ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA TOÁN - CƠ - TIN HỌC

BÁO CÁO HỌC PHẦN
CÁC THÀNH PHẦN PHẦN MỀM

Đề tài: Design Patterns

Nhóm sinh viên:

Nguyễn Ngọc Tỉnh - 19000297

Phùng Thị Thu An - 19000241

Lớp: K64A2 - Toán Tin

HÀ NỘI - 2022

Lời cảm ơn

Đầu tiên, nhóm chúng em xin gửi lời cảm ơn chân thành đến khoa Toán-Cơ-Tin, trường Đại học Khoa học Tự nhiên đã tạo điều kiện thuận lợi cho chúng em học tập và hoàn thành học phần.

Đặc biệt, chúng em xin cảm ơn thầy Quản Thái Hà, người đã giúp đỡ và hướng dẫn tận tình chúng em suốt thời gian học tập trong tình hình khó khăn của bệnh dịch vừa qua, bên cạnh đó cũng cung cấp cho chúng em những kiến thức để tiếp cận, phân tích và giải quyết những vấn đề. Nhờ đó mà chúng em hoàn thành được công việc của mình một cách tốt hơn. Chúng em cũng xin cảm ơn những người bạn đã tận tình chỉ bảo, giúp đỡ trong quá trình hoàn thành học phần.

Chúng em đã vận dụng tối đa những kiến thức mà mình có được trong kì học vừa qua để hoàn thành bài báo cáo của mình. Nhưng do kiến thức vẫn còn hạn chế và chưa có nhiều kinh nghiệm nên khó có thể tránh được những thiếu sót trong quá trình thực hiện. Chúng em mong nhận được những phê bình, góp ý của Thầy để đề tài của nhóm chúng em được hoàn thiện hơn.

Chúng em xin chân thành cảm ơn Thầy!

Nguyễn Ngọc Tỉnh
Phùng Thị Thu An

Lời nói đầu

Trong phát triển phần mềm, bất kể chúng ta là ai, làm việc ở đâu, sử dụng ngôn ngữ lập trình gì hay xây dựng ứng dụng gì thì **sự thay đổi** luôn luôn hiện hữu bên cạnh chúng ta. Khi có một yêu cầu thay đổi phần mềm mới xuất hiện, nếu như chưa có kinh nghiệm chúng ta sẽ cố gắng viết ra các đoạn code mới đáp ứng được sự thay đổi này một cách tạm thời mà không hề quan tâm đến tính tái sử dụng, tính mở rộng và tính bảo trì của đoạn code đó trong trường hợp có yêu cầu thay đổi mới xuất hiện. Chính thói quen không tốt này dẫn đến việc chúng ta phải dành phần lớn thời gian cho giai đoạn bảo trì sau đó. Trường hợp xấu nhất là nếu như thiết kế hiện tại không thể đáp ứng được bất kỳ một yêu cầu thay đổi nào đồng nghĩa với việc chúng ta phải xóa bỏ toàn bộ phần mềm hiện tại và tiến hành xây dựng lại mọi thứ từ bước đầu tiên. Đây chính là lý do mà **Design Patterns** xuất hiện.

Design patterns sẽ không "đi thẳng" vào code, mà trước hết chúng sẽ tác động trực tiếp vào suy nghĩ, tư duy của chúng ta. Khi đã có một kiến thức đủ sâu sắc về patterns thì chúng ta có thể áp dụng chúng vào các thiết kế mới và cải thiện code cũ của mình khi phát hiện chúng còn tồn đọng nhiều hạn chế.

Mục lục

Tổng quan về Design Patterns

1.1 Design Patterns là gì?

Design patterns là một kỹ thuật trong lập trình hướng đối tượng, được các nhà nghiên cứu đúc kết và tạo ra các mẫu thiết kế chuẩn. Và design patterns không phải là một ngôn ngữ lập trình cụ thể nào cả, nó có thể sử dụng được trong hầu hết các ngôn lập trình có hỗ trợ OOP.

Trong công nghệ phần mềm, một design pattern(mẫu thiết kế) là một giải pháp tổng thể cho các vấn đề chung trong thiết kế phần mềm. Chúng giống như một bản thiết kế được tạo sẵn mà chúng ta có thể tùy chỉnh để giải quyết vấn đề thiết kế lặp đi lặp lại trong code của mình. Một design pattern không phải là một thiết kế hoàn thiện để có thể được chuyển đổi trực tiếp thành code, nó chỉ là một mô tả hay là sườn (template) mô tả cách giải quyết một vấn đề mà có thể được dùng trong nhiều tình huống khác nhau. Các mẫu thiết kế hướng đối tượng thường cho thấy mối quan hệ và sự tương tác giữa các lớp hay các đối tượng, mà không cần chỉ rõ các lớp hay đối tượng của từng ứng dụng cụ thể. Giữa giải thuật và design patterns có thể có những sự mập mờ, nhưng thực tế các giải thuật không được xem là design patterns, vì chúng giải quyết các vấn đề về tính toán hơn là các vấn đề về thiết kế.

Chúng ta cũng có thể hiểu một cách đơn giản rằng, thuật toán như một công thức, nó có các bước rõ ràng để giải quyết vấn đề. Mặt khác, một design pattern giống như một bản thiết kế, chúng ta có thể xem kết quả và các tính năng của nó nhưng thứ tự hay quy tắc thực hiện là phụ thuộc vào mình.

1.2 Mô tả Design Patterns

- **Tên**
  Tên của patterns thể hiện bản chất của chúng một cách ngắn gọn.

- **Mục đích**
  Là câu trả lời ngắn gọn của các câu hỏi như: Design Pattern thực hiện việc gì? Mục đích của nó là gì? Nó giải quyết vấn đề thiết kế cụ thể nào?

- **Hướng giải quyết**
  Cách tiếp cận để giải quyết vấn đề.

- **Khả năng áp dụng**
  Tình huống mà design pattern có thể áp dụng? Ví dụ về trường hợp thiết kế không tốt mà pattern có thể cải thiện? Cách để nhận biết tình huống?

- **Cấu trúc**
  Sử dụng biểu đồ lớp(UML).

- **Thành phần tham gia**
  Các lớp và(hoặc) các đối tượng tham gia vào design patterns và nhiệm vụ của chúng.

- **Hệ quả**
  Ảnh hưởng của pattern đối với hệ thống.

- **Cài đặt**
  Các ví dụ để cài đặt pattern.

- **Code minh họa**
  Các đoạn code minh họa cách cài đặt pattern.

- **Patterns liên quan**
  Những design patterns nào có liên quan đến pattern này? Sự khác biệt giữa chúng? Với patterns nào khác thì nên sử dụng pattern này?

1.3 Lợi ích của Design Patterns

- Giúp tái sử dụng code và dễ dàng mở rộng.
- Có thêm vốn từ vựng về thiết kế phần mềm.
- Cung cấp kinh nghiệm, không cung cấp code để tái sử dụng.
- Giúp chương trình linh hoạt hơn, dễ kiểm soát và bảo trì hơn.
- Ưu tiên sự ủy quyền, tránh tạo ra sự phân cấp lớn khi kế thừa, giúp giảm độ phức tạp.

- Giúp chúng ta xây dựng những hệ thống có chất lượng thiết kế object-oriented tốt.
- Cung cấp các giải pháp chung cho vấn đề thiết kế từ đó có thể áp dụng vào ứng dụng cụ thể.
- Giúp chúng ta có thể hiểu code của người khác một cách nhanh chóng, từ đó dễ dàng trao đổi với nhau mà không mất quá nhiều thời gian.

1.4 Phân loại Design Patterns

Design Patterns được phân loại dựa theo ý định và mục đích sử dụng, cụ thể chúng được chia làm 3 nhóm: Creational(Mẫu khởi tạo) - Structural (Mẫu cấu trúc)- Behavioral(Mẫu hành vi).

Creational Pattern: những design patterns loại này cung cấp một giải pháp để tạo ra các object và che giấu được logic của việc tạo ra nó, thay vì tạo ra object một cách trực tiếp bằng cách sử dụng method **new**. Điều này giúp cho chương trình trở nên linh hoạt hơn trong việc quyết định object nào cần được tạo ra trong những tình huống được đưa ra.

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural Pattern: những Design pattern loại này liên quan tới class và các thành phần của object. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.

- Adapter
- Bridge
- Composite

- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Pattern: xác định và định nghĩa các mẫu giao tiếp chung giữa các đối tượng, qua đó làm tăng tính linh hoạt trong việc thực hiện giao tiếp của các đối tượng.

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

1.5 Lựa chọn Design Patterns

Với việc có hơn 20 Design Pattern để lựa chọn, điều này khiến chúng ta gặp một chút khó khăn trong việc tìm được pattern để giải quyết một vấn đề cụ thể. Để dễ dàng tìm ra design pattern phù hợp với vấn đề gặp phải, chúng ta có thể tiếp cận một số cách sau:

- **Xem xét cách mà mỗi design pattern giải quyết vấn đề thiết kế**

  Thảo luận, suy nghĩ về cách desgins patterns giải quyết vấn đề giúp chúng ta tìm ra đối tượng thích hợp, xác định mức độ chi tiết của đối tượng, xác định Interface của đối tượng,... từ đó có thể tìm ra design pattern phù hợp.

- **Mục đích của từng pattern**

  Đọc qua mục đích của từng pattern để có thể tìm ra một hoặc một số pattern phù hợp với vấn đề. Chúng ta có thể phân loại theo nhóm design pattern để thu hẹp vùng tìm kiếm.

- **Tìm hiểu về sự tương tác giữa các pattern**

  Thể hiện mối quan hệ giữa các pattern bằng sơ đồ. Điều này hướng chúng ta đến một hoặc một nhóm các pattern phù hợp.

- **Tìm hiểu các pattern có mục đích tương tự**

  Hiểu về sự giống và khác nhau giữa các pattern có mục đích tương tự.

- **Nhìn lại nguyên nhân tại sao phải thiết kế lại**

  Nhìn lại các nguyên nhân tại sao phải thiết kế lại để xem liệu rằng vấn đề gặp phải có liên quan đến một hay nhiều trong số chúng hay không. Từ đó lựa chọn pattern giúp chúng ta tránh khỏi những nguyên nhân, hạn chế đó.

- **Xem xét những gì nên được thay đổi trong thiết kế**

  Việc này ngược lại hoàn toàn so với việc tìm hiểu nguyên nhân tại sao phải thiết kế lại. Thay vì xem xét vì điều gì mà phải thay đổi một thiết kế, hãy xem xét những gì có thể thay đổi mà không cần phải thiết kế lại.

1.6 Cách sử dụng Design Patterns

Sau khi lựa chọn được desgin patterns phù hợp, chúng ta nên tham khảo một số bước tiếp cận sau để có thể áp dụng design patterns một cách hiệu quả:

(i) **Đọc qua patterns một số lần để có cái nhìn tổng quan.** Đặc biệt chú ý đến khả năng áp dụng và kết quả để dám chắc rằng patterns phù hợp với vấn đề gặp phải.

(ii) **Xem kĩ lại các phần cấu trúc, thành phần tham gia.** Chắc chắn rằng các lớp và đối tượng trong patterns có sự liên quan đến nhau.

(iii) **Xem code mẫu để tìm ví dụ cụ thể.** Thông qua việc này, ta sẽ hiểu được cách triển khai patterns.

(iv) **Đặt tên các thành phần tham gia phải có ý nghĩa trong ngữ cảnh ứng dụng.** Điều này giúp patterns rõ ràng hơn trong quá trình triển khai và phát triển.

(v) **Xác định các lớp.** Khai báo các interface, thiết lập các mối quan hệ kế thừa, xác định các biến đại diện cho dữ liệu và các tham chiếu đối tượng. Xác định các lớp trong chương trình sẽ bị ảnh hưởng bởi patterns và sửa đổi chúng cho phù hợp.

(vi) **Xác định tên dành riêng cho các hành động trong patterns.**

(vii) **Triển khai cài đặt các hoạt động để thực hiện chức năng trong patterns.**

Đây chỉ là một số bước cơ bản để chúng ta làm quen với Design Patterns. Chúng ta sẽ phải phát triển cách làm việc của riêng mình với các design patterns. Không nên sử dụng design patterns một cách bừa bãi vì điều này có thể làm thiết kế thêm phức tạp và ảnh hưởng đến hiệu suất. Tóm lại, một design pattern chỉ nên được áp dụng khi tính linh hoạt mà nó mang lại thực sự cần thiết.

FACTORY METHOD

Factory Method là một mẫu thiết kế khởi tạo(creational), cung cấp một interface để tạo các đối tượng trong một lớp cha nhưng cho phép các lớp con thay đổi kiểu đối tượng sẽ được tạo.

Cụ thể, nhiệm vụ của Factory Method là quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đối tượng một cách linh hoạt hơn. Factory Method đúng nghĩa là một nhà máy và "sản xuất" các đối tượng theo yêu cầu của chúng ta. Trong Factory Method, chúng ta tạo đối tượng mà không để lộ logic tạo đối tượng ở phía người dùng và tham chiếu đến đối tượng được tạo ra bằng cách sử dụng một interface chung. Factory Pattern được sử dụng khi có một lớp cha với nhiều lớp con, dựa trên đầu vào và phải trả về một trong những lớp con đó.

2.1 Đặt vấn đề

Giả sử chúng ta đang tạo ra một ứng dụng quản lí vận tải(logistics), ban đầu ứng dụng chỉ nhằm mục đích phục vụ cho việc vận chuyển hàng hóa với phương tiện là xe tải(Truck). Sau đó, ứng dụng trở nên phổ biến và nhận được các yêu cầu từ các công ty vận tải đường biển để kết hợp thêm dịch vụ vận tải đường biển vào ứng dụng. Tuy nhiên, hầu hết các code hiện tại gắn chặt với phương tiện là xe tải(class Truck). Việc thêm class Ship vào ứng dụng sẽ phải thay đổi toàn bộ code. Hơn nữa, nếu muốn thêm nhiều hình thức vận tải vào ứng dụng có thể chúng ta sẽ cần thực hiện lại tất cả những thay đổi này, đây thực sự là vấn đề lớn đối với chương trình.

2.2 Giải pháp

Factory Method đưa ra một cách để thay thế việc khởi tạo đối tượng trong constructor bởi từ khóa *new* bằng việc gọi một phương thức đặc biệt gọi là factory. Với cách này các đối tượng vẫn được tạo ra bởi từ khóa *new* nhưng nó được gọi ở factory method. Đối tượng trả về từ method này sẽ là các sản phẩm(product) tương ứng theo yêu cầu.

Nhìn qua, có vẻ thay đổi này vô nghĩa, chỉ là chuyển từ việc tạo đối tượng từ constructor sang một factory method. Tuy nhiên, để ý kĩ chúng

ta có thể thấy tại một lớp con đều có thể ghi đè phương thức tạo object, điều này giúp chúng ta tạo ra từng loại đối tượng với từng nhu cầu, nó dễ dàng thêm đối tượng(cụ thể trong vấn đề nêu trên là thêm các loại phương tiện) mà không cần thay đổi code hiện tại.

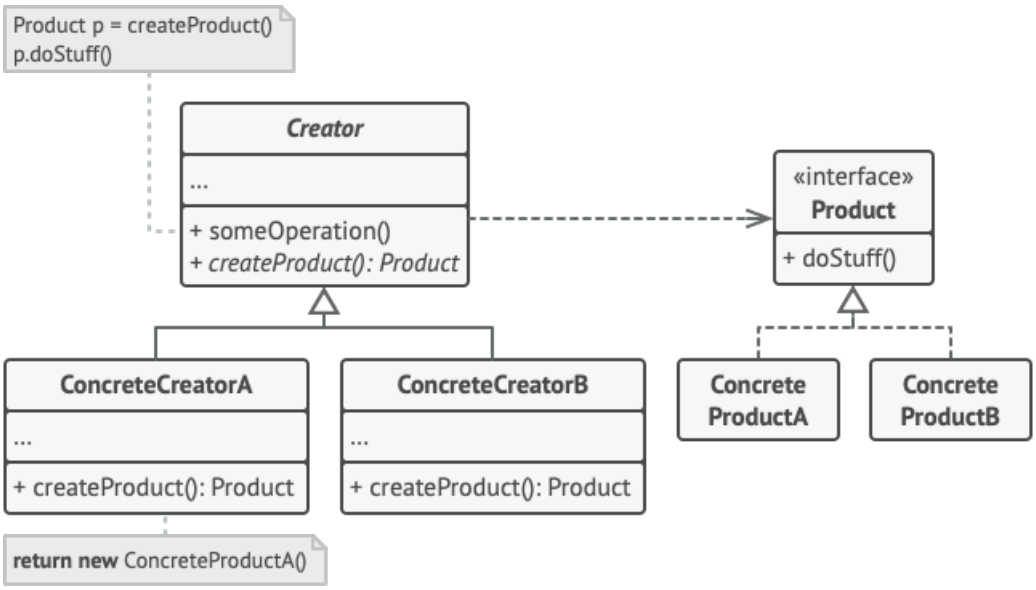
Lưu ý rằng lớp con chỉ có thể trả về các product khác nhau khi chúng cùng triển khai một base class hoặc interface chung. Factory method sẽ trả về kiểu base class hoặc interface này. Ở bài toán vận tải trên thì ta cần một interface $MeanOfTransport$, các lớp $Truck$, $Ship$ sẽ implements interface này. Lớp $TransportFactory$ là lớp chứa factory method để tạo ra các đối tượng Truck hoặc Ship.

2.3 Khả năng áp dụng

Sử dụng Factory Method khi:

- Không biết trước kiểu và các phụ thuộc của object mà code sẽ làm việc với nó.
- Muốn cung cấp cho người dùng thư viện hoặc framework của chúng ta một cách để mở rộng các thành phần sẵn có bên trong nó.
- Các lớp ủy quyền trách nhiệm cho một trong một số lớp con.

2.4 Cấu trúc



Hình 1

Trong đó:

- **Product:** là interface chung cho tất cả các đối tượng có thể được tạo ra bởi *Creator*.
- **Concrete Product:** triển khai interface *Product*.
- **Creator:** khai báo abstract method factory trả về interface *Product*.
- **Concrete Creator:** override method factory gốc để trả về đại diện của một *ConcreteProduct* cụ thể.

2.5 Ưu nhược điểm

Ưu điểm:

- Tránh được liên kết chặt chẽ giữa Creator và các Concrete Product.
- Single Responsibility Principle: có thể chuyển code tạo sản phẩm đến một nơi trong chương trình, làm cho việc maintain dễ dàng hơn.
- Open/Closed Principle: có thể dễ dàng thêm các loại Product mới vào chương trình mà không làm ảnh hưởng đến code hiện tại.

Nhược điểm:

- Code sẽ trở nên phức tạp khi có quá nhiều class con để triển khai pattern.

2.6 Cách triển khai

1. Các product phải triển khai cùng một interface. Interface này khai báo các phương thức có ý nghĩa trong mọi product.

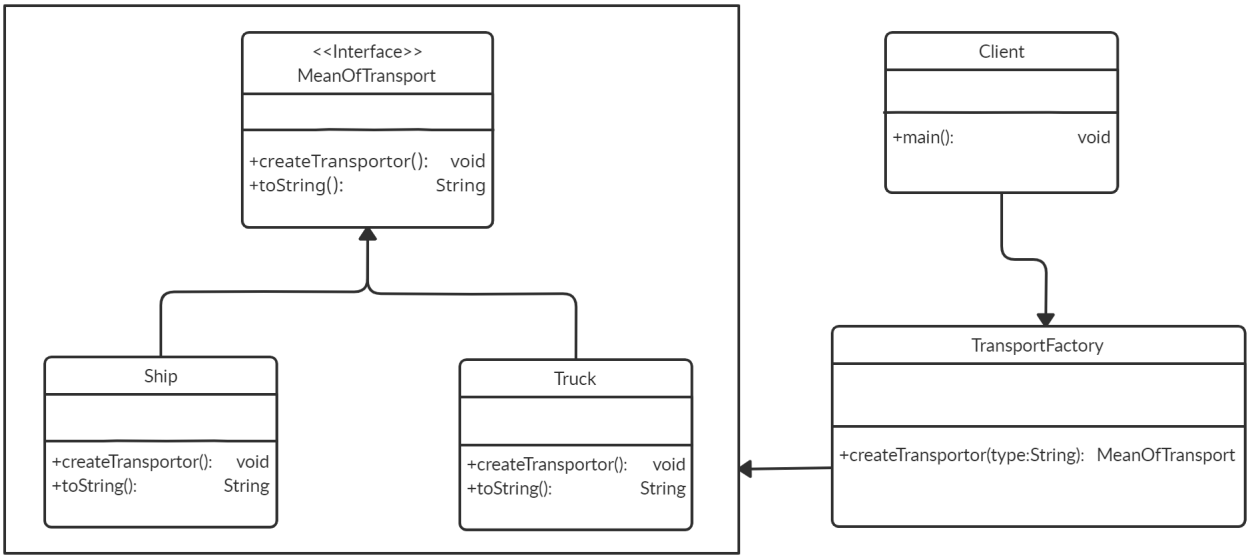
2. Thêm một phương thức gốc(factory method) trống bên trong lớp Creator. Kiểu trả về phải phù hợp với interface Product chung.

3. Trong code của Creator, tìm tất cả các tham chiếu đến constructor của Product. Thay thế chúng bằng cách gọi đến factory method.

4. Tạo tập hợp các lớp con của Creator cho mỗi loại product được liệt kê trong factory method. Override factory method trong các lớp con để trả về product tương ứng.

5. Cuối cùng, nếu base factory method(gốc) không còn bất cứ mã gì, ta có thể chuyển nó thành abstract method. Ngược lại, có thể đặt nó là phương thức hành vi mặc định.

2.7 Code minh họa



Tạo interface mà tất cả các phương tiện đều triển khai(Product):

```
package factorymethod;
public interface MeanOfTransport {
    void createTransportor();
    String toString();
}
```

Tạo các lớp cụ thể implements interface MeanOfTransport(Concrete product):

```
package factorymethod;

public class Truck implements MeanOfTransport{
    public void createTransportor() {
        System.out.println("Created Truck");
    }
    public String toString() {
        return "Truck!";
    }
}
```

```
package factorymethod;

public class Ship implements MeanOfTransport{
    public void createTransportor() {
        System.out.println("Created Ship");
    }
    public String toString() {
        return "Ship!";
    }
}
```

Tạo lớp nhà máy TransportFactory để sinh ra các đối tượng của các lớp cụ thể dựa trên thông tin đã cho(Conrete Creator):

```
package factorymethod;

public class TransportFactory {
    MeanOfTransport createTransportor(String type) {
        switch (type) {
            case "truck":
            case "Truck":
                return new Truck();
            case "ship":
            case "Ship":
                return new Ship();
            default:
                return null;
        }
    }
}
```

Sinh ra các phương tiện bằng cách sử dụng TransportFactory để lấy đối tượng của lớp cụ thể bằng cách truyền thông tin như "Truck" hoặc "Ship":

```
1  package factorymethod;
2
3  public class TestFactoryMethod {
4      public static void main(String[] args) {
5          TransportFactory tranFactory = new TransportFactory();
6          MeanOfTransport transportor = tranFactory.createTransportor("
       Ship");
7          transportor.createTransportor();
8          System.out.println(transportor.toString());
9
10         transportor = tranFactory.createTransportor("Truck");
11         transportor.createTransportor();
12         System.out.println(transportor.toString());
13
14
15     }
16 }
```

Kết quả:

```
Created Ship
Ship!
Created Truck
Truck!
```

2.8 Mối quan hệ với các Design Patterns khác

- Nhiều thiết kế bắt đầu bằng cách sử dụng Factory Method(ít phức tạp hơn và có thể tùy chỉnh nhiều hơn thông qua các lớp con) và phát triển theo hướng Abstract Factory, Prototype hoặc Builder(linh hoạt nhưng phức tạp hơn).

- Các lớp Abstract Factory thường dựa trên một tập hợp các Factory Method, nhưng cũng có thể sử dụng Protype để tạo ra các phương thức trên lớp này.

- Có thể sử dụng Factory Method cùng Iterator để cho phép các lớp con của tập hợp trả về các kiểu vòng lặp khác nhau tương thích với tập hợp.

ABSTRACT FACTORY

Abstract Factory là một Creational Design Pattern, cung cấp một interface có nhiệm vụ tạo ra các đối tượng liên quan hoặc phụ thuộc tới nhau mà không cần phải chỉ ra cụ thể lớp của các đối tượng đó.

Cụ thể, nhiệm vụ của Abstract Factory Pattern là tạo ra các factory của các đối tượng có liên quan tới nhau. Các factory này được tạo ra giống như trong Factory Method Pattern. Có thể xem như Abstract Factory như một nhà máy lớn chứa nhiều nhà máy nhỏ và mỗi nhà máy nhỏ sẽ có các phân xưởng và mỗi phân xưởng sẽ sản xuất các sản phẩm khác nhau.

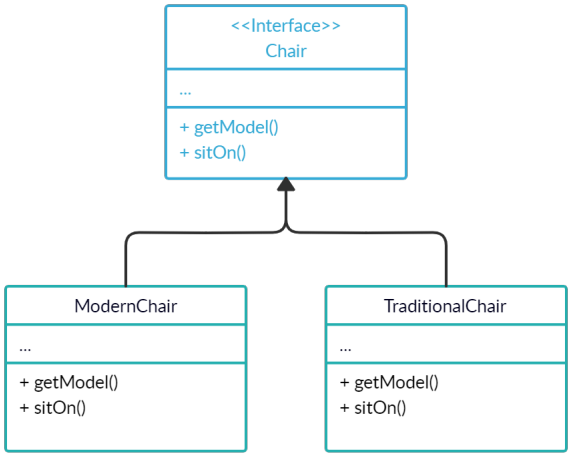
3.1 Đặt vấn đề

Giả sử chúng ta tạo ra ứng dụng quản lí một kho hàng cung cấp đồ nội thất. Có hai loại sản phẩm liên quan tới nhau và được xếp vào một nhóm: *Chair* và *Table*. Tuy nhiên, mỗi sản phẩm này thì có nhiều phân loại như: Modern, Traditional.

Ví dụ như một khách hàng đã từng mua một chiếc Table, sau đó một thời gian khách mới đặt thêm một Chair nhưng kho hàng gửi đến Chair thuộc phân loại khác của Table, khiến khách hàng không hài lòng. Do đó, chúng ta cần có một cách để tạo ra các đồ vật nội thất riêng lẻ để chúng phù hợp với các đồ vật khác trong cùng một nhóm. Ngoài ra, chúng ta cũng không muốn thay đổi code hiện có khi thêm sản phẩm mới hoặc nhóm sản phẩm vào chương trình.

