**DESIGN PATTERN**

1. Strategy Design Pattern

1.1. Phân tích vấn đề

- Giả sử ta có một ứng dụng về quản lý nhóm Vịt

+ Với giải pháp bình thường kế thừa:

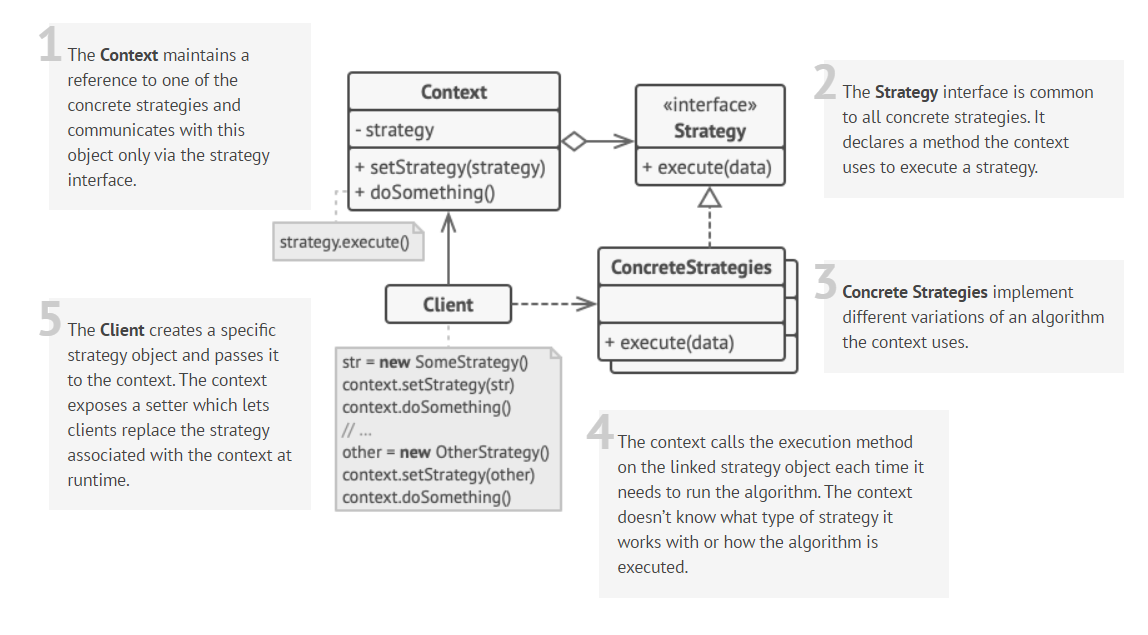
* Ta sẽ tạo ra một class Vit chứa các thuộc tính và phương thức (ăn, uống, bay, chạy,..).
* Sau đó ta sẽ tiến hành tạo các class con Vịt trời,… kế thừa Vịt ban đầu.
* Vấn đề sảy ra khi Vịt Cỏ xuất hiện. Bởi Vịt Cỏ không hề biết bay nên ta phải override lại phương thức bay. Và sau đó các con Vịt Xiêm, Vịt Nhà,… xuất hiện, ta lại tiếp tục override lại phương thức bay.
* Tuy nhiên tiếp theo vấn để lại tiếp tục xuất hiện khi Khách Hàng của chúng ta yêu cầu trong phương thức Bay của chúng ta sẽ xuất ra một câu “Tao biết bay” nếu vịt biết bay, và “Tao không biết bay” với vịt không biết bay. Thế là ta ngồi và chỉnh từng phương thức Bay trong các class ta tạo ra. Công việc như vậy quá tốn thời gian.
* Tuy nhiên, sáng mai thức dậy, ta lại tiếp tục nhận được yêu cầu của khách hàng. Khách hàng lại muốn phân chia nhóm Vịt biết bay ra thành nhóm vịt bay cao và bay thấp, và lại tiếp tục in ra câu “Tao bay cao vãi”, “Tao bay thấp quá”. Đến lúc này không còn cách nào khác ta phải sửa lại từ logic đến các chuổi String trong phương thức Bay.
* Không dừng lại ở đó, Khách hàng của chúng ta rất thích kiềm chuyện. Ngày hôm sau, ông ta lại bảo. “Thôi, tôi thấy chia vịt bay cao thấp quá vớ vẫn, bạn sửa lại như cũ giúp tôi”. What the f\*\*\* man. Nhưng thôi, bạn có chửi thề hay làm gì thì bạn cũng phải sửa lại theo yêu cầu khách hàng. Cuộc sống mà biết sao giờ.
* Như bạn thấy đấy, việc sử dụng kế thừa trong hướng đối tượng cơ bản trong trường hợp này, sẽ khiền code chúng ta trước hết là không clean dẫn đến ta phải sữa đi sữa lại rất nhiều lần. Và tất nhiên điều đó là không chấp nhận được với một tín đồ Java như tôi. Ở java có một câu rất nổi tiếng “Write One, Run AnyWhere” nghĩa là code một lần và chạy bất cứ đâu. Ta phải tìm ra phương pháp để chúng ta chỉ cần sửa một lần và các lớp Vịt con đều sẽ nhận được. Và chúng ta sẽ biết đến nó ở phần tiếp theo Giải pháp.

