ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA TOÁN - CƠ - TIN HỌC BÁO CÁO CUỐI KỲ

CÁC THÀNH PHẦN PHẦN MỀM

Tên Đề Tài: DESIGN PATTERNS *(nhóm 1)*

Sinh viên thực hiện:
Phạm Bá Thắng - 20001976
La Thị Anh Thư - 20001980

HÀ NỘI, 1/2022

LỜI CẢM ƠN

Đầu tiên, chúng em xin gửi lời cảm ơn chân thành đến Trường Đại học Khoa học Tự nhiên đã đưa môn học Các thành phần phần mềm vào chương trình giảng dạy. Đặc biệt, chúng em xin gửi lời cảm ơn sâu sắc đến giảng viên bộ môn - Thầy Quản Thái Hà đã dạy dỗ, truyền đạt những kiến thức quý báu cho chúng em trong suốt thời gian học tập vừa qua. Trong thời gian tham gia lớp học Các thành phần phần mềm của thầy, chúng em đã có thêm cho mình nhiều kiến thức bổ ích, tinh thần học tập hiệu quả, nghiêm túc. Đây chắc chắn sẽ là hành trang để chúng em có thể vững bước sau này.

Bộ môn Các thành phần phần mềm là môn học thú vị, vô cùng bổ ích và có tính thực tế cao. Đảm bảo cung cấp đủ kiến thức, gắn liền với nhu cầu phát triển của sinh viên. Tuy nhiên, khả năng tiếp nhận kiến thức của mỗi người luôn tồn tại những hạn chế nhất định. Do đó, bài báo cáo khó có thể tránh khỏi những thiếu sót. Kính mong thầy xem xét và góp ý để bài báo cáo của chúng em được hoàn thiện hơn.

Kính chúc thầy sức khỏe, hạnh phúc, thành công trên con đường sự nghiệp giảng dạy.

Mục Lục

Phần I

Tóm tắt

Design pattern là các giải pháp tổng thể đã được tối ưu hóa, được tái sử dụng cho các vấn đề phổ biến trong thiết kế phần mềm mà chúng ta thường gặp phải hàng ngày. Đây là tập các giải pháp đã được suy nghĩ, đã giải quyết trong tình huống cụ thể.

Design pattern có thể thực hiện được ở phần lớn các ngôn ngữ lập trình. Nó giúp bạn giải quyết vấn đề một cách tối ưu nhất, cung cấp cho bạn các giải pháp trong lập trình hướng đối tượng (OOP).

Design pattern gồm 23 loại được chia thành 3 nhóm:

- **Creational Pattern (nhóm khởi tạo) gồm: Factory Method, Abstract Factory, Builder, Prototype, Singleton. Những Design pattern loại này cung cấp một giải pháp để tạo ra các object và che giấu được logic của việc tạo ra nó, thay vì tạo ra object một cách trực tiếp bằng cách sử dụng method new. Điều này giúp cho chương trình trở nên mềm dẻo hơn trong việc quyết định object nào cần được tạo ra trong những tình huống được đưa ra**
- **Structural Pattern (nhóm cấu trúc) gồm: Adapter, Bridge, Composite, Decorator, Facade, Flyweight và Proxy. Những Design pattern loại này liên quan tới class và các thành phần của object. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.**
- **Behavioral Pattern (nhóm hành vi) gồm: Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy và Visitor. Nhóm này dùng trong thực hiện các hành vi của đối tượng, sự giao tiếp giữa các object với nhau.**

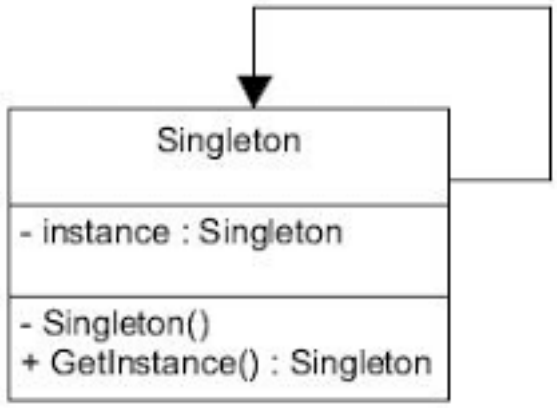
Phần II

Nhóm Khởi tạo - Creational Pattern

1 Singleton

1.1 Định Nghĩa Và Mô Hình Cấu Trúc

- **Singleton đảm bảo chỉ duy nhất một instance được tạo ra và cung cấp một method để có thể truy xuất được instance đó mọi lúc mọi nơi trong chương trình.**
- **Singleton ẩn đi hàm dựng của class và sử dụng hàm các hàm có sẵn bên trong để tạo đối tượng**



1.2 Đặc điểm

- **Đảm bảo rằng một lớp chỉ có một instance**
- **Dễ dàng truy cập vào instance này**
- **Kiểm soát việc khởi tạo nó**
- **Giới hạn số lượng instance**
- **Có thể truy cập như một biến toàn cục**

1.3 Mục Đích Sử Dụng

- **Là dạng class nên có thể sử dụng với nhiều design pattern khác nhau**
- **Sử dụng làm biến toàn cục với các ưu điểm như:**