3.2 Giải pháp

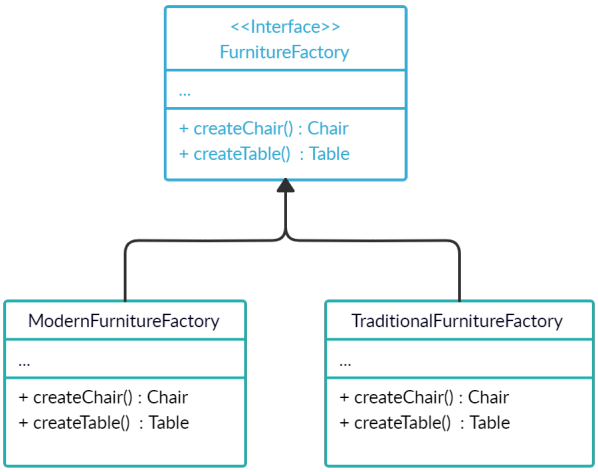
Đầu tiên Abstract Factory Pattern đề xuất khai báo rõ ràng các Interface cho từng sản phẩm riêng biệt của nhóm sản phẩm (ví dụ: *Chair*, *Table*). Sau đó tạo các đối tượng con khai triển theo Interface. Ví dụ: tất cả các phân loại của ghế đều có thể triển khai theo Interface *Chair* như: *ModernChair*, *TraditionalChair*, ...



Hình 2: Interface Chair và các đối tượng con

Tạo ra một Interface với danh sách các phương thức tạo cho tất cả các sản phẩm là một phần của nhóm sản phẩm (ví dụ: $createChair()$, $createTable()$). Các phương thức này phải trả về các loại sản phẩm trừu tượng thông qua các Interface như: $Chair$, $Table$.

Đối với mỗi phân loại của một nhóm sản phẩm, tạo một class Factory riêng trả lại các sản phẩm của một phân loại cụ thể như: ModernFurnitureFactory chỉ có thể tạo các đối tượng $ModernChair$, $ModernTable$.



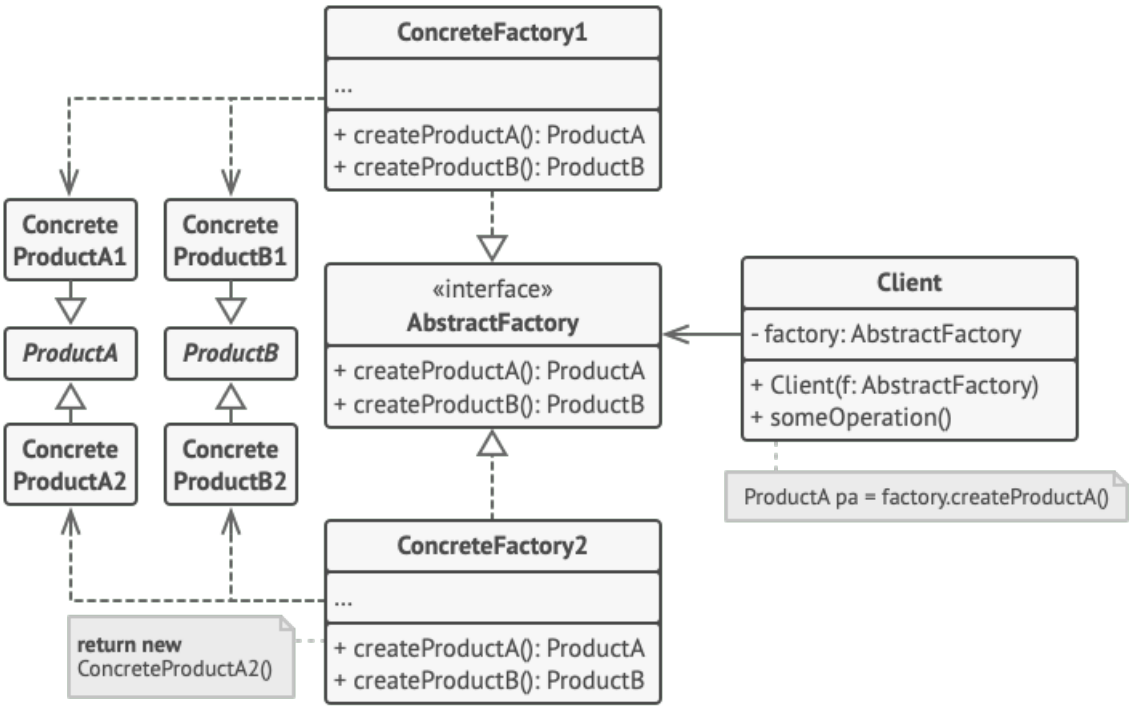
Hình 3: Mỗi nhà máy tương ứng với một phân loại nhóm sản phẩm cụ thể

Khi cần tạo ra một đối tượng sản phẩm thuộc phân loại bất kì thì nhà máy ứng với phân loại đó sẽ tạo ra đúng sản phẩm cần tạo bằng phương thức $create()$.

3.3 Khả năng áp dụng

- Làm việc với nhóm các sản phẩm liên quan tới nhau, có nhiều phân loại và chúng biến đổi nhiều, nhưng không muốn phụ thuộc vào concrete class của các đối tượng này.
- Có một class với một tập $FactoryMethod$.
- Muốn dễ quản lí một $FactoryMethod$ lớn (có nhiều if-else hay switch-case).

3.4 Cấu trúc



Hình 4

Trong đó:

- **Abstract Product:** là các interfaces cho các loại riêng biệt nhưng các sản phẩm có liên quan sẽ tạo thành một nhóm sản phẩm.
- **Concrete Product:** triển khai interface $AbstractProduct$ nhóm lại với nhau theo phân loại.
- **Abstract Factory:** Interface khai báo một tập các phương thức cho việc tạo mỗi $AbstractProduct$.

- **Concrete Factories:** triển khai các phương thức tạo $ConcreteProduct$ của $AbstractFactory$. Mỗi $ConcreteFactory$ có trách nhiệm đến một phân loại của sản phẩm và chỉ tạo các sản phẩm của phân loại này.
- **Client:** là đối tượng sử dụng $AbstractFactory$ và các $AbstractProduct$.

3.5 Ưu nhược điểm

Ưu điểm:

- Đảm bảo rằng sản nhận từ factory tương thích với những cái khác.
- Tránh được việc gắn chặt giữa một $ConcreteProduct$ và *clientcode*.
- $SingleResponsibilityPrinciple$: Có thể tách code tạo product tới một nơi, giúp code dễ dàng để hỗ trợ.
- $Open/ClosedPrinciple$: Dễ dàng để thêm phân loại sản phẩm mới mà không làm ảnh ảnh hưởng đến code cũ.
- Ẩn đi sự phức tạp trong quá trình khởi tạo các Object đối với người dùng

Nhược điểm:

- Code có thể phức tạp hơn vì sử dụng nhiều Class và Interface.

3.6 Cách triển khai

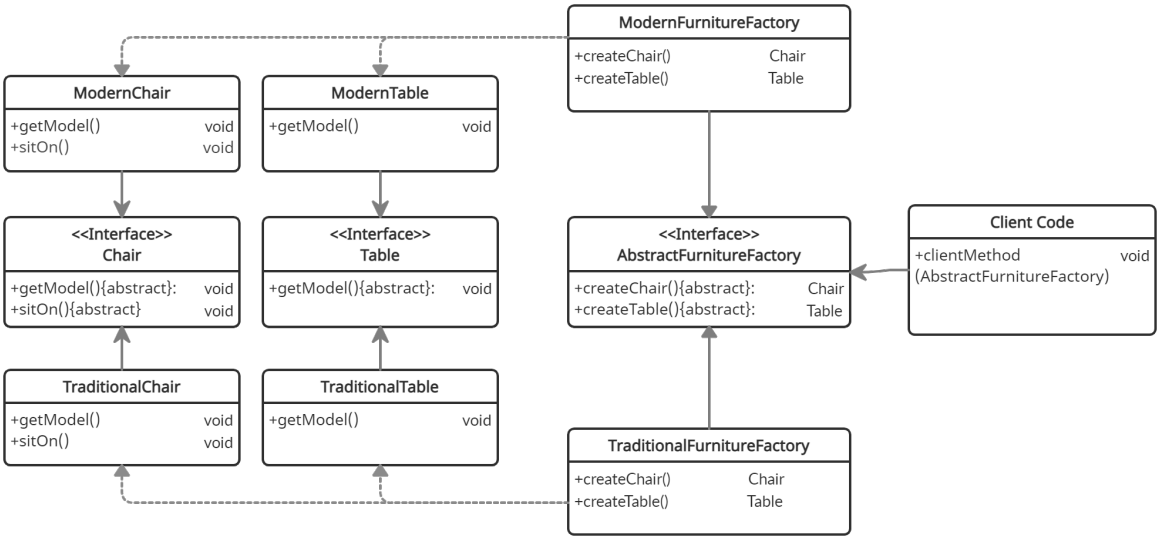
1. Ánh xạ một ma trận của các sản phẩm và phân loại của chúng
2. Khai báo abstract product interfaces cho tất cả các sản phẩm. Tiếp theo tạo tất cả concrete product class để implement các interfaces này.
3. Khai báo abstract factory interface với một tập các creation methods cho tất cả abstract products.
4. Implement một tập các concrete factory classes, một class cho mỗi phân loại từng sản phẩm.

5. Khởi tạo factory trong ứng dụng. Nó nên khởi tạo một trong những concrete factory classes, phụ thuộc trên cấu hình ứng dụng hoặc môi trường hiện tại. Truyền đối tượng factory này đến tất cả class nơi mà xây dựng sản phẩm.

6. Quét qua code và tìm tất cả chỗ gọi trực tiếp đến product constructor. Thay thế chúng với creation method phù hợp cho đối tượng factory.

3.7 Code minh họa

Xử lí vấn đề nêu ra ở mục Đặt vấn đề



Tạo interface của các sản phẩm Chair và Table:

Hai đối tượng này sẽ được chia nhỏ ra thành các phân loại.

```
package abstractfactory;

public interface Chair {
    String getModel();
    void sitOn();
}
```

```
package abstractfactory;

public interface Table {
    String getModel();
}
```

Tạo các Concrete Product của từng sản phẩm:

Đây là các phân loại của 2 đối tượng bên trên.
Các đối tượng con của Chair:

```
package abstractfactory;

public class ModernChair implements Chair {
    public String getModel() {
        return "Modern Chair";
    }

    public void sitOn() {
        return;
    }

}
```

```
package abstractfactory;

public class TraditionalChair implements Chair {
    public String getModel() {
        return "Traditional Chair";
    }

    public void sitOn() {
        return;
    }

}
```

Các đối tượng con của Table:

```
package abstractfactory;

public class ModernTable implements Table {
    public String getModel() {
        return "Modern Table";
    }

}
```

```
package abstractfactory;

public class TraditionalTable implements Table {
    public String getModel() {
        return "Traditional Table";
    }

}
```

Khai báo Abstract Factory interface:

Tạo ra nhà máy abstract để làm việc với Client Code.

```
package abstractfactory;

public interface AbstractFurnitureFactory {
    Chair createChair();
    Table createTable();
}
```

Tạo các Concrete Factory theo 2 phân loại là Modern và Traditional:

Hai nhà máy abstract được tạo ra, mỗi nhà máy sẽ tạo ra cả Chair và Table nhưng cùng một phân loại.

Nhà máy tạo ra các sản phẩm Modern:

```
package abstractfactory;

public class ModernFurnitureFactory implements AbstractFurnitureFactory
    {
    public Chair createChair() {
        return new ModernChair();
    }

    public Table createTable() {
        return new ModernTable();
    }

}
```

Nhà máy tạo ra các sản phẩm Traditional:

```
package abstractfactory;

public class TraditionalFurnitureFactory implements
    AbstractFurnitureFactory {
    public Chair createChair() {
        return new TraditionalChair();
    }

    public Table createTable() {
        return new TraditionalTable();
    }

}
```

Tạo lớp Client: tạo các Concrete Product thông qua Abstract Factory

```
package abstractfactory;

```

```
public class Client {
    public void clientMethod(AbstractFurnitureFactory factory) {
        Chair chair = factory.createChair();
        Table table = factory.createTable();
        System.out.println(chair.getModel());
        System.out.println(table.getModel());
    }
}
```

Thực thi TestMain:

```
package abstractfactory;

public class TestAbstractFactory {
    public static void main(String[] args) {
        Client client = new Client();
        client.clientMethod(new ModernFurnitureFactory());
        client.clientMethod(new TraditionalFurnitureFactory());
    }

}
```

Kết quả:

```
Modern Chair
Modern Table
Traditional Chair
Traditional Table
```

3.8 Mối quan hệ với các Design Patterns khác

- Nhiều thiết kế bắt đầu bằng cách sử dụng Factory Method (ít phức tạp hơn và có thể tùy chỉnh nhiều hơn thông qua các lớp con) và phát triển theo hướng Abstract Factory, Prototype hoặc Builder (linh hoạt hơn nhưng phức tạp hơn).

- Builder tập trung vào việc xây dựng các đối tượng phức tạp theo từng bước. Abstract Factory chuyên tạo các nhóm đối tượng liên quan. Abstract Factory trả lại sản phẩm ngay lập tức, trong khi Builder cho phép chạy một số bước xây dựng bổ sung trước khi tìm nạp sản phẩm.

- Các lớp Abstract Factory thường dựa trên một tập hợp các Factory Method, nhưng chúng ta cũng có thể sử dụng Prototype để soạn các phương thức trên các lớp này.

- Abstract Factory có thể thay thế Facade khi chúng ta chỉ muốn ẩn cách các đối tượng hệ thống con được tạo ra khỏi client code.
- Chúng ta có thể sử dụng Abstract Factory cùng với Bridge. Việc ghép nối này rất hữu ích khi một số trừu tượng được xác định bởi Bridge chỉ có thể hoạt động với các triển khai cụ thể. Trong trường hợp này, Abstract Factory có thể đóng gói các mối quan hệ này và ẩn sự phức tạp khỏi client code.
- Một số design pattern như Abstract Factory, Builder, Prototype có thể dùng cùng với Singleton.

BUILDER

Builder pattern là một mẫu thiết kế khởi tạo cho phép chúng ta xây dựng các đối tượng phức tạp theo từng bước. Pattern này cho phép chúng ta tạo ra các kiểu và biểu diễn khác nhau của một đối tượng bằng cách sử dụng cùng một mã khởi tạo(construction code).

4.1 Đặt vấn đề

Giả sử nhiệm vụ của chúng ta là tạo ra một đối tượng phức tạp, khởi tạo từng bước của nhiều trường và các đối tượng lồng nhau, constructor có rất nhiều tham số. Trường hợp xấu hơn là chúng nằm rải rác trên toàn bộ client code. Trong hầu hết các trường hợp, các tham số không phải lúc nào cũng cần sử dụng tới làm cho việc gọi constructor trở nên khá xấu xí.

4.2 Giải pháp

Builder pattern cho phép chúng ta xây dựng các đối tượng phức tạp theo từng bước. Builder không cho phép các đối tượng khác truy cập vào sản phẩm khi nó đang được xây dựng.

Pattern này sắp xếp việc xây dựng đối tượng thành một tập hợp các bước. Để tạo một đối tượng, chúng ta cần thực hiện một loạt các bước này trên một đối tượng Builder. Quan trọng là chúng ta không cần phải gọi tất cả các bước, mà chỉ cần gọi những bước cần thiết để tạo ra một cấu hình cụ thể của một đối tượng.

Một số bước xây dựng có thể yêu cầu thực hiện khác nhau khi chúng ta cần xây dựng các hình ảnh đại diện khác nhau của sản phẩm.

Director class:

Chúng ta có thể trích xuất một loạt các lệnh gọi đến các bước của trình tạo mà chúng ta sử dụng để xây dụng một sản phẩm thành một lớp riêng biệt được gọi là *Director*. Lớp *Director* xác định thứ tự thực hiện các bước xây dụng, trong khi *Builder* cung cấp việc triển khai cho các bước đó. *Director class* không thực sự cần thiết trong chương trình khi chúng ta có thể gọi các bước xây dựng theo thứ tự cụ thể trực tiếp từ Client. Tuy nhiên *Director class* hữu ích trong việc đưa các quá trình xây dựng khác nhau để chúng ta có thể sử dụng lại chúng trong chương trình. Ngoài ra,

Director hoàn toàn giấu kín các chi tiết cấu tạo sản phẩm với client code. Client chỉ cần liên kết *Builder* với *Director*, khởi tạo *Director* và nhận kết quả từ *Builder*.

4.3 Khả năng áp dụng

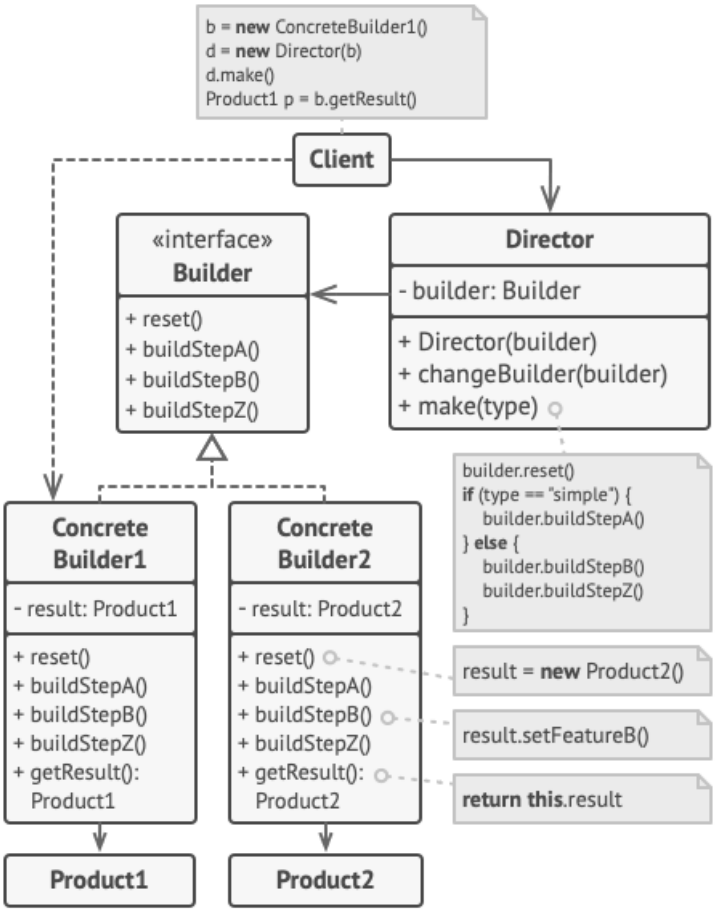
Chúng ta nên nghĩ tới Builder pattern khi:

- Tạo một object phức tạp: có nhiều thuộc tính(nhiều hơn 4) và một số bắt buộc(requried), một số không bắt buộc(optional).
- Có quá nhiều constructor.
- Muốn tách rời quá trình xây dựng một đối tượng phức tạp từ các phần tạo nên đối tượng.
- Muốn kiểm soát quá trình xây dựng.
- Khi phía Client mong đợi nhiều cách khác nhau cho đối tượng được xây dựng.

4.4 Cấu trúc

Trong đó:

- **Builder:** là abstract class hoặc interface khai báo phương thức tạo đối tượng.
- **Concrete Builder:** kế thừa Builder và triển khai 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 lại các thể hiện mà nó đã tạo ra trước đó.
- **Product:** đại diện cho đối tượng cần tạo, đối tượng này phức tạp, có nhiều thuộc tính.
- **Director/Client:** là nơi sẽ gọi tới Builder để tạo đối tượng.



Hình 5

4.5 Ưu nhược điểm

Ưu điểm:

- Hỗ trợ, loại bớt việc phải viết nhiều constructor. Code dễ đọc, dễ bảo trì hơn khi số lượng thuộc tính(propery) bắt buộc để tạo một object có từ 4 hoặc 5 thuộc tính. Giảm bớt số lượng constructor, không cần truyền giá trị null cho các tham số không sử dụng.
- Ít bị lỗi do việc gán sai tham số khi mà có nhiều tham số trong constructor: bởi vì người dùng đã biết được chính xác giá trị gì khi gọi phương thức tương ứng.
- Đối tượng được xây dựng an toàn hơn: bởi vì nó đã được tạo hoàn chỉnh trước khi sử dụng.

Nhược điểm:

- Duplicate code khá nhiều do cần phải copy tất cả các thuộc tính từ Product trong Builder.

- Tăng độ phức tạp của code(tổng thể) do số lượng class tăng lên.

4.6 Cách triển khai

1. Xác định rõ ràng các bước xây dựng chung.

2. Khai báo các bước trong Builer interface.

3. Tạo Concrete Builder cho từng Product và triển khai các bước xây dựng chúng.

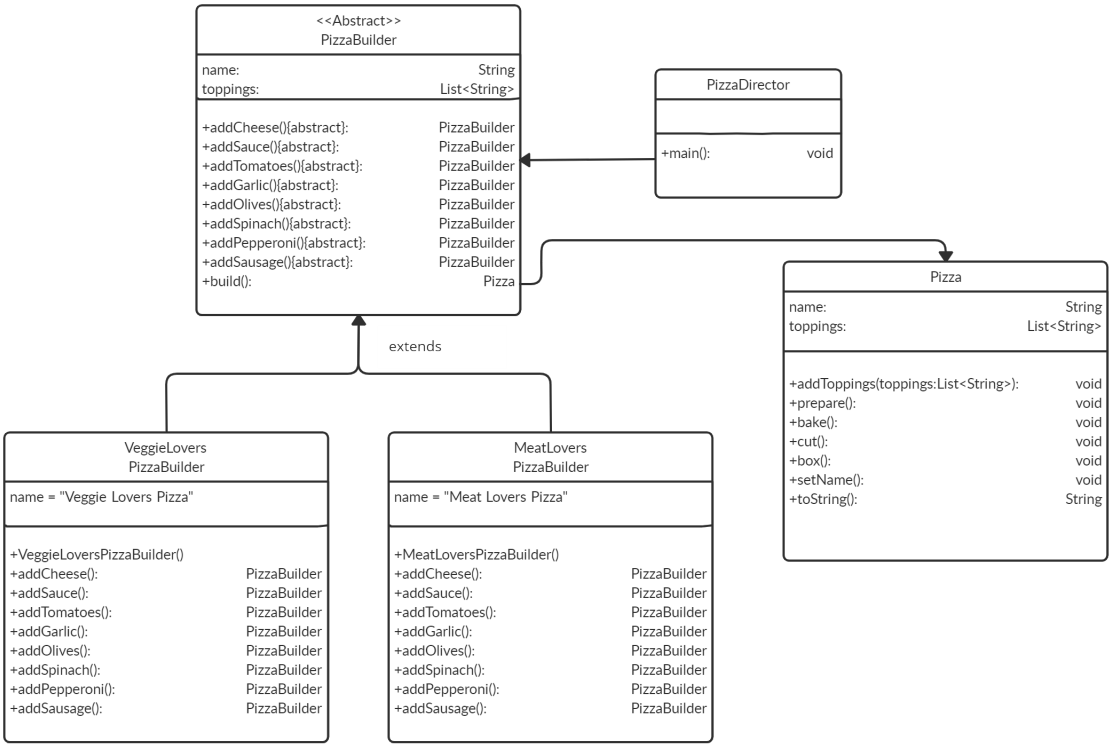
4. Nên tạo một Director class vì nó có thể bao gồm nhiều cách khau nhau để xây dựng một Product bằng cách sử dụng cùng một đối tượng builder.

5. Client code tạo cả Builder và Director. Trước khi xây dụng, Client phải chuyển một đối tượng Builder cho Director. Thông thường Client chỉ thực hiện việc này một lần thông qua các tham số của Director constructor. Director sử dụng đối tượng Builder trong tất cả công trình xây dựng khác. Một cách tiếp cận khác là Builder được chuyển trực tiếp đến construction method của Director.

6. Chỉ có thể nhận được kết quả xây dựng trực tiếp từ Director nếu tất cả các Product cùng tuân theo một interface. Nếu không Client sẽ lấy kết quả từ Builder.

4.7 Code minh họa

Sử dụng Builder xây dựng chương trình order Pizza.



