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

Mục lục

I Tổng quát 1

0.1. Định nghĩa 1

0.2. Đặc điểm 1

0.3. Phân loại 1

II Nhóm Khởi tạo - Creational Pattern 3

1. Singleton 4

1.1. Định Nghĩa Và Mô Hình Cấu Trúc 4

1.2. Đặc điểm 4

1.3. Mục Đích Sử Dụng. 4

1.4. Code khuôn mẫu 5

1.5. Giải thích Design Pattern 5

1.6. Ví Dụ Sử Dụng Trong Thực Tế 6

2. Builder. 7

2.1. Định Nghĩa Và Mô Hình Cấu Trúc 7

2.2. Đặc điểm 7

2.3. Mục Đích Sử Dụng. 7

2.4. Code khuôn mẫu 8

2.5. Giải thích Design Pattern 10

2.6. Ví Dụ Sử Dụng Trong Thực Tế 11

3. Factory 12

3.1. Định Nghĩa Và Mô Hình Cấu Trúc 12

3.2. Đặc điểm 12

3.3. Mục Đích Sử Dụng. 13

3.4. Code khuôn mẫu 13

3.5. Giải thích Design Pattern 14
3.6. Ví Dụ Sử Dụng Trong Thực Tế 15

III Nhóm Cấu trúc - Structural Pattern 17
4. Adapter 18
4.1. Định Nghĩa Và Mô Hình Cấu Trúc 18
4.2. Đặc điểm 18
4.3. Mục Đích Sử Dụng. 18
4.4. Code khuôn mẫu 19
4.5. Giải thích Design Pattern 20
4.6. Ví Dụ Sử Dụng Trong Thực Tế 20
5. Bridge 22
5.1. Định Nghĩa Và Mô Hình Cấu Trúc 22
5.2. Đặc điểm 22
5.3. Mục Đích Sử Dụng. 22
5.4. Code khuôn mẫu 23
5.5. Giải thích Design Pattern 24
5.6. Ví Dụ Sử Dụng Trong Thực Tế 25
6. Decorator. 27
6.1. Định Nghĩa Và Mô Hình Cấu Trúc 27
6.2. Đặc điểm 27
6.3. Mục Đích Sử Dụng. 28
6.4. Code khuôn mẫu 28
6.5. Giải thích Design Pattern 29
6.6. Ví Dụ Sử Dụng Trong Thực Tế 30

IV Nhóm hành vi - Behavioral Pattern 32
7. Observer 33
7.1. Định Nghĩa Và Mô Hình Cấu Trúc 33
7.2. Đặc điểm 33

7.3. Mục Đích Sử Dụng. 33
7.4. Code khuôn mẫu 34
7.5. Giải thích Design Pattern 36
7.6. Ví Dụ Sử Dụng Trong Thực Tế 36
8. Iterator 38
8.1. Định Nghĩa Và Mô Hình Cấu Trúc 38
8.2. Đặc điểm 38
8.3. Mục Đích Sử Dụng. 39
8.4. Code khuôn mẫu 39
8.5. Giải thích Design Pattern 39
8.6. Ví Dụ Sử Dụng Trong Thực Tế 39
9. Command 41
9.1. Định Nghĩa Và Mô Hình Cấu Trúc 41
9.2. Đặc điểm 41
9.3. Mục Đích Sử Dụng. 42
9.4. Code khuôn mẫu 42
9.5. Giải thích Design Pattern 42
9.6. Ví Dụ Sử Dụng Trong Thực Tế 42
10. Strategy 43
10.1. Định Nghĩa Và Mô Hình Cấu Trúc 43
10.2. Đặc điểm 43
10.3. Mục Đích Sử Dụng. 43
10.4. Code khuôn mẫu 44
10.5. Giải thích Design Pattern 45
10.6. Ví Dụ Sử Dụng Trong Thực Tế 45

V THAM KHẢO 47
11. Tài liệu tham khảo. 47
12. Link Project bài báo cáo 47

Phần I

Tổng quát

0.1. Định nghĩa

- **Design pattern là một kỹ thuật lập trình quan trọng trong lập trình hướng đối tượng cung cấp các khuôn mẫu thiết kế có thể dễ dàng áp dụng. 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 và có thể dùng để giải quyết các vấn đề tương tự**

0.2. Đặc điểm

- **Design pattern có thể thực hiện được ở bất cứ ngôn ngữ lập trình nào sử dụng OOP. 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).**
- **Việc có khuôn mẫu sẵn nên tăng tốc độ lập trình, chỉ cần hiểu và áp dụng, không cần nghiên cứu mới, những người cùng team lập trình cũng dễ đọc dễ hiểu cũng như giảm bớt rủi ro do chưa được tối ưu hay có lỗi do nghiên cứu chưa hoàn thiện nên bớt tốn kém thời gian.**
- **Tái sử dụng các mã nguồn, dễ dàng nâng cấp bảo trì để đáp ứng nhu cầu dự án**

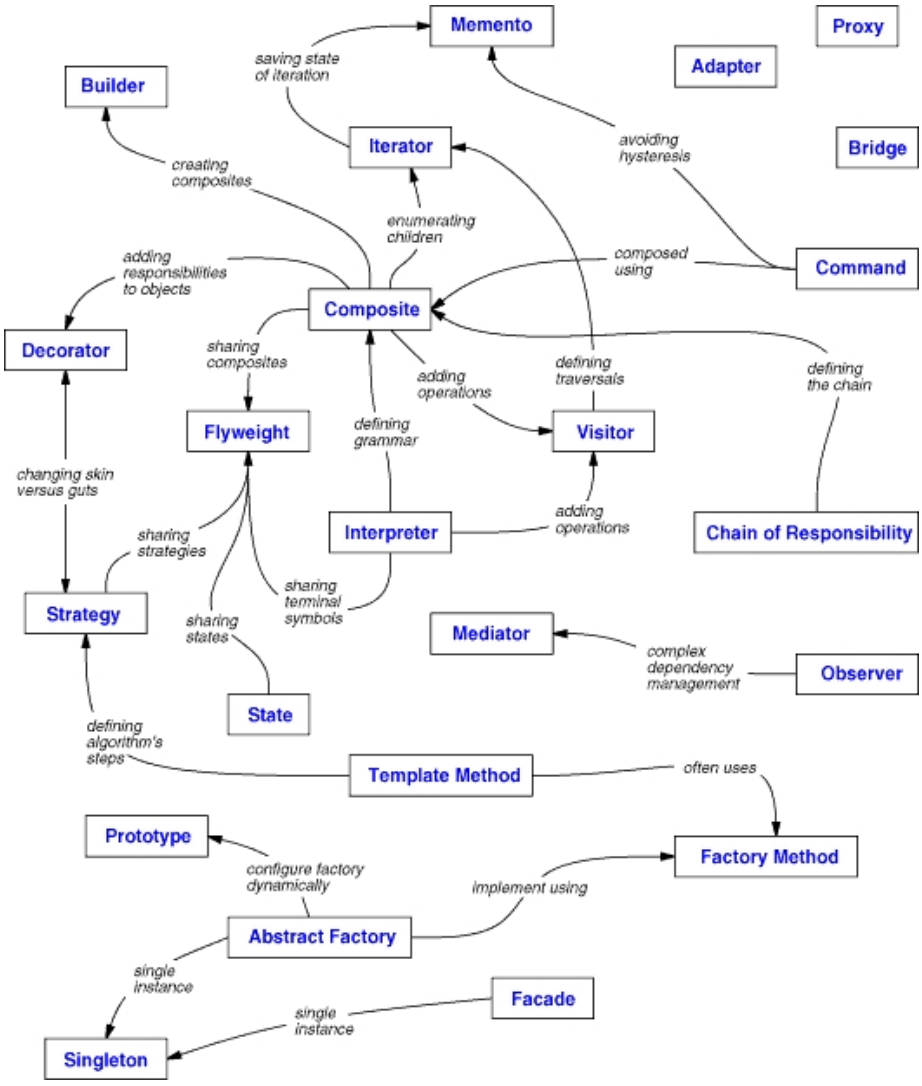
0.3. Phân loại

- **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.**

- Mối quan hệ giữa các design pattern:



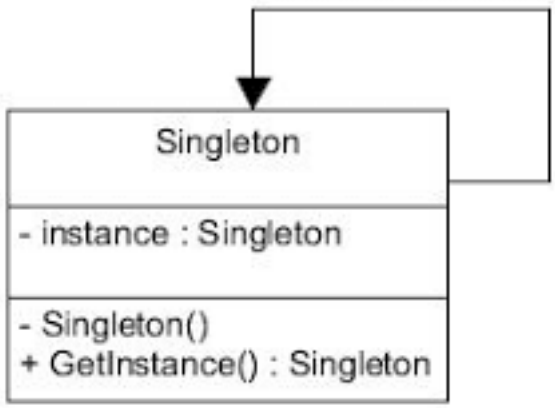
Phần II

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

```
1  public class Singleton {
2      private static final Singleton INSTANCE = new Singleton();
3      private Singleton() {}
4      public static Singleton getInstance(){
5          if(INSTANCE == null){
6              INSTANCE = new Singleton();
7          }
8          return INSTANCE;
9      }
10 }
```

1.5. Giải thích Design Pattern

- 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.

