UNIVERSITY OF INFORMATION TECHNOLOGY

IE303
JAVA TECHNOLOGY
DESIGN PATTERNS

Lecturer: MSc. Kiet Van Nguyen

Department of Data Science

Faculty of Information Science and Engineering

About me!

- Name: Kiet Van Nguyen.
- Email: kietnv@uit.edu.vn
- Department: Data Science, ISE, UIT.
- Topics:
  - Subjects: Java Technology, Methodology of Scientific Research, Social Media Mining, Natural Language Processing, Programming Languages (Java, Python, Prolog, ..), Intro. to Information Technology, …
  - Research Interests: Natural Language Processing and Data Mining.

Tại sao nên học design patterns?

- Các mẫu thiết kế là một bộ công cụ gồm các **giải pháp đã** được **thử nghiệm** và kiểm tra cho các **vấn đề phổ biến** trong thiết kế phần mềm.
- Các mẫu thiết kế xác định một **ngôn ngữ chung** mà bạn và đồng đội của mình có thể sử dụng để **giao tiếp hiệu quả** hơn.

Lịch sử của design patterns

- Khái niệm về các patterns được Christopher Alexander mô tả lần đầu tiên trong **A Pattern Language: Towns, Buildings, Construction**.
- Năm 1994, Erich Gamma, John Vlissides, Ralph Johnson và Richard Helm xuất bản **Design Patterns: Elements of Reusable Object-Oriented Software**.

Design pattern (mẫu thiết kế) là gì?

Các mẫu thiết kế là **giải pháp** điển hình cho các **vấn đề** thường xảy ra trong **thiết kế phần mềm**. Chúng giống như các bản thiết kế được tạo sẵn mà bạn có thể tùy chỉnh để giải quyết vấn đề thiết kế định kỳ trong mã của mình.

Một pattern bao gồm những gì?

Dưới đây là các phần thường có trong một pattern:

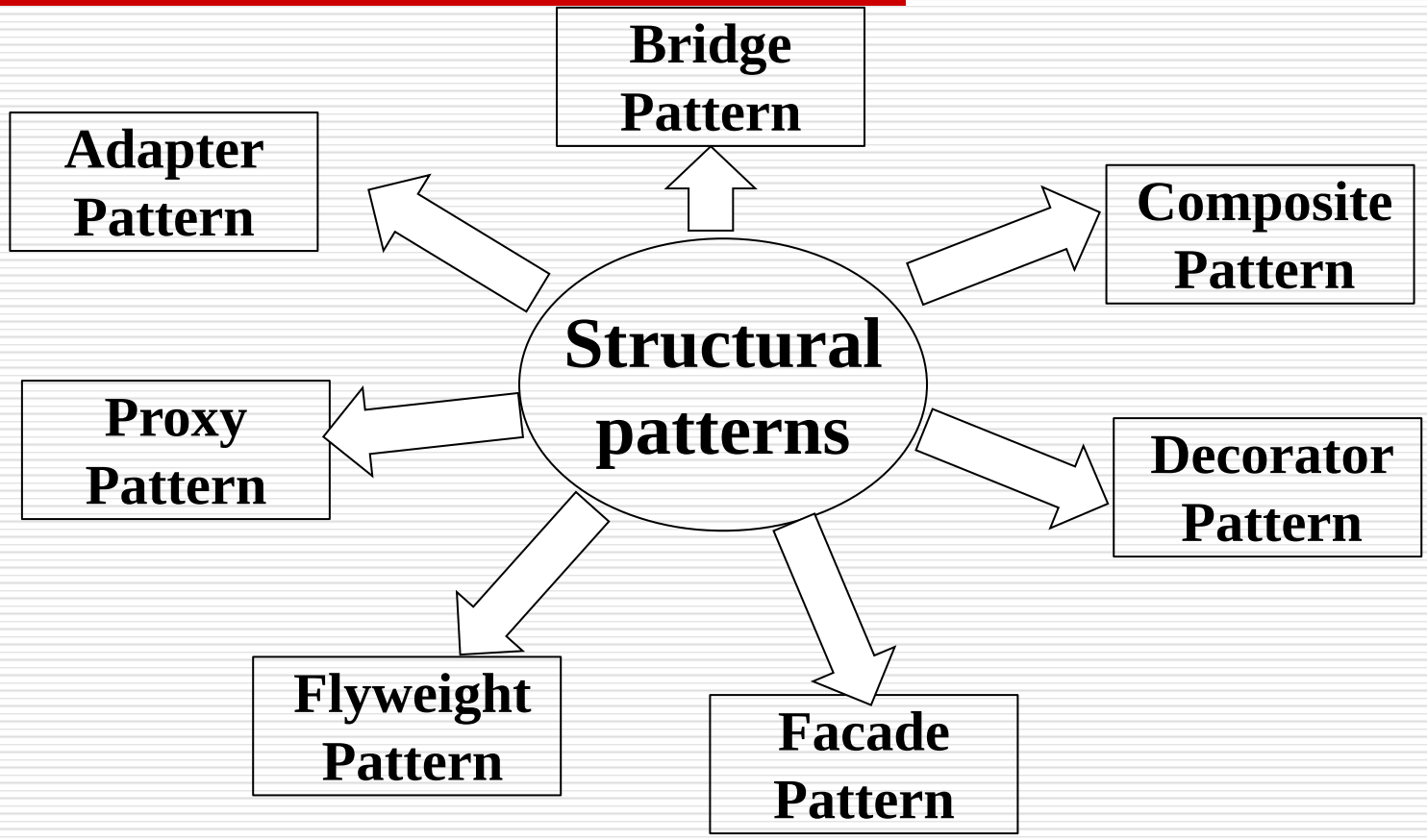
- **Intent** của pattern mô tả ngắn gọn cả vấn đề và giải pháp.
- **Motivation** giải thích thêm về vấn đề và giải pháp mà pattern có thể thực hiện được.
- **Structure** của các lớp thể hiện từng phần của pattern và cách chúng liên quan với nhau.
- **Code example** bằng một trong những ngôn ngữ lập trình phổ biến giúp bạn dễ dàng nắm bắt ý tưởng.

