảnh hưởng hàng loạt class khác
- **Code duplication** — cùng logic lặp lại ở nhiều nơi
- **Khó mở rộng** — thêm feature mới = viết lại cả module
- **Bảo trì tốn kém** — fix bug chỗ này, sinh bug chỗ khác
- **Giao tiếp khó khăn** — mỗi dev giải quyết cùng bài toán theo cách khác nhau

# Hậu quả trong thực tế

## Dự án phần mềm thất bại vì...

- Hệ thống “cứng” — không thích ứng khi requirement thay đổi
- Technical debt tích lũy — càng sửa càng rối
- Onboarding chậm — dev mới không hiểu kiến trúc
- Refactor = viết lại từ đầu — vì code quá phụ thuộc lẫn nhau

⇒ *Những vấn đề này đã được giải quyết từ lâu...*

# Design Patterns giải quyết những gì?

## Giải pháp đã được kiểm chứng

- **Giảm coupling** — các thành phần giao tiếp qua abstraction
- **Tăng extensibility** — thêm mới mà không sửa code cũ (OCP)
- **Tái sử dụng** — giải pháp template, áp dụng cho nhiều bài toán
- **Chuẩn hóa giao tiếp** — “dùng Observer pattern” > giải thích 20 dòng

*Không phải code cụ thể — mà là cách tư duy thiết kế.*

# Ba nhóm Design Patterns

## Creational

Tạo object

Kiểm soát cách khởi tạo đối tượng, giảm dependency vào concrete class

## Structural

Tổ chức cấu trúc

Kết hợp objects/classes thành cấu trúc lớn hơn mà vẫn linh hoạt

## Behavioral

Phân phối hành vi

Quản lý giao tiếp và trách nhiệm giữa các objects

# Lưu ý khi sử dụng Design Patterns

## Dùng đúng lúc

- Pattern không phải “silver bullet” — dùng sai = over-engineering
- Chỉ áp dụng khi gặp vấn đề cụ thể mà pattern giải quyết
- Pattern ≠ Framework — pattern là ý tưởng, framework là implementation

## Tránh lạm dụng

- KISS — Keep It Simple, Stupid
- YAGNI — You Aren't Gonna Need It
- Đừng dùng pattern chỉ vì “nghe hay”

# Factory Method

Creational Pattern

# Factory Method: Bối cảnh thực tế

## Tình huống phổ biến:

- Hệ thống cần tạo đối tượng (Product) nhưng chưa biết chính xác loại cụ thể lúc thiết kế.
- Việc khởi tạo phụ thuộc vào cấu hình hoặc logic tại runtime.

## Ví dụ thực tế:

- **Logistics**: Ban đầu chỉ có *Truck*, sau đó cần mở rộng thêm *Ship*, *Train*.
- **UI Framework**: Nút bấm thay đổi theo hệ điều hành.
- **Converters**: Xuất file sang PDF, Word, hoặc HTML.

**Mẫu chốt:** Phụ thuộc vào **Concrete Class** làm hệ thống bị "cứng", khó mở rộng mà không sửa code cũ.

# Factory Method: Nỗi đau khi thiếu Pattern

## Cách làm “Ngây thơ”:

Dùng new trực tiếp và switch-case

```
class Logistics {  
    void planDelivery(String type) {  
        Transport t;  
        if (type == "Truck")  
            t = new Truck();  
        else if (type == "Ship")  
            t = new Ship();  
        t.deliver();  
    }  
}
```

## Hậu quả:

- **Tight Coupling**: Phụ thuộc trực tiếp vào class cụ thể.
- **Vi phạm OCP**: Thêm phương tiện mới = Sửa logic cốt lõi.
- **Khó Unit Test**: Không thể mock các class bị "hard-code".

## Virtualize Object Creation

Ý tưởng then chốt:

1. **Đóng gói sự biến thiên:** Tách biệt logic *sử dụng* và *tạo đối tượng*.
2. **Ủy quyền cho Subclass:** Class con quyết định *Product* nào sẽ được tạo.
3. **Lập trình với Interface:** Client không quan tâm Concrete class là gì.

