ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
o0o

DESIGN PATTERN

Báo cáo kết thúc học phần

Môn báo cáo : Software Components

Giáo viên hướng dẫn: Quản Thái Hà
Sinh viên thực hiện : Lê Quang Trường - 20001589
Phan Thế Sơn - 20001579
Lớp : K65A7 Máy tính và Khoa học thông tin CLC

Hà Nội - 2021

Mục lục

Giới thiệu

Design Patterns (mẫu thiết kế) là một kỹ thuật trong lập trình hướng đối tượng, nó khá quan trọng và có thể nói mọi lập trình viên muốn giỏi đều phải biết. Năm 1994, bốn tác giả Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides đã cho xuất bản một cuốn sách với tiêu đề Design Patterns – Elements of Reusable Object-Oriented Software, đây là khởi nguồn của khái niệm design pattern trong lập trình phần mềm. Bốn tác giả trên được biết đến rộng rãi dưới tên Gang of Four (bộ tứ). Theo quan điểm của bốn người, design pattern chủ yếu được dựa theo những quy tắc sau đây về thiết kế hướng đối tượng:

- Lập trình cho interface chứ không phải để implement interface đó.
- Ưu tiên object composition (chứa trong) hơn là inheritance (thừa kế).

Design Pattern được sử dụng thường xuyên trong các ngôn ngữ OOP. Nó cung cấp cho chúng ta các "mẫu thiết kế", giải pháp để giải quyết các vấn đề chung, thường gặp trong lập trình. Các vấn đề mà chúng ta gặp phải có thể chúng ta sẽ tự nghĩ ra cách giải quyết nhưng có thể nó chưa phải là tối ưu. Design Pattern giúp chúng ta giải quyết vấn đề một cách tối ưu nhất, cung cấp cho chúng ta các giải pháp trong lập trình OOP.

Có thể chúng ta đã gặp design patterns ở đâu đó trong các ứng dụng, cũng có thể chúng ta đã từng sử dụng những mẫu tương tự như design pattern để giải quyết những tình huống của mình như khi chúng ta giải một bài toán/vấn đề bất kỳ, nhưng chúng ta không rõ hoặc không có một khái niệm gì về nó.

Trong báo cáo này chúng em xin được làm rõ về 10 design patterns ở nhóm 1, về khái niệm cũng như mục đích sử dụng cho đến mô hình cấu trúc với các ví dụ cụ thể, cũng như ứng dụng của chúng trong lập trình thực tế.

Nội dung của báo cáo được trình bày trong 10 chương chính tương ứng với 10 design patterns thuộc nhóm 1. Trong mỗi chương trình bài về một design pattern với các phần: định nghĩa, mục đích sử dụng, mô hình cấu trúc kèm với code minh họa cùng với việc giải thích, và cuối cùng là những ứng dụng của design pattern đó trong thực tế.

Công việc của từng thành viên trong nhóm:

- **Phan Thế Sơn** trình bày về Singleton Pattern, Observer Pattern, Iterator Pattern, Decorator Pattern và Factory Pattern (chương 1-5)
- **Lê Quang Trường** trình bày về Adapter Pattern, Command Pattern, Bridge Pattern, Strategy Pattern và Builder Pattern (chương 6-10)

Chúng em xin cảm ơn thầy Quản Thái Hà vì những giờ giảng dạy đầy nhiệt tình và đam mê. Nếu như trong báo cáo còn có những thiếu sót, kính mong thầy nhiệt tình đóng góp ý kiến xây dựng để báo cáo ngày càng hoàn thiện hơn.

Chương 1

Singleton Pattern

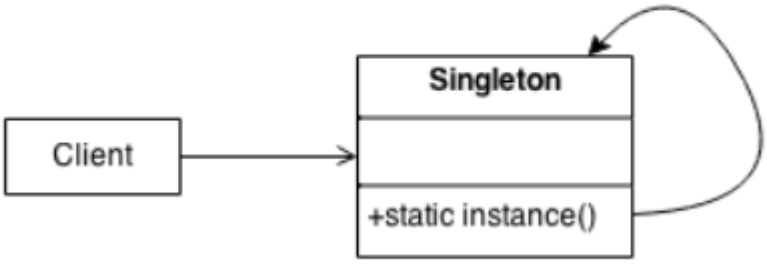
1.1 Định nghĩa

Singleton Pattern là một trong những mẫu Design Pattern đơn giản nhất trong Java.
Trong design pattern, mẫu thiết kế Singleton Pattern được dùng để đảm bảo chỉ có duy nhất một instance trong một class, và class đó sẽ cung cấp phương thức toàn cục để truy cập đến thực thể đó

1.2 Mục đích sử dụng

Khi cần một đối tượng chính xác để điều phối hành động trên toàn hệ thống. Các hệ thống vận hành hiệu quả hơn khi chỉ có một đối tượng tồn tại hoặc hạn chế sự khởi tạo cho một số lượng đối tượng nhất định. Thuật ngữ này xuất phát từ khái niệm toán học của một singleton.

1.3 Mô hình cấu trúc



Mô tả đôi nét: để đảm bảo chỉ có duy nhất một instance trong một class thì ta tạo ra một construct mang thuộc tính private.
Nhưng nó nảy sinh ra một vấn đề là private chỉ có thể được khai báo ở nội bộ của class đó.

-

Ta có thể vừa khắc phục vấn đề chỉ có một instance bằng phương thức public static(phương thức tĩnh) vì thành phần static chỉ có thể gọi một lần. Cụ thể :

- Lớp Singleton khai báo static method getInstance() trả về cùng một thể hiện của lớp riêng của nó.
- Constructor của Singleton phải là private. Gọi phương thức getInstance() là cách duy nhất để lấy đối tượng Singleton. Chúng Ta sẽ đi sâu hơn vào ví dụ để tìm hiểu:

```
3  public class ChocolateBoiler {
4
5    private boolean empty;
6    private boolean boiled;
7    private static ChocolateBoiler uniqueInstance;
8
9    private ChocolateBoiler() {
10     empty = true;
11     boiled = false;
12   }
13
14   public static ChocolateBoiler getInstance() {
15     if (uniqueInstance == null) {
16       System.out.println("Creating unique instance of Chocolate Boiler");
17       uniqueInstance = new ChocolateBoiler();
18     }
19     System.out.println("Returning instance of Chocolate Boiler");
20     return uniqueInstance;
21   }
22
```

```
23   public void fill() {
24     if (isEmpty()) {
25       empty = false;
26       boiled = false;
27       // fill the boiler with a milk/chocolate mixture
28     }
29   }
30
31   public void drain() {
32     if (!isEmpty() && isBoiled()) {
33       // drain the boiled milk and chocolate
34       empty = true;
35     }
36   }
37
38   public void boil() {
39     if (!isEmpty() && !isBoiled()) {
40       // bring the contents to a boil
41       boiled = true;
42     }
43   }
```

+) trong ví dụ về nồi hơi chocolate ở trên:
Ta có một construct với phạm vi là private (private ChocolateBoiler()) và một đối tượng singleton sẽ không cần phải khởi tạo mà chỉ cần gọi đến phương thức tĩnh getInstance(). Đặc biệt việc khởi tạo đối tượng thông qua phương thức tĩnh getInstance() chỉ được thực hiện 1 lần.
Dây là kết quả:

```java
public class ChocolateController {

  public static void main(String args[]) {
    ChocolateBoiler boiler = ChocolateBoiler.getInstance();
    boiler.fill();
    boiler.boil();
    boiler.drain();

    // will return the existing instance
    ChocolateBoiler boiler2 = ChocolateBoiler.getInstance();
  }
}
```

```
Creating unique instance of Chocolate Boiler
Returning instance of Chocolate Boiler
Returning instance of Chocolate Boiler
```

Có thể thấy chỉ có duy nhất một đối tượng được khởi tạo trong lớp dù có khai báo bao nhiêu lần đi nữa.

1.4 Singleton Pattern trong thực tế

Dưới đây là một số trường hợp sử dụng của Singleton Pattern thường gặp:

- Vì class dùng Singleton chỉ tồn tại 1 Instance (thể hiện) nên nó thường được dùng cho các trường hợp giải quyết các bài toán cần truy cập vào các ứng dụng như: Shared resource, Logger, Configuration, Caching, Thread pool, ...

- Một số design pattern khác cũng sử dụng Singleton để triển khai: Abstract Factory, Builder, Prototype, Facade,...

- Đã được sử dụng trong một số class của core java như: java.lang.Runtime, java.awt.Desktop.

Chương 2

Observer Pattern

2.1 Định nghĩa

Observer Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó định nghĩa mối phụ thuộc một – nhiều giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.

2.2 Mục đích sử dụng

Mục đích và lợi ích:

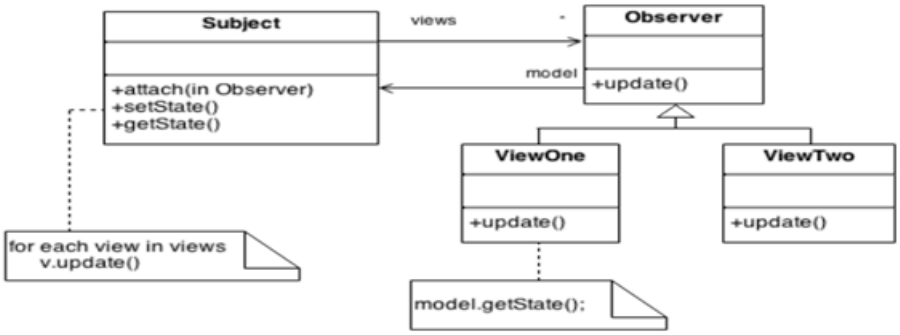
- Dễ dàng mở rộng với ít sự thay đổi : mẫu này cho phép thay đổi Subject và Observer một cách độc lập. Chúng ta có thể tái sử dụng các Subject mà không cần tái sử dụng các Observer và ngược lại. Nó cho phép thêm Observer mà không sửa đổi Subject hoặc Observer khác. Vì vậy, nó đảm bảo nguyên tắc Open/Closed Principle (OCP).

- Sự thay đổi trạng thái ở 1 đối tượng có thể được thông báo đến các đối tượng khác mà không phải giữ chúng liên kết quá chặt chẽ.

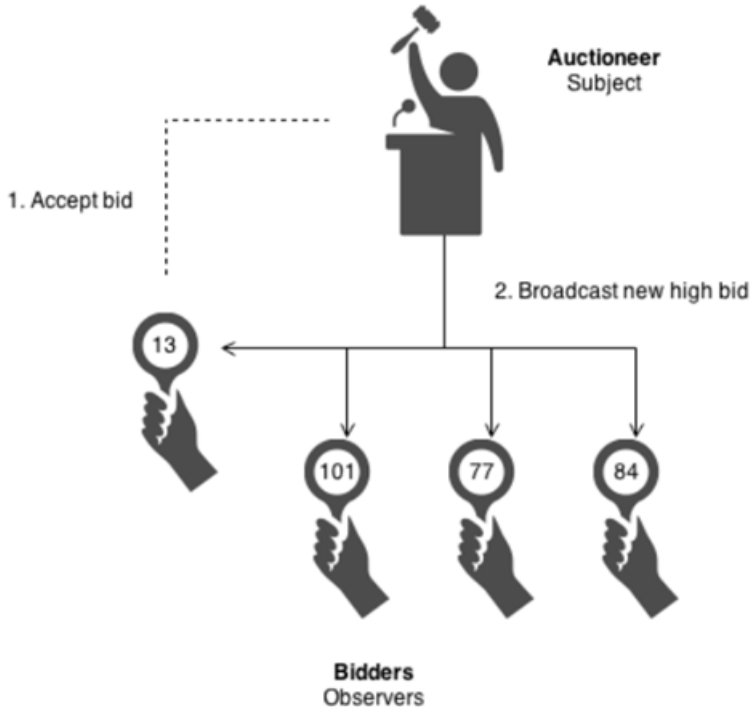
- Một đối tượng có thể thông báo đến một số lượng không giới hạn các đối tượng khác.

Bên cạnh những lợi ích, chúng ta cần xem xét đến trường hợp cập nhật không mong muốn (Unexpected update) của Subject. Bởi vì các Observer không biết về sự hiện diện của nhau, nó có thể gây tốn nhiều chi phí của việc thay đổi Subject. cấu

2.3 Mô hình cấu trúc



Subject : cung cấp các phương thức để thêm, loại bỏ, thông báo 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.
ViewOne,ViewTwo .. : có thể là bất cứ lớp con nào implement Observer interface. Mỗi observer đăng ký với một subject cụ thể để nhận được cập nhật.
Khi có một thay đổi nào đó, subject sẽ thông báo khi các observer có sự thay đổi. VD:



Trong một cuộc đấu giá, khi một người tham gia giơ bảng giá lên,đấu giá viên bắt đầu quan sát , khi được chấp thuận, bảng giá mới sẽ được phát cho tất cả các nhà thầu tham gia khác.
Để trực quan hơn chúng ta sẽ đến với phần ví dụ minh họa:

Đây là một minh họa về trạm thời tiết Weather Station:

```
public interface Subject {

  public void registerObserver(Observer o);

  public void removeObserver(Observer o);

  public void notifyObservers();
}
```

Trong subject interfact có:
+) public void registerObserver(Observer o) và public void removerObserver(Observer o) : cả 2 phương thức này lấy một Observer làm đối số, đó là Observer được remove hoặc register.
+)phương thức notifyObservers() : phương thức này được gọi để thông báo các thành viên khi subject thay đổi.

```
public interface Observer {

  public void update(float temp, float humidity, float pressure);
}
```

Giao diện observer được implement bởi tất cả các nhà quan sát, vì vậy tất cả phải thực hiện phương thức update() với đối số là các trạng thái mà observer nhận được khi subject đo thời tiết thay đổi.

```
public interface DisplayElement {

  public void display();
}
```

Đây là phần giao diện để hiển thị,chúng sẽ gọi phần tử hiển thị khi cần hiển thị.

```
import java.util.*;

public class WeatherData implements Subject {

  private List<Observer> observers;
  private float temperature;
  private float humidity;
  private float pressure;

  public WeatherData() {
    observers = new ArrayList<Observer>();
  }

  public void registerObserver(Observer o) {
    observers.add(o);
  }

  public void removeObserver(Observer o) {
    observers.remove(o);
  }
public void notifyObservers() {
  for (Observer observer : observers) {
    observer.update(temperature, humidity, pressure);
  }
}
```

```
public void setMeasurements(float temperature,
        float humidity, float pressure) {
  this.temperature = temperature;
  this.humidity = humidity;
  this.pressure = pressure;
  measurementsChanged();
}

public float getTemperature() {
  return temperature;
}

public float getHumidity() {
  return humidity;
}

public float getPressure() {
  return pressure;
}
```

Đây là phần chứa dữ liệu về thời tiết, lớp này được implement từ subject gồm:
+) có một ArrayList<Observer> (phạm vi private) dùng để lưu trữ các observer, nó được tạo ở trong construct.
+) các thuộc tính có bản như temperature (nhiệt độ), humidity(độ ẩm), pressure(áp suất) có phạm vi truy cập là private và kiểu dữ liệu float.
+) các phương thức registerObserver(Observer o) thêm các observer vào trong List khi muốn đăng ký Phương thức removeObserver(Observer o) khi một observer muốn un-register nó sẽ xóa nó ra khỏi List
+) phương thức notifyObservers() sẽ thông báo update() trạng thái đến tất cả các observer
+) phương thức measurementsChanged() sẽ gọi đến notifyObservers() nhằm thông báo đến Observer khi có sự thay đổi phép đo.
+) phương thức setMeasurements() sẽ thay đổi giá trị phép đo và gọi đến measurementsChanged() nhằm thông báo đến các observers là giá trị đã thay đổi.
+) cuối cùng là các phương thức getTemperature() , getHumidity() và getPressure().
Chúng ta sẽ đi đến phần màn hình hiển thị:

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {

  private float temperature;
  private float humidity;
  private WeatherData weatherData;

  public CurrentConditionsDisplay(WeatherData weatherData) {
    this.weatherData = weatherData;
    weatherData.registerObserver(this);
  }

  public void update(float temperature, float humidity, float pressure) {
    this.temperature = temperature;
    this.humidity = humidity;
    display();
  }

  public void display() {
    System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
  }
}
```

Dây là màn hình thống kê và dự báo hiển thị cho nhiệt độ và độ ẩm, có một construct dùng để thông qua WeatherData và nó được dùng để đăng ký observer. Phương thức update dùng để lưu nhiệt độ và áp suất đo được, từ đó đưa ra dự báo, khi có thay đổi . đặc biệt, WeatherData không được sử dụng, nhưng nó sẽ giúp cho việc hủy đăng ký với tư cách một observer.