Tạo lớp trừu tượng PizzaBuilder(Builder), lớp này chứa các phương thức trừu tượng tương ứng với việc thêm các toppings khác nhau cho từng loại pizza:

```
1  package builder;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public abstract class PizzaBuilder {
7    String name;
8    List<String> toppings = new ArrayList<String>();
9
10   public abstract PizzaBuilder addCheese();
11   public abstract PizzaBuilder addSauce();
12   public abstract PizzaBuilder addTomatoes();
13   public abstract PizzaBuilder addGarlic();
14   public abstract PizzaBuilder addOlives();
15   public abstract PizzaBuilder addSpinach();
16   public abstract PizzaBuilder addPepperoni();
17   public abstract PizzaBuilder addSausage();
18   public Pizza build() {
19     Pizza pizza = new Pizza();
20     pizza.setName(this.name);
```

```
    pizza.addToppings(toppings);
    return pizza;
  }
}
```

Tạo lớp Pizza(Product), lớp này đại diện cho loại pizza được tạo ra:

```
package builder;

import java.util.*;

public class Pizza {
  String name;
  List<String> toppings;

  void addToppings(List<String> toppings) {
    this.toppings = toppings;
  }

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

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

```
    }
    return display.toString();
  }
}
```

Các lớp MeatLoversPizzaBuilder và VeggieLoversPizzaBuilder (Concrete Builder) triển khai các phương thức trừu tượng của PizzaBuilder:

```
package builder;

public class MeatLoversPizzaBuilder extends PizzaBuilder {
  public MeatLoversPizzaBuilder() {
    this.name = "Meat Lovers Pizza";
  }

  public PizzaBuilder addCheese() {
    this.toppings.add("mozzerella");
    return this;
  }

  public PizzaBuilder addSauce() {
    this.toppings.add("NY style sauce");
    return this;
  }

  public PizzaBuilder addTomatoes() {
    this.toppings.add("sliced tomatoes");
    return this;
  }

  public PizzaBuilder addGarlic() {
    this.toppings.add("garlic");
    return this;
  }

  public PizzaBuilder addOlives() {
    return this;
  }

  public PizzaBuilder addSpinach() {
    return this;
  }

  public PizzaBuilder addPepperoni() {
    this.toppings.add("pepperoni");
    return this;
  }

  public PizzaBuilder addSausage() {
```

```
    this.toppings.add("sausage");
    return this;
  }
}
```

```
package builder;

public class VeggieLoversPizzaBuilder extends PizzaBuilder {
  public VeggieLoversPizzaBuilder() {
    this.name = "Veggie Lovers Pizza";
  }
  public PizzaBuilder addCheese() {
    this.toppings.add("parmesan");
    return this;
  }
  public PizzaBuilder addSauce() {
    this.toppings.add("sauce");
    return this;
  }
  public PizzaBuilder addTomatoes() {
    this.toppings.add("chopped tomatoes");
    return this;
  }
  public PizzaBuilder addGarlic() {
    this.toppings.add("garlic");
    return this;
  }
  public PizzaBuilder addOlives() {
    this.toppings.add("green olives");
    return this;
  }
  public PizzaBuilder addSpinach() {
    this.toppings.add("spinach");
    return this;
  }
  public PizzaBuilder addPepperoni() {
    return this;
  }
  public PizzaBuilder addSausage() {
    return this;
  }
}
```

Lớp Client/Director:

```
package builder;

public class PizzaDirector {
  public static void main(String[] args) {
    PizzaBuilder veggieBuilder = new VeggieLoversPizzaBuilder();
    Pizza veggie = veggieBuilder.addSauce().addCheese().addOlives().
   addTomatoes().addSausage().build();
```

```
7       veggie.prepare();
8       veggie.bake();
9       veggie.cut();
10      veggie.box();
11      System.out.println(veggie);
12
13      PizzaBuilder meatBuilder = new MeatLoversPizzaBuilder();
14      Pizza meat = meatBuilder.addSauce().addTomatoes().addCheese().
        addSausage().addPepperoni().build();
15      meat.prepare();
16      meat.bake();
17      meat.cut();
18      meat.box();
19      System.out.println(meat);
20
21      StringBuilder sb = new StringBuilder();
22      sb.append("\nTesting String Builder\n").append(veggie).insert(0, "
        ====");
23      System.out.println("Length of the String Builder: " + sb.length());
24      System.out.println("Result of the String Builder: " + sb.toString())
        ;
25
26      String sb2 = new StringBuilder().append("\nTesting String Builder\n"
        ).append(meat).insert(0, "====").toString();
27      System.out.println(sb2);
28    }
29  }
```

Kết quả:

```
Prepare Veggie Lovers Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   sauce
   parmesan
   green olives
   chopped tomatoes
Bake for 25 minutes at 350
Cut the pizza into diagonal slices
Place pizza in official PizzaStore box
---- Veggie Lovers Pizza ----
sauce
parmesan
green olives
chopped tomatoes

Prepare Meat Lovers Pizza
```

```
Tossing dough...
Adding sauce...
Adding toppings:
   NY style sauce
   sliced tomatoes
   mozzerella
   sausage
   pepperoni
Bake for 25 minutes at 350
Cut the pizza into diagonal slices
Place pizza in official PizzaStore box
---- Meat Lovers Pizza ----
NY style sauce
sliced tomatoes
mozzerella
sausage
pepperoni

Length of the String Builder: 103
Result of the String Builder: ====
Testing String Builder
---- Veggie Lovers Pizza ----
sauce
parmesan
green olives
chopped tomatoes

====
Testing String Builder
---- Meat Lovers Pizza ----
NY style sauce
sliced tomatoes
mozzerella
sausage
pepperoni
```

4.8 Mối quan hệ với các Design Patterns khác

- Nhiều thiết kế bắt đầu bằng cách sử dụng Factory Method (ít phức tạp hơn và có thể tùy chỉnh nhiều hơn thông qua các lớp con) và phát triển theo hướng Abstract Factory, Prototype hoặc Builder (linh hoạt hơn nhưng phức tạp hơn).

- Builder tập trung vào việc xây dựng các đối tượng phức tạp theo từng bước. Abstract Factory chuyên tạo các family đối tượng liên quan. Abstract Factory trả lại sản phẩm ngay lập tức, trong khi Builder cho phép chạy một số bước xây dựng bổ sung trước khi tìm nạp sản phẩm.

- Có thể sử dụng Builder khi tạo các cây Composite phức tạp vì có thể lập trình các bước xây dựng của nó để hoạt động một cách đệ quy.

- Có thể kết hợp Builder với Bridge: lớp director đóng vai trò trừu tượng, trong khi các trình xây dựng khác nhau đóng vai trò triển khai.

- Abstract Factories, Builders và Prototypes đều có thể được thực hiện dưới dạng các Singleton.

PROTOTYPE

Prototype pattern là một trong những patterns thuộc nhóm khởi tạo. Nhiệm vụ của nó là khởi tạo một đối tượng bằng cách clone một đối tượng đã tồn tại thay vì khởi tạo với từ khóa *new*. Đối tượng mới là một bản sao có thể giống 100% với đối tượng gốc, chúng ta có thể thay đổi dữ liệu của nó mà không ảnh hưởng đến đối tượng gốc.

5.1 Đặt vấn đề

Giả sử chúng ta có một đối tượng và muốn tạo bản sao chính xác của đối tượng đó. Đầu tiên, chúng ta phải tạo một đối tượng mới cùng một lớp. Sau đó phải sao chép các thuộc tính gốc sang đối tượng mới. Nhưng, không phải đối tượng nào cũng có thể sao chép theo cách đó vì một số thuộc tính có thể là *private* và không thể nhìn thấy từ bên ngoài của chính đối tượng.

5.2 Giải pháp

Prototype pattern ủy quyền quá trình nhân bản cho các đối tượng thực tế đang được nhân bản. Pattern khai báo một interface chung cho tất cả các đối tượng hỗ trợ nhân bản. Interface này cho phép sao chép một đối tượng mà không ghép code của chúng ta với class của đối tượng đó. Thông thường những interface như vậy chỉ chứa duy nhất một clone method.

Việc triển khai của clone method giống nhau ở tất cả các class. Phương thức này tạo một đối tượng của lớp hiện tại và chuyển tất cả các thuộc tính của đối tượng gốc sang đối tượng mới. Một đối tượng có khả năng nhân bản thì được gọi là Prototype. Khi các đối tượng có số lượng thuộc tính đủ lớn thì việc nhân bản có thể là một giải pháp thay thế cho phân lớp(subclassing). Tóm lại, cách Prototype hoạt động như sau: tạo một tập hợp các đối tượng, được định nghĩa theo nhiều cách khác nhau. Khi muốn tạo một đối tượng giống đối tượng đã định nghĩa thì chỉ cần sao chép một nguyên mẫu thay vì xây dựng một đối tượng mới từ đầu.

5.3 Khả năng áp dụng

Trong một số trường hợp sau, chúng ta nên áp dụng Prototype pattern:

- Chúng ta có một object và cần phải tạo một object mới khác dựa trên object ban đầu mà không thể sử dụng toán tử new hay các hàm contructor để khởi tạo. Lý do đơn giản là chúng ta không hề được biết thông tin nội tại của object đó hoặc object đó đã có thể bị che dấu đi nhiều thông tin khác mà chỉ cho ta một thông tin rất giới hạn không đủ để hiểu được. Do vậy ta ko thể dùng toán tử new để khởi tạo nó được. Giải pháp: để cho chính object mẫu tự xác định thông tin và dữ liệu sao chép.

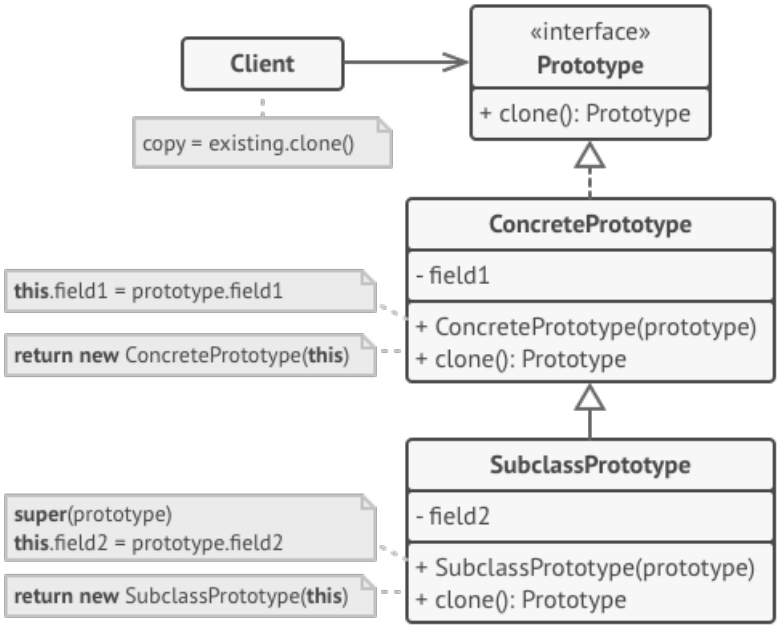
- Khởi tạo đối tượng lúc run-time: chúng ta có thể xác định đối tượng cụ thể sẽ được khởi tạo lúc runtime nếu class được implement / extend từ một Prototype.

- Muốn truyền đối tượng vào một method nào đó để xử lý, thay vì truyền đối tượng gốc có thể ảnh hưởng dữ liệu thì ta có thể truyền đối tượng sao chép.

- Chi phí của việc tạo mới đối tượng (bằng cách sử dụng toán tử new) là lớn.

- Ẩn độ phức tạp của việc khởi tạo đối tượng từ phía Client.

5.4 Cấu trúc



Hình 6

Trong đó:

- **Prototype:** khai báo một class, interface hoặc abstract class cho việc clone chính nó.
- **Concrete Prototype:** các lớp này implements interface hoặc kế thừa lớp abstract được cung cấp bởi Prototype để nhân bản chính nó. Các lớp này thực hiện cụ thể phương thức clone(). Lớp này sẽ không thực sự cần thiết nếu Prototype là một class và nó đã thực hiện việc nhân bản chính nó.
- **Client:** tạo object mới bằng cách gọi Prototype thực hiện nhân bản chính nó.

5.5 Ưu nhược điểm

Ưu điểm:

- Cải thiện hiệu suất: giảm chi phí để tạo ra một đối tượng mới, điều này làm tăng hiệu suất so với việc sử dụng từ khóa *new* để tạo đối tượng mới.
- Giảm độ phức tạp cho việc khởi tạo đối tượng: do mỗi lớp chỉ implement cách clone của chính nó.
- Giảm việc phân lớp, tránh việc tạo nhiều lớp con cho việc khởi tạo đối tượng như Abstract Factory Pattern.

Nhược điểm:

- Quá mức cần thiết cho một dự án sử dụng rất ít đối tượng và(hoặc) không có trọng tâm cơ bản về việc mở rộng chuỗi nguyên mẫu.
- Mỗi lớp con của nguyên mẫu phải triển khai hoạt động clone() có thể khó, khi các lớp đang được xem xét đã tồn tại. Ngoài ra việc triển khai clone() cũng có thể khó khăn khi bên trong của chúng bao gồm các đối tượng không hỗ trợ sao chép hoặc có các tham chiếu vòng tròn.
- Nó cũng ẩn các lớp sản phẩm cụ thể khỏi khách hàng.

5.6 Cách triển khai

1. Tạo interface nguyên mẫu cùng clone method.

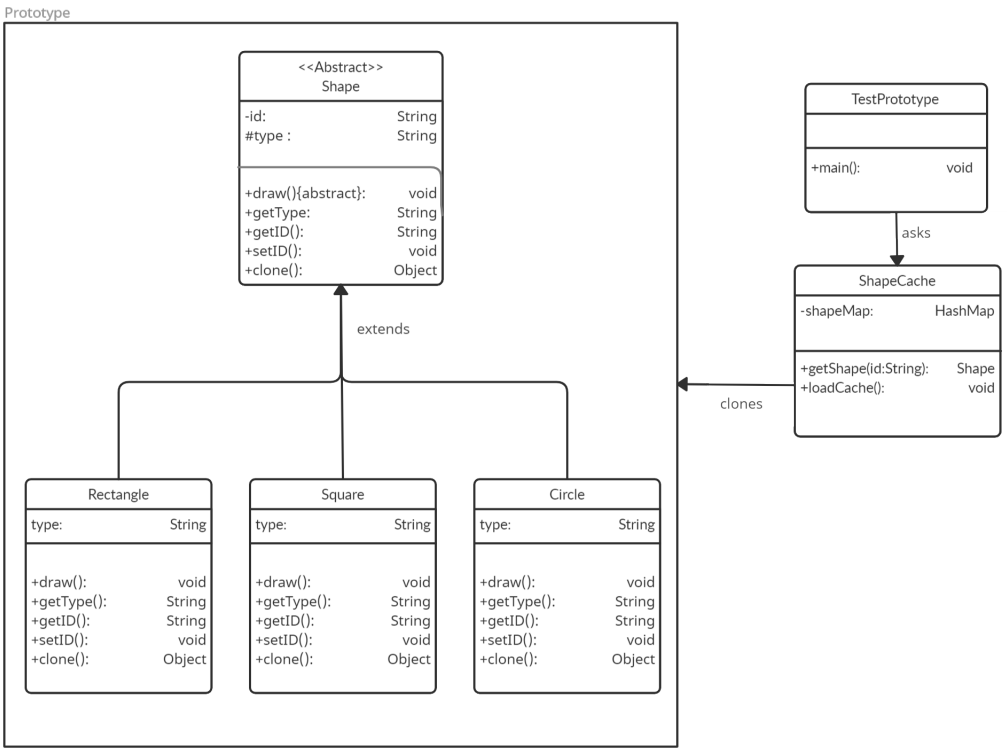
2. Prototype class phải xác định constructor chấp nhận đối tượng của lớp làm đối số. Constructor phải sao chép các giá trị của tất cả các thuộc tính từ đối tượng được truyền vào đối tượng mới được tạo. Nếu thay đổi lớp con thì phải gọi constructor của lớp cha(super()) để lớp cha xử lí việc sao chép các private attribute.

3. Clone method thường chỉ gồm một dòng: chạy một toán tử *new* với bản nguyên mẫu của constructor.

4. Tạo một danh sách lưu trữ các nguyên mẫu thường xuyên được sử dụng. Thay thế việc gọi trực tiếp constructor của các lớp con bằng việc gọi đến phương thức gốc của danh sách lưu trữ nguyên mẫu.

5.7 Code minh họa

Chương trình tạo ra các bản sao của các đối tượng Shape.



Tạo một abstract class implements Cloneable interface:

```java
package prototype;

public abstract class Shape implements Cloneable {
    private String id;
    protected String type;
    abstract void draw();

    public String getType(){
        return type;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Object clone() {
        Object clone = null;
        try {
            clone = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clone;
    }
}
```

Tạo concrete classes kế thừa lớp trừu tượng trên:

```java
package prototype;

public class Rectangle extends Shape {
    public Rectangle(){
        type = "Rectangle";
    }

    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}
```

```java
package prototype;

public class Square extends Shape {
    public Square(){
        type = "Square";
    }

```

```
    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}
```

```
package prototype;

public class Circle extends Shape {
    public Circle(){
        type = "Circle";
    }

    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}
```

Tạo lớp ShapeCache để lấy các lớp concrete từ dữ liệu và lưu trữ chúng vào Hashtable:

```
package prototype;

import java.util.Hashtable;

public class ShapeCache {
    private static Hashtable<String, Shape> shapeMap  = new Hashtable<
   String, Shape>();

    public static Shape getShape(String shapeId) {
        Shape cachedShape = shapeMap.get(shapeId);
        return (Shape) cachedShape.clone();
    }

    public static void loadCache() {
        Circle circle = new Circle();
        circle.setId("1");
        shapeMap.put(circle.getId(),circle);

        Square square = new Square();
        square.setId("2");
        shapeMap.put(square.getId(),square);

        Rectangle rectangle = new Rectangle();
        rectangle.setId("3");
        shapeMap.put(rectangle.getId(), rectangle);
    }
}
```

Sử dụng lớp ShapeCache để lấy các bản sao của các hình được

lưu lại ở Hashtable:

```
package prototype;

public class TestPrototype {
    public static void main(String[] args) {
        ShapeCache.loadCache();

        Shape clonedShape = (Shape) ShapeCache.getShape("1");
        System.out.println("Shape : " + clonedShape.getType());

        Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
        System.out.println("Shape : " + clonedShape2.getType());

        Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
        System.out.println("Shape : " + clonedShape3.getType());
    }
}
```

Kết quả:

```
Shape : Circle
Shape : Square
Shape : Rectangle
```

5.8 Mối quan hệ với các Design Patterns khác

- Nhiều thiết kế bắt đầu bằng cách sử dụng Factory Method (ít phức tạp hơn và có thể tùy chỉnh nhiều hơn thông qua các lớp con) và phát triển theo hướng Abstract Factory, Prototype hoặc Builder (linh hoạt hơn nhưng phức tạp hơn).

- Các lớp Abstract Factory thường dựa trên một tập hợp các Factory Methods, nhưng cũng có thể sử dụng Prototype để soạn các phương thức trên các lớp này.

- Các thiết kế sử dụng nhiều Composite và Decorator thường có thể được hưởng lợi từ việc sử dụng Prototype. Áp dụng pattern cho phép clone các cấu trúc phức tạp thay vì xây dựng lại chúng từ đầu.

- Abstract Factories, Builders và Prototype đều có thể được triển khai dưới dạng các Singleton.

SINGLETON

Singleton pattern là một mẫu thiết kế khởi tạo cho phép chúng ta đảm bảo rằng một lớp chỉ có duy nhất một instance(khởi tạo), đồng thời cung cấp một điểm truy cập toàn cục cho instance đó.

6.1 Đặt vấn đề

Đôi khi trong quá trình phân tích thiết kế một hệ thống, chúng ta mong muốn có những đối tượng tồn tại duy nhất và có thể truy xuất mọi lúc mọi nơi. Làm thế nào để tạo nên một đối tượng như vậy khi triển khai code? Chúng ta có thể nghĩ tới việc sử dụng một biến toàn cục (*globalvariable*). Tuy nhiên, việc sử dụng biến toàn cục sẽ phá vỡ quy tắc đóng gói (*encapsulation*) của OOP. Để giải quyết vấn đề này, người ta hướng đến một giải pháp là sử dụng Singleton Pattern.

6.2 Giải pháp

Để triển khai một Singleton, có nhiều cách tiếp cận khác nhau nhưng tất cả chúng đều có các đặc điểm chung sau đây:

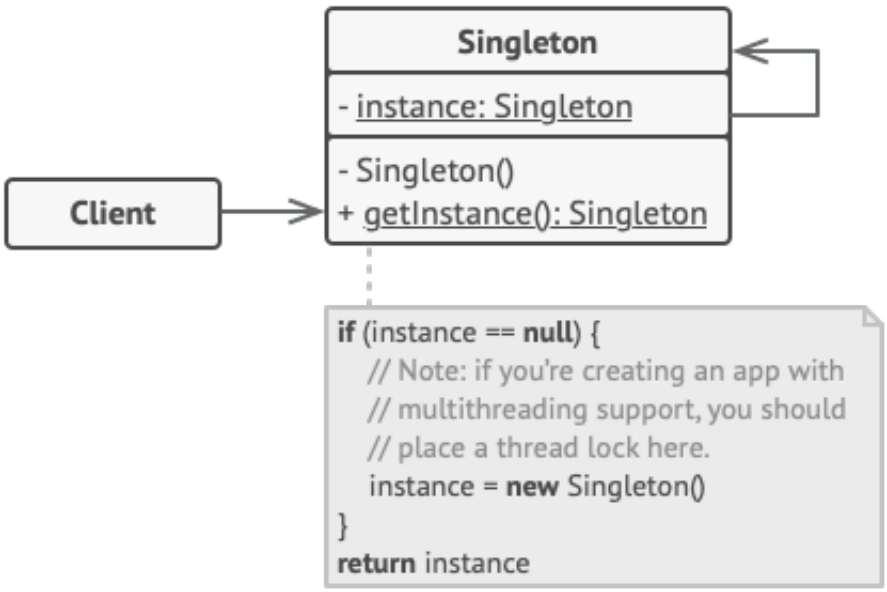
- Để modifier của constructor là **private** để hạn chế khởi tạo lớp từ các lớp khác.
- Đặt **private static final variable** đảm bảo biến chỉ được khởi tạo trong class.
- Có một **public static method** để return **instance** được khởi tạo ở trên, đây là điểm truy cập toàn cục cho các lớp bên ngoài để lấy thể hiện của lớp Singleton.

6.3 Khả năng áp dụng

- Vì class dùng Singleton chỉ tồn tại một instance nên nó thường được dùng để giải quyết các vấn đề như: thread pool, caches, dialog boxes, đối tượng được sử dụng để ghi log và các đối tượng đóng vai trò là trình điều khiển thiết bị cho các thiết bị như máy in, card đồ họa,...

- Sử dụng mẫu Singleton khi cần kiểm soát chặt chẽ hơn đối với các biến toàn cục.

6.4 Cấu trúc



Singleton chỉ liên quan đến một lớp duy nhất(thường không được gọi là Singleton) với các thuộc tính và phương thức khác(không được show ra).

Lớp Singleton có một biến tĩnh trỏ đến một thể hiện duy nhất của lớp.

Có một private constructor và một phương thức tĩnh cung cấp quyền truy cập vào single instance.

6.5 Ưu nhược điểm

Ưu điểm:

- Ai cũng có thể truy cập vào instance của singleton class, thực hiện gọi nó ở bất cứ đâu.
- Dữ liệu ứng dụng không thay đổi bởi chỉ có một instance duy nhất.
- Đối tượng Singleton chỉ được khởi tạo khi nó được yêu cầu lần đầu tiên (giống String pool).

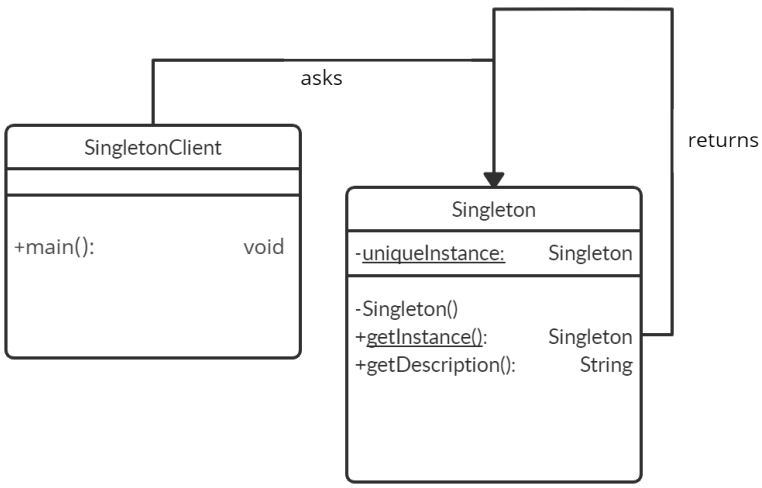
Nhược điểm:

- Khó triển khai một cách hiệu quả để đảm bảo rằng một class chỉ có một đối tượng.
- Chỉ tạo một instance duy nhất.
- Hạn chế số lượng các instance của một class.
- Không sử dụng được đa hình.

6.6 Cách triển khai

1. Định nghĩa một attribute là private static và đó là thể hiện duy nhất của class này.
2. Định nghĩa public static getInstance() dùng để khởi tạo đối tượng (hàm accessor).
3. Thực hiện "lazy-init" trong hàm accessor (chỉ khi gọi mới khởi tạo thể hiện).
4. Đặt modifier của constructor là *private*. Phương thức tĩnh của lớp vẫn có thể gọi constructor nhưng không thể gọi các đối tượng khác.
5. Với code của Client, thay thế việc gọi trực tiếp *constructor* của Singleton bằng việc gọi đến *static creation method* (getInstance()) của nó.

6.7 Code minh họa



```java
package singleton;

public class Singleton {
    // cung cap thuoc tinh co kieu tra ve la Singleton
  private static Singleton uniqueInstance;

  private Singleton() {

    }

    // tao phuong thuc truy cap kieu tra ve la mot tham chieu den thuoc tinh uniqueInstance
  public static Singleton getInstance() {
        // thuc hien lazy-initialization
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }

  // other useful methods here
  public String getDescription() {
    return "I'm a classic Singleton!";
  }
}
```

```java
package singleton;

public class SingletonClient {
  public static void main(String[] args) {
        //Client goi doi tuong Singleton va truy cap phuong thuc getInstance()
        //Neu doi tuong Singleton nay chua duoc khoi tao lan nao thi no tien hanh tao doi tuong moi
        // Nguoc lai se tra ve Single co tu truoc
    Singleton singleton = Singleton.getInstance();
    System.out.println(singleton.getDescription());
  }
}
```

Kết quả:

```
I'm a classic Singleton!
```

6.8 Mối quan hệ với các Design Patterns khác

- Một số design pattern như Abstract Factory, Builder, Prototype có thể dùng cùng với Singleton.

- Các đối tượng Facade và State cũng thường là Singleton.

ADAPTER

Adapter Pattern là một design pattern thuộc nhóm Structural Pattern, cho phép các đối tượng có Interface không tương thích làm việc với nhau.

7.1 Đặt vấn đề

Giả sử chúng ta có một số các đối tượng, nhưng các đối tượng đó lại không sử dụng được với phần mềm đang xây dựng vì nó có Interface không tương thích. Làm thế nào để không tốn thời gian sửa lại các đối tượng cũ và tiếp tục sử dụng trực tiếp chúng được? Để giải quyết vấn đề này, chúng ta sử dụng Adapter Pattern.

7.2 Giải pháp

Tạo nên Interface mới cho các đối tượng đó bằng lớp Interface trung gian gọi là "Adapter". Lớp này chỉ có trách nhiệm chuyển đổi Interface, các phương thức của nó sẽ gọi lại các phương thức của đối tượng mà nó "adapt".

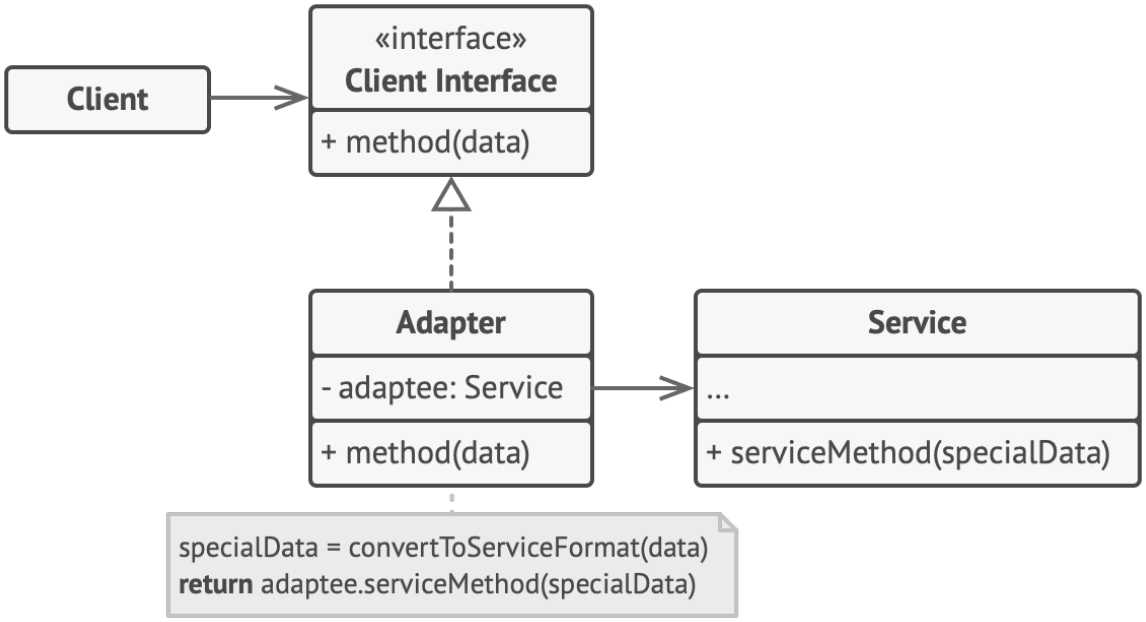
7.3 Khả năng áp dụng

- sử dụng lại một số lớp con hiện có thiếu một số chức năng phổ biến không thể thêm vào lớp cha.
- sử dụng một số lớp hiện có, nhưng Interface của lớp đó không tương thích với phần còn lại của code.

7.4 Cấu trúc

Design Pattern này có thể triển khai theo 2 cách:

Object Adapter

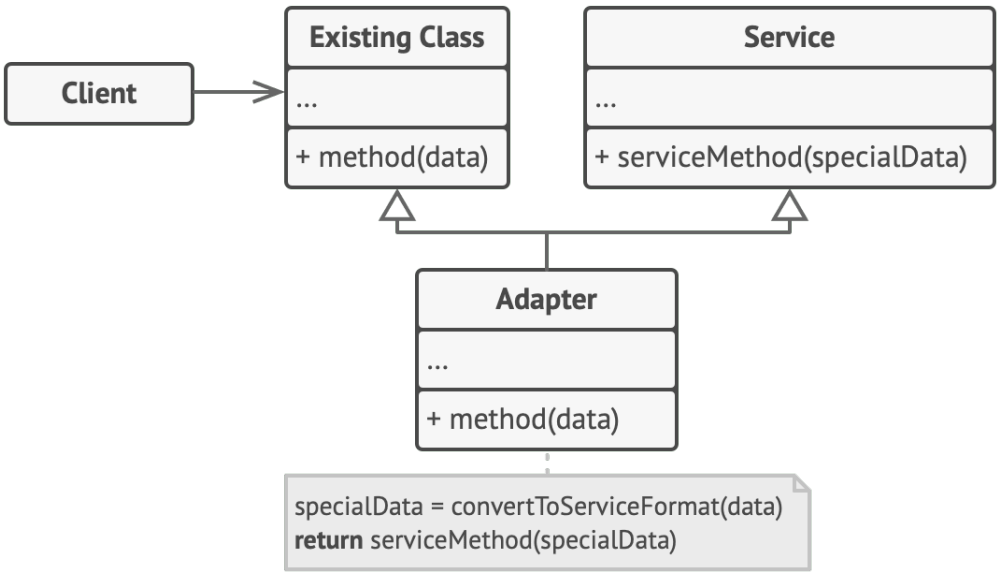


Hình 7

Trong đó:

- **Client:** chứa business logic của chương trình.
- **Client interface:** mô tả một giao thức mà các lớp khác phải tuân theo để có thể collab với client code.
- **Service (Adaptee):** là một class hữu ích (thường là bên thứ 3 hoặc kế thừa). Client không thể sử dụng trực tiếp lớp này vì nó có interface không tương thích.
- **Adapter:** là một class có thể hoạt động với cả Client và Service: nó implements Client interface, trong khi đóng gói Service object. Adapter khi được gọi từ Client thông qua Adapter Interface sẽ chuyển chúng thành các cuộc gọi service object được bao bọc ở định dạng mà nó có thể hiểu được.

Class Adapter



Hình 8

Trong đó:

- **Class Adapter:** không cần phải bọc bất kỳ object nào vì nó kế thừa các hành vi từ Client và Service. Adaptation xảy ra trong các phương thức override. Kết quả của Adapter có thể được sử dụng thay cho một Client class hiện có.

7.5 Ưu nhược điểm

Ưu điểm:

- Cho phép nhiều đối tượng interface khác nhau giao tiếp với nhau.
- Phân tách việc chuyển đổi interface với business logic chính của chương trình.
- Cách tiếp cận này có thêm một ưu điểm là ta có thể sử dụng adapter với các class con của adaptee. (Liskov substitution principle).
- Làm việc với adapter class thay vì sửa đổi bên trong adaptee class đã có sẵn, thuận tiện cho việc mở rộng (Open/closed principle).
- Client tiếp cận thông qua interface, thay vì implementation (Software design principle).

Nhược điểm:

- Tất cả yêu đầu phải được chuyển tiếp thông qua adapter, làm tăng thêm một ít chi phí
- Độ phức tạp của code nhìn chung tăng lên vì phải thêm interface và lớp.
- Vì không phải lúc nào ta cũng có thể thích nghi các method của các interface khác nhau với nhau, nên exception có thể xảy ra. Vấn đề này có thể tránh được nếu client cẩn thận hơn và adapter có tài liệu hướng dẫn rõ ràng.

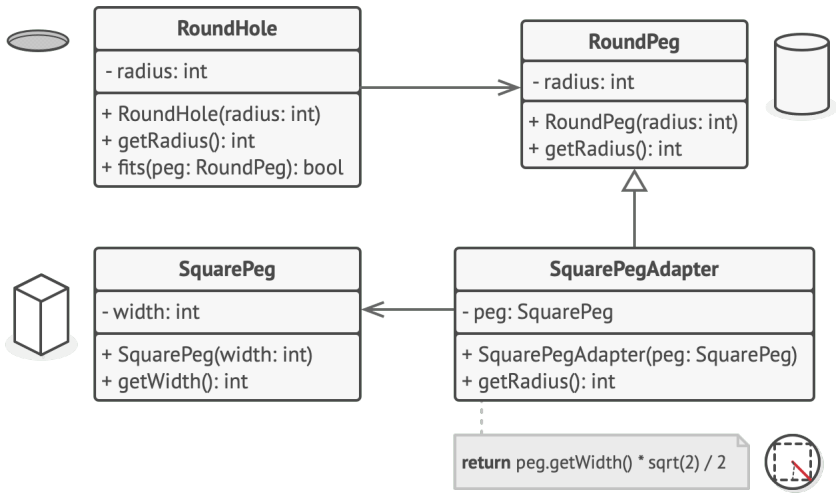
7.6 Cách triển khai

1. Client gửi yêu cầu ở interface.
2. Tạo một lớp Adapter để triển khai Client interface đó.
3. Lớp Adapter giữ reference đến Adaptee (cách phổ biến là truyền nó vào tham số của constructor của Adapter).
4. Adapter lần lượt triển khai các methods của Client interface, làm những công việc như chuyển đổi data 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.

7.7 Code minh họa

Ban đầu ta có một lỗ hình tròn và chỉ có cọc tròn mới chui qua lỗ đó, nhưng chương trình muốn tạo thêm 1 đối tượng là cọc vuông nhưng vẫn muốn đối tượng mới này có thể chui qua lỗ tròn.

Đối tượng RoundHole có phương thức fits với tham số truyền vào là RoundPeg peg



Hình 9: Chuyển đổi SquarePeg thành RoundPeg để cho vừa RoundHole

```java
package adapter;

public class RoundHole {
    private int radius;
    public RoundHole(int radius) {
        this.radius = radius;
    }
    public int getRadius() {
        return radius;
    }
    public boolean fits(RoundPeg peg) {
        return radius >= peg.getRadius();
    }
}
```

Đối tượng RoundPeg

```java
package adapter;

public class RoundPeg {
    private int radius;
    public RoundPeg(int radius) {
        this.radius = radius;
    }
    public int getRadius() {
        return radius;
    }
}
```

Đối tượng SquarePeg không thể được sử dụng với RoundHole

```java
package adapter;

public class SquarePeg {
    private int width;
    public SquarePeg(int width) {
```

```
        this.width = width;
    }
    public int getWidth() {
        return width;
    }
}
```

Đối tượng Adapter là lớp con của RoundPeg và được tạo bằng cách truyền Adaptee(peg) vào Constructor của Adapter và được lưu nội bộ

```
package adapter;

public class SquarePegAdapter extends RoundPeg {
    /*
     * It extends the RoundPeg class to let
     * the adapter objects act as round pegs.
     */
    private SquarePeg peg;
    public int getRadius() {
        /*
         * The adapter pretends that it's a round peg
         * with a radius that could fit the square peg
         * that the adapter actually wraps.
         */
        return (int) (peg.getWidth() * Math.sqrt(2) / 2);
    }
    public SquarePegAdapter(int radius) {
        super(radius);
    }
    public SquarePegAdapter(SquarePeg peg) {
        super((int) (peg.getWidth() * Math.sqrt(2) / 2));
        this.peg = peg;
    }
}
```

Client Code

```
package adapter;

public class TestMain {

    public static void main(String[] args) {
        // test RoundHole and RoundPeg
        RoundHole hole = new RoundHole(10);
        RoundPeg rPeg = new RoundPeg(8);
        System.out.println(hole.fits(rPeg));// true

        // test SquarePeg
        SquarePeg sPeg = new SquarePeg(5);
//        System.out.println(hole.fits(sPeg)); // can't compile
```

```
14
15          // test Adapter Object
16          SquarePegAdapter adapter = new SquarePegAdapter(sPeg);
17          System.out.println(hole.fits(adapter));// true
18      }
19  }
```

Kết quả:

```
true
true
```

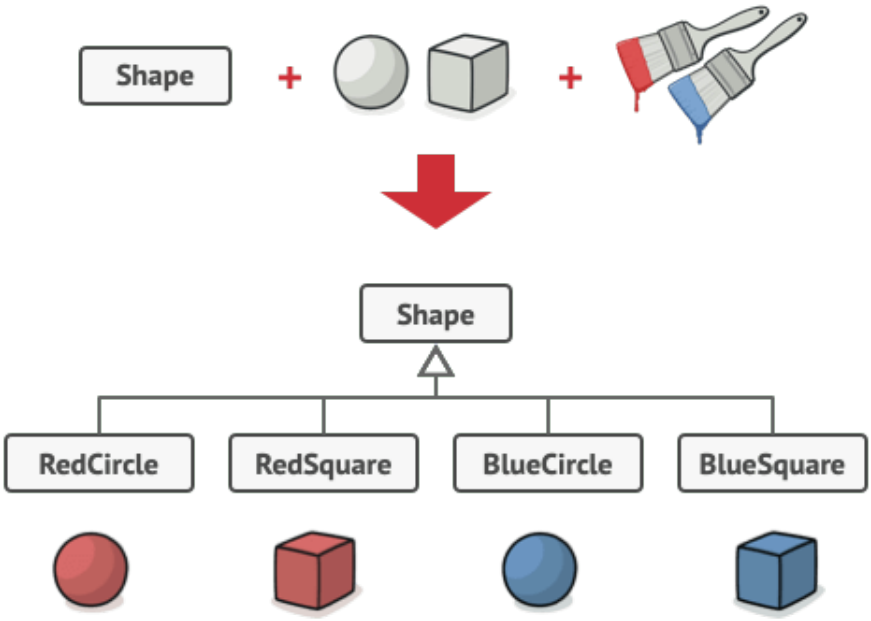
7.8 Mối quan hệ với các Design Patterns khác

- Adapter giúp các class hoạt động sau khi nó đã được thiết kế, Bridge giúp chúng hoạt động trước khi được thiết kế.
- Adapter cung cấp một interface khác, Proxy cung cấp cùng một loại interface, Decorator cung cấp một interface phức tạp hơn.
- Facade định nghĩa một interface mới, Adapter sử dụng lại một interface.

BRIDGE

Bridge pattern là một mẫu thiết kế cấu trúc. Nó thể hiện rõ sự phân cấp giữa giao diện trong cả interface và các thành phần implement. Ý tưởng của nó là tách tính trừu tượng(abstraction) ra khỏi tính hiện thực(implementation) của nó. Từ đó chúng ta có thể chỉnh sửa hoặc thay đổi mà không làm ảnh hưởng đến những nơi có sử dụng lớp ban đầu. Điều đó có nghĩa là, ban đầu chúng ta thiết kê một class với rất nhiều xử lý, bây giờ chúng ta không muốn để nhưng xử lý trong class đấy nữa. Vì thế, chúng ta sẽ tạo ra một class khác và chuyển các xử lý đó qua class mới. Khi đó, class 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.

8.1 Đặt vấn đề

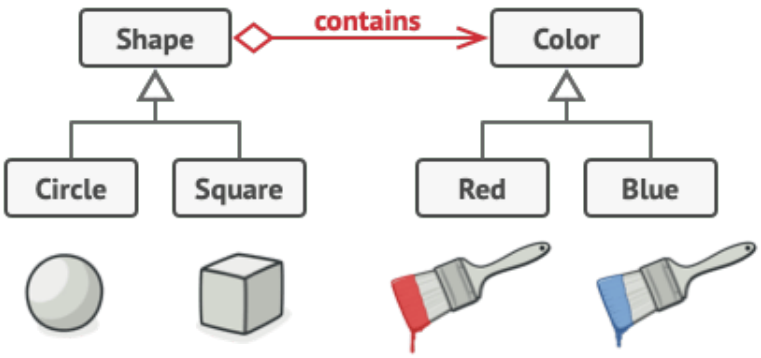


Giả sử chúng ta có một lớp Shape, Circle và Square là các lớp con của nó. Chúng ta muốn mở rộng hệ thống phân cấp của lớp này để kết hợp thêm màu sắc(Red và Blue), vì vậy chúng ta sẽ phải tạo các lớp con của Shape ứng với từng màu. Tuy nhiên chúng ta đang có hai hình là Circle và Square, như vậy chúng ta sẽ phải tạo ra bốn lớp kết hợp, chẳng hạn như RedCircle, RedSquare, BlueCircle, BlueSquare. Nếu cứ như vậy, khi thêm các loại Shape và màu mới thì hệ thống phân cấp sẽ mở rộng theo hàm

mũ. Ví dụ, khi thêm một đối tượng Triangle thì chúng ta sẽ phải tạo ra hai class, mỗi class dành cho mỗi màu(Red và Blue). Sau đó, thêm một màu mới, chúng ta lại phải tạo ra ba class, mỗi class dành cho mỗi hình(Circle, Square và Triangle). Về lâu dài thì điều này thực sự không tốt.

8.2 Giải pháp

Vấn đề này xảy ra vì chúng ta mở rộng các lớp Shape theo chiều hướng: mỗi hình ứng với mỗi màu. Bridge pattern sẽ giải quyết vấn đề này bằng việc chuyển đổi từ kế thừa(inheritance) sang cấu thành đối tượng(object composition). Điều này có nghĩa là chúng ta sẽ tách một trong các thành phần vào hệ thống phân cấp riêng biệt để các lớp ban đầu sẽ tham chiếu đến một đối tượng của hệ thống phân cấp mới thay vì có tất cả trạng thái và hành vi của nó trong một lớp. Theo cách tiếp cận này, chúng ta sẽ tách các màu liên quan thành lớp Color với lớp con là: Red và Blue. Lớp Shape có thể tham chiếu đến một trong các đối tượng màu. Bây giờ Shape có thể ủy quyền mọi công việc liên quan đến màu sắc cho đối tượng màu được liên kết. Tham chiếu đó sẽ hoạt động như cầu nối(Bridge) giữa các lớp Shape và Color. Qua giải pháp này, việc thêm các hình và màu mới sẽ không yêu cầu thay đổi hệ thống phân cấp hình dạng và ngược lại.



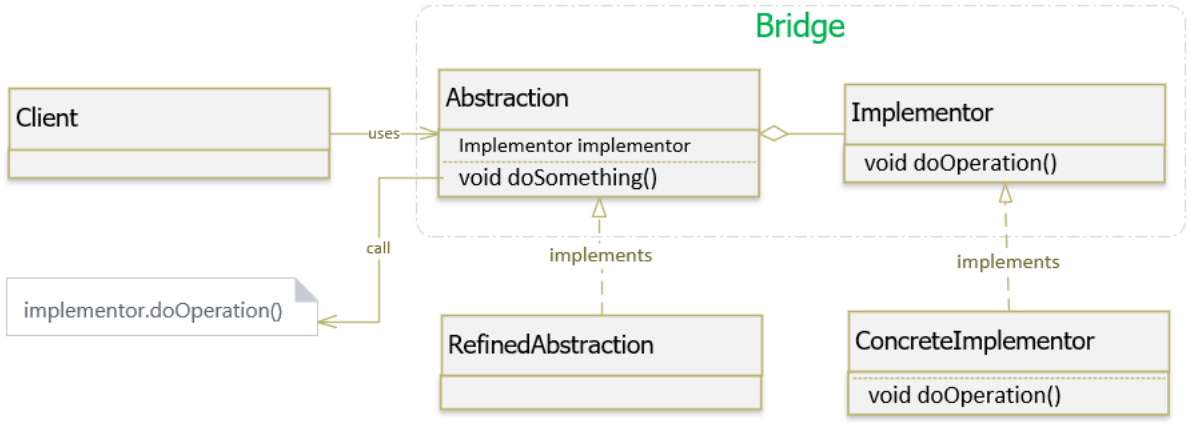
8.3 Khả năng áp dụng

Sử dụng Bridge Pattern khi:

- Muốn tách ràng buộc giữa Abstraction và Implementation để có thể dễ dàng mở rộng độc lập nhau.
- Cả Abstraction và Implementation 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.

8.4 Cấu trúc



Trong đó:

- **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).
- **Refinded Abstraction:** hiện thực(implement) các phương thức đã được định ra trong Abstraction bằng cách sử dụng một tham chiếu đến một đối tượng của Implemnter.
- **Implementor** định các interface cho các lớp hiện thực. Thông thường nó là interface định ra các tác vụ nào đó của Abstraction.
- **Concrete Implementor:** hiện thực Implementor interface.

8.5 Ưu nhược điểm

Ưu điểm:

- Giảm sự phụ thuộc giữa abstraction và implementation.
- Giảm số lượng những lớp con không cần thiết.

- Code gọn gàng hơn và kích thước ứng dụng sẽ nhỏ hơn.
- Dễ bảo trì hơn, dễ mở rộng về sau.
- Cho phép ẩn các chi tiết implement từ Client.

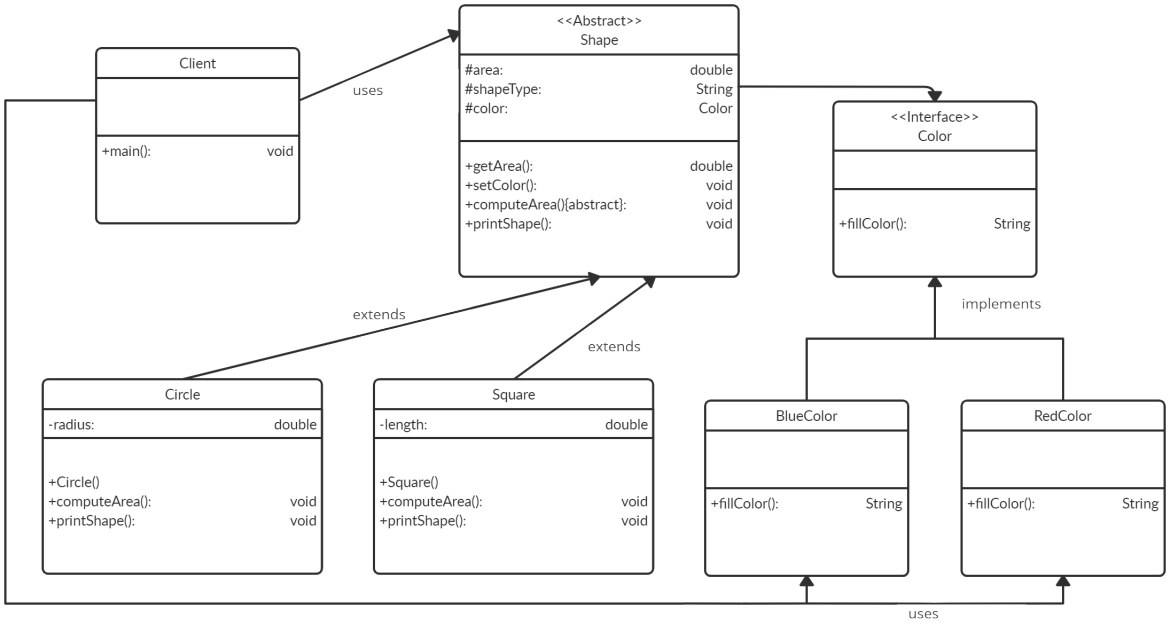
Nhược điểm:

- Có thể làm code phức tạp hơn nếu áp dụng Bridge pattern cho lớp có tính liên kết cao.

8.6 Cách triển khai

1. Xác định các thành phần có thể kết hợp.
2. Xem những hành động mà Client cần và định nghĩa chúng trong base abstraction class.
3. Xác định các hành động có sẵn trên tất cả các nền tảng, khai báo những thứ mà phần abstraction cần trong interface triển khai chung.
4. Đối với các nền tảng, tạo các lớp triển khai cụ thể và chúng phải tuân theo interface triển khai.
5. Trong Abstraction class thêm thuộc tính tham chiếu cho kiểu Implementor. Abstraction ủy quyền hầu hết các công việc cho đối tượng Implementor được tham chiếu trong thuộc tính đó.
6. Nếu có một biến thể logic cấp cao, hãy tạo các Abstraction tinh chỉnh cho mỗi biến thể bằng việc mở rộng base abstraction class.
7. Client phải chuyển một Implementor đến constructor của Abstraction để liên kết đối tượng này với đối tượng kia. Sau đó chỉ cần làm việc với abstraction object.

8.7 Code minh họa



Tạo bridge Implementor interface:

```
package bridge;

public interface Color {
    String fillColor();
}
```

Tạo Concrete Implementor class:

```
package bridge;

public class RedColor implements Color{
    @Override
    public String fillColor() {
        return "filled with Red color.";
    }
}
```

```
package bridge;

public class BlueColor implements Color {
    @Override
    public String fillColor() {
        return "filled with Blue color.";
    }
}
```

Tạo Shape(Abstraction) có sử dụng Color interface(Implementor):

```java
package bridge;

public abstract class Shape {
    protected double area;
    protected String shapeType;
    protected Color color;

    public Shape(Color color) {
        this.color = color;
    }

    public double getArea() {
        return this.area;
    }

    public void setColor(Color color) {
        this.color = color;
    }

    public abstract void computeArea();

    public void printShape() {
        System.out.println(this.shapeType + " " + color.fillColor());
    }
}
```

Tạo lớp các đối tượng hình(Refinded Abstraction) kế thừa từ Shape(Abstraction):

```java
package bridge;

public class Circle extends Shape{
    private double radius;
    public Circle(double r, Color color) {
        super(color);
        this.radius = r;
        this.shapeType = "Circle";
    }

    @Override
    public void computeArea() {
        this.area = Math.PI * this.radius * this.radius;
    }
}
```

```java
package bridge;

public class Square extends Shape {
    private double length;
    public Square(double length, Color color) {
        super(color);
```

```
        this.length = length;
        this.shapeType = "Square";
    }

    @Override
    public void computeArea() {
        this.area = this.length * this.length;
    }
}
```

Client: sử dụng Shape và Color để vẽ các hình với các màu khác nhau.

```
package bridge;

public class Client {
    public static void main(String[] args) {
        Color red = new RedColor();
        Color blue = new BlueColor();

        Shape redCircle = new Circle(2, new RedColor());
        redCircle.printShape();
        System.out.println("------------------------------------");

        Shape redSquare = new Square(3, red);
        redSquare.printShape();
        System.out.println("------------------------------------");

        Shape blueCircle = new Circle(5, blue);
        blueCircle.printShape();
        System.out.println("------------------------------------");

        Shape blueSquare = new Square(4, blue);
        blueSquare.printShape();
        blueSquare.setColor(red);
        blueSquare.printShape();
    }
}
```

Kết quả:

```
Circle filled with Red color.
------------------------------------
Square filled with Red color.
------------------------------------
Circle filled with Blue color.
------------------------------------
Square filled with Blue color.
Square filled with Red color.
```

8.8 Mối quan hệ với các Design Patterns khác

- Bridge, State, Strategy (ở mức độ nào đó là Adapter) có cấu trúc khá giống nhau. Chúng đều dựa trên bố cục, tức là ủy thác công việc cho các đối tượng khác.
- Có thể sử dụng Abstract Factory cùng với Bridge.
- Có thể kết hợp Builder với Bridge.

COMPOSITE

Composite Pattern là một mẫu thiết kế cấu trúc. Nó là một sự tổng hợp những thành phần có quan hệ với nhau để tạo ra thành phần lớn hơn. Nó cho phép thực hiện các tương tác với tất cả đối tượng trong mẫu tương tự nhau. Composite Pattern được sử dụng khi chúng ta cần xử lý một nhóm đối tượng tương tự theo cách xử lý một object. Composite Pattern sắp xếp các object theo cấu trúc cây để diễn giải một phần cũng như toàn bộ hệ thống phân cấp. Pattern này tạo một lớp chứa nhóm đối tượng của riêng nó. Lớp này cung cấp các cách để sửa đổi nhóm của cùng một object. Pattern này cho phép Client có thể viết code giống nhau để tương tác với composite object này, bất kể đó là một đối tượng riêng lẻ hay tập hợp các đối tượng.

9.1 Đặt vấn đề

Việc sử dụng Composite Pattern chỉ có ý nghĩa nếu mô hình cốt lõi của ứng dụng có thể được biểu diễn dưới dạng cấu trúc cây. Giả sử, chúng ta có hai đối tượng: Products và Boxes. Một chiếc hộp(Box) có thể chứa một số đồ vật(Products) cũng như một số hộp nhỏ hơn. Những cái hộp nhỏ hơn lại chứa một số đồ vật và một số hộp nhỏ hơn nữa,... tương tự như vậy. Chúng ta xây dựng một hệ thống đặt hàng sử dụng các lớp này. Đơn hàng có thể chứa các sản phẩm đơn giản mà không chứa bất kì bao bì nào, cũng như các hộp chứa các sản phẩm và các hộp khác. Vấn đề là làm sao để có thể tính được tổng giá của một đơn đặt hàng như vậy? Ngoài thực tế, chúng ta có thể mở các hộp rồi xem qua các sản phẩm và tính tổng giá đơn hàng nhưng trong chương trình nó không đơn giản như chạy một vòng lặp. Chúng ta phải biết các lớp trước của Products và Boxes mà chúng ta đang xét, mức lồng nhau của các hộp và các chi tiết khác. Những điều này khiến cho cách tiếp cận trực tiếp đến từng đồ vật trở nên quá khó khăn, thậm chí không thể?

9.2 Giải pháp

Composite Pattern cho phép chúng ta làm việc với Products và Boxes thông qua một giao diện chung khai báo một phương thức tính tổng giá.

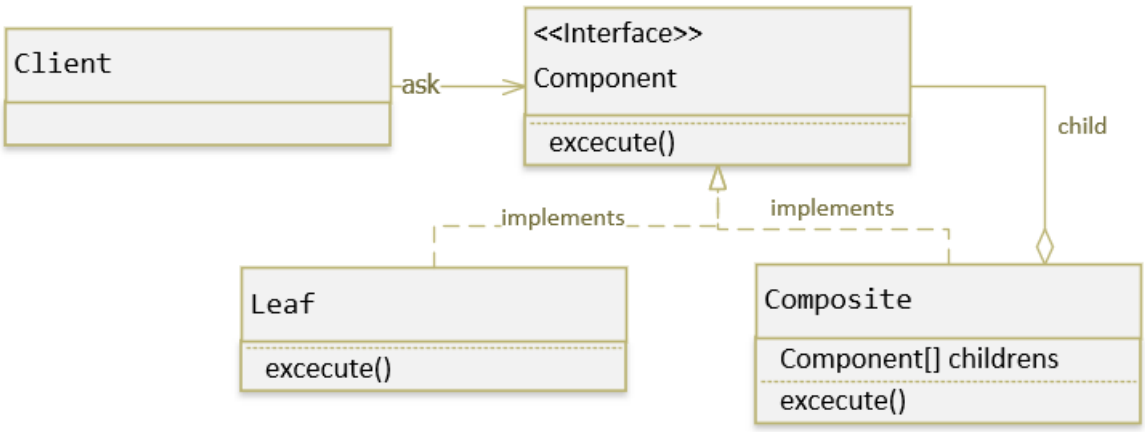
Với phương thức này, nó chỉ đơn giản là trả lại giá trị của sản phẩm đối với mỗi sản phẩm. Đối với một chiếp hộp, nó sẽ xem xét từng mục trong hộp sau đó trả về tổng giá trị của hộp đó. Nếu một trong những mặt hàng là một hộp nhỏ hơn thì nó cũng xem xét từng mục, tiếp tục như vậy đến khi tính được tổng giá các thành phần bên trong. Lợi ích lớn nhất của phương pháp này là chúng ta không cần quan tâm đến các lớp cụ thể của các đối tượng tạo nên cây. Chúng ta không cần biết một vật là một sản phẩm đơn giản hay là một chiếc hộp cầu kì. Chúng ta có thể xử lý tất cả chúng như nhau thông qua giao diện chung.

9.3 Khả năng áp dụng

Composite Pattern chỉ nên được áp dụng khi nhóm đối tượng phải hoạt động như một đối tượng duy nhất(theo cùng một cách).

Composite Pattern có thể được sử dụng để tạo ra một cấu trúc giống như cấu trúc cây.

9.4 Cấu trúc



Trong đó:

- **Base Component:** là một interface hoặc abstract class quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.
- **Composite:** lưu trữ tập hợp các Leaf và cài đặt các phương thức của Base Component. Composite cài đặt các phương thức được định

nghĩa trong interface Component bằng cách ủy nhiệm cho các thành phần con xử lý.

- **Leaf:** là lớp hiện thực (implements) các phương thức của Component. Nó là các object không có con.
- **Client:** sử dụng Base Component để làm việc với các đối tượng trong Composite.

9.5 Ưu nhược điểm

Ưu điểm:

- Cung cấp cùng một cách sử dụng đối với từng đối tượng riêng lẻ hoặc nhóm các đối tượng với nhau.

Nhược điểm:

- Khó có thể cung cấp một interface chung cho các lớp có các chức năng khác nhau quá nhiều. Trong vài trường hợp, chúng ta cần tổng thể hóa quá mức giao diện thành phần khiến nó trở nên khó hiểu.

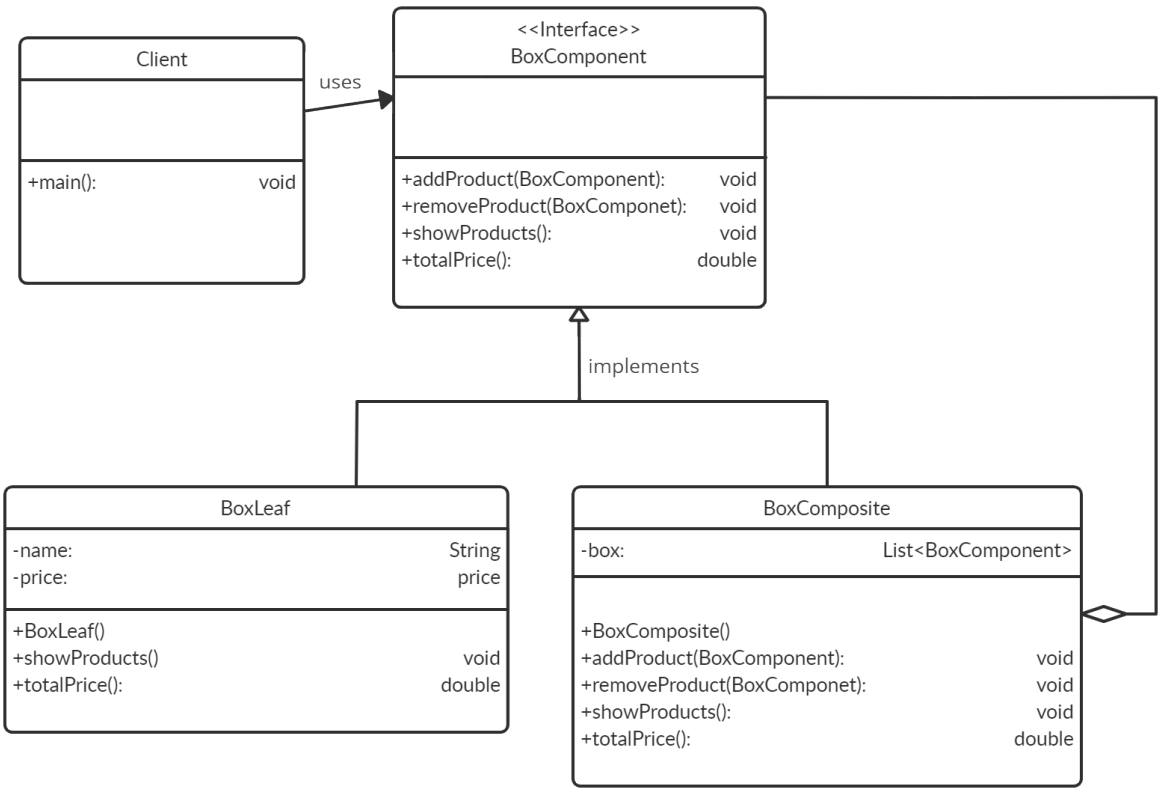
9.6 Cách triển khai

1. Đảm bảo rằng mô hình cốt lõi của chương trình có thể biểu diễn được dưới dạng cấu trúc cây. Cố gắng chia nhỏ nó thành các phần tử và vùng chứa khác.
2. Khai báo Component interface với các method phù hợp với tất cả các thành phần.
3. Tạo lớp lá(Leaf class) để biểu diễn các phần tử đơn giản. Một chương trình có thể có nhiều lớp lá khác nhau.
4. Tạo một lớp vùng chứa(container) để biểu diễn các thành phần phức tạp. Trong lớp này cung cấp một array để lưu trữ các tham chiếu đến các phần tử con. Mảng phải được khai báo với kiểu Component interface.
5. Khi triển khai phương thức của Component interface, lưu ý rằng một vùng chứa ủy thác mọi công việc cho các phần tử con.

6. Cuối cùng, xác định phương thức để thêm/xóa các phần tử con trong vùng chứa.

9.7 Code minh họa

Cài đặt Composite Pattern về hệ thống đặt hàng.



Tạo Component interface:

```
package composite;

public interface BoxComponent {
    void addProduct(BoxComponent item);
    void removeProduct(BoxComponent item);
    void showProducts();
    double totalPrice();
}
```

Tạo hộp chứa các sản phẩm(Box) - Composite implements Component:

```
package composite;

import java.util.*;

public class BoxComposite implements BoxComponent {
```

```
    private List<BoxComponent> box;

    public BoxComposite() {
        this.box = new ArrayList<BoxComponent>();
    }

    @Override
    public void addProduct(BoxComponent item) {
        this.box.add(item);
    }

    @Override
    public void removeProduct(BoxComponent item) {
        this.box.remove(item);
    }

    @Override
    public void showProducts() {
        for (BoxComponent product : box) {
            product.showProducts();
        }
    }

    @Override
    public double totalPrice() {
        double total = 0;
        for (BoxComponent product : box) {
            total += product.totalPrice();
        }
        return total;
    }
}
```

Lớp tạo ra các sản phẩm(Product) - Leaf implements Component:

```
package composite;

public class BoxLeaf implements BoxComponent {
    private String name;
    private double price;

    public BoxLeaf(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public void showProducts() {
        String s = String.format("BoxLeaf[name = %s, price = %.2f]"
                                 , this.name, this.price);
```

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

    @Override
    public double totalPrice() {
        return this.price;
    }

    @Override
    public void addProduct(BoxComponent item) {
        System.out.println("BoxLeaf can't contain item");
    }

    @Override
    public void removeProduct(BoxComponent item) {
        System.out.println("BoxLeaf can't contain item");
    }
}
```

Client:

```
package composite;

public class Client {
    public static void main(String[] args) {
        BoxLeaf      product1 = new BoxLeaf("Phone", 5);
        BoxComponent product2 = new BoxLeaf("Earphone", 3);
        BoxComponent product3 = new BoxLeaf("Charger", 3);
        BoxComponent product4 = new BoxLeaf("Radio", 2);
        BoxComponent product5 = new BoxLeaf("TV", 9);


        BoxComponent smallBox1 = new BoxComposite();
        smallBox1.addProduct(product1);
        smallBox1.addProduct(product2);


        BoxComponent smallBox2 = new BoxComposite();
        smallBox2.addProduct(product3);
        smallBox2.addProduct(product4);

        BoxComponent bigBox = new BoxComposite();
        bigBox.addProduct(product5);
        bigBox.addProduct(smallBox1);
        bigBox.addProduct(smallBox2);

        bigBox.showProducts();
        System.out.println("Price of all products: " + bigBox.totalPrice
    ());
    }
}
```

Kết quả:

```
BoxLeaf[name = TV, price = 9.00]
BoxLeaf[name = Phone, price = 5.00]
BoxLeaf[name = Earphone, price = 3.00]
BoxLeaf[name = Charger, price = 3.00]
BoxLeaf[name = Radio, price = 2.00]
Price of all products: 22.0
```

9.8 Mối quan hệ với các Design Patterns khác

- Có thể sử dụng Builder khi tạo các cây Composite phức tạp.
- Chain of Responsibility thường được sử dụng cùng với Composite.
- Có thể sử dụng Iterator để duyệt qua các cây Composite.
- Có thể sử dụng Visitor để thực hiện một thao tác trên toàn bộ cây Composite.
- Composite và Decorator có các sơ đồ cấu trúc tương tự.
- Các thiết kế sử dụng nhiều Composite và Decorator thường có thể được hưởng lợi lớn từ việc sử dụng Prototype vì nó cho phép sao chép các cấu trúc phức tạp thay vì phải xây dựng lại chúng từ đầu.

DECORATOR

Decorator Pattern là một Structural Design 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 (*decoratorclass*). Decorator pattern sử dụng composition thay vì inheritance (thừa kế) để mở rộng đối tượng.

10.1 Đặt vấn đề

Đôi khi chúng ta cần mở rộng một phương thức trong đối tượng, và cách thông thường là chúng ta sẽ kế thừa đối tượng đó. Việc này trong một vài trường hợp sẽ làm cho code trở lên phức tạp hơn và khó kiểm soát hơn. Để giải quyết vấn đề này, chúng ta có thể sử dụng Decorator Pattern.

10.2 Giải pháp

Decorator pattern hoạt động dựa trên một đối tượng đặc biệt, được gọi là decorator (hay wrapper). Nó có cùng một interface như một đối tượng mà nó cần bao bọc (wrap), vì vậy phía client sẽ không nhận thấy khi chúng ta đưa cho nó một wrapper thay vì đối tượng gốc.

Tất cả các wrapper có một trường để lưu trữ một giá trị của một đối tượng gốc. Hầu hết các wrapper khởi tạo trường đó với một đối tượng được truyền vào constructor của chúng.

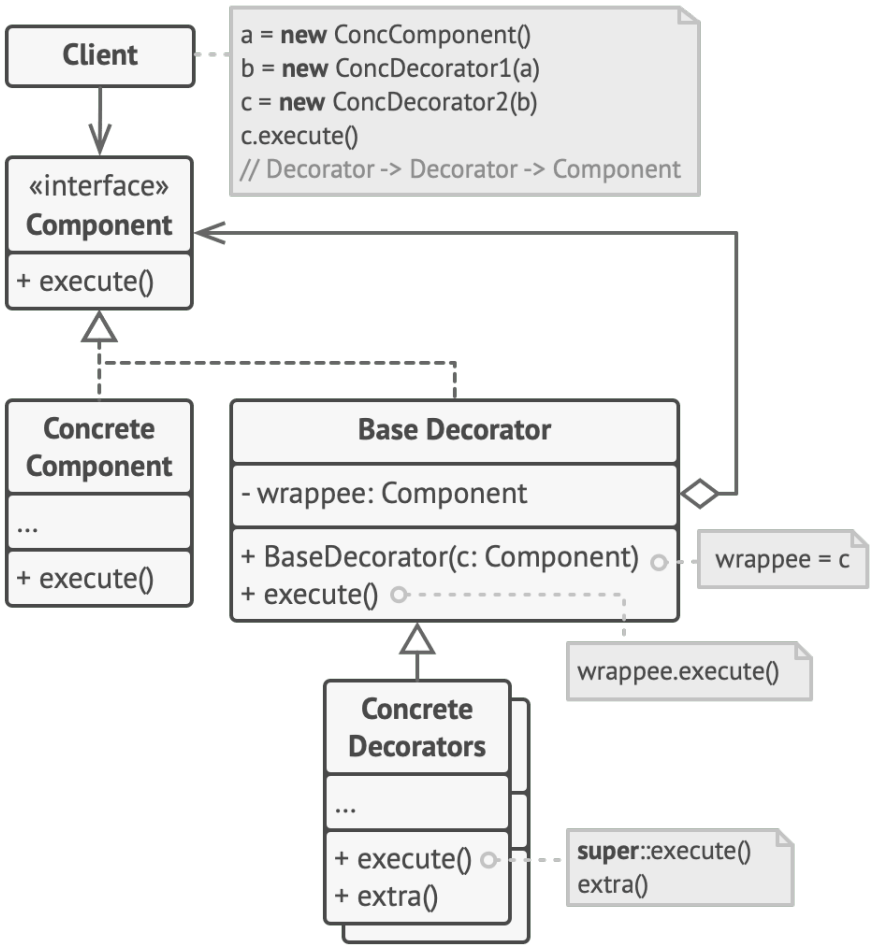
Như đã đề cập, wrapper có cùng interface với các đối tượng đích. Khi chúng ta gọi một phương thức decorator, nó thực hiện cùng một phương thức trong một đối tượng được wrap và sau đó thêm một cái gì đó (tính năng mới) vào kết quả, công việc này tùy thuộc vào logic nghiệp vụ.

10.3 Khả năng áp dụng

- 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.

10.4 Cấu trúc



Trong đó:

- **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 object thông qua Component Interface.

10.5 Ưu nhược điểm

Ưu điểm

- 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 chúng ta 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).

Nhược điểm

- Khó xóa một wrapper cụ thể khỏi wrapper stack.
- Khó triển khai decorator theo cách mà hành vi của nó không phụ thuộc vào thứ tự trong decorator stack.

10.6 Cách triển khai

1. Đảm bảo rằng business domain thể hiện như một thành phần chính với nhiều lớp tùy chọn trên đó.
2. Tìm ra những phương thức nào phổ biến cho cả thành phần chính và các lớp tùy chọn. Tạo Component Interface rồi khai báo các phương thức đó.

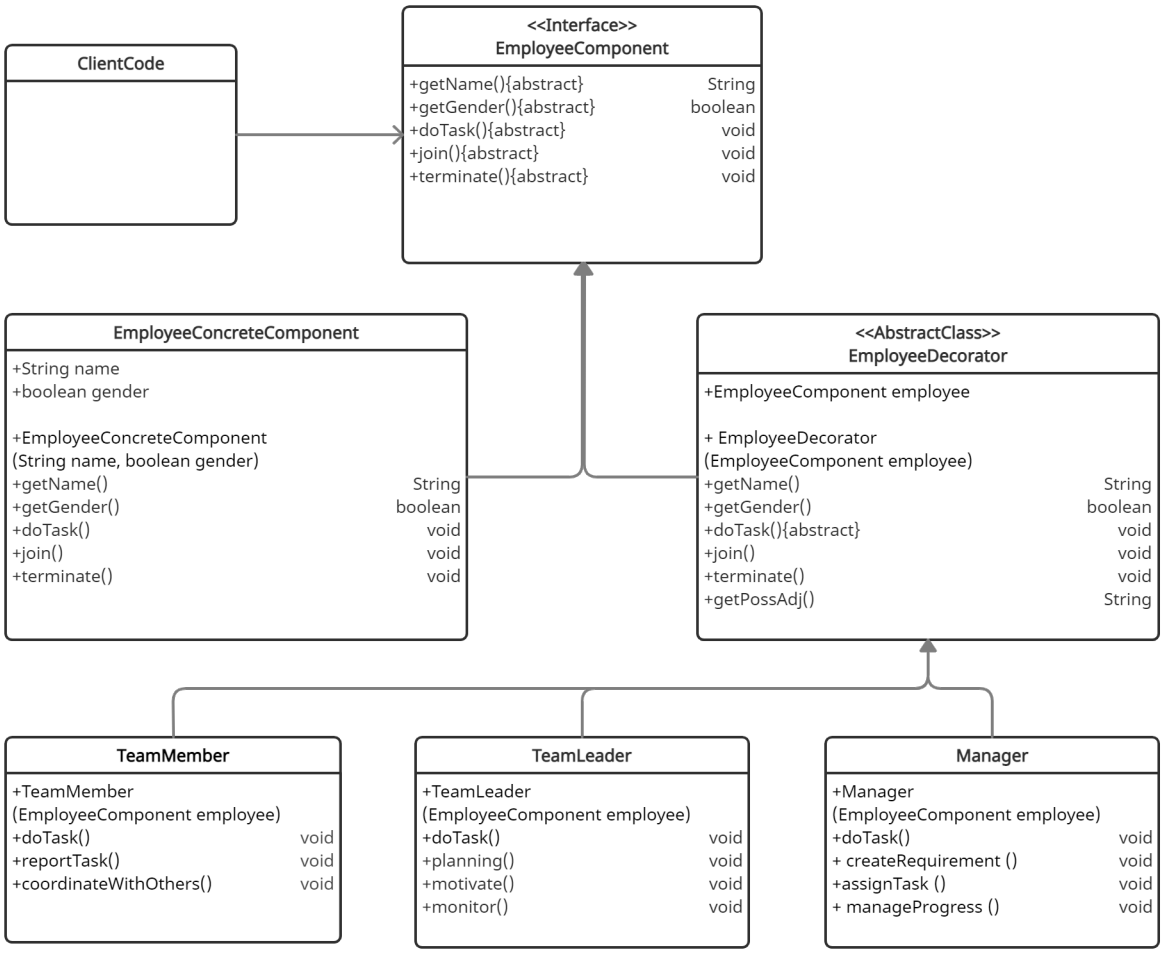
3. Tạo một lớp Concrete Component implements Component Interface và định nghĩa hành vi cơ sở trong đó.

4. Tạo Base Decorator class implements Component Interface. Nó phải có một trường để lưu trữ một tham chiếu đến wrapped object. Trường phải được khai báo với Component Interface để cho phép liên kết với các Concrete Component cũng như các decorator. Base Decorator phải delegate các wrapped object.

5. Tạo các Concrete Decorator bằng cách kế thừa từ Base Decorator.

10.7 Code minh họa

Chúng ta cần tạo một hệ thống quản lí công việc nhân sự trong một phòng, các nhân viên có thể đóng các chức vụ khác nhau, tuy nhiên một người có thể đóng nhiều chức vụ cùng lúc.



Tạo Component Interface và khai báo các phương thức

```java
package decorator;

import java.text.*;
import java.util.*;

public interface EmployeeComponent {

    String getName();

    boolean getGender();

    void doTask();

    void join(Date joinDate);

    void terminate(Date terminateDate);

    default String formatDate(Date theDate) {
        DateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        return sdf.format(theDate);
    }

    default void showBasicInformation() {
        System.out.println("-----------");
        System.out.println("The basic information of " + getName());
        Calendar date = Calendar.getInstance();

        terminate(date.getTime());

        date.add(Calendar.YEAR, -2);
        join(date.getTime());

    }
}
```

Tạo Concrete Component implements Component Interface

```java
package decorator;

import java.util.Date;

public class EmployeeConcreteComponent implements EmployeeComponent {

    private String name;
    private boolean gender; // true means male, false means female.

    public EmployeeConcreteComponent(String name, boolean gender) {
        this.name = name;
        this.gender = gender;
    }
```

```
14
15      @Override
16      public String getName() {
17          return name;
18      }
19
20      @Override
21      public boolean getGender() {
22          return gender;
23      }
24
25      @Override
26      public void join(Date joinDate) {
27          System.out.println(this.getName()
28                  + " joined on " + formatDate(joinDate));
29      }
30
31      @Override
32      public void terminate(Date terminateDate) {
33          System.out.println(this.getName()
34                  + " terminated on " + formatDate(terminateDate));
35      }
36
37      @Override
38      public void doTask() {
39          // Unassigned task
40      }
41
42  }
```

Tạo Base Decorator là abstract class implements Component Interface bằng cách sử dụng composition

```
1   package 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
```

```
    public boolean getGender() {
        return employee.getGender();
    }

    @Override
    public void join(Date joinDate) {
        employee.join(joinDate);
    }

    @Override
    public void terminate(Date terminateDate) {
        employee.terminate(terminateDate);
    }

    public String getPossAdj() {
        if (employee.getGender()) {
            return " his ";
        }
        return " her ";
    }
}
```

Tạo các Concrete Decorator, mỗi Concrete Decorator sẽ có các phương thức khác nhau

```
package decorator;

public class TeamMember extends EmployeeDecorator {

    protected TeamMember(EmployeeComponent employee) {
        super(employee);
    }

    public void reportTask() {
        System.out.println(this.employee.getName()
                + " is reporting" + getPossAdj() + "assigned tasks.");
    }

    public void coordinateWithOthers() {
        System.out.println(this.employee.getName()
                + " is coordinating with other members of"
                + getPossAdj() + "team.");
    }

    @Override
    public void doTask() {
        employee.doTask();
        reportTask();
        coordinateWithOthers();
    }
}
```

```java
package decorator;

public class TeamLeader extends EmployeeDecorator {

    protected TeamLeader(EmployeeComponent employee) {
        super(employee);
    }

    public void planing() {
        System.out.println(this.employee.getName() + " is planing.");
    }

    public void motivate() {
        System.out.println(this.employee.getName()
                + " is motivating" + getPossAdj() + "members.");
    }

    public void monitor() {
        System.out.println(this.employee.getName()
                + " is monitoring" + getPossAdj() + "members.");
    }

    @Override
    public void doTask() {
        employee.doTask();
        planing();
        motivate();
        monitor();
    }
}
```

```java
package decorator;

public class Manager extends EmployeeDecorator {

    protected Manager(EmployeeComponent employee) {
        super(employee);
    }

    public void createRequirement() {
        System.out.println(this.employee.getName()
                + " is create requirements.");
    }

    public void assignTask() {
        System.out.println(this.employee.getName()
                + " is assigning tasks.");
    }

    public void manageProgress() {
        System.out.println(this.employee.getName()
```

```
                + " is managing the progress.");
    }

    @Override
    public void doTask() {
        employee.doTask();
        createRequirement();
        assignTask();
        manageProgress();
    }
}
```

Tạo Client Code

Client Code tạo ra đối tượng EmployeeComponent thông qua class EmployeeConcreteComponent. Khi muốn thêm các chức vụ cho đối tượng, ta dùng các class ConcreteDecorator với base là đối tượng EmployeeComponent ban đầu đã tạo hoặc đối tượng EmployeeComponent được tạo ra qua các ConcreteDecorator khác (wrapper object). Như vậy, từ một đối tượng được tạo ra, ta có thể biến nó đóng nhiều chức vụ trong công ty.

```
package decorator;

public class TestMain {
    public static void main(String[] args) {
        EmployeeComponent employee = new EmployeeConcreteComponent("An
   Phung", false);

        System.out.println("NORMAL EMPLOYEE: ");
        employee.showBasicInformation();
        employee.doTask();

        System.out.println("\nTEAM MEMBER: ");
        EmployeeComponent teamMember = new TeamMember(employee);
        teamMember.showBasicInformation();
        teamMember.doTask();

        System.out.println("\nTEAM LEADER: ");
        EmployeeComponent teamLeader = new TeamLeader(employee);
        teamLeader.showBasicInformation();
        teamLeader.doTask();

        System.out.println("\nMANAGER: ");
        EmployeeComponent manager = new Manager(employee);
        manager.showBasicInformation();
        manager.doTask();

        System.out.println("\nTEAM LEADER AND MANAGER: ");
        EmployeeComponent teamLeaderAndManager = new Manager(teamLeader)
   ;
```

```
28          teamLeaderAndManager.showBasicInformation();
29          teamLeaderAndManager.doTask();
30      }
31  }
```

Kết quả:

```
NORMAL EMPLOYEE:
-----------
The basic information of An Phung
An Phung terminated on 06/01/2022
An Phung joined on 06/01/2020

TEAM MEMBER:
-----------
The basic information of An Phung
An Phung terminated on 06/01/2022
An Phung joined on 06/01/2020
An Phung is reporting her assigned tasks.
An Phung is coordinating with other members of her team.

TEAM LEADER:
-----------
The basic information of An Phung
An Phung terminated on 06/01/2022
An Phung joined on 06/01/2020
An Phung is planing.
An Phung is motivating her members.
An Phung is monitoring her members.

MANAGER:
-----------
The basic information of An Phung
An Phung terminated on 06/01/2022
An Phung joined on 06/01/2020
An Phung is create requirements.
An Phung is assigning tasks.
An Phung is managing the progress.

TEAM LEADER AND MANAGER:
-----------
The basic information of An Phung
An Phung terminated on 06/01/2022
An Phung joined on 06/01/2020
An Phung is planing.
```

```
An Phung is motivating her members.
An Phung is monitoring her members.
An Phung is create requirements.
An Phung is assigning tasks.
An Phung is managing the progress.
```

10.8 Mối quan hệ với các Design Patterns khác

- Adapter: Decorator khác với adapter ở chỗ decorator chỉ thay đổi trách nhiệm của một đối tượng chứ không phải giao diện của nó.
- Composite: Decorator có thể xem là một degenerate Composite với chỉ một component. Tuy nhiên, decorator thêm các trách nhiệm bổ sung - nó không dành cho việc tập hợp object.
- Strategy: Decorator cho phép chúng ta thay đổi "da" của một đối tượng, Strategy cho phép chúng ta thay đổi "ruột". Đây là hai cách thay thế để thay đổi một đối tượng.

FACADE

Cung cấp một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con(subsystem). Facade Pattern định nghĩa một giao diện ở một cấp độ cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này. Nó cho phép các đối tượng truy cập trực tiếp giao diện chung này để giao tiếp với các giao diện có trong hệ thống con. Mục tiêu là che giấu các hoạt động phức tạp bên trong hệ thống con, làm cho hệ thống con dễ sử dụng hơn.

11.1 Đặt vấn đề

Giả sử chúng ta có chuỗi các hành động được thực hiện theo thứ tự, và các hành động này lại được yêu cầu ở nhiều nơi trong phạm vi ứng dụng, vậy mỗi lúc cần dùng đến nó chúng ta lại phải copy-paste hoặc viết lại đoạn code đó vào những nơi cần sử dụng trong chương trình. Nhưng vấn đề là nếu chúng ta làm xong và nhận ra cần phải thay đổi lại cấu trúc và code xử lý trong hầu hết chuỗi hành động đó, vậy sẽ phải giải quyết như thế nào? Đây là mấu chốt vấn đề, chúng ta sẽ phải dò lại đoạn code đó ở tất cả các nơi rồi sửa lại nó. Điều này quá tốn thời gian và khiến chúng ta mất đi sự kiểm soát code của mình, xảy ra nguy cơ phát sinh lỗi.

