



BÁO CÁO MÔN CÁC THÀNH PHẦN PHẦN MỀM

DESIGN PATTERNS

Người hướng dẫn: **Thầy QUẢN THÁI HÀ**

Người thực hiện: **TRẦN NAM KHÁNH - 20001559**

NGUYỄN ANH NGUYỄN - 19000458

Lớp: **MAT3372 6CLCMTKHTT**

HÀ NỘI, NĂM 2022

Mục lục

1	1. Giới thiệu tổng quan về Design Pattern	1
1.1	Lợi ích của Design Pattern	1
1.2	Creational design pattern	1
1.3	Structural design patterns	2
1.4	Behavioral design patterns	3
2	2. Facade Pattern	4
2.1	Đặt vấn đề - Rạp hát tại gia	4
2.2	Dịnh nghĩa và Mô hình cấu trúc	6
2.2.1	Dịnh nghĩa	6
2.2.2	Mô hình cấu trúc	6
2.3	Facade Pattern: Rạp hát - Set ON!	7
2.4	Thực tế	9
3	3. Singleton Pattern	10
3.1	Giới thiệu	10
3.1.1	Đặt vấn đề	10
3.1.2	Mục đích sử dụng	10
3.2	Dịnh nghĩa và mô hình cấu trúc	11
3.2.1	Dịnh nghĩa	11
3.2.2	Mô hình cấu trúc	11
3.3	Cách cài đặt	11
3.3.1	Cài đặt chung	11
3.3.2	Cài đặt cho ví dụ cụ thể	12
3.4	Ví dụ thực tế	13
4	4. Template Pattern	14
4.1	Đặt vấn đề - Pha cà phê và Pha trà	14
4.2	Dịnh nghĩa và Mô hình cấu trúc	17
4.2.1	Dịnh nghĩa	17
4.2.2	Mô hình cấu trúc	17
4.3	Thực tế	17
5	5. Observer Pattern	18
5.1	Giới thiệu	18
5.1.1	Đặt vấn đề	18
5.1.2	Mục đích sử dụng	18
5.2	Dịnh nghĩa và mô hình cấu trúc	19
5.2.1	Dịnh nghĩa	19
5.2.2	Mô hình cấu trúc	19
5.3	Cách cài đặt	20
5.3.1	Cài đặt chung	20
5.3.2	Cài đặt cho một bài toán cụ thể	21

5.4	Ví dụ thực tế	26
6	6. State Pattern	28
6.1	Dặt vấn đề - Máy Gumball	28
6.2	Dịnh nghĩa và Mô hình cấu trúc	36
6.2.1	Dịnh nghĩa	36
6.2.2	Mô hình cấu trúc	36
6.3	Thực tế	36
7	7. Factory Pattern	38
7.1	Giới thiệu	38
7.1.1	Dặt vấn đề	38
7.1.2	Mục đích sử dụng	38
7.2	Dịnh nghĩa và mô hình cấu trúc	39
7.2.1	Dịnh nghĩa	39
7.2.2	Mô hình cấu trúc	39
7.3	Cách cài đặt	40
7.4	Ví dụ thực tế	43
8	8. Composite Pattern	44
8.1	Giới thiệu tổng quan	44
8.2	Dịnh nghĩa và Mô hình cấu trúc	44
8.2.1	Dịnh nghĩa	44
8.2.2	Mô hình cấu trúc	45
8.3	Ví dụ	46
8.4	Thực tế	49
9	9. Iterator	50
9.1	Giới thiệu	50
9.1.1	Dặt vấn đề	50
9.1.2	Mục đích sử dụng	50
9.2	Dịnh nghĩa và mô hình cấu trúc	51
9.2.1	Dịnh nghĩa	51
9.2.2	Mô hình cấu trúc	51
9.3	Cách cài đặt	52
9.4	Ví dụ thực tế	54
10	10. Builder Pattern	55
10.1	Dặt vấn đề	55
10.2	Dịnh nghĩa và Mô hình cấu trúc	55
10.2.1	Dịnh nghĩa	55
10.3	Mô hình cấu trúc	56
10.4	Cài đặt	56
10.5	Thực tế	59
11	11. Decorator Pattern	60
11.1	Giới thiệu	60
11.1.1	Dặt vấn đề	60
11.1.2	Mục đích sử dụng	60

11.2 Định nghĩa và mô hình cấu trúc	60
11.2.1 Định nghĩa	60
11.2.2 Mô hình cấu trúc	60
11.3 Cách cài đặt	61
11.4 Ví dụ thực tế	68
12 12. Kết luận	69

1. Giới thiệu tổng quan về Design Pattern

Trong kĩ thuật phần mềm, **Design Pattern** là giải pháp chung có thể lặp lại cho những vấn đề thường xảy ra khi thiết kế phần mềm. Design Pattern không phải là một thiết kế hoàn chỉnh mà chuyển hóa trực tiếp thành code. Nó là một khuôn mẫu được đúc kết từ những người đi trước nhằm giải quyết vấn đề mà có thể được sử dụng trong những trường hợp khác nhau

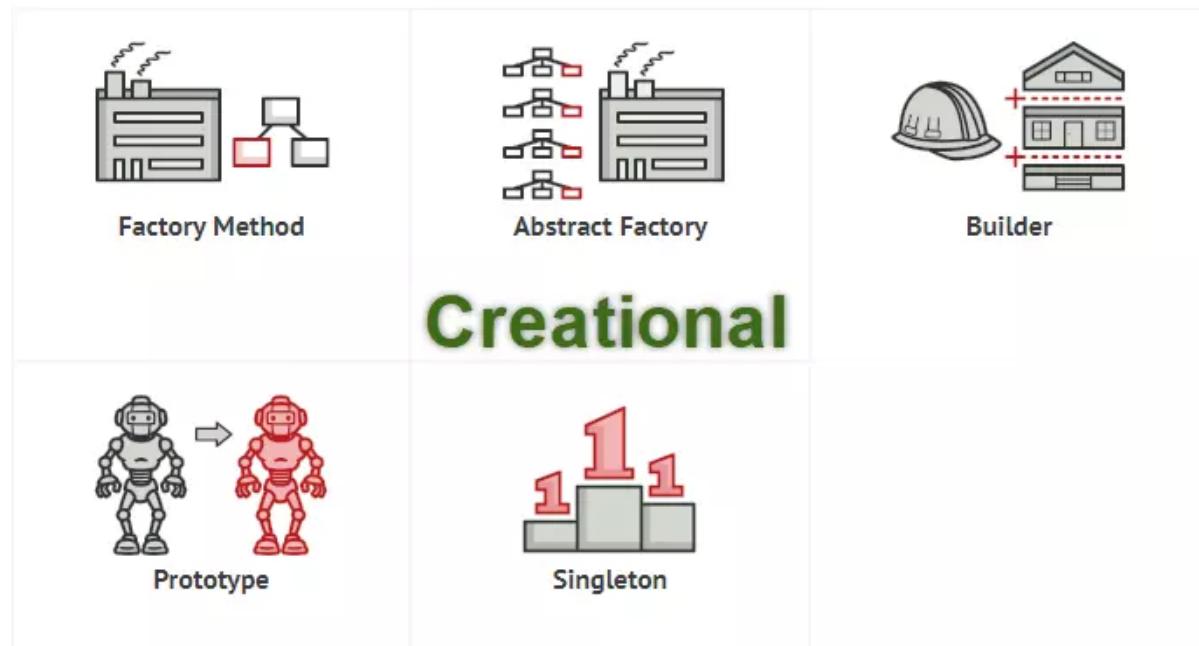
1.1 Lợi ích của Design Pattern

Design Pattern có thể tăng tốc quá trình phát triển bằng cách cung cấp các mô hình phát triển đã được kiểm nghiệm, chứng minh. Thiết kế phần mềm hiệu quả đòi hỏi các vấn đề đáng kể có thể không hiện hữu cho đến khi đến bước triển khai sau này. Việc sử dụng lại các design pattern giúp ngăn chặn các vấn đề nghiêm trọng và cải thiện kĩ năng đọc code cho lập trình viên và kiến trúc sư quen thuộc với các pattern

Thông thường, mọi người chỉ hiểu cách để áp dụng một số kĩ thuật thiết kế phần mềm vào những vấn đề nhất định. Những kĩ thuật này khó để áp dụng với những vấn đề ở tầm rộng hơn. Design Pattern cung cấp một giải pháp chung, được ghi lại ở định dạng không bị bó chặt với một vấn đề cụ th

Hơn nữa, các design pattern được cải tiến theo thời gian, làm chúng trở nên mạnh mẽ hơn so với các thiết kế đặc biệt

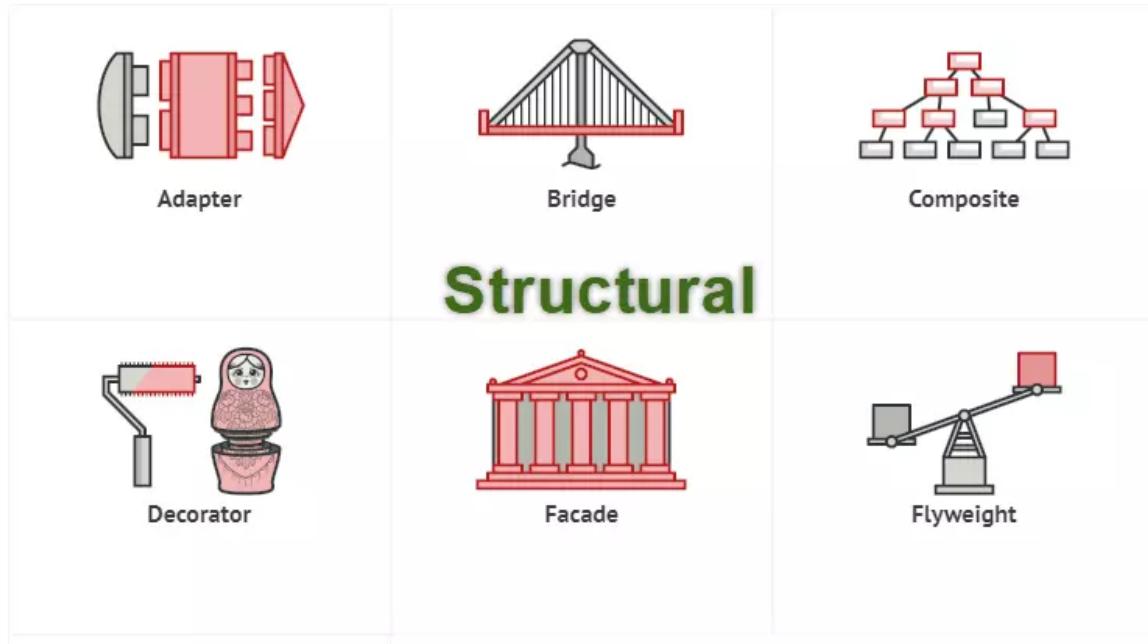
1.2 Creational design pattern



Những design pattern này có thiên hướng khởi tạo lớp:

- **Abstract Factory:** Tạo thực thể cho một số họ của các lớp
- **Builder:** Tách biệt cách dựng đối tượng và đối tượng được tạo ra
- **Factory Method:** Tạo thực thể cho một số lớp dẫn xuất
- **Object Pool:** Kiểm soát tài nguyên bằng việc tái sử dụng những đối tượng không dùng đến nữa
- **Prototype:** Một thực thể được khởi tạo đầy đủ được dùng để sao chép hoặc nhân bản
- **Singleton:** Một lớp mà chỉ một thực thể duy nhất có thể tồn tại

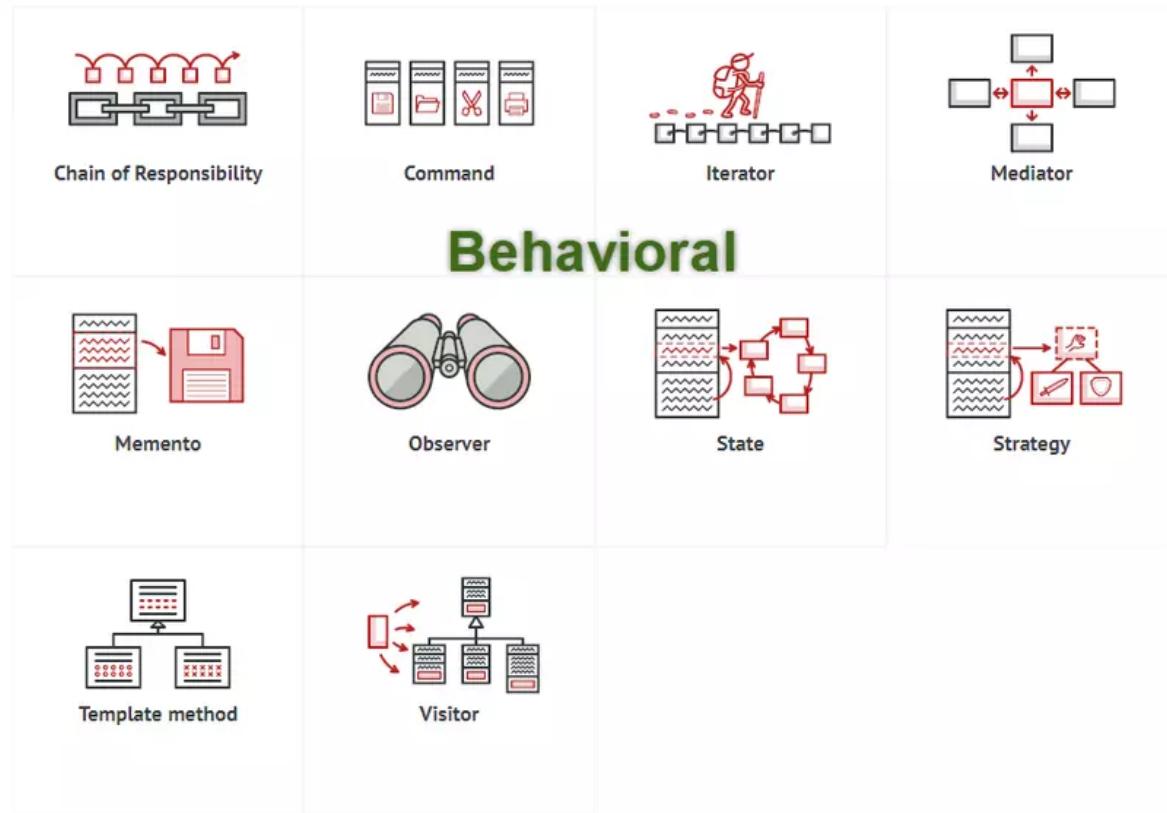
1.3 Structural design patterns



Những design pattern này có thiên hướng liên kết giữa lớp và đối tượng với nhau:

- **Adapter:** Kết nối Interface của nhiều lớp khác nhau
- **Bridge:** Tách biệt Interface của đối tượng với cách mà nó được triển khai
- **Composite:** Một cấu trúc cây của các đối tượng đơn giản và đối tượng composite
- **Decorator:** Thêm hành vi cho các đối tượng một cách linh động
- **Facade:** Một lớp đơn lẻ đại diện cho cả một hệ thống con
- **Flyweight:** Tái sử dụng đối tượng tương tự đã tồn tại để tối ưu bộ nhớ
- **Proxy:** Một đối tượng đại diện cho một đối tượng khác

1.4 Behavioral design patterns



Những design pattern này có thiên hướng giao tiếp giữa các đối tượng

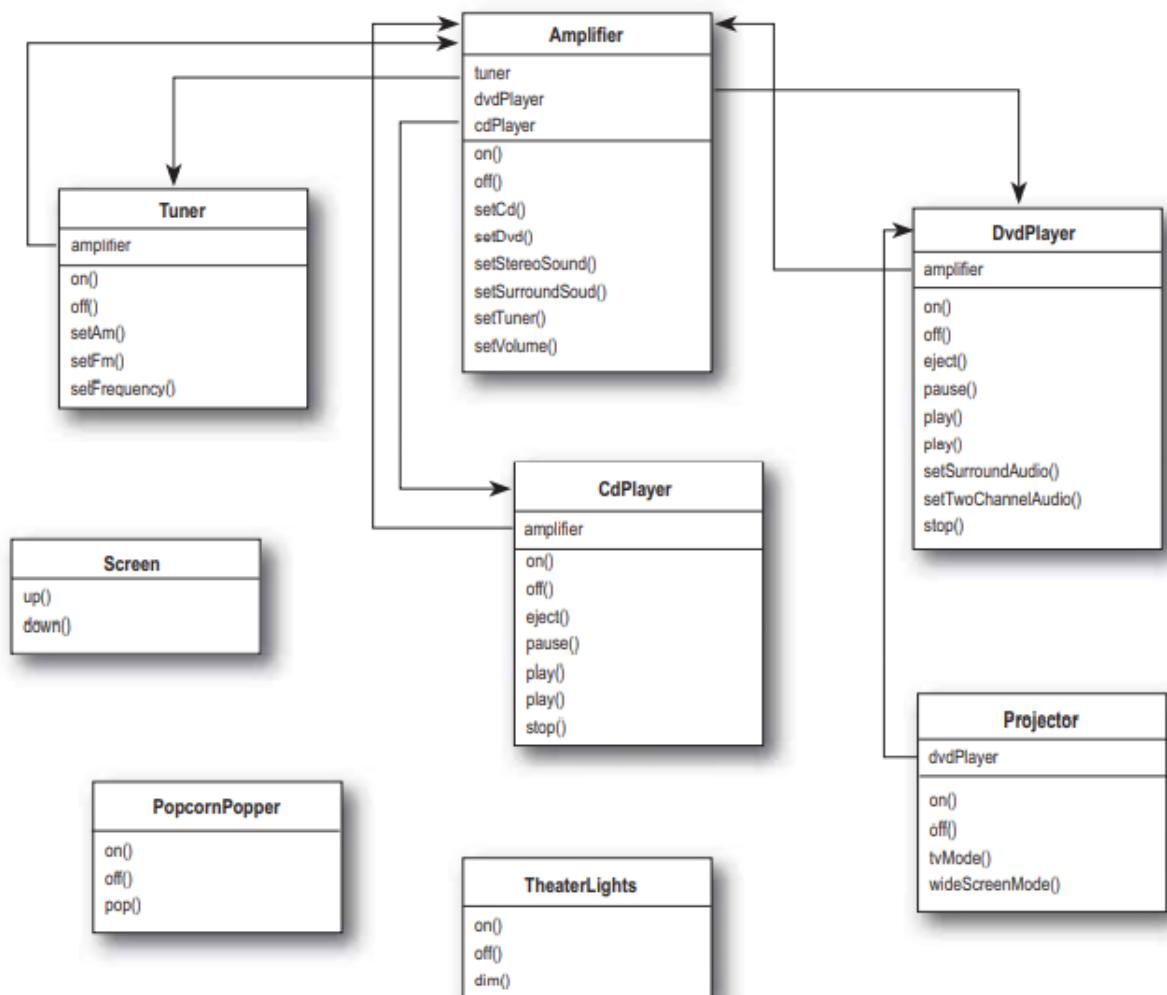
- **Chain of responsibility:** Một cách truyền yêu cầu tới một chuỗi các đối tượng
- **Command:** Đóng gói một lệnh dưới dạng một đối tượng
- **Interpreter:** Một cách bao gồm các yếu tố ngôn ngữ trong chương trình
- **Iterator:** Truy cập tuần tự các phần tử của một tập hợp
- **Mediator:** Định nghĩa cách giao tiếp đơn giản giữa các lớp
- **Memento:** Lưu giữ và khôi phục trạng thái bên trong của đối tượng
- **Null Object:** Thiết kế để hoạt động như một giá trị mặc định của một đối tượng
- **Observer:** Một cách để thông báo sự thay đổi tới một số lớp
- **State:** Thay đổi hành vi đối tượng khi trạng thái của nó thay đổi
- **Strategy:** Đóng gói một thuật toán bên trong một lớp
- **Template method:** Triển khai các bước chính xác của thuật toán cho một lớp con
- **Visitor:** Định nghĩa các thao tác tới một lớp mà không làm thay đổi