1.2. Giải pháp

- Như vấn đề chúng ta đã phân tích, ta đang gặp vấn đề với phương thức Bay, nó khiến chúng ta phải cực khổ thay đi thay lại n lần.

- Vậy sao chúng ta không định nghĩa một Interface/Abtract class Bay. Sau đó các Class Bay đủ thể loại của ta hoàn toàn có thể implement Bay. Trong Bay chúng ta sẽ có Không biết bay, Bay Cao, Bay thấp, và thậm chí sau này Khách Hàng có muốn Bay Xuống Địa Phủ hay Bay lên Thương Đường, ta cũng chỉ định nghĩa implement Bay. Và Vit cũng chỉ cần một phương thức get để nhận kết quả từ Bay mà chẳng cần quan tâm vấn đề khác.

1.3. Kiến trúc (Class Diagram)



1.4. Ví dụ

- Ta sẽ tiếp tục thực hiện với ví dụ ta đã nhắc tới trong phần Phân tích.

- Đầu tiên ta sẽ tạo một Class Duck

public class Duck {  
 private int \_id;  
 private String \_gender;  
 private String \_price;  
 private IFlyStrategy \_flyStrategy;  
  
 public Duck() {  
  
 }  
  
 public Duck(IFlyStrategy *flyStrategy*) {  
 \_flyStrategy = *flyStrategy*;  
 }  
  
 public int get\_id() {  
 return \_id;  
 }  
  
 public void set\_id(int *\_id*) {  
 this.\_id = *\_id*;  
 }  
  
 public String get\_gender() {  
 return \_gender;  
 }  
  
 public void set\_gender(String *\_gender*) {  
 this.\_gender = *\_gender*;  
 }  
  
 public String get\_price() {  
 return \_price;  
 }  
  
 public void set\_price(String *\_price*) {  
 this.\_price = *\_price*;  
 }  
  
 public IFlyStrategy get\_flyStrategy() {  
 return \_flyStrategy;  
 }  
  
 public void set\_flyStrategy(IFlyStrategy *\_flyStrategy*) {  
 this.\_flyStrategy = *\_flyStrategy*;  
 }  
  
 public void fly() {  
 this.\_flyStrategy.doFly();  
 }  
}

- Trong Duck ngoài các thông tin cơ bản ta sẽ khao báo thêm một IflyStrategy là một interface chứa phương thức doFly().

- Ta sẽ bỏ qua các thông tin khác, trong ví dụ này, ta chỉ tập trung vào phương thức fly() trong Duck. Phương thức fly của chúng ta đang gọi đến doFly() trong Interface

public interface IFlyStrategy {  
 void doFly();  
}

- Và tất nhiên Interface của chúng ta định nghĩa chỉ có khai báo phương thức, ta sẽ tiến hành khởi tạo các class implement interface này để thực hiện các loại Bay.