Phân loại

Tất cả các pattern có thể được phân loại theo **intent** hoặc **purpose** của chúng. Gồm ba nhóm mẫu chính:

- **Creational patterns**
- **Structural patterns**
- **Behavioral patterns**

Structural patterns



Structural patterns

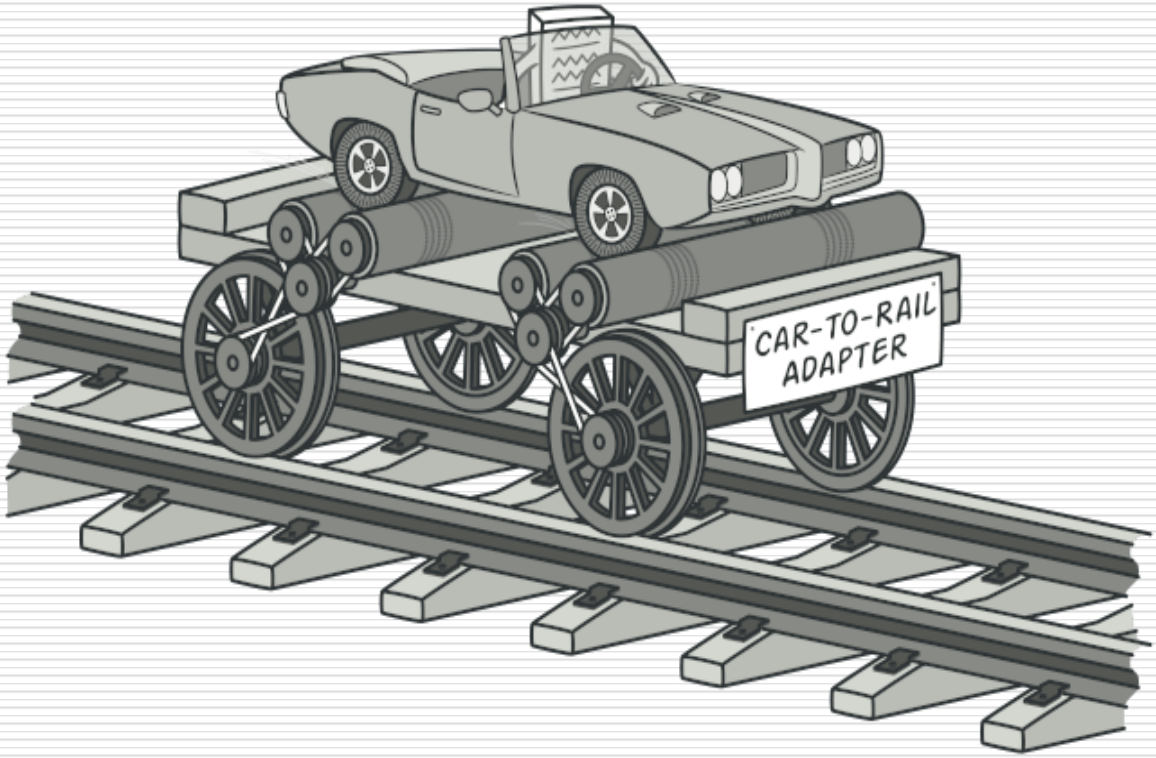
- **Adapter Pattern:** Mẫu thiết kế này giúp tạo ra một interface tương thích với một interface khác mà không cần sửa đổi các đối tượng hiện có.
- **Bridge Pattern:** Mẫu thiết kế này giúp tách rời sự trừu tượng hóa và cài đặt của một lớp, giúp cho việc thay đổi cài đặt dễ dàng hơn mà không ảnh hưởng đến sự trừu tượng hóa.
- **Composite Pattern:** Mẫu thiết kế này giúp xây dựng các cây đối tượng phức tạp bằng cách sử dụng các đối tượng đơn giản và các đối tượng con.

