

Chương 4: Nguyên Lý Thiết Kế & Design Patterns Cơ Bản

Chương này sẽ đưa bạn vào thế giới thiết kế phần mềm chuyên nghiệp, nơi code không chỉ chạy được mà còn phải dễ bảo trì, mở rộng và kiểm thử. Đây là bước nhảy vọt từ lập trình viên sang kỹ sư phần mềm.

NÂNG CAO

DESIGN PATTERNS

SOLID

Mục Tiêu Học Tập

Sau khi hoàn thành chương này, bạn sẽ nắm vững các kỹ năng thiết kế phần mềm chuyên nghiệp và có thể áp dụng vào dự án thực tế.



Nguyên lý SOLID

Hiểu và áp dụng các nguyên lý S, O, D trong thiết kế code



UML Class Diagram

Đọc, vẽ và phân tích sơ đồ lớp cơ bản



Dependency Inversion

Nắm vững nguyên lý phụ thuộc đảo ngược



Design Patterns

Implement Singleton và Factory Pattern

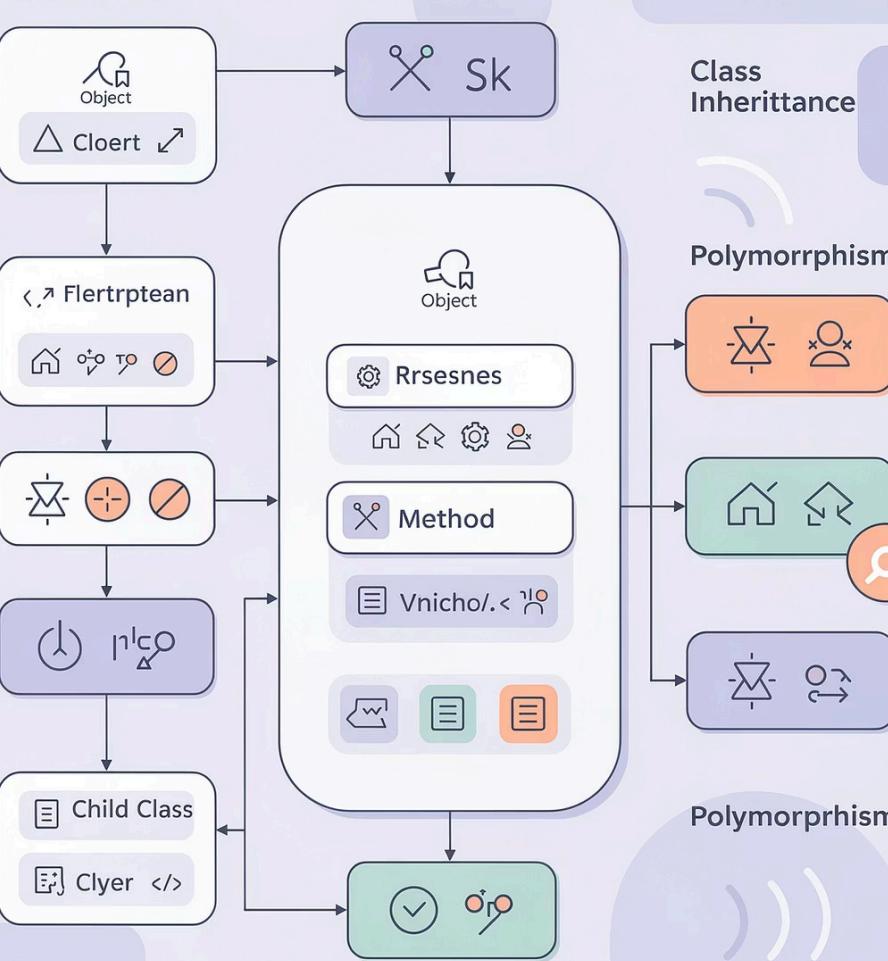


Ứng dụng thực tế

Áp dụng patterns vào dự án thực tế

Kiến Thức Cần Có (Prerequisites)

Weter Class



Class Inheritance

Polymorrphism

⚠️ **LƯU Ý QUAN TRỌNG:** Chương này là phần nâng cao. Nếu bạn chưa nắm vững Chương 2 và 3, hãy ôn tập lại trước. Phần này có thể khó với người mới, nhưng rất quan trọng cho việc viết code chuyên nghiệp.

1

Class và Object

Thiết kế class với encapsulation, constructors, getters/setters

Tham khảo: *Chương 2*

2

Inheritance

Kế thừa với extends, super, quan hệ IS-A

Tham khảo: *Chương 3, phần 3.1*

3

Interface

Định nghĩa và implement interface, multiple interfaces

Tham khảo: *Chương 3, phần 3.4*

4

Polymorphism

Upcasting, downcasting, dynamic binding

Tham khảo: *Chương 3, phần 3.5*

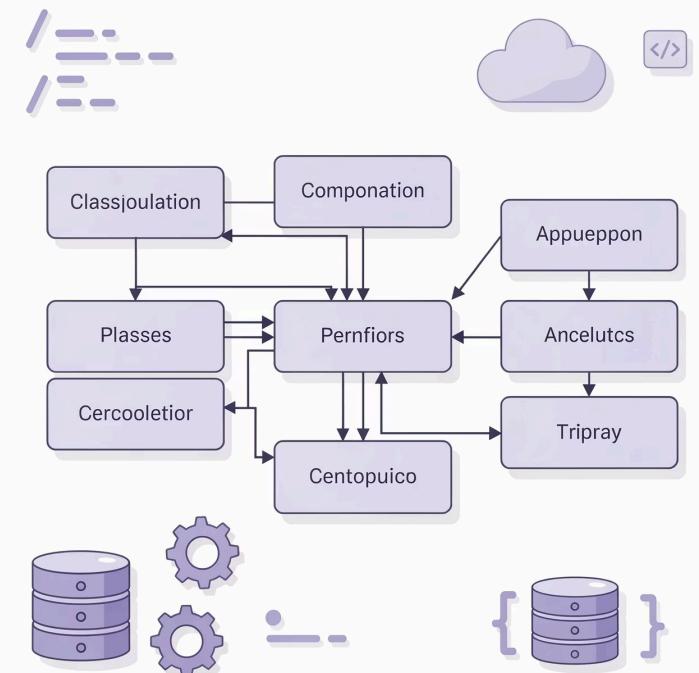
UML Class Diagram - Ngôn Ngữ Thiết Kế Phần Mềm

UML là gì?

UML (Unified Modeling Language) là ngôn ngữ mô hình hóa chuẩn được sử dụng rộng rãi trong ngành công nghiệp phần mềm.

Mục đích chính:

- Mô tả thiết kế và kiến trúc hệ thống
- Giao tiếp giữa các developer và stakeholder
- Tài liệu hóa cấu trúc phần mềm
- Lập kế hoạch trước khi code



UML Class Diagram đặc biệt hữu ích để thể hiện classes, attributes, methods và quan hệ giữa chúng một cách trực quan.



Cấu Trúc Một Class Trong UML

Biểu diễn chuẩn

```

classDiagram
    class ClassName {
        - attribute1: type
        + attribute2: type
        # attribute3: type
    }
    class ClassName {
        + method1(): type
        - method2(): void
        # method3(): type
    }

```

Access Modifiers

- (minus) = private
- + (plus) = public
- # (hash) = protected
- ~ (tilde) = package/default

Ví dụ: Class Student

```

classDiagram
    class Student {
        - name: String
        - age: int
        - gpa: double
    }
    class Student {
        + getName(): String
        + setAge(int): void
        + displayInfo()
    }

```

Các Loại Quan Hệ Trong UML

UML định nghĩa nhiều loại quan hệ giữa các class. Mỗi loại quan hệ thể hiện một mức độ phụ thuộc và ngữ nghĩa khác nhau.

1 Association (Liên kết)

Quan hệ "có liên quan đến". Class A biết về class B và có thể tương tác.



2 Aggregation (Tập hợp)

Quan hệ "HAS-A" yếu. Object có thể tồn tại độc lập.



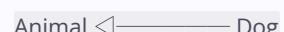
3 Composition (Kết hợp)

Quan hệ "HAS-A" mạnh. Object không thể tồn tại độc lập.



4 Inheritance (Kế thừa)

Quan hệ "IS-A". Class con kế thừa từ class cha.



5 Implementation (Triển khai)

Class implement interface.

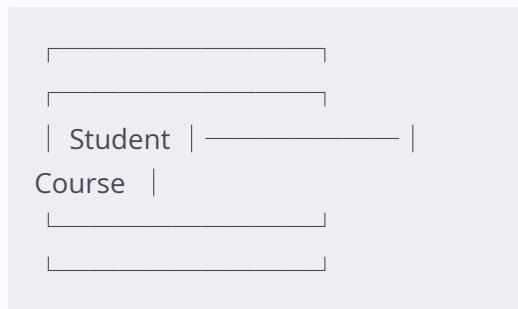


Association - Quan Hệ Liên Kết

Đặc điểm

Association là quan hệ cơ bản nhất, thể hiện sự liên kết giữa hai class. Một object của class này có thể tham chiếu đến object của class kia.

Biểu diễn UML



Ý nghĩa

Student "học" Course. Student có thể tham chiếu đến Course.

Code minh họa

```
public class Student {  
    private String name;  
    private Course course;  
  
    public Student(String name) {  
        this.name = name;  
    }  
  
    public void enrollCourse(Course c) {  
        this.course = c;  
    }  
}  
  
public class Course {  
    private String courseName;  
  
    public Course(String name) {  
        this.courseName = name;  
    }  
}
```



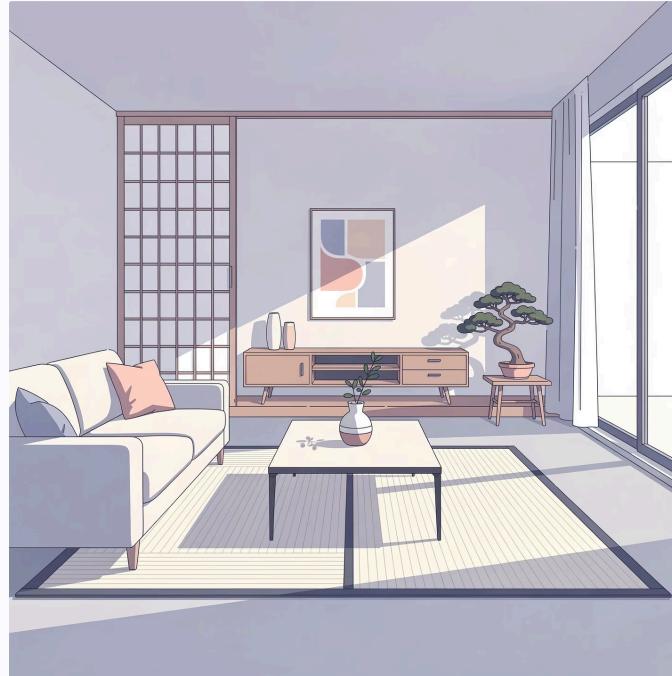
Aggregation vs Composition

Cả hai đều là quan hệ "HAS-A" nhưng khác nhau về mức độ phụ thuộc. Đây là sự khác biệt quan trọng trong thiết kế hướng đối tượng.

Aggregation (\diamond) - Tập hợp



Composition (\blacklozenge) - Kết hợp



Đặc điểm:

- Quan hệ "HAS-A" **yếu**
- Object có thể tồn tại độc lập
- Shared ownership



Code:

```
public class Team {  
    private List<Player> players;  
    // Player tồn tại độc lập  
}
```

Ví dụ: Nếu Team giải tán, Player vẫn tồn tại

Đặc điểm:

- Quan hệ "HAS-A" **mạnh**
- Object không thể tồn tại độc lập
- Exclusive ownership



Code:

```
public class House {  
    private List<Room> rooms;  
    // Room phụ thuộc House  
}
```

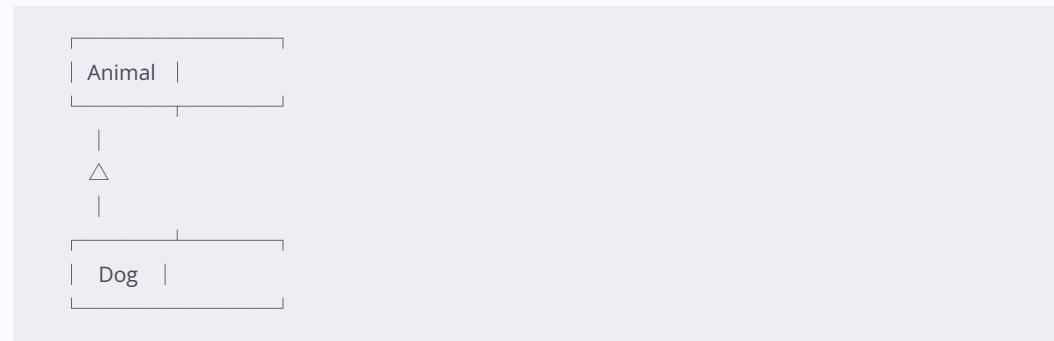
Ví dụ: Nếu House bị phá, Room cũng mất

Inheritance và Implementation

Inheritance (Kế thừa)

Class con kế thừa tất cả thuộc tính và phương thức từ class cha.