11.2 Giải pháp

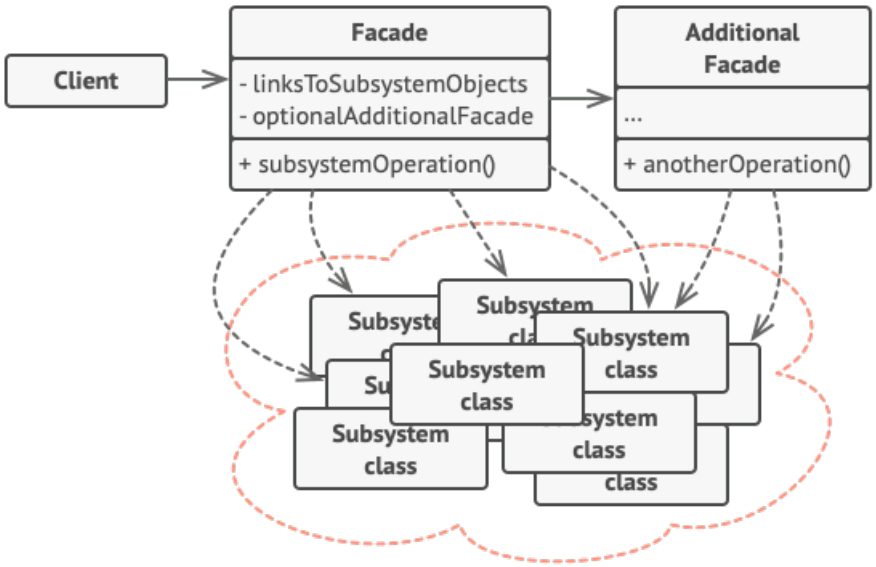
Những gì chúng ta cần làm là thiết kế một Facade và trong đó phương thức facade sẽ xử lý các đoạn code dùng đi dùng lại. Từ xu hướng quan điểm trên, chúng ta chỉ cần gọi Facade để thực thi các hành động dựa trên các parameters(tham số) được cung cấp. Bây giờ nếu chúng ta cần bất kỳ thay đổi nào trong quá trình trên, công việc sẽ đơn giản hơn rất nhiều, chỉ cần thay đổi các xử lý trong phương thức facade và mọi thứ sẽ được đồng bộ thay vì thực hiện sự thay đổi ở những nơi sử dụng cả chuỗi các mã code đó.

11.3 Khả năng áp dụng

Sử dụng Facade pattern khi:

- Hệ thống có rất nhiều lớp làm người sử dụng rất khó để có thể hiểu được quy trình xử lý của chương trình. Và khi có rất nhiều hệ thống con mà mỗi hệ thống con đó lại có những giao diện riêng lẻ nên rất khó cho việc sử dụng phân phối. Khi đó có thể sử dụng Facade Pattern để tạo ra một giao diện đơn giản cho người sử dụng một hệ thống phức tạp.
- Người sử dụng phụ thuộc nhiều vào các lớp cài đặt. Việc áp dụng Facade Pattern sẽ tách biệt hệ thống của người dùng và các hệ thống con khác, do đó tăng khả năng độc lập của hệ thống con, dễ chuyển đổi nâng cấp trong tương lai.
- Muốn phân lớp các hệ thống con.
- Muốn che giấu tính phức tạp trong hệ thống con đối với phía Client.

11.4 Cấu trúc



Trong đó:

- **Facade:** biết rõ lớp của hệ thống con nào đảm nhận việc đáp ứng yêu cầu của Client, sẽ chuyển yêu cầu của Client đến các đối tượng của hệ thống con tương ứng.
- **Additional Facade:** một lớp Facade bổ sung có thể được tạo để ngăn ảnh hưởng xấu đến một Facade đơn lẻ với các tính năng không

liên quan, có thể làm nó trở nên phức tạp. Lớp Facade bổ sung có thể được sử dụng bởi Client và các Facade khác.

- **Subsystems:** cài đặt các chức năng của hệ thống con, xử lý công việc được gọi bởi Facade. Các lớp này không cần biết Facade và không tham chiếu đến nó.

- **Client:** đối tượng sử dụng Facade để tương tác với các subsystems. Đối tượng Facade thường là Singleton vì chỉ cần duy nhất một đối tượng Facade.

11.5 Ưu nhược điểm

Ưu điểm:

- Giúp hệ thống trở nên đơn giản hơn trong việc sử dụng và hiểu nó vì một mẫu Facade có các phương thức tiện lợi cho các tác vụ của chúng.

- Giảm sự phụ thuộc của code bên ngoài với hiện thực bên trong của thư viện vì hầu hết các code đều dùng Facade, điều đó cho phép sự linh động trong phát triển các hệ thống.

Nhược điểm:

- Class Facade có thể trở lên quá lớn, làm quá nhiều nhiệm vụ với nhiều hàm chức năng trong nó. Điều này sẽ dễ bị phá vỡ các quy tắc trong SOLID(nguyên lý lập trình hướng đối tượng).

11.6 Cách triển khai

1. Xem xét liệu có thể cung cấp một interface đơn giản hơn những gì một hệ thống con hiện tại đã cung cấp hay không.

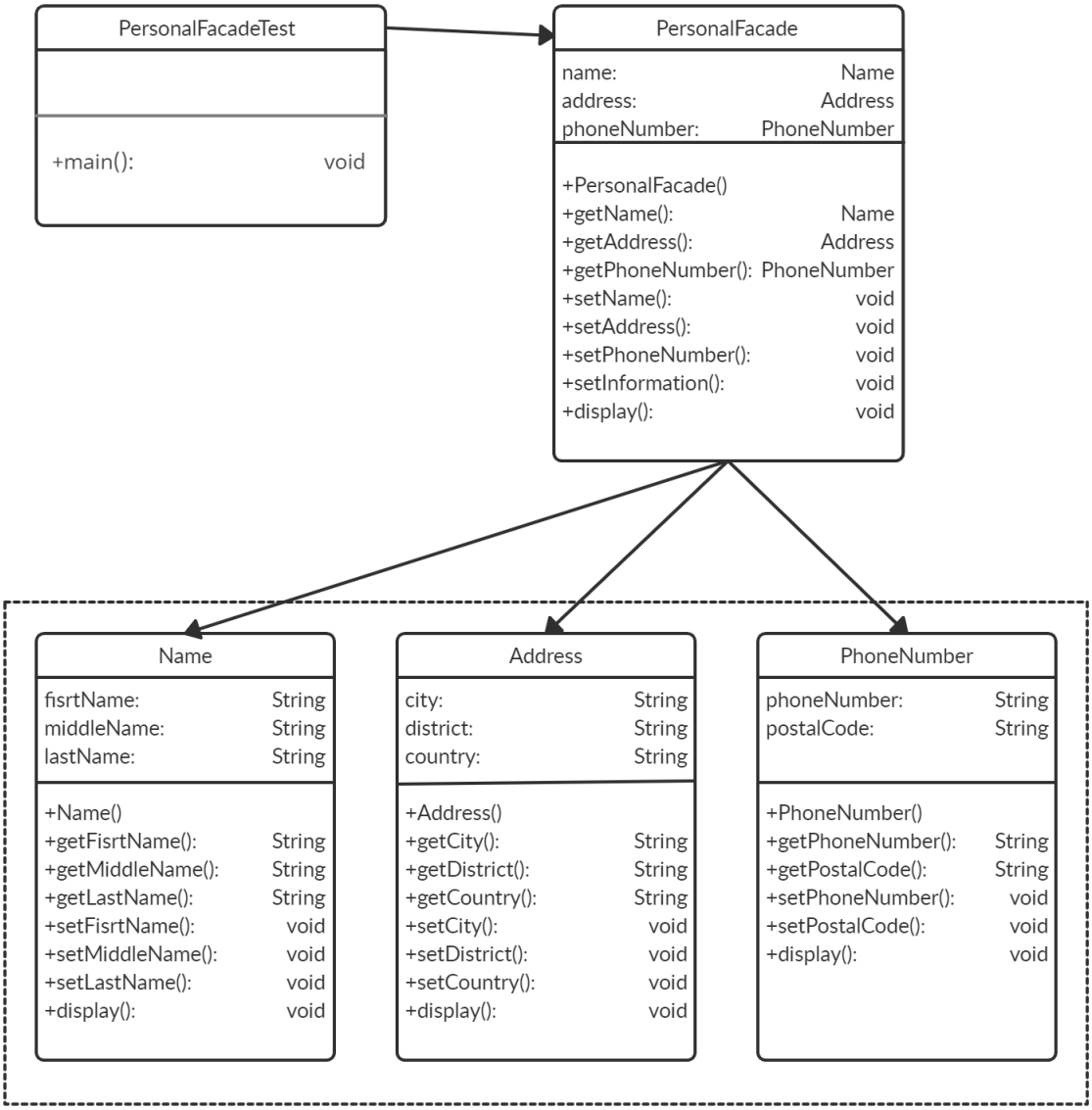
2. Khai báo và triển khai interface này trong một lớp Facade mới. Facade phải chuyển hướng các lệnh gọi từ Client code sang các đối tượng thích hợp của hệ thống con. Facade phải chịu trách nhiệm khởi tạo hệ thống con và quản lí vòng đời tiếp theo của nó trừ khi Client code đã thực hiện điều này.

3. Để tận dụng toàn bộ lợi ích của facade pattern, hãy để Client code chỉ giao tiếp với hệ thống con thông qua Facade. Điều này giúp Client code được bảo vệ khỏi bất kì thay đổi nào trong hệ thống con.

4. Nếu Facade trở nên quá lớn, hãy xem xét trích xuất một phần hành vi của nó sang một Facade class mới.

11.7 Code minh họa

Quản lí thông tin cá nhân.



Tạo các lớp subsystems:

```
1 package facade;
2
```

```
public class Name {
    String firstName;
    String middleName;
    String lastName;

    public Name() {
    }

    public Name(String firstName, String middleName, String lastName) {
        this.firstName = firstName;
        this.middleName = middleName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getMiddleName() {
        return middleName;
    }

    public void setMiddleName(String middleName) {
        this.middleName = middleName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public void display() {
        System.out.println("Name: " + this.firstName + " " + this.
   middleName + " " + this.lastName);
    }
}
```

```
package facade;

public class Address {
    String city;
    String district;
    String country;

```

```
    public Address() {
    }

    public Address(String district, String city, String country) {
        this.district = district;
        this.city = city;
        this.country = country;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getDistrict() {
        return district;
    }

    public void setDistrict(String district) {
        this.district = district;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }

    public void display() {
        System.out.println("Address: " + this.district + " - " + this.
   city + " - " + this.country);
    }
}
```

```
package facade;

public class PhoneNumber {
    String phoneNumber;
    String postalCode;

    public PhoneNumber() {
    }

    public PhoneNumber(String phoneNumber, String postalCode) {
        this.phoneNumber = phoneNumber;
        this.postalCode = postalCode;
```

```
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public String getPostalCode() {
        return postalCode;
    }

    public void setPostalCode(String postalCode) {
        this.postalCode = postalCode;
    }

    public void display() {
        System.out.println("PhoneNumber: " + this.postalCode + " " +
    this.phoneNumber);
    }
}
```

Tạo lớp Facade:

```
package facade;

public class PersonalFacade {
    Name name;
    Address address;
    PhoneNumber phoneNumber;

    public PersonalFacade() {

    }

    public PersonalFacade(Name name, Address address, PhoneNumber
    phoneNumber) {
        this.name = name;
        this.address = address;
        this.phoneNumber = phoneNumber;
    }

    public Name getName() {
        return name;
    }

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

```
25
26      public Address getAddress() {
27          return address;
28      }
29
30      public void setAddress(Address address) {
31          this.address = address;
32      }
33
34      public PhoneNumber getPhoneNumber() {
35          return phoneNumber;
36      }
37
38      public void setPhoneNumber(PhoneNumber phoneNumber) {
39          this.phoneNumber = phoneNumber;
40      }
41
42      public void display() {
43          this.name.display();
44          this.address.display();
45          this.phoneNumber.display();
46          System.out.println("-----------------------------------------"
    );
47      }
48
49      public void setInformation(Name name, Address address, PhoneNumber
    phoneNumber) {
50          this.address = address;
51          this.name = name;
52          this.phoneNumber = phoneNumber;
53      }
54  }
```

Client:

```
1  package facade;
2
3  public class PersonalFacadeTest {
4      public static void main(String[] args) {
5          Name name = new Name("Nguyen", "Ngoc", "Tinh");
6          Address address = new Address("Hung Ha", "Thai Binh", "Viet Nam"
    );
7          PhoneNumber phoneNumber = new PhoneNumber("0362xxxxxx", "034
    xxxxxxxxx");
8
9          PersonalFacade personal = new PersonalFacade(name, address,
    phoneNumber);
10         personal.display();
11
12         PersonalFacade personal1 = new PersonalFacade();
13         personal1.setInformation(name, address, phoneNumber);
```

```
14          personal1.display();
15      }
16  }
```

Kết quả:

```
Name: Nguyen Ngoc Tinh
Address: Hung Ha - Thai Binh - Viet Nam
PhoneNumber: 034xxxxxxxx 0362xxxxxx
-----------------------------------------
Name: Nguyen Ngoc Tinh
Address: Hung Ha - Thai Binh - Viet Nam
PhoneNumber: 034xxxxxxxx 0362xxxxxx
-----------------------------------------
```

11.8 Mối quan hệ với các Design Patterns khác

- Abstract Factory có thể là một giải pháp thay thế cho Facade khi chỉ muốn ẩn cách các đối tượng subsystems được tạo ra từ Client code.
- Facade và Mediator có những mục đích tương tự nhau: cố gắng tổ chức các lớp được kết hợp chặt chẽ với nhau.
- Một lớp Facade thường được chuyển đổi thành một Singleton.

OBSERVER

Observer pattern là một mẫu thiết kế hành vi cho phép chúng ta xác định cơ chế đăng kí để thông báo cho nhiều đối tượng về bất kỳ sự kiện nào xảy ra với đối tượng mà chúng đang quan sát. Hay nói cách khác, Observer pattern xác định một phụ thuộc một-nhiều giữa các đối tượng để khi một đối tượng thay đổi trạng thái, tất cả các phụ thuộc của nó được thông báo và tự động cập nhật.

12.1 Đặt vấn đề

Giả sử chúng ta có một bảng tính excel với nhiều trang tính chứa các dữ liệu cần thống kê. Ta có thể tạo ra vô số biểu đồ sử dụng dữ liệu ở các trang tính đó để hiển thị ra kết quả thống kê. Khi ta thay đổi dữ liệu ở một trang tính thì các biểu đồ sử dụng dữ liệu ấy cũng phải được cập nhật để có số liệu thống kê chính xác. Ta có thể thấy là số lượng biểu đồ có thể dữ liệu ở một trang tính là không giới hạn. Ta không thể tìm từng biểu đồ và cập nhật dữ liệu mới cho từng biểu đồ đó .

12.2 Giải pháp

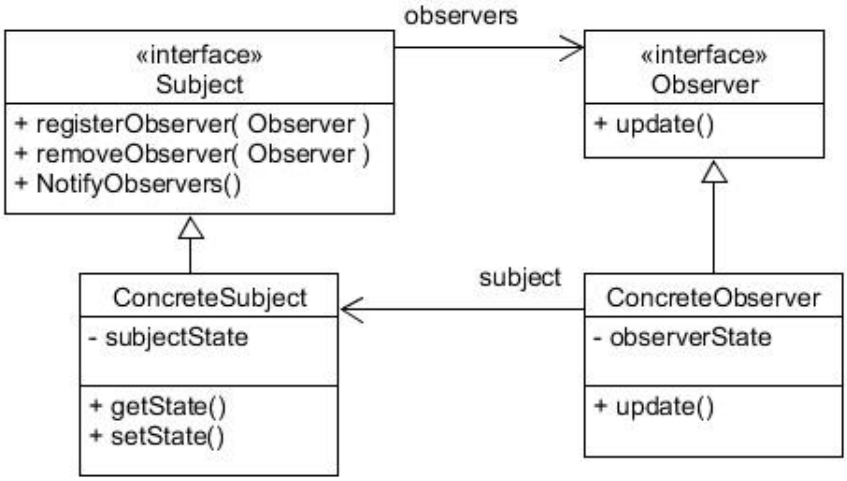
Trong tình huống này, hướng giải quyết là sử dụng Observer pattern. Trang tính đóng vai trò là **subject**, còn các biểu đồ chính là các **observer**. Mỗi khi trang tính được cập nhật dữ liệu thì ta sẽ gọi cập nhật đến các biểu đồ phụ thuộc vào dữ liệu của trang tính đó.

12.3 Khả năng áp dụng

Observer pattern được áp dụng khi:

- Các đối tượng có mối quan hệ một-nhiều. Trong đó một đối tượng thay đổi và muốn thông báo cho tất cả các object liên quan biết về sự thay đổi đó.
- 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 và cần thay đổi những đối tượng nào.

12.4 Cấu trúc



Trong đó:

- **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.
- **Concrete Subject:** cài đặt cái phương thức của *Subject*, lưu trữ trạng thái danh sách các *Concrete Observer*, gửi thông báo đến các *Observer* của nó khi có sự thay đổi trạng thái.
- **Concrete Observer:** cài đặt các phương thức của *Observer*, lưu trữ trạng thái của *Subject* thực thi việc cập nhật để giữ trạng thái đồng nhất với *Subject* gửi thông báo đến.

Tóm lại, sự tương tác giữa *Subject* và các *Observer* như sau: mỗi khi *Subject* có sự thay đổi trạng thái, nó sẽ duyệt qua danh sách các *Observer* của nó và gọi phương thức cập nhật trạng thái ở từng *Observer*, có thể truyền chính nó vào phương thức để các *Observer* có thể lấy ra trạng thái của nó và xử lý.

12.5 Ưu nhược điểm

Ưu điểm:

- Dễ dàng mở rộng với ít sự thay đổi: pattern 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 cần sửa đổi Subject hoặc Observer khác. Vì thế nó đảm bảo nguyên tắc Open/Closed Principle.
- Một đối tượng có thể thông báo đến một số lượng không giới hạn các đối tượng khác.

Nhược điểm:

- Các Observer được thông báo theo thứ tự ngẫu nhiên.

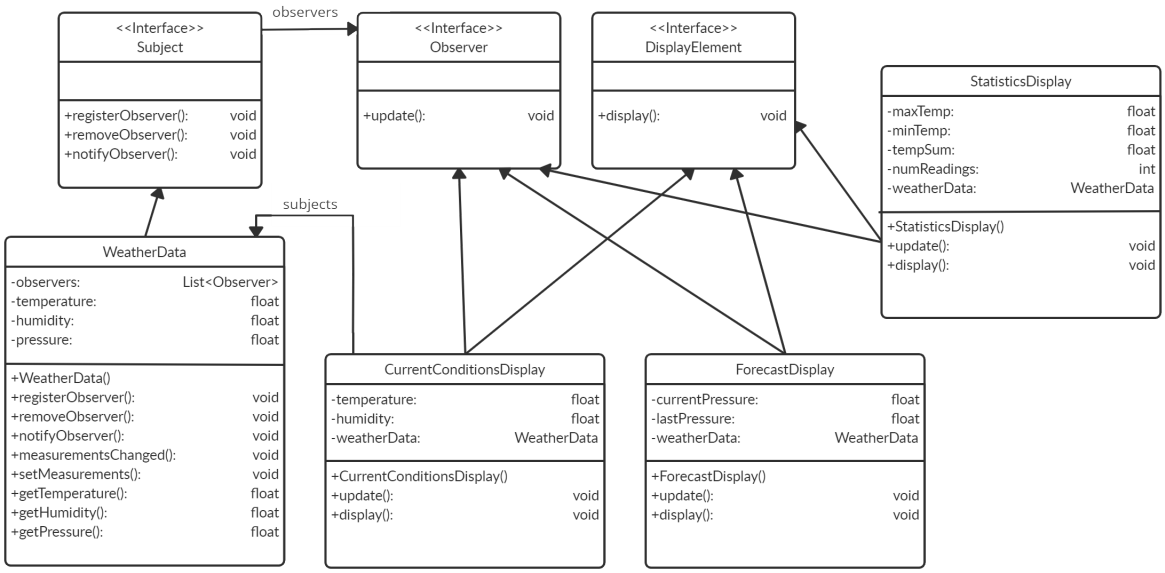
12.6 Cách triển khai

1. Xem xét vấn đề và chia nó thành hai phần: phần có chức năng cốt lõi, độc lập sẽ đóng vai trò là **Subject**, phần còn lại là các **Observer**.
2. Khai báo interface của Observer, phải có ít nhất một phương thức update.
3. Khai báo interface của Subject với cặp phương thức thêm Observer và xóa Observer khỏi danh sách. Subject chỉ được làm việc với các Observer thông qua Observer interface.
4. Quyết định nơi đặt danh sách Observer thực tế và triển khai các phương thức Observer.
5. Tạo các Subject class cụ thể. Khi có điều gì thay đổi thì Subject phải thông báo đến tất cả các Observer.
6. Thực hiện các phương thức cập nhật trong các lớp Observer cụ thể.
7. Client phải tạo tất cả các Observer cần thiết và đăng kí chúng với Subject phù hợp.

12.7 Code minh họa

Chúng ta có chương trình theo dõi thời tiết. Có 3 thành phần chính trong chương trình đó là: **WeatherStation**(thu thập dữ liệu thời tiết thực tế), **WeatherData**(theo dõi dữ liệu đến từ WeatherStation và cập nhật

màn hình) và màn hình hiển thị điều kiện thời tiết hiện tại. Chương trình này sẽ sử dụng đối tượng **WeatherData** để hiển thị điều kiện hiện tại(nhiệt độ, độ ẩm, áp suất), thống kê thời tiết và dự báo.



Tạo interface Subject và Observer, tất cả các thành phần thời tiết sẽ thực hiện giao diện Observer:

```
package observer;

public interface Subject {
  public void registerObserver(Observer o);
  public void removeObserver(Observer o);
  public void notifyObservers();
}
```

```
package observer;

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

Tạo Concrete Subject, WeatherData sẽ implements Subject interface:

```
package observer;

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 measurementsChanged() {
    notifyObservers();
  }

  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;
  }
}
```

Tạo interface DisplayElement cho tất cả các thành phần hiển thị:

```
package observer;

public interface DisplayElement {
```

```
  public void display();
}
```

Lớp hiển thị dự báo thời tiết:

```
package observer;

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");
    }
  }
}
```

Lớp theo dõi và hiển thị số liệu thống kê như Avg/Min/Max:

```
package observer;

public class StatisticsDisplay implements Observer, DisplayElement {
  private float maxTemp = 0.0f;
  private float minTemp = 200;
  private float tempSum= 0.0f;
  private int numReadings;
  private WeatherData weatherData;

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

```

```
15    public void update(float temp, float humidity, float pressure) {
16      tempSum += temp;
17      numReadings++;
18
19      if (temp > maxTemp) {
20        maxTemp = temp;
21      }
22
23      if (temp < minTemp) {
24        minTemp = temp;
25      }
26
27      display();
28    }
29
30    public void display() {
31      System.out.println("Avg/Max/Min temperature = " + (tempSum /
       numReadings)
32          + "/" + maxTemp + "/" + minTemp);
33    }
34  }
```

Lớp hiển thị các phép đo hiện tại từ đối tượng WeatherData:

```
1  package observer;
2
3  public class CurrentConditionsDisplay implements Observer,
       DisplayElement {
4    private float temperature;
5    private float humidity;
6    private WeatherData weatherData;
7
8    public CurrentConditionsDisplay(WeatherData 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 }
```

Lớp WeatherStation thu thập dữ liệu thời tiết:

```java
package observer;

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);

        System.out.println("--------------------------------");
    weatherData.setMeasurements(80, 65, 30.4f);
        System.out.println("--------------------------------");
    weatherData.setMeasurements(82, 70, 29.2f);
        System.out.println("--------------------------------");
    weatherData.setMeasurements(78, 90, 29.2f);
        System.out.println("--------------------------------");
    weatherData.removeObserver(forecastDisplay);
    weatherData.setMeasurements(62, 90, 28.1f);
  }
}
```