Structural patterns

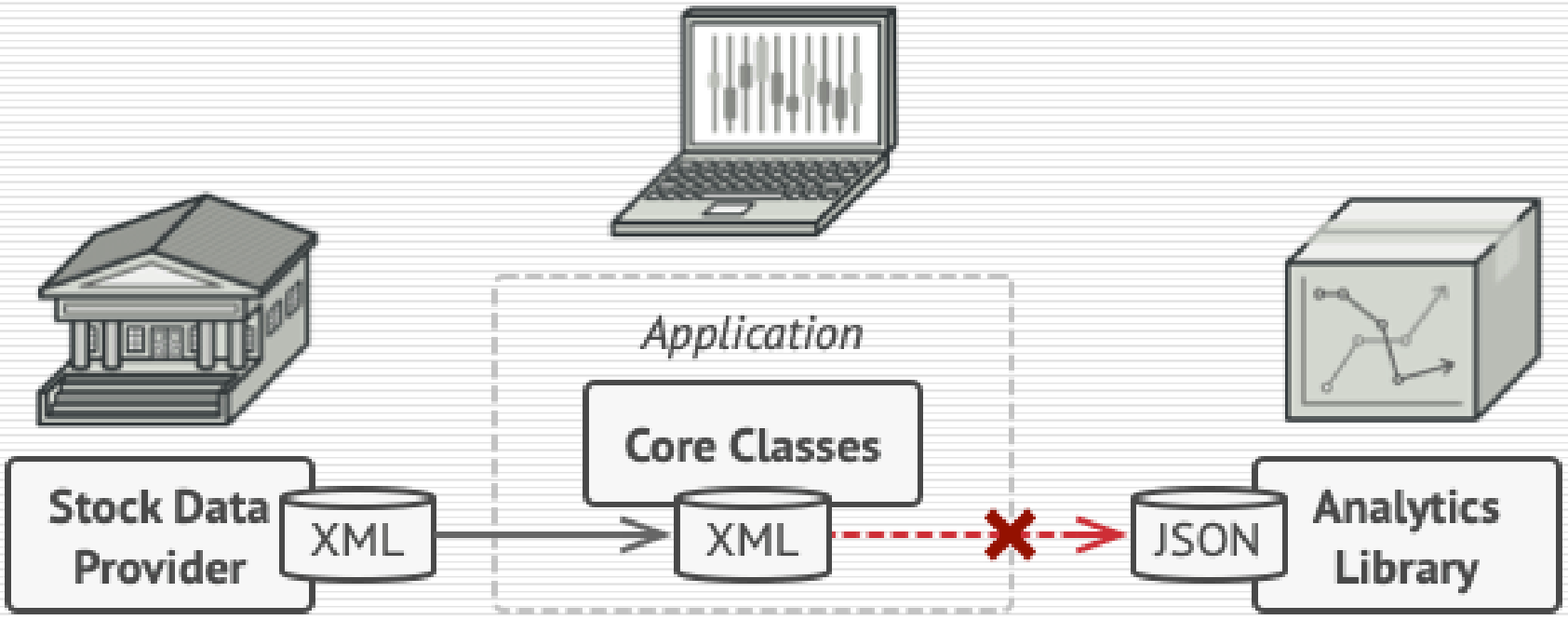
- **Decorator Pattern:** Mẫu thiết kế này cho phép thêm chức năng mới cho một đối tượng mà không ảnh hưởng đến các đối tượng khác trong chương trình.
- **Facade Pattern:** Mẫu thiết kế này cung cấp một interface đơn giản để truy cập đến các thành phần phức tạp của hệ thống.
- **Flyweight Pattern:** Mẫu thiết kế này giúp tối ưu hóa việc sử dụng bộ nhớ bằng cách chia sẻ các đối tượng giống nhau giữa nhiều đối tượng khác nhau.
- **Proxy Pattern:** Mẫu thiết kế này giúp tạo ra một đối tượng trung gian để truy cập đến một đối tượng khác, giúp tăng hiệu năng và kiểm soát truy cập đến đối tượng gốc.