- 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

- **Ví dụ sử dụng Singleton pattern Simple Logger**
  https://github.com/blaplafla13th/design-patterns/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.**

- **Đầu tiên tạo một biến thành viên hằng private là 1 instance Log mới và hàm dựng private của Log để giới hạn được số lượng đối tượng được tạo chỉ có duy nhất một đối tượng.**

- **Tạo accessor getInstance với if để kiểm tra xem đối tượng Log có null không, nếu null thì tạo lại, sau đó trả về đối tượng Log được khai báo ở trên. Ta thêm các biến thành viên phục vụ cho việc ghi file như File, ghi ngày như SimpleDateFormat**

- **Hàm check sẽ kiểm tra xem file được truyền vào qua tham số có tồn tại không, không tồn tại thì sẽ tạo file mới, hai hàm read và write sẽ đọc và ghi file theo đường dẫn nhập từ tham số.**

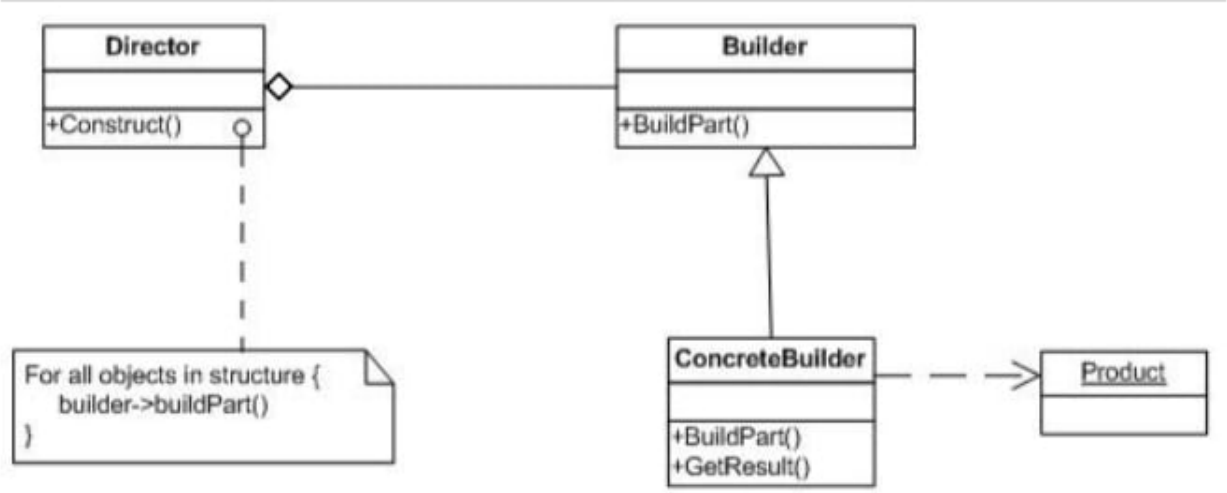
- **Sang bên test, chúng ta gọi đối tượng Log bằng cách sử dụng getInstance. Khai báo hằng LOG_PATH để ghi đường dẫn của file log. Tạo thử 1 lỗi rồi ghi lỗi đó ra log bằng lệnh write rồi đọc lại bằng lệnh read**

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

- 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 vềvề các 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.

- 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 pattern
https://github.com/blaplafla13th/design-patterns/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 đóng vai trò builder 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 kế thừa AccountBuilder của signIn và signUp đóng vai trò ConcreteBuilder để tạo form điền ở step1 và xử lý form ở step2.

- Chúng ta tạo một AccountDirector đóng vai trò Director 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 tài 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 Director 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.

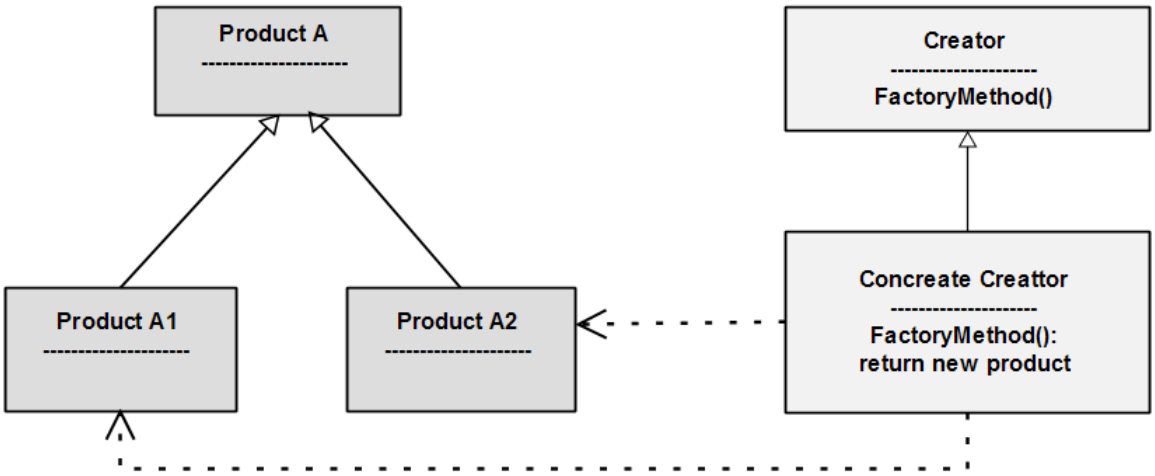
3. Factory

3.1. Định Nghĩa Và Mô Hình Cấu Trúc

- Factory Pattern hay Factory Method 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 và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đối tượng một cách linh hoạt hơn. Nôm na Factory như phân xưởng sản xuất 1 sản phẩm

- Abstract Factory là bản mở rộng của Factory khi object chúng ta khởi tạo bao gồm nhiều sử dụng nhiều những object khác liên quan. Trong Abstract Factory pattern, một interface có nhiệm vụ tạo ra các Factory của các object có liên quan tới nhau mà không cần phải chỉ ra trực tiếp các class của object. Nôm na Abstract Factory như một nhà máy với nhiều phân xưởng là các Factory Method

- Mô hình cho Factory:



3.2. Đặc điểm

- Người dùng không biết logic thực sự được khởi tạo bên dưới phương thức factory

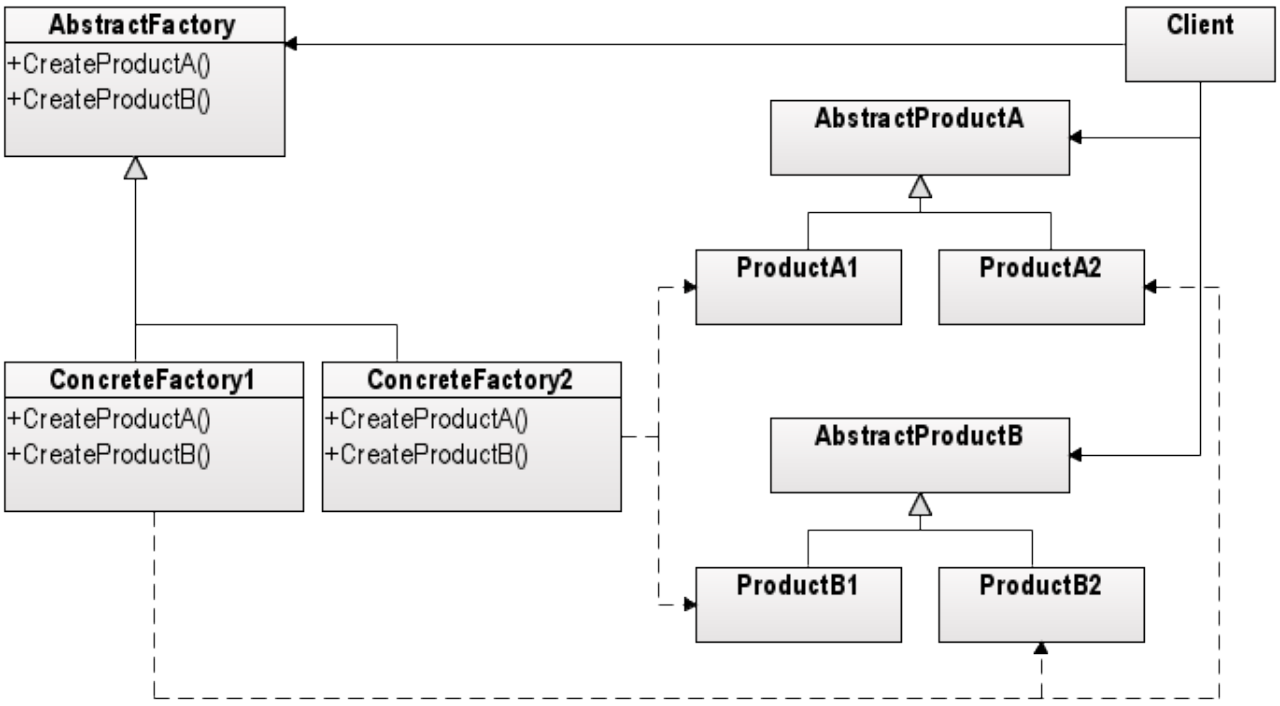
- Cho phép các lớp con chọn kiểu đối tượng cần tạo

- 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
- Khi Factory có quá nhiều trường hợp, sử dụng Abstract Factory giúp phân tách điều kiện đơn giản hóa logic
- Mô hình cho Abstract Factory:



3.4. Code khuôn mẫu

```
1 package factory.pattern;
2 //Factory.java
3 public class Factory {
4     public Product createProduct(String productType){
```

```
5           switch (productType){
6               case "a": return new ProductA();
7               case "b": return new ProductB();
8                ...
9               default: return null;
10          }
11      };
12  }
13
14  //Product.java
15  public abstract class Product {
16      ...
17  }
18
19  //ProductA.java
20  public class ProductA extends Product{
21      public ProductA(){
22          ...
23      }
24      ...
25  }
26
27  //ProductB, ... similar as class ProductA
28
29  //Client.java
30  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

- 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 ConcreteCreator 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**

+ **Đối với Abstract Factory, sẽ có nhiều Factory ConcreteCreator hay hơn kế thừa từ AbstractFactory. Client sẽ yêu cầu AbstractFactory và AbstractFactory sẽ gọi đến Factory tương ứng để thực hiện các hàm dựng các đối tượng theo chỉ định**

- 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 pattern
https://github.com/blaplafla13th/design-patterns/tree/main/src/factory

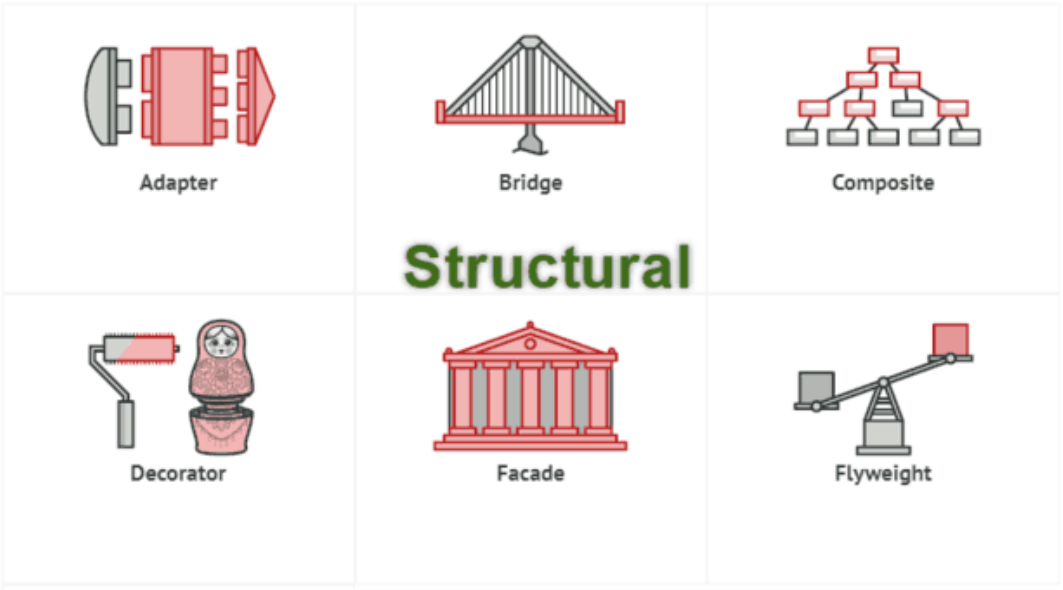
- chúng ta tạo 1 package paymentmethod để chứa interface PaymentMethod và các class implements từ PaymentMethod. Mỗi class đều chứa tên phương thức, số tài khoản, tên người dùng và đoạn code ẩn để thanh toán

- 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

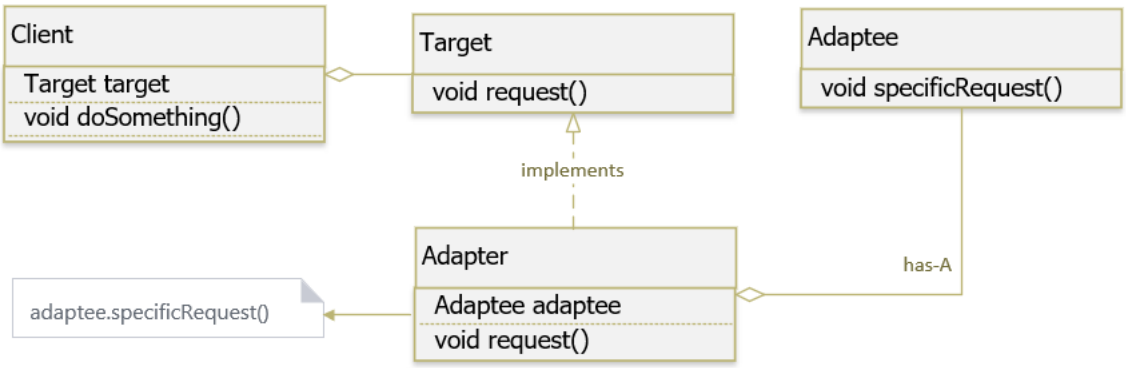
Nhóm Cấu trúc - Structural Pattern



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

```
package adapter.pattern;
//Adaptee.java
public class Adaptee {
    ...
    public void specificRequest() {
        ...
    }
}

//Adapter.java
public class Adapter implements Target {
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee){
        this.adaptee = adaptee;
    }

    @Override
    public void request() {
        adaptee.specificRequest();
    }
}

//Client.java
public class Client {
    public static void main(String[] args) {
        Target target = new Adapter(new Adaptee());
        target.request();
    }
}

```