Biểu diễn UML:



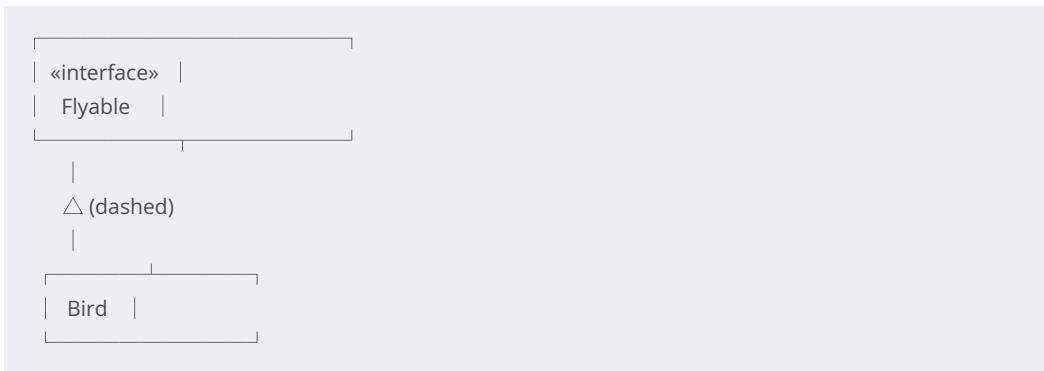
Code:

```
public class Animal {  
    protected String name;  
  
    public void eat() {  
        System.out.println("Eating");  
    }  
}  
  
public class Dog extends Animal {  
    public void bark() {  
        System.out.println("Woof!");  
    }  
}
```

Implementation (Triển khai)

Class triển khai các phương thức được định nghĩa trong interface.

Biểu diễn UML:



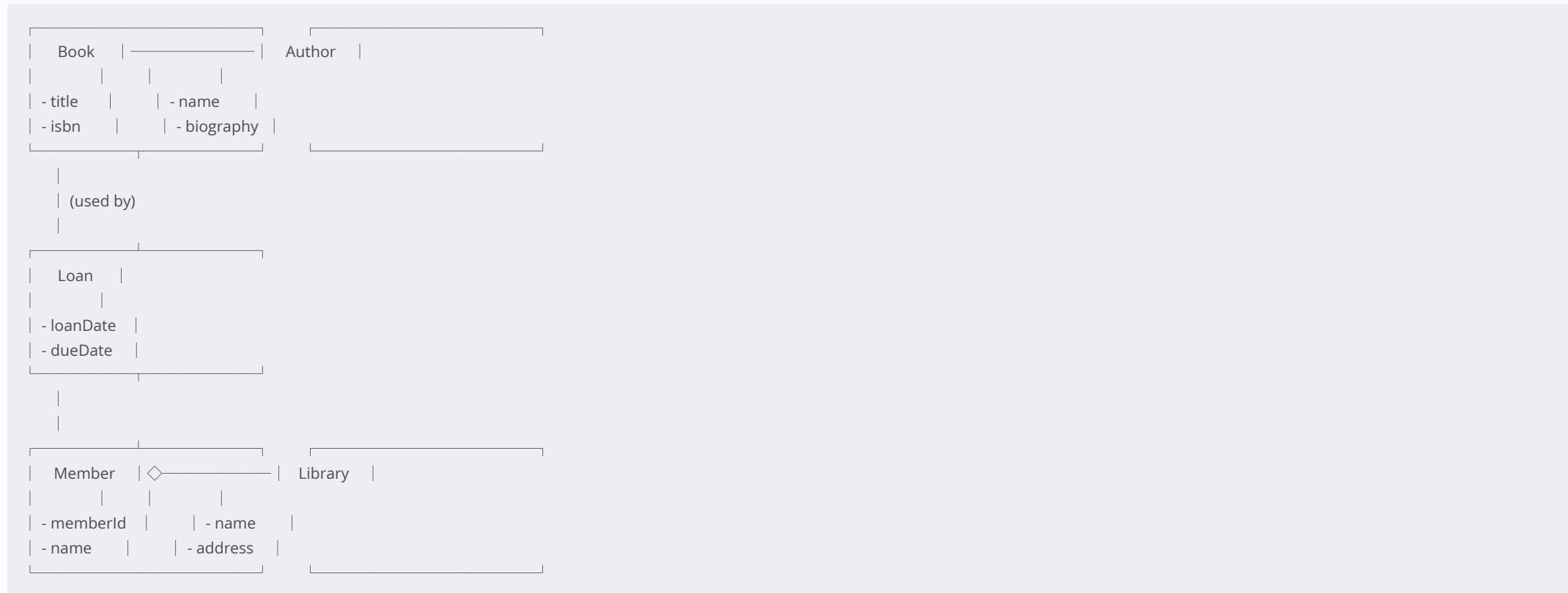
Code:

```
public interface Flyable {  
    void fly();  
}  
  
public class Bird  
implements Flyable {  
  
    @Override  
    public void fly() {  
        System.out.println("Flying");  
    }  
}
```

Ví Dụ UML Hoàn Chỉnh: Hệ Thống Thư Viện

Dưới đây là một ví dụ thực tế về cách các quan hệ UML làm việc cùng nhau trong một hệ thống hoàn chỉnh.

Sơ đồ UML:



Phân tích quan hệ:

- Book - Author:** Association (Sách liên kết với tác giả)
- Book - Loan:** Association (Sách có thể được mượn)
- Member - Loan:** Association (Thành viên mượn sách)
- Library - Member:** Aggregation (Thư viện quản lý thành viên, nhưng thành viên có thể tồn tại độc lập)

SOLID - Nền Tảng Thiết Kế Phần Mềm

SOLID là tập hợp 5 nguyên lý thiết kế hướng đối tượng được đề xuất bởi Robert C. Martin (Uncle Bob). Đây là nền tảng để viết code sạch, dễ bảo trì và mở rộng.

S - Single Responsibility

Một class chỉ nên có một lý do để thay đổi

O - Open/Closed

Mở cho mở rộng, đóng cho sửa đổi

L - Liskov Substitution

Class con có thể thay thế class cha

I - Interface Segregation

Không ép client implement interface không dùng

D - Dependency Inversion

Phụ thuộc vào abstraction, không phải concrete

❑  **Lưu ý:** Trong chương này, chúng ta tập trung vào **S, O, D** - ba nguyên lý quan trọng và thực tiễn nhất cho lập trình viên.

Single Responsibility Principle (SRP)

Một Class, Một Trách Nhiệm

Định nghĩa: "Một class chỉ nên có một lý do để thay đổi" - nghĩa là mỗi class chỉ nên có một trách nhiệm duy nhất.



Ví dụ thực tế: Người Bồi Bàn

Nhiệm vụ đúng: Ghi order và bê món ăn

Nếu kiêm nhiệm: Nấu ăn + Rửa bát + Giữ xe

Hậu quả: Quá tải! Nếu xe khách bị mất, bồi bàn phải chịu trách nhiệm - không hợp lý!

✓ Giải pháp

Thuê riêng cho từng vai trò:

- **Đầu bếp (Cook)** - Nấu ăn
- **Tạp vụ (Cleaner)** - Rửa bát
- **Bảo vệ (Security)** - Giữ xe
- **Bồi bàn (Waiter)** - Ghi order và phục vụ

Mỗi người một việc, chuyên trách nhiệm rõ ràng!

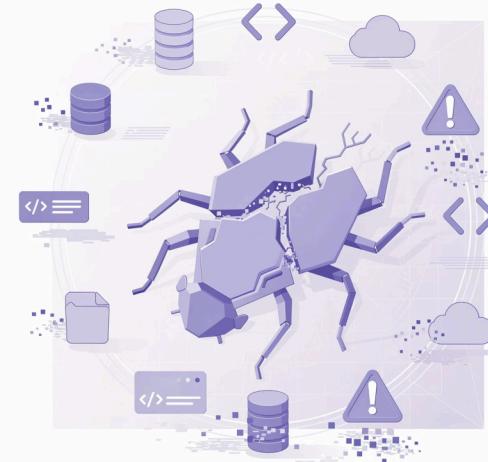
SRP: Ví Dụ Vi Phạm

Hãy xem một ví dụ điển hình về việc vi phạm Single Responsibility Principle và hậu quả của nó.

✗ Code tồi - Nhiều trách nhiệm

```
public class Student {  
    private String name;  
    private int age;  
  
    // Trách nhiệm 1: Quản lý thông tin  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    // Trách nhiệm 2: Lưu Database  
    public void saveToDatabase() {  
        // Code lưu database  
    }  
  
    // Trách nhiệm 3: Gửi Email  
    public void sendEmail(String msg) {  
        // Code gửi email  
    }  
  
    // Trách nhiệm 4: Tạo Report  
    public void generateReport() {  
        // Code tạo report  
    }  
}
```

💣 Tác động khi vi phạm



Kịch bản thực tế:

1. **Yêu cầu:** Đổi MySQL → MongoDB
2. **Hành động:** Sửa saveToDatabase()
3. **Tai nạn:** Lỡ tay xóa import cho sendEmail()
4. **Hậu quả:** Chức năng Email chết ngắc!

Kết luận: Một thay đổi ở A làm hỏng B → Code bị **coupling** quá chặt!

SRP: Giải Pháp Đúng Đắn

Tách class Student thành nhiều class, mỗi class một trách nhiệm rõ ràng.

Student

Trách nhiệm: Quản lý thông tin sinh viên

```
public class Student {  
    private String name;  
    private int age;  
  
    // Getters/Setters  
}
```

StudentRepository

Trách nhiệm: Lưu trữ và truy xuất dữ liệu

```
public class  
StudentRepository {  
    public void save(  
        Student s) {  
        // Database  
    }  
}
```

EmailService

Trách nhiệm: Gửi email

```
public class  
EmailService {  
    public void send(  
        String to,  
        String msg) {  
        // Email  
    }  
}
```

ReportGenerator

Trách nhiệm: Tạo báo cáo

```
public class  
ReportGenerator {  
    public void  
    generate(  
        Student s) {  
        // Report  
    }  
}
```

✓ Lợi ích:

- Mỗi class chỉ có một lý do để thay đổi
- Dễ test từng phần riêng biệt
- Dễ bảo trì và mở rộng
- Giảm coupling, tăng cohesion

Cách Nhận Biết Vi Phạm SRP

红旗 Dấu hiệu nhận biết

- Class có quá nhiều methods

Nếu class có hơn 10-15 methods, có thể đang làm quá nhiều việc

- Khó đặt tên class

Phải dùng "và", "hoặc" trong tên (VD:
StudentAndCourseManager)

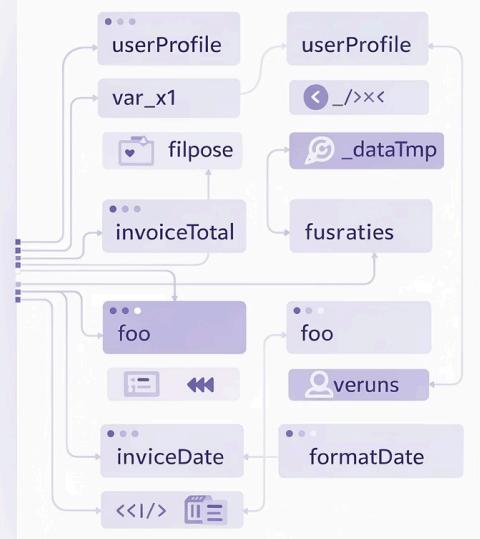
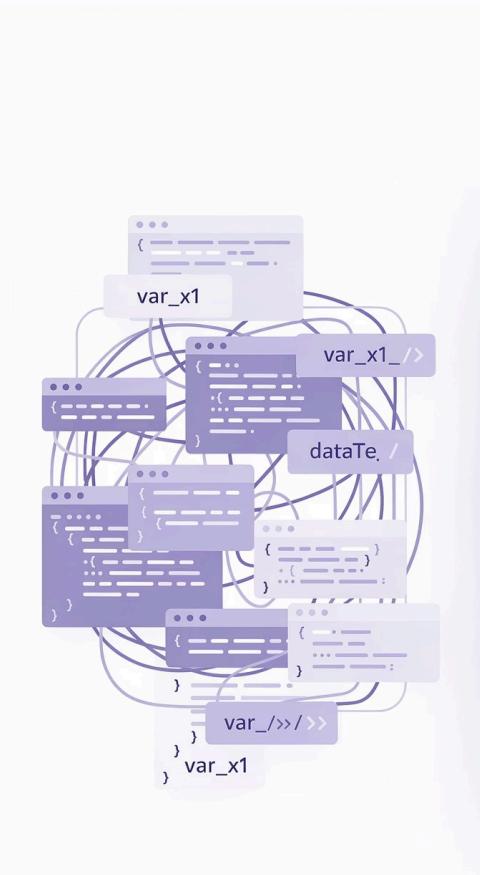
- Nhiều lý do để thay đổi

Thay đổi database, UI, business logic đều phải sửa cùng một class

- Khó test

Phải mock quá nhiều dependencies để test một phần nhỏ

✓ Ví dụ refactor tên class



Trước (Vi phạm SRP):

//

Open/Closed Principle (OCP)

Mở Cho Mở Rộng, Đóng Cho Sửa Đổi

Định nghĩa: "Software entities should be open for extension, but closed for modification"

Ví dụ: Điện thoại thông minh

Khi bạn muốn thêm tính năng "Chỉnh sửa ảnh":

- **Cách đúng:** Cài thêm App (Extension)
- **Cách sai:** Tháo tung điện thoại, hàn thêm chip (Modification)

Điện thoại được thiết kế "**Mở**" để cài App, nhưng "**Đóng**" với phần cứng bên trong.