+ **Không làm rối loạn danh sách các biến toàn cục vì không tạo ra các biến không cần thiết**
+ **Chúng cho phép phân bổ và khởi tạo khi cần (call-by-need), trong khi tạo nhiều các biến toàn cục ngay từ đầu sẽ luôn tiêu tốn tài nguyên.**

- **Ví dụ như dùng để ghi log rất thích hợp vì chúng ta chỉ cần 1 bản ghi log duy nhất cho mỗi phiên đồng thời lại cần ở tất cả các bộ phận của chương trình**

1.4 Code khuôn mẫu

```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();
    private Singleton() {}
    public static Singleton getInstance(){
        if(INSTANCE == null){
            INSTANCE = new Singleton();
        }
        return INSTANCE;
    }
}
```

1.5 Giải thích Design Pattern

- **Giải thích theo mô hình**
  + **Private constructor để hạn chế truy cập từ class bên ngoài.**
  + **Đặt private static final variable đảm bảo biến chỉ được khởi tạo trong class và duy nhất 1 lần.**
- **Giải thích theo code khuôn mẫu**
  + **Có một method public static để return instance được khởi tạo ở trên và một đoạn if để kiểm tra xem instance tồn tại chưa, nếu chưa sẽ gọi hàm khởi tạo. Hàm này được gọi là accessor.**
  + **Sau đó là các hàm để sử dụng.**

1.6 Ví Dụ Sử Dụng Trong Thực Tế

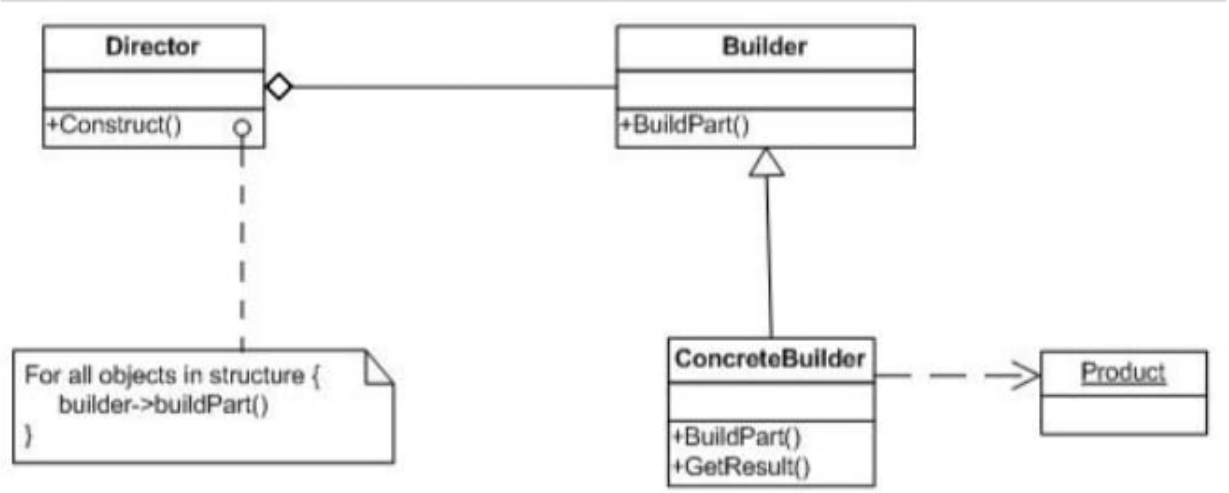
Simple Logger
https://github.com/blaplafla13th/design-partterns/tree/main/src/singleton

Tạo một logger theo Singleton pattern, sau đó chúng ta thêm các hàm kiểm tra file (check), hàm ghi file (write) và hàm đọc file (read). Khi runtime, các Exception được ghi lại và chuyển sang String rồi ghi ra file, sau đó file được đọc. Khi dùng trong package, người dùng có thể gọi 3 hàm read write và chỉ cần gọi đối tượng mà không cần khởi tạo lại đối tượng Log.

2 Builder

2.1 Định Nghĩa Và Mô Hình Cấu Trúc

- **Builder là mẫu thiết kế được tạo ra để chia việc khởi tạo một đối tượng phức tạp thành từng phần riêng rẽ, từ đó có thể tiến hành đối tượng khởi tạo từ các đối tượng đơn giản hơn.**
- **Mô hình:**



2.2 Đặc điểm

- **Không cần truyền giá trị null cho các tham số không sử dụng**
- **Kiểm soát tốt hơn quá trình xây dựng đối tượng**
- **Có thể tạo đối tượng immutable**

2.3 Mục Đích Sử Dụng

- **Tạo một đối tượng phức tạp: có nhiều thuộc tính (nhiều hơn 4) và một số bắt buộc, một số không bắt buộc**
- **Tách rời quá trình xây dựng một đối tượng phức tạp từ các phần tạo nên đối tượng**
- **Khi người dùng (client) mong đợi nhiều cách khác nhau cho đối tượng được xây dựng**