```
32 //Target.java
33 public interface Target {
34     public void request();
35 }
```

4.5. Giải thích Design Pattern

- Theo Mô hình

  + Adaptee: là một class hoặc interface không tương thích, cần được tích hợp vào trong client.

  + 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. Adapter thường chứa adaptee hoặc kế thừa từ adaptee. Các phương thức không cần thiết sẽ bị interface ẩn đi. Nếu adaptee là 1 object chứa bên trong adapter, adapter có thể làm được nhiều việc được với nhiều adaptee hơn thay vì chỉ một adapter với 1 adaptee

  + Target: một interface chứa các chức năng được sử dụng bởi Client,Client: lớp sử dụng các đối tượng có interface Target.

- Theo code khuôn mẫu

  + 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-patterns/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 Language và target Vietnamese**

- **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 quả 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**

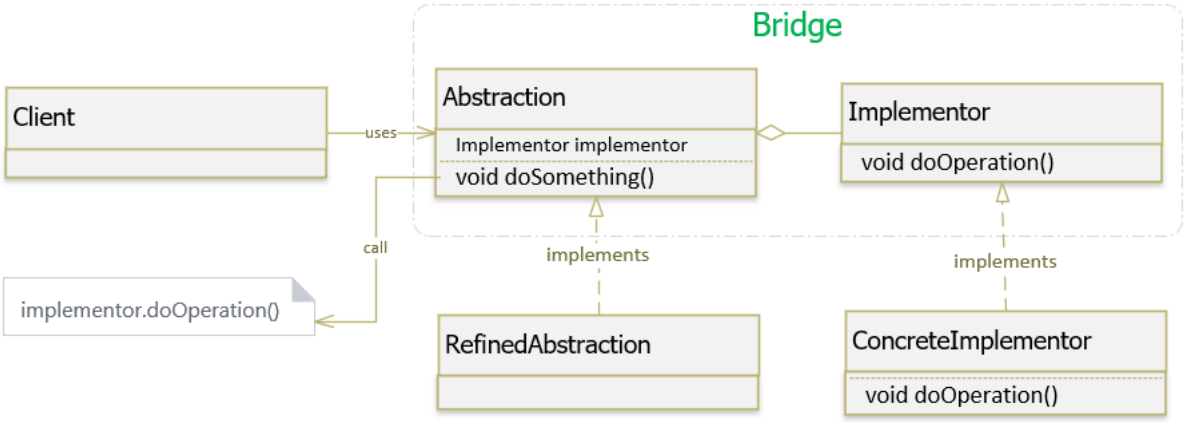
- **Trong thực tế cách này được dùng kết hợp với factory pattern để dịch các chương trình sử dụng bộ các keyword:value để in thông tin, keyword:value thường được lưu ra file riêng, sẽ có hàm đọc file này để ghi vào mảng languagePack ở trên, cách này không dịch được các thông tin cứng được in trực tiếp 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**

```
1  package bridge.pattern;
2  //Abstraction.java
3  public interface Abstraction {
4      void operation();
5  }
6
7  //Client.java
8  public class Client {
9      public static void main(String[] args) {
10         Abstraction[] abstractions = new Abstraction[2];
11         abstractions[0] = new RefinedAbstraction(new ImplementorA());
12         abstractions[1] = new RefinedAbstraction(new ImplementorB());
13
14         for(Abstraction abstraction:abstractions) {
15             abstraction.operation();
16         }
17     }
18 }
19
20 //ImplementorA.java
21 public class ImplementorA implements Implementor{
22     public void operation() {
23         System.out.println("This is implementation");
24     }
25 }
26
27 //ImplementorB, ... similar as class ImplementorA
28
29 //Implementor.java
30 public interface Implementor {
31     void operation();
32 }
33
```

```java
//RefinedAbstraction.java
public class RefinedAbstraction implements Abstraction{
    private Implementor implementator;

    public RefinedAbstraction(Implementor implementator){
        this.implementator = implementator;
    }