Adaper Pattern

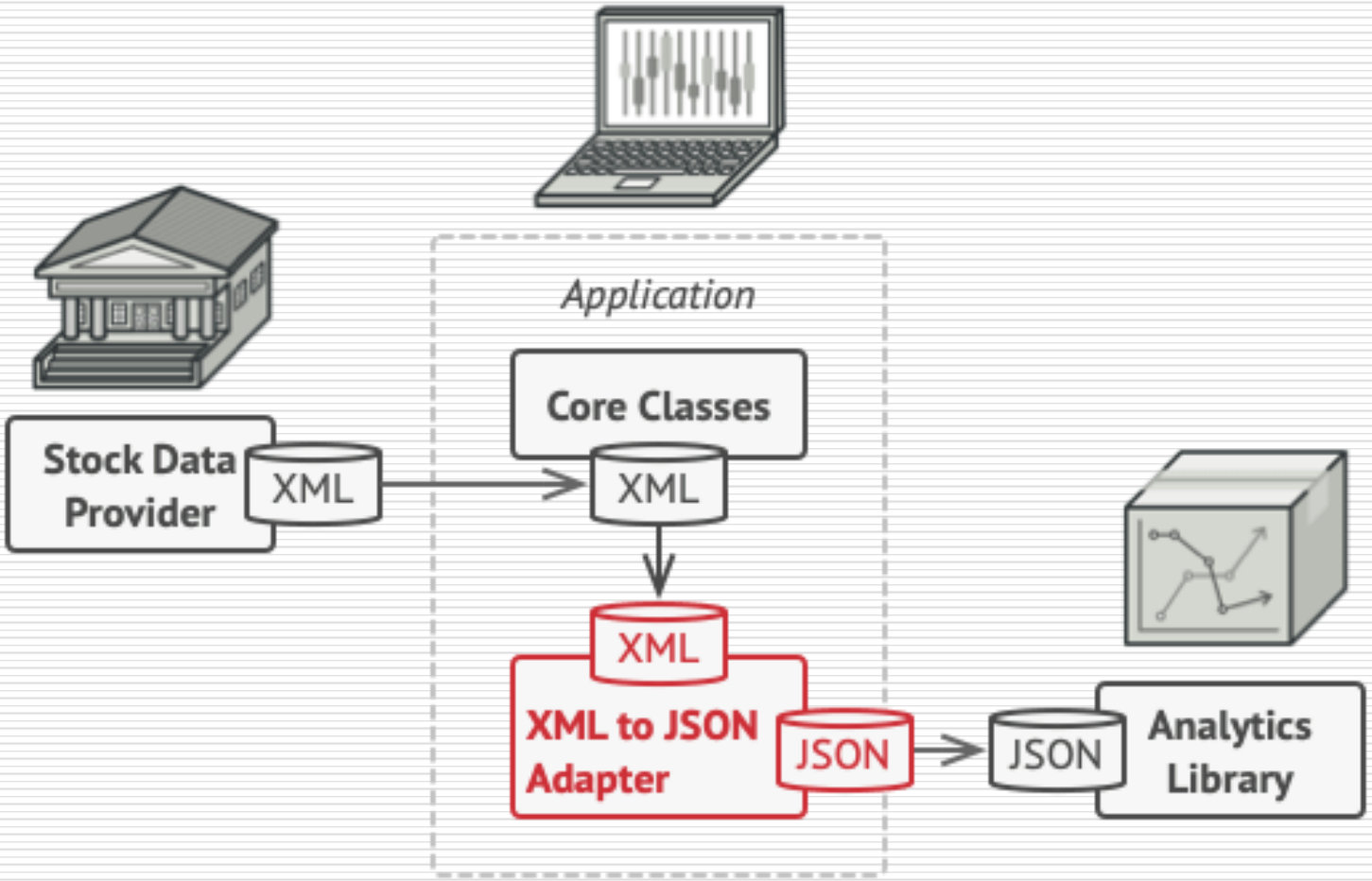
Adapter Design Pattern là một mẫu thiết kế (design pattern) phần mềm cho phép các **đối tượng** với các **interface không tương thích hoạt động cùng nhau**.



Đặt vấn đề


Application
Core Classes
Stock Data Provider
XML
XML
JSON
Analytics Library

Đặt vấn đề


Application
Core Classes
Stock Data Provider
XML
XML
XML
XML to JSON Adapter
JSON
JSON
Analytics Library

Nguyên tắc khởi tạo

- **Adaptee**: định nghĩa interface, class không tương thích, cần được tích hợp vào.
- **Adapter**: lớp tích hợp, giúp interface, class không tương thích tích hợp được với interface, class đang làm việc. Thực hiện việc chuyển đổi giữa các interface, class không tương thích.
- **Target**: một interface chứa các chức năng được sử dụng bởi Client.
- **Client:** là lớp sử dụng các đối tượng có interface Target.

Ví dụ

Giả sử chúng ta đang **có một lớp chứa phương thức print()**. Tuy nhiên, trong tương lai chúng ta lại **muốn sử dụng một lớp khác** để thực hiện phương thức print(), và **lớp này có phương thức tương thích là write() thay vì print()**. Để làm được điều này, chúng ta có thể **sử dụng adapter design pattern**.

Source code

```
public class OldPrinter {
    public void print(String text) {
        System.out.println(text);
    }
}

public interface NewPrinter {
    void write(String text);
}
```

Source code

```
public interface NewPrinter {
    void write(String text);
}

public class PrinterAdapter implements NewPrinter {
    private OldPrinter oldPrinter;

    public PrinterAdapter(OldPrinter oldPrinter) {
        this.oldPrinter = oldPrinter;
    }

    @Override
    public void write(String text) {
        oldPrinter.print(text);
    }
}
```

Source code

```
public static void main(String[] args) {
    OldPrinter oldPrinter = new OldPrinter();
    NewPrinter newPrinter = new PrinterAdapter(oldPrinter);

    newPrinter.write("Hello Teacher Qui");
}
```

```
Hello Teacher Qui
```

Ưu điểm của Adapter Design Pattern

- Tính linh hoạt.
- Tái sử dụng.
- Dễ bảo trì.

Nhược điểm của Adapter Design Pattern

- Tăng độ phức tạp.
- Sự chậm trễ.
- Khó hiểu.

Ví dụ thực tế

Giả sử chúng ta đang phát triển một ứng dụng quản lý tài khoản ngân hàng và chúng ta muốn tích hợp một API từ bên thứ ba để lấy thông tin tài khoản khách hàng. Tuy nhiên, API của bên thứ ba sử dụng một interface hoàn toàn khác với các interface của ứng dụng của chúng ta. Trong trường hợp này, chúng ta có thể sử dụng Adapter để tạo ra một Adapter để chuyển đổi interface của API bên thứ ba thành interface mà ứng dụng của chúng ta sử dụng.