```java
public class ForecastDisplay implements Observer, DisplayElement {

  private float currentPressure = 29.92f;
  private float lastPressure;
  private WeatherData weatherData;

  public ForecastDisplay(WeatherData weatherData) {
    this.weatherData = weatherData;
    weatherData.registerObserver(this);
  }

  public void update(float temp, float humidity, float pressure) {
    lastPressure = currentPressure;
    currentPressure = pressure;

    display();
  }

  public void display() {
    System.out.print("Forecast: ");
    if (currentPressure > lastPressure) {
      System.out.println("Improving weather on the way!");
    } else if (currentPressure == lastPressure) {
      System.out.println("More of the same");
    } else if (currentPressure < lastPressure) {
      System.out.println("Watch out for cooler, rainy weather");
    }
  }
}
```

Dây là màn hình thống kê và dự báo hiển thị cho áp suất chênh lệch.

```
public class HeatIndexDisplay implements Observer, DisplayElement {

  float heatIndex = 0.0f;
  private WeatherData weatherData;

  public HeatIndexDisplay(WeatherData weatherData) {
    this.weatherData = weatherData;
    weatherData.registerObserver(this);
  }

  public void update(float t, float rh, float pressure) {
    heatIndex = computeHeatIndex(t, rh);
    display();
  }

  private float computeHeatIndex(float t, float rh) {
    float index = (float) ((16.923 + (0.185212 * t) + (5.37941 * rh) - (0.100254 * t * rh)
            + (0.00941695 * (t * t)) + (0.00728898 * (rh * rh))
            + (0.000345372 * (t * t * rh)) - (0.000814971 * (t * rh * rh))
            + (0.0000102102 * (t * t * rh * rh)) - (0.000038646 * (t * t * t)) + (0.0000291583
            * (rh * rh * rh)) + (0.00000142721 * (t * t * t * rh))
            + (0.000000197483 * (t * rh * rh * rh)) - (0.0000000218429 * (t * t * t * rh * rh))
            + 0.000000000843296 * (t * t * rh * rh * rh))
            - (0.0000000000481975 * (t * t * t * rh * rh * rh)));
    return index;
  }

  public void display() {
    System.out.println("Heat index is " + heatIndex);
  }
}
```

Dây là màn hình thống kê và dự báo hiển thị cho chỉ số nhiệt.

```
public class HeatIndexDisplay implements Observer, DisplayElement {

  float heatIndex = 0.0f;
  private WeatherData weatherData;

  public HeatIndexDisplay(WeatherData weatherData) {
    this.weatherData = weatherData;
    weatherData.registerObserver(this);
  }

  public void update(float t, float rh, float pressure) {
    heatIndex = computeHeatIndex(t, rh);
    display();
  }

  private float computeHeatIndex(float t, float rh) {
    float index = (float) ((16.923 + (0.185212 * t) + (5.37941 * rh) - (0.100254 * t * rh)
            + (0.00941695 * (t * t)) + (0.00728898 * (rh * rh))
            + (0.000345372 * (t * t * rh)) - (0.000814971 * (t * rh * rh))
            + (0.0000102102 * (t * t * rh * rh)) - (0.000038646 * (t * t * t)) + (0.0000291583
            * (rh * rh * rh)) + (0.00000142721 * (t * t * t * rh))
            + (0.000000197483 * (t * rh * rh * rh)) - (0.0000000218429 * (t * t * t * rh * rh))
            + 0.000000000843296 * (t * t * rh * rh * rh))
            - (0.0000000000481975 * (t * t * t * rh * rh * rh)));
    return index;
  }

  public void display() {
    System.out.println("Heat index is " + heatIndex);
  }
}
```

Đây là màn hình thống kê và dự báo hiển thị cho thống kê nhiệt độ.
Toàn bộ dữ liệu được thu thập từ trạm thời tiết

```
public class WeatherStation {

  public static void main(String[] args) {
    WeatherData weatherData = new WeatherData();

    CurrentConditionsDisplay currentDisplay
            = new CurrentConditionsDisplay(weatherData);
    StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
    ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

    weatherData.setMeasurements(80, 65, 30.4f);
    weatherData.setMeasurements(82, 70, 29.2f);
    weatherData.setMeasurements(78, 90, 29.2f);

    weatherData.removeObserver(forecastDisplay);
    weatherData.setMeasurements(62, 90, 28.1f);
  }
}
```

Trạm thời tiết sẽ hiển thi thông tin
Đây là kết quả:

```
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
Current conditions: 62.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 75.5/82.0/62.0
```

Mẫu thiết kế Observer (quan sát) có thể được sử dụng bất cứ khi nào mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.
tế

2.4 Observer Pattern trong thực tế

Observer Pattern được áp dụng khi:

- Khi thay đổi một đối tượng, yêu cầu thay đổi đối tượng khác và chúng ta không biết có bao nhiêu đối tượng cần thay đổi, những đối tượng này là ai.

- 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) : trong MVC, mẫu này được sử dụng để tách Model khỏi View. View đại diện cho Observer và Model là đối tượng Observable.

Chương 3

Iterator pattern

3.1 Định nghĩa

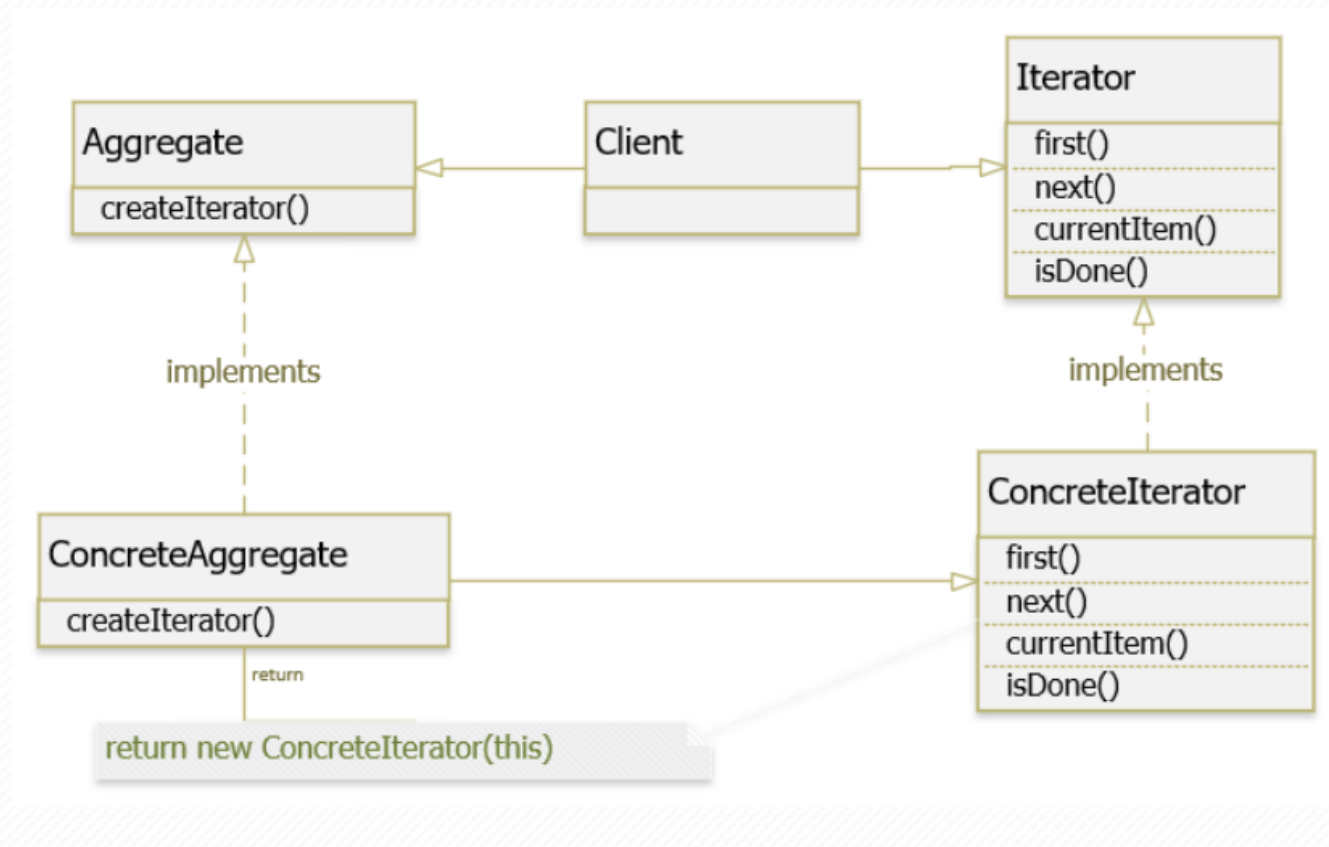
Iterator Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó được sử dụng để "Cung cấp một cách thức truy cập tuần tự tới các phần tử của một đối tượng tổng hợp, mà không cần phải tạo dựng riêng các phương pháp truy cập cho đối tượng tổng hợp này".

Nói cách khác, một Iterator được thiết kế cho phép xử lý nhiều loại tập hợp khác nhau bằng cách truy cập những 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 của những tập hợp này.

3.2 Mục đích sử dụng

Nếu trong project của các bạn có quá nhiều cấu trúc dạng danh sách như cấu trúc cây, mảng, ngăn xếp, hàng đợi.... Và bạn muốn có một quy tắc chung cho chúng như đều có thêm sửa xoá chẳng hạn. Như vậy lúc này chúng ta có thể tìm tới Iterator.

3.3 Mô hình cấu trúc



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.

VD cụ thể:
Tình huống: mô tả thực đơn cho những bữa ăn của hai cửa hàng có PancakeHouseMenu và DinnerMenu. Nhưng hai menu này được thực hiện khá là khác nhau

```java
public class MenuItem {

  String name;
  String description;
  boolean vegetarian;
  double price;

  public MenuItem(String name,
          String description,
          boolean vegetarian,
          double price) {
    this.name = name;
    this.description = description;
    this.vegetarian = vegetarian;
    this.price = price;
  }
```

```java
public String getDescription() {
  return description;
}

public double getPrice() {
  return price;
}

public boolean isVegetarian() {
  return vegetarian;
}

public String toString() {
  return (name + ", $" + price + "\n   " + description);
}
public String getName() {
  return name;
}
```

Trong class này chúng ta có những hàm cơ bản:
Hàm construct khởi tạo name,description,vegetarian,price Các thuộc tính name(String),description(String),vegetarian(boolean),price(double) các thuộc tính này đều có phạm vi là private. Các hàm getter và setter cho các trường giá trị.

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

public class PancakeHouseMenu implements Menu {

  List<MenuItem> menuItems;

  public PancakeHouseMenu() {
    menuItems = new ArrayList<MenuItem>();

    addItem("K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs and toast",
            true,
            2.99);

    addItem("Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            false,
            2.99);

    addItem("Blueberry Pancakes",
            "Pancakes made with fresh blueberries",
            true,
            3.49);

    addItem("Waffles",
            "Waffles with your choice of blueberries or strawberries",
            true,
            3.59);
  }
```

```
public void addItem(String name, String description,
        boolean vegetarian, double price) {
  MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
  menuItems.add(menuItem);
}

public List<MenuItem> getMenuItems() {
  return menuItems;
}

public Iterator createIterator() {
  return new PancakeHouseMenuIterator(menuItems);
}

public String toString() {
  return "Objectville Pancake House Menu";
}

// other menu methods here
```

Tại PancakeHouseMenu:
- Có ArrayList<> dùng để lưu trữ dữ liệu.
- Trong construct của lớp mỗi menu item được thêm vào ArrayList<>, mỗi menuItem có tên, mô tả, đồ ăn chay hay không và giá tiền.
- Phương thức addItem() dùng để thêm menu mới vào trong ArrayList<>, Phương thức getMenuItem dùng để trả về danh sách các menu.
- Phương thức createIterator() cho phép chúng ta trả về danh sách menu theo iterator pattern mà chúng ta sẽ nói rõ hơn nữa ở phía sau.

```
public class DinerMenu implements Menu {

  static final int MAX_ITEMS = 6;
  int numberOfItems = 0;
  MenuItem[] menuItems;

  public DinerMenu() {
    menuItems = new MenuItem[MAX_ITEMS];

    addItem("Vegetarian BLT",
            "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);
    addItem("BLT",
            "Bacon with lettuce & tomato on whole wheat", false, 2.99);
    addItem("Soup of the day",
            "Soup of the day, with a side of potato salad", false, 3.29);
    addItem("Hotdog",
            "A hot dog, with sauerkraut, relish, onions, topped with cheese",
            false, 3.05);
    addItem("Steamed Veggies and Brown Rice",
            "Steamed vegetables over brown rice", true, 3.99);
    addItem("Pasta",
            "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
            true, 3.89);
  }
```

```
public void addItem(String name, String description,
        boolean vegetarian, double price) {
    MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
    if (numberOfItems >= MAX_ITEMS) {
      System.err.println("Sorry, menu is full!  Can't add item to menu");
    } else {
      menuItems[numberOfItems] = menuItem;
      numberOfItems = numberOfItems + 1;
    }
  }

  public MenuItem[] getMenuItems() {
    return menuItems;
  }

  public Iterator createIterator() {
    return new DinerMenuIterator(menuItems);
    // To test Alternating menu items, comment out above line,
    // and uncomment the line below.
    //return new AlternatingDinerMenuIterator(menuItems);
  }

  // other menu methods here
```

Trong DinnerMenu có cách thức hơi khác so với PancakeHouseMenu:

- Trong class này sử dụng Array thay cho ArrayList<> nhằm mục đích quản lí số lượng các menu, và lấy menu một cách đơn giản hơn.
- Phương thức addItem() sẽ kiểm tra nếu số lượng thực đơn đã đầy, item đó sẽ không được thêm vào mảng.
- Phương thức createIterator() có tác dụng tương tự như trong class PancakeHouseMenu. Đặc biệt cách thức này chúng ta sẽ chỉ biết là nó return về menuItem mà không biết nó được triển khai như thế nào.