    public void operation(){
        implementator.operation();
    }
}
```

5.5. Giải thích Design Pattern

- Theo Mô hình

  + Client: đại diện cho khách hàng sử dụng các chức năng thông qua Abstracion.

  + Abstraction : định ra một abstract interface quản lý việc tham chiếu đến đối tượng hiện thực cụ thể (Implementor).

  + Refined Abstraction (AbstractionImpl) : hiện thực (implement) các phương thức đã được định ra trong Abstraction bằng cách sử dụng một tham chiếu đến một đối tượng của Implementer.

  + Implementor : định ra các interface cho các lớp hiện thực. Thông thường nó là interface định ra các tác vụ nào đó của Abstraction.

  + ConcreteImplementor : hiện thực Implementor interface.

- Theo code khuôn mẫu

  + Client tạo 1 mảng Abstraction mỗi phần tử của Abstraction tạo một instance của RefinedAbstraction sau đó gọi operation từ từng phần tử

  + Interface Abstraction chứa hàm operation để sử dụng ở Client. RefinedAbstraction implements từ Abstraction. Trong RefinedAbstraction có 1 object thành viên là Implementor. Hàm operation kế thừa từ Implement sẽ chạy hàm operation của Implementor.

+ **Interface Implementor chứa khuôn mẫu hàm chạy của các Implementor. Các ImplementorA,B... implement từ Implemention**

+ **Khi Client chạy, mảng Abstraction được khởi tạo, tạo mới Implementor làm đối của hàm dựng RefinedAbstraction. Khi Client gọi operation của Abstraction, RefinedAbstraction sẽ gọi đến operation của Implementor.**

5.6. Ví Dụ Sử Dụng Trong Thực Tế

- **Sử dụng bridge pattern để tạo chương trình hoạt động trên đa nên tảng** https://github.com/blaplafla13th/design-patterns/tree/main/src/brigde

- **Chúng ta có 1 interface là Program làm implementor, 2 Class Class và Java kế thừa từ Program. Đại khái chương trình sẽ in nội dung file .class và file .java ra màn hình**

- **Có một vấn đề là phân cấp đường dẫn trong hai nhóm hệ điều hành Windows và Unix khác nhau và lệnh xóa dữ liệu đã in trên console khác nhau vậy nên ta cần một bridge để có các biến, lệnh tương ứng trên mỗi hệ điều hành. ở đây abstract class Bridge là Abstraction sẽ có 1 hàm run để gọi hàm của Program, 1 hàm clear để clear console hằng public final char SEPARATE để lưu dấu phân cấp của OS và 2 class Windows và Unix trong package bridge.os đóng vai trò RefinedAbstraction.**

- **Trong mã nguồn java, giá trị SEPARATE này sẽ được hệ điều hành quy định và có thể dùng java.util.Properties getProperty("file.separator") để gọi ra. Abstract class java.io.FileSystem là 1 Implementor chứa hằng char slash để lưu giá trị này. WinNTFileSystem ở bản jdk Windows và UnixFileSystem ở bản jdk Unix là 2 ConcreteImplementor sẽ gọi hàm getProperties để lấy giá trị này. Sau đó java.io.File là Abstraction lấy tên OS, rồi tạo 1 đối tượng hằng FileSystem với OS tương ứng hằng separate gán bằng giá trị slash lấy từ FileSystem.**

- **Ở đây tham số truyền vào của Bridge là dấu phân cấp tương ứng của hệ điều hành và Program. Trong Class của OS dấu phân cấp được quy định sẵn và truyền thêm tham số của Program. Windows hoặc Unix sẽ gọi hàm dựng của super class với 2 đối số là dấu phân cấp và Program. Hàm clear được viết tương ứng với OS và hàm readFile sẽ sẽ gọi đến hàm readFile của Program**

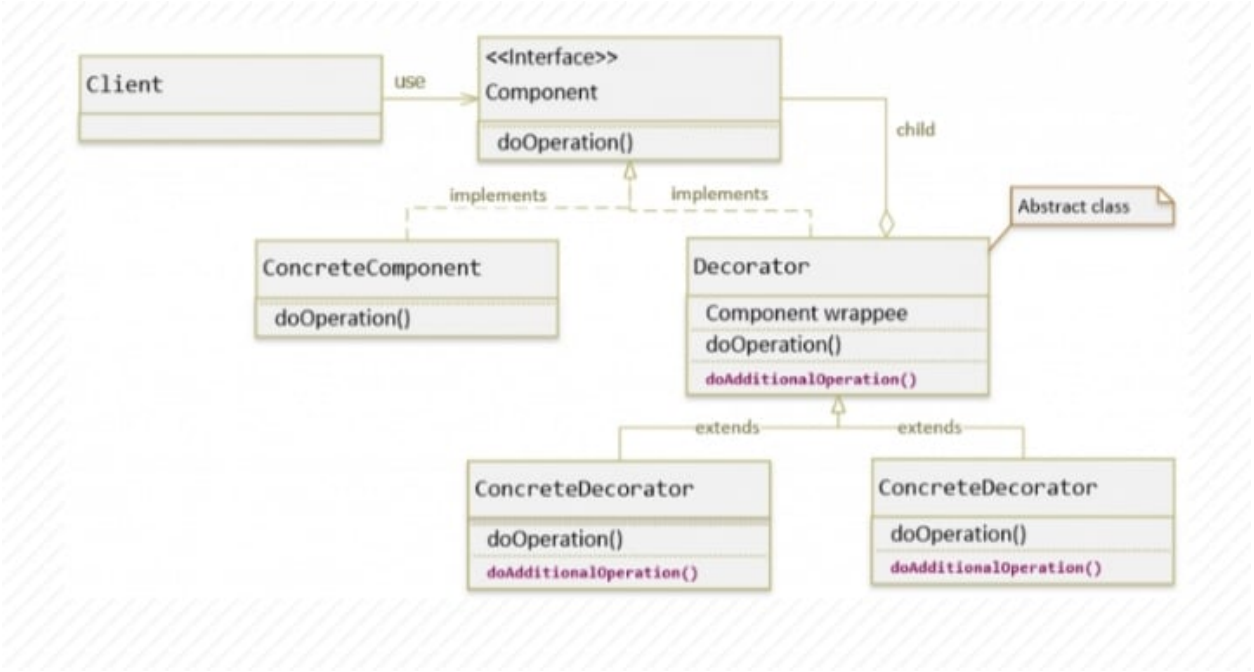
- **Về phía Client, tạo ra 2 instance Bridge gồm bridgeJava và bridgeClass, lấy tên của OS. Với mỗi OS sẽ tạo ra cặp bridge tương ứng. Qua đó chúng ta có thể dùng hàm trong Bridge để gọi các biến và các hàm trong Program trên hai nhóm OS như code bên dưới**
- **Trong lúc đóng gói phần mềm, thường thì phần mềm chỉ hoạt động trên một nền tảng nhất định, các thư viện thừa cửa hệ điều hành khác sẽ được loại bỏ**

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**
- **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 //Other ConcreteDecorator similar as this
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

  + Component: là một interface quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.

+ **ConcreteComponent: là lớp hiện thực (implements) các phương thức của Component.**
+ **Decorator: là một abstract class dùng để duy trì một tham chiếu của đối tượng Component và đồng thời cài đặt các phương thức của Component interface.**
+ **ConcreteDecorator : là lớp hiện thực (implements) các phương thức của Decorator, nó cài đặt thêm các tính năng mới cho Component.**
+ **Client : đối tượng sử dụng Component.**

- Theo code khuôn mẫu

+ **Component là interface tổng để class ConcreteComponent và abstract class Decorator kế thừa. Tất cả đều có chung 1 hành động là doOperation. ConcreteComponent là một class chứa các hàm sẽ được quy định mặc định như ở đây là doOperation.**
+ **Biến thành viên của abstract class Decoration có 1 instance của Component ở trạng thái protected cùng hàm dựng có tham số là Component. Các hàm chung mà các class con của Decoration sẽ sử dụng đều gọi hàm từ instance của Component tức sẽ gọi hàm trong ConcreteComponent. Class con extend từ Decorator là ConcreteDecorator sẽ có hàm dựng với tham số là 1 instance Component và gọi đến hàm khởi tạo của hàm cha qua super. Hàm doOperation của các class con của Decorator có thể thoải mái tùy biến.**
+ **Do đã có ConcreteComponent làm chung, Client hoàn toàn có thể khai báo đối tượng kiểu ConcreteComponent hoặc ConcreteDecoration và sau đó có thể chuyển qua lại giữa các ConcreteDecoration.**

6.6. Ví Dụ Sử Dụng Trong Thực Tế

- Hệ thống phân quyền sử dụng decorator pattern
https://github.com/blaplafla13th/design-patterns/tree/main/src/decorator

- Tạo một interface User đóng vai trò Component với các setter và getter cơ bản các giá trị như PersonalInformation, listAction, hàm để xem (readListAction), thêm (createListAction), sửa (updateListAction), xóa (deleteListAction) (CRUD) ArrayList listAction, hàm thực thi action

- **Class SimpleUser đóng vai trò ConcreteComponent với 2 biến thành viên là String PersonalInformation và ArrayList listAction và chi tiết cho các hàm abstract từ interface**

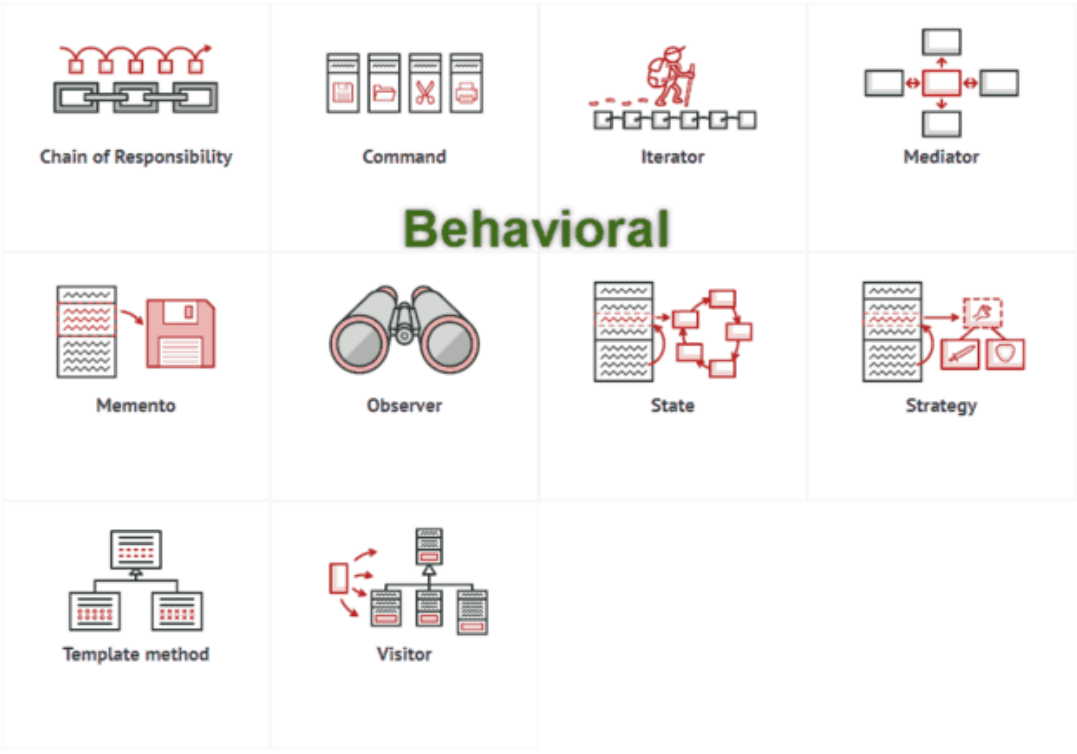
- **AAbstract class ExtendUser đóng vai trò Decorator để chúng ta có thể mở rộng từ SimpleUser thành SuperUser và NormalUser. Class sẽ compose với interface User, có một biến thành viên là instance của User với kiểu protected để có thể sử dụng ở các hàm mở rộng phía dưới và hàm dựng có tham số là instance của User,hàm dựng không tham số của ExtendUser sẽ tạo mới SimpleUser. Mọi hàm trong ExtendUser kế thừa từ User sẽ được gọi lại thông qua instance user để kế thừa code đã được viết ở SimpleUser. Hai hàm abstract action và resetActionList để các class extend từ class này bắt buộc có**

- **Hai hàm SuperUser và NormalUser sẽ có 2 hàm action khác nhau, 2 hàm resetAction khác nhau. Instance user đang ở protected nên hai hàm con có thể sử dụng thoải mái. Hàm dựng không biến của cả 2 sẽ gọi đến hàm dựng không biến của hàm cha để khởi tạo instance user để sử dụng resetListAction. Hàm dựng có tham số user và 1 boolean để có thể chuyển đổi kiểu giữa 2 class với boolean để chạy tùy chọn resetActionList. Khi mở rộng thêm các loại User chúng ta chỉ cần tùy biến 2 hàm resetActionList và action**

- **Ở Client chúng ta có thể sử dụng các hàm được quy định sẵn ở interface User, các hàm dựng. Để chuyển đổi giữa các dạng user chúng ta sử dụng hàm dựng với đối là user cần chuyển**

Phần IV

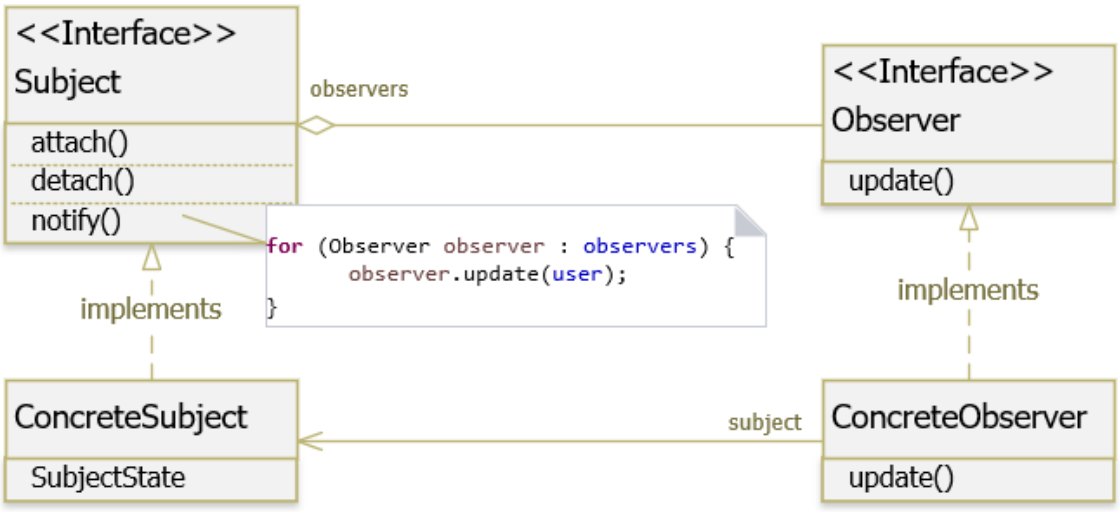
Nhóm hành vi - Behavioral Pattern



7. Observer

7.1. Định Nghĩa Và Mô Hình Cấu Trúc

- **Observer Pattern định nghĩa mối phụ thuộc một - nhiều giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất cả các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.**
- **Observer có thể đăng ký với hệ thống. Khi hệ thống có sự thay đổi, hệ thống sẽ thông báo cho Observer biết. Khi không cần nữa, mẫu Observer sẽ được gỡ khỏi hệ thống.**
- **Mô hình:**



7.2. Đặc điểm

- **Cho phép thay đổi Subject và Observer một cách độc lập, có thể tái sử dụng các Subject mà không cần tái sử dụng các Observer và ngược lại.**
- **Cho phép thêm Observer mà không sửa đổi Subject hoặc Observer khác.**
- **Một đối tượng có thể thông báo đến một số lượng không giới hạn các đối tượng khác.**

7.3. Mục Đích Sử Dụng

- **Thường được sử dụng trong mối quan hệ một - nhiều giữa các object với**

nhau. Trong đó, một đối tượng thay đổi và muốn thông báo cho tất cả các object liên quan biết về sự thay đổi đó.

- Sử dụng trong ứng dụng broadcast - type communication

- Sử dụng để quản lý sự kiện (Event management)

- Sử dụng trong mẫu mô hình MVC (Model View Controller Pattern)

7.4. Code khuôn mẫu