2. Facade Pattern

2.1 Đặt vấn đề - Rạp hát tại gia

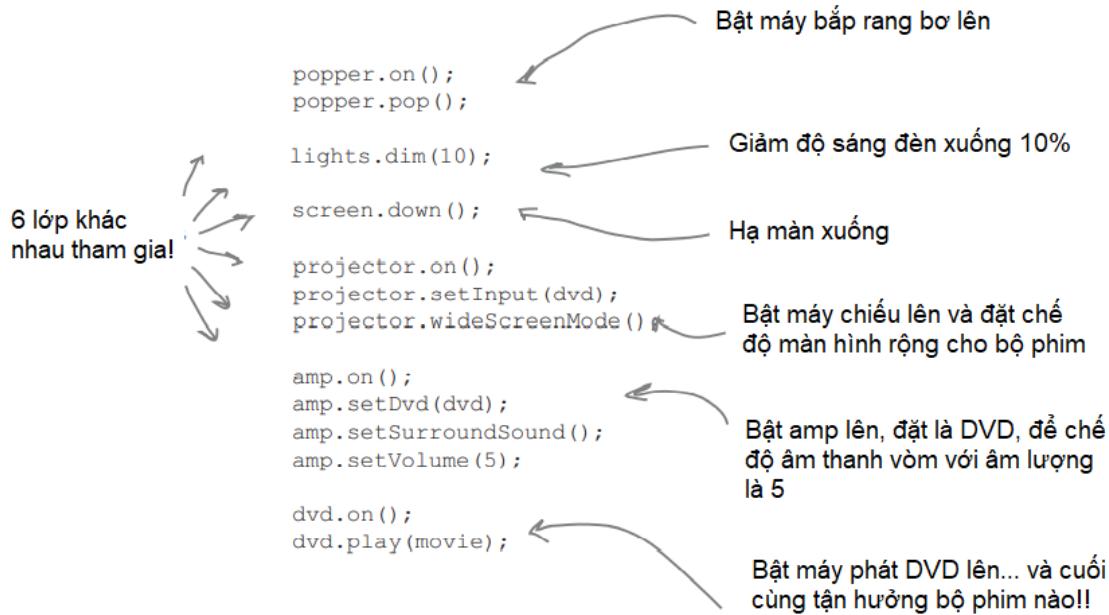
Trước khi chúng ta đi chi tiết vào Facade Pattern, hãy tưởng tượng việc xây dựng rạp hát cho ngôi nhà thân yêu của mình

Sau dày công nghiên cứu, cuối cùng ta có được một hệ thống hoàn chỉnh với một đĩa DVD, máy chiếu kỹ thuật số, màn hình tự động và cả một chút bông ngô nữa được mô tả trong hình dưới:



/

Lấy 1 chiếc đĩa DVD, pha tách trà và thư giãn. Nhưng mà chờ đã, trước khi xem phim ta cần phải thực hiện những việc sau: bật bếp rang bơ lên, giảm độ sáng, đặt màn hình xuống, bật máy chiếu lên,... Với việc gọi lớp và phương thức thông thường



Việc này phát sinh 1 số vấn đề như sau:

- Khi bộ phim kết thúc, làm sao để tắt mọi thứ? Hay phải làm hết những việc này theo chiều ngược lại?
- Việc nghe đĩa CD hay Radio cũng phức tạp không kém gì DVD?
- Khi nâng cấp hệ thống, chương trình trong Client cũng phải sửa đổi

Facade Pattern là giải pháp cho những vấn đề trên. Mục đích chính là để giảm độ phức tạp trong hệ thống và đơn giản hóa interface. Không những vậy, Facade thể hiện tính trừu tượng: giấu đi những thứ được triển khai bên trong và chỉ để lại Interface cho Client dễ dàng sử dụng

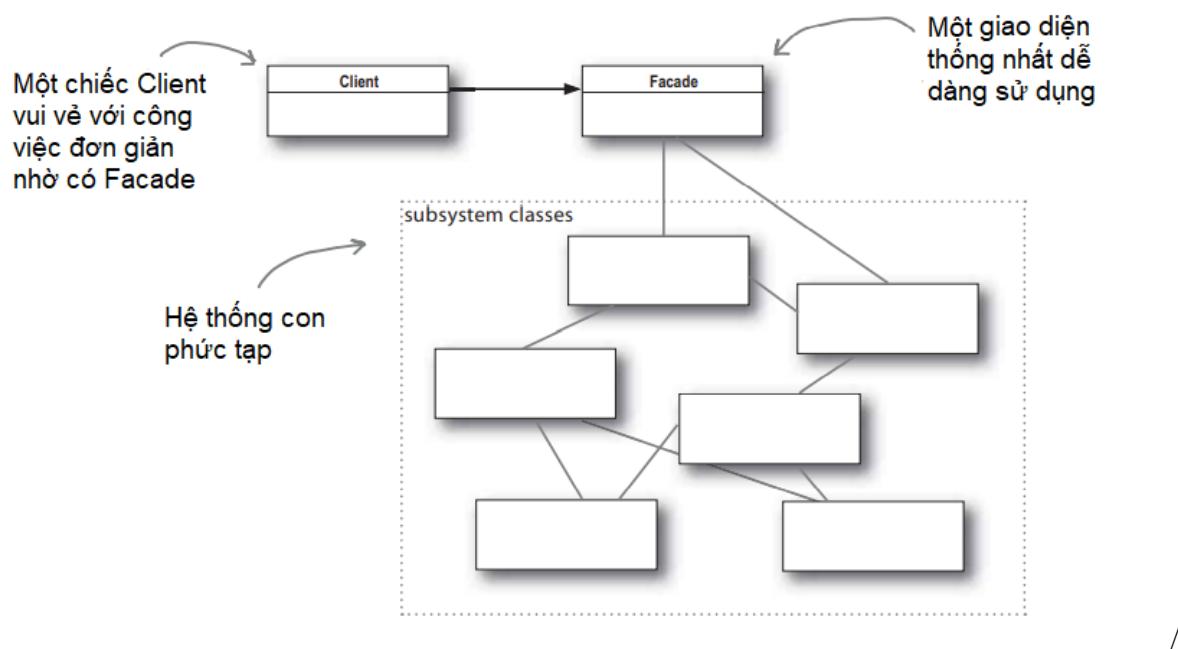
2.2 Định nghĩa và Mô hình cấu trúc

2.2.1 Định nghĩa

Facade Pattern cung cấp một giao diện thống nhất một tập hợp những giao diện trong một hệ thống con. Facade xác định một giao diện ở mức cao để thuận tiện trong việc sử dụng hệ thống con

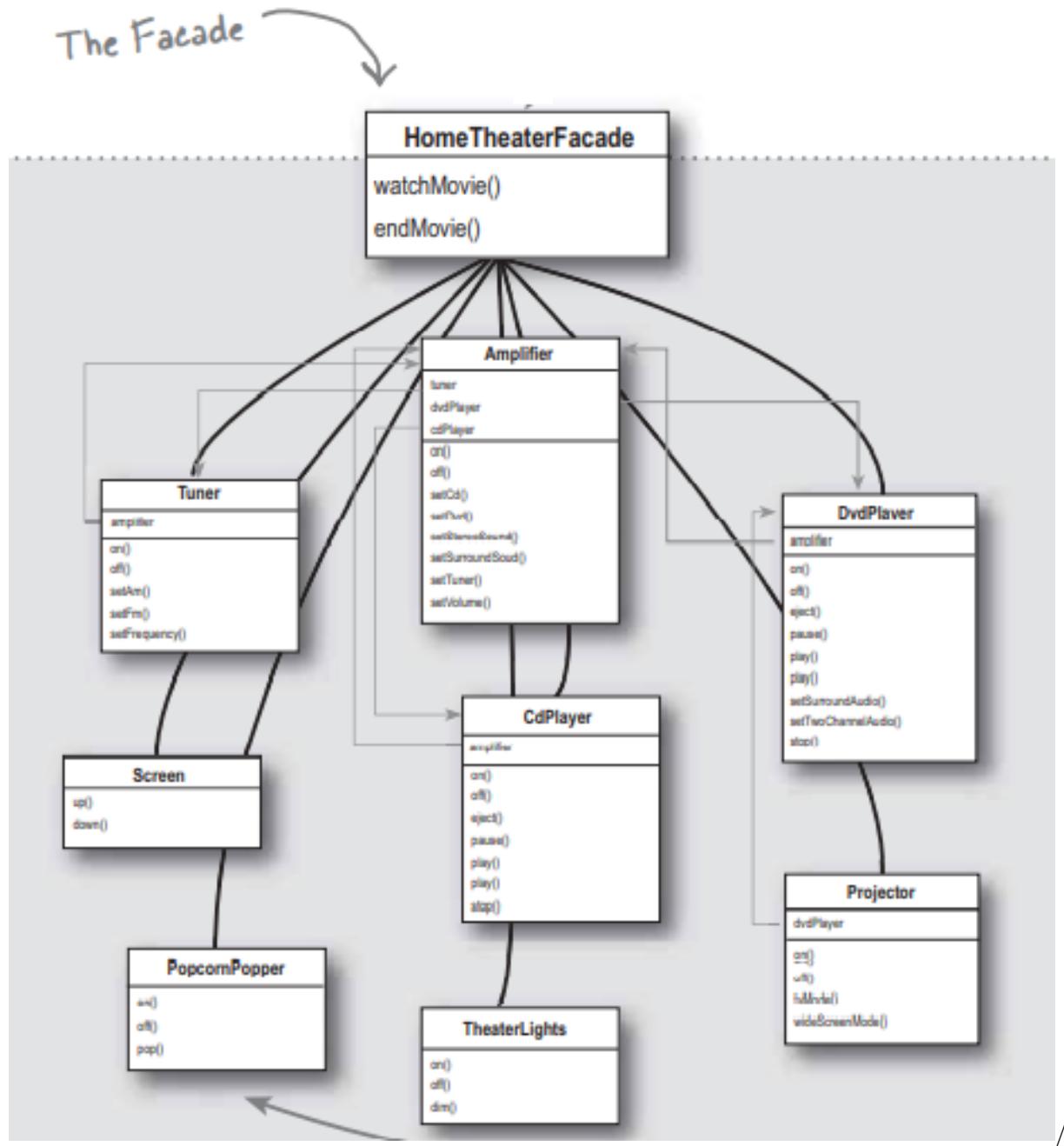
2.2.2 Mô hình cấu trúc

Mô hình cấu trúc của Facade Pattern được biểu diễn như sau:



2.3 Facade Pattern: Rạp hát - Set ON!

Nhờ vào mô hình cấu trúc đã được định nghĩa ở trên, ta có thể nâng cấp rạp hát với mô hình như sau:



Về mặt ý tưởng, ta sẽ tạo một lớp mới tên là **HomeTheaterFacade**, nơi chứa một vài phương thức như `watchMovie()`. Lúc này, lớp Facade sẽ coi những thành phần trong rạp như một hệ thống con và gọi chúng trong phương thức `watchMovie()`. Client sẽ gọi những phương thức trong lớp Facade, không phải hệ thống con. Nói cách khác, khi cần xem phim thì việc gọi phương thức `watchMovie()` và phương thức đó sẽ tự động kết nối đến đèn, đầu đĩa DVD, máy chiếu, bộ khuếch đại,...

Dưới đây là đoạn code minh họa cho lớp HomeTheaterFacade:

```

public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
        Tuner tuner,
        DvdPlayer dvd,
        CdPlayer cd,
        Projector projector,
        Screen screen,
        TheaterLights lights,
        PopcornPopper popper) {
        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

    public void watchMovie(String movie) {
        System.out.println("Get ready to watch a movie...");
        popper.on();
        popper.pop();
        lights.dim(10);
        screen.down();
        projector.on();
        projector.wideScreenMode();
        amp.on();
        amp.setDvd(dvd);
        amp.setSurroundSound();
        amp.setVolume(5);
        dvd.on();
        dvd.play(movie);
    }

    public void endMovie() {
        System.out.println("Shutting movie theater down...");
        popper.off();
        lights.on();
        screen.up();
        projector.off();
        amp.off();
        dvd.stop();
        dvd.eject();
        dvd.off();
    }
}

```

Đây được gọi là composition. Những thứ này là các thành phần trong hệ thống con mà ta sẽ dùng

Facade được truyền một tham chiếu tới từng thành phần trong hệ thống con trong hàm dựng. Sau đó facade sẽ gán giá trị cho biến instance tương ứng

watchMovie() theo các bước ta làm thủ công trước đó nhưng giờ sẽ đưa hết vào một phương thức làm mọi việc. Lưu ý rằng mỗi việc ta đang ủy quyền công việc tới thành phần tương ứng trong hệ thống con

và endMovie() đảm nhận việc tắt mọi thứ cho ta. Một lần nữa, mỗi công việc sẽ được ủy quyền tới thành phần phù hợp trong hệ thống con

Đến giờ thưởng thức phim sau bao công sức:

```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here

        HomeTheaterFacade homeTheater =
            new HomeTheaterFacade(amp, tuner, dvd, cd,
                projector, screen, lights, popper); ← Đầu tiên ta khởi tạo
                                                Facade với những thành phần trong hệ thống con

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }
}
```

Đầu tiên ta khởi tạo Facade với những thành phần trong hệ thống con

Sử dụng giao diện được đơn giản hóa để bật bộ phim rồi sau đó tắt nó đi

2.4 Thực tế

Trong ET++ application framework [WGM88], một ứng dụng có thể được tích hợp sẵn các công cụ duyệt để kiểm tra các đối tượng của nó tại thời điểm chạy. Những công cụ duyệt này được triển khai trong một hệ thống con riêng biệt bao gồm một lớp **Facade** được gọi là "ProgrammingEnvironment". Những facade này định nghĩa các toán tử như InspectObject và InspectClass để truy cập các trình duyệt

Hệ điều hành The Choices [CIRM93] sử dụng facade để hợp thành nhiều framework thành một. Có những phần chính trong Choices là quy trình, không gian lưu trữ và không gian địa chỉ. Ứng với mỗi phần sẽ có một hệ thống con tương ứng, triển khai như một framework hỗ trợ chuyển Choices tới nhiều nền tảng phần cứng khác nhau. 2 trong những hệ thống con có một "representative" (i.e.facade). Những representatives là FileSystemInterface (không gian lưu trữ) và Domain (không gian địa chỉ)

3. Singleton Pattern

3.1 Giới thiệu

3.1.1 Đặt vấn đề

Hầu hết các đối tượng trong một ứng dụng đều chịu trách nhiệm cho công việc của chúng và truy xuất dữ liệu tự lưu trữ (self-contained data) và các tham chiếu trong phạm vi được đưa ra của chúng. Tuy nhiên, có nhiều đối tượng có thêm những nhiệm vụ và có ảnh hưởng rộng hơn, chẳng hạn như quản lý các nguồn tài nguyên bị giới hạn hoặc theo dõi toàn bộ trạng thái của hệ thống. Ví dụ có nhiều máy in trong một hệ thống nhưng chỉ tồn tại duy nhất một Sprinter Spooler (Phần quản lí máy in).

Hay giả sử trong ứng dụng có chức năng bật tắt nhạc nền chẳng hạn, khi người dùng mở app thì ứng dụng sẽ tự động mở nhạc nền và nếu người dùng muốn tắt thì phải vào setting trong app để tắt nó, trong setting của app cho phép người dùng quản lí việc mở hay tắt nhạc, và trong trường hợp này sẽ cần sử dụng singleton để quản lí. Chắc chắn chúng ta phải cần duy nhất 1 instance để có thể ra lệnh bật hay tắt, tại sao ? vì đơn giản bạn không thể tạo 1 instance để mở nhạc rồi sau đó lại tạo 1 instance khác để tắt nhạc, lúc này sẽ có 2 instance được tạo ra, 2 instance này không liên quan đến nhau nên không thể thực hiện thực hiện việc cho nhau được, nói cách khác instance nào bật thì chỉ có instance đó mới được phép tắt nên dẫn đến phải cần 1 instance.

Để phục vụ nhu cầu kể trên Singleton Pattern đã được tạo ra.

3.1.2 Mục đích sử dụng

Mô hình thiết kế singleton giải quyết các vấn đề bằng cách cho phép nó:

- Đảm bảo rằng một class chỉ có một instance
- Dễ dàng truy cập instance duy nhất của một class
- Kiểm soát sự khởi tạo của instance
- Hạn chế số lượng phiên bản
- Truy cập một biến toàn cục

Mô hình thiết kế singleton mô tả cách giải quyết các vấn đề như:

- Ẩn các constructor của một class .
- Định nghĩa một hoạt động tĩnh công khai (getInstance()) trả về thể hiện duy nhất của class.