Source code

```
// Interface của ứng dụng
public interface BankAccountInfo {
    public String getAccountNumber();
    public String getAccountName();
    public double getBalance();
}

// Interface của API của bên thứ ba
public interface ThirdPartyBankAccountInfo {
    public String getAccNo();
    public String getAccName();
    public double getBal();
}
```

Source code

```
// Adapter để chuyển đổi interface của API của bên thứ ba sang interface
    của ứng dụng
public class ThirdPartyBankAdapter implements BankAccountInfo {
    private ThirdPartyBankAccountInfo thirdPartyInfo;

    public ThirdPartyBankAdapter(ThirdPartyBankAccountInfo
        thirdPartyInfo) {
        this.thirdPartyInfo = thirdPartyInfo;
    }
    public String getAccountNumber() {
        return thirdPartyInfo.getAccNo();
    }
    public String getAccountName() {
        return thirdPartyInfo.getAccName();
    }
    public double getBalance() {
        return thirdPartyInfo.getBal();
    }
}
```

Source code

```
// Sử dụng Adapter để lấy thông tin tài khoản từ API của bên thứ ba
    và đưa vào ứng dụng của bạn
public class ClientTest {
    public static void main(String[] args) {
//        Customer customer = new BusinessAnalyst(new Developer());
//        customer.sendRequest("I want to work from home!");
        // Tạo một đối tượng ThirdPartyBankAccountInfo
        ThirdPartyBankAccountInfo thirdPartyInfo = new
            ThirdPartyBankAccountInfo() {
            public String getAccNo() {
                return "09090909";
            }
            public String getAccName() {
                return "Nguyen Van B";
            }

            public double getBal() {
                return 1900.0;
            }
        };
```

Source code

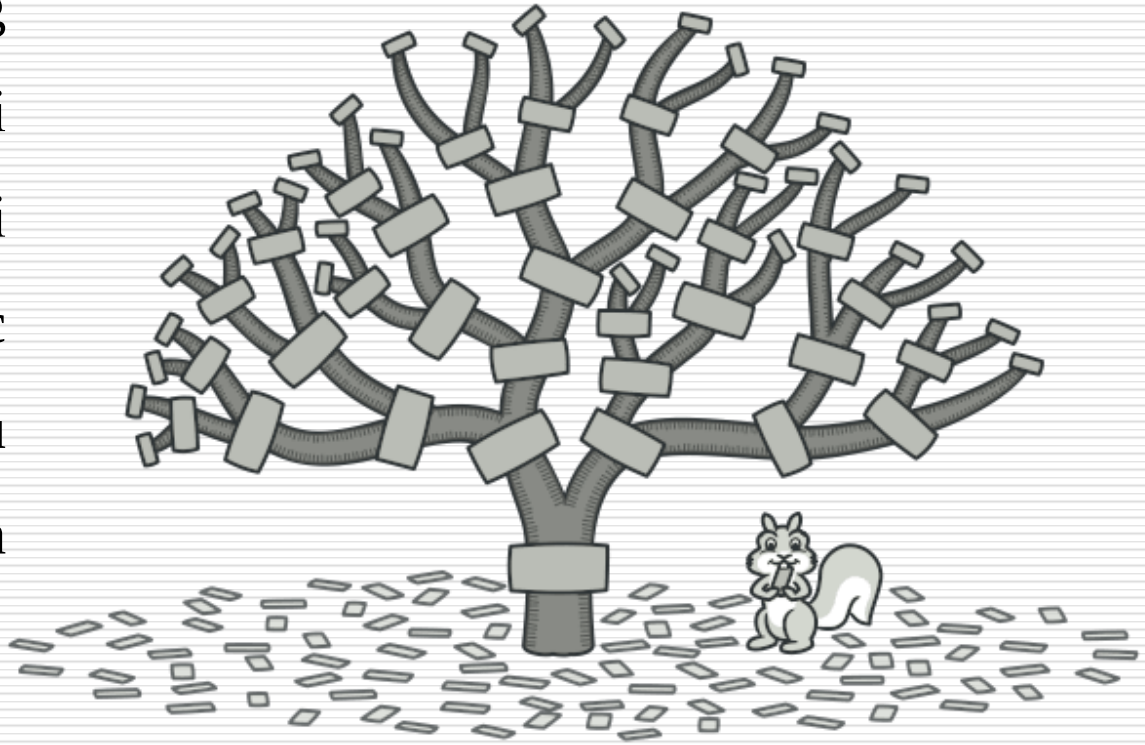
```
        // Tạo một đối tượng ThirdPartyBankAdapter để chuyển đổi
            interface của thirdPartyInfo sang BankAccountInfo
        BankAccountInfo adapter = new ThirdPartyBankAdapter
            (thirdPartyInfo);

        // Sử dụng đối tượng adapter để lấy thông tin tài khoản ngân
            hàng của khách hàng
        System.out.println("Account Number: " + adapter
            .getAccountNumber());
        System.out.println("Account Name: " + adapter.getAccountName
            ());
        System.out.println("Balance: " + adapter.getBalance());
    }
}
```

```
Account Number: 09090909
Account Name: Nguyen Van B
Balance: 1900.0
```

Composite Design Pattern

Composite cho phép chúng ta xây dựng các cây đối tượng **phức tạp** từ các đối tượng **đơn giản** hơn từ các nhóm con của chúng, và sau đó xử lý chúng một cách thống nhất.