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
```

12.8 Mối quan hệ với các Design Patterns khác

- Chain of Responsibility, Command, Mediator và Observer đều giải quyết vấn đề gửi-nhận các yêu cầu.

- Mediator và Observer có những điểm tương đồng. Trong một vài trường hợp chúng ta có thể áp dụng đồng thời cả hai pattern này.

STRATEGY

Strategy (Policy) là một Behavioral Design Pattern. Nó cho phép định nghĩa tập hợp các thuật toán, đóng gói từng thuật toán lại, và dễ dàng thay đổi linh hoạt các thuật toán bên trong object. Strategy cho phép thuật toán biến đổi độc lập khi người dùng sử dụng chúng.

13.1 Đặt vấn đề

Có một vài trường hợp, các lớp chỉ khác nhau về hành vi của chúng. Điều này dẫn đến phải thay đổi thuật toán ở những hành vi đã có và có thể khiến chương trình trở nên khó bảo trì và nhiều khi còn gây nên những bug trên những phần đang hoạt động tốt. Strategy sinh ra để xử lý trường hợp này.

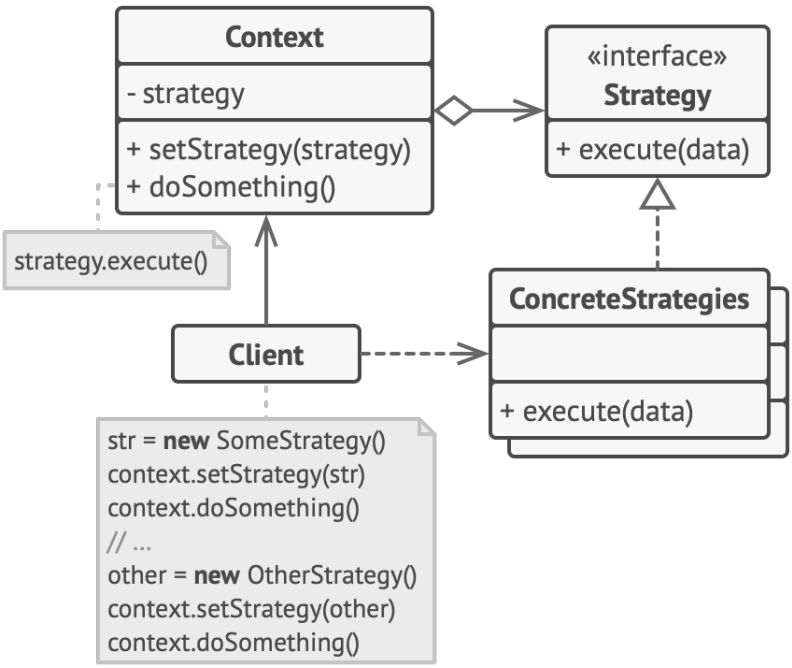
13.2 Giải pháp

Strategy Pattern tách rời phần xử lý một hành vi cụ thể ra khỏi đối tượng. Sau đó tạo ra một tập hợp các thuật toán để thực thi hành vi đó và lựa chọn thuật toán nào mà chúng ta thấy đúng đắn nhất. Mẫu thiết kế này thường được sử dụng để thay thế cho Inheritance, khi muốn chấm dứt việc theo dõi và chỉnh sửa một hành vi qua nhiều lớp con.

13.3 Khả năng áp dụng

- Muốn sử dụng các biến thể khác nhau của một xử lý trong một đối tượng và có thể chuyển đổi giữa các xử lý trong runtime.
- Khi có nhiều lớp tương đương chỉ khác cách chúng thực thi một vài hành vi.
- Khi muốn tách biệt business logic của một lớp khỏi implementation details của các xử lý.
- Khi lớp có toán tử điều kiện lớn chuyển đổi giữa các biến thể của cùng một xử lý.

13.4 Cấu trúc



Trong đó:

- **Context:** Class sử dụng các strategy object và chỉ giao tiếp với các Strategy Object thông qua interface.
- **Strategy:** Cung cấp một interface chung cho context giao tiếp với các Strategy Object.
- **Concrete Strategy:** Implement các thuật toán khác nhau cho context sử dụng.
- **Client:** Có trách nhiệm tạo ra các Strategy Object và truyền vào cho context sử dụng.

13.5 Ưu nhược điểm

Ưu điểm

- Có thể thay thế các thuật toán linh hoạt với nhau.
- Tách biệt phần thuật toán khỏi phần sử dụng thuật toán.
- Có thể thay thế việc kế thừa bằng việc encapsulate thuật toán.

- Open/Closed Principle: Khi thay đổi thuật toán hoặc khi thêm mới thuật toán, không cần thay đổi code phần context.

Nhược điểm

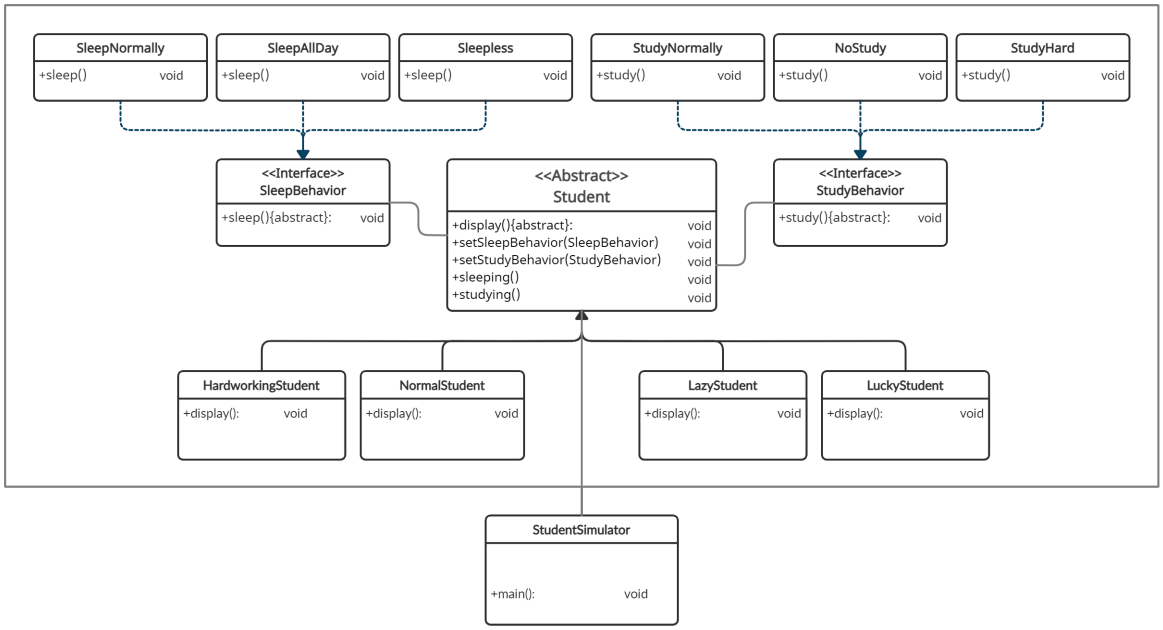
- Không nên áp dụng nếu chỉ có một vài thuật toán hiếm khi thay đổi.
- Client phải nhận biết được sự khác biệt giữa các strategy để chọn ra một strategy phù hợp.

13.6 Cách triển khai

1. Xác định logic cần tách ra một nhóm strategy riêng từ class gốc.
2. Tạo Strategy Interface.
3. Tạo các concrete class của strategy interface trên.
4. Ở lớp gốc, tạo một reference tới strategy interface (không phải concrete class). Cung cấp một cách thức để khởi tạo concrete class cho class gốc.

13.7 Code minh họa

Giả sử như chúng ta tạo ra chương trình diễn tả một sinh viên với hai hành vi là Study và Sleep.



Tạo Strategy

Hai hành vi là Sleep và Study được cung cấp cho đối tượng Student

```
package strategy;

// Encapsulated study behaviors
public interface StudyBehavior {
    public void study();
}
```

```
package strategy;

// Encapsulated sleep behaviors
public interface SleepBehavior {
    public void sleep();
}
```

Tạo các Concrete Strategy

Đối với hành vi Study:

```
package strategy;

public class StudyHard implements StudyBehavior {
    @Override
    public void study() {
        System.out.println("I love studying.");
    }
}
```

```
package strategy;

public class StudyNormally implements StudyBehavior {
    @Override
    public void study() {
        System.out.println("I will study until I am sleepy.");
    }
}
```

```
package strategy;

public class NoStudy implements StudyBehavior {
    @Override
    public void study() {
        System.out.println("I have no time for studying.");
    }
}
```

Đối với hành vi Sleep:

```
package strategy;

public class Sleepless implements SleepBehavior {
    @Override
```

```
    public void sleep() {
        System.out.println("I can't sleep until I finish my homework.");
    }
}
```

```
package strategy;

public class SleepAllDay implements SleepBehavior {
    @Override
    public void sleep() {
        System.out.println("I sleep 15 hours per day.");
    }
}
```

```
package strategy;

public class SleepNormally implements SleepBehavior {
    @Override
    public void sleep() {
        System.out.println("I sleep 8 hours per day.");
    }
}
```

Tạo Context: đối tượng abstract có hai hành vi trên

```
package strategy;

public abstract class Student {
    StudyBehavior studyBehavior;
    SleepBehavior sleepBehavior;

    public Student() {
    }

    public void setStudyBehavior(StudyBehavior studyBehavior) {
        this.studyBehavior = studyBehavior;
    }

    public void setSleepBehavior(SleepBehavior sleepBehavior) {
        this.sleepBehavior = sleepBehavior;
    }

    abstract void display();

    public void studying() {
        // delegate to study behavior
        studyBehavior.study();
    }

    public void sleeping() {
        // delegate to sleep behavior
        sleepBehavior.sleep();
```

```
    }
}
```

Tạo Strategy Object

```
package strategy;

public class HardworkingStudent extends Student {

    public HardworkingStudent() {
        sleepBehavior = new Sleepless();
        studyBehavior = new StudyHard();
    }

    @Override
    void display() {
        System.out.println("I want to get a scholarship.");
    }
}
```

```
package strategy;

public class NormalStudent extends Student {

    public NormalStudent() {
        sleepBehavior = new SleepNormally();
        studyBehavior = new StudyNormally();
    }

    @Override
    void display() {
        System.out.println("I am a normal student.");
    }
}
```

```
package strategy;

public class LazyStudent extends Student {

    public LazyStudent() {
        sleepBehavior = new SleepAllDay();
        studyBehavior = new NoStudy();
    }

    @Override
    void display() {
        System.out.println("I am a lazy student.");
    }
}
```

```
package strategy;

```

```
public class LuckyStudent extends Student {

    public LuckyStudent() {
        sleepBehavior = new SleepAllDay();
        studyBehavior = new StudyNormally();
    }

    @Override
    void display() {
        System.out.println("I didn't study much, but I still got good
    grades.");
    }

}
```

Tạo Client Code

Tạo ra các đối tượng Student bằng hàm dựng của các StrategyObject.

```
package strategy;

public class StudentSimulator {
    public static void main(String[] args) {
        Student[] students = new Student[6];
        students[0] = new LazyStudent();
        students[1] = new HardworkingStudent();
        students[2] = new NormalStudent();
        students[3] = new LazyStudent();
        students[4] = new LuckyStudent();
        students[5] = new NormalStudent();

        processStudent(students);
    }

    private static void processStudent(Student[] students) {
        for (Student student : students) {
            System.out.println("-----------------------------------");
            student.display();
            student.sleeping();
            student.studying();
        }
    }
}
```

Kết quả:

```
---------------------------------------
I am a lazy student.
I sleep 15 hours per day.
I have no time for studying.
---------------------------------------
I want to get a scholarship.
```

```
I can't sleep until I finish my homework.
I love studying.
---------------------------------------
I am a normal student.
I sleep 8 hours per day.
I will study until I am sleepy.
---------------------------------------
I am a lazy student.
I sleep 15 hours per day.
I have no time for studying.
---------------------------------------
I didn't study much, but I still got good grades.
I sleep 15 hours per day.
I will study until I am sleepy.
---------------------------------------
I am a normal student.
I sleep 8 hours per day.
I will study until I am sleepy.
```

13.8 Mối quan hệ với các Design Patterns khác

- Bridge: Có chung cấu trúc, dựa trên composition (Giao phó trách nhiệm cho các đối tượng khác) tuy nhiên giải quyết các vấn đề khác nhau.
- Command: Khá giống nhau khi đều tham số hoá một đối tượng với một vài hành động tuy nhiên có intents khác nhau.
- State: Có thể coi như một extension của Strategy, đều dựa trên composition. Tuy nhiên State không hạn chế sự phụ thuộc giữa các concrete states.

TEMPLATE METHOD

Template method là một mẫu thiết kế hành vi, nó định nghĩa một bộ khung của một thuật toán trong một chức năng, chuyển giao việc thực hiện nó cho các lớp con. Template Method Pattern cho phép lớp con định nghĩa lại cách thực hiện của một thuật toán mà không phải thay đổi cấu trúc thuật toán. Nói cách khác, Template Method giúp chúng ta tạo nên một bộ khung(template) cho một vấn đề đang cần giải quyết. Trong đó các đối tượng cụ thể sẽ có cùng các bước thực hiện, nhưng trong mỗi bước thực hiện đó có thể khác nhau. Điều này tạo nên một cách thức truy cập giống nhau nhưng có hành động và kết quả khác nhau.

14.1 Đặt vấn đề

Giả sử chúng ta có một thuật toán để xây dựng một ngôi nhà, các bước cần làm để xây được ngôi nhà đó như sau: đổ móng nhà → dựng các cột, trụ → xây tường và làm cửa ra vào, cửa sổ → đổ mái nhà. Vấn đề ở đây là chúng ta không thể thay đổi thứ tự thực hiện các công đoạn trên, chẳng hạn chúng ta không thể làm tường xong mới đổ móng, hay đổ mái nhà xong mới làm tường.

14.2 Giải pháp

Để giải quyết vấn đề nêu trên, chúng ta cần tạo ra một Template để đưa vào đó các bước khác nhau theo thứ tự để xây dựng ngôi nhà.

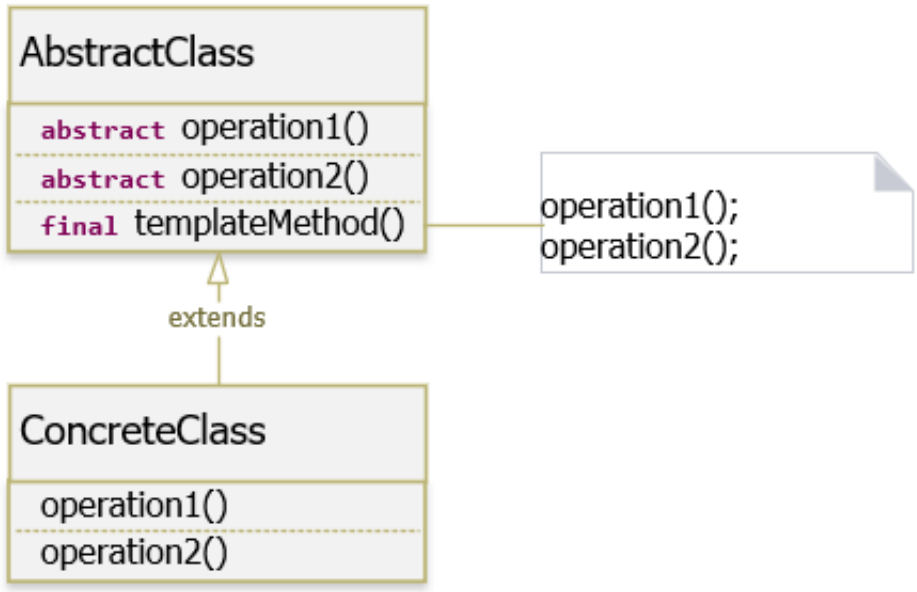
14.3 Khả năng áp dụng

Sử dụng Template Method khi:

- Có một thuật toán với nhiều bước và mong muốn cho phép tùy chỉnh chúng trong lớp con.
- Muốn chỉ có một triển khai phương thức trừu tượng duy nhất của một thuật toán.
- Muốn hành vi chung giữa các lớp con nên được đặt ở một lớp chung.

- Các lớp cha có thể gọi hành vi trong các lớp con của chúng một cách thống nhất(step by step).

14.4 Cấu trúc



Trong đó:

- **Abstract Class:** Định nghĩa các phương thức trừu tượng cho từng bước có thể được điều chỉnh bởi lớp con; cài đặt một phương thức duy nhất điều khiển thuật toán và gọi các bước riêng lẻ đã được cài đặt ở các lớp con.
- **Concrete Class:** là một thuật toán cụ thể, cài đặt các phương thức của Abstract Class. Các thuật toán này ghi đè lên các phương thức trừu tượng để cung cấp các triển khai thực sự. Nó không thể ghi đè phương thức duy nhất đã được cài đặt ở Abstract Class(Template Method).

14.5 Ưu nhược điểm

Ưu điểm:

- Tái sử dụng code, tránh trùng lặp code: đưa những phần trùng lặp vào lớp cha(Abstract Class).

- Cho phép Client override chỉ một số phần nhất định của thuật toán lớn, làm chúng ít bị ảnh hưởng hơn bởi những thay đổi xảy ra với các phần khác của thuật toán.

Nhược điểm:

- Một số Client có thể bị giới hạn bởi khung thuật toán được cung cấp.
- Có thể vi phạm nguyên tắc thay thế Liskov(Liskov Substitution Principle) bằng cách không cho phép triển khai bước mặc định thông qua lớp con.
- Có xu hướng khó duy trì hơn khi có nhiều bước hơn.

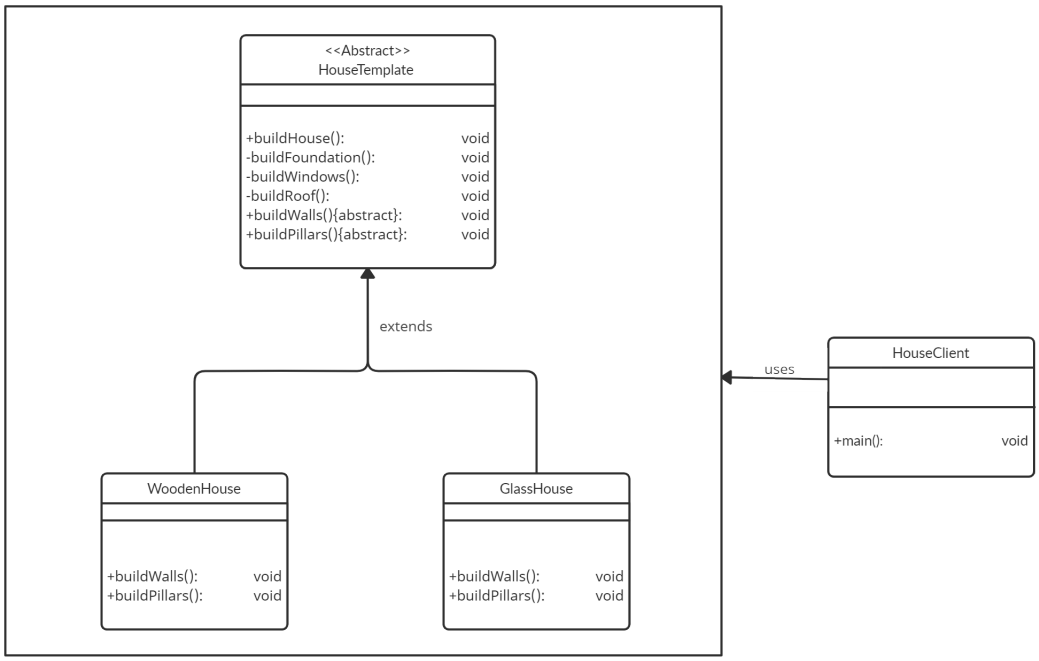
14.6 Cách triển khai

1. Phân tích mục tiêu thuật toán để xem có thể chia nó thành các bước hay không. Nếu có, bước nào sẽ là bước chung cho tất cả các subclass, bước nào sẽ phải @*Override*.
2. Tạo abstract class và khai báo các phương thức mẫu và các abstract method đại diện cho các bước của thuật toán. Phác thảo cấu trúc của thuật toán trong template method bằng việc thực hiện các bước tương ứng. Nên đặt template method là *final* để tránh bị các lớp con ghi đè.
3. Tất cả các bước có thể là trừu tượng(abstract) nhưng một số nên triển khai mặc định, các lớp con sẽ không triển khai những bước mặc định đó.
4. Hãy nghĩ đến việc thêm các móc nối(Hook) giữa các bước quan trọng của thuật toán.
5. Đối với mỗi biến thể của thuật toán, hãy tạo một lớp con cụ thể mới(new concrete subclass). Nó phải triển khai các bước trừu tượng nhưng cũng có thể @*Override* lại một số bước tùy chọn.

14.7 Code minh họa

Chúng ta có thể sử dụng Template Method trong việc xây nhà, kế hoạch thiết kế ban đầu có thể có một số điểm mở rộng cho phép điều chỉnh một

số chi tiết của ngôi nhà khi hoàn thiện, chẳng hạn mỗi công đoạn như xây móng nhà, dựng cột, xây tường, xây cửa sổ, đổ mái,... có thể có một chút sự thay đổi để phù hợp với từng loại nhà.



Tạo Abstract Class: lớp này có một template method, phương thức này chúng ta nên để là final tránh việc bị ghi đè và các abstract method(các bước xây nhà).

```
1  package templatemethod;
2
3  public abstract class HouseTemplate {
4      // template method, "final" not allow subclass override
5      public final void buildHouse() {
6          buildFoundation();
7          buildPillars();
8          buildWalls();
9          buildWindows();
10         buildRoof();
11         System.out.println("House is built.");
12     }
13
14     // default implementation
15     private void buildFoundation() {
16         System.out.println("Building foundation with cement, iron, stone
       and sand.");
17     }
18
19     private void buildWindows() {
20         System.out.println("Building windows.");
```

```
    }

    private void buildRoof() {
        System.out.println("Building roof.");
    }

    // methods to be implemented by subclasses
    public abstract void buildWalls();
    public abstract void buildPillars();
}
```

Tạo Concrete Class: kế thừa lớp Abstract và triển khai các abstract method của lớp cha(mỗi kiểu nhà sẽ có những vật liệu khác nhau).

```
package templatemethod;

public class WoodenHouse extends HouseTemplate {
    @Override
    public void buildWalls() {
        System.out.println("Building wooden walls.");
    }

    @Override
    public void buildPillars() {
        System.out.println("Building pillars with wood.");
    }
}
```

```
package templatemethod;

public class GlassHouse extends HouseTemplate {
  @Override
  public void buildWalls() {
    System.out.println("Building glass walls.");
  }

  @Override
  public void buildPillars() {
    System.out.println("Building pillars with glass.");
  }
}
```

Client: sử dụng phương thức buildHouse()(template method) để thể hiện việc xây ngôi nhà đã xác định.

```
package templatemethod;

public class HouseClient {
     public static void main(String[] args) {
        HouseTemplate houseType = new WoodenHouse();
        // using template method
        houseType.buildHouse();
```

```
8           System.out.println("----------------------------------------")
    ;
9
10          houseType = new GlassHouse();
11          houseType.buildHouse();
12      }
13
14  }
```

Kết quả:

```
Building foundation with cement, iron, stone and sand.
Building pillars with wood.
Building wooden walls.
Building windows.
Building roof.
House is built.
----------------------------------------
Building foundation with cement, iron, stone and sand.
Building pillars with glass.
Building glass walls.
Building windows.
Building roof.
House is built.
```

14.8 Mối quan hệ với các Design Patterns khác

- Strategy và Template Method đều đóng gói cái thuật toán, một theo kế thừa, một theo compose.
- Factory Method là một cụ thể hóa của Template Method.

COMMAND

Command (Action, Transaction) là một Behavioral Design Pattern. Nó cho phép chúng ta chuyển đổi một request thành một object độc lập chứa tất cả thông tin về request. Việc chuyển đổi này cho phép chúng ta tham số hoá các phương thức với các yêu cầu khác nhau như log, queue (undo/redo), transtraction.

15.1 Đặt vấn đề

Đôi khi chúng ta cần gửi các yêu cầu cho các đối tượng mà không biết bất cứ điều gì về hoạt động được yêu cầu hoặc người nhận yêu cầu. Chẳng hạn chúng có một ứng dụng văn bản, khi click lên button undo/ redo, save, cancel, ... yêu cầu sẽ được chuyển đến hệ thống xử lý, chúng ta sẽ không thể biết được đối tượng nào sẽ nhận xử lý, cách nó thực hiện như thế nào. Command Pattern là một Pattern được thiết kế cho những ứng dụng như vậy.

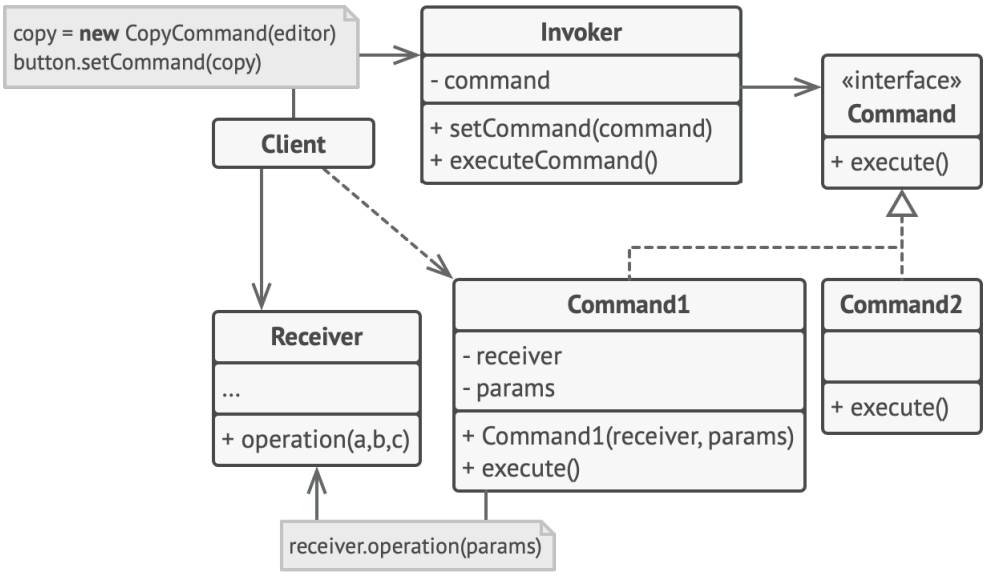
15.2 Giải pháp

Command Pattern cho phép tất cả những Request gửi đến Object được lưu trữ trong chính Object đó dưới dạng một Command Object. Command Object giống như một lớp trung gian được tạo ra để lưu trữ các câu lệnh và trạng thái của object tại một thời điểm nào đó.

15.3 Khả năng áp dụng

- Khi cần tham số hóa các đối tượng theo một hành động thực hiện (biến action thành parameter).
- Khi cần tạo và thực thi các yêu cầu vào các thời điểm khác nhau (delay action).
- Khi cần hỗ trợ tính năng undo, log, callback hoặc transaction.
- Phối hợp nhiều Command với nhau theo thứ tự.

15.4 Cấu trúc



Trong đó:

- **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.
- **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 thức execute() của ConcreteCommand chúng ta sẽ gọi phương thức thích hợp trong Receiver.

15.5 Ưu nhược điểm

Ưu điểm:

- Single Responsibility Principle: tách lớp gọi các thao tác từ các lớp thực hiện các thao tác này.
- Open/Closed Principle: đưa các Command mới vào chương trình mà không cần phá vỡ code hiện có.
- Có thể thực hiện hoàn tác.
- Giảm kết nối phụ thuộc giữa Invoker và Receiver.
- Cho phép đóng gói yêu cầu thành đối tượng, dễ dàng chuyển dữ liệu giữa các thành phần hệ thống.

Nhược điểm:

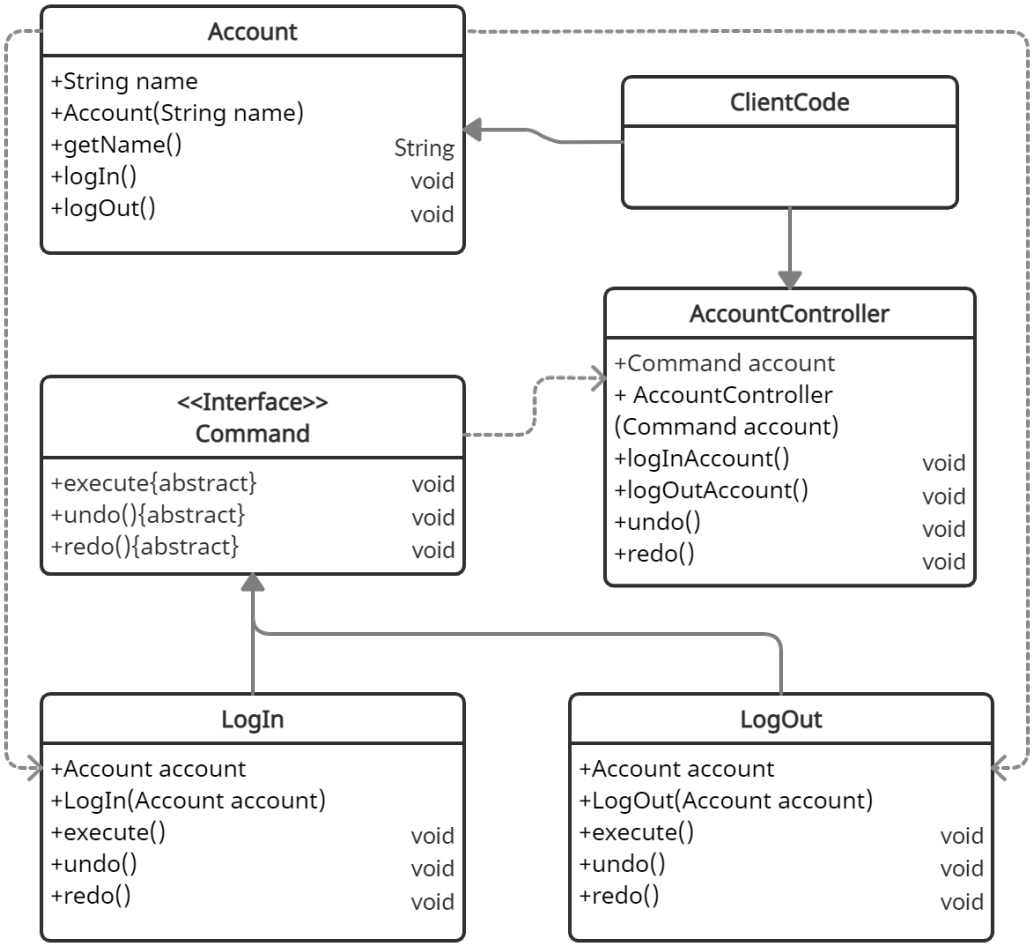
- Khiến code trở nên phức tạp hơn, sinh ra các class mới gây phức tạp cho mã nguồn.

15.6 Cách triển khai

1. Khai báo Command Interface và Receiver.
2. Bắt đầu trích xuất các request vào các ConcreteCommand cụ thể implements Command Interface. Mỗi lớp phải có một tập hợp các trường để lưu trữ các đối số request cùng với một tham chiếu đến Receiver Object. Tất cả các giá trị này phải được khởi tạo thông qua Command.
3. Xác định các lớp sẽ đóng vai trò là Invoker. Thêm các trường để lưu trữ lệnh vào các lớp này. Invoker chỉ nên giao tiếp với các lệnh của chúng thông qua Command Interface. Invoker thường không tự tạo command objects mà lấy chúng từ client code.
4. Thay đổi Invoker để họ thực hiện lệnh thay vì gửi request trực tiếp đến Receiver.
5. Client nên khởi tạo các đối tượng theo thứ tự sau:
    - Tạo các Receiver.
    - Tạo Command và liên kết chúng với các Receiver nếu cần.
    - Tạo Invoker và liên kết chúng với các lệnh cụ thể.

15.7 Code minh họa

Hiển thị thông tin login và log out của một tài khoản.



Tạo Command Interface

```
package command;

public interface Command {
    public void execute();

    public void undo();

    public void redo();
}
```

Tạo Receiver

Đối tượng Account có hai trạng thái là logIn và logOut.