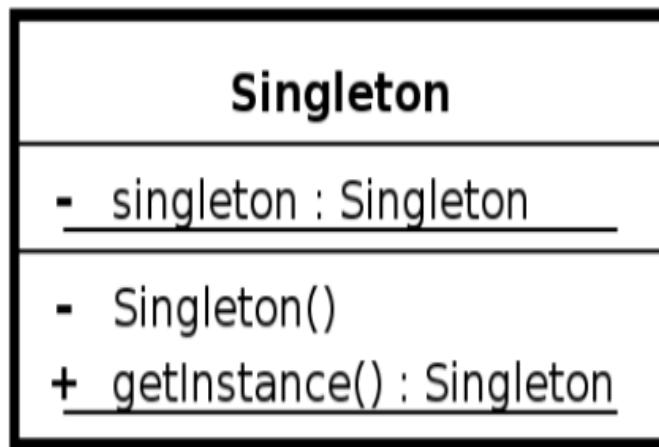
Về bản chất, mô hình Singleton buộc nó phải chịu trách nhiệm đảm bảo rằng nó chỉ được khởi tạo một lần. Một phương thức khởi tạo ẩn — được khai báo private hoặc protected đảm bảo rằng lớp không bao giờ có thể được khởi tạo từ bên ngoài lớp. Hoạt động tĩnh công cộng có thể được truy cập bằng cách sử dụng tên class và tên operation. Ví dụ Singleton.getInstance():

3.2 Định nghĩa và mô hình cấu trúc

3.2.1 Định nghĩa

Singleton là một Design Pattern thuộc nhóm khởi tạo(Creational). Nó đảm bảo rằng một class chỉ có duy nhất một instance được khởi tạo và chỉ có một cách (tùy cần) để có quyền truy cập vào instance đó.

3.2.2 Mô hình cấu trúc



Hình 3-1: Mô hình cấu trúc Singleton Pattern

-Singleton chỉ liên quan đến một lớp duy nhất (thường không được gọi là Singleton). Lớp đó là một lớp đầy đủ với các thuộc tính và phương thức khác (không được hiển thị).

-Lớp có một biến static trả về một thể hiện duy nhất của lớp.

-Lớp có một phương thức khởi tạo riêng (để ngăn mã khác khởi tạo lớp) và một phương thức tĩnh cung cấp quyền truy cập vào cá thể đơn lẻ.

3.3 Cách cài đặt

3.3.1 Cài đặt chung

Để cài đặt Singleton Pattern chúng ta cần làm hai bước.

Bước 1: Cần để class có duy nhất một instance

- Private constructor của class đó để đảm bảo rằng class khác không thể truy cập vào constructor và tạo ra instance mới.
- Tạo một biến private static là thể hiện của class đó để đảm bảo rằng nó là duy nhất và chỉ được tạo ra trong class đó thôi.

Bước 2: Cung cấp một cách toàn cầu để truy cập tới instance đó

- Tạo một public static method trả về instance vừa khởi tạo bên trên, đây là cách duy nhất để class khác có thể truy cập vào instance của class này.

Dưới đây là code minh họa cài đặt Singleton Pattern.

```
source > singleton > Singleton.java > Singleton > getInstance()
1 package source.singleton;
2
3 public final class Singleton {
4
5     private static final Singleton INSTANCE = new Singleton();
6
7     private Singleton() {}
8
9     public static Singleton getInstance() {
10         return INSTANCE;
11     }
12 }
```

Hình 3-2: Singleton class

3.3.2 Cài đặt cho ví dụ cụ thể

Dưới đây là ví dụ áp dụng Singleton Pattern.

Link code cài đặt: <https://github.com/nanhus/OOP-DesginPatten/tree/master/source/singleton>

```
source > singleton > DemoSingleton.java > ...
1 package source.singleton;
2
3 public class DemoSingleton {
4     public static void main(String[] args) {
5         Singleton singleton = Singleton.getInstance();
6         Singleton anotherSingleton = Singleton.getInstance();
7         System.out.println(singleton.hashCode());
8         System.out.println(anotherSingleton.hashCode());
9     }
10 }
11 }
```

Hình 3-3: Demo Singleton Class

```
PS C:\Users\pc\Desktop\Final OOP> & 'C:\Program Files\Java\jdk-16.0.2\bin\java' '-cp' 'C:\Users\pc\AppData\Roaming\Code\User\workspaceStorage\t_ws\Final OOP_33bd73b6\bin' 'source.singleton.DemoSingleton'
1523554304
1523554304
PS C:\Users\pc\Desktop\Final OOP>
```

Hình 3-4: Output

Như chúng ta đã thấy output cho ra kết quả là 2 ô nhớ cùng địa chỉ.Như vậy Singleton Pattern đã được cài đặt.

3.4 Ví dụ thực tế

- Centralized manager of resources
 - Window manager
 - File system manager
 - ...
- Logger classes
- Factories
 - Đặc biệt là những vấn đề IDs
 - Singleton thường được kết hợp với các mẫu Factory Method và Abstract Factory

4. Template Pattern

4.1 Đặt vấn đề - Pha cà phê và Pha trà

Một số người không thể sống nếu thiếu đi cà phê, một số người không thể sống nếu thiếu đi trà. Việc pha cà phê và pha trà khá giống nhau

Công thức làm cà phê:

- Dun sôi nước
- Pha cà phê vào nước sôi
- Rót cà phê vào cốc
- Thêm đường và sữa

Công thức làm trà:

- Dun sôi nước
- Ngâm trà trong nước sôi
- Rót trà vào cốc
- Thêm chanh

Hãy thử pha cà phê và trà bằng code:

Cà phê:

```
public class Coffee {  
    void prepareRecipe() {  
        boilWater();  
        brewCoffeeGrinds();  
        pourInCup();  
        addSugarAndMilk();  
    }  
  
    public void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    public void brewCoffeeGrinds() {  
        System.out.println("Dripping Coffee through filter");  
    }  
  
    public void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
  
    public void addSugarAndMilk() {  
        System.out.println("Adding Sugar and Milk");  
    }  
}
```

Đây là công thức pha cà phê của ta

Mỗi bước được thực hiện vào một phương thức riêng

Mỗi phương thức thực hiện một bước của thuật toán. Ở đây có phương thức để dun sôi nước, pha cà phê, rót cà phê vào cốc và thêm đường, sữa

Trà:

```

public class Tea {
    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
}

```

Cái này nhìn khá giống với gì ta làm với Coffee chỉ khác ở bước thứ hai và bước thứ tư

chú ý rằng 2 bước này y hệt như Coffee. Vậy ta đang gấp vấn đề lặp code ở đây

2 phương thức này cụ thể hóa cho Tea

Rõ ràng có thể thấy code bị dư thừa và lặp lại khá nhiều. Ta phải hạn chế bằng cách nhóm những điểm chung vào một lớp vì cách thức làm cà phê và trà khá tương đồng

Lưu ý rằng cả 2 công thức đều được xây dựng trên một thuật toán:

- Dun sôi nước
- Dùng nước sôi trích xuất trà / cà phê
- Đổ thành phẩm vào cốc
- Thêm gia vị thích hợp

Về mặt ý tưởng để cải tiến, ta sẽ tạo một lớp cha tên là **CaffeineBeverage**. Bên trong lớp này sẽ có một phương thức **prepareRecipe()** thực hiện thuật toán chung cho cả cà phê lẫn trà

Do thuật toán chung có 4 bước nên trong phương thức **prepareRecipe()** cũng sẽ có 4 phương thức đại diện: **boilWater()**, **brew()**, **pourInCup()**, **addCondiments()**

Do cách trích xuất và thêm gia vị của cà phê, trà là khác nhau nên phương thức **brew()**, **addCondiments()** sẽ đánh dấu abstract. Còn lại sẽ được triển khai trong lớp cha

Dưới đây là code minh họa cho lớp **CaffeineBeverage**:

```
public abstract class CaffeineBeverage {
    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

Bây giờ, phương thức **prepareRecipe()** sẽ được sử dụng cho việc làm trà và cà phê. **prepareRecipe()** được khai báo là **final** vì ta không muốn các lớp con ghi đè lên và thay đổi công thức. Ta đã khái quát bước 2 và bước 4 để **brew()** thức uống và **addCondiments()**

Vì cà phê và trà xử lý những phương thức theo cách riêng nên ta sẽ để chúng là **abstract**. Hãy để lớp con tự lo việc của mình!

Nhớ rằng ta đã chuyển những thứ này sang lớp **CaffeineBeverage**

Dây là code cho 2 lớp Trà và cà phê với cách thiết kế mới:

```
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
```

Như cách thiết kế của ta, Tea và Coffee giờ sẽ kế thừa từ CaffeineBeverage

Tea cần định nghĩa **brew()** và **addCondiments()** - 2 phương thức trừu tượng từ Beverage

```
public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Với cà phê ta làm tương tự

Tổng kết lại, những gì ta đã làm là cài đặt **Template Method Pattern**. Phương thức **prepareRecipe()** được gọi là **Template Method**

Qua ví dụ trên, **Template Method Pattern** giúp tối đa hóa việc tái sử dụng code giữa các lớp. Hơn nữa, thuật toán sẽ nằm yên một chỗ thuận tiện trong quá trình sửa đổi. Cuối cùng, Template cung cấp một Framework hỗ trợ các loại đồ uống chứa caffeine khác. Đó cũng là mục đích sử dụng chính của **Template Method Pattern**.

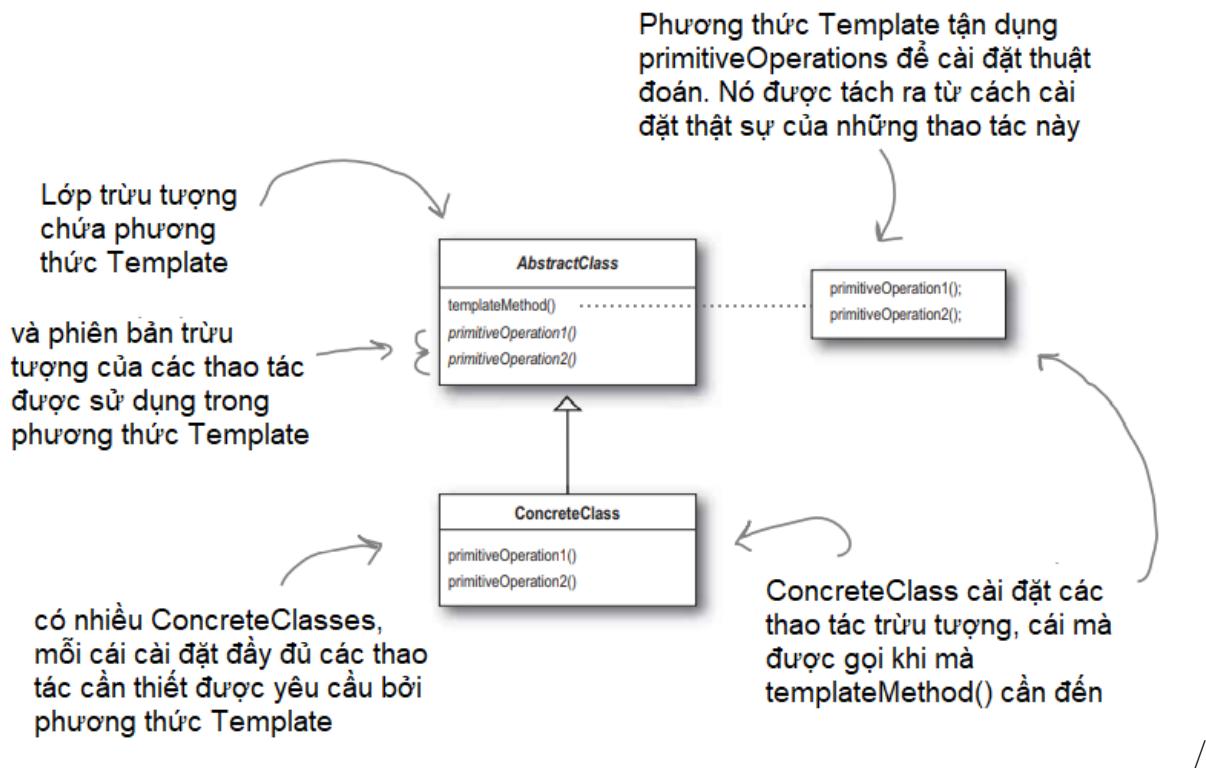
4.2 Định nghĩa và Mô hình cấu trúc

4.2.1 Định nghĩa

Template Method Pattern định ra khung xương cho thuật toán trong một phương thức, trì hoãn một số bước tới những lớp con. Template Method cho phép những lớp con tự định nghĩa lại một số bước nhất định mà không làm thay đổi cấu trúc thuật toán

4.2.2 Mô hình cấu trúc

Mô hình cấu trúc của Template Method Pattern được mô tả như sau:



4.3 Thực tế

Template Method Pattern khá là cơ bản đến nỗi nó được thấy ở hầu hết các lớp trừu tượng

Dưới đây là những phương thức Template trong bộ thư viện chuẩn của Java:

- Tất cả các phương thức không phải trừu tượng của `java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader`, `java.io.Writer`
- Tất cả các phương thức không phải trừu tượng của `java.util.AbstractList`, `java.util.AbstractSet`, `java.util.AbstractMap`

5. Observer Pattern

5.1 Giới thiệu

5.1.1 Đặt vấn đề

Chúng ta cần thiết kế và xây dựng một hệ thống phần mềm cho một trạm quan sát thời tiết. Dữ liệu của hệ thống được xây dựng thông qua một đối tượng gọi là WeatherData - đóng vai trò theo dõi điều kiện thời tiết hiện tại (nhiệt độ, độ ẩm, áp suất).

Yêu cầu ứng dụng cung cấp 3 phần tử màn hình hiển thị:

- CurrentConditionDisplay
- StatisticsDisplay
- ForecastDisplay

Tất cả sẽ được cập nhật theo thời gian thực mỗi khi đối tượng WeatherData lấy được dữ liệu mới nhất.

Ngoài ra, trạm quan sát cũng muốn "public" những API để những lập trình viên khác có thể viết ra những màn hình hiển thị thời tiết của riêng họ.

Trong trường hợp trên, để có một thiết kế tốt có nghĩa là tách rời càng nhiều càng tốt và giảm sự phụ thuộc. 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 cả các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.

5.1.2 Mục đích sử dụng

Observer Pattern phục vụ mục đích:

- Thường được sử dụng trong mối quan hệ 1-n giữa các đối tượng với nhau. Trong đó một đối tượng thay đổi và muốn thông báo cho tất cả các đối tượng liên quan biết về sự thay đổi đó.
- Cần đảm bảo rằng khi một đối tượng thay đổi trạng thái, một số đối tượng phụ thuộc kết thúc mở sẽ được cập nhật tự động.
- Có thể một đối tượng có thể thông báo cho một số đối tượng khác kết thúc mở.

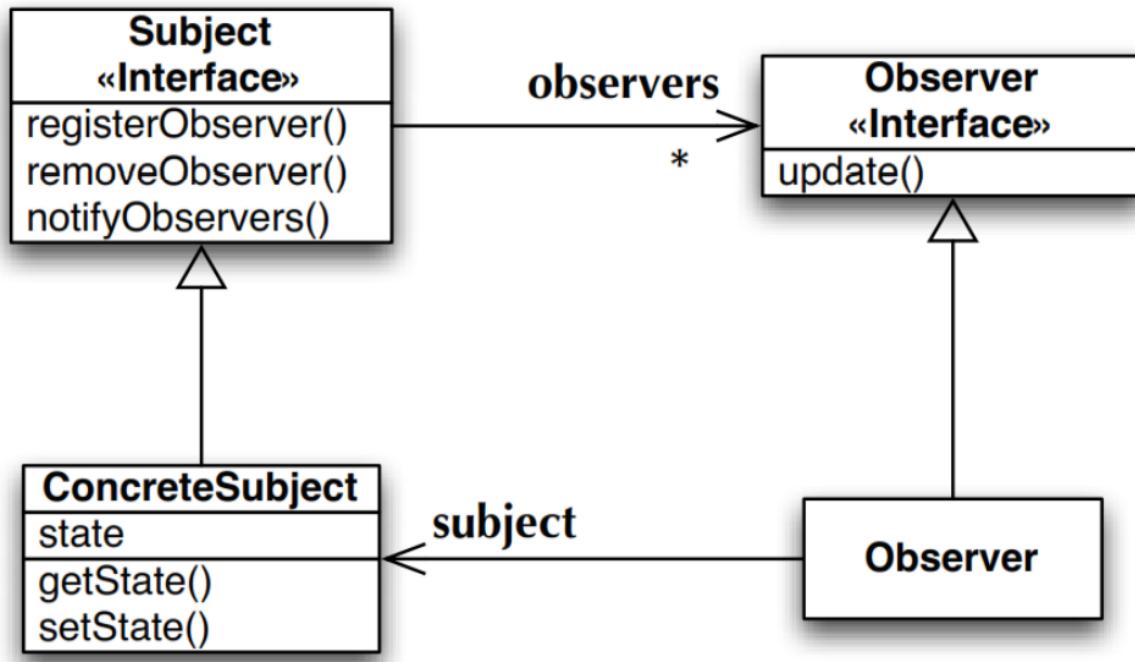
Việc xác định phụ thuộc một-nhiều giữa các đối tượng bằng cách xác định một đối tượng (chủ thể) cập nhật trạng thái của các đối tượng phụ thuộc một cách trực tiếp là không linh hoạt vì nó kết hợp chủ thể với các đối tượng phụ thuộc cụ thể. Tuy nhiên, nó có thể có ý nghĩa từ quan điểm hiệu suất hoặc nếu việc triển khai đối tượng được kết hợp chặt chẽ (hãy nghĩ đến các cấu trúc nhân cấp thấp thực thi hàng nghìn lần một giây). Các đối tượng được kết hợp chặt chẽ có thể khó triển khai trong một số trường hợp và khó sử dụng lại vì chúng tham chiếu và biết về (và cách cập nhật) nhiều đối tượng khác nhau với các giao diện khác nhau. Trong các tình huống khác, các đối tượng được kết hợp chặt chẽ có thể là một lựa chọn tốt hơn vì trình biên dịch sẽ có thể phát hiện lỗi tại thời điểm biên dịch và tối ưu hóa mã ở cấp lệnh CPU.

5.2 Định nghĩa và mô hình cấu trúc

5.2.1 Định nghĩa

Observer Pattern là một Design Pattern thuộc nhóm hành vi (Behavioral). Nó xác định sự phụ thuộc một-nhiều giữa một tập hợp các đối tượng, chẳng hạn như rằng khi một đối tượng thay đổi, tất cả những phụ thuộc của nó (observers) đều được thông báo và được cập nhật tự động.

5.2.2 Mô hình cấu trúc



Hình 5-1: Mô hình cấu trúc Observer Pattern

- **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.