Áp dụng vào code

- **Open for extension:** Có thể thêm tính năng mới bằng cách thêm class mới
- **Closed for modification:** Không cần sửa code cũ đã hoạt động tốt

Lợi ích:

- Giảm rủi ro khi thêm tính năng mới
- Code cũ ổn định, không bị ảnh hưởng
- Dễ bảo trì và mở rộng

OCP: Ví Dụ Vi Phạm

✗ Code tồi - Phải sửa khi thêm loại mới

```
public class AreaCalculator {  
    public double calculateArea(Object shape) {  
        if (shape instanceof Circle) {  
            Circle circle = (Circle) shape;  
            return Math.PI * circle.getRadius() * circle.getRadius();  
        }  
        else if (shape instanceof Rectangle) {  
            Rectangle rect = (Rectangle) shape;  
            return rect.getWidth() * rect.getHeight();  
        }  
        else if (shape instanceof Triangle) {  
            Triangle triangle = (Triangle) shape;  
            return 0.5 * triangle.getBase() * triangle.getHeight();  
        }  
    //
```

OCP: Giải Pháp Với Polymorphism

Áp dụng nguyên lý Open/Closed Principle (OCP) bằng cách sử dụng kế thừa và tính đa hình (polymorphism). Điều này cho phép mở rộng hệ thống bằng cách thêm các loại hình mới mà không cần sửa đổi mã nguồn hiện có.

✓ Code tốt - Sử dụng Abstract Class và Polymorphism

```
// 1. Lớp trừu tượng cơ sở: Shape
public abstract class Shape {
    public abstract double calculateArea();
}

// 2. Các lớp con triển khai Shape
// Lớp Circle
public class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

// Lớp Rectangle
public class Rectangle extends Shape {
    private double width, height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double calculateArea() {
        return width * height;
    }
}

// 3. Lớp AreaCalculator đã đơn giản hóa
public class AreaCalculator {
    public double calculateShapeArea(Shape shape) {
        return shape.calculateArea();
    }
}
```

Lợi ích của giải pháp này

Sử dụng lớp trừu tượng Shape và tính đa hình giúp tách biệt trách nhiệm tính toán diện tích. Mỗi hình tự chịu trách nhiệm về cách tính diện tích của chính nó.

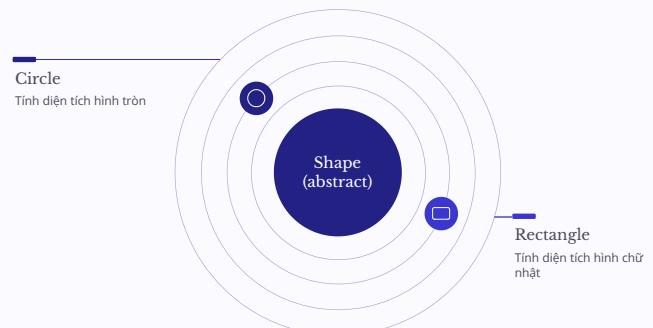
Điều này hiện thực hóa nguyên tắc **Open for extension, Closed for modification**:

- **Mở để mở rộng:** Dễ dàng thêm loại hình mới (ví dụ: Hexagon) bằng cách tạo lớp con mới kế thừa Shape và triển khai calculateArea().
- **Đóng để sửa đổi:** Lớp AreaCalculator không cần sửa đổi khi thêm loại hình mới, giảm rủi ro lỗi.

So sánh:

- **Trước (Vi phạm OCP):** Mỗi khi thêm hình mới, phải sửa đổi lớp AreaCalculator.
- **Nay (Tuân thủ OCP):** Khi thêm hình mới, chỉ cần tạo lớp con kế thừa Shape, lớp AreaCalculator giữ nguyên.

Cấu trúc kế thừa:



OCP: Thực Hành Refactoring

Hãy cùng refactor một ví dụ thực tế về hệ thống thanh toán từ vi phạm OCP sang tuân thủ OCP.

✗ Code cũ - Vi phạm OCP

```
public class PaymentProcessor {  
    public void processPayment(String type, double amount) {  
        if (type.equals("credit")) {  
            // Process credit card  
            System.out.println("Processing credit card: " + amount);  
        }  
        else if (type.equals("paypal")) {  
            // Process PayPal  
            System.out.println("Processing PayPal: " + amount);  
        }  
    }  
}
```

Dependency Inversion Principle (DIP)

Phụ Thuộc Vào Abstraction

Định nghĩa: "High-level modules should not depend on low-level modules. Both should depend on abstractions."

Ví dụ: Phích cắm điện

- **Cái đèn (High-level):** Cần điện để sáng
- **Ổ cắm (Low-level):** Cung cấp điện

Cái đèn không quan tâm ổ cắm ở đâu (tường, dây nối dài). Nó chỉ quan tâm **phích cắm có vừa lõi** (Interface/Abstraction) hay không.

✗ Vi phạm DIP: Hàn chết dây đèn vào 1 ổ cắm cụ thể → Không thể mang đi chỗ khác

✓ Tuân thủ DIP: Dùng phích cắm chuẩn → Cắm được mọi nơi

Nguyên lý cốt lõi

01

High-level không phụ thuộc Low-level

Module cấp cao không biết chi tiết implement

02

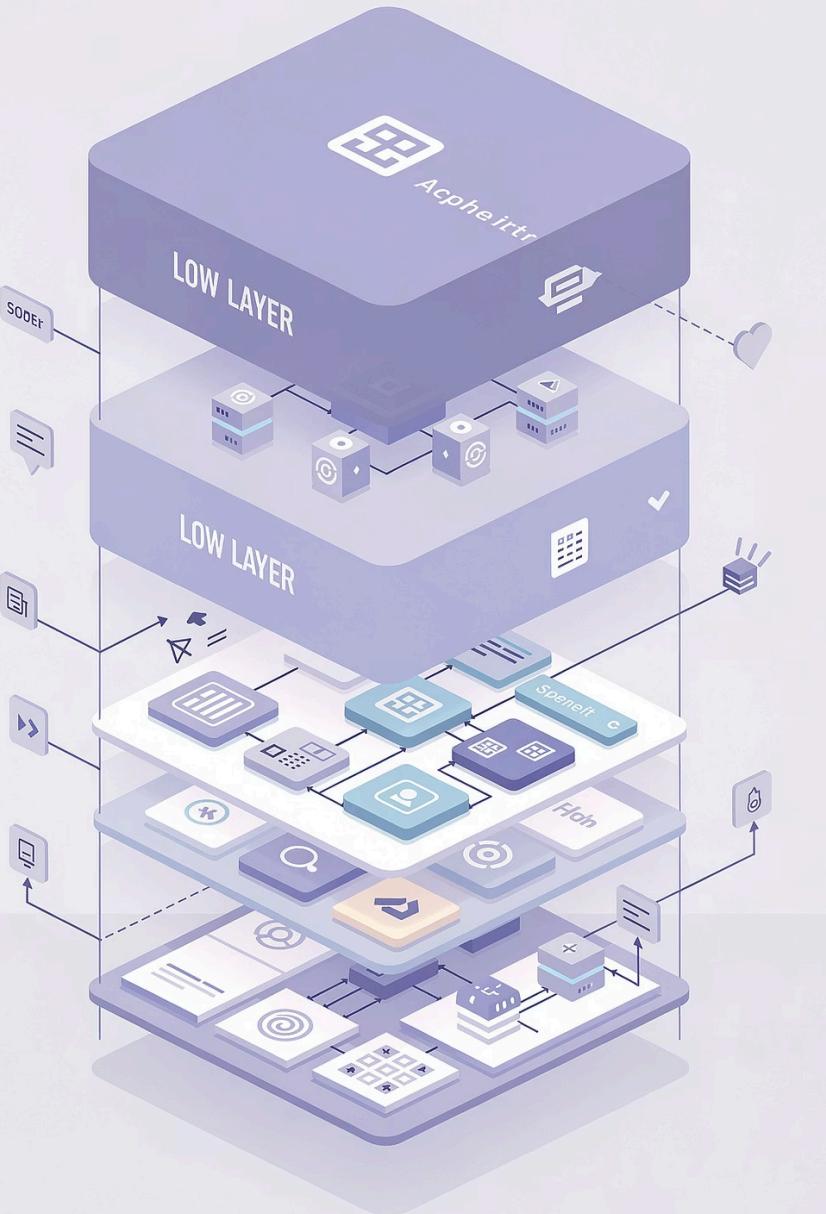
Cả hai phụ thuộc Abstraction

Dùng Interface làm cầu nối

03

Dễ thay đổi implementation

Chỉ cần thay class implement, không sửa logic



High-Level vs Low-Level Modules

Hiểu rõ sự khác biệt giữa module cấp cao và cấp thấp là chìa khóa để áp dụng DIP đúng cách.

High-Level Module

Module cấp cao

- Chứa **business logic**
- Quyết định "**Làm gì**" (What)
- Không quan tâm chi tiết

Ví dụ:

- PaymentService
- OrderService
- UserManager

Low-Level Module

Module cấp thấp

- Chứa **implementation chi tiết**
- Quyết định "**Làm thế nào**" (How)
- Xử lý kỹ thuật cụ thể

Ví dụ:

- MySQLDatabase
- FileRepository
- EmailService

Mỗi quan hệ lý tưởng:

High-Level ——

DIP: Ví Dụ Vi Phạm

✗ Code tồi - High-level phụ thuộc Low-level

```
// Low-level module
public class MySQLDatabase {
    public void save(String data) {
        System.out.println(
            "Saving to MySQL: " + data);
    }
}
```

```
// High-level module - VI PHẠM DIP
public class PaymentService {
    //
```

DIP: Giải Pháp Đúng Đắn

Để giải quyết vấn đề phụ thuộc giữa các module high-level và low-level, nguyên lý Đảo ngược Phụ thuộc (DIP) đề xuất sử dụng các abstraction (ví dụ: Interface) làm cầu nối. Thay vì phụ thuộc trực tiếp vào module low-level cụ thể, cả hai module high-level và low-level sẽ cùng phụ thuộc vào abstraction.

✓ Code tốt - Sử dụng Interface

```
// 1. Định nghĩa Interface (Abstraction)
public interface Database {
    void save(String data);
}

// 2. Các Low-level Module cụ thể implement Interface
public class MySQLDatabase implements Database {
    @Override
    public void save(String data) {
        System.out.println("Lưu dữ liệu vào MySQL: " + data);
    }
}

public class PostgreSQLDatabase implements Database {
    @Override
    public void save(String data) {
        System.out.println("Lưu dữ liệu vào PostgreSQL: " + data);
    }
}

// 3. High-level Module phụ thuộc vào Abstraction (Database Interface)
public class PaymentService {
    private final Database database; // Phụ thuộc vào interface, không phải class cụ thể

    // Sử dụng Constructor Injection để truyền Database implementation
    public PaymentService(Database database) {
        this.database = database;
    }

    public void processPayment(String paymentDetails) {
        System.out.println("Đang xử lý giao dịch: " + paymentDetails);
        database.save(paymentDetails); // Gọi phương thức qua interface
        System.out.println("Giao dịch hoàn tất.");
    }
}

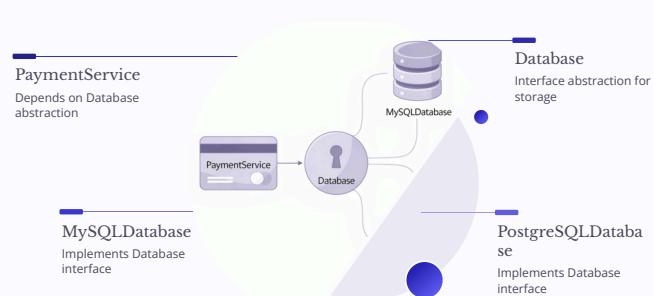
// 4. Ví dụ sử dụng: Client code kết nối High-level với Low-level qua Interface
public class Main {
    public static void main(String[] args) {
        // Sử dụng MySQL
        Database mySQLDb = new MySQLDatabase();
        PaymentService mySQLPaymentService = new PaymentService(mySQLDb);
        mySQLPaymentService.processPayment("Thanh toán đơn hàng #123");

        System.out.println("\n--- Chuyển đổi database ---\n");

        // Sử dụng PostgreSQL
        Database postgreSQLDb = new PostgreSQLDatabase();
        PaymentService postgreSQLPaymentService = new PaymentService(postgreSQLDb);
        postgreSQLPaymentService.processPayment("Thanh toán đơn hàng #456");
    }
}
```

⌚ Lợi ích của DIP

- High-level không phụ thuộc low-level:** Module high-level (PaymentService) không phụ thuộc trực tiếp vào các module low-level cụ thể (MySQLDatabase, PostgreSQLDatabase). Cả hai đều phụ thuộc vào một abstraction (Database Interface).
- Dễ dàng thay đổi database:** Có thể dễ dàng thay thế implementation của database (ví dụ: từ MySQL sang PostgreSQL) mà không cần thay đổi code của PaymentService.
- Dễ dàng test với mock:** Giúp tăng cường khả năng kiểm thử (Testability) bằng cách dễ dàng "mock" Database interface để kiểm thử PaymentService một cách độc lập.
- Code linh hoạt và bảo trì tốt:** Kiến trúc trở nên linh hoạt hơn, dễ dàng mở rộng và bảo trì trong tương lai.



Dependency Injection (DI)

Dependency Injection là kỹ thuật quan trọng để thực hiện Dependency Inversion Principle. Thay vì tạo dependency bên trong class, chúng ta nhận nó từ bên ngoài.