Nguyên lý thiết kế:

- **DIP:** Phụ thuộc vào abstraction, không phụ thuộc vào concrete class.
- **Encapsulation of Variation:** Biến thiên về loại sản phẩm được khóa sau Factory Method.

# Factory Method: Cách hoạt động

## Các vai trò trong pattern:

- **Product**: Interface quy định các khả năng của sản phẩm.
- **Creator**: Lớp chứa logic nghiệp vụ, khai báo Factory Method.
- **ConcreteCreator**: Thực thi Factory Method để trả về sản phẩm cụ thể.

## Flow thực thi:

```
// Client chỉ làm việc với lớp trừu tượng
Logistics app = new SeaLogistics();
app.planDelivery();
// Inside planDelivery:
//   Transport t = createTransport(); // Lấy "Ship" tại runtime
//   t.deliver();
```

**Kết quả:** Logic nghiệp vụ không hề biết mình đang dùng *Truck* hay *Ship*.

# Factory Method: Được và Mất

## Được gì:

- **Loose Coupling:** Tránh phụ thuộc cứng.
- **Tuân thủ SRP:** Gom logic tạo object về một nơi.
- **Tuân thủ OCP:** Mở rộng dễ dàng.

## Mất gì:

- **Class Explosion:** Tăng số lượng class con.
- **Phức tạp hóa:** Cần nhiều lớp trung gian hơn.

**Lưu ý:** Phù hợp khi logic tạo đối tượng phức tạp và cần mở rộng thường xuyên.

# Factory Method: Khi nào dùng? Khi nào là thừa?

## Nên dùng khi:

- Cần cho phép người dùng framework mở rộng thành phần nội bộ.
- Muốn tái sử dụng đối tượng thay vì tạo mới liên tục (caching).

## Over-engineering khi:

- Số lượng sản phẩm là **cố định** và rất ít
- Việc tạo object quá đơn giản, không có kế hoạch mở rộng.

**Câu hỏi:** Nếu loại bỏ pattern này, business rule nào sẽ bị phá đầu tiên?

# Abstract Factory

Creational Pattern

# Abstract Factory: Bối cảnh thực tế

## Tình huống phổ biến:

- Hệ thống cần tạo ra các **họ sản phẩm liên quan**.
- Các sản phẩm phải đảm bảo tính **tương thích** tuyệt đối.

## Ví dụ thực tế:

- Furniture Shop: Ghế, bàn, sofa theo bộ (*Modern, Victorian*).
- Cross-Platform UI: Toolkit (Button, Input) đồng bộ theo HĐH.
- Game Themes: Bộ quái vật, vũ khí theo chủ đề *Fantasy*.

**Mâu chốt:** Tránh "râu ông nọ cắm cằm bà kia"(vd: ghế *Modern* trong bộ bàn *Victorian*).

# Abstract Factory: Nỗi đau khi thiếu Pattern

## Cách làm “Thủ công”:

Kiểm tra điều kiện ở mọi nơi

```
// Code rải rác khắp nơi
if (style == "Modern") {
    chair = new ModernChair();
    sofa = new ModernSofa();
} else if (style == "Victorian") {
    chair = new VictorianChair();
    sofa = new VictorianSofa();
}
```

## Hậu quả:

- **Inconsistency**: Dễ chọn nhầm sản phẩm không cùng họ.
- **Rigidity**: Thêm một phong cách mới yêu cầu sửa 'if/else' ở tất cả các module.
- **Tight Coupling**: Client phụ thuộc vào hàng tá concrete classes của sản phẩm.

# Abstract Factory: Insight cốt lõi

## Factory of Factories

### Ý tưởng then chốt:

1. **Interface họ:** Một đối tượng chứa nhiều Factory Methods cho cả họ sản phẩm.
2. **Composition:** Client ủy quyền việc tạo cho đối tượng Factory.
3. **Invariant:** Cam kết sản phẩm tạo ra luôn cùng biến thể.