5.3 Cách cài đặt

5.3.1 Cài đặt chung

Observer tạo ra sự tương tác được kết hợp lỏng lẻo giữa chủ thể và người quan sát

- Điều này có nghĩa là họ có thể tương tác với rất ít kiến thức về nhau

Chú ý

- đối tượng chỉ biết rằng Observer thực hiện giao diện Observer interface.
 - Chúng ta có thể thêm / bớt Observer thuộc bất kỳ loại nào bất kỳ lúc nào.
 - Chúng ta không bao giờ phải sửa đổi đối tượng để thêm một loại Observer mới.
- Chúng ta có thể sử dụng lại đối tượng và Observer trong các trường hợp khác.
 - Giao diện plug-and-play ở bất kỳ nơi nào Observer được sử dụng.
- Observer có thể phải biết về lớp ConcreteSubject nếu nó cung cấp nhiều phương thức liên quan đến trạng thái khác nhau.
 - Một khác, dữ liệu có thể được chuyển cho Observer thông qua phương thức update () .

Dưới đây là code minh họa cài đặt Observer Pattern.

```

source > observer >  Subject.java >  Subject >  notifyObservers()
1   package source.observer;
2
3   public interface Subject {
4       public void registerObserver(Observer o);
5
6       public void removeObserver(Observer o);
7
8       public void notifyObservers();
9   }
10

```

Hình 5-2: Subject Interface

```
source > observer > DisplayElement.java > DisplayElement > display()
1 package source.observer;
2
3 public interface DisplayElement {
4     public void display();
5 }
6
```

Hình 5-3: Display Element Interface

```
source > observer > Observer.java > ...
1 package source.observer;
2
3 public interface Observer {
4     public void update(float temp, float humidity, float pressure);
5 }
6
```

Hình 5-4: Observer Interface

5.3.2 Cài đặt cho một bài toán cụ thể

Trong bài toán đưa ra ở trên, ta có thể thấy rằng mối quan hệ 1-n ở đây đó là 1-WeatherData và n-Screen. Mỗi khi WeatherData có sự thay đổi về trạng thái (nhiệt độ, độ ẩm, áp suất) thì nó sẽ "thông báo" cho các màn hình đang "quan sát" sát nó để cập nhật lại việc hiển thị thông tin.

Do đó chúng ta có Subject ở đây là WeatherData, Observer là các màn hình hiển thị. Mỗi màn hình hiển thị có thể khác nhau, vì thế cách tốt nhất là chúng ta tạo ra một interface cho việc hiển thị.

Code minh họa cho bài toán thực tế.

Link code cài đặt: <https://github.com/nanhus/OOP-DesginPatten/tree/master/source/observer>

```
source > observer > WeatherData.java > ...
1  package source.observer;
2
3  import java.util.ArrayList;
4
5  public class WeatherData implements Subject {
6      private ArrayList<Observer> observers;
7      private float temperature;
8      private float humidity;
9      private float pressure;
10
11     public WeatherData() {
12         observers = new ArrayList<Observer>();
13     }
14
15     public void registerObserver(Observer o) {
16         observers.add(o);
17     }
18
19     public void removeObserver(Observer o) {
20         int i = observers.indexOf(o);
21         if (i >= 0) {
22             observers.remove(i);
23         }
24     }
25
26     public void notifyObservers() {
27         for (Observer o : observers) {
28             o.update(temperature, humidity, pressure);
29         }
30     }
31 }
```

Hình 5-5: Weather Data Class

```
source > observer > WeatherData.java > ...
31     public void measurementsChanged() {
32         notifyObservers();
33     }
34
35
36     public void setMeasurements(float temperature,
37         float humidity,
38         float pressure) {
39         this.temperature = temperature;
40         this.humidity = humidity;
41         this.pressure = pressure;
42         measurementsChanged();
43     }
44
45     // other WeatherData methods here
46     public float getTemperature() {
47         return temperature;
48     }
49
50     public float getHumidity() {
51         return humidity;
52     }
53
54     public float getPressure() {
55         return pressure;
56     }
57 }
58 }
```

Hình 5-6: Weather Data Class(continue)

```
source > observer >  StatisticsDisplay.java >  StatisticsDisplay
1  package source.observer;
2
3  public class StatisticsDisplay implements Observer, DisplayElement {
4      private float maxTemp = 0.0f;
5      private float minTemp = 200;
6      private float tempSum = 0.0f;
7      private int numReadings;
8      private WeatherData weatherData;
9
10     public StatisticsDisplay(WeatherData weatherData) {
11         this.weatherData = weatherData;
12         weatherData.registerObserver(this);
13     }
14
15     public void update(float temp, float humidity, float pressure) {
16         tempSum += temp;
17         numReadings++;
18         if (temp > maxTemp) {
19             maxTemp = temp;
20         }
21         if (temp < minTemp) {
22             minTemp = temp;
23         }
24         display();
25     }
26
27     public void display() {
28         System.out.println("Avg/Max/Min temperature = "
29                            + (tempSum / numReadings) + "/" + maxTemp + "/" + minTemp);
30     }
31 }
```

Hình 5-7: Statistics Display Class

```
source > observer > ForecastDisplay.java > ForecastDisplay
1 package source.observer;
2
3 public class ForecastDisplay implements Observer, DisplayElement{
4     private float currentPressure = 29.92f;
5     private float lastPressure;
6     private WeatherData weatherData;
7
8     public ForecastDisplay(WeatherData weatherData) {
9         this.weatherData = weatherData;
10        weatherData.registerObserver(this);
11    }
12
13    public void update(float temp, float humidity, float pressure) {
14        lastPressure = currentPressure;
15        currentPressure = pressure;
16        display();
17    }
18
19    public void display() {
20        System.out.print("Forecast: ");
21        if (currentPressure > lastPressure) {
22            System.out.println("Improving weather on the way!");
23        } else if (currentPressure == lastPressure) {
24            System.out.println("More of the same");
25        } else if (currentPressure < lastPressure) {
26            System.out.println("Watch out for cooler, rainy weather");
27        }
28    }
29}
```

Hình 5-8: Forecast Display Class

```

source > observer > CurrentConditionsDisplay.java > CurrentConditionsDisplay
● 1 package source.observer;
2
3 public class CurrentConditionsDisplay implements Observer, DisplayElement {
4     private float temperature;
5     private float humidity;
6     private Subject weatherData;
7
8     public CurrentConditionsDisplay(Subject weatherData) {
9         this.weatherData = weatherData;
10        weatherData.registerObserver(this);
11    }
12
13     public void update(float temperature, float humidity, float pressure) {
14         this.temperature = temperature;
15         this.humidity = humidity;
16         display();
17     }
18
19     public void display() {
20         System.out.println("Current conditions: " + temperature
21             + " F degrees and " + humidity + "% humidity");
22     }
23
24 }

```

Hình 5-9: Current Conditions Display Class

```

-----
Current conditions: 80.0 F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
-----
Current conditions: 82.0 F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
-----
Current conditions: 78.0 F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
-----
PS C:\Users\pc\Desktop\Final OOP>

```

Hình 5-10: Output

5.4 Ví dụ thực tế

- Sử dụng trong ứng dụng broadcast-type communication.

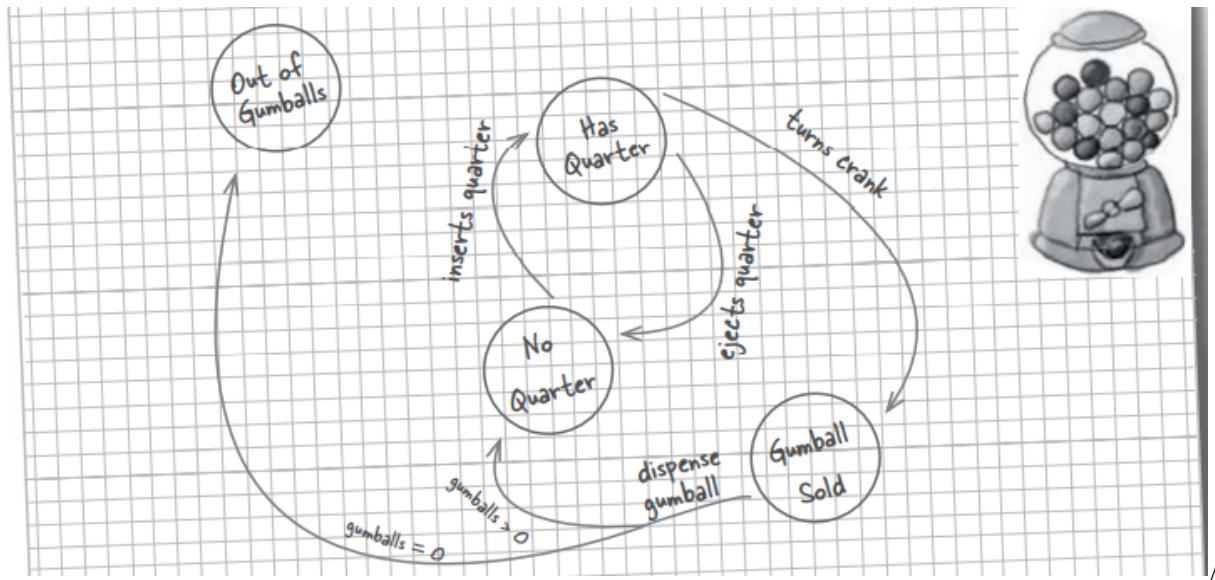
- Sử dụng để quản lý sự kiện (Event management).
- Sử dụng trong mẫu mô hình MVC (Model View Controller Pattern) : trong MVC, mẫu này được sử dụng để tách Model khỏi View. View đại diện cho Observer và Model là đối tượng Observable.

6. State Pattern

6.1 Đặt vấn đề - Máy Gumball

Ngôn ngữ Java đã và đang trở nên rất thịnh hành ở thời điểm hiện tại. Rất nhiều thiết bị hiện đại có thể xây dựng dựa trên Java. Hãy cùng thử làm một chiếc Máy Gumball!

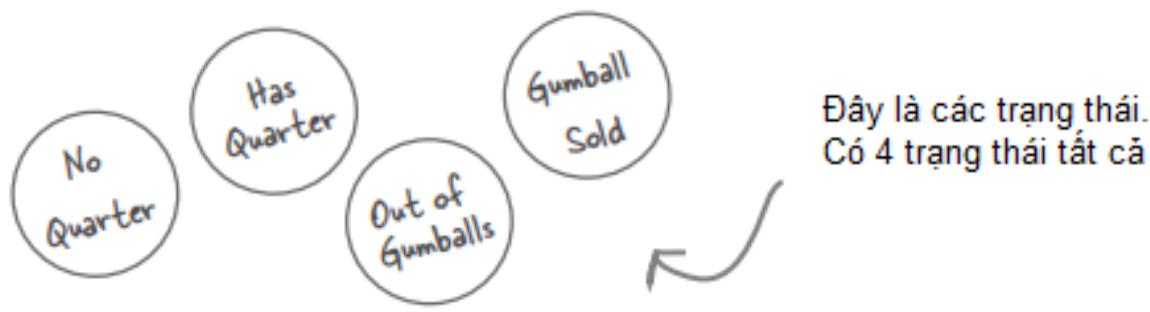
Trong máy Gumball có rất nhiều trạng thái được mô tả thông qua lược đồ dưới đây:



Bức hình có 4 đường tròn, đó là những trạng thái. **No Quarter** là trạng thái bắt đầu khi mà trong máy chưa có đồng xu nào. Khi nhét đồng xu vào, nó sẽ chuyển sang trạng thái **Has Quarter**. Lúc này, ta có thể gạt tay quay để lấy kẹo. Sau đó, máy sẽ kiểm tra lượng kẹo còn lại trong trạng thái **Gumball Sold** mà quyết định trở về trạng thái **No Quarter** hay là **Out of Gumballs**.

Ta có thể tạo ra lớp **GumballMachine** như sau:

- Bước 1: Tập hợp lại các trạng thái:



- Bước 2: Tạo các biến static để lưu trạng thái hiện tại và xác định giá trị cho mỗi trạng thái

Let's just call "Out of Gumballs"
"Sold Out" for short.

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;
```

Here's each state represented as a unique integer...

...and here's an instance variable that holds the current state. We'll go ahead and set it to "Sold Out" since the machine will be unfilled when it's first taken out of its box and turned on.

- Bước 3: Tập hợp lại những hành động có thể xảy ra trong hệ thống

inserts quarter turns crank
ejects quarter dispense

có tổng cộng
5 hành động

- Bước 4: Tạo phương thức ứng với mỗi hành động và sử dụng biểu thức điều kiện để quyết định những hành vi hợp lý ứng với mỗi trạng thái

Dưới đây là code minh họa cho lớp **GumballMachine**:

```

public class GumballMachine {
    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;
}

public GumballMachine(int count) {
    this.count = count;
    if (count > 0) {
        state = NO_QUARTER;
    }
}

Now we start implementing
the actions as methods...

```

Here are the four states; they match the states in Mighty Gumball's state diagram.

Here's the instance variable that is going to keep track of the current state we're in. We start in the SOLD_OUT state.

We have a second instance variable that keeps track of the number of gumballs in the machine.

The constructor takes an initial inventory of gumballs. If the inventory isn't zero, the machine enters state NO_QUARTER, meaning it is waiting for someone to insert a quarter, otherwise it stays in the SOLD_OUT state.

```

public void insertQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("You can't insert another quarter");
    } else if (state == NO_QUARTER) {
        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't insert a quarter, the machine is sold out");
    } else if (state == SOLD) {
        System.out.println("Please wait, we're already giving you a gumball");
    }
}

If the customer just bought a
gumball he needs to wait until the
transaction is complete before

```

When a quarter is inserted, if...

a quarter is already inserted we tell the customer;

otherwise we accept the quarter and transition to the HAS_QUARTER state.

and if the machine is sold out, we reject the quarter.

```

public void ejectQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("Quarter returned");
        state = NO_QUARTER;
    } else if (state == NO_QUARTER) {
        System.out.println("You haven't inserted a quarter");
    } else if (state == SOLD) {
        System.out.println("Sorry, you already turned the crank");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }
}

You can't eject if the machine is sold
out, it doesn't accept quarters!

```

Now, if the customer tries to remove the quarter...

If there is a quarter, we return it and go back to the NO_QUARTER state.

Otherwise, if there isn't one we can't give it back.

The customer tries to turn the crank...

If the customer just turned the crank, we can't give a refund; he already has the gumball!

```

public void turnCrank() {
    if (state == SOLD) {
        System.out.println("Turning twice doesn't get you another gumball!");
    } else if (state == NO_QUARTER) {
        System.out.println("You turned but there's no quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You turned, but there are no gumballs");
    } else if (state == HAS_QUARTER) {
        System.out.println("You turned...");
        state = SOLD;
        dispense();
    }
}

Someone's trying to cheat the machine.
We need a quarter first.
We can't deliver gumballs; there are none.
Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method.

```

```

    ↓ Called to dispense a gumball.
public void dispense() {
    if (state == SOLD) {
        System.out.println("A gumball comes rolling out the slot");
        count = count - 1;
        if (count == 0)
            System.out.println("Oops, out of gumballs!");
        state = SOLD_OUT;
    } else {
        state = NO_QUARTER;
    }
} else if (state == NO_QUARTER) {
    System.out.println("You need to pay first");
} else if (state == SOLD_OUT) {
    System.out.println("No gumball dispensed");
} else if (state == HAS_QUARTER) {
    System.out.println("No gumball dispensed");
}
}

// other methods here like toString() and refill()
}

```

We're in the SOLD state; give 'em a gumball!

Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD_OUT; otherwise, we're back to not having a quarter.

None of these should ever happen, but if they do, we give 'em an error, not a gumball.

Chương trình ta đã làm rất tuyệt vời nhưng tiếc là đời không như là mơ. Chỉ viết máy Gumball bằng một suy nghĩ thấu đáo như vậy không có nghĩa là nó sẽ dễ dàng phát triển. Giờ khi cải tiến máy với tính năng gấp đôi kẹo nhận được cho người may mắn thì chuyện gì sẽ xảy ra? Khi nghĩ lại đoạn code để nghĩ cách sửa đổi nó thì...