- Đầu tiên ta dịnh nghĩa lớp Không biết bay

public class NoFlyStrategy implements IFlyStrategy{  
  
 *@Override* public void doFly() {  
 System.out.println("I don't know how to fly, please!");  
 }  
}

- Lớp này sẽ implement Interface của chúng ta, và override lại phương thức doFly()

- Tương tự ta tạo các lớp khác

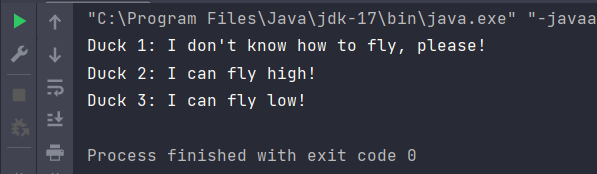
public class FlyHighStrategy implements IFlyStrategy{  
 *@Override* public void doFly() {  
 System.out.println("I can fly high!");  
 }  
}

public class FlyLowStrategy implements IFlyStrategy{  
 *@Override* public void doFly() {  
 System.out.println("I can fly low!");  
 }  
}

- Tiếp đó ta sẽ khai báo một vài duck để test fly()

public class Main {  
 public static void main(String[] *args*) {  
 Duck duck1 = new Duck();  
 duck1.set\_flyStrategy(new NoFlyStrategy());  
 System.out.print("Duck 1: ");  
 duck1.fly();  
  
 Duck duck2 = new Duck();  
 duck2.set\_flyStrategy(new FlyHighStrategy());  
 System.out.print("Duck 2: ");  
 duck2.fly();  
  
 Duck duck3 = new Duck();  
 duck3.set\_flyStrategy(new FlyLowStrategy());  
 System.out.print("Duck 3: ");  
 duck3.fly();  
 }  
}

- Ta tạm khai báo ba Duck và ta setFly cho từng loại là các loại bay mà ta muốn. Sau đó ta cho fly và kết quả:



- Các con vịt của ta đã bay đúng như định nghĩa, và bạn thấy đấy ta chỉ cần sửa các strategy implement từ IflyStrategy của chúng ta, nếu muốn thay đổi String. Không như kế thừa, ta phải đi sửa trong từng class con.

- Hoặc nếu ta có một thêm Bay lên trời bay xuống đất bay đi đâu cũng được, ta chỉ cần tạo thêm lớp FlyToSkyStrategy, FlyToAnywhereStrategy, … thì hoàn toàn code của chúng ta vẫn không bị thay đổi, và việc thêm cũng không tốn thời gian. Lợi ích lớn nhất ở đây là tính encapsulation (đóng gói). Ta không cần phải thay đổi gì trong Duck.

**Tổng kết lại:** Trên đây là ví dụ đơn giản về Strategy Pattern, trong thực tế, khi nhận một task chúng ta nên phân tích sẽ nghĩ đến các tình huống có thể diễn ra để có thể cân nhắc chúng ta có nên dùng Pattern hay không. Không nên dùng Pattern một cách tùy tiện. Nên nhớ Pattern sinh ra là để chúng ta có thể dễ dàng thay đổi maintance code, không phải làm phức tạp hóa code.

2. Observer Design Pattern

2.1. Phân tích vấn đề

- Giả sử ta có một ứng dụng quản lý video, và ta đang được yêu cầu thực hiện một tính năng là bất cứ khi nào chủ video thay đổi các properties của video như title, description, fileName thì ẽ có thông báo được gửi đến tất cả người quan tâm đến video đó hoặc kênh đó qua mail, phone, youtube.

Đầu tiên ta sẽ tiến hành thiết kế VideoMedia

public class VideoMedia {  
 private int id;  
 private String title;  
 private String description;  
 private String fileName;  
 private final EmailNotification emailNotification = new EmailNotification();  
 private final PhoneNotification phoneNotification = new PhoneNotification();  
 private final YoutubeNotification youtubeNotification = new YoutubeNotification();  
  
 public VideoMedia() {  
  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int *id*) {  
 this.id = *id*;  
 }  
  
 public String getTitle() {  
 return title;  
 }  
  
 public void setTitle(String *title*) {  
 this.title = *title*;  
 videoDataChange();  
 }  
  
 public String getDescription() {  
 return description;  
 }  
  
 public void setDescription(String *description*) {  
 this.description = *description*;  
 videoDataChange();  
 }  
  
 public String getFileName() {  
 return fileName;  
 }  
  
 public void setFileName(String *fileName*) {  
 this.fileName = *fileName*;  
 videoDataChange();  
 }  
  
 public void videoDataChange() {  
 String title = getTitle();  
 String description = getDescription();  
 String fileName = getFileName();  
  
 emailNotification.notifyToUser(title, description, fileName);  
 phoneNotification.notifyToUser(title, description, fileName);  
 youtubeNotification.notifyToUser(title, description, fileName);  
 }  
}

- Và tất nhiên ta sẽ tách các lớp Notify ra ngoài để có thể hiệu quả trong việc xử lý maintance code.