Constructor Injection

Khuyến nghị sử dụng

Inject qua constructor, đảm bảo dependency luôn có sẵn

```
public PaymentService(  
    Database db) {  
    this.database = db;  
}
```



Setter Injection

Dùng khi cần thay đổi

Inject qua setter method, cho phép thay đổi sau khi khởi tạo

```
public void setDatabase(  
    Database db) {  
    this.database = db;  
}
```



Field Injection

Không khuyến nghị

Inject trực tiếp vào field, phụ thuộc framework

```
@Inject  
private Database  
database;
```

Lợi ích của Dependency Injection:

- ✓ Dễ thay đổi implementation
- ✓ Dễ test với mock objects
- ✓ Tách biệt creation và usage
- ✓ Tuân thủ DIP và OCP

Các Cách Inject Dependency

Hãy xem chi tiết cách implement từng loại Dependency Injection.

Constructor Injection ✓ (Khuyên nghị)

```
1 public class PaymentService {  
    private final Database database; // final để immutable  
  
    // Inject qua constructor  
    public PaymentService(Database database) {  
        this.database = database;  
    }  
  
    public void processPayment(double amount) {  
        database.save("Payment: " + amount);  
    }  
}  
  
// Sử dụng  
Database db = new MySQLDatabase();  
PaymentService service = new PaymentService(db);
```

Ưu điểm: Dependencies rõ ràng, không thể null, immutable

Setter Injection

```
2 public class PaymentService {  
    private Database database;  
  
    // Inject qua setter  
    public void setDatabase(Database database) {  
        this.database = database;  
    }  
  
    public void processPayment(double amount) {  
        if (database != null) {  
            database.save("Payment: " + amount);  
        }  
    }  
}  
  
// Sử dụng  
PaymentService service = new PaymentService();  
service.setDatabase(new MySQLDatabase());
```

Ưu điểm: Linh hoạt thay đổi, **Nhược điểm:** Có thể null

Field Injection ! (Tránh dùng)

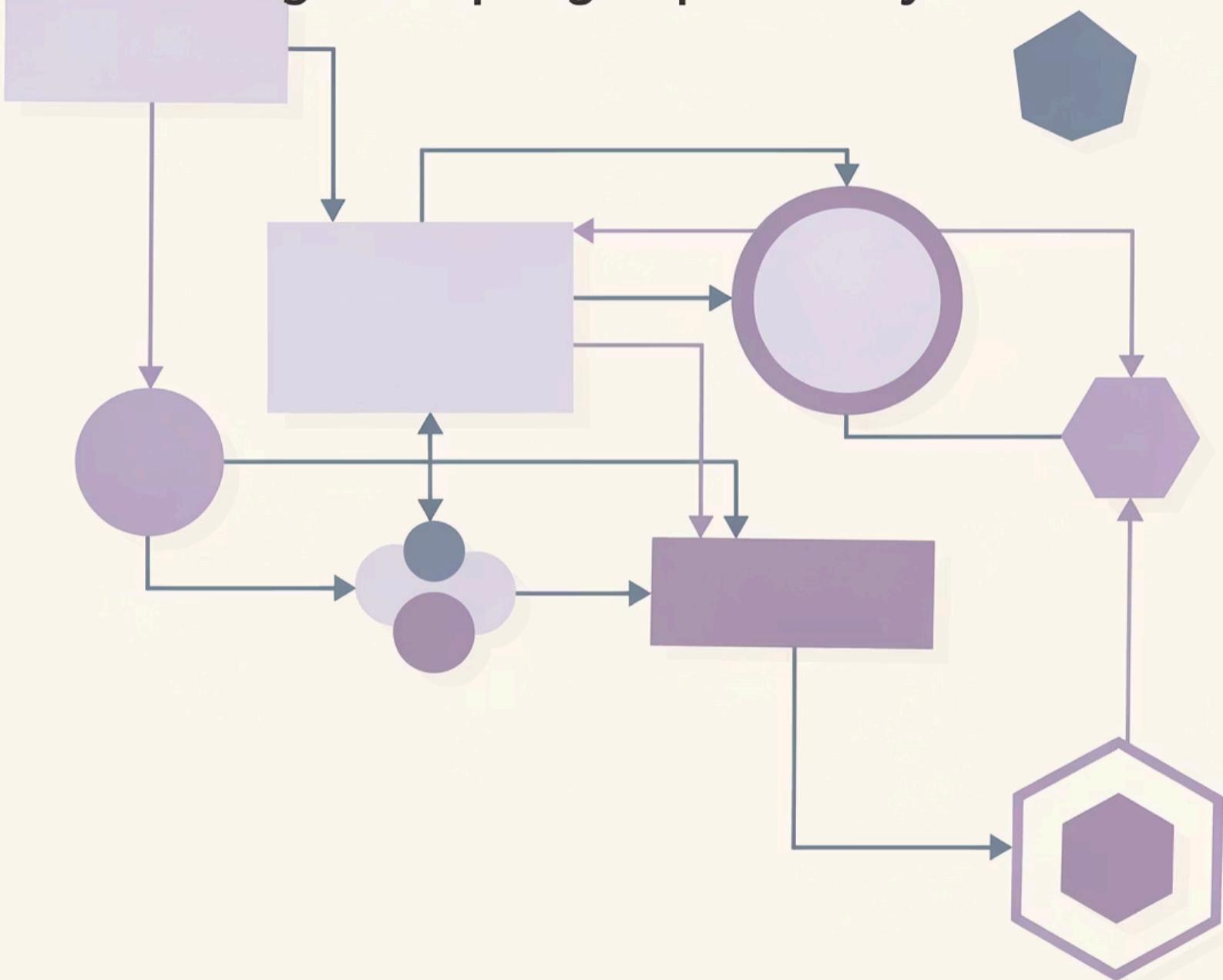
```
3 public class PaymentService {  
    @Inject // Framework annotation (Spring, CDI)  
    private Database database;  
  
    public void processPayment(double amount) {  
        database.save("Payment: " + amount);  
    }  
}
```

Nhược điểm: Phụ thuộc framework, khó test, dependencies ẩn

Tại Sao Từ Khóa "new" Là Kẻ Thủ?

✗ Vấn đề với "new"

New keyword problem tight coupling dependency



tight coupling dependency

```
public class PaymentService {  
    private Database database;  
  
    public PaymentService() {  
        //
```

Quy Tắc "new": Tạo Ở Đâu, Dùng Ở Đâu

Nguyên tắc vàng: "Nơi tạo object (new) khác với nơi sử dụng object"



Ví dụ phân tầng đúng:

```
// 1. MAIN METHOD - Nơi tạo tất cả dependencies
public class Main {
    public static void main(String[] args) {
        // Tạo low-level modules (các thành phần cấp thấp)
        Database database = new MySQLDatabase();

        // Tạo high-level modules (các thành phần cấp cao) và inject dependencies (tiêm phụ thuộc)
        PaymentService paymentService = new PaymentService(database);

        // Sử dụng logic nghiệp vụ
        paymentService.processPayment("Order #123");
    }
}

// 2. BUSINESS LOGIC - Chỉ nhận dependencies, KHÔNG tạo
public class PaymentService {
    private Database database;

    // Constructor Injection - Nhận từ bên ngoài
    public PaymentService(Database database) {
        this.database = database;
    }

    public void processPayment(String data) {
        // Chỉ SỬ DỤNG database, không tạo mới instance
        database.save(data);
    }
}

// 3. INTERFACE - Lớp trừu tượng (Abstraction layer)
public interface Database {
    void save(String data);
}

// 4. LOW-LEVEL IMPLEMENTATION - Triển khai cụ thể cho lớp trừu tượng
public class MySQLDatabase implements Database {
    @Override
    public void save(String data) {
        System.out.println("Saving to MySQL: " + data);
    }
}
```

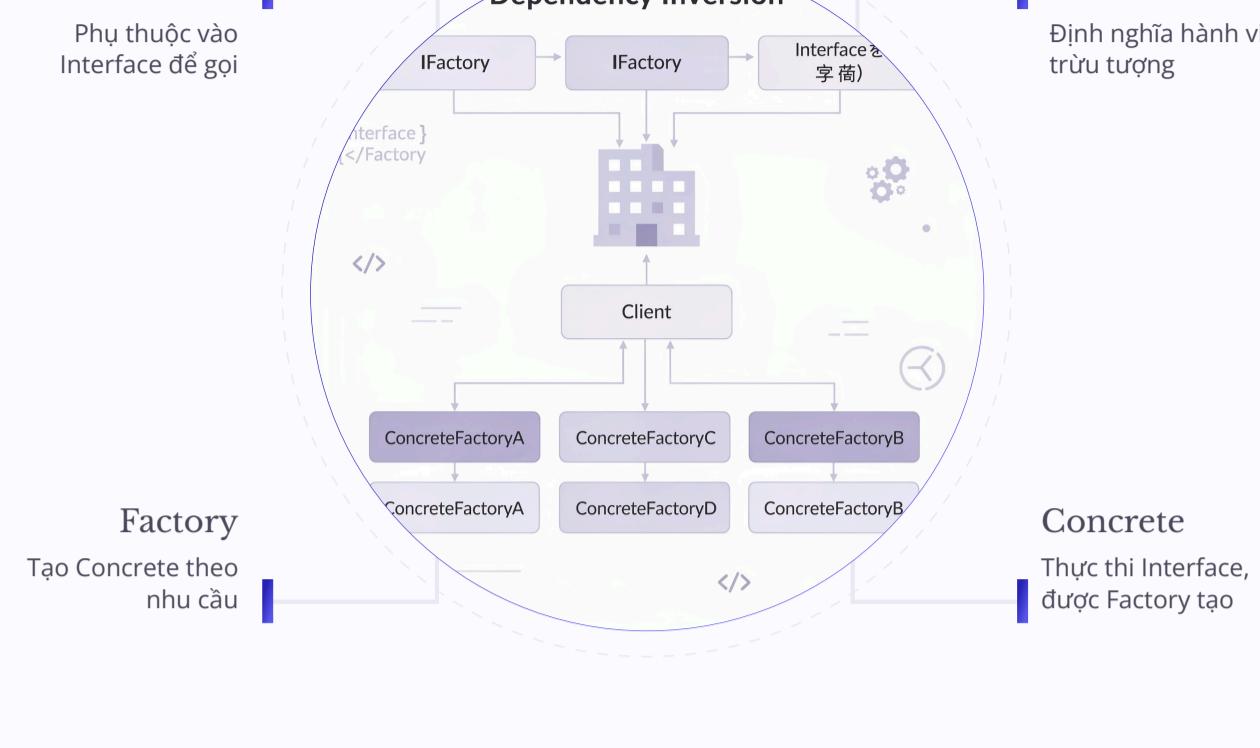
- Lợi ích: Main method biết tất cả, Business logic không biết gì về implementation cụ thể!

Liên Hệ DIP Với Factory & Interface

Nguyên tắc đảo ngược phụ thuộc (DIP), Factory Pattern và Interface là bộ ba mạnh mẽ, hoạt động cùng nhau để xây dựng các hệ thống phần mềm linh hoạt, dễ bảo trì và mở rộng. Chúng cho phép tách biệt rõ ràng giữa logic nghiệp vụ (business logic) và logic khởi tạo đối tượng (object creation logic), đảm bảo code của bạn tuân thủ các nguyên tắc thiết kế hướng đối tượng (OOP) như SOLID.

DIP quy định rằng các module cấp cao không nên phụ thuộc vào các module cấp thấp. Cả hai nên phụ thuộc vào các abstraction (Interface). Factory Pattern giúp chúng ta tạo ra các đối tượng mà không cần chỉ định chính xác lớp cụ thể nào sẽ được tạo, trong khi Interface cung cấp định nghĩa abstraction mà cả module cấp cao và cấp thấp đều phụ thuộc vào.

Luồng thiết kế chuẩn:



Sơ đồ trên minh họa:

- **Client** (ví dụ: PaymentService) chỉ phụ thuộc vào **Interface** (ví dụ: Database), không phải vào các lớp triển khai cụ thể.
- **Factory** (ví dụ: DatabaseFactory) chịu trách nhiệm tạo ra các **Concrete Implementations** (ví dụ: MySQLDatabase, PostgreSQLDatabase).
- Các **Concrete Implementations** triển khai **Interface**, đảm bảo chúng tuân thủ hợp đồng đã định.
- Client nhận đối tượng thông qua Interface, thường bằng cách sử dụng Dependency Injection (Constructor Injection là phổ biến nhất).