Những phương thức khác đều hoạt động tương tự như PancakeHoueMenu.Nhưng một vấn đề nảy ra: nếu muốn in ra menuItem của 2 menu này cho một nữ phục vụ (Waitress) sẽ rất là khó khăn vì cú pháp của array và arrayList khác nhau, và cữ mỗi menu thực hiện in một lần cũng mất thời gian hay là việc phân loại bữa trưa và bữa chiều của 2 menu và in ra thực đơn bữa trưa (chiều). vì thế chúng ta cần phải cho chúng implements menu chung để giảm thiểu các vòng lặp. Class menu sẽ giải quyết điều đó:

```
public interface Iterator {

  boolean hasNext();

  MenuItem next();
}
```

Trong interface Iterator, có 2 phương thức gói gọn là hasNext() cho chúng ta biết có còn phần tử để duyệt qua hay không và next() trả về đối tượng tiếp theo trong tập hợp

Giao diện interface Menu tạo ra phương thức createIterator().

```
public interface Menu {

  public Iterator createIterator();
}
```

```java
public class DinerMenuIterator implements Iterator {

  MenuItem[] items;
  int position = 0;

  public DinerMenuIterator(MenuItem[] items) {
    this.items = items;
  }

  public MenuItem next() {
    /*
        MenuItem menuItem = items[position];
        position = position + 1;
        return menuItem;
     */

    // or shorten to
    return items[position++];
  }

  public boolean hasNext() {
    /*
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
     */

    // or shorten to
    return items.length > position;
  }
```

Class này được implement từ interface Iterator.
Với một position dùng để duy trì vị trí lặp trên mảng . mặc định bằng 0.
Hàm construct lấy mảng mà chúng duyệt qua.
Việc cho các menu implement một interface iterator đã giúp giảm bớt độ phức tạp dành cho nữ phục vụ.

```java
public class Waitress {

  Menu pancakeHouseMenu;
  Menu dinerMenu;

  public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {
    this.pancakeHouseMenu = pancakeHouseMenu;
    this.dinerMenu = dinerMenu;
  }

  public void printMenu() {
    Iterator pancakeIterator = pancakeHouseMenu.createIterator();
    Iterator dinerIterator = dinerMenu.createIterator();

    System.out.println("MENU\n----\nBREAKFAST");
    printMenu(pancakeIterator);
    System.out.println("\nLUNCH");
    printMenu(dinerIterator);

  }

  private void printMenu(Iterator iterator) {
    while (iterator.hasNext()) {
      MenuItem menuItem = iterator.next();
      System.out.print(menuItem.getName() + ", ");
      System.out.print(menuItem.getPrice() + " -- ");
      System.out.println(menuItem.getDescription());
    }
  }

  public void printVegetarianMenu() {
    printVegetarianMenu(pancakeHouseMenu.createIterator());
    printVegetarianMenu(dinerMenu.createIterator());
  }
```

Trong hàm khởi tạo của Waitress sẽ khởi tạo ra 2 menu.
Phương thức printMenu khởi tạo 2 Iterator.
Sau đó printMenu được overload với mỗi iterator.
Phương thức private printMenu in ra danh sách menu.

```
public boolean isItemVegetarian(String name) {
  Iterator breakfastIterator = pancakeHouseMenu.createIterator();
  if (isVegetarian(name, breakfastIterator)) {
    return true;
  }
  Iterator dinnerIterator = dinerMenu.createIterator();
  if (isVegetarian(name, dinnerIterator)) {
    return true;
  }
  return false;
}

private void printVegetarianMenu(Iterator iterator) {
  while (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
    if (menuItem.isVegetarian()) {
      System.out.print(menuItem.getName());
      System.out.println("\t\t" + menuItem.getPrice());
      System.out.println("\t" + menuItem.getDescription());
    }
  }
}

private boolean isVegetarian(String name, Iterator iterator) {
  while (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
    if (menuItem.getName().equals(name)) {
      if (menuItem.isVegetarian()) {
        return true;
      }
    }
  }
  return false;
}
```

Tương tự chúng ta sẽ triển khai được các lớp khác.
Và đây là kết quả:

```java
public class MenuTestDrive {

  public static void main(String args[]) {
    Menu pancakeHouseMenu = new PancakeHouseMenu();
    Menu dinerMenu = new DinerMenu();

    Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu);

    // With iterators
    waitress.printMenu();

    printMenus();
  }

  /*
     * Without the Waitress, we need the code below...
   */
  public static void printMenus() {
    PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
    DinerMenu dinerMenu = new DinerMenu();

    List<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();
    MenuItem[] lunchItems = dinerMenu.getMenuItems();

    // print as Iterable
    printMenu(breakfastItems);
    printMenu(Arrays.asList(lunchItems));

    // print with forEach
    System.out.println("=== forEach ===");
    breakfastItems.forEach(item -> System.out.println(item));
    Arrays.asList(lunchItems).forEach(item -> System.out.println(item));
    System.out.println("=== forEach ===");
```

```
  // Using enhanced for loop
  System.out.println("USING ENHANCED FOR");
  for (MenuItem menuItem : breakfastItems) {
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
  }
  for (MenuItem menuItem : lunchItems) {
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
  }

  // Exposing implementation
  System.out.println("USING FOR LOOPS");
  for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = (MenuItem) breakfastItems.get(i);
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
  }

  for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
  }
}

public static void printMenu(Iterable<MenuItem> a) {
  for (MenuItem menuItem : a) {
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
  }
```

```
MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with sauerkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread
K&B's Pancake Breakfast         2.99
        Pancakes with scrambled eggs and toast
Regular Pancake Breakfast               2.99
        Pancakes with fried eggs, sausage
Blueberry Pancakes              3.49
        Pancakes made with fresh blueberries
Waffles         3.59
        Waffles with your choice of blueberries or strawberries
Vegetarian BLT          2.99
        (Fakin') Bacon with lettuce & tomato on whole wheat
BLT             2.99
        Bacon with lettuce & tomato on whole wheat
Soup of the day         3.29
        Soup of the day, with a side of potato salad
Hotdog          3.05
        A hot dog, with sauerkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice          3.99
```

Đây là một phần của kết quả.
Như vậy Iterator pattern giúp chúng ta in ra danh sách các phần tử một cách rất đơn giản, nó là một trong những pattern được ứng dụng khá cao trong thực tế.

3.4 Iterator Pattern trong thực tế

Iterator Pattern được áp dụng khi:

- Cần truy cập nội dung của đối tượng trong tập hợp mà không cần biết nội dung cài đặt bên trong nó.

- Hỗ trợ truy xuất nhiều loại tập hợp khác nhau.

- Cung cấp một interface duy nhất để duyệt qua các phần tử của một tập hợp.

Chương 4

Decorator Pattern

4.1 Định nghĩa

Decorator pattern là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Nó cho phép người dùng thêm chức năng mới vào đối tượng hiện tại mà không muốn ảnh hưởng đến các đối tượng khác. Kiểu thiết kế này có cấu trúc hoạt động như một lớp bao bọc (wrap) cho lớp hiện có. Mỗi khi cần thêm tính năng mới, đối tượng hiện có được wrap trong một đối tượng mới (decorator class).

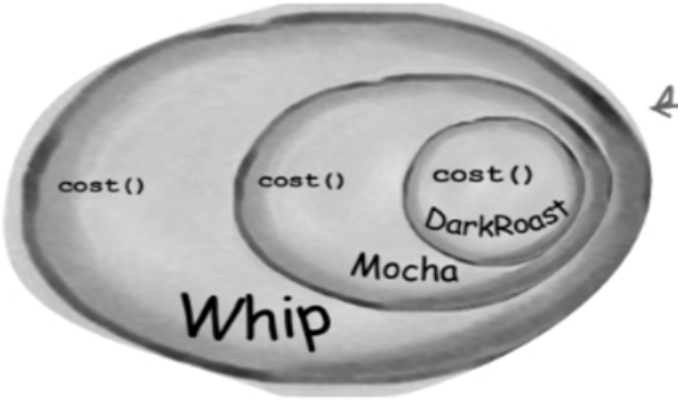
Decorator pattern sử dụng composition thay vì inheritance (thừa kế) để mở rộng đối tượng. Decorator pattern còn được gọi là Wrapper hay Smart Proxy.

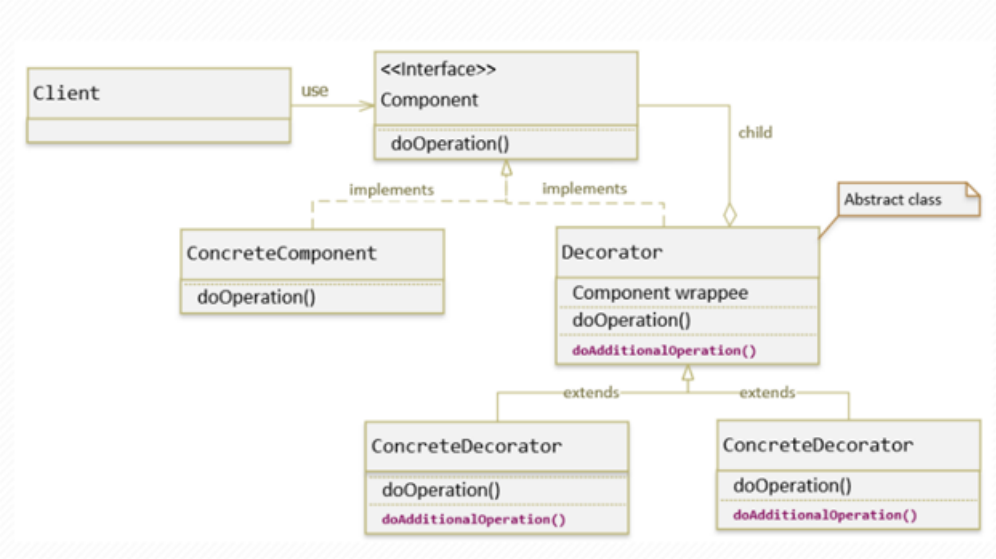
4.2 Mục đích sử dụng

Mục đích và lợi ích:

- Tăng cường khả năng mở rộng của đối tượng, bởi vì những thay đổi được thực hiện bằng cách implement trên các lớp mới.
- Client sẽ không nhận thấy sự khác biệt khi bạn đưa cho nó một wrapper thay vì đối tượng gốc.
- Một đối tượng có thể được bao bọc bởi nhiều wrapper cùng một lúc.
- Cho phép thêm hoặc xóa tính năng của một đối tượng lúc thực thi (run-time).

4.3 Mô hình cấu trúc





Các thành phần trong mẫu thiết kế Decorator:
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.
VD:

Tình huống đặt ra trong quán cafee Starbuzz, giúp quản lí các loại đồ uống.
Khởi đầu là một lớp đồ uống :

```
public abstract class Beverage {

  String description = "Unknown Beverage";

  public String getDescription() {
    return description;
  }

  public abstract double cost();
}
```

Beverage là một lớp trừu tượng với một biến String description và hai phương thức trừu tượng getDescription() lấy ra thông tin đồ uống và cost() lấy ra giá của đồ uống.

Lớp Beverage khá là đơn giản, nhưng đây là lớp cha của tất cả các lớp đồ uống – mọi lớp con đều phải extends từ đây.

```
public class Espresso extends Beverage {

  public Espresso() {
    description = "Espresso";
  }

  public double cost() {
    return 1.99;
  }
}
```

Lớp cà phê Espresso có một construct khởi tạo thông tin là tên của loại cafee này và giá tiền là 1.99. Chúng ta sẽ tìm hiểu thêm một số loại sản phẩm khác:

```
public class HouseBlend extends Beverage {

  public HouseBlend() {
    description = "House Blend Coffee";
  }

  public double cost() {
    return .89;
  }
}
```

```
public class Decaf extends Beverage {

  public Decaf() {
    description = "Decaf Coffee";
  }

  public double cost() {
    return 1.05;
  }
}
```

```
public abstract class CondimentDecorator extends Beverage {

  Beverage beverage;

  public abstract String getDescription();
}
```

Đây là lớp gia vị, nó phải kế thừa từ lớp đồ uống, ở đây có một phương thức trừu tượng getDescription() mô tả thông tin của gia vị.

Ở đây có 1 biến beverage và một lớp trừu tượng getDescription(). Nếu muốn một đồ uống nào đó có thêm gia vị , lớp đồ uống đó sẽ cần extends lại lớp này.