### Nguyên lý thiết kế:

- **DIP:** Chỉ làm việc với Abstract interfaces.
- **OCP:** Thêm họ sản phẩm mới mà không sửa Client.

# Abstract Factory: Cách hoạt động

## Các vai trò trong pattern:

- **Abstract Factory**: Khai báo các phương thức tạo từng loại sản phẩm trừu tượng.
- **Concrete Factory**: Triển khai việc tạo các sản phẩm cụ thể cho một biến thể duy nhất.
- **Abstract Products**: Các interface cho từng loại sản phẩm trong họ.

## Flow thực thi:

1. **Cấu hình**: Lúc khởi tạo, app chọn một Concrete Factory (vd: ModernFactory).
2. **Truyền Factory**: Client nhận factory này thông qua constructor/setter.
3. **Tạo bộ sản phẩm**: Client gọi factory.createChair() và factory.createSofa().

# Abstract Factory: Được và Mất

## Được gì:

- **Consistency:** Đảm bảo các sản phẩm luôn đi kèm đúng bộ.
- **Decoupling:** Client và Concrete Products không biết nhau.
- **SRP:** Logic tạo các họ sản phẩm được tập trung hóa.

## Mất gì:

- **Complexity:** Quá nhiều interface và class mới làm code trở nên cồng kềnh.
- **Hard to Extend:** Thêm một loại sản phẩm mới (vd: CoffeeTable) vào họ yêu cầu sửa interface Factory gốc.

**Lưu ý:** Thường tiến hóa từ **Factory Method** khi số lượng họ sản phẩm bắt đầu tăng lên.

# Factory Method vs Abstract Factory

**Giống nhau:** Creational Pattern ■ Ẩn việc new khỏi client ■ Tuân theo OCP

## Khác nhau cốt lõi

	Factory Method	Abstract Factory
Tạo gì?	Một Product	Một họ Product
Mức độ	Class override	Object composition
Mục tiêu	Chọn implementation	Chọn ecosystem
Scale	Nhỏ, đơn giản	Nhiều product đi cùng
Ví dụ	1 loại Document	Cả bộ UI Windows/Mac

## Tóm tắt

Factory Method → tạo **một** sản phẩm

Abstract Factory → tạo **cả một hệ sinh thái** sản phẩm.

# Abstract Factory: Khi nào dùng?

## Nên dùng khi:

- Code cần làm việc với nhiều họ sản phẩm mà không muốn phụ thuộc vào class cụ thể.
- Muốn cưỡng chế việc sử dụng đồng bộ các đối tượng liên quan.

## Tránh dùng khi:

- Các lớp sản phẩm cụ thể **không bao giờ thay đổi** hoặc chỉ có một biến thể duy nhất.
- Bài toán đơn giản, việc tách class làm tăng Cognitive Load vô ích.

**Checklist Kiến trúc:** Liệu chi phí vận hành (Cognitive Load) có tương xứng với tính an toàn (Invariants) mà họ sản phẩm này mang lại?

# Decorator

Structural Pattern

# Decorator: Bối cảnh thực tế

## Tình huống phổ biến:

- Bạn có một đối tượng cơ bản đã hoạt động tốt
- Yêu cầu mới: *thêm hành vi bổ sung* mà **không sửa** code cũ
- Các hành vi bổ sung có thể **kết hợp tự do** với nhau

## Ví dụ thực tế:

- **I/O Streams**: đọc file → thêm buffer → thêm mã hoá → thêm nén
- **HTTP Middleware**: request → logging → auth → rate-limit
- **UI Components**: text field → scroll bar → border → shadow

**Điểm chung:** Số lượng tổ hợp tính năng **tăng theo cấp số nhân**, nhưng mỗi tính năng lại **độc lập** về logic.

# Decorator: Nỗi đau khi không có pattern

## Cách tiếp cận “bình thường”:

kế thừa cho mọi tổ hợp

```
class Coffee { ... }  
class CoffeeWithMilk  
    extends Coffee  
class CoffeeWithSugar  
    extends Coffee  
class CoffeeWithMilkAndSugar  
    extends Coffee  
class CoffeeWithMilkAndWhip  
    extends Coffee  
// ... 2^n subclasses
```