Ví dụ Code Minh Họa:

```
// 1. Interface (Abstraction) - Cung cấp định nghĩa mà các lớp cấp cao và cấp thấp đều phụ thuộc vào
public interface Database {
    void save(String data);
    void connect();
}

// 2. Concrete Implementations (Low-level modules) - Triển khai cụ thể của Interface
public class MySQLDatabase implements Database {
    @Override
    public void connect() {
        System.out.println("Connecting to MySQL Database...");
        // Logic kết nối MySQL
    }

    @Override
    public void save(String data) {
        System.out.println("Saving data to MySQL: " + data);
        // Logic lưu dữ liệu vào MySQL
    }
}

public class PostgreSQLDatabase implements Database {
    @Override
    public void connect() {
        System.out.println("Connecting to PostgreSQL Database...");
        // Logic kết nối PostgreSQL
    }

    @Override
    public void save(String data) {
        System.out.println("Saving data to PostgreSQL: " + data);
        // Logic lưu dữ liệu vào PostgreSQL
    }
}

// 3. Factory Pattern - Chịu trách nhiệm tạo ra các Concrete Implementations
// giúp tách biệt logic khởi tạo khỏi Business Logic.
public class DatabaseFactory {
    public static Database getDatabase(String type) {
        if ("mysql".equalsIgnoreCase(type)) {
            MySQLDatabase db = new MySQLDatabase(); // Nơi duy nhất sử dụng 'new' cho MySQLDatabase
            db.connect(); // Kết nối ngay khi tạo
            return db;
        } else if ("postgresql".equalsIgnoreCase(type)) {
            PostgreSQLDatabase db = new PostgreSQLDatabase(); // Nơi duy nhất sử dụng 'new' cho PostgreSQLDatabase
            db.connect(); // Kết nối ngay khi tạo
            return db;
        } else {
            throw new IllegalArgumentException("Unsupported database type: " + type);
        }
    }
}

// 4. Business Logic (High-level module) - Chỉ phụ thuộc vào Interface (Database)
// và nhận nó thông qua Constructor Injection, không biết gì về Concrete Implementations.
public class PaymentService {
    private final Database database;

    // Constructor Injection: PaymentService nhận một đối tượng Database
    // mà nó sẽ tương tác, không quan tâm nó là MySQL hay PostgreSQL.
    public PaymentService(Database database) {
        this.database = database;
    }

    public void processPayment(String orderId, double amount) {
        System.out.println("Processing payment for Order ID: " + orderId + ", Amount: " + amount);
        String transactionData = "Order:" + orderId + ", Amount:" + amount + ", Status:Completed";
        database.save(transactionData); // Sử dụng phương thức của Database interface
        System.out.println("Payment processed and data saved.");
    }
}

// 5. Main Method - Nơi điều phối, sử dụng Factory để tạo đối tượng
// và Inject chúng vào các lớp Business Logic.
public class Main {
    public static void main(String[] args) {
        // Sử dụng Factory để tạo ra một đối tượng Database cụ thể
        // Main method quyết định loại database sẽ dùng.
        Database mysqlDb = DatabaseFactory.getDatabase("mysql");
        // Hoặc
        // Database postgresDb = DatabaseFactory.getDatabase("postgresql");

        // Tạo PaymentService và Inject đối tượng Database đã tạo
        // PaymentService không cần biết Factory hay loại database cụ thể nào.
        PaymentService paymentService = new PaymentService(mysqlDb);
        // Hoặc
        // PaymentService paymentService = new PaymentService(postgresDb);

        // Chạy Business Logic
        paymentService.processPayment("ORD-2024-001", 99.99);
    }
}
```

Lợi ích khi kết hợp DIP, Factory Pattern và Interface:

1 Tách biệt Logic Khởi tạo và Logic Nghệp vụ

Factory Pattern tập trung trách nhiệm khởi tạo đối tượng vào một nơi duy nhất, giúp Business Logic của bạn sạch sẽ, dễ đọc và chỉ tập trung vào nhiệm vụ chính của nó.

2 Dễ dàng thay đổi triển khai

Khi bạn cần chuyển từ MySQL sang PostgreSQL (hoặc bất kỳ hệ thống database nào khác), bạn chỉ cần thay đổi trong Factory và Main method. Các lớp Business Logic (như PaymentService) hoàn toàn không cần chỉnh sửa.

3 Tuân thủ Nguyên tắc SOLID

- **DIP (Dependency Inversion Principle):** Các module cấp cao (PaymentService) không phụ thuộc vào module cấp thấp (MySQLDatabase), cả hai đều phụ thuộc vào Abstraction (Database Interface).
- **OCP (Open/Closed Principle):** Hệ thống "mở để mở rộng, đóng để thay đổi". Bạn có thể thêm Database loại mới mà không cần sửa đổi PaymentService.

4 Code linh hoạt và dễ kiểm thử (Testable)

Vì PaymentService phụ thuộc vào Interface, bạn có thể dễ dàng Mock (giả lập) hoặc Stub Database trong các bài kiểm thử đơn vị, giúp việc kiểm thử trở nên hiệu quả và độc lập hơn.

5 Tăng khả năng tái sử dụng (Reusability)

Các module được thiết kế độc lập và chỉ phụ thuộc vào Abstraction có thể dễ dàng được sử dụng lại trong các ngữ cảnh khác nhau.

Design Patterns - Giải Pháp Đã Được Kiểm Chứng

Học Từ Kinh Nghiệm Của Những Người Đi Trước

Design Pattern là các giải pháp tái sử dụng cho các vấn đề thiết kế phổ biến. Chúng được tổng hợp từ kinh nghiệm của hàng ngàn lập trình viên qua nhiều thập kỷ.

Lợi ích của Design Patterns



Đã được kiểm chứng

Giải pháp đã được test bởi cộng đồng



Dễ giao tiếp

Tên pattern là ngôn ngữ chung của developers



Tái sử dụng

Áp dụng cho nhiều dự án khác nhau

Patterns trong chương này

Singleton Pattern

Đảm bảo chỉ có một instance duy nhất

Factory Pattern

Tạo object mà không chỉ định class cụ thể

Singleton Pattern - Một Instance Duy Nhất

Mục đích: Đảm bảo một class chỉ có một instance duy nhất trong toàn bộ ứng dụng và cung cấp điểm truy cập toàn cục đến instance đó.

Khi nào dùng Singleton?

- Database Connection

Chỉ cần một connection pool

- Logger

Một logger cho toàn ứng dụng

- Configuration

Một object lưu config chung

- Cache

Một cache manager chung

Đặc điểm chính

- Private constructor (không cho tạo từ bên ngoài)
- Static method getInstance() để lấy instance
- Static field lưu instance duy nhất

Ví dụ sử dụng:

```
// Lấy instance  
Logger logger1 = Logger.getInstance();  
Logger logger2 = Logger.getInstance();
```

```
// Cùng một instance  
System.out.println(  
    logger1 == logger2); // true
```

```
// Sử dụng  
logger1.log("Message 1");  
logger2.log("Message 2");
```

Singleton: Cách Cài Đặt Cơ Bản (Không Thread-Safe)

⚠ **Lưu ý:** Cách này KHÔNG thread-safe. Chỉ dùng cho mục đích học tập hoặc ứng dụng single-threaded.

Mã nguồn Java (Không Thread-Safe)

Đây là cách triển khai Singleton đơn giản nhất, thường được gọi là "Lazy Initialization" vì instance chỉ được tạo khi nó thực sự cần thiết.

```
public class BasicSingleton {  
  
    // 1. Private static instance variable:  
    // Biến này sẽ giữ duy nhất một instance của lớp BasicSingleton.  
    // 'static' đảm bảo nó thuộc về lớp chứ không phải đối tượng.  
    // 'private' đảm bảo chỉ lớp BasicSingleton mới có thể truy cập nó.  
    private static BasicSingleton instance;  
  
    // 2. Private constructor:  
    // Ngăn chặn việc tạo các instance mới của lớp từ bên ngoài  
    // bằng cách sử dụng từ khóa 'new'.  
    private BasicSingleton() {  
        System.out.println("BasicSingleton: Constructor được gọi - Tạo một instance mới.");  
    }  
  
    // 3. Public static getInstance() method:  
    // Phương thức này cung cấp điểm truy cập toàn cục đến instance duy nhất.  
    // Nó kiểm tra xem instance đã tồn tại chưa, nếu chưa thì tạo mới.  
    public static BasicSingleton getInstance() {  
        // Kiểm tra nếu instance chưa được tạo  
        if (instance == null) {  
            // Nếu chưa, tạo một instance mới  
            instance = new BasicSingleton();  
        }  
        // Trả về instance duy nhất  
        return instance;  
    }  
  
    // Một phương thức ví dụ để kiểm tra hoạt động của Singleton  
    public void showMessage() {  
        System.out.println("Xin chào từ BasicSingleton!");  
    }  
  
    // Ví dụ sử dụng trong phương thức main  
    public static void main(String[] args) {  
        System.out.println("--- Bắt đầu ví dụ Singleton cơ bản ---");  
  
        // Lần gọi đầu tiên: instance sẽ được tạo  
        System.out.println("\nGọi getInstance() lần 1:");  
        BasicSingleton singleton1 = BasicSingleton.getInstance();  
        singleton1.showMessage();  
  
        // Lần gọi thứ hai: instance đã tồn tại, sẽ trả về instance cũ  
        System.out.println("\nGọi getInstance() lần 2:");  
        BasicSingleton singleton2 = BasicSingleton.getInstance();  
        singleton2.showMessage();  
  
        // Kiểm tra xem cả hai biến có trả về cùng một instance không  
        System.out.println("\nKiểm tra các instance:");  
        if (singleton1 == singleton2) {  
            System.out.println("singleton1 và singleton2 trả về CÙNG MỘT instance.");  
        } else {  
            System.out.println("singleton1 và singleton2 trả về CÁC instance KHÁC NHAU.");  
        }  
  
        System.out.println("--- Kết thúc ví dụ Singleton cơ bản ---");  
    }  
}
```

Giải thích chi tiết mã nguồn

- `private static BasicSingleton instance;`: Đây là biến tĩnh (`static`) và riêng tư (`private`) sẽ lưu trữ instance duy nhất của lớp `BasicSingleton`. Từ khóa `static` đảm bảo rằng biến này thuộc về lớp chứ không phải một đối tượng cụ thể nào, và `private` hạn chế việc truy cập trực tiếp từ bên ngoài lớp.
- `private BasicSingleton() { ... }`: Constructor được khai báo là `private`. Điều này là then chốt để ngăn chặn bất kỳ mã nào bên ngoài lớp tạo ra một instance mới của `BasicSingleton` bằng cách sử dụng toán tử `new`. Do đó, chỉ bốn thàn lớp mới có thể tạo ra instance của chính nó.
- `public static BasicSingleton getInstance() { ... }`: Đây là phương thức công khai (`public`) và tĩnh (`static`) mà qua đó toàn bộ ứng dụng sẽ lấy được instance của `BasicSingleton`. Bên trong phương thức này:
 - Nó kiểm tra xem instance đã được khởi tạo hay chưa (`if (instance == null)`).
 - Nếu chưa, nó sẽ tạo một instance mới (`instance = new BasicSingleton();`). Đây là cơ chế "Lazy Initialization" – instance chỉ được tạo khi có yêu cầu lần đầu tiên, giúp tiết kiệm tài nguyên nếu instance không bao giờ được sử dụng.
 - Cuối cùng, nó trả về instance duy nhất đó.

Vấn đề: Không Thread-Safe (Race Condition)

Cách cài đặt Singleton cơ bản này KHÔNG an toàn cho môi trường đa luồng (multi-threaded). Race condition có thể xảy ra trong các trường hợp sau:

- **Scenario Race Condition:** Giả sử có hai luồng (Thread A và Thread B) cùng gọi phương thức `getInstance()` tại cùng một thời điểm, và cả hai đều thấy `instance == null`.

Kết quả là, ứng dụng sẽ có hai instance khác nhau của `BasicSingleton`, vi phạm nguyên tắc cơ bản của Singleton Pattern (chỉ có một instance duy nhất). Điều này dẫn đến hành vi không nhất quán và khó dự đoán trong ứng dụng.

- a. **Thread A** kiểm tra `if (instance == null)`, thấy là `true`.
- b. Ngay trước khi Thread A kip tạo instance mới, **Thread B** cũng kiểm tra `if (instance == null)`, cũng thấy là `true`.
- c. **Thread A** tiếp tục và thực thi `instance = new BasicSingleton();`, tạo ra Instance 1.
- d. Sau đó, **Thread B** cũng tiếp tục và thực thi `instance = new BasicSingleton();`, tạo ra Instance 2, ghi đè lên Instance 1 (hoặc trả về Instance 2 cho Thread B).

Khi nào sử dụng cách cài đặt này?

- **Ứng dụng đơn luồng (Single-threaded applications):** Nếu bạn chắc chắn rằng ứng dụng của mình chỉ chạy trên một luồng duy nhất, cách này hoàn toàn an toàn và hiệu quả.
- **Mục đích học tập và hiểu concept:** Đây là điểm khởi đầu tuyệt vời để hiểu cơ chế hoạt động của Singleton Pattern trước khi tìm hiểu các cách triển khai phức tạp hơn, an toàn cho đa luồng.

Singleton: Eager Initialization (Thread-Safe)

Cách tiếp cận "Eager Initialization" tạo instance của lớp Singleton ngay khi class được tải vào bộ nhớ bởi ClassLoader. Phương pháp này đảm bảo tính thread-safe một cách tự nhiên mà không cần cơ chế đồng bộ hóa.