Như vậy có vẻ các lớp cơ sở khá là đầy đủ, chúng ta sẽ tiến hành đi sâu hơn vào các lớp con của chúng:
Lớp Espresso:

```
public class Whip extends CondimentDecorator {

  public Whip(Beverage beverage) {
    this.beverage = beverage;
  }

  public String getDescription() {
    return beverage.getDescription() + ", Whip";
  }

  public double cost() {
    return .10 + beverage.cost();
  }
}
```

```
public class Soy extends CondimentDecorator {

  public Soy(Beverage beverage) {
    this.beverage = beverage;
  }

  public String getDescription() {
    return beverage.getDescription() + ", Soy";
  }

  public double cost() {
    return .15 + beverage.cost();
  }
}
```

Điều thú vị ở đây là lớp nước uống có thêm gia vị này giá tiền sẽ được cộng thêm với giá cơ sở cho nước đó. Nó giảm độ phức tạp tính toán của nhà hàng.
Và đây là kết quả:

```java
public class StarbuzzCoffee {

  public static void main(String args[]) {
    Beverage beverage = new Espresso();
    System.out.println(beverage.getDescription()
            + " $" + beverage.cost());

    Beverage beverage2 = new DarkRoast();
    beverage2 = new Mocha(beverage2);
    beverage2 = new Mocha(beverage2);
    beverage2 = new Whip(beverage2);
    System.out.println(beverage2.getDescription()
            + " $" + beverage2.cost());

    Beverage beverage3 = new HouseBlend();
    beverage3 = new Soy(beverage3);
    beverage3 = new Mocha(beverage3);
    beverage3 = new Whip(beverage3);
    System.out.println(beverage3.getDescription()
            + " $" + beverage3.cost());
  }
}
```

```
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
```

tế

4.4 Decorator Pattern trong thực tế

Decorator Pattern được áp dụng khi:

- Khi muốn thêm tính năng mới cho các đối tượng mà không ảnh hưởng đến các đối tượng này.
- Khi không thể mở rộng một đối tượng bằng cách thừa kế (inheritance). Chẳng hạn, một class sử dụng từ khóa final, muốn mở rộng class này chỉ còn cách duy nhất là sử dụng decorator.
- Trong một số nhiều trường hợp mà việc sử dụng kế thừa sẽ mất nhiều công sức trong việc viết code. Ví dụ trên là một trong những trường hợp như vậy.

Chương 5

Factory Pattern

5.1 Factory Pattern – Factory method

5.1.1 Định nghĩa

Factory Method Design Pattern hay gọi ngắn gọn là Factory Pattern là một trong những Pattern thuộc nhóm Creational Design Pattern. Factory Pattern xác định một interface để tạo một đối tượng, nhưng cho phép các lớp con quyết định lớp nào sẽ khởi tạo. Factory Pattern giao việc khởi tạo một đối tượng cụ thể cho lớp con.

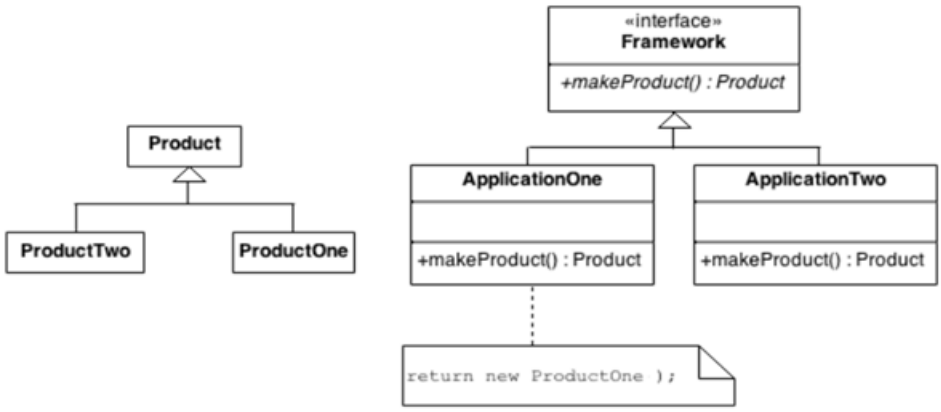
Factory Pattern đúng nghĩa là một nhà máy, và nhà máy này sẽ "sản xuất" các đối tượng theo yêu cầu của chúng ta.

5.1.2 Mục đích sử dụng

Mục đích và lợi ích:

- Factory Pattern giúp giảm sự phụ thuộc giữa các module (loose coupling): cung cấp 1 hướng tiếp cận với Interface thay thì các implement. Giúp chương trình độc lập với những lớp cụ thể mà chúng ta cần tạo 1 đối tượng, code ở phía client không bị ảnh hưởng khi thay đổi logic ở factory hay sub class.
- Mở rộng code dễ dàng hơn: khi cần mở rộng, chỉ việc tạo ra sub class và implement thêm vào factory method.
- Khởi tạo các Objects mà che giấu đi xử lí logic của việc khởi tạo đấy. Người dùng không biết logic thực sực được khởi tạo bên dưới phương thức factory.
- Dễ dạng quản lý life cycle của các Object được tạo bởi Factory Pattern.
- Thống nhất về naming convention: giúp cho các developer có thể hiểu về cấu trúc source code.

5.1.3 Mô hình câu trúc



mô tả mô hình : Product là obj cần phải khởi tạo, Framework là factory và Application là nơi quyết định xem đối tượng nào khởi tạo.
VD:
Giả sử trong một của hàng pizza, khách hàng muốn order 1 loại pizza, chúng có nhiểu loại, chúng ta xé quản lí những loại pizza đó một cách hợp lí. Hãy quan sát đoạn code để làm rõ vấn đề hơn, nhưng có một vấn đề là mỗi một vùng miền lại có một kiểu Pizza khác nhau,thương mại khác nhau. Cùng quan sát vấn đề:

Trong facory method những vấn đề liên quan đến tạo đối tượng sẽ được tạo một lớp riêng, cụ thể là một lớp trừu tượng.

```
public abstract class PizzaStore {

  abstract Pizza createPizza(String item);

  public Pizza orderPizza(String type) {
    Pizza pizza = createPizza(type);
    System.out.println("--- Making a " + pizza.getName() + " ---");
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }
}
```

Đây là class PizzaStore :

Phương thức abstract createPizza() với phạm vi là protected dùng để xử lí việc tạo đối tượng và gói nó trong một lớp con. Điều này tách client code ra khỏi code tạo object dưới subclass.

Trong phương thức public orderPizza() sẽ có một biến Pizza để khởi tạo chiếc bánh mà khách hàng muốn đặt.

Các phương thức prepare(),bake(),cut(),box() thuộc về lớp pizza, nó là các thay đổi trên chiếc bánh.

Mỗi cửa hàng pizza ở mỗi vùng miền cũng khác nhau. Chúng sẽ được kế thừa từ lớp cơ sở PizzaStore.

```
public class NYPizzaStore extends PizzaStore {

  Pizza createPizza(String item) {
    if (item.equals("cheese")) {
      return new NYStyleCheesePizza();
    } else if (item.equals("veggie")) {
      return new NYStyleVeggiePizza();
    } else if (item.equals("clam")) {
      return new NYStyleClamPizza();
    } else if (item.equals("pepperoni")) {
      return new NYStylePepperoniPizza();
    } else {
      return null;
    }
  }
}
```

Đây là một cửa hàng pizza NY

```
public class ChicagoPizzaStore extends PizzaStore {

  Pizza createPizza(String item) {
    if (item.equals("cheese")) {
      return new ChicagoStyleCheesePizza();
    } else if (item.equals("veggie")) {
      return new ChicagoStyleVeggiePizza();
    } else if (item.equals("clam")) {
      return new ChicagoStyleClamPizza();
    } else if (item.equals("pepperoni")) {
      return new ChicagoStylePepperoniPizza();
    } else {
      return null;
    }
  }
}
```

```java
import java.util.ArrayList;

public abstract class Pizza {

  String name;
  String dough;
  String sauce;
  ArrayList<String> toppings = new ArrayList<String>();

  void prepare() {
    System.out.println("Prepare " + name);
    System.out.println("Tossing dough...");
    System.out.println("Adding sauce...");
    System.out.println("Adding toppings: ");
    for (String topping : toppings) {
      System.out.println("    " + topping);
    }
  }

  void bake() {
    System.out.println("Bake for 25 minutes at 350");
  }

  void cut() {
    System.out.println("Cut the pizza into diagonal slices");
  }

  void box() {
    System.out.println("Place pizza in official PizzaStore box");
  }

  public String getName() {
    return name;
  }

  public String toString() {
    StringBuffer display = new StringBuffer();
    display.append("---- " + name + " ----\n");
    display.append(dough + "\n");
    display.append(sauce + "\n");
    for (String topping : toppings) {
      display.append(topping + "\n");
    }
    return display.toString();
  }
}
```

Lớp pizza này gồm các topping đi kèm của chiếc bánh.
Mỗi pizza có một loại tên(name), một loại bootk(dough), một loại nước sauce, và những loại topping đi kèm riêng biệt.

Ở đây chúng ta sử dụng StringBuffer để in ra thông tin của chiếc bánh đó. mỗi pizza của mỗi vùng miền cụ thể có thể có cách làm khác nhau, chúng sẽ được kế thừa từ lớp Pizza.

```java
public class NYStyleCheesePizza extends Pizza {

  public NYStyleCheesePizza() {
    name = "NY Style Sauce and Cheese Pizza";
    dough = "Thin Crust Dough";
    sauce = "Marinara Sauce";

    toppings.add("Grated Reggiano Cheese");
  }
}
```

```java
public class ChicagoStyleCheesePizza extends Pizza {

  public ChicagoStyleCheesePizza() {
    name = "Chicago Style Deep Dish Cheese Pizza";
    dough = "Extra Thick Crust Dough";
    sauce = "Plum Tomato Sauce";

    toppings.add("Shredded Mozzarella Cheese");
  }

  void cut() {
    System.out.println("Cutting the pizza into square slices");
  }
}
```

Nhận xét: ở đây chúng ta có thể thấy đoạn code này phân thành 2 class với nhiệm vụ rõ ràng, đó là Product Classes có nhiệm vụ phát triển chiếc bánh và Creator Classes với nhiệm vụ createPizza() quyết định xem chiếc bánh pizza được phát triển như thế nào .
Đây là kết quả:

```java
public class PizzaTestDrive {

  public static void main(String[] args) {
    PizzaStore nyStore = new NYPizzaStore();
    PizzaStore chicagoStore = new ChicagoPizzaStore();

    Pizza pizza = nyStore.orderPizza("cheese");
    System.out.println("Ethan ordered a " + pizza.getName() + "\n");

    pizza = chicagoStore.orderPizza("cheese");
    System.out.println("Joel ordered a " + pizza.getName() + "\n");

    pizza = nyStore.orderPizza("clam");
    System.out.println("Ethan ordered a " + pizza.getName() + "\n");

    pizza = chicagoStore.orderPizza("clam");
    System.out.println("Joel ordered a " + pizza.getName() + "\n");

    pizza = nyStore.orderPizza("pepperoni");
    System.out.println("Ethan ordered a " + pizza.getName() + "\n");

    pizza = chicagoStore.orderPizza("pepperoni");
    System.out.println("Joel ordered a " + pizza.getName() + "\n");

    pizza = nyStore.orderPizza("veggie");
    System.out.println("Ethan ordered a " + pizza.getName() + "\n");
```

```
--- Making a NY Style Sauce and Cheese Pizza ---
Prepare NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Grated Reggiano Cheese
Bake for 25 minutes at 350
Cut the pizza into diagonal slices
Place pizza in official PizzaStore box
Ethan ordered a NY Style Sauce and Cheese Pizza

--- Making a Chicago Style Deep Dish Cheese Pizza ---
Prepare Chicago Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Shredded Mozzarella Cheese
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
Joel ordered a Chicago Style Deep Dish Cheese Pizza
```

5.1.4 Factory Pattern trong thực tế

Factory Pattern được sử dụng khi:

- Chúng ta có một super class với nhiều class con và dựa trên đầu vào, chúng ta cần trả về một class con. Mô hình này giúp chúng ta đưa trách nhiệm của việc khởi tạo một lớp từ phía người dùng (client) sang lớp Factory.
- Chúng ta không biết sau này sẽ cần đến những lớp con nào nữa. Khi cần mở rộng, hãy tạo ra sub class và implement thêm vào factory method cho việc khởi tạo sub class này.
  Bạn có thể thấy Factory Pattern được áp dụng trong:
- JDK: java.util.Calendar, ResourceBundle, NumberFormat, ...
- BeanFactory trong Spring Framework.
- SessionFactory trong Hibernate Framework.

Bạn có thể thấy Factory Pattern được áp dụng trong:

- JDK: java.util.Calendar, ResourceBundle, NumberFormat, ...
- BeanFactory trong Spring Framework.
- SessionFactory trong Hibernate Framework.

5.2 Factory Pattern – Abstract Factory method

5.2.1 Định nghĩa

Abstract Factory pattern là một trong những Creational pattern. Nó là phương pháp tạo ra một Super-factory dùng để tạo ra các Factory khác. Hay còn được gọi là Factory của các Factory. Abstract Factory Pattern là một Pattern cấp cao hơn so với Factory Method Pattern.

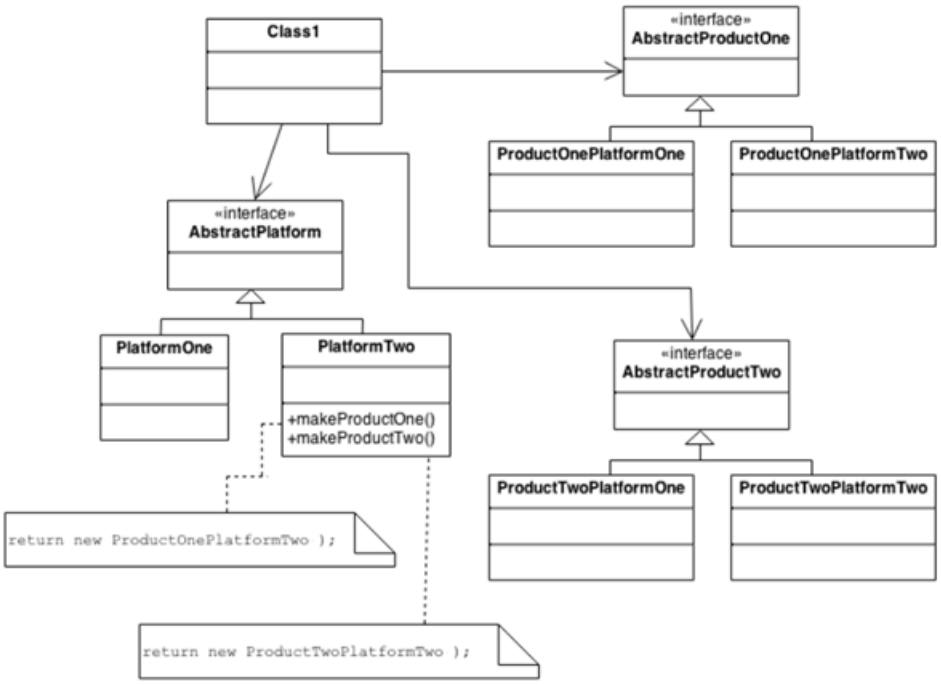
5.2.2 Mục đích sử dụng

Mục đích và lợi ích:

- Các lợi ích của Factory Pattern cũng tương tự như Factory Method Pattern như: cung cấp hướng tiếp cận với Interface thay thì các implement, che giấu sự phức tạp của việc khởi tạo các đối tượng với người dùng (client), độc lập giữa việc khởi tạo đối tượng và hệ thống sử dụng, ...
- Giúp tránh được việc sử dụng điều kiện logic bên trong Factory Pattern. Khi một Factory Method lớn (có quá nhiều sử lý if-else hay switch-case), chúng ta nên sử dụng theo mô hình Abstract Factory để dễ quản lý hơn (cách phân chia có thể là gom nhóm các sub-class cùng loại vào một Factory).