## Hậu quả:

- **Subclass Explosion**:  $n$  topping  $\Rightarrow 2^n$  class
- **Vi phạm OCP**: thêm topping = sửa hierarchy
- **Code duplication**: logic lặp lại nhiều class
- **Rigid design**: không đổi tổ hợp lúc runtime

# Decorator: Các giải pháp thay thế

## Giải pháp 1: Dùng boolean flags **Vấn đề:**

```
class Coffee {  
    bool hasMilk, hasSugar;  
  
    double getCost() {  
        double cost = baseCost;  
        if (hasMilk)    cost += 5;  
        if (hasSugar)   cost += 2;  
        // ... grows forever  
        return cost;  
    }  
}
```

- God class — biết quá nhiều
- Thêm topping = sửa class (**vi phạm OCP**)
- Không duplicate topping (2 shot milk)

## Giải pháp 2: Strategy?

- Không hỗ trợ stacking và thứ tự

## Composition over Inheritance

Ý tưởng then chốt:

1. Thay vì *tạo subclass* cho mỗi tổ hợp, ta **wrap** đối tượng gốc bằng các lớp bổ sung hành vi
2. Mỗi wrapper (decorator) **cùng interface** với đối tượng gốc  
⇒ client không cần biết đang dùng bao nhiêu lớp wrap
3. Decorator có thể **stack lên nhau** tự do, thay đổi lúc runtime

Nguyên lý thiết kế:

- **Open/Closed Principle:** mở rộng hành vi mà **không sửa** code cũ
- **Single Responsibility:** mỗi decorator chỉ lo **một** hành vi
- **Transparent wrapping:** decorator là “*invisible*” với client code



# Decorator: Cách hoạt động

## Các vai trò trong pattern:

- **Component** (interface): contract chung cho tất cả
- **ConcreteComponent**: đối tượng gốc cần mở rộng
- **BaseDecorator**: giữ reference đến component, delegate call
- **ConcreteDecorator**: thêm hành vi cụ thể trước/sau delegate

## Flow thực thi:

```
Component coffee = new SimpleCoffee();           // cost = 10
coffee = new MilkDecorator(coffee);              // +5 → 15
coffee = new SugarDecorator(coffee);             // +2 → 17
coffee.getCost();    // → SugarDecorator.getCost()
                    //   → MilkDecorator.getCost()
                    //     → SimpleCoffee.getCost() = 10
                    //   return 10 + 5
                    // return 15 + 2 = 17
```

Kết quả: Thêm topping = tạo class mới, **zero changes** vào code cũ.

# Decorator: Trade-offs

## Được gì:

- Mở rộng hành vi **không cần sửa** code cũ
- Kết hợp hành vi **linh hoạt** lúc runtime
- Tuân thủ SRP — mỗi class một nhiệm vụ
- Thay thế kế thừa phức tạp bằng composition

## Mất gì:

- **Nhiều small classes** — khó navigate codebase
- **Ordering matters**: thứ tự wrap ảnh hưởng kết quả (Encrypt → Compress ≠ Compress → Encrypt)
- **Khó debug**: stack trace dài, nhiều lớp indirection
- **Unwrapping**: khó lấy lại object gốc bên trong

**Lưu ý:** Decorator phù hợp khi số *loại* hành vi nhiều, nhưng mỗi hành vi *đơn giản* và *độc lập*.

# Decorator: Khi nào dùng? Khi nào là thừa?

## Dùng khi:

- **Không được sửa** class gốc (library, legacy code)
- Tổ hợp hành vi **lớn** và thay đổi thường xuyên
- Cần thay đổi **lúc runtime**

## Over-engineering khi:

- Chỉ **1–2 biến thể cố định** — kế thừa đơn giản hơn
- Hành vi **không thể tách rời** khỏi object gốc
- Team nhỏ, domain đơn giản — thêm cognitive load
- Wrap để “**đúng pattern**” mà không cần mở rộng

## Dấu hiệu over-design:

“Chỉ có 1 decorator và không có kế hoạch thêm.”