- Giảm bớt số lượng hàm constructor

2.4 Code khuôn mẫu

```
package builder.pattern;
//Builder.java
public interface Builder {
    public void buildPartOne();
    public void buildPartTwo();
    public Product getProduct();
}

//Client.java
public class Client {
    public static void main(String[] args) {
        Director director = new Director(new ConcreteBuilder());
        director.makeProduct();

        Product product = director.getProduct();

        System.out.println("Product part: " + product.getPartOne());
        System.out.println("Product part: " + product.getPartTwo());
    }
}

//ConcreteBuilder.java
public class ConcreteBuilder implements Builder{
    private Product product;

    public ConcreteBuilder() {
        this.product = new Product();
    }

    @Override
    public void buildPartOne() {
```

```
32          product.setPartOne("Part One");
33      }
34
35      @Override
36      public void buildPartTwo() {
37          product.setPartTwo("Part Two");
38      }
39
40      @Override
41      public Product getProduct() {
42          return product;
43      }
44  }
45
46  //Director.java
47  public class Director {
48      private Builder builder;
49
50      public Director(Builder builder){
51          this.builder = builder;
52      }
53
54      public void makeProduct(){
55          builder.buildPartOne();
56          builder.buildPartTwo();
57      }
58
59      public Product getProduct(){
60          return builder.getProduct();
61      }
62  }
63
64  //Product.java
65  public class Product {
66      private String partOne;
```

```
67      private String partTwo;
68
69      public String getPartOne() {
70          return partOne;
71      }
72
73      public void setPartOne(String partOne) {
74          this.partOne = partOne;
75      }
76
77      public String getPartTwo() {
78          return partTwo;
79      }
80
81      public void setPartTwo(String partTwo) {
82          this.partTwo = partTwo;
83      }
84  }
```

2.5 Giải thích Design Pattern

- Giải thích theo mô hình

  + Product: đại diện cho đối tượng cần tạo, đối tượng này phức tạp, có nhiều thuộc tính.

  + Builder: là abstract class hoặc interface khai báo phương thức tạo đối tượng.

  + ConcreteBuilder: kế thừa Builder và cài đặt chi tiết cách tạo ra đối tượng. Nó sẽ xác định và nắm giữ các instance mà nó tạo ra, đồng thời nó cũng cung cấp phương thức để trả các các thể hiện mà nó đã tạo ra trước đó. Hay nói cách khác là bản thiết kế từng bước của đối tượng Product

  + Director/Client: là nơi sẽ gọi tới Builder để tạo ra đối tượng.

- Giải thích theo code khuôn mẫu

  + Khi chạy main của Client, đối tượng Director được khởi tạo, với đối đầu

vào là 1 instance ConcreteBuilder được kế thừa từ Builder chứa thông tin từng bước

+ Tiếp theo Client gọi hàm thực thi trong Director là hàm makeProduct chứa các bước trong interface builder từ đó gọi các hàm override bởi các hàm trong Builder ConcreteBuilder để thực hiện các bước dựng đối tượng

+ Sau khi Director dựng xong đối tượng, Client lấy đối tượng Product về từ Director từ đó có thể sử dụng Product như bình thường

2.6 Ví Dụ Sử Dụng Trong Thực Tế

- Hệ thống quản lý tài khoản sử dụng builder parttern https://github.com/blaplafla13th/design-partterns/tree/main/src/builder

- Do tạo tài khoản hay đăng nhập, chúng ta cùng có hai bước là ghi nhận thông tin và xử lý thông tin như vậy chúng ta cần tạo một interface AccountBuilder với 2 step để xử lý với step 1 là điền thông tin, step 2 là xử lý dữ liệu và đọc ghi vào mảng cụ thể là accountArrayList ở Client.java. Tiếp theo tạo class builder của signIn và signUp để tạo form điền ở step1 và xử lý form ở step2.

- Chúng ta tạo một AccountDirector để tạo đối tượng builder để chạy hai builder vừa tạo. Hàm dựng sẽ nhận bản thiết kế của class builder và thực hiện các bước. sau đó là hàm để return đối tượng