- Abstract Factory Pattern là factory của các factory, có thể dễ dạng mở rộng để chứa thêm các factory và các sub-class khác.
- Dễ dàng xây dựng một hệ thống đóng gói (encapsulate): sử dụng được với nhiều nhóm đối tượng (factory) và tạo nhiều product khác nhau.

5.2.3 Mô hình câu trúc



AbstractFactory: Khai báo dạng interface hoặc abstract class chứa các phương thức để tạo ra các đối tượng abstract.
ConcreteFactory: Xây dựng, cài đặt các phương thức tạo các đối tượng cụ thể.
AbstractProduct: Khai báo dạng interface hoặc abstract class để định nghĩa đối tượng abstract.
Product: Cài đặt của các đối tượng cụ thể, cài đặt các phương thức được quy định tại Abstract-Product.
Client: là đối tượng sử dụng AbstractFactory và các AbstractProduct.
VD:

Như ở phần Factory method đã có ví dụ về chuỗi cửa hàng pizza, giờ đây chúng ta sẽ phát triển chúng:

Do mỗi loại bánh sẽ có nguyên liệu khác nhau vd: Chicago sẽ là một loại nguyên liệu khác so với New York.

Do đó chúng ta sẽ cần tạo ra một bộ nguyên liệu :

```
public interface PizzaIngredientFactory {

  public Dough createDough();

  public Sauce createSauce();

  public Cheese createCheese();

  public Veggies[] createVeggies();

  public Pepperoni createPepperoni();

  public Clams createClam();

}
```

Do tính vùng miền, lớp con sẽ được implement từ PizzaIngredientFactory.
Implement một tập hợp các lớp nguyên liệu sẽ được sử dụng như ReggianoCheese, RedPeppers và ThickCrustDough. Các lớp này có thể được chia sẻ giữa các khu vực khi thích hợp.
Sau đó, chúng ta cần kết nối tất cả những điều này bằng cách đưa các nhà máy sản xuất nguyên liệu mới vào PizzaStore.

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

  public Dough createDough() {
    return new ThinCrustDough();
  }

  public Sauce createSauce() {
    return new MarinaraSauce();
  }

  public Cheese createCheese() {
    return new ReggianoCheese();
  }

  public Veggies[] createVeggies() {
    Veggies veggies[] = {new Garlic(), new Onion(),
            new Mushroom(), new RedPepper()};
    return veggies;
  }

  public Pepperoni createPepperoni() {
    return new SlicedPepperoni();
  }

  public Clams createClam() {
    return new FreshClams();
  }
}
```

Phương thức createDough() trả về kiểu bột để làm bánh.
Tuong tự chúng ta có createSauce(), createCheese(), createVeggies() , createPepperoni() ..v..v..
Phương thức createVeggies() trả về một mảng các loại rau củ.
Khi đó sẽ có thay đổi trên lớp Pizza:

```java
public abstract class Pizza {

  String name;

  Dough dough;
  Sauce sauce;
  Veggies veggies[];
  Cheese cheese;
  Pepperoni pepperoni;
  Clams clam;

  abstract void prepare();

  void bake() {
    System.out.println("Bake for 25 minutes at 350");
  }

  void cut() {
    System.out.println("Cutting the pizza into diagonal slices");
  }

  void box() {
    System.out.println("Place pizza in official PizzaStore box");
  }

  void setName(String name) {
    this.name = name;
  }
```

Bây giờ chiếc bánh đã linh động hơn về việc chuẩn bị gia vị.

```
public String toString() {
  StringBuffer result = new StringBuffer();
  result.append("---- " + name + " ----\n");
  if (dough != null) {
    result.append(dough);
    result.append("\n");
  }
  if (sauce != null) {
    result.append(sauce);
    result.append("\n");
  }
  if (cheese != null) {
    result.append(cheese);
    result.append("\n");
  }
  if (veggies != null) {
    for (int i = 0; i < veggies.length; i++) {
      result.append(veggies[i]);
      if (i < veggies.length - 1) {
        result.append(", ");
      }
    }
    result.append("\n");
  }
  if (clam != null) {
    result.append(clam);
    result.append("\n");
  }
  if (pepperoni != null) {
    result.append(pepperoni);
    result.append("\n");
  }
  return result.toString();
```

Lớp con của Pizza lúc này cũng đã có sự thay đổi:

```
public class CheesePizza extends Pizza {

  PizzaIngredientFactory ingredientFactory;

  public CheesePizza(PizzaIngredientFactory ingredientFactory) {
    this.ingredientFactory = ingredientFactory;
  }

  void prepare() {
    System.out.println("Preparing " + name);
    dough = ingredientFactory.createDough();
    sauce = ingredientFactory.createSauce();
    cheese = ingredientFactory.createCheese();
  }
}
```

Từ những thay đổi trên chúng ta cùng khám phá cửa hàng pizza mới sau khi sửa đổi:

```
public class NYPizzaStore extends PizzaStore {

  protected Pizza createPizza(String item) {
    Pizza pizza = null;
    PizzaIngredientFactory ingredientFactory
            = new NYPizzaIngredientFactory();

    if (item.equals("cheese")) {

      pizza = new CheesePizza(ingredientFactory);
      pizza.setName("New York Style Cheese Pizza");

    } else if (item.equals("veggie")) {

      pizza = new VeggiePizza(ingredientFactory);
      pizza.setName("New York Style Veggie Pizza");

    } else if (item.equals("clam")) {

      pizza = new ClamPizza(ingredientFactory);
      pizza.setName("New York Style Clam Pizza");

    } else if (item.equals("pepperoni")) {

      pizza = new PepperoniPizza(ingredientFactory);
      pizza.setName("New York Style Pepperoni Pizza");

    }
    return pizza;
  }
```

Và đây là kết quả :

```
public class PizzaTestDrive {

  public static void main(String[] args) {
    PizzaStore nyStore = new NYPizzaStore();
    PizzaStore chicagoStore = new ChicagoPizzaStore();

    Pizza pizza = nyStore.orderPizza("cheese");
    System.out.println("Ethan ordered a " + pizza + "\n");

    pizza = chicagoStore.orderPizza("cheese");
    System.out.println("Joel ordered a " + pizza + "\n");

    pizza = nyStore.orderPizza("clam");
    System.out.println("Ethan ordered a " + pizza + "\n");

    pizza = chicagoStore.orderPizza("clam");
    System.out.println("Joel ordered a " + pizza + "\n");

    pizza = nyStore.orderPizza("pepperoni");
    System.out.println("Ethan ordered a " + pizza + "\n");

    pizza = chicagoStore.orderPizza("pepperoni");
    System.out.println("Joel ordered a " + pizza + "\n");

    pizza = nyStore.orderPizza("veggie");
    System.out.println("Ethan ordered a " + pizza + "\n");

    pizza = chicagoStore.orderPizza("veggie");
    System.out.println("Joel ordered a " + pizza + "\n");
  }
```

```
--- Making a New York Style Cheese Pizza ---
Preparing New York Style Cheese Pizza
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box
Ethan ordered a ---- New York Style Cheese Pizza ----
Thin Crust Dough
Marinara Sauce
Reggiano Cheese


--- Making a Chicago Style Cheese Pizza ---
Preparing Chicago Style Cheese Pizza
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box
Joel ordered a ---- Chicago Style Cheese Pizza ----
ThickCrust style extra thick crust dough
Tomato sauce with plum tomatoes
Shredded Mozzarella
```

Chúng ta cũng cần cung cấp cho cửa hàng một tham chiếu (reference) đến các nhà máy sản xuất nguyên liệu của địa phương của họ (Chicago sẽ có ChicagoPizzaIngredientFactory, NY sẽ có NYPizzaIngredientFactory. . .)
Abstract Factory Pattern cho phép client sử dụng giao diện trừu tượng để tạo ra một bộ sản phẩm liên quan mà không cần biết (hoặc quan tâm) về các sản phẩm cụ thể được tạo ra thực sự

Chương 6

Adapter Pattern

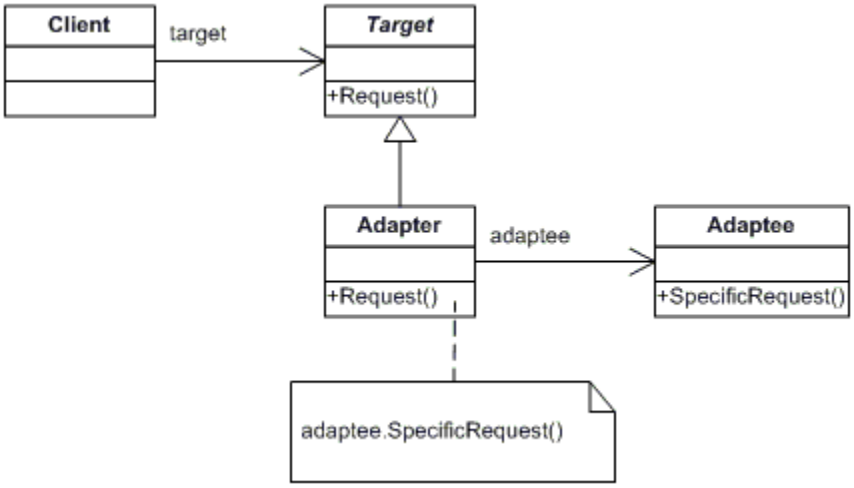
6.1 Định nghĩa

Adapter Pattern chuyển đổi giao diện của một lớp thành giao diện khác mà client mong muốn. Adapter cho phép các lớp hoạt động cùng nhau mà thông thường đó là không thể vì các giao diện không tương thích.

6.2 Mục đích sử dụng

Trong thực tế có những tình huống khi muốn sử dụng một loại dữ liệu hay thư viện mà API của nó không tương thích với API mà mình đang sử dụng. Khi đó chúng ta không muốn thay đổi hay chỉnh sửa code/hệ thống của nhà cung cấp và của mình vì những thay đổi đó có thể chỉ là nhất thời, trong tương lai có thể không sử dụng đến API của nhà cung cấp nữa. Vì thế, Adapter Pattern được sử dụng để chuyển đổi giao diện(interface) của một lớp thành giao diện khác mà client mong muốn.

6.3 Mô hình cấu trúc



Các thành phần:

- Client làm việc trực tiếp qua Target interface.
- Adapter sẽ implement Target interface đó.
- Adapter dịch yêu cầu của client thành những yêu cầu cụ thể mà adaptee hiểu.
- Adaptee (tạm gọi là class thích ứng, có nhiệm vụ thích ứng với client) là class sẽ đáp ứng yêu cầu của client nhưng hiểu theo cách mà adapter truyền lại. Những class này thường chứa những dịch vụ hữu dụng mà nhiều class khác cần dùng tới, thường là những legacy class (những class ở phiên bản trước, được thay thế thành class phiên bản cao cấp hơn), những class bên thứ ba hoặc có nhiều dependencies.

Các bước triển khai:

1. Client gửi yêu cầu ở interface.
2. Tạo một lớp adapter để triển khai client interface đó.
3. Lớp adapter giữ reference (tham chiếu) đến adaptee (cách phổ biến là truyền nó vào tham số của constructor (hàm dựng) của adapter).
4. Adapter lần lượt triển khai các methods (phương thức) của client interface, làm những công việc như chuyển đổi dữ liệu trước khi điều hướng các trách nhiệm cho lớp adaptee thực sự xử lý.
5. Client nhận được kết quả họ muốn và không biết có một adapter ở giữa gắn kết 2 bên. Ta có thể thay đổi hoặc mở rộng adapter mà không ảnh hưởng đến code của client.

Ta xét một ví dụ cụ thể sau: Nếu có một con vật nào đó đi lại như một con vịt và kêu như một con vịt, như vậy nó sẽ là một con vịt! Hoặc... nó cũng có thể là một con gà tây với một adapter của con vịt! Trước hết, ta có một mô hình mô phỏng một con vịt có hơi khác một chút (nó sẽ có hai khả năng là kêu và bay – tất nhiên sẽ có kiểu kêu và bay đặc trưng):

Duck.java

```
public interface Duck {
  public void quack();
  public void fly();
}
```

MallarDuck.java

```
public class MallardDuck implements Duck {
  public void quack() {
    System.out.println("Quack");
  }

  public void fly() {
    System.out.println("I'm flying");
  }
}
```

Có một kẻ gian xảo nào đấy muốn xâm nhập vào quá trình mô phỏng này. Nó có kiểu kêu, kiểu bay không hề giống một con vịt:

Turkey.java

```
public interface Turkey {
  public void gobble();
  public void fly();
}
```

WildTurkey.java

```
public class WildTurkey implements Turkey {
  public void gobble() {
    System.out.println("Gobble Gobble");
  }

  public void fly() {
    System.out.println("I'm flying a short distance");
  }
}
```

But the duck simulator doesn't know how to handle turkeys, only ducks!

Rõ ràng trình mô phỏng một con vịt sẽ không biết xử lý những con gà tây (Turkey) này như thế nào, chỉ có thể xử lý được những con vịt. Và giải pháp ở đây sẽ là... viết một adapter mà làm cho một con gà tây trông giống một con vịt:

TurkeyAdapter.java