# **Adapter**

Structural Pattern

# Adapter: Bối cảnh thực tế

## Vấn đề:

- Hệ thống (*Client*) đang chạy ổn định.
- Cần tích hợp module mới (*Service*) nhưng **lệch Interface**.
- **Ràng buộc:** Không thể sửa code của Service (do là thư viện đóng gói hoặc code cũ rủi ro cao).

## Ví dụ dễ hiểu:

- **Du lịch:** Ở cắm khách sạn 3 chấu (UK) ↔ Phích cắm máy sấy 2 chấu (VN).
- **Dữ liệu:** App đọc XML ↔ API chứng khoán trả về JSON.
- **Cổng sạc:** Điện thoại cổng Lightning ↔ Dây cáp USB-C.

**Mẫu chốt:** Hai bên muốn nói chuyện nhưng **bất đồng ngôn ngữ**.

# Adapter: Code xấu khi thiếu Pattern

**Cách làm “cưỡng ép”:** Sửa Client để chiều lòng từng thư viện.

```
void processPayment(Object gw) {  
    if (gw instanceof Paypal) {  
        // API cũ  
        ((Paypal)gw).sendMoney(100);  
    }  
    else if (gw instanceof Stripe) {  
        // API mới khác tên hàm!  
        ((Stripe)gw).makeCharge(100);  
    }  
}
```

**Hậu quả:**

- **Code Rác:** Logic nghiệp vụ lẫn lộn với logic chuyển đổi.
- **Vi phạm OCP:** Thêm công thanh toán mới → Phải sửa ‘if/else’ trong Client.
- **Khó bảo trì:** Client dính chặt (tightly coupled) vào từng thư viện cụ thể.

# Adapter: Tại sao không sửa trực tiếp?

## Cách 1: Sửa code Thư viện (Service)

- **Bất khả thi:** Thường ta chỉ có file '.dll' hoặc '.jar' (mã đóng).
- **Rủi ro:** Nếu thư viện update phiên bản mới, code sửa sẽ mất sạch.

## Cách 2: Sửa code Client

- **Nguy hiểm:** Hệ thống hiện tại quá lớn, sửa Interface chuẩn sẽ làm lỗi dây chuyền (breaking changes).
- **Sai nguyên tắc:** Không nên sửa code đang chạy tốt (Open/Closed).

⇒ Cần một **người phiên dịch** đứng giữa.

## Wrapper / Translator

### Cơ chế hoạt động:

1. Tạo class **Adapter** đóng vai trò trung gian.
2. Adapter **giả dạng** làm Interface mà Client mong muốn.
3. Bên trong, Adapter **dịch** lời gọi đó sang ngôn ngữ của Service.

### Lợi ích:

- **Client**: Không biết mình đang dùng thư viện lạ.
- **Service**: Không biết mình đang được gọi bởi Client.
- **Kết nối**: Hai bên khớp nhau mà không cần sửa code bên nào.

# Adapter: Ví dụ minh họa

**Bài toán:** Client muốn sạc cổng MicroUSB, nhưng chỉ có dây Lightning.

```
// 1. Target (Cái Client cần)
interface MicroUsbPhone { void recharge(); }

// 2. Adaptee (Cái ta đang có)
class iPhoneLightning {
    void useLightning() { print("Sạc bằng Lightning..."); }
}

// 3. Adapter (Đầu chuyển đổi)
class LightningToMicroAdapter implements MicroUsbPhone {
    private iPhoneLightning iphone;
    public LightningToMicroAdapter(iPhoneLightning ip) {
        this.iphone = ip;
    }
    @Override
    public void recharge() {
        // Chuyển đổi lời gọi
        this.iphone.useLightning();
    }
}
```

Client gọi `recharge()`, Adapter lén gọi `useLightning()`.