```
1  package observer.pattern;
2  //Observer.java
3  public interface Observer {
4      void update();
5  }
6
7  //Subject.java
8  public interface SubjectA {
9      public void add(Observer o);
10
11     public int getState();
12
13     public void setState(int value);
14
15     private void execute();
16 }
17
18 //SubjectA.java
19 public class SubjectA implements Subject{
20     private List<Observer> observers = new ArrayList<>();
21     private int state;
22
23     public void add(Observer o) {
24         observers.add(o);
25     }
26
```

```
27      public int getState() {
28          return state;
29      }
30
31      public void setState(int value) {
32          this.state = value;
33          execute();
34      }
35
36      private void execute() {
37          for (Observer observer : observers) {
38              observer.update();
39          }
40      }
41  }
42
43  //SubjectB,C similar as SubjectA
44
45  //ObserverA.java
46  public class ObserverA implements Observer {
47      protected Subject subject;
48      public ObserverA(Subject subject) {
49          this.subject = subject;
50          this.subject.add(this);
51      }
52
53      public void update() {
54          ...
55      }
56  }
57
58  //ObserverB,C similar as ObserverA
```

7.5. Giải thích Design Pattern

- Theo Mô hình

+ Subject: chứa danh sách các observer, cung cấp phương thức để có thể thêm và loại bỏ observer.

+ Observer: định nghĩa một phương thức update() cho các đối tượng sẽ được subject thông báo đến khi có sự thay đổi trạng thái.

+ ConcreteSubject: cài đặt các phương thức của Subject, lưu trữ trạng thái danh sách các ConcreateObserver, gửi thông báo đến các observer của nó khi có sự thay đổi trạng thái.

+ ConcreteObserver: cài đặt các phương thức của Observer, lưu trữ trạng thái của subject, thực thi việc cập nhật để giữ cho trạng thái đồng nhất với subject gửi thông báo đến.

- Theo code khuôn mẫu

+ Subject: cung cấp phương thức để có thể thêm observer, lấy giá trị, đặt giá trị cho observer và đồng bộ các observer. SubjectA,B,... sẽ có ArrayList các observer sẽ đồng bộ và state là dữ liệu lưu trữ trạng thái danh sách mà observer sẽ đồng bộ. Các phương thức kế thừa ở trên như add để thêm, getState để lấy giá trị, setState để đặt giá trị và execute để gọi tất cả các phần tử update. Nếu như SubjectA,B,... giống nhau chúng ta có thể để Subject là abstract class để extend, thậm chí giống hệt nhau thì chỉ cần để mỗi class Subject để tái sử dụng tối đa code

+ Observer: định nghĩa một phương thức update() cho các đối tượng sẽ được subject thông báo đến khi có sự thay đổi trạng thái. ObserverA,B implements từ observer có hàm dựng với tham số bắt buộc là Subject để khi làm việc các object sẽ có chung 1 đối tượng thực hiện ra lệnh update trạng thái. Và tương tự với Subject có thể rút gọn code bằng cách sử dụng abstract class hoặc class

7.6. Ví Dụ Sử Dụng Trong Thực Tế

- Hệ thống chat đơn giản sử dụng Observer pattern
https://github.com/blaplafla13th/design-patterns/tree/main/src/observer

- Ở đây thay vì dùng Interface để đơn giản và tái sử dụng được nhiều code nên sử dụng class. Ở đây có User đóng vai trò là Subject vừa là Concrete

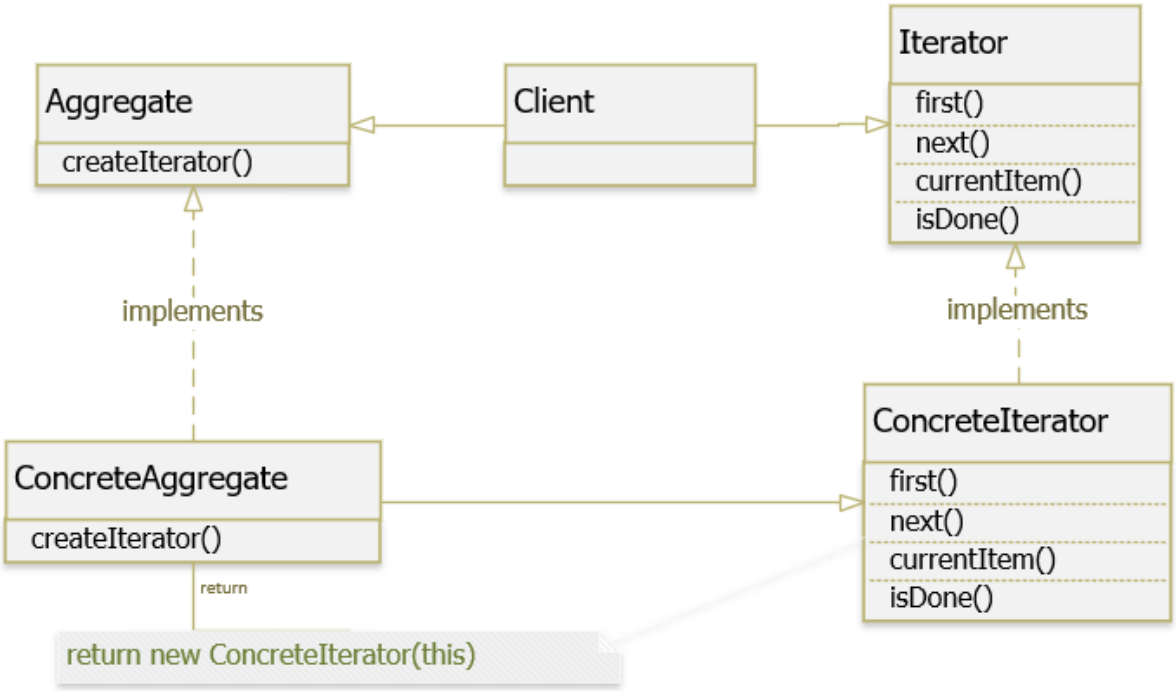
Subject. Observer và Concrete sẽ là ChatUpdater. Server đóng vai trò giao tiếp với người dùng. Trong ứng dụng thực tế thì ngoài 3 class trên do user ở 2 máy khác nhau nên cần 1 class để chuyển dữ liệu qua lại bằng socket qua mạng và các tin nhắn được lưu file riêng. Code này để chạy thử nên tạo 2 user1,2 làm ConcreteSubject và có thể tự chat giữa hai user

- **ChatUpdater sẽ có 1 ArrayList<User> user để lưu list các người dùng tham gia luồng chat này. String message để lưu tin nhắn cuối cùng nhận được. Hàm add với tham số là User sẽ thêm user vào ArrayList users. Trong hàm add có 1 lệnh users.add(user) để thêm user và 1 lệnh addChat để thông báo có user mới tham gia. Hàm getChat sẽ trả về message còn hàm addChat với tham số truyền vào là String sẽ cập nhật message và chạy hàm execute. Trong hàm execute, 1 vòng for để gọi mọi User trong user chạy hàm update**
- **User sẽ có biến thành viên String userName để lưu tên người dùng, ChatUpdater để có luồng chat nhận và gửi thông tin và ArrayList<String> listMessage để lưu các tin nhắn đã nhận. Thêm các hàm như hàm update để nhận dữ liệu từ hàm getChat() ở ChatUpdater cho vào listMessage, hàm getAllChat để in ra tất cả chat đã lưu, getLatestChat để in tin nhắn cuối cùng nhận được và hàm addChat sẽ gọi ChatUpdater ở phía trên gọi đến hàm addChat để gửi tin nhắn. Thêm 2 hàm để xem và sửa tên người dùng. Hàm dựng sẽ gồm 2 tham số là userName và ChatUpdater, trong hàm dựng sẽ thêm lệnh chatUpdater(this) để thêm người dùng vào luồng chat**
- **Tiếp theo đến Server, chúng ta tạo một chatUpdater để tên là chatThread. Tạo mới User tên user1 với 2 đối số là user với giá trị là "Blaplafla" và chatUpdater là chatThread. Khi chạy user1.getLatestChat sẽ nhận được câu chào của chatUpdater khi thêm user mới, tương tự với user2. Dùng chatThread.addChat để xem đồng bộ giữa 2 user. Khi getAllMessage, user1 sẽ có 3 tin là chào user1, chào user2, và chào mừng đến với server còn user2 do được thêm vào sau user1 nên không có tin chào user1 tức toàn bộ tin nhắn phía trước. Sau đó chúng ta chat thử, dùng user1.addChat và xem toàn bộ tin nhắn của user2 sẽ thấy tin nhắn của user1 ở dưới**