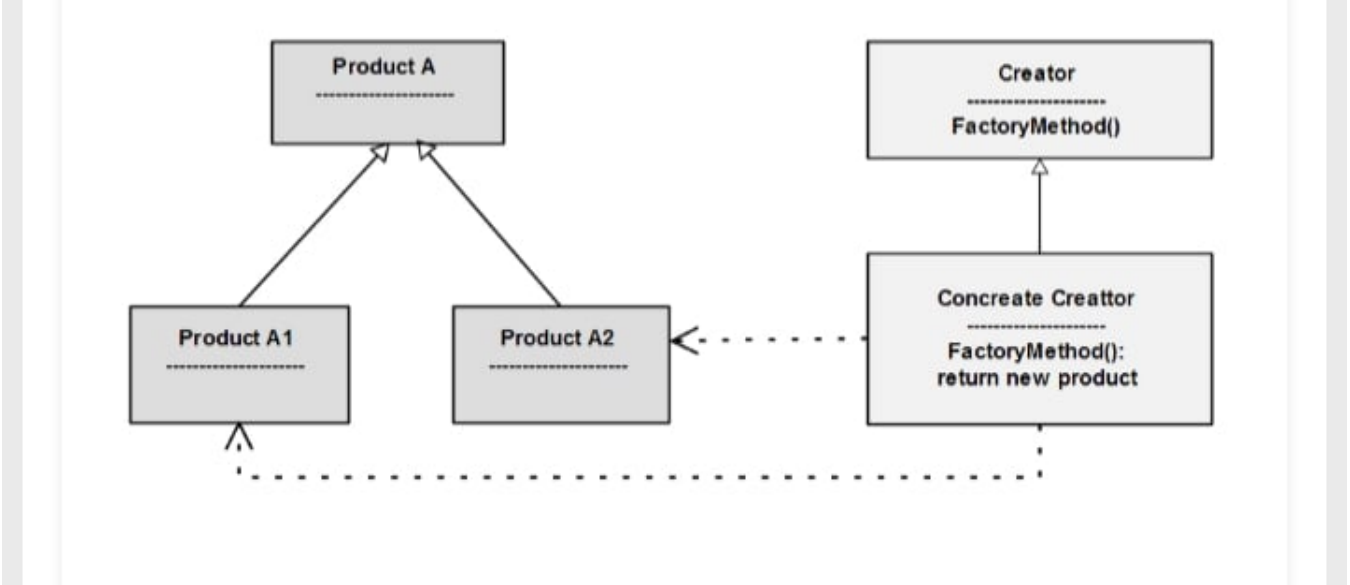
- Về phía Client, ta cần tạo 1 arraylist để lưu tại khoản và hàm để gọi arraylist này. Trong main ta tạo biến current để lưu tài khoản hiện tại đóng vai trò quản lý phiên. Tạo một vòng lặp để lựa chọn đăng kí và đăng nhập và 2 director tương ứng với 2 lựa chọn đăng kí và đăng nhập. Tiếp theo chúng ta sẽ chạy hàm sign của Director để Directỏ chạy các bước như thiết kế.

- Như vậy ta hoàn toàn có thể đọc sửa thông tin của account như ví dụ và lưu current vào mảng thay thế cho account hiện đăng nhập. Khi log out chúng ta đặt current = null để thoát phiên.

3 Factory

3.1 Định Nghĩa Và Mô Hình Cấu Trúc

- **Factory sử dụng một interface hay một abstract class mà tất cả các lớp chúng ta cần khởi tạo đối tượng sẽ kế thừa. Factory sẽ khởi tạo một đối tượng mới mà không cần thiết phải chỉ ra chính xác class nào sẽ được khởi tạo**
- **Mô hình:**



3.2 Đặc điểm

- **Người dùng không biết logic thực sự được khởi tạo bên dưới phương thức factory**
- **Cho phép các lớp con chọn kiểu đối tượng cần tạo**
- **Code chỉ tương tác với interface hoặc lớp abstract**
- **Dễ dàng quản lý file cycle của các đối tượng được tạo**
- **Thống nhất về mặt naming convention**

3.3 Mục Đích Sử Dụng

- **Tạo ra một cách mới trong việc khởi tạo đối tượng thông qua một interface chung**

- Che giấu đi xử lý logic của việc khởi tạo đối tượng
- Giảm sự phụ thuộc giữa các module và tăng tính mở rộng code

3.4 Code khuôn mẫu

```
package factory.pattern;
//Factory.java
public class Factory {
    public Product createProduct(String productType){
        switch (productType){
            case "a": return new ProductA();
            case "b": return new ProductB();
             ...
            default: return null;
        }
    };
}

//Product.java
public abstract class Product {
     ...
}

//ProductA.java
public class ProductA extends Product{
    public ProductA(){
         ...
    }
     ...
}

//ProductB, ... similar as class ProductA

//Client.java
public class Client{
```

```
31      public static void main(String[] args){
32          Factory factory = new Factory();
33          Product product = factory.createProduct("a");
34      }
35  }
```

3.5 Giải thích Design Pattern

- Giải thích theo mô hình
  - + Super Class: môt super class (class A trong mô hình) trong Factory Pattern có thể là một interface, abstract class hay một class thông thường. các Sub Class (A1, A2,...) là các class kế thừa từ class A theo nhiệm vụ của nó
  - + Class Concrete Creator hay Factory sẽ chịu tránh nhiệm khởi tạo các đối tượng sub class dựa theo tham số đầu vào và trả về cho Client. Thường thì class này là 1 Singleton.
  - + Sau đó class Client hay Creator sẽ gọi đến Concrete Creator để lấy đối tượng mà chúng ta cần
- Giải thích theo code khuôn mẫu
  - + Chúng ta có 1 superclass Product và các class con kế thừa class Product như ProductA, ProductB,...
  - + Class Factory có nhiệm vụ tạo đối tượng với 1 hàm createProduct với tham số là loại product. Trong hàm là 1 switch case để khi nhập 1 giá trị, hàm sẽ trở về 1 đối tượng mới tương ứng của Product.
  - + Về phía Client chúng ta tạo một instance của Factory rồi dùng hàm createProduct từ đối tượng factory để tạo ra đối tượng mình mong muốn

3.6 Ví Dụ Sử Dụng Trong Thực Tế

- Hệ thống thanh toán đơn giản sử dụng factory parttern https://github.com/blaplafla13th/design-partterns/tree/main/src/factory
- chúng ta tạo 1 package paymentmethod để chứa interface PaymentMethod và các class implements từ PaymentMethod. Mõi class đều chứa tên phương thức, số tài khoản, tên người dùng và đoạn code ẩn để thanh toán