Mã nguồn Java (Eager Initialization)

```
public class EagerSingleton {  
  
    // 1. Private static final instance:  
    // Instance được khởi tạo NGAY LẬP TỨC khi class EagerSingleton được tải vào bộ nhớ.  
    // 'final' đảm bảo biến 'instance' sẽ chỉ được gán một lần duy nhất.  
    private static final EagerSingleton instance = new EagerSingleton();  
  
    // 2. Private constructor:  
    // Ngăn chặn việc tạo các instance mới của lớp này từ bên ngoài  
    // bằng cách sử dụng toán tử 'new'.  
    private EagerSingleton() {  
        System.out.println("EagerSingleton: Constructor được gọi - Instance duy nhất đã được tạo.");  
    }  
  
    // 3. Public static getInstance() method:  
    // Phương thức này cung cấp điểm truy cập toàn cục để lấy instance duy nhất đã được tạo sẵn.  
    public static EagerSingleton getInstance() {  
        return instance;  
    }  
  
    // Một phương thức ví dụ để minh họa chức năng của Singleton  
    public void showMessage() {  
        System.out.println("Xin chào từ EagerSingleton!");  
    }  
  
    // Ví dụ sử dụng trong phương thức main  
    public static void main(String[] args) {  
        System.out.println("--- Bắt đầu ví dụ EagerSingleton ---");  
  
        // Khi class EagerSingleton được tải, instance 'instance' đã được tạo.  
        // Lần gọi đầu tiên của getInstance() chỉ đơn giản là trả về instance đã có.  
        System.out.println("\nGọi getInstance() lần 1:");  
        EagerSingleton singleton1 = EagerSingleton.getInstance();  
        singleton1.showMessage();  
  
        // Lần gọi thứ hai cũng trả về cùng một instance đã tồn tại.  
        System.out.println("\nGọi getInstance() lần 2:");  
        EagerSingleton singleton2 = EagerSingleton.getInstance();  
        singleton2.showMessage();  
  
        // Kiểm tra xem cả hai biến có trả đến cùng một instance trong bộ nhớ không.  
        System.out.println("\nKiểm tra các instance:");  
        if (singleton1 == singleton2) {  
            System.out.println("singleton1 và singleton2 trả đến CÙNG MỘT instance.");  
        } else {  
            System.out.println("singleton1 và singleton2 trả đến CÁC instance KHÁC NHAU. (Lỗi trong triển khai Eager Singleton!)");  
        }  
  
        System.out.println("--- Kết thúc ví dụ EagerSingleton ---");  
    }  
}
```

Loading Initialization

%

Mare Loading Initialization Ready

Giải thích chi tiết

- Tạo Instance Ngay Lập Tức:** Thay vì tạo instance khi được yêu cầu lần đầu (Lazy Initialization), Eager Singleton khởi tạo biến instance ngay tại thời điểm khai báo. Điều này xảy ra khi lớp EagerSingleton được tải vào bộ nhớ bởi ClassLoader của JVM.
- Thread-Safe Tự Nhiên:** JVM đảm bảo rằng quá trình tải và khởi tạo một class là thread-safe. Điều này có nghĩa là, khi class EagerSingleton được tải, instance instance sẽ được tạo hoàn chỉnh trước khi bất kỳ luồng nào có thể truy cập vào nó thông qua phương thức getInstance(). Do đó, không có "race condition" nào có thể xảy ra, và không cần sử dụng các từ khóa đồng bộ hóa (như synchronized).
- Constructor Riêng Tư:** Tương tự như các cách triển khai Singleton khác, constructor được khai báo là private để ngăn chặn việc tạo instance mới từ bên ngoài lớp.
- Phương Thức getInstance():** Phương thức này chỉ đơn giản là trả về instance instance đã được tạo sẵn.

Ưu điểm

- Đơn giản và dễ hiểu:** Đây là cách triển khai Singleton đơn giản nhất để đảm bảo thread-safe.
- Thread-Safe tự nhiên:** Không cần lo lắng về vấn đề đồng bộ hóa trong môi trường đa luồng, vì instance được tạo trước khi có bất kỳ sự cạnh tranh nào.
- Không có overhead của synchronization:** Vì không sử dụng synchronized, không có chi phí hiệu suất nào liên quan đến việc khóa và mở khóa luồng.

Nhược điểm

- Không hỗ trợ Lazy Loading:** Instance được tạo ngay khi class được tải, bất kể ứng dụng có cần đến nó hay không. Điều này có thể dẫn đến lãng phí tài nguyên (bộ nhớ và CPU) nếu instance lớn và không bao giờ được sử dụng.
- Không linh hoạt:** Không thể truyền tham số cho constructor để khởi tạo instance.

Khi nào sử dụng cách cài đặt này?

- Instance có kích thước nhỏ:** Nếu instance của Singleton không tiêu tốn nhiều tài nguyên (bộ nhớ, thời gian khởi tạo), thì việc tạo nó ngay lập tức không gây ra vấn đề đáng kể.
- Chắc chắn sẽ sử dụng instance:** Nếu bạn biết chắc chắn rằng instance của Singleton sẽ được sử dụng trong suốt vòng đời của ứng dụng, thì việc tạo sớm sẽ không có nhược điểm về lãng phí tài nguyên.
- Cần một giải pháp thread-safe đơn giản:** Khi sự đơn giản và tính thread-safe là ưu tiên hàng đầu, và nhược điểm về Lazy Loading không quá quan trọng.

Singleton: Lazy Initialization với Synchronized

Tạo instance của lớp Singleton chỉ khi cần thiết (lazy loading) và đảm bảo an toàn luồng (thread-safe) bằng cách sử dụng từ khóa synchronized trên phương thức getInstance().

Mã nguồn Java (Lazy Initialization với Synchronized)

```
public class LazySynchronizedSingleton {  
  
    // 1. Private static instance:  
    // Instance chưa được khởi tạo ngay, để null.  
    // Instance chỉ được tạo khi phương thức getInstance() được gọi lần đầu tiên.  
    private static LazySynchronizedSingleton instance;  
  
    // 2. Private constructor:  
    // Ngăn chặn việc tạo các instance mới của lớp này từ bên ngoài  
    // bằng cách sử dụng toán tử 'new'.  
    private LazySynchronizedSingleton() {  
        System.out.println("LazySynchronizedSingleton: Constructor được gọi - Instance duy nhất đã được tạo.");  
    }  
  
    // 3. Public static synchronized getInstance() method:  
    // Phương thức này cung cấp điểm truy cập toàn cục để lấy instance duy nhất.  
    // Từ khóa 'synchronized' đảm bảo rằng chỉ có MỘT luồng (thread) có thể thực thi  
    // phương thức này tại một thời điểm, do đó đảm bảo tính thread-safe.  
    public static synchronized LazySynchronizedSingleton getInstance() {  
        // Kiểm tra xem instance đã được tạo chưa.  
        if (instance == null) {  
            // Nếu chưa, tạo instance mới.  
            // Đoạn code này chỉ được thực thi bởi một luồng duy nhất tại một thời điểm  
            // nhờ vào từ khóa 'synchronized'.  
            instance = new LazySynchronizedSingleton();  
        }  
        // Trả về instance đã có hoặc vừa được tạo.  
        return instance;  
    }  
  
    // Một phương thức ví dụ để minh họa chức năng của Singleton  
    public void showMessage() {  
        System.out.println("Xin chào từ LazySynchronizedSingleton!");  
    }  
  
    // Ví dụ sử dụng trong phương thức main  
    public static void main(String[] args) {  
        System.out.println("--- Bắt đầu ví dụ LazySynchronizedSingleton ---");  
  
        // Instance chưa được tạo ở đây. Nó chỉ được tạo khi getInstance() được gọi.  
        System.out.println("Trước khi gọi getInstance(), instance chưa được tạo.");  
  
        // Lần gọi đầu tiên của getInstance() sẽ tạo instance.  
        System.out.println("\nGọi getInstance() lần 1:");  
        LazySynchronizedSingleton singleton1 = LazySynchronizedSingleton.getInstance();  
        singleton1.showMessage();  
  
        // Lần gọi thứ hai sẽ trả về cùng một instance đã tồn tại, không tạo mới.  
        System.out.println("\nGọi getInstance() lần 2:");  
        LazySynchronizedSingleton singleton2 = LazySynchronizedSingleton.getInstance();  
        singleton2.showMessage();  
  
        // Kiểm tra xem cả hai biến có trả đến cùng một instance trong bộ nhớ không.  
        System.out.println("\nKiểm tra các instance:");  
        if (singleton1 == singleton2) {  
            System.out.println("singleton1 và singleton2 trả đến CÙNG MỘT instance.");  
        } else {  
            System.out.println("singleton1 và singleton2 trả đến CÁC instance KHÁC NHAU. (Lỗi trong triển khai Singleton!)");  
        }  
  
        System.out.println("--- Kết thúc ví dụ LazySynchronizedSingleton ---");  
    }  
}
```

Giải thích chi tiết

- Lazy Loading:** Instance của lớp LazySynchronizedSingleton chỉ được tạo ra khi phương thức getInstance() được gọi lần đầu tiên. Trước đó, biến instance sẽ có giá trị là null. Điều này giúp tiết kiệm tài nguyên nếu instance không bao giờ được sử dụng trong suốt vòng đời của ứng dụng.
- Synchronized:** Từ khóa synchronized được áp dụng cho phương thức getInstance(). Điều này có nghĩa là tại bất kỳ thời điểm nào, chỉ có duy nhất một luồng có thể truy cập và thực thi đoạn code bên trong phương thức getInstance(). Các luồng khác sẽ phải đợi nếu có một luồng khác đang thực thi phương thức này.
- Thread-Safe:** Nhờ có cơ chế synchronized, chúng ta đảm bảo rằng khi nhiều luồng cố gắng gọi getInstance() cùng một lúc, sẽ không có trường hợp hai luồng khác nhau cùng đi qua điều kiện if (instance == null) và cùng tạo ra hai instance riêng biệt. Điều này đảm bảo rằng dù trong môi trường đa luồng, chỉ có một instance duy nhất của Singleton được tạo ra.

Ưu điểm

- Lazy Loading:** Instance chỉ được tạo khi thực sự cần thiết, giúp tiết kiệm bộ nhớ và tài nguyên khởi tạo nếu instance không bao giờ được sử dụng.
- Thread-Safe:** Đảm bảo an toàn khi hoạt động trong môi trường đa luồng nhờ vào từ khóa synchronized, ngăn chặn việc tạo ra nhiều instance.
- Đơn giản và dễ hiểu:** Cách triển khai này tương đối dễ hiểu và cài đặt, là một trong những giải pháp đầu tiên được nghĩ đến khi cần Lazy Loading và Thread-Safe.

Nhược điểm

- Performance Overhead:** Mỗi lần phương thức getInstance() được gọi, luồng sẽ phải trải qua quá trình khóa và mở khóa (locking/unlocking) do từ khóa synchronized. Điều này gây ra một chi phí hiệu suất đáng kể, đặc biệt trong các ứng dụng có tần suất truy cập cao.
- Chậm hơn Eager Initialization:** Do chi phí đóng bộ hóa, phương thức getInstance() có thể chậm hơn đáng kể so với Eager Initialization (nếu so sánh các lần gọi sau lần đầu tiên).
- Không cần thiết synchronized sau lần đầu:** Sau khi instance đã được tạo lần đầu tiên, tất cả các lần gọi getInstance() tiếp theo chỉ đơn giản là trả về instance đã có. Việc đóng bộ hóa cho các lần gọi này là không cần thiết và gây lãng phí tài nguyên.

Khi nào sử dụng cách cài đặt này?

- Cần Lazy Loading:** Khi bạn muốn instance chỉ được tạo khi nó thực sự được yêu cầu, chứ không phải ngay khi ứng dụng khởi động.
- Instance tồn nhiều tài nguyên:** Nếu việc khởi tạo instance của Singleton tiêu tốn nhiều bộ nhớ hoặc thời gian xử lý, Lazy Loading sẽ giúp hoãn việc này cho đến khi cần.
- Không quan tâm performance của getInstance():** Trong các ứng dụng mà phương thức getInstance() không được gọi quá thường xuyên hoặc hiệu suất của nó không phải là yếu tố quan trọng.
- Ứng dụng multi-threaded đơn giản:** Phù hợp với các ứng dụng đa luồng có yêu cầu đồng bộ hóa đơn giản, không quá phức tạp về mặt hiệu suất.

Singleton: Double-Checked Locking (Tối Ưu)

Double-Checked Locking là một mẫu thiết kế Singleton tối ưu, kết hợp ưu điểm của lazy loading, thread-safety và hiệu năng cao. Nó chỉ thực hiện đồng bộ hóa (synchronized) trong lần đầu tiên instance được tạo, giảm thiểu chi phí hiệu năng cho các lần truy cập sau.