```
1  public class TurkeyAdapter implements Duck {
2    private Turkey turkey;
3
4    public TurkeyAdapter(Turkey turkey) {
5      this.turkey = turkey;
6    }
7
8    public void quack() {
9      turkey.gobble();
10   }
11
12   public void fly() {
13     for (int i = 0; i < 5; i++) {
14       turkey.fly();
15     }
16   }
17 }
```

Chúng ta tiến hành theo các bước như sau:

1. Adapter Turkey implements (thừa kế) giao diện mục tiêu (Duck) để có "hình hài" của con vịt - Duck (trong mối quan hệ is-a, hay nói cách khác đó là việc adapter tương thích dữ liệu với Duck) và nó sẽ thể hiện hành vi của con gà tây (Turkey). Tức là API giống của vịt nhưng kết quả là tiếng kêu của con gà tây.

2. Adaptee (Turkey) được chuyển qua hàm dựng và được lưu trữ nội bộ.

3. Các yêu cầu bằng code của client được ủy quyền (kỹ thuật deligation) cho các phương thức tương ứng trong adaptee (ở đây là để thực hiện hành vi của con gà tây).

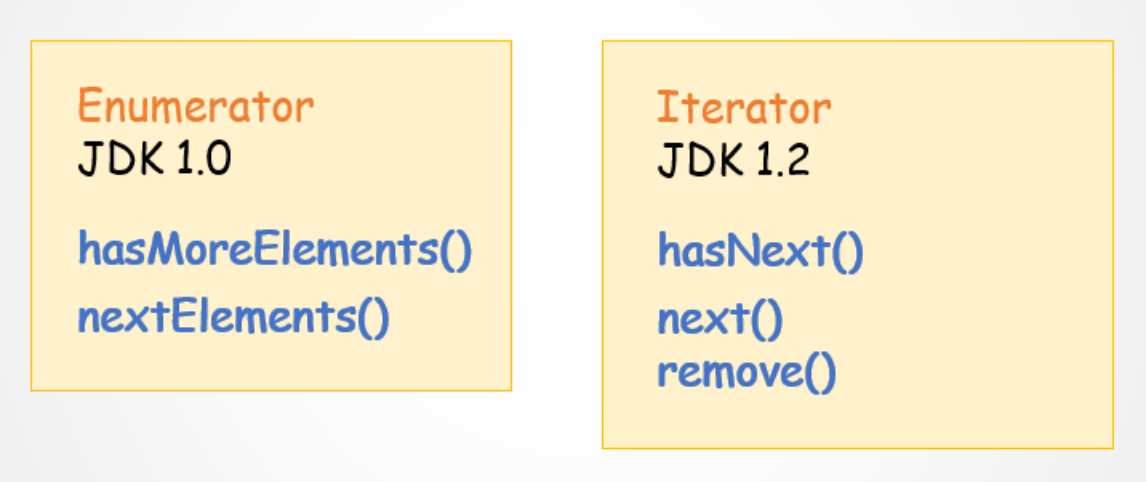
4. Adapter là một lớp chính thức, có thể chứa các biến bổ sung và các phương thức để hoàn thành công việc của mình; có thể sử dụng đa hình như một con vịt (Duck). Để sử dụng hành vi/phương thức riêng của adapter thì phải tiến hành downcast để về dữ liệu thật (Duck -> adapter). Tóm lại, adapter chỉ đơn giản là để "chuyển" từ con gà tây sang con vịt.

DuckSimulator.java

```java
import java.util.LinkedList;
import java.util.List;

public class DuckSimulator {
  public static void main(String[] args) {
    MallardDuck mallardDuck = new MallardDuck();

    WildTurkey wildTurkey = new WildTurkey();
    Duck turkeyAdapter = new TurkeyAdapter(wildTurkey);

    List<Duck> ducks = new LinkedList<Duck>();
    ducks.add(mallardDuck);
    ducks.add(turkeyAdapter);

    for (Duck duck : ducks) {
      duck.quack();
      duck.fly();
    }
  }
}
```

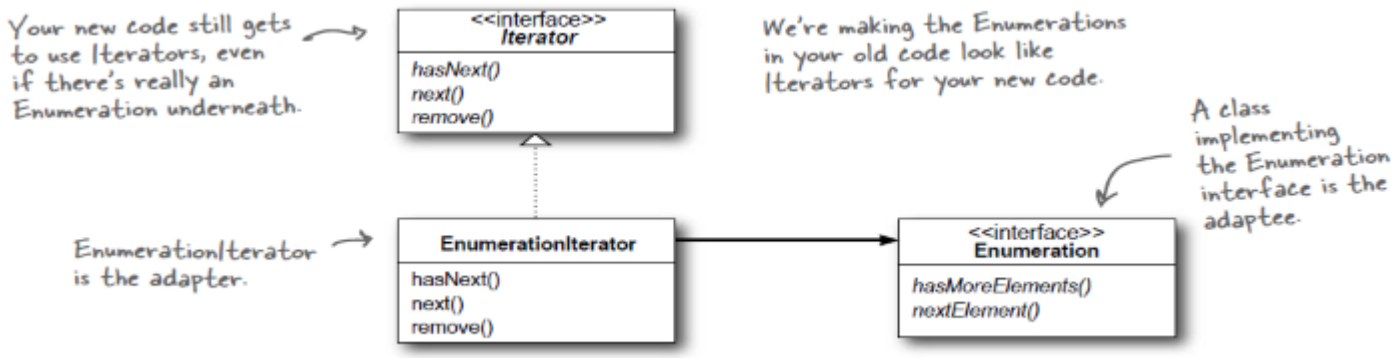
6.4 Adapter Pattern trong thực tế

Adapter được ứng dụng rộng rãi trong nhiều trường hợp, ta thường thấy Adapter xuất hiện trong những tình huống cần nâng cấp hệ thống cũ và có nhiều class cũ nhưng vẫn chứa phương thức quan trọng, làm cho hệ thống hiệu quả hơn thông qua việc làm cho các component giao tiếp với nhau dù không liên quan đến nhau. Một ví dụ thực tiễn: Sử dụng Enumerator như một Iterator trong Java:



Ở Java, Enumerator và Iterator đều là những con trỏ để duyệt và truy cập phần tử của một collection như Vector, Stack, Hashtable,... Enumerator xuất hiện ở JDK 1.0 và Iterator được ra mắt ở JDK 1.2. Enumerator cung cấp 2 phương thức hasMoreElement() và nextElement() để kiểm tra sự tồn tại và lấy phần tử tiếp theo trong collection. Tuy nhiên, nó không hỗ trợ các phương thức để thay đổi cấu trúc của collection, và Iterator xuất hiện như là phiên bản cải tiến của Enumerator, với các phương thức hasNext(), next() và remove().

Điều này sẽ dẫn đến tình huống đôi khi chúng ta vẫn gặp code sử dụng Enumerator interface, nhưng ta chỉ muốn sử dụng Iterator. Và đó sẽ là lúc những lập trình viên thiết kế Iterator phải sử dụng tới Adapter: chuyển hóa những phương thức của Iterator thành những phương thức của Enumerator tương ứng.



Adapter chuyển từ hasNext(), next() sang hasMoreElements() và nextElement() là hoàn toàn có thể, nhưng vì Enumeration vốn không hỗ trợ remove() nên adapter cũng không thể làm được, không thể triển khai một hàm remove() đủ chức năng ở trong class adapter được. Lúc này, chúng ta chỉ có thể throw exception để báo cho client.

Chương 7

Command Pattern

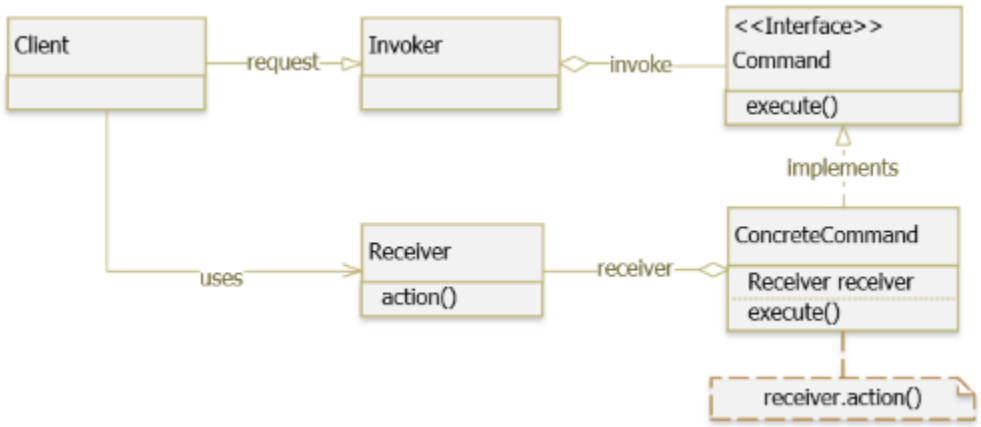
7.1 Định nghĩa

- Command Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép 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, Command Pattern cho phép tất cả những request gửi đến object được lưu trữ trong chính object đó dưới dạng một object Command. Khái niệm Command Object giống như một class trung gian được tạo ra để lưu trữ các câu lệnh và trạng thái của object tại một thời điểm nào đó.
- Command dịch ra nghĩa là ra lệnh. Commander nghĩa là chỉ huy, người này không làm mà chỉ ra lệnh cho người khác làm. Như vậy, phải có người nhận lệnh và thi hành lệnh. Người ra lệnh cần cung cấp một class đóng gói những mệnh lệnh. Người nhận mệnh lệnh cần phân biệt những interface nào để thực hiện đúng mệnh lệnh.
- Command Pattern còn được biết đến như là Action hoặc Transaction.

7.2 Mục đích sử dụng

- 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
- Hỗ trợ tính năng undo, log, callback hoặc transaction

7.3 Mô hình cấu trúc

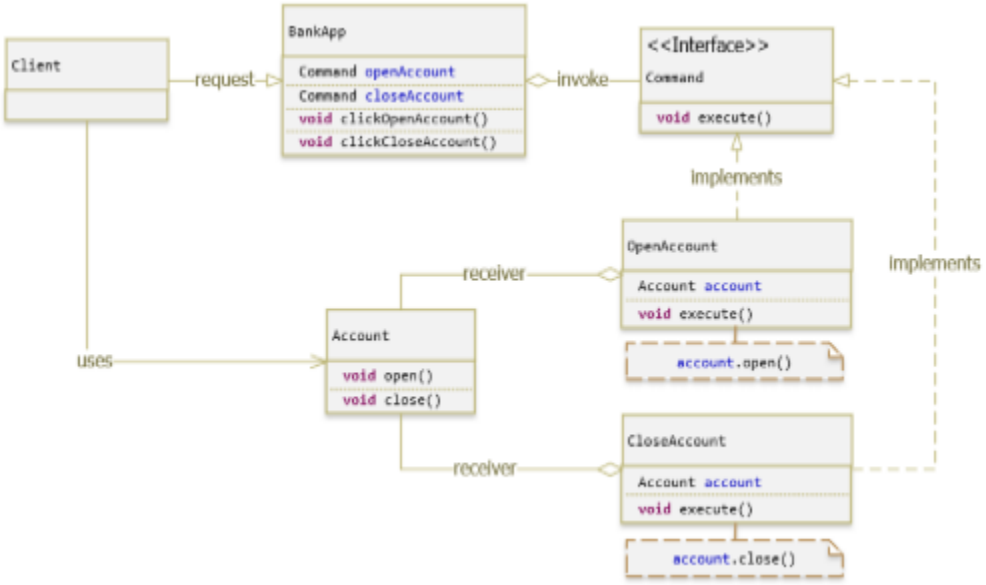


Các thành phần:

- 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.

Như vậy, Client và Invoker sẽ thực hiện việc tiếp nhận request. Còn việc thực thi request sẽ do Command, ConcreteCommand và Receiver đảm nhận.

Để có thể hiểu rõ hơn, chúng ta sẽ xét ví dụ mà Command Pattern được trong ứng dụng mở tài khoản ngân hàng: Một hệ thống ngân hàng cung cấp ứng dụng cho khách hàng (client) có thể mở (open) hoặc đóng (close) tài khoản trực tuyến. Hệ thống này được thiết kế theo dạng module, mỗi module sẽ thực hiện một nhiệm vụ riêng của nó, chẳng hạn mở tài khoản (OpenAccount), đóng tài khoản (CloseAccount). Do hệ thống không biết mỗi module sẽ làm gì, nên khi có yêu cầu client (chẳng hạn clickOpenAccount, clickCloseAccount), nó sẽ đóng gói yêu cầu này và gọi module xử lý. Chúng ta có thể hình dung thông qua hình dưới đây:



Ứng dụng của chúng ta bao gồm các lớp xử lý sau:

- Account : là một request class.
- Command : là một interface của Command Pattern, cung cấp phương thức execute().
- OpenAccount, CloseAccount : là các ConcreteCommand, cài đặt các phương thức của Command, sẽ thực hiện các xử lý thực tế.
- BankApp : là một class, hoạt động như Invoker, gọi execute() của ConcreteCommand để thực thi request.
- Client : tiếp nhận request từ phía người dùng, đóng gói request thành ConcreteCommand thích hợp và gọi thực thi các Command.

Cụ thể các lớp của chúng ta sẽ được thiết kế như sau:
Account.java:

```
public class Account {
    private String name;

    public Account(String name) {
        this.name = name;
    }

    public void open() {
        System.out.println("Account [" + name + "] Opened\n");
    }

    public void close() {
        System.out.println("Account [" + name + "] Closed\n");
    }
}
```

Command.java

```java
public interface Command {

    void execute();
}
```

OpenAccount.java

```java
public class OpenAccount implements Command {

    private Account account;

    public OpenAccount(Account account) {
        this.account = account;
    }

    @Override
    public void execute() {
        account.open();
    }
}
```

CloseAccount.java

```java
public class CloseAccount implements Command {

    private Account account;

    public CloseAccount(Account account) {
        this.account = account;
    }

    @Override
    public void execute() {
        account.close();
    }
}
```

BankApp.java

```java
public class BankApp {

    private Command openAccount;
    private Command closeAccount;

    public BankApp(Command openAccount, Command closeAccount) {
        this.openAccount = openAccount;
        this.closeAccount = closeAccount;
    }

    public void clickOpenAccount() {
        System.out.println("User click open an account");
        openAccount.execute();
    }

    public void clickCloseAccount() {
        System.out.println("User click close an account");
        closeAccount.execute();
    }
}
```

Client.java

```java
public class Client {

    public static void main(String[] args) {
        Account account = new Account("gpcoder");

        Command open = new OpenAccount(account);
        Command close = new CloseAccount(account);
        BankApp bankApp = new BankApp(open, close);

        bankApp.clickOpenAccount();
        bankApp.clickCloseAccount();
    }
}
```

Và cuối cùng là output của chương trình