8. Iterator

8.1. Định Nghĩa Và Mô Hình Cấu Trúc

- **Iterator Pattern cho phép xử lý nhiều loại tập hợp khác nhau bằng cách truy cập từng phần tử của tập hợp với cùng một phương pháp, cùng một cách thức định sẵn mà không cần phải hiểu rõ về những chi tiết bên trong những tập hợp này.**
- **Iterator thường được viết trong Java như là những lớp độc lập - một lớp chỉ có duy nhất một công việc để làm**
- **Mô hình:**



8.2. Đặc điểm

- **Tách phần cài đặt các phương thức của tập hợp (collection) và phần duyệt qua các phần tử (iterator) theo từng class riêng lẻ**
- **Có thể implement các loại collection mới và iterator mới, sau đó chuyển chúng vào code hiện có**

- Có thể truy cập song song trên cùng một collection vì mỗi đối tượng iterator có chứa trạng thái riêng của nó

8.3. Mục Đích Sử Dụng

- Truy cập nội dung của đối tượng trong tập hợp mà không cần biết nội dung cài đặt bên trong

- Hỗ trợ truy xuất nhiều loại tập hợp khác nhau

- Cung cấp một interface duy nhất, đơn giản, nhất quán để làm việc với các tập hợp khác nhau

8.4. Code khuôn mẫu

8.5. Giải thích Design Pattern

- Theo Mô hình

+ Aggregate: là một interface định nghĩa định nghĩa các phương thức để tạo Iterator object.

+ ConcreteAggregate: cài đặt các phương thức của Aggregate, nó cài đặt interface tạo Iterator để trả về một thể hiện của ConcreteIterator thích hợp.

+ Iterator: là một interface hay abstract class, định nghĩa các phương thức để truy cập và duyệt qua các phần tử.

+ ConcreteIterator: cài đặt các phương thức của Iterator, giữ index khi duyệt qua các phần tử.

+ Client: đối tượng sử dụng Iterator Pattern, nó yêu cầu một iterator từ một đối tượng collection để duyệt qua các phần tử mà nó giữ. Các phương thức của iterator được sử dụng để truy xuất các phần tử từ collection theo một trình tự thích hợp.

- Theo code khuôn mẫu

8.6. Ví Dụ Sử Dụng Trong Thực Tế

- <ý 1>

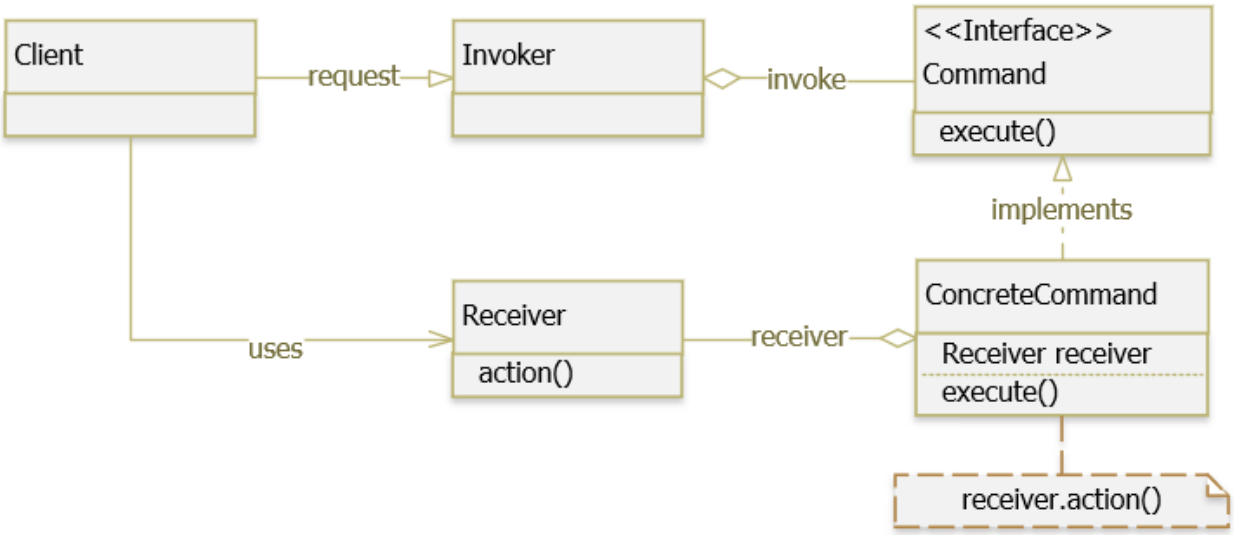
- <ý 2>

9. Command

9.1. Định Nghĩa Và Mô Hình Cấu Trúc

- **Command Pattern chuyển yêu cầu thành đối tượng độc lập, có thể được sử dụng để tham số hóa các đối tượng với các yêu cầu khác nhau như log, queue (undo/redo), transtraction. Nói cách khác, nó cho phép tất cả những request gửi đến object được lưu trữ trong chính object đó dưới dạng một object command (một class trung gian lưu trữ các câu lệnh và trạng thái của object tại một thời điểm nào đó**

- **Mô hình:**



9.2. Đặc điểm

- **Dễ dàng thêm các Command mới trong hệ thống mà không cần thay đổi trong các lớp hiện có.**

- **Tách đối tượng gọi operation từ đối tượng thực sự thực hiện operation. Giảm kết nối giữa Invoker và Receiver.**

- **Dễ dàng chuyển dữ liệu dưới dạng đối tượng giữa các thành phần hệ thống.**

- **Tăng số lớp cho từng lệnh riêng lẻ.**

9.3. Mục Đích Sử Dụng

- Khi cần tham số hóa các đối tượng theo một hành động thực hiện.

- Tạo và thực thi các yêu cầu vào các thời điểm khác nhau.

- Khi cần hỗ trợ tính năng undo, log , callback hoặc transaction.

9.4. Code khuôn mẫu

9.5. Giải thích Design Pattern

- Theo Mô hình

+ Command: là một interface hoặc abstract class, chứa một phương thức trừu tượng thực thi (execute) một hành động (operation). Request sẽ được đóng gói dưới dạng Command.

+ ConcreteCommand: là các implementation của Command. Định nghĩa một sự gắn kết giữa một đối tượng Receiver và một hành động. Thực thi execute() bằng việc gọi operation đang hoãn trên Receiver. Mỗi một ConcreteCommand sẽ phục vụ cho một case request riêng.

+ Client: tiếp nhận request từ phía người dùng, đóng gói request thành ConcreteCommand thích hợp và thiết lập receiver của nó.

+ Invoker: tiếp nhận ConcreteCommand từ Client và gọi execute() của ConcreteCommand để thực thi request.

+ Receiver: đây là thành phần thực sự xử lý business logic cho case request. Trong phương execute() của ConcreteCommand chúng ta sẽ gọi method thích hợp trong Receiver.

- Theo code khuôn mẫu

9.6. Ví Dụ Sử Dụng Trong Thực Tế

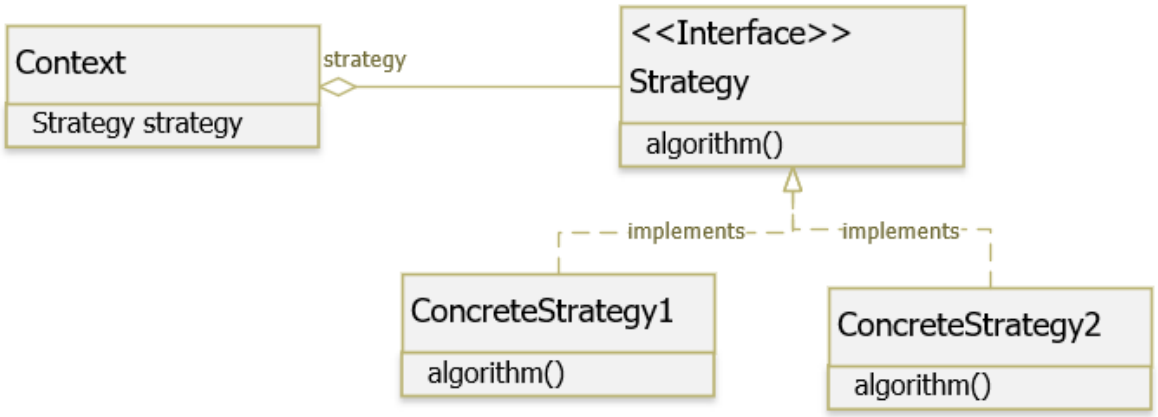
- <ý 1>

- <ý 2>

10. Strategy

10.1. Định Nghĩa Và Mô Hình Cấu Trúc

- **Strategy Pattern định nghĩa tập hợp các thuật toán, đóng gói từng thuật toán lại và dễ dàng thay đổi linh hoạt các thuật toán bên trong object. Strategy cho phép thuật toán biến đổi độc lập khi người dùng sử dụng chúng.**
- **Cụ thể và dễ hiểu hơn, Strategy Pattern tách rời phần xử lý một chức năng cụ thể ra khỏi đối tượng. Sau đó tạo ra một tập hợp các thuật toán để xử lý chức năng đó và lựa chọn thuật toán nào mà chúng ta thấy đúng đắn nhất khi thực thi chương trình.**



10.2. Đặc điểm

- **Strategy Pattern là một mô hình khởi tạo một lần, không có sự luân chuyển trạng thái.**
- **Bất kỳ loại trạng thái nào cũng có thể được khởi tạo một cách độc lập, chúng không biết được sự tồn tại của các Strategy khác.**
- **Dễ dàng mở rộng và kết hợp hành vi mới mà không thay đổi ứng dụng.**

10.3. Mục Đích Sử Dụng

- **Muốn thay đổi các thuật toán được sử dụng bên trong một đối tượng tại thời điểm run-time.**
- **Bảo trì những đoạn code dễ thay đổi**

- Tránh sự rắc rối, khi phải hiện thực một chức năng nào đó qua quá nhiều lớp con.
- Che dấu sự phức tạp, cấu trúc bên trong của thuật toán.

10.4. Code khuôn mẫu