public class EmailNotification {  
 public void notifyToUser(String *title*, String *description*, String *fileName*) {  
 String notifyString = "Email Notify Change Data\nTitle: " + *title* + "\nDescription: " + *description* + "\nFileName: " + *fileName*;  
 System.out.println(notifyString);  
 }  
}

public class PhoneNotification {  
 public void notifyToUser(String *title*, String *description*, String *fileName*) {  
 String notifyString = "Phone Notify Change Data\nTitle: " + *title* + "\nDescription: " + *description* + "\nFileName: " + *fileName*;  
 System.out.println(notifyString);  
 }  
}

public class YoutubeNotification {  
 public void notifyToUser(String *title*, String *description*, String *fileName*) {  
 String notifyString = "Youtube Notify Change Data\nTitle: " + *title* + "\nDescription: " + *description* + "\nFileName: " + *fileName*;  
 System.out.println(notifyString);  
 }  
}

- Mọi thứ đang hoạt động đúng như ta mong muốn.

- Tuy nhiên một ngày đẹp trời, khách hàng của chúng ta rất là vui tính, ông bảo “Tôi quên mất thông báo Facebook, bạn thêm tính FacebookNotify giúp tôi, nhanh gọn lẹ”. Và vâng, tôi rất sẵn lòng thêm tính năng, đơn giản ta chỉ cần tạo ra một class FacebookNotify, sau đó thêm vào VideoMedia, mọi chuyện khá đơn giãn. Nhưng đến đây dừng khoảng chừng là 5s. Thêm như vậy thì còn encapsulation (đóng gói) gì nữa ta. Nhưng không sao, chắc chỉ có lần này nên mọi chuyện có vẻ vẫn chấp nhận. Và tất nhiên sữa code như vậy ta bắt buộc phải deploy lại và không còn cách nào khác và tất nhiên điều đó là tốn TIỀN.

- Hai ngày sau, khách hàng của chúng tôi làm ăn thua lỗ (lỗi tại Tây Bán Nhà), và ông ta đang cần thêm một chút tiền để All in Pháp. Sau khi suy nghĩ một lúc, ông ta bảo “Tao nghĩ chỉ cần thông báo qua mail là được rồi, không cần phải thông báo lắm như vậy, quá tốn tiền, các cậu sửa lại giúp tôi”. Vâng và diều gì đến cũng sẽ đến, điều gì đi cũng sẽ đi, what the f\*\*\* man. Tuy nhiên việc này chỉ là xóa các đoạn code thông báo nên cũng không tốn quá nhiều thời gian. Và tất nhiên lại phải deploy lại.

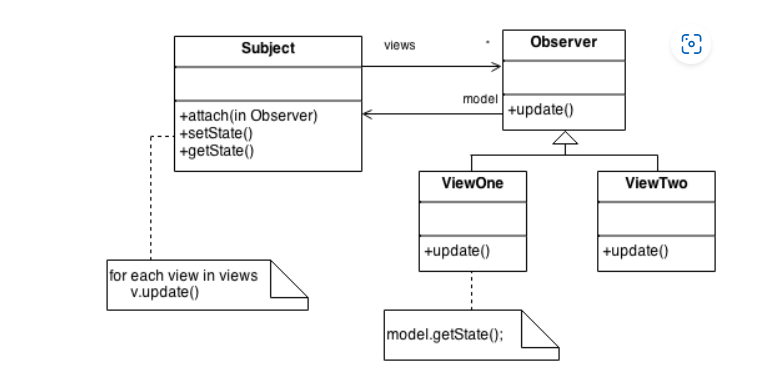
- Vâng tối hôm đó Pháp thắng, khách hàng của chúng tôi cực kì vui, tuy nhiên cuộc sống mà “Người tàn ác thường sống thảnh thơi”. Vì ông ta muốn thảnh thời chứ nên “Giờ tao có tiền rồi, tao thấy chỉ có email là chưa đủ, thêm lại đủ Notify như củ giúp tao”. Vâng, tôi đã rất bình tĩnh vì méo bình tĩnh là tao múc ổng rồi.

- Lần này tôi không dại gì phá vỡ tính đóng gói của OOP nữa, biết đâu mai ổng lại all in thua thì lại mệt. Và tất nhiên vấn đề được đặt ra ở đây.

2.2. Giải pháp

- Chắc chắn rồi, ta phải tìm cách tách Notify ra ngoài VideoMedia.