Các thành phần trong Composite Design Pattern

- **BaseComponent**: là một interface hoặc abastract class quy định các method cần phải có cho tất cả các thành phần tham gia vào mẫu.
- **Leaf** : là lớp thực hiện kế thừa từ BaseComponent và không có con (giống như file thì không thể chứa file hoặc folder khác).
- **Composite**: Lưu trữ các Leaf và cũng kế thừa từ BaseComponent. Chúng cài đặt các phương thức có trong BaseComponent và khi gọi sẽ gọi thực hiện tại các thành phần con.
- **Client**: Đại diện cho nơi sử dụng Composite.

Source code

```
public abstract class Employee {
    protected int age;
    protected String name;
    protected double salary;

    public Employee(int age, String name, double salary) {
        this.age = age;
        this.name = name;
        this.salary = salary;
    }
    public abstract void add(Employee employee);
    public abstract void remove(Employee employee);
    public abstract void print();
```

Source code

```
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getSalary() {
        return salary;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
}
```

Source code

```
import java.util.ArrayList;
public class Boss extends Employee {
    private ArrayList<Employee> employeeList = new ArrayList();
    public Boss(int age, String name, double salary) {
        super(age, name, salary);
    }
    @Override
    public void add(Employee employee) {
        employeeList.add(employee);
    }
    @Override
    public void remove(Employee employee) {
        employeeList.remove(employee);
    }
    @Override
    public void print() {
        for (Employee employee : employeeList) {
            employee.print();
        }
    }
}
```

Source code

```
import java.util.ArrayList;
import java.util.List;

public class Leader extends Employee {

    private ArrayList<Employee> employeeList = new ArrayList();

    public Leader(int age, String name, double salary) {
        super(age, name, salary);
    }

    @Override
    public void add(Employee employee) {
        employeeList.add(employee);
    }

    @Override
    public void remove(Employee employee) {
        employeeList.remove(employee);
    }
```

Source code

```
    @Override
    public void print() {
        System.out.println("====================");
        System.out.println("Name : " + this.name);
        System.out.println("Age: " + this.age);
        System.out.println("Salary: " + this.salary);
        System.out.println("====================");
        for (Employee employee: employeeList) {
            employee.print();
        }
    }
}
```

Source code

```
public class Developer extends Employee {
    public Developer(int age, String name, double salary) {
        super(age, name, salary);
    }
    @Override
    public void add(Employee employee) {
    }
    @Override
    public void remove(Employee employee) {
    }
    @Override
    public void print() {
        System.out.println("==================");
        System.out.println("Name : " + this.name);
        System.out.println("Age: " + this.age);
        System.out.println("Salary: " + this.salary);
        System.out.println("==================");
    }
}
```

Source code

```
public class BusinessAnalyst extends Employee{
    public BusinessAnalyst(int age, String name, double salary) {
        super(age, name, salary);
    }
    @Override
    public void add(Employee employee) {
    }
    @Override
    public void remove(Employee employee) {
    }
    @Override
    public void print() {
        System.out.println("==================");
        System.out.println("Name : " + this.name);
        System.out.println("Age: " + this.age);
        System.out.println("Salary: " + this.salary);
        System.out.println("==================");
    }
}
```

Source code

```
public class ClientTest {
    public static void main(String[] args) {
        Developer dev1 = new Developer(30,"Anh",2000);
        Developer dev2 = new Developer(25,"Banh",1800);

        Leader leader = new Leader(45,"Sanh",5000);
        leader.add(dev1);
        leader.add(dev2);

        BusinessAnalyst ba = new BusinessAnalyst(27,"Hanh",2100);

        Boss boss = new Boss(60,"Tranh",8000);
        boss.add(leader);
        boss.add(ba);
        boss.print();

    }
}
```

Ưu điểm của Composite Design Pattern

- Dễ dàng tạo ra cấu trúc phức tạp.
- Dễ dàng thêm các thành phần mới.
- Dễ dàng duyệt cây.
- Dễ dàng thao tác với các đối tượng.
- Tăng tính linh hoạt.