```

final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

public void insertQuarter() {
    // insert quarter code here
}

public void ejectQuarter() {
    // eject quarter code here
}

public void turnCrank() {
    // turn crank code here
}

public void dispense() {
    // dispense code here
}

```

First, you'd have to add a new WINNER state here. That isn't too bad...

... but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

turnCrank() will get especially messy, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.

Vì vấn đề này, ta cần có hướng thiết kế mới. Ta cần cải thiện sao cho code dễ dàng phát triển. Sau đây là cách có thể làm:

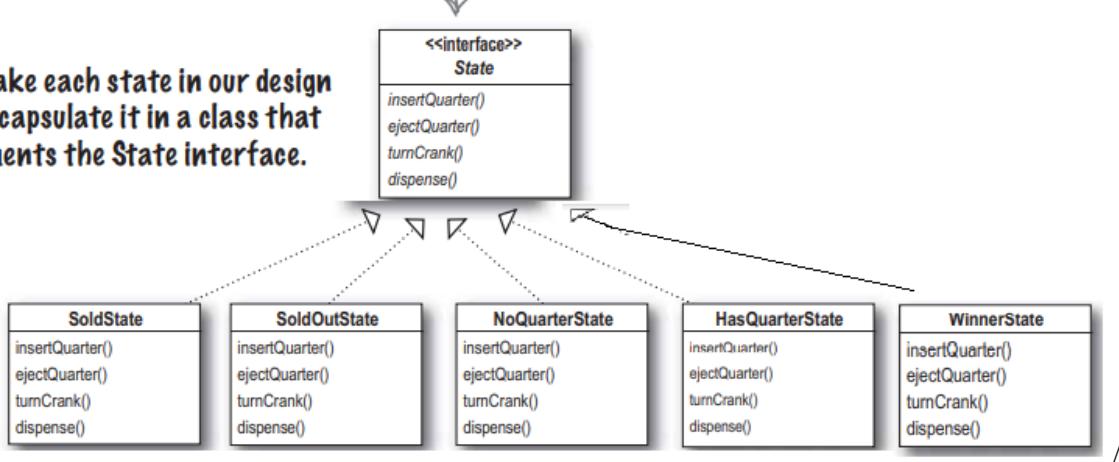
- Trước hết, ta sẽ xác định một **State Interface** chứa phương thức cho mỗi hành động của máy Gumball
- Sau đó, chúng ta bắt đầu cài đặt **State Class** ứng với mỗi trạng thái của máy. Những lớp này phụ trách cho hành vi của máy khi ở trạng thái tương ứng
- Cuối cùng, ta sẽ thay thế biểu thức điều kiện với sự ủy quyền tới những **State class** để thực hiện công việc

Dịnh nghĩa **State Interface** và các lớp:

First let's create an interface for State, which all our states implement:

Here's the interface for all states. The methods map directly to actions that could happen to the Gumball Machine (these are the same methods as in the previous code).

Then take each state in our design and encapsulate it in a class that implements the State interface.



Sau đó, tiến hành cài đặt các lớp của **State**. Bắt đầu với **NoQuarterState**:

```

First we need to implement the State interface.
public class NoQuarterState implements State {
    GumballMachine gumballMachine;
    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }
    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }
    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter");
    }
    public void turnCrank() {
        System.out.println("You turned, but there's no quarter");
    }
    public void dispense() {
        System.out.println("You need to pay first");
    }
}

```

We get passed a reference to the Gumball Machine through the constructor. We're just going to stash this in an instance variable.

If someone inserts a quarter, we print a message saying the quarter was accepted and then change the machine's state to the HasQuarterState.

You'll see how these work in just a sec...

You can't get money back if you never gave it to us!

And, you can't get a gumball if you don't pay us.

We can't be dispensing gumballs without payment

Tiếp theo là **HasQuarterState** và **SoldState**:

```

public class HasQuarterState implements State {
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        gumballMachine.setState(gumballMachine.getSoldState());
    }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}

public class SoldState implements State {
    //constructor and instance variables here

    public void insertQuarter() {
        System.out.println("Please wait, we're already giving you a gumball");
    }

    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank");
    }

    public void turnCrank() {
        System.out.println("Turning twice doesn't get you another gumball!");
    }

    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}

```

When the state is instantiated we pass it a reference to the GumballMachine. This is used to transition the machine to a different state.

An inappropriate action for this state.

Return the customer's quarter and transition back to the NoQuarterState.

When the crank is turned we transition the machine to the SoldState state by calling its setState() method and passing it the SoldState object. The SoldState object is retrieved by the getSoldState() getter method (there is one of these getter methods for each state).

Here are all the inappropriate actions for this state

And here's where the real work begins...

We're in the SoldState, which means the customer paid. So, we first need to ask the machine to release a gumball.

Then we ask the machine what the gumball count is, and either transition to the NoQuarterState or the SoldOutState.

Tương tự với 2 **SoldOutState** và **WinnerState**. Cuối cùng là lớp **GumballMachine** hoàn chỉnh:

```

public class GumballMachine {
    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State state = soldOutState;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        }
    }

    public void insertQuarter() {
        state.insertQuarter();
    }

    public void ejectQuarter() {
        state.ejectQuarter();
    }

    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

    void setState(State state) {
        this.state = state;
    }

    void releaseBall() {
        System.out.println("A gumball comes rolling out the slot...");
        if (count != 0) {
            count = count - 1;
        }
    }

    // More methods here including getters for each State...
}

```

The diagram shows the `GumballMachine` class with several annotations:

- State variables:** `soldOutState`, `noQuarterState`, `hasQuarterState`, `soldState`, `state`, and `count`. Arrows point from these variable declarations to the corresponding annotations.
- Annotations for state variables:**
 - `state`: "Here are all the States again..."
 - `count`: "The count instance variable holds the count of gumballs – initially the machine is empty."
 - `state` (inside constructor): "Our constructor takes the initial number of gumballs and stores it in an instance variable. It also creates the State instances, one of each."
 - `state` (inside `insertQuarter`): "If there are more than 0 gumballs we set the state to the NoQuarterState."
- Annotations for actions:**
 - `insertQuarter`, `ejectQuarter`, and `turnCrank`: "Now for the actions. These are VERY EASY to implement now. We just delegate to the current state."
 - `setState`: "Note that we don't need an action method for `dispense()` in `GumballMachine` because it's just an internal action; a user can't ask the machine to dispense directly. But we do call `dispense()` on the `State` object from the `turnCrank()` method."
 - `releaseBall`: "This method allows other objects (like our `State` objects) to transition the machine to a different state."
 - `releaseBall`: "The machine supports a `releaseBall()` helper method that releases the ball and decrements the count instance variable."
- Annotation for getters:**
 - A bracket annotation over the `// More methods here` comment points to the `getNoQuarterState()` and `getCount()` methods with the text: "This includes methods like `getNoQuarterState()` for getting each state object, and `getCount()` for getting the gumball count".

Nhìn lại những gì trước đó đã làm, ta đã thay đổi cấu trúc khác biệt so với bản đầu tiên nhưng chức năng vẫn không đổi. Cách thiết kế này có tên gọi là **State Pattern**. Bằng cách này, ta đã:

- Tách biệt hành vi của từng trạng thái đến từng lớp
- Loại bỏ tất cả biểu thức điều kiện rắc rối gây khó khăn trong việc bảo trì
- Dòng từng trạng thái cho việc sửa đổi nhưng vẫn để Máy Gumball được phát triển bằng cách thêm mới nhiều lớp trạng thái

- Tạo một code base và cấu trúc lớp ánh xạ tương đồng với bản thiết kế ban đầu, dễ đọc, dễ hiểu

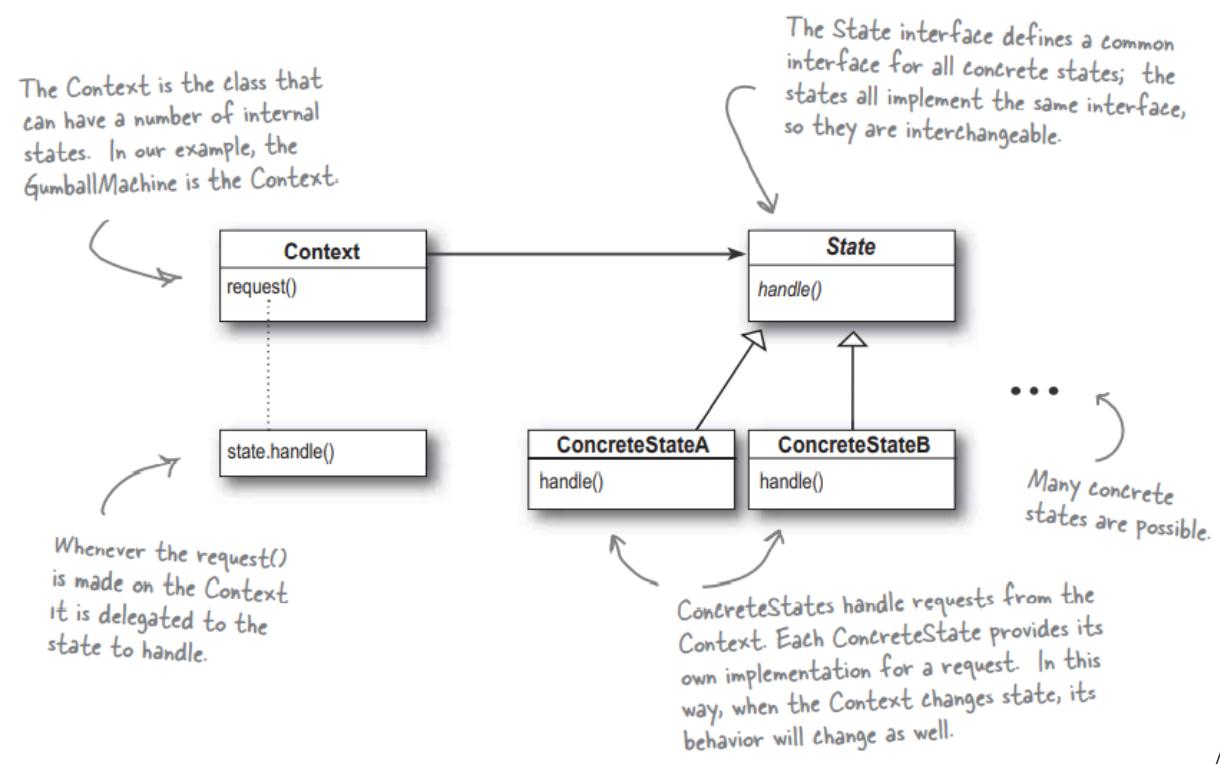
6.2 Định nghĩa và Mô hình cấu trúc

6.2.1 Định nghĩa

State Pattern cho phép một đối tượng thay đổi hành vi của nó khi trạng thái bên trong thay đổi. Đối tượng sẽ xuất hiện để thay đổi lớp của nó

6.2.2 Mô hình cấu trúc

Dưới đây là mô hình cấu trúc của **State Pattern**:



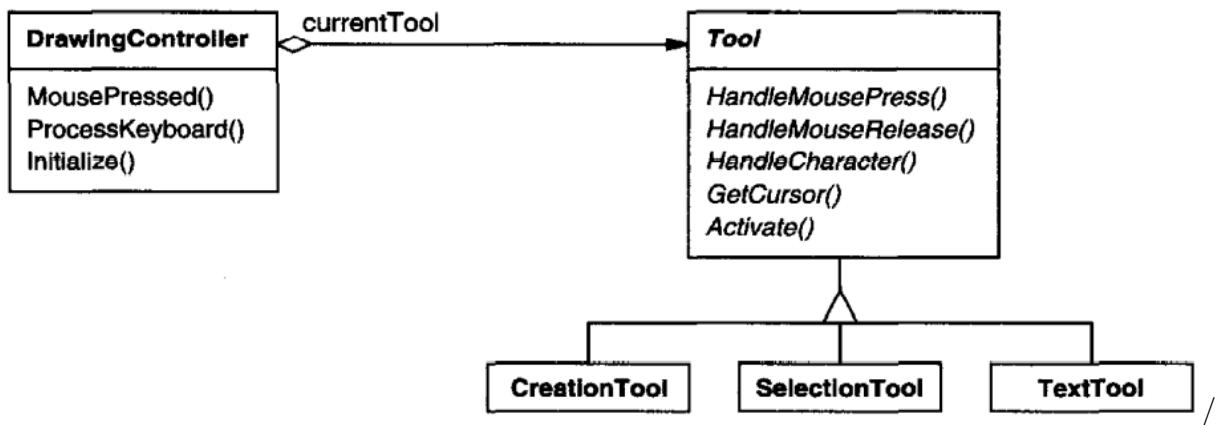
6.3 Thực tế

Johnson and Zweig [JZ91] biểu thị State pattern và ứng dụng tới giao thức kết nối TCP

Hầu hết các chương trình vẽ phác họa đều cung cấp "công cụ" để thực hiện các thao tác trực tiếp. Ví dụ: một công cụ vẽ đoạn thẳng cho phép người dùng nhập và kéo để tạo một đường mới. Một công cụ lựa chọn cho phép người dùng chọn các hình khối. Thường có một bảng các công cụ để lựa chọn. Người dùng coi việc này như nhặt một dụng cụ và sử dụng nó, nhưng trên thực tế, hành vi của editor thay đổi với công cụ hiện tại: Khi một công cụ vẽ hoạt động, ta tạo ra các hình khối; khi mà công cụ lựa chọn đang hoạt động, ta chọn các hình khối đó; và cứ thế. Ta sử dụng State pattern để thay đổi hành vi của editor tùy thuộc vào công cụ hiện tại.

Ta có thể định nghĩa lớp trùu tượng Tool để từ đó định nghĩa các lớp con mà cài đặt các hành vi cụ thể. Trình chỉnh sửa bản vẽ duy trì một đối tượng Tool và ủy quyền các yêu cầu tới nó. Trình sẽ thay thế đối tượng khi người dùng chọn công cụ mới khiến hành vi của trình vẽ thay đổi theo

Kỹ thuật này được áp dụng trong những framework hỗ trợ chỉnh sửa bản vẽ điển hình như HotDraw [Joh92] và Unidraw [VL90]. Nó cho phép phía Client định nghĩa nhiều loại công cụ một cách dễ dàng. Trong HotDraw, lớp DrawingController chuyển tiếp các yêu cầu đến đối tượng Tool hiện tại. Còn Unidraw sẽ có lớp tương ứng là Viewer và Tool. Dưới đây là mô hình cấu trúc cho Tool và DrawingController interfaces:



7. Factory Pattern

7.1 Giới thiệu

7.1.1 Đặt vấn đề

Vấn đề.

- Làm thế nào một ứng dụng có thể độc lập với cách các đối tượng của nó được tạo ra?
- Làm thế nào một lớp có thể độc lập với cách các đối tượng mà nó yêu cầu được tạo ra?
- Làm thế nào có thể tạo mối quan hệ của các đối tượng liên quan hoặc phụ thuộc?

Việc tạo các đối tượng trực tiếp bên trong lớp yêu cầu các đối tượng là không linh hoạt vì nó gán lớp với các đối tượng cụ thể và khiến nó không thể thay đổi việc khởi tạo sau này một cách độc lập với (mà không cần phải thay đổi) lớp. Nó ngăn lớp không thể sử dụng lại nếu các đối tượng khác được yêu cầu và nó làm cho lớp khó kiểm tra vì không thể thay thế các đối tượng thực bằng các đối tượng giả.

Factory Pattern mô tả cách giải quyết các vấn đề như vậy.

7.1.2 Mục đích sử dụng

Mục đích sử dụng:

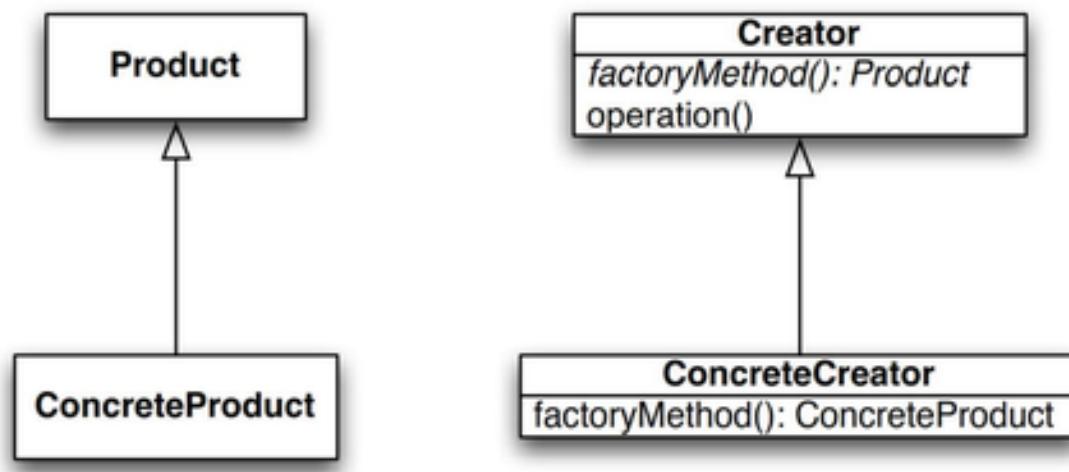
- Giúp việc khởi tạo các đối tượng mà che giấu đi xử lý logic của việc khởi tạo đó. Người dùng không biết logic thực sự được khởi tạo bên dưới phương thức factory.
- Mẫu thiết kế này cho phép các lớp con chọn kiểu đối tượng cần tạo.
- Nó thúc đẩy sự liên kết lỏng lẻo bằng cách loại bỏ sự cần thiết phải ràng buộc các lớp cụ thể vào code. Nghĩa là code chỉ tương tác với interface hoặc lớp abstract, để nó sẽ làm việc với bất kỳ lớp nào implements interface đó hoặc extends lớp abstract.
- Factory Pattern giúp giảm sự phụ thuộc giữa các module: cung cấp 1 hướng tiếp cận với Interface thay vì các implement. Giúp chương trình độc lập với những lớp cụ thể mà chúng ta cần tạo 1 đối tượng, code ở phía client sẽ không bị ảnh hưởng khi thay đổi logic ở factory hay sub class.
- Việc mở rộng code dễ dàng hơn: khi cần mở rộng, chỉ việc tạo ra những sub class và implement thêm vào factory method.
- Dễ dàng quản lý life cycle của các đối tượng được tạo bởi Factory Method Pattern.
- Thông nhất về mặt naming convention: giúp cho các developer có thể hiểu về cấu trúc source code.

7.2 Định nghĩa và mô hình cấu trúc

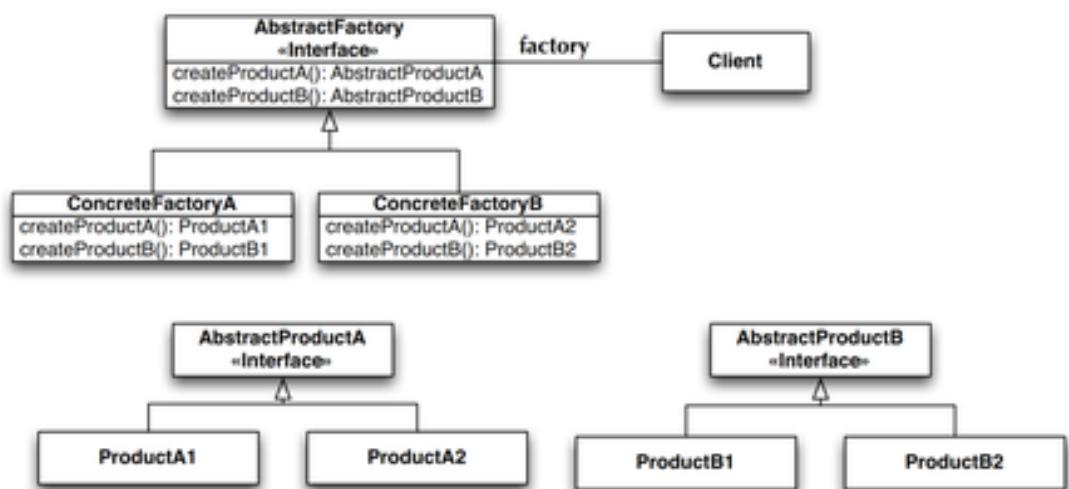
7.2.1 Định nghĩa

Mẫu thiết kế Factory Method xác định một giao diện để tạo một đối tượng, nhưng cho phép các lớp con quyết định lớp sẽ khởi tạo. Factory Method cho phép một class trì hoãn việc khởi tạo thành các lớp subclass. Mẫu thiết kế Abstract Factory cung cấp một giao diện để tạo các họ có liên quan hoặc các đối tượng phụ thuộc mà không chỉ định các lớp cụ thể của chúng.

7.2.2 Mô hình cấu trúc



Hình 7-1: Mô hình cấu trúc Factory Method



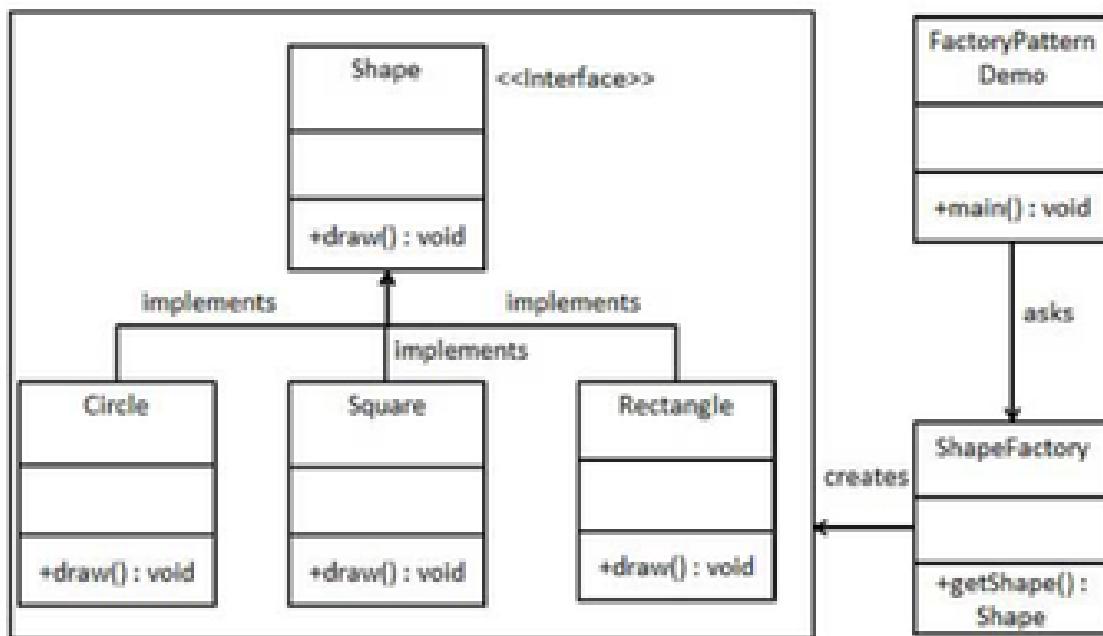
Hình 7-2: Mô hình cấu trúc Factory Pattern

- Super Class: một super class trong Factory Pattern có thể là một interface, abstract class hay một class thông thường.

- Sub Classes: các sub class sẽ cài đặt các phương thức của supper class theo nghiệp vụ riêng của nó.
- Factory Class: một class chịu trách nhiệm khởi tạo các đối tượng sub class dựa theo tham số đầu vào. Lưu ý: lớp này là Singleton hoặc cung cấp một public static method cho việc truy xuất và khởi tạo đối tượng. Factory class sử dụng if-else hoặc switch-case để xác định class con đầu ra.

7.3 Cách cài đặt

Cài đặt Factory Pattern qua 1 bài toán thực tế theo sơ đồ dưới đây.



Hình 7-3: Mô hình ví dụ

Cách cài đặt.

Link code cài đặt: <https://github.com/nanhus/OOP-DesginPatten/tree/master/source/factory>

Bước 1: Tạo 1 giao diện.