```
User click open an account
Account [gpcoder] Opened

User click close an account
Account [gpcoder] Closed
```

7.4 Command Pattern trong thực tế

Command Pattern được ứng dụng khá nhiều trong thực tế, có thể kể đến vài trường hợp áp dụng Command Pattern:

- Graphical User Interface (GUI) - Giao diện đồ họa người dùng: Trong GUI và các mục menu, chúng ta sử dụng Command Pattern. Bằng cách nhấp vào một nút nào đó, chúng ta có thể đọc thông tin hiện tại của GUI và thực hiện một hành động tương ứng.
- Marco Recording (Ghi Macro): Nếu mỗi hành động của người dùng được triển khai dưới dạng một mệnh lệnh (Command) riêng biệt, chúng ta có thể ghi lại tất cả các hành động của người dùng trong Macro dưới dạng một chuỗi các mệnh lệnh. Chúng ta có thể sử dụng chuỗi này để triển khai tính năng "alPhát lại" (Playback). Bằng cách này, Macro có thể tiếp tục thực hiện cùng một nhóCommandm hành động với mỗi lần phát lại.
- Multi-step Undo (Hoàn tác nhiều bước): Khi mỗi bước được ghi lại dưới dạng một mệnh lệnh (Command), chúng ta có thể sử dụng nó để triển khai tính năng Undo (Hoàn tác) mà mỗi bước có thể hoàn tác. Nó được sử dụng trong các trình soạn thảo văn bản như MS-Word.
- Networking (Kết nối mạng): Chúng ta cũng có thể gửi một mệnh lệnh (Command) hoàn chỉnh qua mạng tới một máy từ xa nơi tất cả các hành động được gói gọn trong một mệnh lệnh (Command) được thực thi.
- Progress Bar (Thanh tiến trình): Chúng ta có thể triển khai một quy trình cài đặt dưới dạng một chuỗi các mệnh lệnh (Command). Mỗi mệnh lệnh cung cấp thời gian ước tính. Khi chúng ta thực hiện quy trình cài đặt, với mỗi mệnh lệnh chúng ta có thể hiển thị thanh tiến trình.
- Wizard: Trong quy trình wizard, chúng ta có thể triển khai các bước dưới dạng các mệnh lệnh. Mỗi bước có thể có nhiệm vụ phức tạp mà chỉ được thực hiện trong một lệnh.
- Transactions: Trong một mã hành vi transactional có nhiều tác vụ/cập nhật. Khi tất cả các nhiệm vụ được thực hiện thì chỉ có transaction được cam kết. Nếu không, chúng tôi phải khôi phục transaction (rollback the transaction). Trong một kịch bản như vậy, mỗi bước được thực hiện dưới dạng mệnh lệnh (Command) riêng biệt.

Chương 8

Bridge Pattern

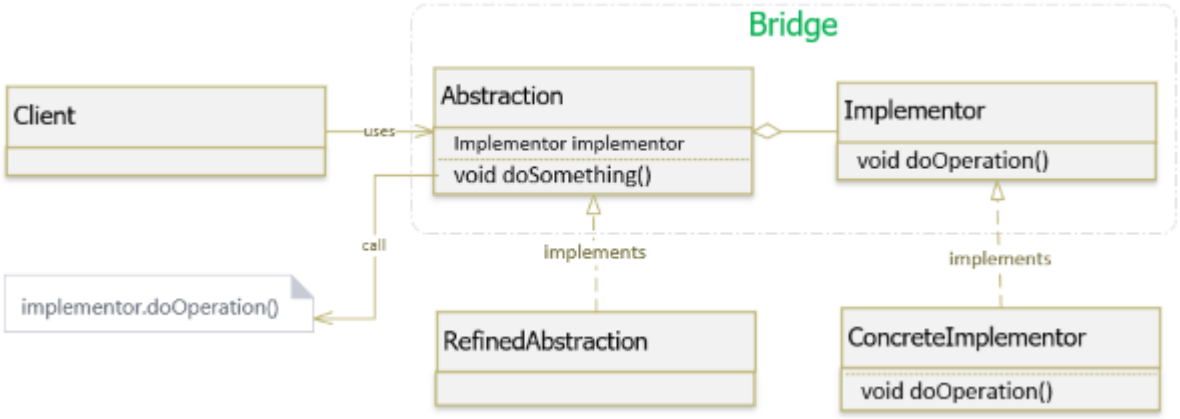
8.1 Định nghĩa

Bridge Pattern là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Ý tưởng của nó là tách phần 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. Điều đó có nghĩa là, ban đầu chúng ta thiết kế một class với rất nhiều tiến trình, bây giờ chúng ta không muốn để những tiến trình đó trong class đó nữa. Vì thế, chúng ta sẽ tạo ra một class khác và di chuyển các tiến trình đó qua class mới. Khi đó, trong lớp cũ sẽ giữ một đối tượng thuộc về lớp mới, và đối tượng này sẽ chịu trách nhiệm xử lý thay cho lớp ban đầu.

8.2 Mục đích sử dụng

- Tách ràng buộc giữa Abstraction (phần trìu tượng) và Implementation (phần thực thi) để có thể dễ dàng mở rộng độc lập nhau. Thay vì liên hệ với nhau bằng quan hệ kế thừa, hai thành phần này liên hệ với nhau thông qua quan hệ "chứa trong" (object composition).
- Sử dụng khi cả Abstraction và Implementation của chúng nên được mở rộng bằng subclass.
- Sử dụng ở những nơi mà những thay đổi được thực hiện trong implement không ảnh hưởng đến phía client.

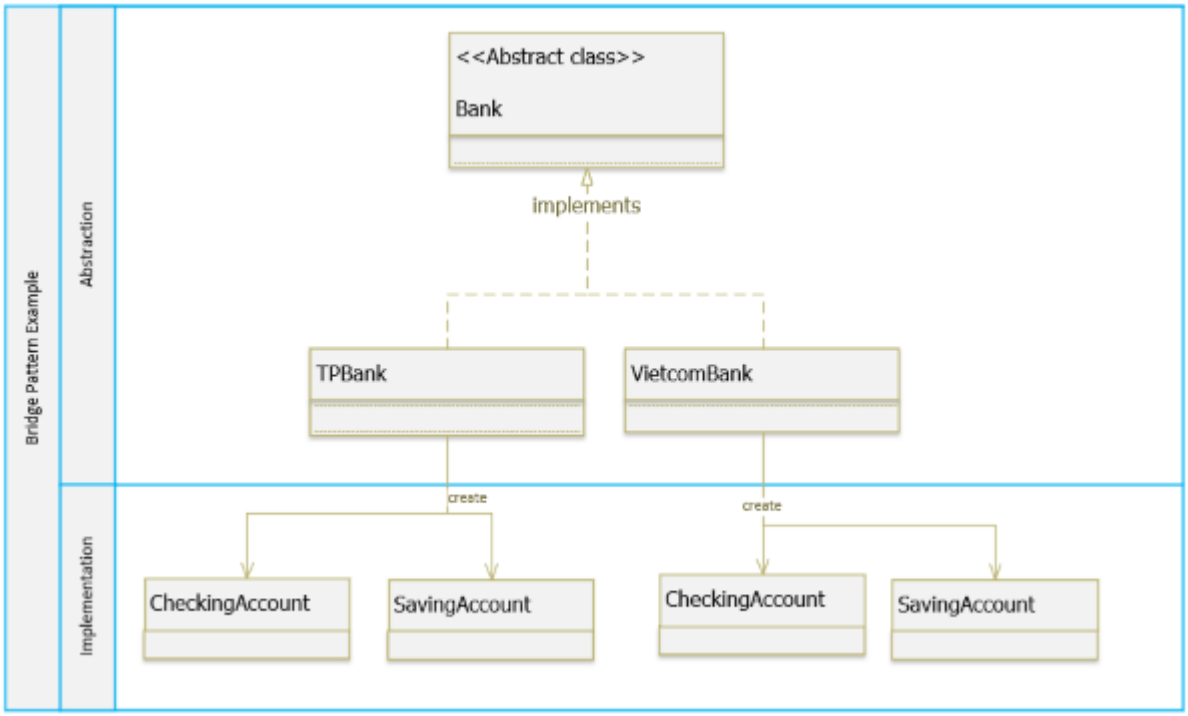
8.3 Mô hình cấu trúc



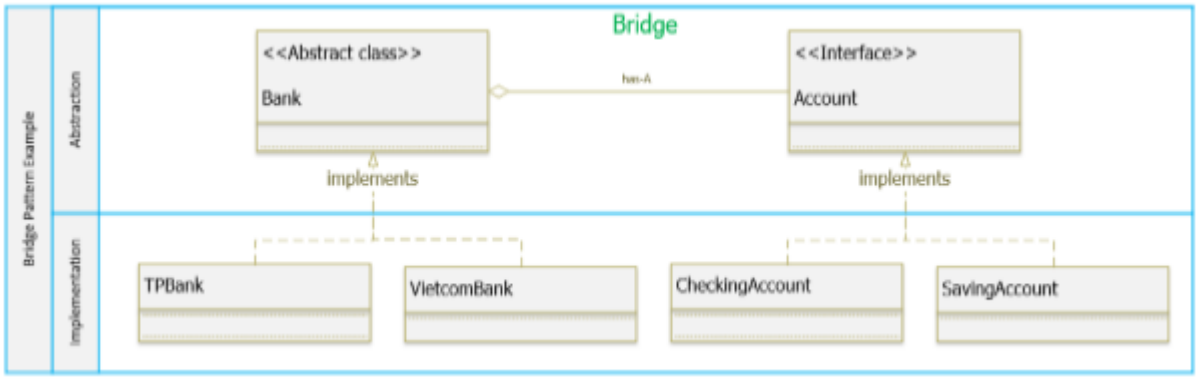
Các thành phần:

- Client: đại diện cho khách hàng sử dụng các chức năng thông qua Abstraction.
- 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.

Để có thể hiểu rõ hơn, chúng ta xét ví dụ cụ thể sau: Một hệ thống ngân hàng cung cấp các loại tài khoản khác nhau cho khách hàng, chẳng hạn: Checking account và Saving account. Chúng ta có sơ đồ như sau:



Với cách thiết kế như vậy, khi hệ thống cần cung cấp thêm một loại tài khoản khác, chúng ta phải tạo class mới cho tất cả các ngân hàng, số lượng class tăng lên rất nhiều. Bây giờ, chúng ta sẽ sử dụng Bridge Pattern để tái cấu trúc lại hệ thống trên như sau:



Với cấu trúc mới như vậy, khi có thêm một loại tài khoản mới, chúng ta đơn giản chỉ việc thêm vào một implement mới cho Account, các thành phần khác của Bank không bị ảnh hưởng. Hoặc cần thêm một ngân hàng mới, chẳng hạn VietinBank chúng ta chỉ cần thêm implement mới cho Bank, các thành phần khác cũng không bị ảnh hưởng và số lượng class chỉ tăng lên 1. Code cho chương trên như sau:

Account.java

```
public interface Account {
    void openAccount();
}
```

CheckingAccount.java

```
public class CheckingAccount implements Account {

    @Override
    public void openAccount() {
        System.out.println("Checking Account");
    }
}
```

SavingAccount.java

```
public class SavingAccount implements Account {

    @Override
    public void openAccount() {
        System.out.println("Saving Account");
    }
}
```

Bank.java

```
public abstract class Bank {

    protected Account account;

    public Bank(Account account) {
        this.account = account;
    }

    public abstract void openAccount();
}
```

VietcomBank.java

```
public class VietcomBank extends Bank {

    public VietcomBank(Account account) {
        super(account);
    }

    @Override
    public void openAccount() {
        System.out.print("Open your account at VietcomBank is a ");
        account.openAccount();
    }
}
```

TPBank.java

```
public class TPBank extends Bank {

    public TPBank(Account account) {
        super(account);
    }

    @Override
    public void openAccount() {
        System.out.print("Open your account at TPBank is a ");
        account.openAccount();
    }
}
```

Client.java

```
public class Client {

    public static void main(String[] args) {
        Bank vietcomBank = new VietcomBank(new CheckingAccount());
        vietcomBank.openAccount();

        Bank tpBank = new TPBank(new CheckingAccount());
        tpBank.openAccount();
    }
}
```

Và cuối cùng là output của chương trình:
Open your account at VietcomBank is a Checking Account
Open your account at TPBank is a Checking Account

8.4 Bridge Pattern trong thực tế

Hầu như trong mọi ứng dụng chúng ta cần lưu trữ dữ liệu. Giả sử chúng ta cần lưu trữ trong 2 cơ chế lưu trữ khác nhau là tệp và cơ sở dữ liệu. Ngoài ra, trước khi chúng ta lưu trữ, chúng ta phải xử lý đối tượng [xác thực, thiết lập một vài dữ liệu, v.v.]. Thông thường, chúng ta sử dụng các lớp riêng biệt để thực hiện các quy trình này như dưới đây:

```
SaveStudentInFile {
    checkAccess;
    validateObject;
    setAuditFields;
    openFileToWrite;
    writeDataInFile;
    closeFile;
    return studentId;
}
```