Nhược điểm của Composite Design Pattern

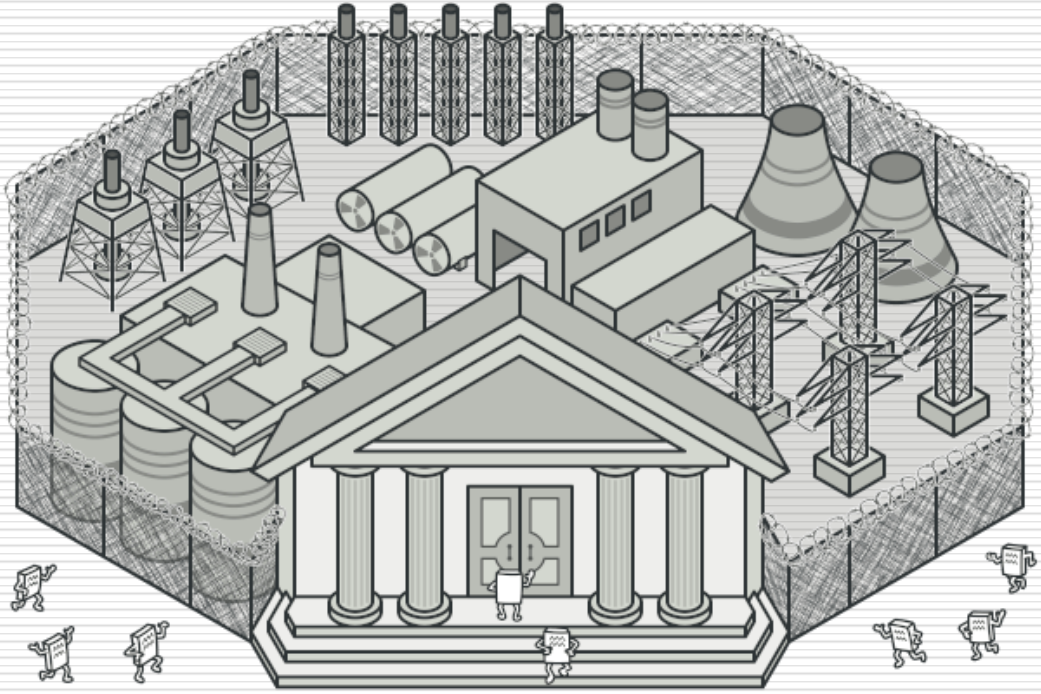
- Phức tạp.
- Không phù hợp cho các trường hợp mà mỗi đối tượng phải được xử lý khác nhau.
- Không phù hợp cho các trường hợp động.
- Khó khăn khi phân biệt giữa thành phần lá và thành phần trung gian.

Bài tập Composite Design Pattern

Hãy thiết kế một ứng dụng quản lý một Website tin tức bao gồm nhiều trang khác nhau. Trong các trang của website này, sẽ bao gồm nhiều phần tử khác nhau cấu thành. Các phần tử ấy có thể là đoạn văn, hình ảnh và video. Các trang này có thể chứa các phần tử con khác ngoài các phần tử trên (nếu bạn muốn bổ sung). Ngoài ra các phần tử cũng có thể có chứa các phần tử con khác (nếu có).

Facade Design Pattern

Facade Pattern cung cấp một **giao diện đơn giản** để truy cập đến một hệ thống phức tạp hơn bên trong, giúp **giảm thiểu sự phức tạp** và **tăng tính module** hóa của mã nguồn.



Các thành phần của Facade Design Pattern

- **Facade**: là thành phần trung gian biết rõ chức năng của những subsystems có thể thực hiện yêu cầu của client.
- **Subsystems**: cài đặt các chức năng của hệ thống con, xử lý công việc được gọi bởi Facade. Các lớp này không cần biết Facade và không tham chiếu đến nó.
- **Client**: đại diện cho nơi gọi tới và sử dụng Facade.

Source code

```
public interface Furniture {
    void make();
}

public class Table implements Furniture {
    @Override
    public void make() {
        System.out.println("Make a table");
    }
}

public class Television implements Furniture{
    @Override
    public void make() {
        System.out.println("make a TV");
    }
}
```

Source code

```
public class FurnitureFacade {
    private static FurnitureFacade INSTANCE;
    private Furniture table;
    private Furniture chair;
    private Furniture tv;
    public FurnitureFacade() {
        this.table = new Table();
        this.chair = new Chair();
        this.tv = new Television();
    }
```

Source code

```
    public static FurnitureFacade getInstance(){
        if (INSTANCE == null)
            INSTANCE = new FurnitureFacade();
        return INSTANCE;
    }
    public void makeTableAndChair(){
        table.make();
        chair.make();
    }
    public void makeTVAndChair(){
        tv.make();
        chair.make();
    }
}
```

Source code

```
public class ClientTest {
    public static void main(String[] args) {
        FurnitureFacade facade = FurnitureFacade.getInstance();
//          facade.makeTableAndChair();
        facade.makeTVAndChair();
    }
}
```

```
make a table
make a chair
```

Ưu điểm của Facade Design Pattern

- Giảm thiểu sự phức tạp của hệ thống.
- Tăng tính module hóa.
- Dễ dàng bảo trì.
- Tăng tính khả chuyển.