```
source > factory > Shape.java > ...
1  package source.factory;
2
3  public interface Shape {
4      void draw();
5  }
```

Hình 7-4: Shape interface

Bước 2: Tạo các đối tượng cụ thể cài đặt giao diện Shape.

```
source > factory > Circle.java > ...
1  package source.factory;
2
3  public class Circle implements Shape {
4      @Override
5      public void draw() {
6          System.out.println("Draw the circle: ");
7      }
8  }
```

Hình 7-5: Circle Class

```
source > factory > Rectangle.java > Rectangle > draw()
1  package source.factory;
2
3  public class Rectangle implements Shape {
4      @Override
5      public void draw() {
6          System.out.println("Draw the rectangle: ");
7      }
8  }
```

Hình 7-6: Rectangle Class

```
source > factory > 🏛 Square.java > ⚒ Square
1 package source.factory;
2
3 public class Square implements Shape {
4     @Override
5     public void draw() {
6         System.out.println("Draw the square: ");
7     }
8 }
9
```

Hình 7-7: Square Class

Bước 3:Tạo một class ShapeFactory để tạo đối tượng của lớp cù thê dựa trên thông tin đã cho.

```
source > factory > 🏛 ShapeFactory.java > ⚒ ShapeFactory > ⚡ getShape(String)
1 package source.factory;
2
3 public class ShapeFactory {
4     // Sử dụng phương thức getShape để lấy đối tượng có kiểu hình dạng
5     public Shape getShape(String shapeType) {
6         if (shapeType == null) {
7             return null;
8         }
9
10        if (shapeType.equalsIgnoreCase("CIRCLE")) {
11            return new Circle();
12
13        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
14            return new Rectangle();
15
16        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
17            return new Square();
18
19        }
20    }
21
22 }
23
```

Hình 7-8: Shape Factory Class

Bước 4:Sử dụng Factory để lấy đối tượng của lớp cù thê bằng cách truyền một thông tin như kiểu.

```

source > factory > DemoFactory.java > DemoFactory > main(String[])
1 package source.factory;
2
3 public class DemoFactory {
4     Run | Debug
5     public static void main(String[] args) {
6         ShapeFactory shapeFactory = new ShapeFactory();
7         Shape shape1 = shapeFactory.getShape("CIRCLE");
8         shape1.draw();
9         Shape shape2 = shapeFactory.getShape("RECTANGLE");
10        shape2.draw();
11        Shape shape3 = shapeFactory.getShape("SQUARE");
12        shape3.draw();
13    }
14

```

Hình 7-9: Demo Factory class

Output:

```

Draw the circle:
Draw the rectangle:
Draw the square:
PS C:\Users\pc\Desktop\Final OOP>

```

Hình 7-10: Output

7.4 Ví dụ thực tế

- Áp dụng quản lí các mặt hàng
- JDK: java.util.Calendar, ResourceBundle, NumberFormat, ...
- BeanFactory trong Spring Framework.
- SessionFactory trong Hibernate Framework.
- Nó định nghĩa các Abstract Factory trong WidgetKit và DialogKit để tạo ra các đối tượng giao diện người dùng có giao diện cụ thể.
- Sử dụng mẫuAbstract Factory để đạt được tính di động trên các hệ thống khác nhau (ví dụ: X Windows và SunView). Lớp trừu tượng WindowSystem xác định giao diện để tạo đối tượng đại diện cho tài nguyên hệ thống cửa sổ (ví dụ: MakeWindow, MakeFont, MakeColor).

8. Composite Pattern

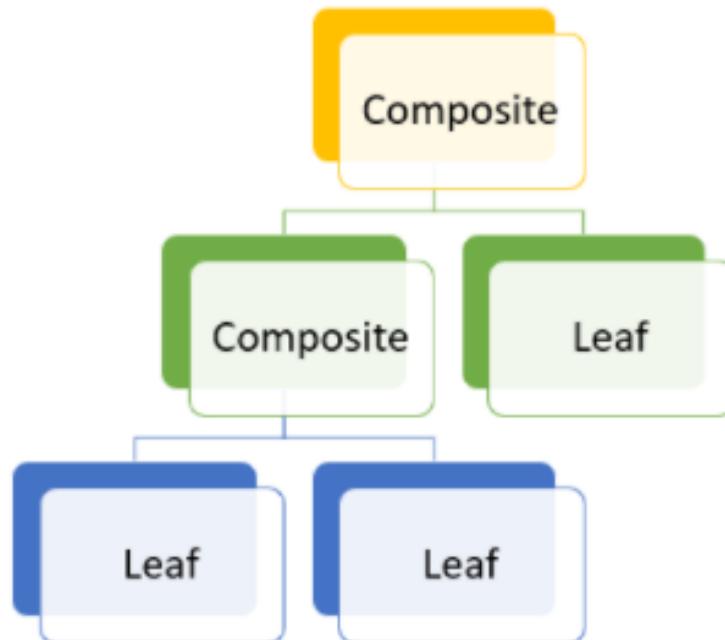
8.1 Giới thiệu tổng quan

Nhắc đến cụm từ **Composite** thì **Composite** được hiểu như một thứ được tạo nên từ các thành phần. Một **Đối tượng Composite** là một đối tượng được tạo nên bởi nhiều đối tượng. Ví dụ như một chiếc ô tô được tạo nên bởi 4 chiếc bánh xe vậy hay nước được tạo bởi Hydro và Oxy vậy. Đó là cách mà **Composite Pattern** được tạo ra. Xét những trường hợp mà ta muốn gom một nhóm đối tượng thành một tập hợp để thuận tiện trong quá trình thao tác

8.2 Định nghĩa và Mô hình cấu trúc

8.2.1 Định nghĩa

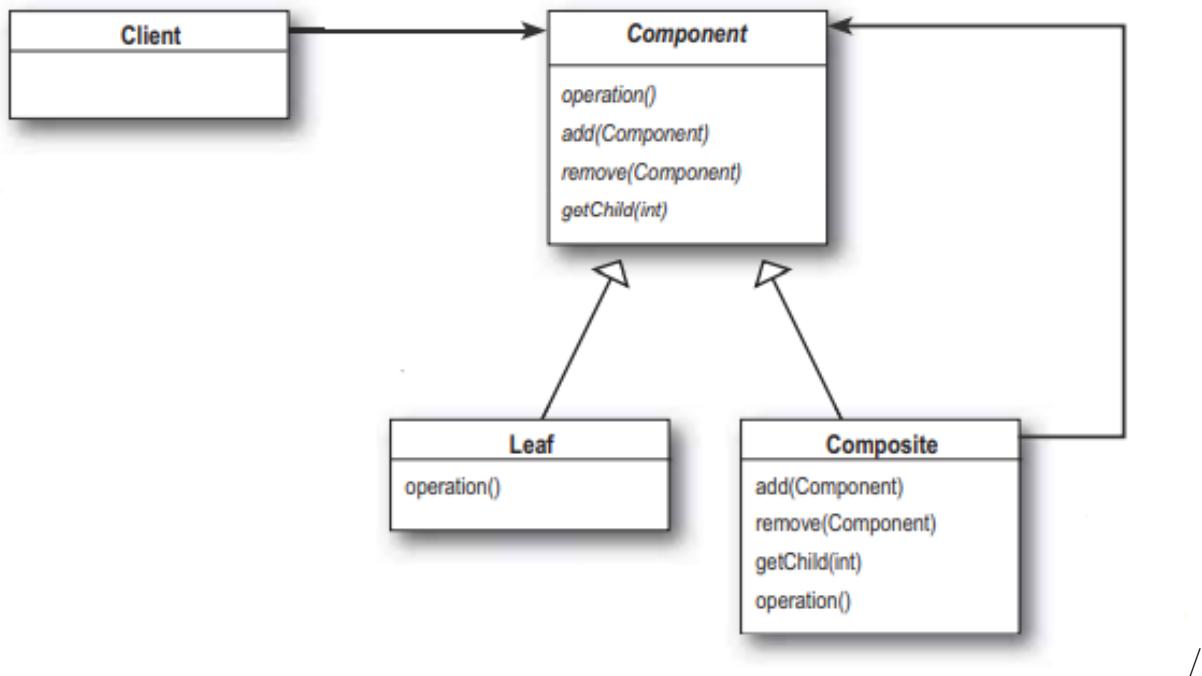
Composite Pattern cho phép ta tổ chức những đối tượng dưới dạng cây phân tầng. **Composite** cho phép coi các đối tượng riêng lẻ và các thành phần của các đối tượng như một thể thống nhất



/

8.2.2 Mô hình cấu trúc

Mô hình cấu trúc của **Composite Pattern** được biểu diễn như sau:

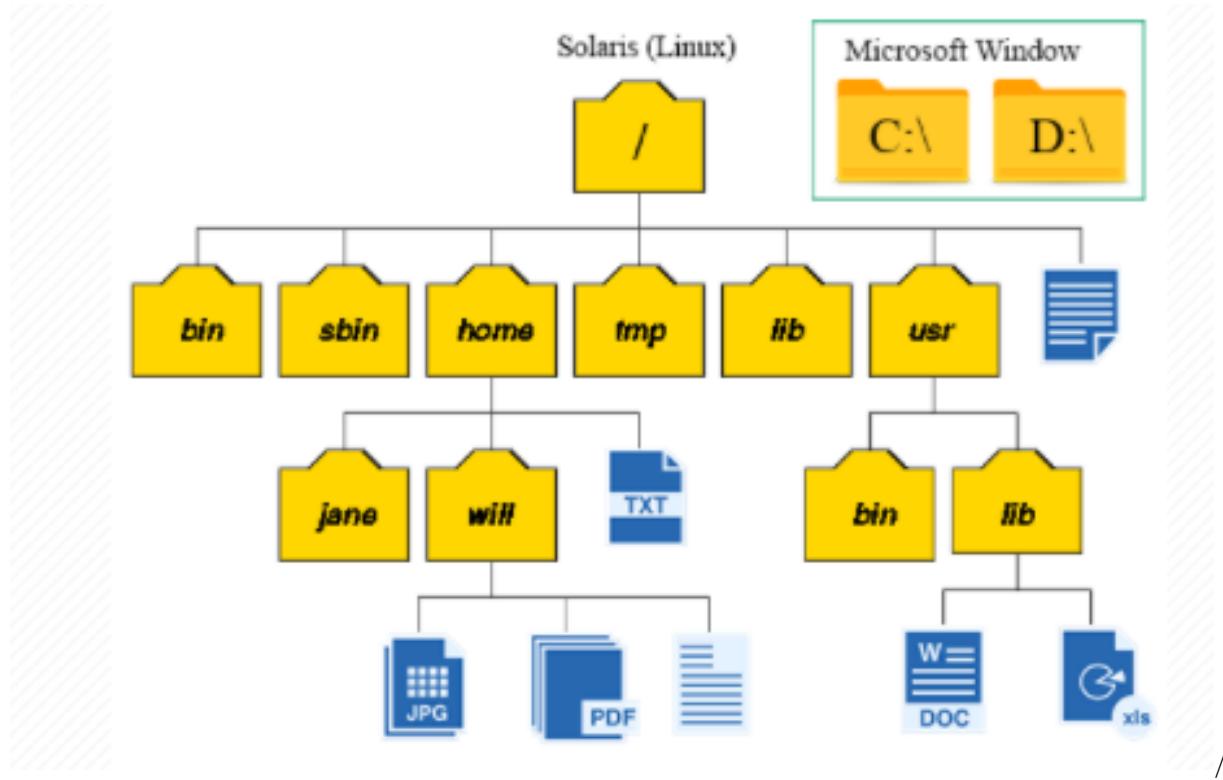


- **Base Component:** Là một **Interface** hoặc **Abstract Class** quy định các phương thức chung cần phải có cho tất cả các thành phần tham gia
- **Leaf:** Là lớp thực hiện cài đặt phương thức của **Component** và không chứa lớp con
- **Composite:** Là lớp có vai trò lưu trữ và thao tác tập hợp các **Leaf**
- **Client:** Là nơi sử dụng **Base Component** để làm việc với các đối tượng trong **Composition**

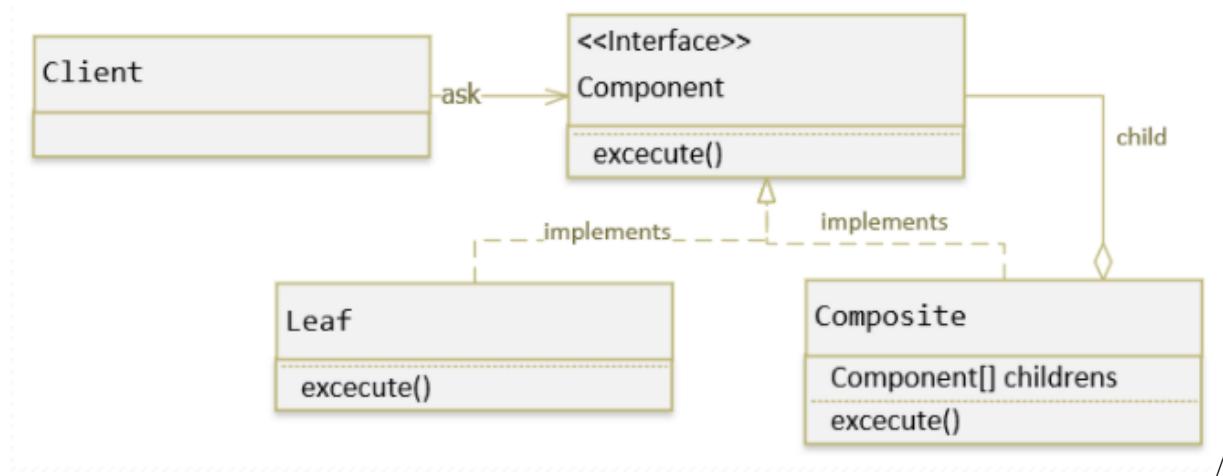
Lưu ý: Một **composite** chứa các **component** vì **component** có thể là **composite** hoặc **leaf**. Nghe vẻ có mùi đệ quy ở đây nhỉ? Có vẻ là vậy vì một **composite** chứa tập hợp các nút con, những nút con có thể là **composite** hoặc **leaf**

8.3 Ví dụ

Để hiểu hơn Composite Pattern, xét ví dụ về bài toán quản lý tệp tin:



Một hệ thống tệp tin là hệ thống có cấu trúc cây phân cấp trong đó có các File và Folder. Một Folder có thể chứa nhiều File và Folder bên trong nên ta có thể coi Folder như **Composite** còn File sẽ là **Leaf**. Ta có mô hình cấu trúc ở bên dưới:



Dầu tiên ta định nghĩa cho **component** Interface có các phương thức chung cho cả File và Folder. Để đơn giản, ví dụ này chỉ bao gồm 2 phương thức **showProperty()** và **totalSize()**. Hai phương thức này sẽ lấy ra thông tin của File/Folder và tổng kích thước của chúng

```
public interface FileComponent {  
    void showProperty();  
    long totalSize();  
}
```

Tiếp theo, ta tạo lớp **Leaf** cài đặt các phương thức của **component**, gọi là **FileLeaf**. Sau đó tạo lớp **Composite** chứa tập hợp các **Leaf** và cài đặt các phương thức của lớp **component**, gọi là **FolderComposite**

Lớp **FileLeaf**:

```
public class FileLeaf implements FileComponent {  
  
    private String name;  
    private long size;  
  
    public FileLeaf(String name, long size) {  
        super();  
        this.name = name;  
        this.size = size;  
    }  
  
    @Override  
    public long totalSize() {  
        return size;  
    }  
  
    @Override  
    public void showProperty() {  
        System.out.println("FileLeaf [name=" + name + ", size=" + size + "]");  
    }  
}
```

Lớp **FolderComposite**:

```

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

public class FolderComposite implements FileComponent {

    private List<FileComponent> files = new ArrayList<>();

    public FolderComposite(List<FileComponent> files) {
        this.files = files;
    }

    @Override
    public void showProperty() {
        for (FileComponent file : files) {
            file.showProperty();
        }
    }