- **Tiếp theo tạo 1 class FactoryPaymentMethod chứa 1 hàm trả về đói tượng PaymentMethod với đối đầu vào là tên phương thức thanh toán và bên trong chứa 1 switch case các phương thức thanh toán và mỗi lựa chọn của switch case trả về 1 instance PaymentMethod tương ứng**
- **Đối với Client ta dùng hàm listFile để gọi ra các phương thức thanh toán khả dụng. Khi người dùng nhập phương thức thanh toán, đối tượng factoryPaymentMethod được khởi tạo, chạy hàm getPaymentMethod để trả về phương thức thanh toán như yêu cầu từ đó người dùng có thể sử dụng các hàm trong phương thức thanh toán không biết được khởi tạo bên dưới phương thức factory**
- **Khi cần thêm phương thức thanh toán, lập trình viên chỉ cần tạo 1 class mới trong package paymentmethod và thêm vào switch case của FactoryPaymentMethod**

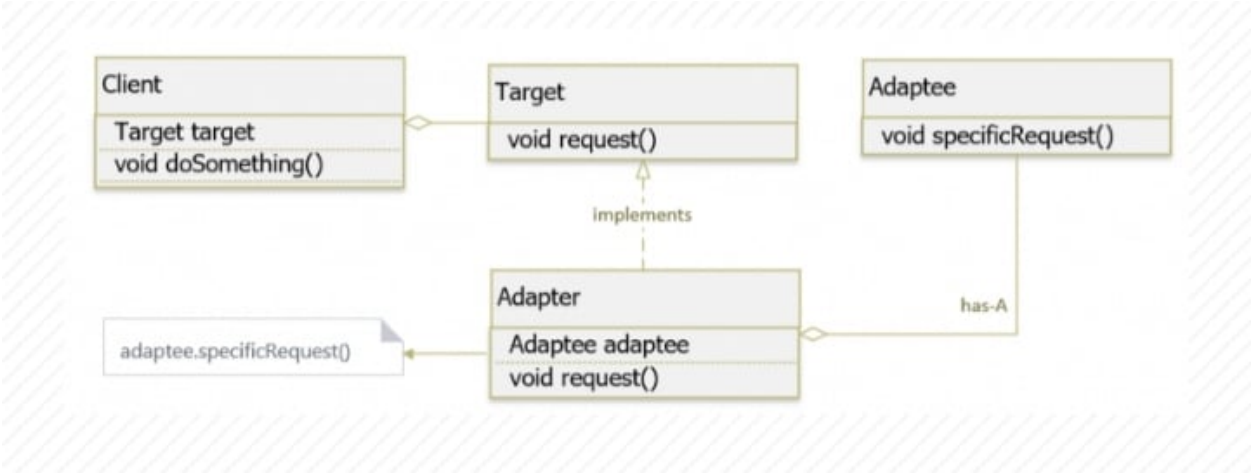
Phần III

Nhóm Cấu trúc - Structural Pattern

4 Adapter

4.1 Định Nghĩa Và Mô Hình Cấu Trúc

- **Adapter Pattern cho phép các interface không liên quan tới nhau có thể làm việc cùng nhau. Đối tượng giúp kết nối các interface gọi là Adapter**
- **Adapter Pattern giữ vai trò trung gian giữa hai lớp, chuyển đổi interface của một hay nhiều lớp có sẵn thành một interface khác thích hợp cho lớp đang viết**
- **Mô hình:**



4.2 Đặc điểm

- **Phân tách việc chuyển đổi interface với business logic chính của chương trình**
- **Có thể sử dụng adapter với các class con của adaptee**
- **Làm việc với adapter class thay vì sửa đổi bên trong adaptee class đã có sẵn**
- **Client tiếp cận thông qua interface thay vì implementation**
- **Tăng chi phí và độ phức tạp của code**

4.3 Mục Đích Sử Dụng

- **Chuyển đổi interface của một class thành interface mà client yêu cầu**

- **Tạo ra những lớp có khả năng sử dụng lại, chúng phối hợp với các lớp không liên quan hay những lớp không thể đoán trước được và những lớp này không có interface tương thích**
- **Khi muốn đảm bảo nguyên tắc Open/Close trong một ứng dụng**

4.4 Code khuôn mẫu

```
1  package adapter.pattern;
2  //Adaptee.java
3  public class Adaptee {
4      public Adaptee(String name){
5
6      }
7
8      public void specificRequest() {
9          System.out.println("Called specific request on Adaptee ");
10     }
11 }
12
13 //Adapter.java
14 public class Adapter implements Target {
15     private Adaptee adaptee;
16
17     public Adapter(Adaptee adaptee){
18         this.adaptee = adaptee;
19     }
20
21     @Override
22     public void request() {
23         adaptee.specificRequest();
24     }
25 }
26
27 //Client.java
28 public class Client {
```

```
29      public static void main(String[] args) {
30          Target target = new Adapter(new Adaptee());
31          target.request();
32      }
33  }
34
35  //Target.java
36  public interface Target {
37      public void request();
38  }
```