2.3. Kiến trúc



2.4. Ví dụ

- Đầu tiên ta cần một Subject và trong đó sẽ định nghĩa list<observer> vì ta đang cố tạo liên kết một nhiều đề khi một đối tượng thay đổi trạng thái, thì tất cả các đồi tượng dựa trên nó sẽ được thông báo và cập nhật tự động.

package Base;  
  
import java.util.ArrayList;  
import java.util.Objects;  
  
public class Subject {  
 private final ArrayList<NotifyObserver> notifyObserverList = new ArrayList<>();  
  
 public void attachObserver(NotifyObserver *notifyObserver*) {  
 notifyObserverList.add(*notifyObserver*);  
 }  
  
 public void detachObserver(NotifyObserver *notifyObserver*) {  
 notifyObserverList.remove(*notifyObserver*);  
 }  
  
 public void notifyObserver(Objects *arg*) {  
 notifyObserverList.forEach((*observer*) -> *observer*.notify(*arg*));  
 }  
}

- Và tất nhiên ta cũng tạo một Observer

package Base;  
  
import java.util.Objects;  
  
public abstract class NotifyObserver {  
 protected Subject subject;  
 public void notify(Objects *arg*) {}  
}

- Sau đó ta lần lượt tạo các Notify kế thừa từ Observer, và override lại phương thức notify

import Base.NotifyObserver;  
import Base.Subject;  
  
import java.util.Objects;  
  
public class EmailNotification extends NotifyObserver {  
 public EmailNotification(Subject *subject*) {  
 this.subject = *subject*;  
 this.subject.attachObserver(this);  
 }  
 *@Override* public void notify(Objects *arg*) {  
 VideoMedia videoMedia = (VideoMedia) subject;  
 String notifyString = "Email Notify Change Data\nTitle: " + videoMedia.getTitle()  
 + "\nDescription: " + videoMedia.getDescription()  
 + "\nFileName: " + videoMedia.getFileName();  
 System.out.println(notifyString);  
 }  
}

import Base.NotifyObserver;  
import Base.Subject;  
  
import java.util.Objects;  
  
public class PhoneNotification extends NotifyObserver {  
 public PhoneNotification(Subject *subject*) {  
 this.subject = *subject*;  
 this.subject.attachObserver(this);  
 }  
 *@Override* public void notify(Objects *arg*) {  
 VideoMedia videoMedia = (VideoMedia) subject;  
 String notifyString = "Phone Notify Change Data\nTitle: " + videoMedia.getTitle()  
 + "\nDescription: " + videoMedia.getDescription()  
 + "\nFileName: " + videoMedia.getFileName();  
 System.out.println(notifyString);  
 }  
}

import Base.NotifyObserver;  
import Base.Subject;  
  
import java.util.Objects;  
  
public class YoutubeNotification extends NotifyObserver {  
 public YoutubeNotification(Subject *subject*) {  
 this.subject = *subject*;  
 this.subject.attachObserver(this);  
 }  
 *@Override* public void notify(Objects *arg*) {  
 VideoMedia videoMedia = (VideoMedia) subject;  
 String notifyString = "Youtube Notify Change Data\nTitle: " + videoMedia.getTitle()  
 + "\nDescription: " + videoMedia.getDescription()  
 + "\nFileName: " + videoMedia.getFileName();  
 System.out.println(notifyString);  
 }  
}

package Notifies;  
  
import Base.NotifyObserver;  
import Base.Subject;  
import Model.VideoMedia;  
  
import java.util.Objects;  
  
public class FacebookNotification extends NotifyObserver {  
 public FacebookNotification(Subject *subject*) {  
 this.subject = *subject*;  
 this.subject.attachObserver(this);  
 }  
 *@Override* public void notify(Objects *arg*) {  
 VideoMedia videoMedia = (VideoMedia) subject;  
 String notifyString = "Facebook Notify Change Data\nTitle: " + videoMedia.getTitle()  
 + "\nDescription: " + videoMedia.getDescription()  
 + "\nFileName: " + videoMedia.getFileName();  
 System.out.println(notifyString);  
 }  
}