    @Override
    public long totalSize() {
        long total = 0;
        for (FileComponent file : files) {
            total += file.totalSize();
        }
        return total;
    }
}

```

Cuối cùng, ta hoàn thiện lớp **Client** để chạy chương trình:

```

import java.util.Arrays;
import java.util.List;

public class Client {

    public static void main(String[] args) {
        FileComponent file1 = new FileLeaf("file 1", 10);
        FileComponent file2 = new FileLeaf("file 2", 5);
        FileComponent file3 = new FileLeaf("file 3", 12);

        List<FileComponent> files = Arrays.asList(file1, file2, file3);
        FileComponent folder = new FolderComposite(files);
        folder.showProperty();
        System.out.println("Total Size: " + folder.totalSize());
    }
}

```

Output của chương trình:

```
FileLeaf [name=file 1, size=10]
FileLeaf [name=file 2, size=5]
FileLeaf [name=file 3, size=12]
Total Size: 27
```

8.4 Thực tế

Composite Pattern có thể được tìm thấy tại hầu hết các hệ thống lập trình theo hướng đối tượng. Lớp Viewer gốc của Smalltalk Model/View/Controller [KP88] là **Composite**, và gần như bộ giao diện người dùng và framework. Viewer thời điểm đó vừa là lớp **Component** vừa là lớp **Composite**. Bản 4.0 của Smalltalk-80 sửa lại Model/View/Controller với một lớp VisualComponent có những lớp con là View và CompositeView

RTL Smalltalk compiler framework sử dụng Composite Pattern một cách chuyên sâu. RTLExpression là một lớp **Component** cho cây phân tích cú pháp. Nó có những lớp con như BinaryExpression chứa những đối tượng RTLExpression. Những lớp này định nghĩa một cấu trúc composite cho cây phân tích cú pháp. RegisterTransfer là một lớp **Component** cho chương trình trung gian Single Static Assignment (SSA). Những lớp con Leaf của RegisterTransfer xác định các nhiệm vụ tinh khác nhau

Một ví dụ khác cho pattern này xảy ra trong lĩnh vực tài chính, nơi mà có một danh mục đầu tư tổng hợp các tài sản riêng lẻ. Ta có thể hỗ trợ các tổng hợp phức tạp của tài sản bằng cách cài đặt "danh mục đầu tư" như một lớp **Composite** phù hợp với giao diện của một tài sản cá nhân [BE93].

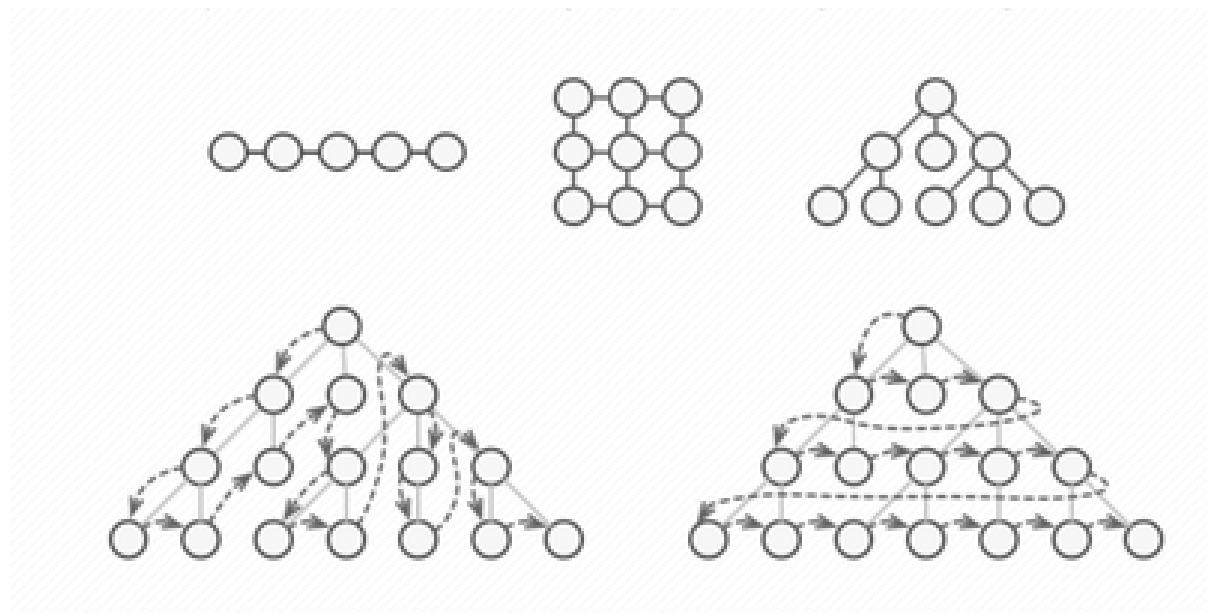
9. Iterator

9.1 Giới thiệu

9.1.1 Đặt vấn đề

Trong khi phát triển các ứng dụng, chúng ta làm việc với nhiều loại tập hợp như: cấu trúc cây, mảng, tập hợp, bảng băm, ngăn xếp, hàng đợi, ... Cách thức mà tập hợp này lưu trữ đối tượng của nó rất khác nhau, và nếu bạn muốn truy cập dữ liệu của những đối tượng này, bạn phải học những kỹ thuật khác nhau cho từng loại tập hợp. Khi đó, mẫu Iterator là một giải pháp tốt.

Chúng ta có thể sử dụng một interface được xác định phương thức cụ thể để truy cập tới từng phần tử của tập hợp. Sử dụng những phương thức này, chúng ta có thể truy xuất tới các phần tử trong tập hợp theo cách dễ dàng nhất.



Hình 9-1: Các loại tập hợp

9.1.2 Mục đích sử dụng

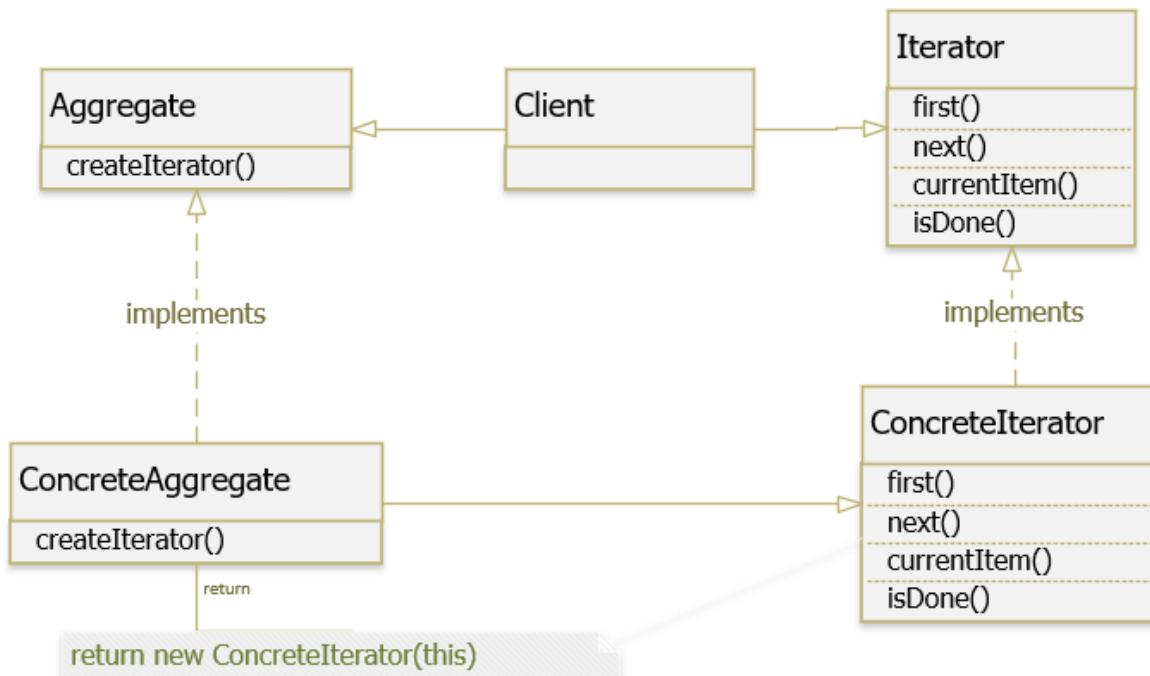
- Dảm bảo nguyên tắc Single responsibility principle (SRP) : chúng ta có thể tách phần cài đặt các phương thức của tập hợp và phần duyệt qua các phần tử (iterator) theo từng class riêng lẻ.
- Dảm bảo nguyên tắc Open/Closed Principle (OCP) : chúng ta có thể implement các loại collection mới và iterator mới, sau đó chuyển chúng vào code hiện có mà không vi phạm bất cứ nguyên tắc gì.
- Chúng ta có thể truy cập song song trên cùng một tập hợp vì mỗi đối tượng iterator có chứa trạng thái riêng của nó.

9.2 Định nghĩa và mô hình cấu trúc

9.2.1 Định nghĩa

Iterator Pattern là một trong những Pattern thuộc nhóm hành vi(Behavioral). 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.”

9.2.2 Mô hình cấu trúc



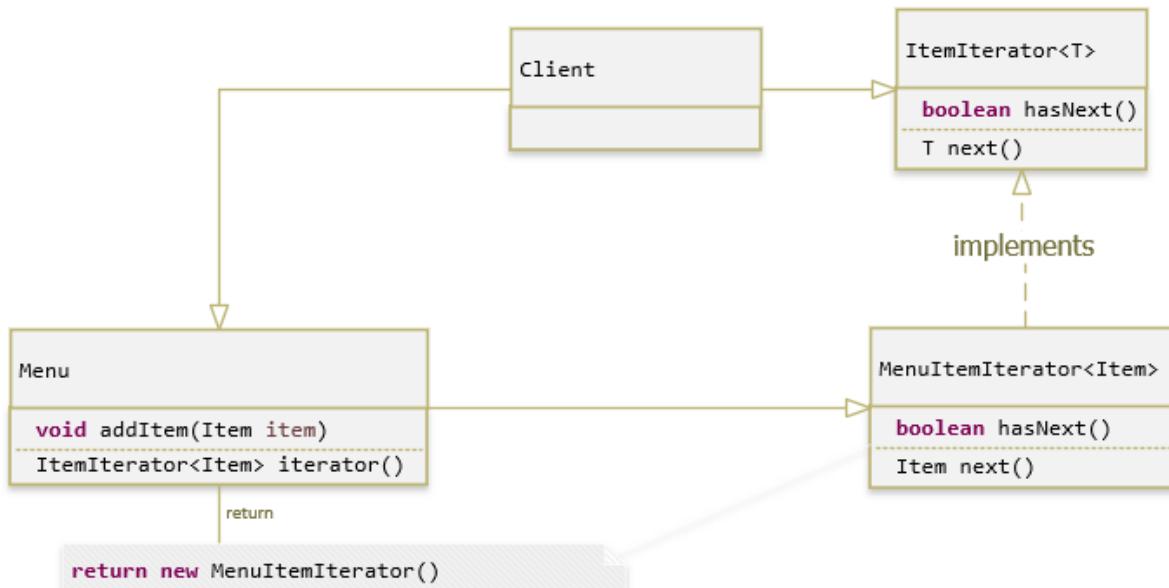
Hình 9-2: Mô hình cấu trúc

- **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

9.3 Cách cài đặt

Cài đặt Iterator để duyệt qua các item trong một menu.

Link code cài đặt: <https://github.com/nanhus/OOP-DesginPatten/tree/master/source/iterator>
 Dưới đây là code minh họa cài đặt:



Hình 9-3: Mô hình cấu trúc

```

1 package source.iterator;
2
3 public class Item {
4     private String title;
5     private String url;
6
7     public Item(String title, String url) {
8         super();
9         this.title = title;
10        this.url = url;
11    }
12
13    @Override
14    public String toString() {
15        return "Item [title=" + title + ", url=" + url + "]";
16    }
17}
18
  
```

Hình 9-4: Item Class

```
● 1 package source.iterator;
2
3   public interface ItemIterator<T> {
4
5     boolean hasNext();
6
7     T next();
8   }
9
```

Hình 9-5: Item Iterator Interface

```
1 package source.iterator;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class Menu {
7   private List<Item> menuItems = new ArrayList<>();
8
9   public void addItem(Item item) {
10    menuItems.add(item);
11  }
12
13   public ItemIterator<Item> iterator() {
14    return new MenuItemIterator();
15  }
16
17   class MenuItemIterator implements ItemIterator<Item> {
18    private int currentIndex = 0;
19
20    @Override
21    public boolean hasNext() {
22      return currentIndex < menuItems.size();
23    }
24
25    @Override
26    public Item next() {
27      return menuItems.get(currentIndex++);
28    }
29  }
30 }
31
```

Hình 9-6: Menu Class

```

source > iterator > Client.java > Client > main(String[])
1 package source.iterator;
2
3 public class Client {
4     Run | Debug
5     public static void main(String[] args) {
6         Menu menu = new Menu();
7         menu.addItem(new Item("Home", "/home"));
8         menu.addItem(new Item("Java", "/java"));
9         menu.addItem(new Item("Yii Frame", "/yii-frame"));
10
11         ItemIterator<Item> iterator = menu.iterator();
12         while (iterator.hasNext()) {
13             Item item = iterator.next();
14             System.out.println(item);
15         }
16     }

```

Hình 9-7: Client Class

```

Item [title=Home, url=/home]
Item [title=Java, url=/java]
Item [title=Yii Frame, url=/yii-frame]
PS C:\Users\pc\Desktop\Final OOP>

```

Hình 9-8: Output

9.4 Ví dụ thực tế

- Truy xuất nhiều loại tập hợp
- Ứng dụng trong chuyển kênh của vô tuyến truyền hình
- Áp dụng trong thư viện java
 - java.util.Iterator
 - java.utilEnumeration

10. Builder Pattern

10.1 Đặt vấn đề

Ta chắc hẳn đã biết về lợi ích to lớn mà **immutability** và immutable instances trong ứng dụng. Chưa được thuyết phục lắm? Vậy hãy nhìn lại về lớp **String** trong Java tại [đây](#)

Hãy bàn luận một chút về vấn đề trong ứng dụng của chúng ta. Trong bất kì module quản lý người dùng nào, thực thể chính luôn là người dùng. Một khi người dùng được tạo ra thì ta không muốn thay đổi trạng thái của nó. Điều này có vẻ mâu thuẫn nhỉ...

Bây giờ, hãy giả sử, đối tượng người dùng của ta có 5 thuộc tính: Họ, tên, tuổi, số điện thoại và địa chỉ

Thông thường, khi ta thiết kế lớp người dùng, ta phải truyền tất cả tham số vào hàm dựng như sau:

```
public User (String firstName, String lastName, int age, String phone, String address){  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.age = age;  
    this.phone = phone;  
    this.address = address;  
}
```

Vậy chuyện gì xảy ra nếu ta không bắt buộc 3 trường tuổi, số điện thoại và địa chỉ? Khi đó sẽ xảy ra vấn đề. Ta cần nhiều hàm dựng hơn và dẫn đến tình trạng **telescoping constructors**. Nói cách khác, cách thiết kế này sẽ mất tính thẩm mĩ và rất là rối

```
public User (String firstName, String lastName, int age, String phone){ ... }  
public User (String firstName, String lastName, String phone, String address){ ... }  
public User (String firstName, String lastName, int age){ ... }  
public User (String firstName, String lastName){ ... }
```

Builder Pattern sinh ra để khắc phục vấn đề này. Nó giúp ta có thể tạo lớp immutable kiểm soát một đối tượng phức tạp có số lượng thuộc tính đủ lớn

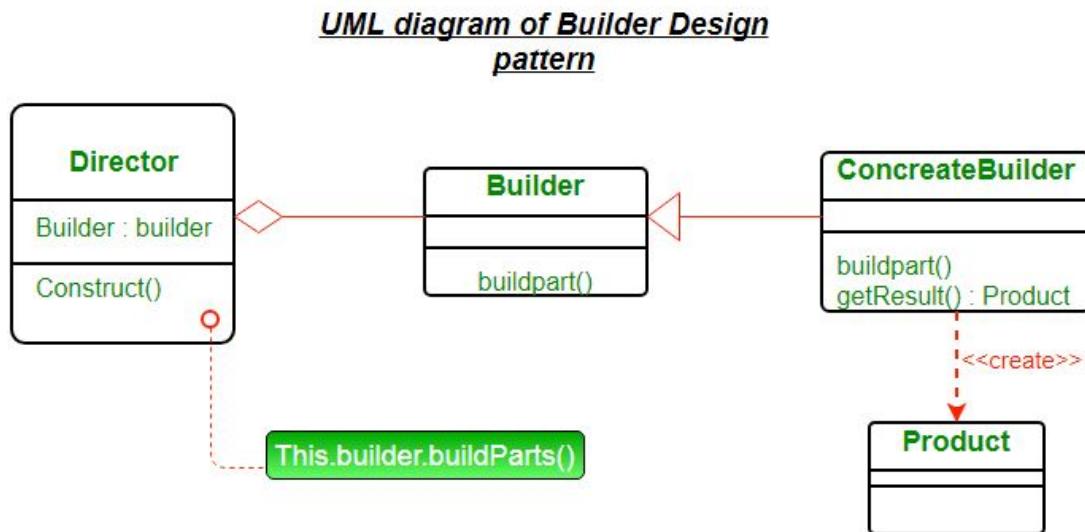
10.2 Định nghĩa và Mô hình cấu trúc

10.2.1 Định nghĩa

Builder Pattern là một pattern tách biệt cách dựng của một đối tượng phức tạp khỏi sản phẩm cuối cùng. Bằng cách này, cùng một cách dựng có thể tạo ra nhiều sản phẩm khác nhau. Cũng như **Factory Pattern**, **Builder Pattern** thuộc nhóm **Creational Pattern** (Nhóm khởi tạo)

10.3 Mô hình cấu trúc

Mô hình cấu trúc của Builder Pattern được mô tả như sau: Trong đó:



Hình 10-1: Mô hình cấu trúc Builder Pattern

- **Product:** Là lớp định nghĩa đối tượng phức tạp để được sinh ra từ builder pattern
 - **Builder:** Là lớp trừu tượng định nghĩa các bước cần thiết để tạo sản phẩm đúng cách
 - **Concrete Builder:** Là những lớp được kế thừa từ **Builder**. Những lớp này chứa chức năng để tạo một đối tượng phức tạp cụ thể
 - **Director:** Là lớp đảm nhận thuật toán sinh ra sản phẩm cuối cùng

10.4 Cài đặt

Áp dụng **Builder Pattern** để giải quyết vấn đề ở mục 1. Về ý tưởng, ta lồng thêm lớp **UserBuilder** vào trong lớp **User** để hỗ trợ việc tạo đối tượng mà không mất đi tính immutability Dưới đây là code minh họa:

Lớp User:

```
public class User
{
    //All final attributes
    private final String firstName; // required
    private final String lastName; // required
    private final int age; // optional
    private final String phone; // optional
    private final String address; // optional

    private User(UserBuilder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.age = builder.age;
        this.phone = builder.phone;
        this.address = builder.address;
    }

    //All getter, and NO setter to prove immutability
    public String getFirstName() {
        return firstName;
    }
    public String getLastname() {
        return lastName;
    }
    public int getAge() {
        return age;
    }
    public String getPhone() {
        return phone;
    }
    public String getAddress() {
        return address;
    }

    @Override
    public String toString() {
        return "User: "+this.firstName+", "+this.lastName+", "+this.age+", "+this.phone+", "+this.address
    }
}
```

Lớp UserBuilder trong lớp User

```
public static class UserBuilder
{
    private final String firstName;
    private final String lastName;
    private int age;
    private String phone;
    private String address;

    public UserBuilder(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public UserBuilder age(int age) {
        this.age = age;
        return this;
    }
    public UserBuilder phone(String phone) {
        this.phone = phone;
        return this;
    }
    public UserBuilder address(String address) {
        this.address = address;
        return this;
    }
    //Return the finally constructed User object
    public User build() {
        User user = new User(this);
        validateUserObject(user);
        return user;
    }
    private void validateUserObject(User user) {
        //Do some basic validations to check
        //if user object does not break any assumption of system
    }
}
```

Lớp Client để xây dựng các đối tượng bằng **UserBuilder**:

```
public static void main(String[] args)
{
    User user1 = new User.UserBuilder("Lokesh", "Gupta")
        .age(30)
        .phone("1234567")
        .address("Fake address 1234")
        .build();

    System.out.println(user1);

    User user2 = new User.UserBuilder("Jack", "Reacher")
        .age(40)
        .phone("5655")
        //no address
        .build();

    System.out.println(user2);

    User user3 = new User.UserBuilder("Super", "Man")
        //No age
        //No phone
        //no address
        .build();

    System.out.println(user3);
}
```

10.5 Thực tế

RTF Converter (Rich Text Format Converter - bộ chuyển đổi văn bản sang nhiều định dạng khác nhau) có khôi xây dựng text sử dụng builder để lưu trữ text trong định dạng RTF

Builder là một pattern phổ biến trong Smalltalk-80[Par90]:

- Lớp Parser trong hệ thống con biên dịch là **Director**. Nó lấy đối tượng ProgramNodeBuilder làm đối số. Một đối tượng Parser thông báo đối tượng ProgramNodeBuilder mỗi lần nó nhận diện một cấu trúc cú pháp. Khi nào trình phân tích cú pháp được thực hiện, nó yêu cầu builder cho cây phân tích cú pháp mà nó đã xây dựng và trả về cho Client
- ClassBuilder là một builder mà các lớp sử dụng để tạo các lớp con cho chính chúng. Trong trường hợp này một lớp vừa là **Director** vừa là **Product**
- ByteCodeStream là một builder tạo ra một phương thức đã được biên dịch như một mảng byte. ByteCodeStream là một cách dùng khác của **Builder Pattern** vì đối tượng phức tạp nó tạo được mã hóa thành một mảng byte, không như một đối tượng Smalltalk thông thường. Nhưng interface của ByteCodeStream là một trường hợp điển hình của builder và sẽ dễ dàng thay thế ByteCodeStream bằng một lớp khác đại diện chương trình như một đối tượng composite

11. Decorator Pattern

11.1 Giới thiệu

11.1.1 Đặt vấn đề

Một trong những khía cạnh quan trọng nhất trong quá trình phát triển một ứng dụng mà các lập trình viên phải đối đầu là sự thay đổi. Khi muốn thêm hoặc loại bỏ một tính năng của một đối tượng, điều đầu tiên chúng ta nghĩ đến là thừa kế (extends). Tuy nhiên, thừa kế không khả thi vì nó là static, chúng ta không thể thêm các lớp con mới vào một chương trình khi nó đã được biên dịch và thực thi.

Để giải quyết vấn đề này, chúng ta có thể sử dụng Decorator Pattern. Mẫu thiết kế này sẽ linh động thay đổi tính chất (functionality) đã có trong một đối tượng khi chương trình đang chạy (runtime) mà không ảnh hưởng đến các tính chất đã tồn tại của các đối tượng khác.

11.1.2 Mục đích sử dụng

- 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).

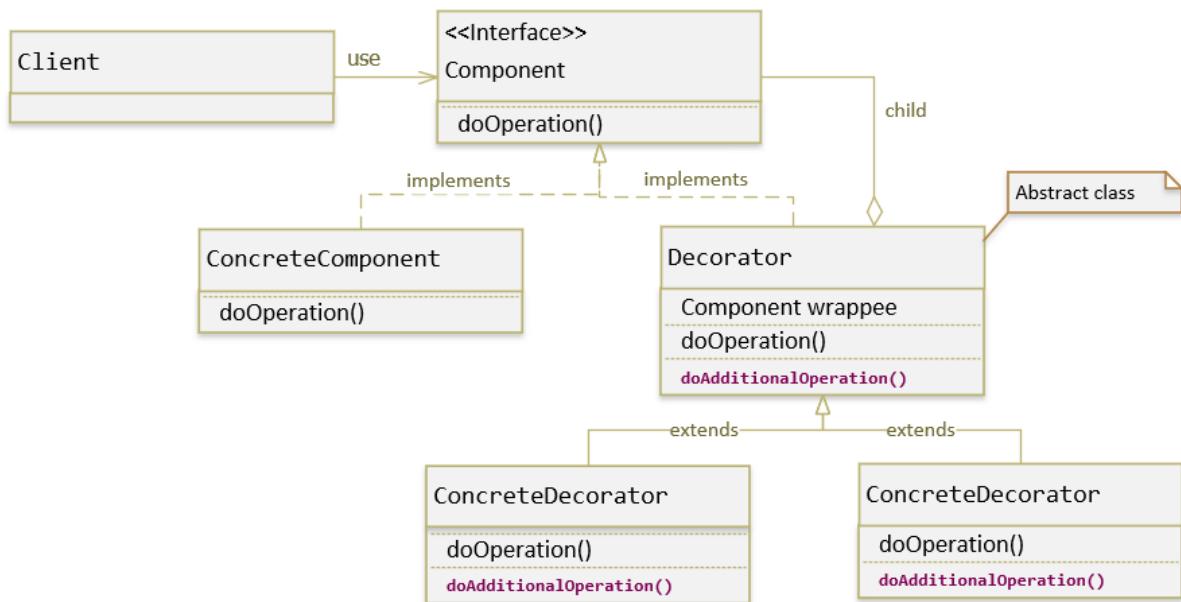
11.2 Định nghĩa và mô hình cấu trúc

11.2.1 Định nghĩa

Decorator pattern là một trong những Pattern thuộc nhóm cấu trúc(Structural). Nó cho phép người dùng thêm chức năng mới vào đối tượng hiện tại mà không 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.

11.2.2 Mô hình cấu trúc



Hình 11-1: Mô hình cấu trúc

- **Component**: là một interface quy định các phương thức chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.
- **ConcreteComponent**: là lớp thực hiện(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.

11.3 Cách cài đặt

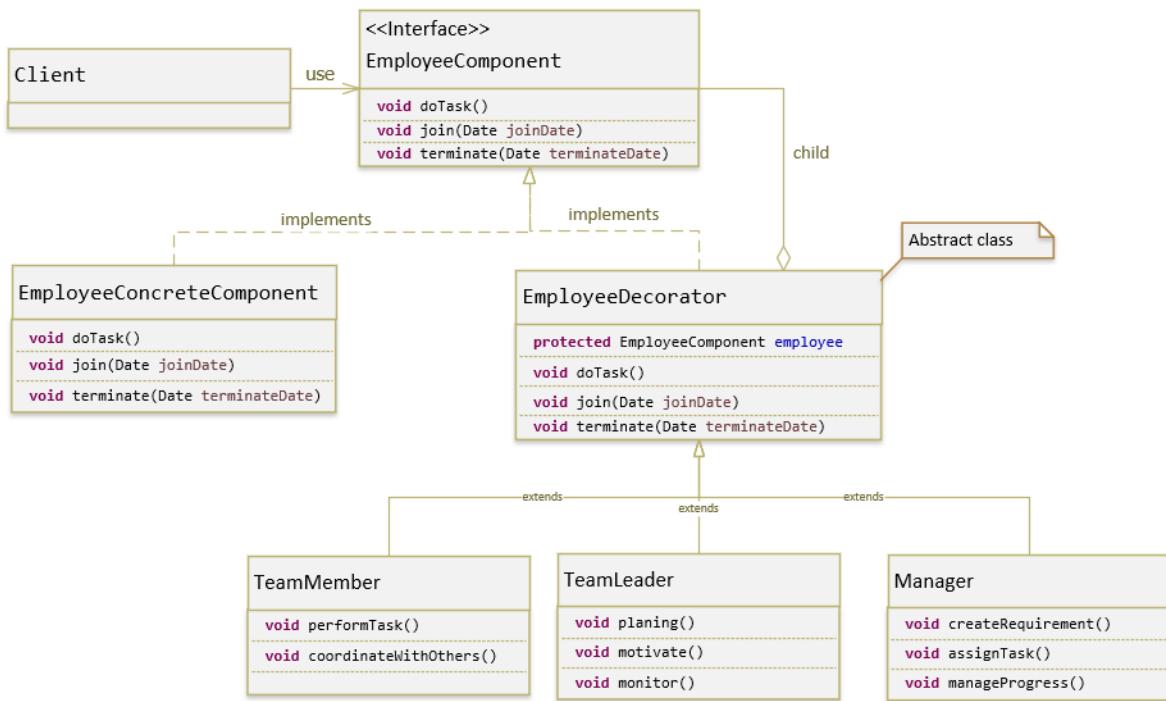
Ví dụ cài đặt cho một hệ thống quản lý dự án, nơi nhân viên đang làm việc với các vai trò khác nhau, chẳng hạn như thành viên nhóm (team member), trưởng nhóm (team lead) và người quản lý (manager). Một thành viên trong nhóm chịu trách nhiệm thực hiện các nhiệm vụ được giao và phối hợp với các thành viên khác để hoàn thành nhiệm vụ nhóm. Mặt khác, một trưởng nhóm phải quản lý và cộng tác với các thành viên trong nhóm của mình và lập kế hoạch nhiệm vụ của họ. Tương tự như vậy, một người quản lý có thêm một số trách nhiệm đối với một trưởng nhóm như quản lý yêu cầu dự án, tiến độ, phân công công việc.

Sau đây là các thành phần tham gia vào hệ thống và hành vi của chúng:

- **Employee**: thực hiện công việc (doTask), tham gia vào dự án (join), rời khỏi dự án (terminate).
- **Team member**: báo cáo task được giao (report task), cộng tác với các thành viên khác (coordinate with others).

- Team lead: lên kế hoạch (planning), hỗ trợ các thành viên phát triển (motivate), theo dõi chất lượng công việc và thời gian (monitor). Manager: tạo các yêu cầu dự án (create requirement), giao nhiệm vụ cho thành viên (assign task), quản lý tiến độ dự án (progress management).

Tiếp theo là mô hình cấu trúc áp dụng Decorator Pattern



Hình 11-2: Mô hình cấu trúc quản lí dự án

Dưới đây là code của cài đặt.

Link code cài đặt: <https://github.com/nanhust/OOP-DesginPatten/tree/master/source/decorator>

```
source > decorator > EmployeeComponent.java > EmployeeComponent > showBasicInformation()
1 package source.decorator;
2
3 import java.text.DateFormat;
4 import java.text.SimpleDateFormat;
5 import java.util.Calendar;
6 import java.util.Date;
7
8 public interface EmployeeComponent {
9
10     abstract String getName();
11
12     abstract void doTask();
13
14     abstract void join(Date joinDate);
15
16     abstract void terminate(Date terminateDate);
17
18     default String formatDate(Date theDate) {
19         DateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
20         return sdf.format(theDate);
21     }
22
23     default void showBasicInformation() {
24         System.out.println("-----");
25         System.out.println("The information of " + getName());
26
27         join(Calendar.getInstance().getTime());
28         Calendar terminateDate = Calendar.getInstance();
29         terminateDate.add(Calendar.MONTH, 2);
30         terminate(terminateDate.getTime());
31     }
32 }
```

Hình 11-3: Employee Component Interface

```
source > decorator > EmployeeDecorator.java > EmployeeDecorator
1 package source.decorator;
2
3 import java.util.Date;
4
5 public abstract class EmployeeDecorator implements EmployeeComponent {
6
7     protected EmployeeComponent employee;
8
9     protected EmployeeDecorator(EmployeeComponent employee) {
10         this.employee = employee;
11     }
12
13     @Override
14     public String getName() {
15         return employee.getName();
16     }
17
18     @Override
19     public void join(Date joinDate) {
20         employee.join(joinDate);
21     }
22
23     @Override
24     public void terminate(Date terminateDate) {
25         employee.terminate(terminateDate);
26     }
27 }
```

Hình 11-4: Employee Decorator Class

```
source > decorator > EmployeeConcreteComponent.java > EmployeeConcreteComponent
1 package source.decorator;
2
3 import java.util.Date;
4
5 public class EmployeeConcreteComponent implements EmployeeComponent {
6     private String name;
7
8     public EmployeeConcreteComponent (String name) {
9         this.name = name;
10    }
11
12    @Override
13    public String getName() {
14        return name;
15    }
16
17    @Override
18    public void join(Date joinDate) {
19        System.out.println(this.getName() + " joined on " + formatDate(joinDate));
20    }
21
22    @Override
23    public void terminate(Date terminateDate) {
24        System.out.println(this.getName() + " terminated on " + formatDate(terminateDate));
25    }
26
27    @Override
28    public void doTask() {
29        // Unassigned task
30    }
31 }
```

Hình 11-5: Employee Concrete Component Class

```

source > decorator > TeamMember.java > TeamMember
1 package source.decorator;
2
3 public class TeamMember extends EmployeeDecorator {
4
5     protected TeamMember(EmployeeComponent employee) {
6         super(employee);
7     }
8
9     public void reportTask() {
10        System.out.println(this.employee.getName() + " is reporting his assigned tasks.");
11    }
12
13    public void coordinateWithOthers() {
14        System.out.println(this.employee.getName() + " is coordinating with other members of his team.");
15    }
16
17    @Override
18    public void doTask() {
19        employee.doTask();
20        reportTask();
21        coordinateWithOthers();
22    }
23}
24

```

Hình 11-6: Team Member Class

```

source > decorator > TeamLeader.java > TeamLeader
1 package source.decorator;
2
3 public class TeamLeader extends EmployeeDecorator {
4
5     protected TeamLeader(EmployeeComponent employee) {
6         super(employee);
7     }
8
9     public void planing() {
10        System.out.println(this.employee.getName() + " is planning.");
11    }
12
13    public void motivate() {
14        System.out.println(this.employee.getName() + " is motivating his members.");
15    }
16
17    public void monitor() {
18        System.out.println(this.employee.getName() + " is monitoring his members.");
19    }
20
21    @Override
22    public void doTask() {
23        employee.doTask();
24        planing();
25        motivate();
26        monitor();
27    }
28}
29

```

Hình 11-7: Team Leader Class

```
source > decorator > Manager.java > Manager
1 package source.decorator;
2
3 public class Manager extends EmployeeDecorator {
4
5     protected Manager(EmployeeComponent employee) {
6         super(employee);
7     }
8
9     public void createRequirement() {
10        System.out.println(this.employee.getName() + " is creating requirements.");
11    }
12
13    public void assignTask() {
14        System.out.println(this.employee.getName() + " is assigning tasks.");
15    }
16
17    public void manageProgress() {
18        System.out.println(this.employee.getName() + " is managing the progress.");
19    }
20
21    @Override
22    public void doTask() {
23        employee.doTask();
24        createRequirement();
25        assignTask();
26        manageProgress();
27    }
28}
29
```

Hình 11-8: Manager Class

```
NORMAL EMPLOYEE:  
-----  
The information of Nguyen_NA  
Nguyen_NA joined on 24/01/2022  
Nguyen_NA terminated on 24/03/2022  
  
TEAM LEADER:  
-----  
The information of Nguyen_NA  
Nguyen_NA joined on 24/01/2022  
Nguyen_NA terminated on 24/03/2022  
Nguyen_NA is planing.  
Nguyen_NA is motivating his members.  
Nguyen_NA is monitoring his members.  
  
MANAGER:  
-----  
The information of Nguyen_NA  
Nguyen_NA joined on 24/01/2022  
Nguyen_NA terminated on 24/03/2022  
Nguyen_NA is create requirements.  
Nguyen_NA is assigning tasks.  
Nguyen_NA is managing the progress.  
  
TEAM LEADER AND MANAGER:  
-----  
The information of Nguyen_NA  
Nguyen_NA joined on 24/01/2022  
Nguyen_NA terminated on 24/03/2022  
Nguyen_NA is planing.  
Nguyen_NA is motivating his members.  
Nguyen_NA is monitoring his members.  
Nguyen_NA is create requirements.  
Nguyen_NA is assigning tasks.  
Nguyen_NA is managing the progress.  
PS C:\Users\pc\Desktop\Final OOP> █
```

Hình 11-9: Output

11.4 Ví dụ thực tế

- Tạo chức các API
- Quản lý nhân viên, hàng hóa
- Sử dụng trong lập trình đa luồng
- Decorator có trách nhiệm bổ sung vào một đối tượng một cách động. Ví dụ những đồ trang trí được thêm vào cây thông như đèn, vòng hoa, kẹo, đồ trang trí bằng thủy tinh, v.v.,

12. Kết luận

Design Pattern là một kỹ thuật trong lập trình hướng đối tượng, nó rất quan trọng và mọi lập trình viên muốn giỏi đều phải biết.

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

Design Patterns không phải là ngôn ngữ cụ thể nào cả. Nó có thể thực hiện được ở phần lớn các ngôn ngữ lập trình, chẳng hạn như Java, C#, thậm chí là Javascript hay bất kỳ ngôn ngữ lập trình nào khác.

Mỗi pattern mô tả một vấn đề xảy ra lặp đi lặp lại, và trình bày trọng tâm của giải pháp cho vấn đề đó, theo cách mà bạn có thể dùng đi dùng lại hàng triệu lần mà không cần phải suy nghĩ.

— Christopher Alexander —