4.5 Giải thích Design Pattern

- Theo Mô hình
  - + Adaptee: là một class không tương thích, cần được tích hợp vào.
  - + Adapter: lớp tích hợp, giúp class không tương thích tích hợp được với interface đang làm việc. Thực hiện việc chuyển đổi interface cho Adaptee và kết nối Adaptee với Client.
  - + Target: một interface chứa các chức năng được sử dụng bởi Client (domain specific).
  - + Client: lớp sử dụng các đối tượng có interface Target.
- Theo code minh họa
  - + Class Adaptee có 1 phương thức cần
  - + Một interface Target sẽ chứa hàm request với mục đích để gọi hàm trong Adaptee
  - + Adapter sẽ implements interface Target, có một hàm dựng với tham số là instance của Adaptee. Hàm request của Adapter sẽ gọi đến hàm cần yêu cầu của Adaptee.
  - + Client: sẽ tạo instance của Adapter với đối số là một đối tượng adaptee mới. Như vậy ta có thể gọi hàm cần yêu cầu qua phương thức request trên target

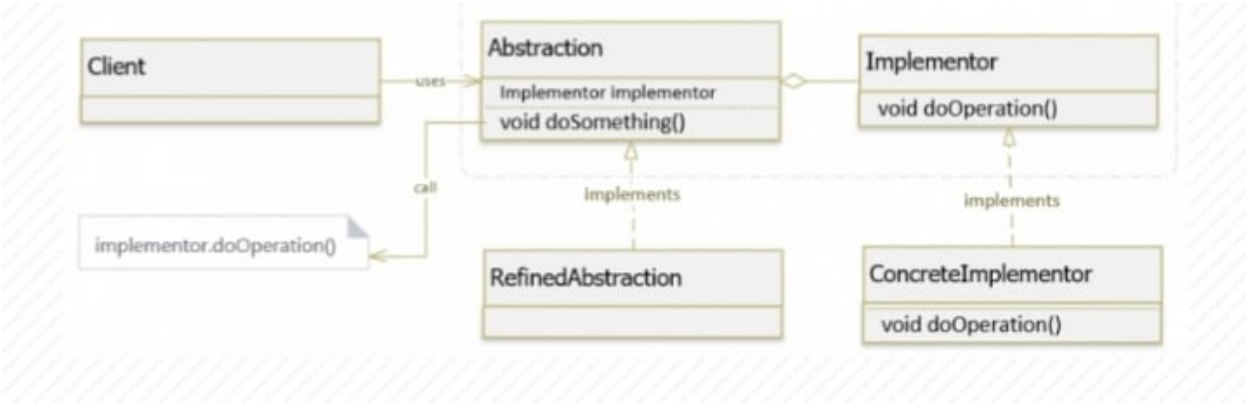
4.6 Ví Dụ Sử Dụng Trong Thực Tế

- **Dịch một chương trình đơn giản bằng adapter pattern https://github.com/blaplafla13th/design-partterns/tree/main/src/adapter**
- **Ban đầu chúng ta có 2 file Program.java và Client.java. Program.java sử dụng hàm command và status để trả về các giá trị String rồi mới in chứ không in trực tiếp String bằng System.out.println.Class Program ở đây đóng vai trò của adaptee, giờ chúng ta sẽ viết 1 adapter và target**
- **Ta tạo một interface Language chứa các hàm sẽ gọi của Program trong main làm target**
- **Tạo một class implememt interface Language ở đây là class Vietnamese. Trong class ta sẽ dùng mảng String 2 chiều để lưu cặp giá trị String gồm từ chương trình trả về và từ chúng ta muốn dịch sang (mảng languagePack). Viết hàm translateMessage với tham số kiểu String để nhận giá trị chương trình trả về. Tạo 1 vòng for để tìm giá trị đó trong languagePack và trả về giá trị tương ứng của cặp.**
- **Trong các hàm implement từ Language chúng ta sẽ trả về giá trị mà translateMessage trả về từ tham số tương ứng với đối số truyền vào là kết qủa của hàm tương ứng bên file Program.java**
  Ví dụ như hàm run trong Vietnamese sẽ lấy tham số truyền vào giống hàm run ở Program trả về là translateMessage với đối số truyền vào là kết quả trả về của hàm run bên Program.
- **Đối với Client.java, chúng ta tạo một instance kiểu Language của class Vietnamese với đối số truyền vào là Program và thay vì gọi đến hàm trong instance Program thì chúng ta gọi đến instance của Language. Hay đơn giản hơn thay vì chúng ta sửa dòng tạo đối tượng Program thành tạo đối tượng kiểu Language của class Vietnamese, đối số là đối tượng Program mới, giữ nguyên tên biến là ta đã dịch thành công chương trình**
- **Trong thực tế cách này được dùng để dịch các chương trình sử dụng bộ các keyword:value để in thông tin, keyword:value thường được lưu ra file riêng, sẽ có hàm đọc file này để ghi vào mảng languagePack ở trên, cách này không dịch được các thông tin cứng được in trực tiếp mà không có trong bộ keyword:value**