Nhược điểm của Facade Design Pattern

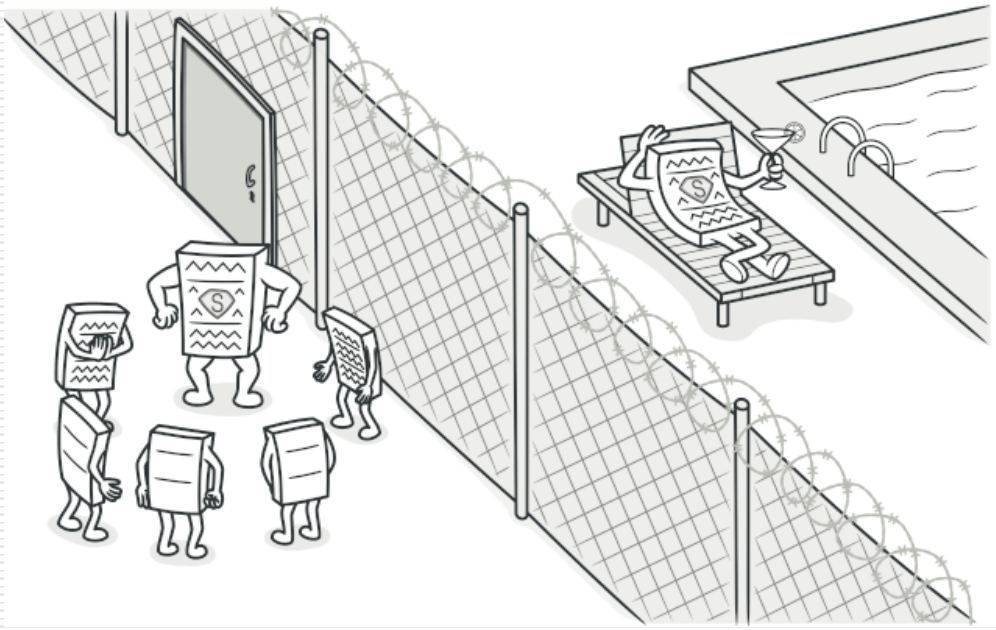
- Giới hạn khả năng tùy biến.
- Tăng chi phí.
- Khó khăn trong việc quản lý phụ thuộc.

Bài tập

1. Viết chương trình quản lý bán hàng cho một cửa hàng thời trang. Chương trình cần hỗ trợ các chức năng như thêm sản phẩm mới, xóa sản phẩm, hiển thị danh sách sản phẩm, tính tổng giá trị của các sản phẩm trong giỏ hàng, ... Sử dụng Facade pattern để giảm bớt sự phức tạp và đơn giản hóa việc gọi các phương thức của các đối tượng.
2. Viết chương trình quản lý thư viện, cho phép người dùng đăng nhập, đăng ký tài khoản, tìm kiếm sách, đặt mượn sách, ... Sử dụng Facade pattern để giảm bớt sự phức tạp của việc gọi các phương thức của các đối tượng khác nhau và đơn giản hóa việc quản lý và thực hiện các chức năng của chương trình.

Proxy Design Pattern

Proxy Design Pattern cung cấp một đối tượng **đại diện** (proxy) cho một **đối tượng khác** để **kiểm soát truy cập** vào của đối tượng đó.



Các thành phần của Proxy Design Pattern

- **Subject** : là một interface định nghĩa các phương thức để giao tiếp với client xác định những phương thức chung cho RealSubject và Proxy để Proxy có thể được sử dụng bất cứ nơi nào mà RealSubject muốn.
- **Proxy** : là một class sẽ thực hiện các bước kiểm tra và gọi tới đối tượng của class RealSubject để thực hiện các thao tác sau khi kiểm tra.
- **RealSubject** : là một class service sẽ thực hiện các thao tác thực sự. Đây là đối tượng chính mà proxy đại diện.
- **Client** : đại diện cho nơi cần sử dụng RealSubject nhưng cần thông qua Proxy.

Source code

```
public interface Image {
    void load();
}
public class ProxyImage implements Image{
    private String url;
    private RealImage realImage;
    public ProxyImage(String url) {
        this.url = url;
    }

    @Override
    public void load() {
        if (realImage == null){
            realImage = new RealImage(url);
        }else {
            System.out.println("Image already existed");
        }
        realImage.load();
    }
}
```

Source code

```
public class RealImage implements Image{
    private String url;

    @Override
    public void load() {
        System.out.println("Load from "+this.url);
    }

    public RealImage(String url) {
        this.url = url;
    }
}
```

Source code

```
public class ClientTest {
    public static void main(String[] args) {
        ProxyImage proxyImage = new ProxyImage("https://abcd.com/dev
            .png");

        System.out.println("First time");
        proxyImage.load();
        System.out.println("Seconds time");
        proxyImage.load();
    }
}
```

```
first load
Init
Load from https://abcd.com/dev.png
seconds load
Image already existed!
Load from https://abcd.com/dev.png
```

Ưu điểm của Proxy Design Pattern

- Bảo vệ quyền truy cập.
- Giảm tải hệ thống.
- Quản lý tài nguyên.
- Tăng cường tính đa dạng.

Nhược điểm của Proxy Design Pattern

- Tăng độ phức tạp của mã.
- Tăng chi phí thực hiện.
- Yêu cầu sự hiểu biết về thiết kế.
- Tăng độ trễ.

Bài tập

Giả sử bạn đang phát triển một ứng dụng xem video trực tuyến. Trong ứng dụng của bạn, người dùng có thể xem các video từ các nguồn khác nhau trên mạng. Tuy nhiên, bạn muốn giới hạn việc tải video để tránh tình trạng quá tải trên server. Để giải quyết vấn đề này, bạn sẽ sử dụng Proxy Pattern để tạo một proxy cho video, cho phép giới hạn số lần tải video.

Thank you!