- Sau đó trong VideoMedia ta có thể bỏ các Notify và chỉ cần extends Subject và gọi đến phương thức notifyObserver()

package Model;  
  
import Base.Subject;  
  
public class VideoMedia extends Subject {  
 private int id;  
 private String title;  
 private String description;  
 private String fileName;  
  
 public VideoMedia() {}  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int *id*) {  
 this.id = *id*;  
 }  
  
 public String getTitle() {  
 return title;  
 }  
  
 public void setTitle(String *title*) {  
 this.title = *title*;  
 videoDataChange();  
 }  
  
 public String getDescription() {  
 return description;  
 }  
  
 public void setDescription(String *description*) {  
 this.description = *description*;  
 videoDataChange();  
 }  
  
 public String getFileName() {  
 return fileName;  
 }  
  
 public void setFileName(String *fileName*) {  
 this.fileName = *fileName*;  
 videoDataChange();  
 }  
  
 public void videoDataChange() {  
 notifyObserver(null);  
 }  
}

- Chúng ta sẽ tạo một ví dụ đơn giãn trong Main để test

import Model.VideoMedia;  
import Notifies.EmailNotification;  
import Notifies.PhoneNotification;  
import Notifies.YoutubeNotification;  
  
public class Main {  
 public static void main(String[] *args*) {  
 VideoMedia videoMedia = new VideoMedia();  
 new EmailNotification(videoMedia);  
 PhoneNotification phoneNotification = new PhoneNotification(videoMedia);  
 YoutubeNotification youtubeNotification = new YoutubeNotification(videoMedia);  
  
 System.out.println("--------------------------------------------------------");  
 videoMedia.setTitle("Design Pattern");  
  
 videoMedia.detachObserver(phoneNotification);  
 videoMedia.detachObserver(youtubeNotification);  
  
 System.out.println("--------------------------------------------------------");  
 videoMedia.setDescription("Design Pattern is good for everyone who want to become senior developer");  
  
 videoMedia.attachObserver(phoneNotification);  
  
 System.out.println("--------------------------------------------------------");  
 videoMedia.setFileName("DS1011");  
 }  
}

- Và ta run code thu kết quả

--------------------------------------------------------

Email Notify Change Data

Title: Design Pattern

Description: null

FileName: null

Phone Notify Change Data

Title: Design Pattern

Description: null

FileName: null

Youtube Notify Change Data

Title: Design Pattern

Description: null

FileName: null

--------------------------------------------------------

Email Notify Change Data

Title: Design Pattern

Description: Design Pattern is good for everyone who want to become senior developer

FileName: null

--------------------------------------------------------

Email Notify Change Data

Title: Design Pattern

Description: Design Pattern is good for everyone who want to become senior developer

FileName: DS1011

Phone Notify Change Data

Title: Design Pattern

Description: Design Pattern is good for everyone who want to become senior developer

FileName: DS1011

- Đoạn code trên ban đầu ta kích hoạt cả 3 notify và khi dữ liệu thay đổi thông báo đã hiển thị.

- Sau đó ta tiến hành detach 2 notify, và sau đó attach 1 notify thì kết quả vẫn như ta mong muốn.

- Từ đó ta có thể thấy ứng dụng của chúng ta hoàn toàn có thể thiết kế để chạy runtime mà không cần phải thay đổi rồi deploy lại như phân tích ban đầu. Chỉ với một trang quảng lý đơn giản, ta sẽ thiết kế các checkbox option cho ứng dụng. Khi người quản trị tick vào các option nào thì chỉ các option đó hoạt động.

- Hơn thế nữa lớp ban đầu của ta VideoMedia hoàn toàn đã encapsulation. Chúng ta không cần phải thay đổi thêm hay xóa trong lớp đó.