5 Bridge

5.1 Định Nghĩa Và Mô Hình Cấu Trúc

- **Bridge Pattern tách tính trừu tượng (abstraction) ra khỏi tính hiện thực (implementation) của nó. Từ đó, có thể dễ dàng chỉnh sửa hoặc thay thế mà không làm ảnh hưởng đến những nơi có sử dụng lớp ban đầu**
- **Mô hình:**



5.2 Đặc điểm

- **Giảm sự phụ thuộc giữa abstraction và implementation**
- **abstraction và implementation thay vì liên hệ bằng quan hệ kế thừa thì sẽ liên hệ với nhau thông qua object composition**
- **Cho phép ẩn các chi tiết implement từ client**
- **Dễ bảo trì và mở rộng về sau**

5.3 Mục Đích Sử Dụng

- **Tách ràng buộc giữa abstraction và implementation để có thể dễ dàng mở rộng độc lập nhau**
- **Thay đổi được thực thi trong implement mà không ảnh hưởng đến phía client**

5.4 Code khuôn mẫu

```
package bridge.pattern;
//Abtraction.java
public interface Abstraction {
    void operacion();
}

//Client.java
public class Client {
    public static void main(String[] args) {
        Abstraction[] abstracciones = new Abstraction[2];
        abstracciones[0] = new RefinedAbstraction(new ImplementorA());
        abstracciones[1] = new RefinedAbstraction(new ImplementorB());

        for(Abstraction abstraccion:abstracciones) {
            abstraccion.operacion();
        }
    }
}

//ConcreteImplementor.java
public class ConcreteImplementor implements Implementor{
    public void operacion() {
        System.out.println("This is implementacion");
    }
}

//Implementor.java
public interface Implementor {
    void operacion();
}

//RefinedAbstraction.java
public class RefinedAbstraction implements Abstraction{
```

```
34      private Implementor implementador;
35
36      public RefinedAbstraction(Implementor implementador){
37          this.implementador = implementador;
38      }
39
40      public void operacion(){
41          implementador.operacion();
42      }
43  }
```

5.5 Giải thích Design Pattern

- Theo Mô hình ct
- Theo code minh họa

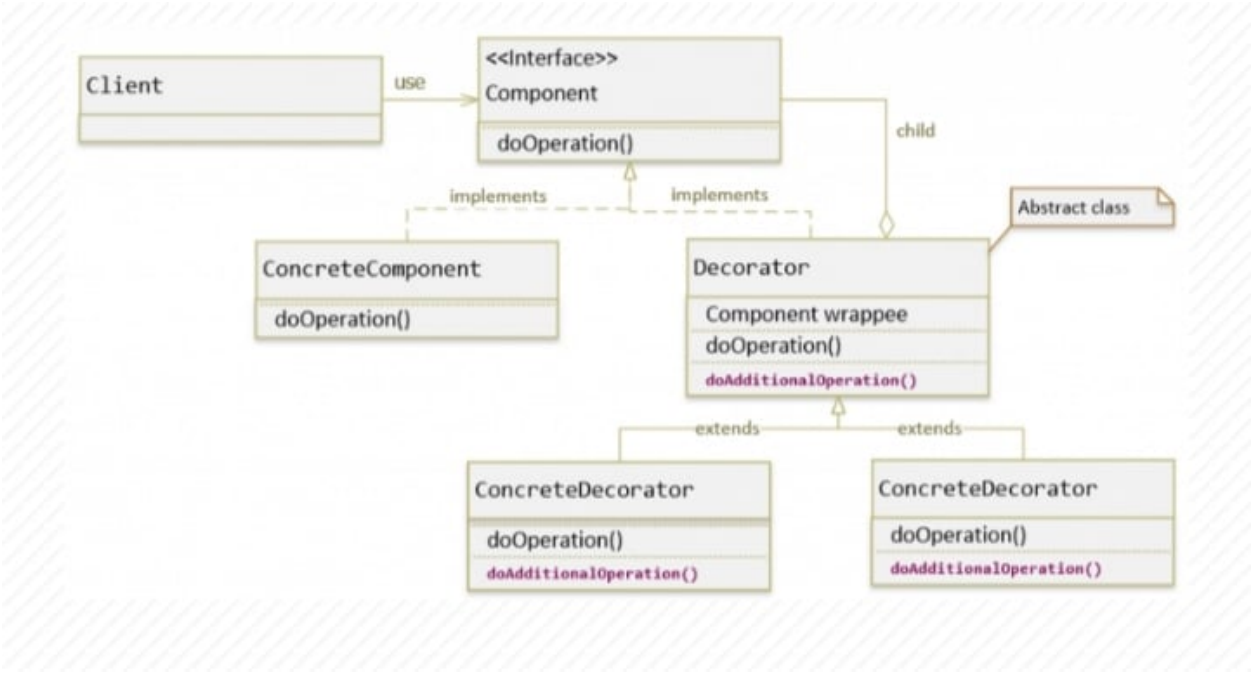
5.6 Ví Dụ Sử Dụng Trong Thực Tế

- <ý 1>
- <ý 2>

6 Decorator

6.1 Định Nghĩa Và Mô Hình Cấu Trúc

- **Decorator Pattern thay đổi một instance riêng lẻ của một class bằng cách tạo một class decorator bao bọc class gốc. Như vậy, việc thay đổi hoặc thêm chức năng của object decorator không ảnh hưởng đến cấu trúc hoặc chức năng của object ban đầu**
- **Mô hình:**