# Adapter: Được và Mất

## Ưu điểm:

- **Tách biệt (Decouple):** Client và Thư viện không phụ thuộc nhau.
- **Tái sử dụng:** Tận dụng được code cũ mà không cần viết lại.
- **Linh hoạt:** Muốn đổi thư viện khác? Chỉ cần viết Adapter mới.

## Nhược điểm:

- **Phức tạp hóa:** Thêm nhiều class mới (Adapter) cho những việc đơn giản.
- **Hiệu năng:** Tốn thêm một bước gọi hàm trung gian (không đáng kể với app thường).

**Lưu ý:** Nếu bạn có quyền sửa code cả 2 bên, hãy sửa trực tiếp (Refactor) thay vì dùng Adapter.

# Adapter: Khi nào dùng?

## Nên dùng:

- Tích hợp thư viện **3rd-party** hoặc API ngoài.
- Làm việc với code cũ (**Legacy**) không thể chỉnh sửa.
- Muốn thống nhất nhiều class có interface khác nhau về một chuẩn chung.

## Không nên dùng (Over-kill):

- Code nội bộ team tự viết (hay sửa interface cho khớp luôn).
- Interface hai bên chỉ khác nhau tên hàm một chút (sửa luôn cho nhanh).

**Dấu hiệu nhận biết:** “Tôi thích thư viện này, nhưng Interface của nó lạ quá, không gắn vào App được.”

# Observer

Behavioral Pattern

# Observer: Bối cảnh thực tế

## Tình huống phổ biến:

- Một đối tượng **thay đổi trạng thái**, và **nhiều đối tượng khác cần biết**
- Danh sách “người cần biết” **không cố định** — có thể thêm/bớt lúc runtime
- Nguồn dữ liệu **không nên biết** chi tiết về từng nơi tiêu thụ

## Ví dụ thực tế:

- **Event system**: user click → update UI + log analytics + validate form
- **Data binding**: model thay đổi → tự động render lại view
- **Stock ticker**: giá cổ phiếu thay đổi → thông báo nhiều dashboard
- **Message broker**: producer gửi event → nhiều consumer xử lý

**Điểm chung:** Quan hệ **one-to-many** giữa nguồn dữ liệu và nơi tiêu thụ, với danh sách consumer **thay đổi liên tục**.



# Observer: Nỗi đau khi không có pattern

## Cách tiếp cận “bình thường”:

gọi trực tiếp

```
class WeatherStation {  
    PhoneDisplay phone;  
    WebDashboard web;  
    Logger logger;  
  
    void updateTemp(double t) {  
        this.temp = t;  
        phone.refresh(t);  
        web.render(t);  
        logger.log(t);  
        // thêm consumer mới?  
        // → sửa class này!  
    }  
}
```

## Hậu quả:

- **Tight coupling:** WeatherStation phụ thuộc **mọi** consumer
- **Vi phạm OCP:** thêm consumer = sửa source
- **Không linh hoạt:** không subscribe/unsubscribe lúc runtime
- **Khó test:** phải mock tất cả dependency
- **God object:** source biết quá nhiều

# Observer: Các giải pháp thay thế

## Giải pháp 1: Polling — consumer tự hỏi source liên tục

- Consumer gọi `station.getTemp()` mỗi  $n$  giây
- **Lãng phí tài nguyên:** hầu hết lần check không có gì mới
- **Delay:** data mới có thể bị trễ tới  $n$  giây
- **Không scale:**  $100 \text{ consumer} \times \text{polling} = \text{flood request}$

## Giải pháp 2: Callback functions trực tiếp

- Source giữ list callback, gọi khi data thay đổi
- **Tốt hơn**, nhưng:
  - Không có contract chung  $\Rightarrow$  mỗi callback khác signature
  - Khó quản lý lifecycle (memory leak nếu quên remove)
  - Không có cấu trúc — dễ thành “callback hell”

**Cần:** Cơ chế thông báo **có cấu trúc, loose coupling, dễ quản lý** lifecycle.

## Đảo ngược hướng phụ thuộc

Source chỉ phát tín hiệu,  
ai quan tâm thì tự đăng ký nhận.