Mã nguồn Java (Double-Checked Locking)

```
public class LazyDoubleCheckedSingleton {  
  
    // 1. Private static volatile instance:  
    // Sử dụng từ khóa 'volatile' là cực kỳ quan trọng ở đây.  
    // - Đảm bảo rằng biến 'instance' luôn được đọc từ bộ nhớ chính (main memory)  
    // thay vì từ bộ nhớ cache của CPU cho mỗi luồng (thread). Điều này giúp đảm bảo  
    // tính hiển thị (visibility) của các thay đổi giữa các luồng.  
    // - Ngăn chặn việc sắp xếp lại lệnh (instruction reordering) của JVM và CPU.  
    // Nếu không có 'volatile', một luồng có thể nhìn thấy 'instance' không null  
    // nhưng chưa được khởi tạo hoàn chỉnh, dẫn đến lỗi.  
    private static volatile LazyDoubleCheckedSingleton instance;  
  
    // 2. Private constructor:  
    // Ngăn chặn việc tạo các instance mới của lớp này từ bên ngoài  
    // bằng cách sử dụng toán tử 'new'.  
    private LazyDoubleCheckedSingleton() {  
        // Kiểm tra để đảm bảo rằng constructor không bị gọi lại thông qua Reflection API  
        if (instance != null) {  
            throw new IllegalStateException("Đã có một instance khác được tạo rồi.");  
        }  
        System.out.println("LazyDoubleCheckedSingleton: Constructor được gọi - Instance duy nhất đã được tạo.");  
    }  
  
    // 3. Public static getInstance() method với Double-Checked Locking:  
    // Phương thức này cung cấp điểm truy cập toàn cục để lấy instance duy nhất.  
    // Nó đảm bảo tính lazy loading, thread-safe và hiệu năng cao.  
    public static LazyDoubleCheckedSingleton getInstance() {  
        // Kiểm tra lần 1 (Check 1): Không cần khóa nếu instance đã được tạo.  
        // Hầu hết các lời gọi đến getInstance() sẽ vượt qua kiểm tra này và trả về  
        // instance mà không cần đi vào khối synchronized, giúp tối ưu hiệu năng.  
        if (instance == null) {  
            // Khóa đối tượng Class để đảm bảo chỉ có MỘT luồng đi vào khối này  
            // tại một thời điểm khi 'instance' chưa được tạo.  
            synchronized (LazyDoubleCheckedSingleton.class) {  
                // Kiểm tra lần 2 (Check 2): Kiểm tra lại xem instance có null không.  
                // Điều này là cần thiết vì nhiều luồng có thể đã vượt qua Check 1  
                // và đang chờ để vào khối synchronized. Luồng đầu tiên sẽ tạo instance,  
                // các luồng sau đó cần kiểm tra lại để không tạo instance thứ hai.  
                if (instance == null) {  
                    // Tại đây, instance mới được tạo.  
                    // Nhờ 'volatile', quá trình khởi tạo này sẽ được thực hiện hoàn chỉnh  
                    // trước khi 'instance' được gán giá trị không null và hiển thị cho các luồng khác.  
                    instance = new LazyDoubleCheckedSingleton();  
                }  
            }  
        }  
        // Trả về instance đã có hoặc vừa được tạo.  
        return instance;  
    }  
  
    // Một phương thức ví dụ để minh họa chức năng của Singleton  
    public void showMessage() {  
        System.out.println("Xin chào từ LazyDoubleCheckedSingleton!");  
    }  
  
    // Ví dụ sử dụng trong phương thức main  
    public static void main(String[] args) {  
        System.out.println("--- Bắt đầu ví dụ LazyDoubleCheckedSingleton ---");  
  
        // Instance chưa được tạo ở đây. Nó chỉ được tạo khi getInstance() được gọi.  
        System.out.println("Trước khi gọi getInstance(), instance chưa được tạo.");  
  
        // Lần gọi đầu tiên của getInstance() sẽ tạo instance.  
        System.out.println("\nGọi getInstance() lần 1:");  
        LazyDoubleCheckedSingleton singleton1 = LazyDoubleCheckedSingleton.getInstance();  
        singleton1.showMessage();  
  
        // Lần gọi thứ hai sẽ trả về cùng một instance đã tồn tại, không tạo mới.  
        System.out.println("\nGọi getInstance() lần 2:");  
        LazyDoubleCheckedSingleton singleton2 = LazyDoubleCheckedSingleton.getInstance();  
        singleton2.showMessage();  
  
        // Kiểm tra xem cả hai biến có trả về cùng một instance trong bộ nhớ không.  
        System.out.println("\nKiểm tra các instance:");  
        if (singleton1 == singleton2) {  
            System.out.println("singleton1 và singleton2 trả về CÙNG MỘT instance.");  
        } else {  
            System.out.println("singleton1 và singleton2 trả về CÁC instance KHÁC NHAU. (Lỗi trong triển khai Singleton!)");  
        }  
  
        System.out.println("--- Kết thúc ví dụ LazyDoubleCheckedSingleton ---");  
    }  
}
```

Giải thích chi tiết Double-Checked Locking

- Kiểm tra lần 1 (Outside Synchronized):** Khi phương thức getInstance() được gọi, nó sẽ kiểm tra xem instance đã được khởi tạo chưa (`if (instance == null)`). Nếu instance đã tồn tại, phương thức sẽ trả về ngay lập tức mà không cần bất kỳ khóa (lock) nào. Đây là yếu tố then chốt giúp tối ưu hiệu năng.
- Khối Synchronized:** Nếu instance là null, luồng sẽ đi vào một khối synchronized. Khối này đảm bảo rằng tại một thời điểm chỉ có duy nhất một luồng có thể truy cập vào đoạn code bên trong để tạo instance. Chúng ta khóa trên đối tượng Class (LazyDoubleCheckedSingleton.class) để có một khóa duy nhất cho toàn bộ lớp.
- Kiểm tra lần 2 (Inside Synchronized):** Ngay sau khi vào khối synchronized, luồng sẽ kiểm tra lại `if (instance == null)`. Kiểm tra thứ hai này là cực kỳ quan trọng. Nó xử lý trường hợp hai luồng cùng lúc vượt qua kiểm tra đầu tiên (Check 1) và cùng chờ đợi để vào khối synchronized. Khi luồng đầu tiên thoát khỏi khối synchronized sau khi đã tạo instance, luồng thứ hai sẽ vào. Nếu không có Check 2, luồng thứ hai sẽ tạo thêm một instance nữa, phá vỡ nguyên tắc Singleton.

Tại sao cần từ khóa volatile?

Từ khóa `volatile` trên biến `instance` là bắt buộc để Double-Checked Locking hoạt động đúng và an toàn trong Java Memory Model (JMM).

- Ngăn chặn sắp xếp lại lệnh (Instruction Reordering):** Quá trình tạo một đối tượng trong Java bao gồm 3 bước:

Tuy nhiên, JVM hoặc CPU có thể tối ưu và sắp xếp lại thứ tự bước 2 và 3 (thực hiện 1 > 3 > 2). Nếu một luồng khác truy cập `instance` sau bước 3 nhưng trước bước 2, nó sẽ thấy `instance` không phải `null` nhưng đối tượng chưa được khởi tạo hoàn chỉnh, dẫn đến lỗi `NullPointerException` hoặc dữ liệu không nhất quán. Từ khóa `volatile` ngăn chặn việc sắp xếp lại này, đảm bảo thứ tự 1 > 2 > 3.

- Cấp phát bộ nhớ cho đối tượng mới.
 - Gọi constructor để khởi tạo các trường của đối tượng.
 - Gán tham chiếu của đối tượng vào biến `instance`.
- Đảm bảo tính hiển thị (Visibility):** `volatile` đảm bảo rằng mọi thay đổi đối với biến `instance` (ví dụ, khi nó được gán một đối tượng mới) sẽ ngay lập tức được ghi vào bộ nhớ chính và hiển thị cho tất cả các luồng khác. Nếu không có `volatile`, một luồng có thể đọc một giá trị cũ của `instance` từ bộ nhớ cache của nó.

Ưu điểm

- Lazy Loading:** Instance chỉ được tạo khi thực sự cần thiết, tiết kiệm tài nguyên.
- Thread-Safe:** Đảm bảo chỉ có một instance duy nhất được tạo ra ngay cả trong môi trường đa luồng.
- Hiệu năng cao:** Khối synchronized chỉ được sử dụng trong lần đầu tiên instance được tạo. Các lần gọi `getInstance()` sau đó sẽ không cần đồng bộ hóa, mang lại hiệu năng tương đương với Eager Initialization cho hầu hết các trường hợp.
- Best Practice:** Được coi là một trong những cách triển khai Singleton hiệu quả và an toàn nhất trong Java cho các ứng dụng thực tế.

Nhược điểm

- Phức tạp hơn:** Cần sự hiểu biết sâu sắc về Java Memory Model và các vấn đề về đa luồng để triển khai đúng và hiệu quả.
- Khó Debug:** Các lỗi liên quan đến sắp xếp lại lệnh hoặc hiển thị có thể khó phát hiện và debug.

Khi nào sử dụng cách cài đặt này?

- Ứng dụng sản xuất (Production Applications):** Khi bạn cần một Singleton vừa thread-safe, vừa lazy loading, và đặc biệt là yêu cầu hiệu năng cao trong môi trường đa luồng.
- Instance tồn nhiều tài nguyên:** Nếu việc khởi tạo instance của Singleton tiêu tốn nhiều bộ nhớ hoặc thời gian xử lý, Double-Checked Locking giúp hoãn việc này cho đến khi cần và đảm bảo nó chỉ xảy ra một lần.
- Môi trường đa luồng:** Bất cứ khi nào bạn cần Singleton trong ứng dụng đa luồng và muốn cân bằng giữa an toàn luồng và hiệu suất.

Singleton: Cảnh Báo Lạm Dụng

Đừng Lạm Dụng Singleton!

Singleton tuy tiện nhưng cũng có nhiều nhược điểm. Hãy cẩn thận khi sử dụng.

✗ KHÔNG NÊN dùng khi:

Có thể dùng DI

Dependency Injection linh hoạt hơn

Cần test

Singleton khó mock và test

Cần nhiều instances

Singleton chỉ có 1 instance

✓ NÊN dùng khi:

Thực sự cần 1 instance

Logger, Configuration, Cache

Resource tồn kém

Database connection pool

Global state hợp lý

Application settings

⚖ Nhược điểm nghiêm trọng:

- Kẻ thù của Unit Test:** Global state gây khó khăn cho testing
- Giấu dependencies:** Class dùng Singleton.getInstance() không khai báo dependency rõ ràng
- Vi phạm SRP:** Class vừa quản lý business logic vừa quản lý lifecycle của chính nó

💡 Lời khuyên:

Trước khi dùng Singleton, hãy tự hỏi: "Tôi có thể dùng Dependency Injection thay thế không?" Nếu có thể, hãy dùng DI!

Factory Pattern - Tạo Object Linh Hoạt

Mục đích: Cung cấp interface để tạo object mà không cần chỉ định class cụ thể. Client không biết class nào được tạo, chỉ biết interface.

Khi nào dùng Factory?

01

Không biết trước loại object

Quyết định tại runtime dựa vào input

02

Tách logic tạo object

Không để client trực tiếp dùng "new"

03

Object phức tạp

Quá trình khởi tạo có nhiều bước

Lợi ích của Factory Pattern

- Tách biệt creation và usage
- Dễ thêm loại mới (OCP)
- Client không phụ thuộc concrete class (DIP)
- Code dễ bảo trì

Ví dụ thực tế:

Bạn vào quán cafe gọi "Một ly cà phê". Bạn không cần biết máy pha là loại nào, hạt cà phê từ đâu. Bạn chỉ cần interface "cà phê", còn Factory (nhân viên) lo phần còn lại.

Simple Factory Pattern

Cách đơn giản nhất của Factory Pattern: Một class chứa static method để tạo objects dựa trên tham số.

```
// 1. Interface
public interface Animal {
    void makeSound();
}

// 2. Implementations
public class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof! Woof!");
    }
}

public class Cat implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow! Meow!");
    }
}

public class Cow implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Moo! Moo!");
    }
}

// 3. Simple 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 if (type.equalsIgnoreCase("cow")) {
            return new Cow();
        }
        throw new IllegalArgumentException("Unknown animal type: " + type);
    }
}
```

Sử dụng:

```
// Client code
Animal dog = AnimalFactory
    .createAnimal("dog");
dog.makeSound(); // "Woof! Woof!"

Animal cat = AnimalFactory
    .createAnimal("cat");
cat.makeSound(); // "Meow! Meow!"
```

Lợi ích:

- Client không dùng "new"
- Dễ thêm loại mới
- Logic tạo object tập trung

Factory Method Pattern

Phiên bản nâng cao: Mỗi loại có một Factory riêng. Abstract Factory định nghĩa interface, các Concrete Factory override để tạo object cụ thể.

```
// Abstract Factory
public abstract class AnimalFactory {
    // Factory Method
    public abstract Animal
        createAnimal();

    // Template method
    public void displayAnimal() {
        Animal animal =
            createAnimal();
        animal.makeSound();
    }
}
```

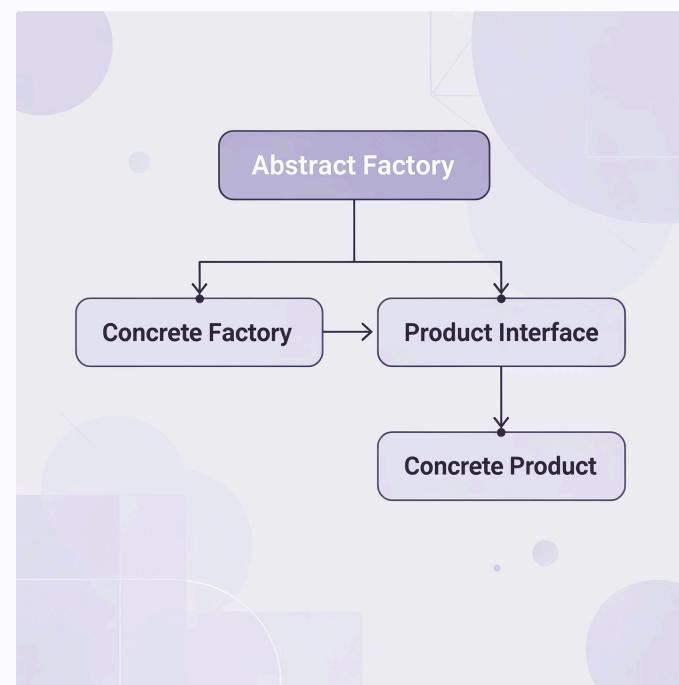
```
// Concrete Factory 1
public class DogFactory
    extends AnimalFactory {
    @Override
    public Animal createAnimal() {
        return new Dog();
    }
}
```

```
// Concrete Factory 2
public class CatFactory
    extends AnimalFactory {
    @Override
    public Animal createAnimal() {
        return new Cat();
    }
}
```

Sử dụng:

```
// Tạo Factory
AnimalFactory factory =
    new DogFactory();
factory.displayAnimal();
// "Woof! Woof!"

factory = new CatFactory();
factory.displayAnimal();
// "Meow! Meow!"
```



So với Simple Factory:

- Tuân thủ OCP hơn
- Mỗi Factory một trách nhiệm
- Phức tạp hơn (nhiều class)

Factory Pattern Kết Hợp DIP

Ví dụ thực tế: Hệ thống thanh toán với nhiều database backends, sử dụng Factory để tạo và DIP để inject.

```
// 1. Interface (Abstraction)
public interface Database {
    void connect();
    void save(String data);
}

// 2. Implementations
public class MySQLDatabase implements Database {
    @Override
    public void connect() {
        System.out.println("Connecting to MySQL...");
    }

    @Override
    public void save(String data) {
        System.out.println("Saving to MySQL: " + data);
    }
}

public class PostgreSQLDatabase implements Database {
    @Override
    public void connect() {
        System.out.println("Connecting to PostgreSQL...");
    }

    @Override
    public void save(String data) {
        System.out.println("Saving to PostgreSQL: " + data);
    }
}

// 3. Factory
public class DatabaseFactory {
    public static Database createDatabase(String type) {
        if (type.equalsIgnoreCase("mysql")) {
            return new MySQLDatabase();
        } else if (type.equalsIgnoreCase("postgres")) {
            return new PostgreSQLDatabase();
        }
        throw new IllegalArgumentException("Unknown database: " + type);
    }
}
```