6.2 Đặc điểm

- **Có thể bổ sung những chức năng mới cho object**
- **Object gốc không thay đổi và không biết gì về những thứ được bổ sung cho nó**
- **Được thực hiện trong thời gian chạy và chỉ áp dụng cho một cá thể**
- **Không cần phải xây dựng class khổng lồ với mọi thứ bên trong**
- **Các object decorator độc lập nhau và có thể tổ hợp với nhau**

6.3 Mục Đích Sử Dụng

- **Thêm tính năng mới cho đối tượng mà không ảnh hưởng đến các đối tượng này**
- **Trong các trường hợp mà việc sử dụng kế thừa sẽ mất nhiều công sức trong việc viết code hoặc không thể mở rộng đối tượng bằng thừa kế**

6.4 Code khuôn mẫu

```
1  package decorator.pattern;
2  //Client.java
3  public class Client {
4      public static void main(String[] args) {
5          Component component = new ConcreteDecoratorOne(new
               ConcreteComponent());
6          component.doOperation();
7          System.out.println("Adding concrete component two...");
8          component = new ConcreteDecoratorOne(new
               ConcreteDecoratorTwo(new ConcreteComponent()));
9          component.doOperation();
10     }
11 }
12
13 //Component.java
14 public interface Component {
15     public void doOperation();
16 }
17
18 //ConcreteComponent.java
19 public class ConcreteComponent implements Component {
20     @Override
21     public void doOperation() {
22         System.out.println("Concrete Component doing operation");
23     }
24 }
```

```
25
26 //ConcreteDecorator.java
27 public class ConcreteDecorator extends Decorator {
28     public ConcreteDecorator(Component component) {
29         super(component);
30     }
31
32     @Override
33     public void doOperation() {
34         super.doOperation();
35         doAdditionalOperation();
36     }
37
38     public void doAdditionalOperation() {
39         System.out.println("Doing additional operation concrete decorator.");
40     }
41 }
42
43 //Decorator.java
44 public abstract class Decorator implements Component {
45     protected Component component;
46
47     public Decorator(Component component){
48         this.component = component;
49     }
50
51     @Override
52     public void doOperation() {
53         component.doOperation();
54     }
55 }
```

6.5 Giải thích Design Pattern

- Theo Mô hình ct

- Theo code minh họa

6.6 Ví Dụ Sử Dụng Trong Thực Tế

- <ý 1>

- <ý 2>

Phần IV

Nhóm hành vi - Behavioral Pattern

7 Observer

7.1 Định Nghĩa Và Mô Hình Cấu Trúc

- <ý 1>

- <ý 2>

7.2 Đặc điểm

- <đặc điểm 1>

- <đặc điểm 2>

7.3 Mục Đích Sử Dụng

- <mục đích 1>

- <mục đích 2>

7.4 Code khuôn mẫu

7.5 Giải thích Design Pattern

- Theo Mô hình ct

- Theo code minh họa

7.6 Ví Dụ Sử Dụng Trong Thực Tế

- <ý 1>

- <ý 2>

8 Iterator

8.1 Định Nghĩa Và Mô Hình Cấu Trúc

- <ý 1>

- <ý 2>

8.2 Đặc điểm

- <đặc điểm 1>

- <đặc điểm 2>

8.3 Mục Đích Sử Dụng

- <mục đích 1>

- <mục đích 2>

8.4 Code khuôn mẫu

8.5 Giải thích Design Pattern

- Theo Mô hình ct

- Theo code minh họa

8.6 Ví Dụ Sử Dụng Trong Thực Tế

- <ý 1>

- <ý 2>

9 Command

9.1 Định Nghĩa Và Mô Hình Cấu Trúc

- <ý 1>

- <ý 2>

9.2 Đặc điểm

- <đặc điểm 1>

- <đặc điểm 2>

9.3 Mục Đích Sử Dụng

- <mục đích 1>

- <mục đích 2>

9.4 Code khuôn mẫu

9.5 Giải thích Design Pattern

- Theo Mô hình ct

- Theo code minh họa

9.6 Ví Dụ Sử Dụng Trong Thực Tế

- <ý 1>

- <ý 2>

10 Strategy

10.1 Định Nghĩa Và Mô Hình Cấu Trúc

- <ý 1>

- <ý 2>

10.2 Đặc điểm

- <đặc điểm 1>

- <đặc điểm 2>

10.3 Mục Đích Sử Dụng

- <mục đích 1>

- <mục đích 2>

10.4 Code khuôn mẫu

10.5 Giải thích Design Pattern

- Theo Mô hình ct

- Theo code minh họa

10.6 Ví Dụ Sử Dụng Trong Thực Tế

- <ý 1>

- <ý 2>

Phần V

TÀI LIỆU THAM KHẢO

1 . Slides Bài giảng

2 . Head First Design Patterns

3 . https://sourcemaking.com/design-patterns

4 . https://github.com/LuisBurgs/design-patterns

5 . https://gpcoder.com/4164-gioi-thieu-design-patterns/