3. Singleton Design Pattern

3.1. Phân tích vấn đề

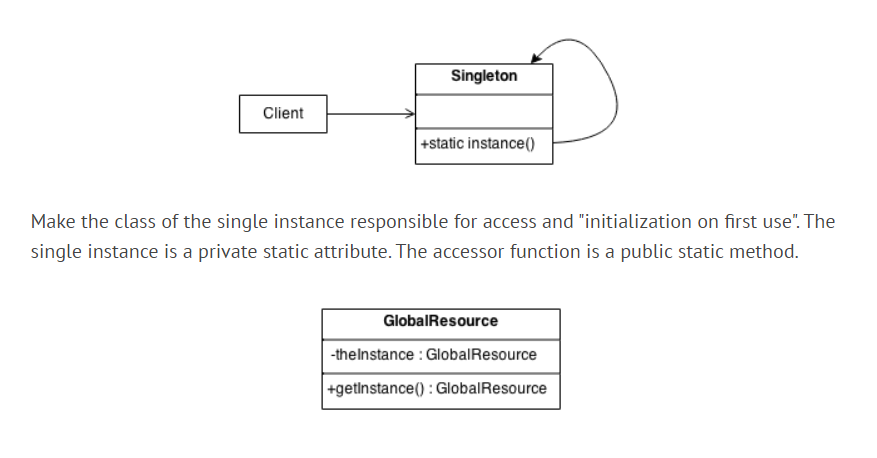
- Singleton được sử dụng khi chúng ta muốn có một instance(thực thể) DUY NHẤT tồn tại trong ứng dụng. Và nó cần được truy cập toàn cục (Globally access). Bên cạnh đó nó có thể được khởi tạo chỉ khi cần (lazy initialization).

- Tuy nhiên singleton có một số vấn đề khi làm việc đa luồng (multi threading) sẽ giải thích ở phần 2

- Các trường hợp tốt nhất sử dụng singleton:

* Các tài nguyên toàn cục (global resource)
* Các cài đặt trong ứng dụng (config)
* Lưu hành vi ứng dụng (logging)
* Lưu bộ nhớ đệm (cache)

3.2. Kiến trúc



Tuy nhiên, sẽ có một vấn đề khi multi threading như ta đã đề cập ở mục trước. Ví dụ

public class GlobalResourceSingleton {  
 GlobalResourceSingleton uniqueInstance;  
  
 public GlobalResourceSingleton() {  
 }  
  
 GlobalResourceSingleton getUniqueInstance() {  
 if (uniqueInstance == null) {  
 uniqueInstance = new GlobalResourceSingleton();  
 }  
 return uniqueInstance;  
 }  
}

Ta sẽ thực thi đoạn getUniqueInstance() với hai thread song song nhau

|  |  |  |
| --- | --- | --- |
| Thread 1 | Thread 2 | uniqueInstance |
| If (uniqueInstane == null) { | If (uniqueInstane == null) { | uniqueInstane == null |
| uniqueInstance = new GlobalResourceSingleton(); | uniqueInstance = new GlobalResourceSingleton(); | uniqueInstance1, uniqueInstance2 |
| Return | Return | uniqueInstance1, uniqueInstance2 |

- Bạn thấy đấy, vấn đề nằm ở đây, các thread chạy song song với nhau nên chúng không thể nào biết được uniqueInstance đã được khởi tạo hay chưa, nếu giả sử ta có 10 thread như vậy ta sẽ tạo ra 10 instance. Tuy nhiên điều đó là không tốt trong quản lý tài nguyên.

- Signleton chỉ mong muốn tạo ra 1 instance DUY NHẤT.

- Chúng ta sẽ khắc phục vấn đề trên ở phần sau.

3.3. Ví dụ

- Ta sẽ tiến hành code một ví dụ sử dụng Singleton

public class GlobalResourceSingleton {  
 private static final GlobalResourceSingleton uniqueInstance = new GlobalResourceSingleton();  
  
 private GlobalResourceSingleton() {  
 }  
  
 public static GlobalResourceSingleton getInstance() {  
 return uniqueInstance;  
 }  
  
 public void sayHelloWorld() {  
 System.out.println("Hello world!");  
 }  
}

- Sau đó ta có thể gọi trong main

public class Main {  
 public static void main(String[] *args*) {  
 GlobalResourceSingleton.**getInstance**().sayHelloWorld();  
 }  
}

- Đây là cách đơn giản để làm, và tất nhiên nếu làm vậy chúng ta không cần quan tâm vấn đề multi threading. Tuy nhiên nó lại không lazy initialization.