```
package command;

import java.util.Date;

```

```
public class Account {
    private String name;

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

    public String getName() {
        return name;
    }

    public void logIn() {
        System.out.println(name + " logged in at " + new Date());
    }

    public void logOut() {
        System.out.println(name + " logged out at " + new Date());
    }
}
```

Tạo Concrete Command

Class LogIn implements Command thể hiện lệnh của trạng thái LogIn.

```
package command;

public class LogIn implements Command {
    private Account account;

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

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

    @Override
    public void undo() {
        account.logOut();
    }

    @Override
    public void redo() {
        account.logIn();
    }
}
```

Class LogOut implements Command thể hiện lệnh của trạng thái LogOut.

```java
package command;

public class LogOut implements Command {
    private Account account;

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

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

    @Override
    public void undo() {
        account.logIn();
    }

    @Override
    public void redo() {
        account.logOut();
    }
}
```

Tạo Invoker

```java
package command;

public class AccountController {
    private Command account;

    public AccountController(Command account) {
        this.account = account;
    }

    public void setCommand(Command command) {
        account = command;
    }

    public void logInAccount() {
        System.out.println("----------------------");
        System.out.println("User want to login.");
        account.execute();
    }

    public void logOutAccount() {
        System.out.println("----------------------");
        System.out.println("User want to log out.");
        account.execute();
    }
```

```
25
26      public void undo() {
27          System.out.println("----------------------");
28          System.out.println("User want to undo.");
29          account.undo();
30      }
31
32      public void redo() {
33          System.out.println("----------------------");
34          System.out.println("User want to redo.");
35          account.redo();
36      }
37
38  }
```

Tạo Client Code

Đầu tiên, chúng ta tạo ra đối tượng Receiver. Sau đó, chúng ta tạo lệnh Command bằng hàm dựng của một ConcreteCommand với đối số là đối tượng Receiver. Rồi chúng ta tạo đối tượng Invoker chính là một đối tượng AccountController bằng hàm dựng với đối số là lệnh Command vừa tạo. Sau đó, client có thể thay đổi và tùy chỉnh các lệnh đối với Receiver thông qua đối tượng Invoker.

```
1  package command;
2
3  public class TestMain {
4      public static void main(String[] args) {
5          // Create Receiver.
6          Account an = new Account("An Thu Phung");
7          // Create Command, and associate it with receiver.
8          Command logOut = new LogOut(an);
9          // Create invoker, and associate it with specific command.
10         AccountController facebook = new AccountController(logOut);
11
12         facebook.logOutAccount();
13         try {
14             Thread.sleep(6000);
15         } catch (InterruptedException e) {
16             e.printStackTrace();
17         }
18         facebook.undo();
19         facebook.redo();
20
21         facebook.setCommand(new LogIn(an));
22         facebook.logInAccount();
23     }
24 }
```

Kết quả:

```
--------------------
User want to log out.
An Thu Phung logged out at Wed Jan 05 23:16:35 ICT 2022
--------------------
User want to undo.
An Thu Phung logged in at Wed Jan 05 23:16:42 ICT 2022
--------------------
User want to redo.
An Thu Phung logged out at Wed Jan 05 23:16:42 ICT 2022
--------------------
User want to login.
An Thu Phung logged in at Wed Jan 05 23:16:42 ICT 2022
```

15.8 Mối quan hệ với các Design Patterns khác

- Chúng ta có thể sử dụng Command và Memento cùng nhau khi thực hiện "undo". Trong trường hợp này, Command chịu trách nhiệm thực hiện các hoạt động khác nhau trên một đối tượng đích, trong khi Memento lưu trạng thái của đối tượng đó ngay trước khi lệnh được thực thi.

- Prototype có thể hữu ích khi chúng ta cần lưu các bản sao của Command vào lịch sử.

- Chúng ta có thể coi Visitor như một phiên bản mạnh mẽ của mẫu Command. Các đối tượng của nó có thể thực thi các hoạt động trên các đối tượng khác nhau của các lớp khác nhau.

ITERATOR

Iterator là một Behavior Design 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".

16.1 Đặt vấn đề

Trong khi phát triển các ứng dụng, chúng ta làm việc với nhiều loại tập hợp như: Tree, Stack, Queue, HashSet, ArrayList, LinkedList, ... Cách thức mà tập hợp này lưu trữ đối tượng của nó rất khác nhau, và nếu chúng ta muốn truy cập dữ liệu của những đối tượng này,chúng ta phải học những kỹ thuật khác nhau cho từng loại tập hợp. Khi đó, Iterator pattern là một giải pháp tốt.

16.2 Giải pháp

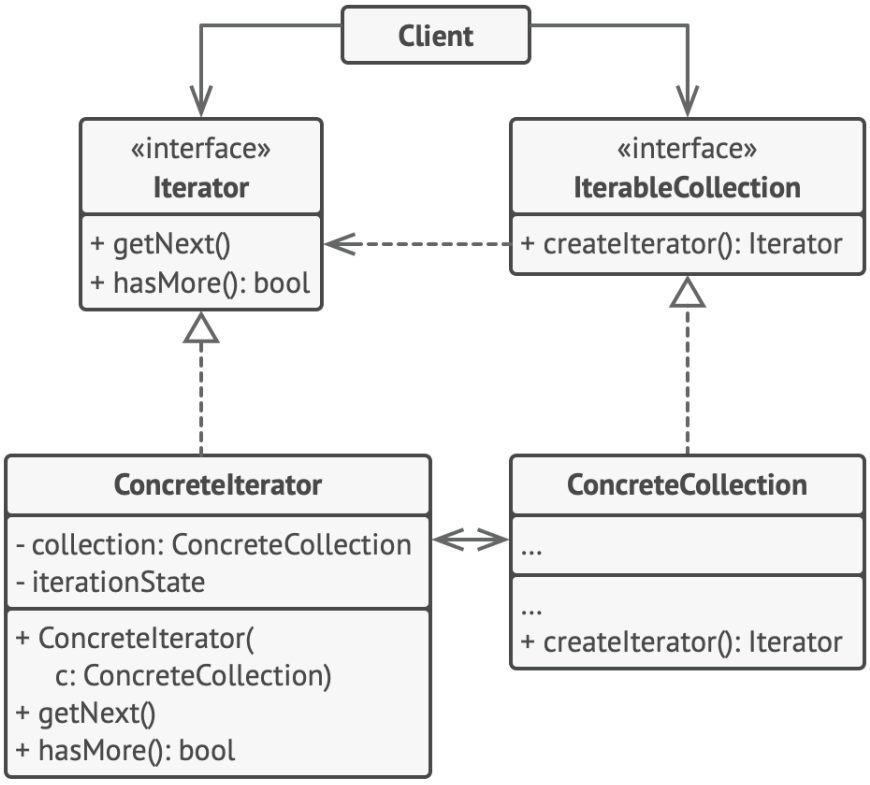
Ý tưởng của Iterator pattern là sử dụng một interface được xác định phương thức cụ thể để truy cập tới từng phần tử của tập hợp. Sử dụng những phương thức này, chúng ta có thể truy xuất tới các phần tử trong tập hợp theo cùng cách dễ dàng nhất.

16.3 Khả năng áp dụng

- 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.

16.4 Cấu trúc

Trong đó:



- **Iterator:** là một interface định nghĩa định nghĩa các phương thức để tạo Iterator object.
- **ConcreteIterator:** cài đặt các phương thức của Iterator, 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.

16.5 Ưu nhược điểm

Ưu điểm

- 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.

- 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.

Nhược điểm

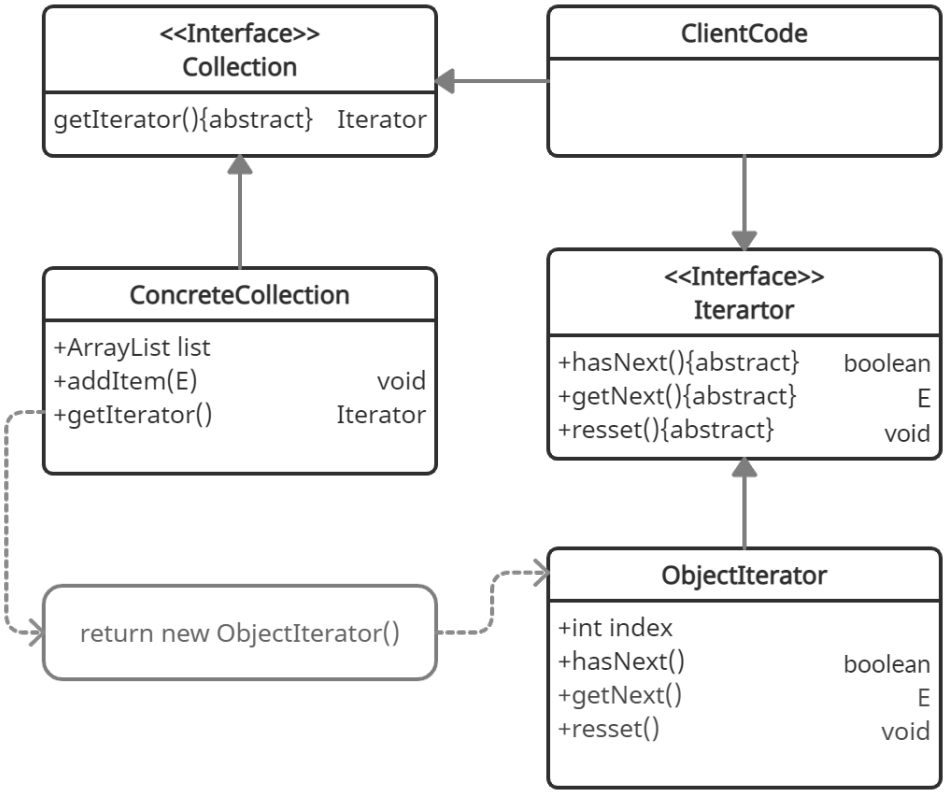
- Sử dụng Iterator có thể kém hiệu quả hơn so với việc duyệt qua các phần tử của collection một cách trực tiếp.
- Có thể không cần thiết nếu ứng dụng chỉ hoạt động với các collection đơn giản.

16.6 Cách triển khai

1. Khai báo Iterator Interface. Ít nhất, nó phải có một phương thức để tìm nạp phần tử tiếp theo từ một collection. Nhưng để thuận tiện, chúng ta có thể thêm một số phương thức khác, chẳng hạn như tìm nạp phần tử trước đó, theo dõi vị trí hiện tại và kiểm tra kết thúc của lần lặp.
2. Khai báo Collection Interface và mô tả một phương thức để tìm nạp các trình vòng lặp. Kiểu trả về phải bằng kiểu của Iterator Interface.
3. Triển khai Concrete Iterator cho các tập hợp mà chúng ta muốn có thể duyệt được bằng trình vòng lặp. Mỗi Iterator Object phải được liên kết với một Collection instance duy nhất thông qua Iterator.
4. Triển khai các Concrete Collection. Ý tưởng chính là cung cấp cho Client một lối tắt để tạo các trình vòng lặp, được điều chỉnh cho một lớp Collection cụ thể. Collection Object phải truyền chính nó đến hàm dựng của Iterator để thiết lập liên kết giữa chúng.
5. Xem qua client code để thay thế tất cả collection traverse code bằng việc sử dụng trình vòng lặp. Client tìm nạp một đối tượng trình lặp mới mỗi khi nó cần lặp qua các phần tử của Collection.

16.7 Code minh họa

Quản lí và hiện danh sách đối tượng Circle.



Tạo Iterator Interface

```
1  package iterator;
2
3  public interface Iterator<E> {
4      boolean hasNext();
5
6      E getNext();
7
8      void reset();
9  }
```

Tạo CollectionInterface với phương thức getIterator()

```
1  package iterator;
2
3  public interface Collection<E> {
4      public Iterator<E> getIterator();
5
6  }
```

Tạo ConcreteIterator class và ConcreteCollection

```java
package iterator;

import java.util.ArrayList;

public class ConcreteCollection<E> implements Collection<E> {

    private ArrayList<E> list = new ArrayList<>();

    @Override
    public Iterator<E> getIterator() {
        return new ObjectIterator();
    }

    public void addItem(E item) {
        this.list.add(item);
    }

    class ObjectIterator implements Iterator<E> {
        int index;

        @Override
        public boolean hasNext() {
            return (index < list.size());
        }

        @Override
        public E getNext() {
            return list.get(index++);
        }

        @Override
        public void reset() {
            index = 0;
        }

    }
}
```

Tạo Client Code

Tạo class Circle để sử dụng kiểu dữ liệu này.

```java
package iterator;

public class Circle {
    private double radius;

    public Circle() {
        radius = 1.0;
    }

    public Circle(double radius) {
```

```
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    public double getArea() {
        return radius * radius * Math.PI;
    }

    public double getCircumference() {
        return 2 * radius * Math.PI;
    }

    public String toString() {
        return "Circle:[Radius = " + radius + "]";
    }

}
```

Đầu tiên tạo đối tượng ConcreteCollection, rồi lần lượt thêm các đối tượng với kiểu dữ liệu Circle. Tạo Iterator bằng phương thức getIterator() và sử dụng theo ý muốn của client.

```
package iterator;

public class TestMain {
    public static void main(String[] args) {
        ConcreteCollection<Circle> collection = new ConcreteCollection
    <>();
        for (int i = 0; i < 12; i++) {
            Circle insert = new Circle(i + 1);
            collection.addItem(insert);
        }

        Iterator<Circle> iterator = collection.getIterator();
        while (iterator.hasNext()) {
            Circle item = iterator.getNext();
            System.out.println(item);
        }
    }
}
```

Kết quả:

```
Circle:[Radius = 1.0]
Circle:[Radius = 2.0]
Circle:[Radius = 3.0]
Circle:[Radius = 4.0]
Circle:[Radius = 5.0]
Circle:[Radius = 6.0]
Circle:[Radius = 7.0]
Circle:[Radius = 8.0]
Circle:[Radius = 9.0]
Circle:[Radius = 10.0]
Circle:[Radius = 11.0]
Circle:[Radius = 12.0]
```

16.8 Mối quan hệ với các Design Patterns khác

- Dùng Iterator để traverse(duyệt) Composite tree.
- Sử dụng Factory Method cùng với Iterator để cho phép các lớp con của collection trả về các loại trình vòng lặp khác nhau tương thích với các collection.
- Sử dụng Memento cùng với Iterator để nắm bắt trạng thái lặp lại hiện tại và khôi phục nó nếu cần.
- Sử dụng Visitor cùng với Iterator để duyệt qua một cấu trúc dữ liệu phức tạp và thực hiện một số thao tác trên các phần tử của nó, ngay cả khi tất cả chúng đều có các lớp khác nhau.

STATE

State Pattern là một Behavioral Design Pattern. Nó cho phép một object có thể biến đổi hành vi của nó khi có những sự thay đổi trạng thái nội bộ.

17.1 Đặt vấn đề

Trong thực tế, có rất nhiều đối tượng tại bất kỳ thời điểm nào cũng có một số trạng thái hữu hạn khác nhau, và chúng có thể chuyển từ trạng thái này sang trạng thái khác. Trong các trạng thái khác nhau đó, chúng có những đặc tính và hành vi khác nhau. Để diễn tả sự thay đổi hành vi và chức năng này, chúng ta có thể sử dụng if-else hoặc switch-case. Tuy nhiên, khi các hành vi và chức năng thay đổi quá nhiều, chúng ta sẽ cần tốn rất nhiều công sức để cài đặt hết trường hợp hành vi thay đổi.

17.2 Giải pháp

State pattern cho rằng nên tạo các class mới cho tất cả các trạng thái có thể có của một object và trích xuất tất cả các hành vi dành riêng cho trạng thái vào các class đó.

Thay vì tự thực hiện tất cả các hành vi, đối tượng ban đầu, được gọi là Context, lưu trữ một tham chiếu đến một trong các đối tượng trạng thái đại diện cho trạng thái hiện tại của nó và ủy quyền tất cả các công việc liên quan đến trạng thái cho đối tượng đó.

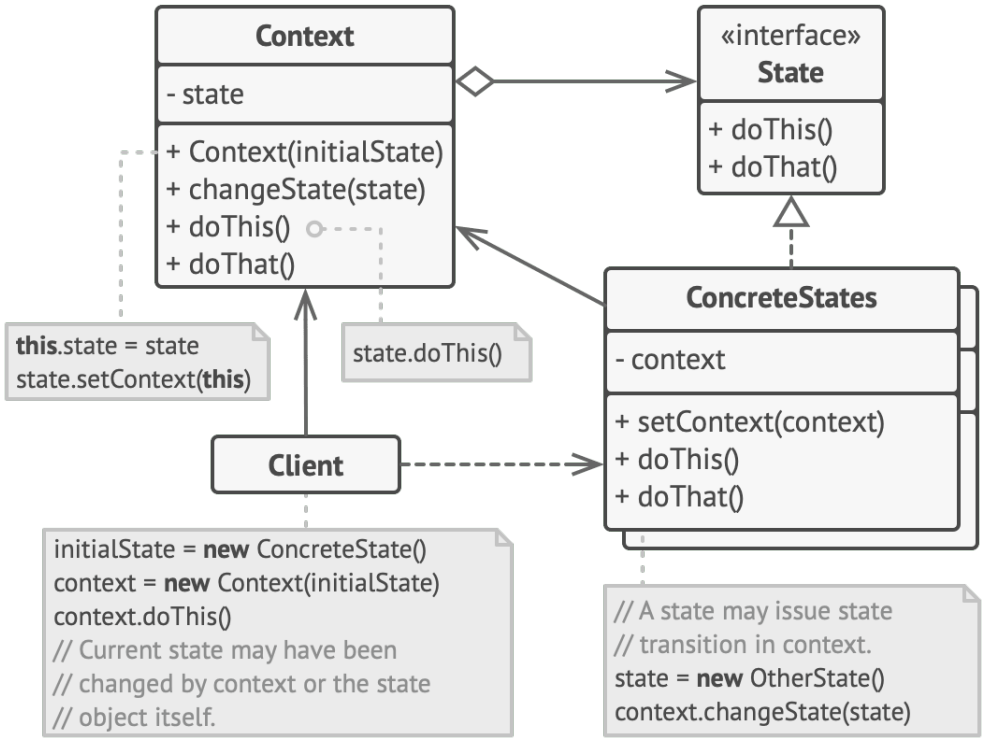
Để chuyển Context sang State khác, hãy thay thế State Object hoạt động bằng một Object khác đại diện cho State mới đó. Điều này chỉ có thể thực hiện được nếu tất cả các State class tuân theo cùng một Interface và Context hoạt động với các Object này thông qua Interface đó.

17.3 Khả năng áp dụng

- Hành vi của một đội tượng phụ thuộc vào trạng thái của nó. Tại thời điểm runtime, khi đối tượng thực hiện hành vi, trạng thái của nó sẽ thay đổi theo.

- Đối tượng có nhiều trường hợp sử dụng với các hành vi của nó, nhiều hành vi phụ thuộc vào trạng thái của đối tượng. Hay nói cách khác, đối tượng có nhiều trạng thái, mỗi trạng thái có những hành vi khác nhau.

17.4 Cấu trúc



Trong đó:

- **Context:** định nghĩa giao diện chính để giao tiếp với clients và nó chứa một thể hiện của ConcreateState tương ứng với trạng thái hiện tại của đối tượng.
- **State:** định nghĩa giao diện để đóng gói những hành vi giao tiếp với từng trạng thái của Context.
- **ConcreateState subclasses:** mỗi một class con implements một hành vi giao tiếp với một trạng thái của Context.

17.5 Ưu nhược điểm

Ưu điểm:

- Single responsibility principle (SRP) : tách biệt mỗi State tương ứng với một class riêng biệt.
- Open/Closed Principle (OCP) : chúng ta có thể thêm một State mới mà không ảnh hưởng đến State khác hay Context hiện có.
- Giữ hành vi cụ thể tương ứng với trạng thái.
- Giúp chuyển trạng thái một cách rõ ràng.

Nhược điểm:

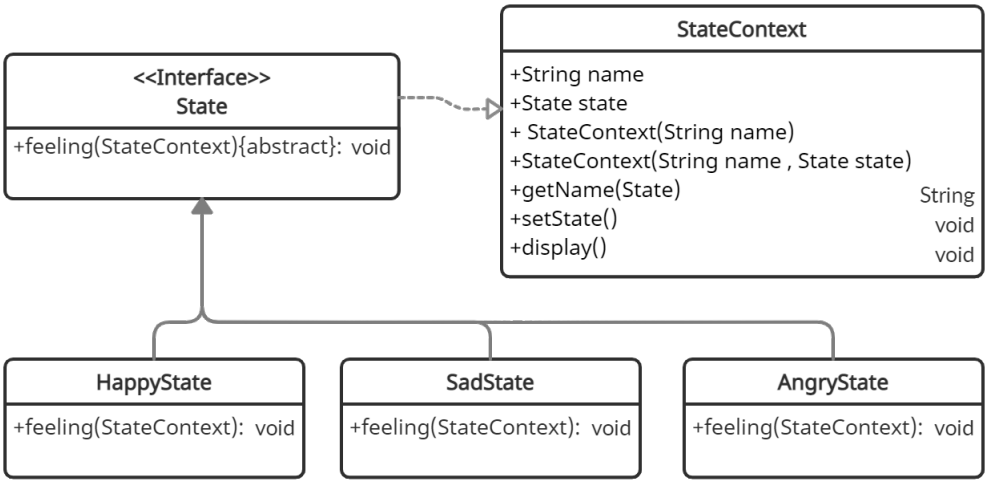
- Việc áp dụng pattern có thể quá mức cần thiết nếu đối tượng có ít trạng thái hoặc chúng hiếm khi thay đổi làm tăng độ phức tạp của code.

17.6 Cách triển khai

1. Khai báo State Interface.
2. Tạo một class Context. Class này chứa thông tin State hiện tại và nhận yêu cầu xử lý trực tiếp từ Client.
3. Tạo các Concrete State subclass implement State Interface.
4. Để đổi State của Context, tạo instance của một trong các Concrete State class và truyền nó vào instance của Context. Ta có thể làm điều này bên trong context, hoặc trong các states, hoặc trong client code. Dù nó được thực hiện ở đâu, class trở nên phụ thuộc vào state object cụ thể.

17.7 Code minh họa

Chúng ta tạo ra các đối tượng con người có cảm xúc chuyển tiếp trong 3 trạng thái là: Happy, Sad và Angry.



Tạo State Interface: Đối tượng abstract để làm việc với Client Code

```
package state;

public interface State {
    void feeling(StateContext person);
}
```

Tạo Context class

```
package state;

public class StateContext {
    private String name;
    private State state;

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

    public StateContext(String name, State state) {
        this.name = name;
        this.state = state;
    }

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

    public String getName() {
        return name;
    }

    public void display() {
        System.out.println("---------------");
```

```
        state.feeling(this);
    }
}
```

Tạo các Concrete State

Trạng thái Happy:

```
package state;

public class HappyState implements State {
    @Override
    public void feeling(StateContext person) {
        System.out.println(person.getName() + ": happy.");
    }
}
```

Trạng thái Sad (chuyển sang trạng thái Angry trong lúc thực thi phương thức feeling()):

```
package state;

public class SadState implements State {
    @Override
    public void feeling(StateContext person) {
        System.out.println(person.getName() + ": sad.");
        person.setState(new AngryState());
    }
}
```

Trạng thái Angry:

```
package state;

public class AngryState implements State {
    @Override
    public void feeling(StateContext person) {
        System.out.println(person.getName() + ": angry.");
    }
}
```

Tạo Client Code

```
package state;

public class TestMain {
    public static void main(String[] args) {
        StateContext lana = new StateContext("Lana");
        lana.setState(new HappyState());
        lana.display();
        lana.setState(new AngryState());
        lana.display();

        State sad = new SadState();
```

```
12          StateContext alan = new StateContext("Alan", sad);
13          alan.display();
14          alan.display();
15
16      }
17  }
```

Kết quả:

```
---------------
Lana: happy.
---------------
Lana: angry.
---------------
Alan: sad.
---------------
Alan: angry.
```

17.8 Mối quan hệ với các Design Patterns khác

- State: Có thể coi như một extension của Strategy, chúng đều dựa trên composition. Tuy nhiên State không hạn chế sự phụ thuộc giữa các ConcreteStates.

Tổng kết

Design Patterns thể hiện tính kinh nghiệm của công việc lập trình, xây dựng và thiết kế phần mềm. Nó đóng một vai trò vô cùng quan trọng trong quá trình phát triển phần mềm. Nếu như chúng ta hiểu rõ và vận dụng được Design Patterns trong quá trình thiết kế hệ thống sẽ tiết kiệm được rất nhiều thời gian, chi phí, dễ dàng phát triển, mở rộng và bảo trì. Tuy nhiên chúng ta không nên quá lạm dụng Design Patterns. Tóm lại, khi chúng ta muốn sử dụng một Design Pattern nào đó thì nên tập trung vào các yếu tố như: mẫu được sử dụng khi nào, vấn đề mà nó giải quyết là gì, tìm hiểu về sơ đồ UML mô tả, xem code minh họa và ứng dụng thực tiễn của design pattern đó. Điều này sẽ giúp chúng ta dễ dàng hình dung và đánh giá được design pattern một cách chính xác hơn.

Tài liệu tham khảo

[1] https://refactoring.guru/design-patterns

[2] https://gpcoder.com

[3] https://sourcemaking.com

[4] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides(1994). Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.

[5] Alexander Shvets(2019). Dive into Design Patterns.

[6] Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates(2004). Head First Design Patterns. O'Reilly Media, USA.