### Ý tưởng then chốt:

1. Định nghĩa **interface chung** cho tất cả observer
2. Subject chỉ biết interface, **không biết** implementation
3. Observer tự subscribe() / unsubscribe()

### Nguyên lý:

- **Loose coupling:** chỉ biết nhau qua interface
- **Open/Closed Principle:** thêm observer = zero changes
- **Hollywood Principle:** “Don’t call us, we’ll call you”

# Observer: Cách hoạt động

## Các vai trò trong pattern:

- **Subject**: giữ list observers, cung cấp subscribe/unsubscribe/notify
- **Observer**: định nghĩa update() — contract chung
- **ConcreteObserver**: implement update()

## Flow thực thi:

```
WeatherStation station = new WeatherStation();

station.subscribe(new PhoneDisplay());
station.subscribe(new WebDashboard());
station.subscribe(new Logger());

station.setTemp(28.5);
// → notifyAll() → loop List<Observer>
//   → PhoneDisplay.update(28.5)
//   → WebDashboard.update(28.5)
//   → Logger.update(28.5)
```

# Observer: Trade-offs

## Được gì:

- Subject và Observer **loosely coupled**
- Thêm/bớt observer **lúc runtime** dễ dàng
- Tuân thủ OCP — mở rộng không sửa code cũ
- Tái sử dụng Subject cho nhiều ngữ cảnh

## Mất gì:

- **Thứ tự thông báo** không đảm bảo
- **Memory leak**: quên unsubscribe ⇒ giữ reference
- **Cascade updates**: observer A → trigger B → vòng lặp
- **Khó debug**: flow “vô hình” — không trace được
- **Performance**: nhiều observer × frequent updates

**Lưu ý:** Hệ thống lớn nên dùng **Event Bus / Message Queue**.

# Observer: Khi nào dùng? Khi nào là thừa?

## Dùng khi:

- Quan hệ **one-to-many** với consumer **không cố định**
- Cần **decouple** source khỏi logic xử lý — cross-module
- Hệ thống **event-driven** với nhiều loại sự kiện

## Over-engineering khi:

- Chỉ **1 consumer duy nhất** và cố định — gọi thằng hơn
- Quan hệ **sẽ không thay đổi** trong tương lai
- Team nhỏ, codebase nhỏ — thêm indirection vô ích
- Event **phải đảm bảo thứ tự** — Observer không guarantee

## Dấu hiệu over-design:

“Đêm được số consumer trên 1 tay và chúng không thay đổi.”

# Strategy

Behavioral Pattern

# Strategy: Bối cảnh thực tế

## Vấn đề:

- Bạn có một tác vụ cụ thể cần thực hiện.
- Có **nhiều cách** (thuật toán) để giải quyết tác vụ đó.
- Cần chuyển đổi linh hoạt giữa các cách này (thậm chí lúc runtime).

## Ví dụ dễ hiểu:

- **Bản đồ (Google Maps):** Đi từ A đến B → Có thể chọn: *Xe hơi, Xe máy, Xe buýt, Đi bộ.*  
(Mục tiêu giống nhau, nhưng thuật toán tìm đường khác nhau).
- **Thanh toán:** Mua hàng → Có thể chọn: *Thẻ tín dụng, Momo, PayPal, Tiền mặt.*
- **Sắp xếp:** List danh sách → Chọn: *QuickSort, MergeSort, BubbleSort.*

**Mẫu chốt:** Cùng một mục đích, nhưng cách làm (chiến thuật) khác

# Strategy: Code xấu khi thiếu Pattern

**Cách làm “Ngây thơ”:** Dồn hết logic vào một hàm khổng lồ.

```
class Navigator {  
    void buildRoute(String type) {  
        if (type.equals("ROAD")) {  
            // 100 dòng code tìm đường bộ  
            // Xử lý kẹt xe, đèn đỏ...  
        }  
        else if (type.equals("WALK")) {  
            // 100 dòng code đường đi bộ  
            // Xử lý via hè, công viên...  
        }  
        // else if ("PUBLIC_TRANSIT")...  
    }  
}
```

**Hậu quả:**

- **Class Khổng Lồ:** Khó đọc, khó bảo trì.
- **Vi phạm OCP:** Muốn thêm loại phương tiện mới → Phải sửa class cũ.
- **Khó test:** Lỗi ở thuật toán "Đi bộ" có thể làm hỏng cả chức năng "Đi xe".