```
SaveStudentInDB {
    checkAccess;
    validateObject;
    setAuditFields;
    openDbConnection;
    storeDataInDb;
    closeConnection;
    return studentId;
}
```

Chúng ta có code để lưu trữ đối tượng Student vào tệp và cơ sở dữ liệu. Đây là một kịch bản phổ biến. Bây giờ, hãy giả sử rằng chúng ta cần thêm code để làm điều tương tự cho đối tượng khóa học (course). Sau đó, chúng ta sẽ cần hai lớp nữa như SaveCourseInFile và SaveCourseInDB.

Ngoài ra, nếu chúng ta cần thêm hệ thống lưu trữ mới như cơ sở dữ liệu hoặc yêu cầu mạng (network call) khác thì chúng ta sẽ phải tạo lớp mới cho mỗi loại. Giống như ví dụ của chúng ta là lớp Student và lớp Course. Điều này sẽ tiếp tục tăng đối với từng lớp và loại lưu trữ mới. Bridge Pattern đơn giản hóa tình huống trên bằng cách giúp chúng ta viết code có thể tái sử dụng.

Chương 9

Strategy Pattern

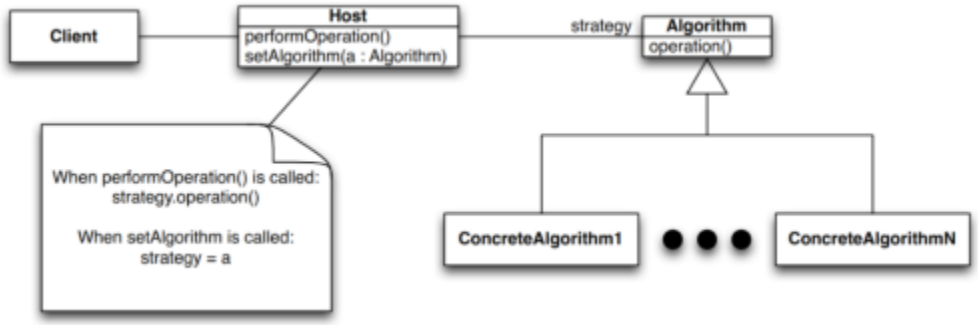
9.1 Định nghĩa

Strategy Pattern là một trong những pattern thuộc nhóm hành vi (Behavior Pattern). Strategy Pattern xác định một nhóm các thuật toán, đóng gói từng cái một và khiến cho chúng có thể hoán đổi vị trí cho nhau một cách linh hoạt bên trong object. Strategy cho phép thuật toán biến đổi độc lập khi người dùng sử dụng chúng.

9.2 Mục đích sử dụng

- Thay đổi các thuật toán được sử dụng bên trong một đối tượng tại thời điểm run-time.
- Khi có một đoạn mã dễ thay đổi, và muốn tách chúng ra khỏi chương trình chính để dễ dàng bảo trì.
- Tránh sự rắc rối khi phải hiện thực một chức năng nào đó qua quá nhiều lớp con.
- Che giấu sự phức tạp, cấu trúc bên trong của thuật toán.

9.3 Mô hình cấu trúc

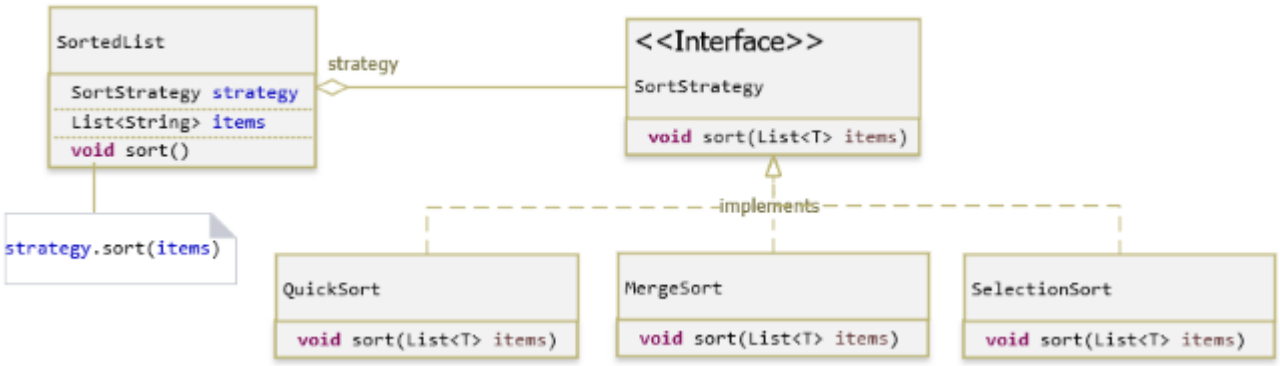


Các thành phần:

- Algorithm: định nghĩa các hành vi/phương thức có thể có của một Algorithm.
- ConcreteAlgorithm : cài đặt các hành vi/phương thức cụ thể của Algorithm.
- Host: chứa một tham chiếu đến đối tượng Algorithm và nhận các yêu cầu từ Client, các yêu cầu này sau đó được ủy quyền cho Algorithm thực hiện.

Cách tiến hành: Client (main) sử dụng Host. Algorithm được kéo ra khỏi Host. Client chỉ sử dụng giao diện công khai của Algorithm và không bị ràng buộc bởi các lớp con (concrete algorithms) cụ thể. Client có thể thay đổi hành vi của mình bằng cách chuyển đổi giữa các thuật toán cụ thể (concrete algorithms) khác nhau.

Chúng ta xét đến một ví dụ cụ thể: ứng dụng sắp xếp. Chương trình của chúng ta cung cấp nhiều giải thuật sắp xếp khác nhau: quick sort, merge sort, selection sort, heap sort, tim sort,... Tùy theo loại dữ liệu, số lượng phần tử,... mà người dùng có thể chọn một giải thuật sắp xếp phù hợp.



SortStrategy.java

```java
import java.util.List;

public interface SortStrategy {

    <T> void sort(List<T> items);
}
```

QuickSort.java

```java
import java.util.List;

public class QuickSort implements SortStrategy {

    @Override
    public <T> void sort(List<T> items) {
        System.out.println("Quick sort");
    }
}
```

MergeSort.java

```java
import java.util.List;

public class MergeSort implements SortStrategy {

    @Override
    public <T> void sort(List<T> items) {
        System.out.println("Merge sort");
    }
}
```

SelectionSort.java

```java
import java.util.List;

public class SelectionSort implements SortStrategy {

    @Override
    public <T> void sort(List<T> items) {
        System.out.println("Selection sort");
    }
}
```

SortedList.java

```
1
```

SortStrategy.java

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

public class SortedList {

    private SortStrategy strategy;
    private List<String> items = new ArrayList<>();

    public void setSortStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }

    public void add(String name) {
        items.add(name);
    }

    public void sort() {
        strategy.sort(items);
    }
}
```

StrategyPatternExample.java

```
public class StrategyPatternExample {

    public static void main(String[] args) {

        SortedList sortedList = new SortedList();
        sortedList.add("Java Core");
        sortedList.add("Java Design Pattern");
        sortedList.add("Java Library");
        sortedList.add("Java Framework");

        sortedList.setSortStrategy(new QuickSort());
        sortedList.sort();

        sortedList.setSortStrategy(new MergeSort());
        sortedList.sort();
    }
}
```

Và cuối cùng là output của chương trình:
Quick sort
Merge sort

9.4 Strategy Pattern trong thực tế

Bất kỳ phần mềm nào cần giải quyết các tác vụ đang chờ xử lý và các vấn đề có khả năng thay đổi, các tùy chọn hành vi và các thay đổi đều là ứng cử viên chính cho Strategy Pattern.

Ví dụ: các chương trình cung cấp các định dạng lưu trữ khác nhau cho các tệp hoặc các chức năng sắp xếp và tìm kiếm khác nhau có thể sử dụng các Strategy Pattern. Tương tự như vậy trong nén dữ liệu, các chương trình được sử dụng để thực hiện các thuật toán nén khác nhau dựa trên design pattern. Bằng cách này, chúng có thể chuyển đổi các video thành định dạng tệp tiết kiệm dung lượng mong muốn hoặc khôi phục các tệp lưu trữ nén (ví dụ: tệp ZIP hoặc RAR) về trạng thái ban đầu bằng cách sử dụng các chiến lược giải nén đặc biệt (special unpacking strategies). Một ví dụ khác là lưu tài liệu hoặc hình ảnh ở các định dạng tệp khác nhau.

Hơn nữa, design pattern liên quan đến việc phát triển và triển khai phần mềm trò chơi, phần mềm này phải phản ứng linh hoạt với các tình huống trò chơi thay đổi trong thời gian chạy. Các

nhân vật khác nhau, trang bị đặc biệt, hành vi của hình người hoặc các bước di chuyển khác nhau (chuyển động đặc biệt của nhân vật trò chơi) có thể được lưu trữ dưới dạng ConcreteStrategies.

Một lĩnh vực ứng dụng khác của các Strategy Pattern là phần mềm thuế. Bằng cách chuyển đổi ConcreteStrategies, mức giá có thể dễ dàng được điều chỉnh cho các nhóm chuyên nghiệp (professional groups), các quốc gia và các khu vực. Hơn nữa, các chương trình chuyển đổi dữ liệu sang các định dạng đồ họa khác nhau (ví dụ như đường, hình tròn hoặc các biểu đồ thanh) sử dụng các Strategy Pattern.

Các ứng dụng cụ thể hơn của các Strategy Pattern có thể được tìm thấy trong thư viện chuẩn Java (Java API) và trong bộ công cụ Java GUI (ví dụ: AWT, Swing và SWT), thứ mà sử dụng trình quản lý bố cục trong việc phát triển và tạo các giao diện đồ họa người dùng (GUI). Điều này có thể thực hiện các chiến lược khác nhau để cấu hình các thành phần trong phát triển giao diện. Các ứng dụng khác của các mẫu thiết kế chiến lược bao gồm các hệ thống cơ sở dữ liệu, các trình điều khiển thiết bị và các chương trình máy chủ.

Chương 10

Builder Pattern

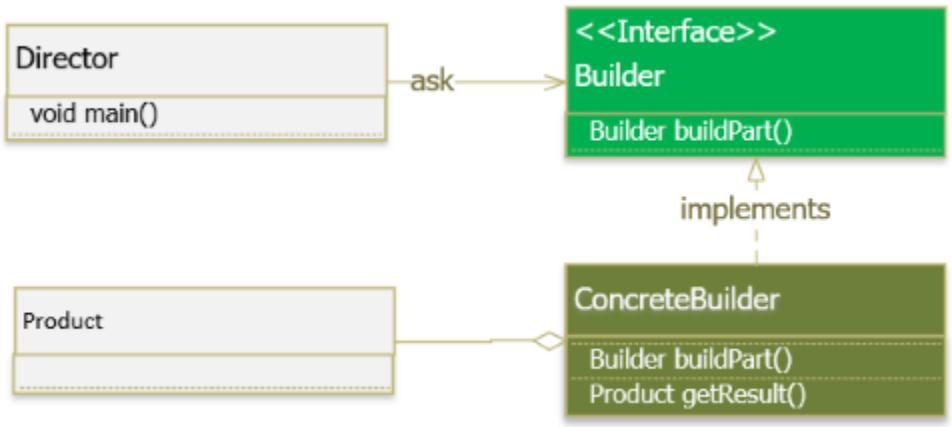
10.1 Định nghĩa

Builder Pattern là một trong những Creational pattern. Builder Pattern là mẫu thiết kế đối tượng được tạo ra để xây dựng một đối tượng phức tạp bằng cách sử dụng các đối tượng đơn giản và sử dụng tiếp cận từng bước, việc xây dựng các đối tượng độc lập với các đối tượng khác. Builder Pattern được xây dựng để khắc phục một số nhược điểm của Factory Pattern và Abstract Factory Pattern khi mà Object có nhiều thuộc tính (các nhược điểm đó là: quá nhiều tham số phải truyền vào từ phía client tới Factory Class; một số tham số có thể là tùy chọn nhưng trong Factory Pattern, chúng ta phải gửi tất cả tham số, với tham số tùy chọn nếu không nhập gì thì sẽ truyền là null; nếu một Object có quá nhiều thuộc tính thì việc tạo sẽ phức tạp).

10.2 Mục đích sử dụng

- Tạo một đối tượng phức tạp: có nhiều thuộc tính (nhiều hơn 4) và một số bắt buộc (requried), một số không bắt buộc (optional).
- Tách rời quá trình xây dựng một đối tượng phức tạp từ các phần tạo nên đối tượng.
- Kiểm soát quá trình xây dựng.
- Tạo nhiều cách khác nhau cho đối tượng được xây dựng.

10.3 Mô hình cấu trúc



Các thành phần cơ bản:

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

Trường hợp đơn giản, chúng ta có thể gộp Builder và ConcreteBuilder thành static nested class bên trong Product.

Ta xét một ví dụ cụ thể: Tạo một static nested class (đây được gọi là builder class) và copy tất cả các tham số từ class bên ngoài vào. Chúng ta nên đặt tên của static nested class này theo định dạng là tên class + Builder. Ví dụ class là Computer thì builder class sẽ là ComputerBuilder. Class Builder có một hàm khởi tạo public với tất cả các thuộc tính bắt buộc, ngoài ra còn có các method setter cho các tham số tùy chọn. Cung cấp method build() trong Class Builder để trả về đối tượng mà client cần.

```java
package builderpattern;

public class Computer {
	// required parameters
	private String HDD;
	private String RAM;
	// optional parameters
	private boolean isGraphicsCardEnabled;
	private boolean isBluetoothEnabled;

	public String getHDD() {
	  return HDD;
	}

	public String getRAM() {
	  return RAM;
	}

	public boolean isGraphicsCardEnabled() {
	  return isGraphicsCardEnabled;
	}

	public boolean isBluetoothEnabled() {
	  return isBluetoothEnabled;
	}

	private Computer(ComputerBuilder builder) {
	  this.HDD = builder.HDD;
	  this.RAM = builder.RAM;
	  this.isGraphicsCardEnabled = builder.isGraphicsCardEnabled;
```

```
    private Computer(ComputerBuilder builder) {
        this.HDD = builder.HDD;
        this.RAM = builder.RAM;
        this.isGraphicsCardEnabled = builder.isGraphicsCardEnabled;
        this.isBluetoothEnabled = builder.isBluetoothEnabled;
    }

    @Override
    public String toString() {
        return "Computer [HDD=" + HDD + ", RAM=" + RAM + ", isGraphicsCardEnabled=" + isGraphicsCardEnabled
                + ", isBluetoothEnabled=" + isBluetoothEnabled + "]";
    }

```

```
    // Builder Class
    public static class ComputerBuilder {
        // required parameters
        private String HDD;
        private String RAM;
        // optional parameters
        private boolean isGraphicsCardEnabled;
        private boolean isBluetoothEnabled;

        public ComputerBuilder(String hdd, String ram) {
            this.HDD = hdd;
            this.RAM = ram;
        }

        public ComputerBuilder setGraphicsCardEnabled(boolean isGraphicsCardEnabled) {
            this.isGraphicsCardEnabled = isGraphicsCardEnabled;
            return this;
        }

        public ComputerBuilder setBluetoothEnabled(boolean isBluetoothEnabled) {
            this.isBluetoothEnabled = isBluetoothEnabled;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }
}
```

Class Computer.java chỉ có method getter và không có hàm khởi tạo public nên chỉ có một cách duy nhất để lấy một đối tượng Computer là thông qua class ComputerBuilder.

Demo:

```
1  package builderpattern;
2
3  public class DemoBuilderPattern {
4        public static void main(String[] args) {
5          // Using builder to get the object in a single line of code and
6          // without any inconsistent state or arguments management issues
7          Computer comp = new Computer.ComputerBuilder("500 GB", "2 GB").setBluetoothEnabled(true)
8              .setGraphicsCardEnabled(true).build();
9
10         System.out.println(comp);
11       }
12 }
```

Và chúng ta có được kết quả:

```
<terminated> DemoBuilderPattern [Java Application] C:\Program Files\Java\jdk-16\bin\javaw.exe  (Jan 25, 2022, 1:10:00 AM – 1:10:01 AM)
Computer [HDD=500 GB, RAM=2 GB, isGraphicsCardEnabled=true, isBluetoothEnabled=true]
```

Tương tự, thay vì tạo nested static class ComputerBuilder bên trong class Computer chúng ta có thể định nghĩa nó thành 1 class riêng như mô hình UML ở trên.

10.4 Builder Pattern trong thực tế

Một số ví dụ sử dụng Buider Pattern trong JDK:

- java.lang.StringBuilder.append()
- java.lang.StringBuffer.append()

Một ví dụ về việc sử dụng Builder khác là xây dựng một tài liệu XML. Ngoài ra Builder Pattern cũng có những ứng dụng thực tế khác khá gần gũi, như cho việc gọi món tại một cửa hàng thức ăn nhanh (trên một phần mềm nào đó).

Tổng kết

Báo cáo đã trình bày một cách chi tiết nhất có thể về 10 design patterns thuộc nhóm 1. Tuy nhiên cách trình bày có thể vẫn chưa phải là tốt nhất, có thể vẫn chưa thật sự chi tiết và đầy đủ, một lần nữa các thành viên trong nhóm rất mong nhận được sự đóng góp ý kiến cũng như đưa ra những nhận xét, đánh giá của thầy.