```
1  package strategy.pattern;
2  //Strategy.java
3  public interface Strategy {
4      public String doSomething();
5  }
6
7  //ConcreteStrategyA.java
8  public class ConcreteStrategyA implements Strategy {
9      @Override
10     public String doSomething() {
11         return "ConcreteStrategyA";
12     }
13 }
14
15 //ConcreteStrategyB,C,... similar as ConcreteStrategyA
16
17 //Context.java
18 public class Context {
19     private Strategy strategy;
20
21     public Context() {
22      ...
23     }
24
25     public void setStrategy(Strategy behavior) {
26         this.strategy = behavior;
27     }
28
29     public Strategy getStrategy() {
```

```
30          return strategy;
31      }
32
33      public void executeStrategy(){
34          System.out.println(strategy.doSomething());
35      }
36  }
```

10.5. Giải thích Design Pattern

- Theo Mô hình
  - + Strategy: định nghĩa các hành vi có thể có của một Strategy.
  - + ConcreteStrategy: cài đặt các hành vi cụ thể của Strategy.
  - + Context: chứa một tham chiếu đến đối tượng Strategy và nhận các yêu cầu từ Client, các yêu cầu này sau đó được ủy quyền cho Strategy thực hiện.
- Theo code khuôn mẫu
  - + Interface Strategy chứa hàm doSomething để các ConcreteStrategy cài đặt sử dụng
  - + Context: chứa một tham chiếu đến đối tượng Strategy và hàm dựng không đối, getter và setter để chọn đối tượng trong các ConcreteStrategy. Hàm executeStrategy sẽ có nhiệm vụ sau chạy hàm doSomething ở Strategy từ object ConcreteStrategy đã được chỉ định

10.6. Ví Dụ Sử Dụng Trong Thực Tế

- Quay lại với ví dụ hệ thống thanh toán đơn giản sử dụng factory pattern https://github.com/blaplafla13th/design-patterns/tree/main/src/factory
- Strategy pattern thường được kết hợp với Factory pattern khi mà Factory pattern sẽ tạo ra đối tượng (nên thộc nhóm Creational) và đối tượng và Strategy pattern sẽ quyết định hành vi của các hàm trong đối tượng (nên thuộc nhóm Behavioral) và cũng khác đôi chút là Context Stragety sẽ làm thêm nhiệm vụ gọi hàm so với Factory

- **Giờ là ví dụ hệ thống thanh toán đơn giản sử dụng stragety pattern https://github.com/blaplafla13th/design-patterns/tree/main/src/stragety**
- **Interface của PaymentMethod có nhiều hàm và mỗi Class implement từ PaymentMethod có nội dung các hàm khác nhau. PaymentMethod sẽ đóng vai trò của Strategy và các class con MoMo, Techcombank đóng vai trò là StragetyA,B**
- **StragetyPaymentMethod lúc này đóng vai trò Context. StragetyPaymentMethod có một biến thành viên là PaymentMethod. Để khởi tạo đối tượng, theo yêu cầu của Client paymentMethod thành viên sẽ được gán một instance mới là 1 trong những class con của PaymentMethod. Các hàm còn lại là các hàm để gọi từ Client sẽ gọi các lệnh từ object PaymentMethod đã tạo, bỏ qua getter và setter, vì setter giống như hàm dựng còn getter ở đây không cần thiết**
- **Khi Client chạy sẽ tạo 1 instance của StragetyPaymentMethod cần sử dụng, StragetyPaymentMethod sẽ tạo ra đối tượng PaymentMethod cần, các hàm Client gọi của StragetyPaymentMethod sẽ được StragetyPaymentMethod gọi thông qua interface PaymentMethod để thực hiện thuật toán của Stragety tương ứng ở đây là MoMo hoặc Techcombank**

Phần V

THAM KHẢO

11. 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/**

12. Link Project bài báo cáo

Đã bao gồm cả file tex, file pdf, file java, file ảnh kèm theo:

https://github.com/blaplafla13th/design-patterns

Source code ở đây sử dụng jdk 17 và IDE: Intellij Idea