

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 \neq 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

Abstract Factory

Creational Pattern

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: Những giải pháp ngây thơ

Thử lần 1: Dùng boolean flags

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

Vấn đề:

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

Thử lần 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:** Ổ 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 cổng 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 (hãy 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: Những giải pháp ngây thơ

Thử lần 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**: $100 \text{ consumer} \times \text{polling} = \text{flood request}$

Thử lần 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 \Rightarrow giữ reference
- **Cascade updates**: observer A \rightarrow trigger B \rightarrow vòng lặp
- **Khó debug**: flow “vô hình” — không trace được
- **Performance**: nhiều observer \times 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 thẳng 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:

“Đếm 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 **nhều 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 vỉa 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 trao 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

Bài toán: Thanh toán đơn hàng với nhiều phương thức khác nhau.

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

// 2. Concrete Strategies (Các thuật toán cụ thể)
class CreditCard implements PaymentStrategy {
    public void pay(int amount) { print("Quẹt thẻ: " + amount); }
}

class PayPal implements PaymentStrategy {
    public void pay(int amount) { print("Trừ ví PayPal: " + amount); }
}

// 3. Context (Lớp ngữ cảnh sử dụng)
class Order {
    private PaymentStrategy strategy;

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

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

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

- *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?