## Encapsulate Algorithms

### Ý tưởng then chốt:

1. Tách từng thuật toán ra khỏi class chính.
2. Đưa mỗi thuật toán vào một class riêng gọi là **Strategy**.
3. Các Strategy này phải tuân thủ cùng một **Interface**.
4. Class chính (*Context*) chỉ giữ một tham chiếu đến Interface này.

### Lợi ích:

- **Context**: Không cần biết chi tiết thuật toán, chỉ cần gọi hàm `execute()`.
- **Linh hoạt**: Có thể tráo đổi thuật toán ngay khi chương trình đang chạy.
- **Độc lập**: Các thuật toán phát triển riêng biệt, không ảnh hưởng nhau.

# Strategy: Ví dụ minh họa

```
// 1. Strategy Interface
interface PaymentStrategy { void pay(int amount); }

// 2. Concrete Strategies
class CreditCard implements PaymentStrategy {
    public void pay(int amount) { print("Quẹt thẻ: " + amount); }
}
class PayPal implements PaymentStrategy {
    public void pay(int amount) { print("Trú ví PayPal: " + amount); }
}

// 3. Context
class Order {
    private PaymentStrategy strategy;
    public void setStrategy(PaymentStrategy s) { this.strategy = s; }

    public void checkout(int money) {
        strategy.pay(money); // Ủy quyền cho Strategy
    }
}
```

Client có thể đổi từ CreditCard sang PayPal dễ dàng.



# Strategy: Được và Mất

## Ưu điểm:

- **Clean Code:** Loại bỏ các khối 'if/else' hoặc 'switch' phức tạp.
- **Open/Closed Principle:** Thêm thuật toán mới không cần sửa code cũ.
- **Tách biệt (Isolation):** Logic nghiệp vụ tách rời khỏi chi tiết triển khai thuật toán.

## Nhược điểm:

- **Số lượng Class tăng:** Mỗi thuật toán là một file class mới.
- **Client phải hiểu:** Người dùng (Client) phải biết sự khác nhau giữa các Strategy để chọn cho đúng.

**Lưu ý:** Các Strategy nên là *stateless* (không lưu trạng thái) để tránh lỗi khi dùng chung.

# Strategy: Khi nào dùng?

## Nên dùng:

- Khi muốn sử dụng các biến thể khác nhau của một thuật toán.
- Khi có quá nhiều điều kiện 'if/else' chỉ để chọn cách xử lý.
- Khi muốn giấu đi sự phức tạp của thuật toán khỏi người dùng.

## Không nên dùng:

- Nếu chỉ có 1-2 thuật toán và chúng hiếm khi thay đổi.
- Logic quá đơn giản, việc tách class làm code rối thêm (Over-engineering).

**Dấu hiệu nhận biết:** “Cùng input đầu vào, nhưng cần output/cách xử lý khác nhau tùy hoàn cảnh.”

# Tổng kết

- Design Patterns là công cụ, không phải mục đích
- 3 nhóm: Creational, Structural, Behavioral
- Mỗi pattern giải quyết vấn đề cụ thể
- Áp dụng đúng lúc, đúng chỗ
- Cân nhắc trade-offs

# Key Takeaways

1. Hiểu vấn đề trước khi áp dụng pattern
2. Patterns giúp giao tiếp hiệu quả
3. Không over-engineer
4. Kết hợp patterns khi phù hợp
5. Thực hành để nắm vững

# Tài liệu tham khảo

- *Design Patterns: Elements of Reusable Object-Oriented Software* (Gang of Four)
- *Head First Design Patterns* (Freeman & Freeman)
- Refactoring.Guru
- SourceMaking.com
- GitHub repositories với implementations

Cảm ơn đã lắng nghe!

Câu hỏi?