Tóm Tắt Chương 4

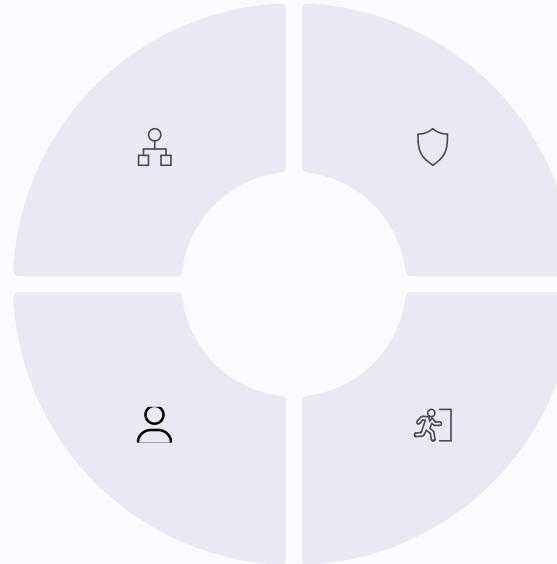
Những Điều Quan Trọng Bạn Đã Học

UML Class Diagram

Đọc và vẽ sơ đồ lớp cơ bản, hiểu các quan hệ: Association, Aggregation, Composition, Inheritance

Design Patterns

Implement Singleton và Factory Pattern, biết khi nào dùng và khi nào không nên dùng



SOLID Principles

Nắm vững SRP, OCP, DIP - ba nguyên lý nền tảng của thiết kế phần mềm chuyên nghiệp

Dependency Inversion

Phụ thuộc vào abstraction, sử dụng Dependency Injection để tạo code linh hoạt

Kỹ năng bạn đã có:

- Đọc và vẽ UML Class Diagram
- Nhận biết code vi phạm SOLID
- Refactor code tuân thủ nguyên lý thiết kế
- Áp dụng DIP trong thiết kế
- Implement Singleton thread-safe
- Implement Factory Pattern
- Kết hợp patterns với SOLID
- Viết code dễ bảo trì và mở rộng

Bài Tập Chương 4

Các bài tập sau giúp bạn củng cố kiến thức và rèn luyện kỹ năng áp dụng Design Patterns và SOLID Principles.

1

UML Class Diagram

Vẽ UML cho hệ thống quản lý sinh viên và môn học

2

Single Responsibility

Refactor class User vi phạm SRP

3

Open/Closed Principle

Refactor AreaCalculator tuân thủ OCP

4

Dependency Inversion

Refactor PaymentService với DIP và DI

5

Singleton Pattern

Tạo Logger thread-safe với Singleton

6

Factory Pattern

Tạo hệ thống Payment với Factory

7

Tổng hợp

Xây dựng hệ thống File Storage hoàn chỉnh

Bài 1: UML Class Diagram

📝 Yêu cầu:

Vẽ UML Class Diagram cho hệ thống Quản lý Sinh viên và Môn học với các yêu cầu sau:

Class Student

- **Fields (private):**
- - id: String
- - name: String
- - age: int
- **Methods (public):**
- + getters/setters
- + displayInfo(): void

Class Course

- **Fields (private):**
- - code: String
- - name: String
- - credits: int
- **Methods (public):**
- + getters/setters
- + getCourseInfo(): String

Class Enrollment

- **Fields (private):**
- - student: Student
- - course: Course
- - grade: double
- **Methods (public):**
- + getters/setters
- + calculateGPA(): double

Quan hệ:

- Student và Course có quan hệ **Association** qua Enrollment
- Enrollment có **Composition** với Student và Course (không tồn tại độc lập)

 **Gợi ý:** Sử dụng ký hiệu đúng cho từng loại quan hệ. Đánh dấu rõ access modifiers (+, -, #).

Bài 2: Single Responsibility Principle

Yêu cầu:

Refactor class User sau để tuân thủ SRP. Giải thích tại sao refactor và lợi ích của việc tách class.

Code cần refactor:

```
public class User {  
    private String name;  
    private String email;  
    private String password;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public void setEmail(String email) {  
        this.email = email;  
    }  
  
    // Lưu vào database  
    public void saveToDatabase() {  
        // Connect to database  
        // SQL INSERT...  
        System.out.println(  
            "Saving user to database");  
    }  
  
    // Gửi email  
    public void sendEmail(String message) {  
        // Connect to SMTP  
        // Send email...  
        System.out.println(  
            "Sending email: " + message);  
    }  
  
    // Tạo report  
    public void generateReport() {  
        // Create PDF  
        System.out.println(  
            "Generating user report");  
    }  
}
```

Hướng dẫn:

01

Xác định trách nhiệm

Liệt kê các trách nhiệm của class User

02

Tách class

Tạo class riêng cho mỗi trách nhiệm

03

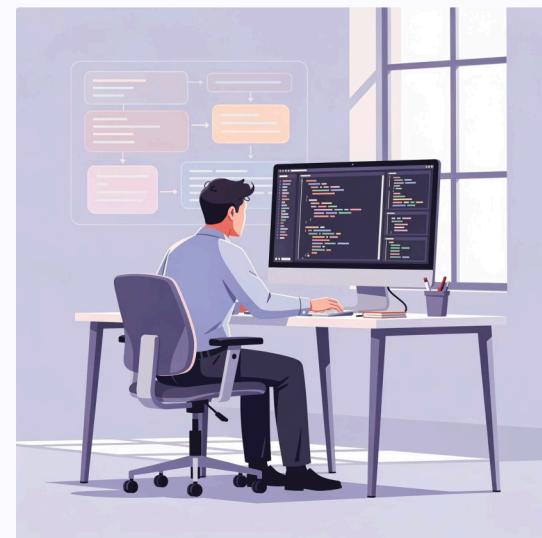
Viết lại code

Implement các class mới

04

Giải thích

Nêu lý do và lợi ích



Bài 3: Open/Closed Principle

📝 Yêu cầu:

Refactor AreaCalculator để tuân thủ OCP. Sau đó thêm class Triangle mà không cần sửa AreaCalculator.

```
// Code cần refactor
public class AreaCalculator {
    public double calculateArea(Object shape) {
        if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return Math.PI * c.getRadius() * c.getRadius();
        }
        else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return r.getWidth() * r.getHeight();
        }
        return 0;
    }
}

// Classes có sẵn
public class Circle {
    private double radius;
    public Circle(double radius) { this.radius = radius; }
    public double getRadius() { return radius; }
}

public class Rectangle {
    private double width, height;
    public Rectangle(double w, double h) {
        this.width = w;
        this.height = h;
    }
    public double getWidth() { return width; }
    public double getHeight() { return height; }
}
```

Các bước thực hiện:

1. Tạo abstract class hoặc interface **Shape** với method calculateArea()
2. Refactor Circle và Rectangle để extend/implement Shape
3. Refactor AreaCalculator để nhận Shape và gọi polymorphic method
4. Tạo class Triangle mới mà **KHÔNG** sửa AreaCalculator

Bài 4: Dependency Inversion Principle

Yêu cầu:

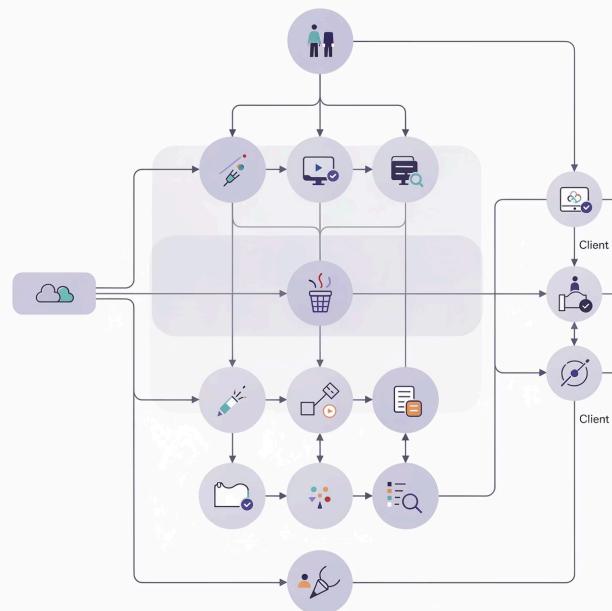
Refactor PaymentService để tuân thủ DIP và sử dụng Dependency Injection.

Code cần refactor:

```
public class MySQLDatabase {  
    public void save(String data) {  
        System.out.println(  
            "MySQL: " + data);  
    }  
  
    public class PaymentService {  
        private MySQLDatabase database;  
  
        public PaymentService() {  
            // Vi phạm DIP  
            this.database =  
                new MySQLDatabase();  
        }  
  
        public void processPayment(  
            double amount) {  
            String data = "Payment: "  
                + amount;  
            database.save(data);  
        }  
    }  
}
```

Yêu cầu chi tiết:

1. Tạo **interface Database** với method `save(String)`
2. Refactor MySQLDatabase để implement Database
3. Tạo class **PostgreSQLDatabase** cũng implement Database
4. Refactor PaymentService:
 - Phụ thuộc vào interface Database
 - Dùng Constructor Injection
5. Viết code test với cả MySQL và PostgreSQL



Bài 5: Singleton Pattern

📝 Yêu cầu:

Tạo class Logger sử dụng Singleton Pattern với các yêu cầu sau:

1

Thread-safe implementation

Sử dụng một trong các cách: Eager Initialization, Synchronized, hoặc Double-Checked Locking

2

Method log(String message)

Ghi log ra console với timestamp: [2024-01-15 10:30:45] Message

3

Method getInstance()

Trả về instance duy nhất của Logger

4

Test code

Viết code kiểm tra hai instance có giống nhau không

Code test mẫu:

```
Logger logger1 = Logger.getInstance();
Logger logger2 = Logger.getInstance();

// Kiểm tra singleton
System.out.println(logger1 == logger2); // Phải in ra: true

// Sử dụng
logger1.log("Application started");
logger2.log("User logged in");
logger1.log("Transaction completed");
```

💡 **Bonus:** Thêm method logError(String message) in ra màu đỏ và logWarning(String message) in ra màu vàng.

Bài 6: Factory Pattern

Yêu cầu:

Tạo hệ thống Payment với Factory Pattern gồm các thành phần sau:

Interface PaymentMethod

Method: pay(double amount)

Factory PaymentFactory

Method: createPaymentMethod(String type)

Implementations

CreditCard, PayPal, Bitcoin - mỗi class implement theo cách riêng

Service PaymentService

Method: processPayment(PaymentMethod, double)

Chi tiết implementations:

- **CreditCard:** In "Processing credit card payment: \$X.XX"
- **PayPal:** In "Processing PayPal payment: \$X.XX"
- **Bitcoin:** In "Processing Bitcoin payment: X BTC"

Code test mẫu:

```
PaymentMethod credit =  
PaymentFactory  
.createPaymentMethod(  
"credit");
```

```
PaymentService service =  
new PaymentService();
```

```
service.processPayment(  
credit, 100.0);
```

Bài 7: Tổng Hợp - Hệ Thống File Storage

Yêu cầu:

Xây dựng hệ thống quản lý file hoàn chỉnh áp dụng tất cả kiến thức đã học. Đây là bài tập tổng hợp kiểm tra khả năng kết hợp SOLID và Design Patterns.

1 Interface FileStorage

Methods: `save(String data, String filename)`, `load(String filename): String`

2 Implementations

LocalFileStorage: Lưu vào ổ cứng local. **CloudFileStorage:** Lưu lên cloud (giả lập bằng print)

3 Factory StorageFactory

Tạo storage dựa trên config từ Singleton Config

4 Class FileManager

Phụ thuộc vào FileStorage (DIP). Methods: `saveFile(String, String)`, `loadFile(String)`

5 Singleton Config

Lưu cấu hình: storage type ("local" hoặc "cloud"), thread-safe

Yêu cầu đặc biệt:

- Tuân thủ SOLID (S, O, D)
- Sử dụng Singleton cho Config
- Sử dụng Factory cho Storage
- Dependency Injection cho FileManager
- JavaDoc đầy đủ cho tất cả classes

Tài Liệu Tham Khảo & Kết Thúc

Chúc Mừng!



Bạn đã hoàn thành Chương 4 - một trong những chương quan trọng nhất trong thiết kế phần mềm. Hãy tiếp tục luyện tập và áp dụng vào dự án thực tế!

Tài liệu tham khảo

SOLID Principles

[Wikipedia: SOLID](#)

Giải thích chi tiết về 5 nguyên lý SOLID

Design Patterns Book

"Head First Design Patterns" - Eric Freeman

Sách dạy Design Patterns dễ hiểu nhất

Refactoring

"Refactoring" - Martin Fowler

Kỹ thuật cải thiện code chất lượng

Bước tiếp theo

01

Làm bài tập

Hoàn thành tất cả 7 bài tập trong chương

02

Áp dụng vào dự án

Refactor code cũ theo SOLID và Design Patterns

03

Học thêm patterns

Observer, Strategy, Decorator, etc.

04

Chia sẻ kiến thức

Dạy lại cho người khác để hiểu sâu hơn

Chúc bạn học tốt!