- Ta sẽ dùng một cách khác tốt hơn để có thể lazy initialization

public class GlobalResourceSingleton {  
 private static GlobalResourceSingleton **uniqueInstance**;  
  
 private GlobalResourceSingleton() {  
 }  
  
 public static GlobalResourceSingleton getInstance() {  
 if (**uniqueInstance** == null) {  
 **uniqueInstance** = new GlobalResourceSingleton();  
 }  
 return **uniqueInstance**;  
 }  
  
 public void sayHelloWorld() {  
 System.out.println("Hello world!");  
 }  
}

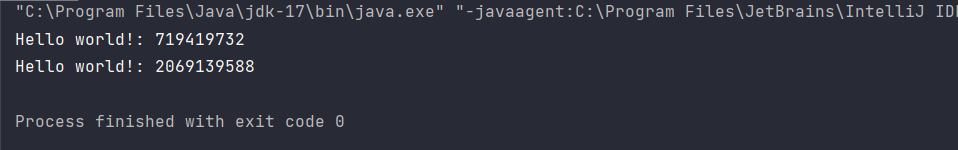
- Tuy nhiên, với cách làm như thế này, ta sẽ gặp vấn đề với multi threading như đã trình bày ở ví dụ 2. Ta có thể thêm một trường index ramdom trong Singleton ta tạo để test, ta sẽ rõ hơn trường hợp này.

import java.util.Random;  
  
public class GlobalResourceSingleton {  
 private static GlobalResourceSingleton **uniqueInstance**;  
 private int index;  
  
 private GlobalResourceSingleton(int *index*) {  
 this.index = *index*;  
 }  
  
 public static GlobalResourceSingleton getInstance() {  
 if (**uniqueInstance** == null) {  
 Random random = new Random();  
 **uniqueInstance** = new GlobalResourceSingleton(random.nextInt());  
 }  
 return **uniqueInstance**;  
 }  
  
 public void sayHelloWorld() {  
 System.out.println("Hello world!: " + index);  
 }  
}

- Trong main ta tiến hành chạy các thread

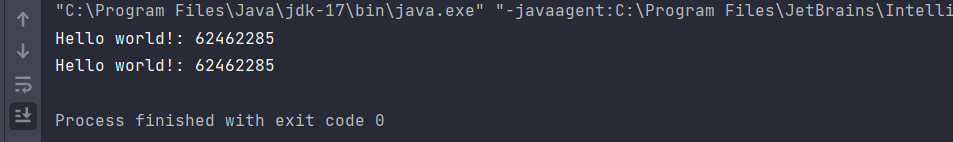
public class Main {  
 public static void main(String[] *args*) {  
 Thread thread1 = new Thread(() -> GlobalResourceSingleton.**getInstance**().sayHelloWorld());  
 Thread thread2 = new Thread(() -> GlobalResourceSingleton.**getInstance**().sayHelloWorld());  
  
 thread1.start();  
 thread2.start();  
 }  
}

- Và ta nhận kết quả



- Các index hoàn toàn khác nhau. Trong Java chúng ta có thể fix trường hợp này với từ khóa **synchronized** , còn trong C# thì có thể dùng lock() để khóa instance

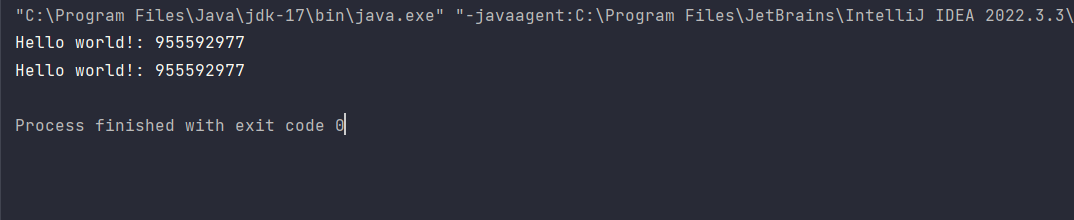
public static synchronized GlobalResourceSingleton getInstance() {  
 if (**uniqueInstance** == null) {  
 Random random = new Random();  
 **uniqueInstance** = new GlobalResourceSingleton(random.nextInt());  
 }  
 return **uniqueInstance**;  
}



- Bằng cách này, ta đã fix được multi threading, tuy nhiên solution này yêu cầu mỗi thread khác phải synchronized trong khi thực tế trong ví dụ này, ta chỉ cần thread1 synchronized.

- Ta sẽ sử dụng cách khác double check synchronized, hoặc cách gọi khác static nested class

import java.util.Random;  
  
public class GlobalResourceSingleton {  
 private static volatile GlobalResourceSingleton **uniqueInstance**;  
 private static final Object lock = new Object();  
 private int index;  
  
 private GlobalResourceSingleton(int *index*) {  
 this.index = *index*;  
 }  
  
 public static GlobalResourceSingleton getInstance() {  
 GlobalResourceSingleton r = **uniqueInstance**;  
 if (r == null) {  
 synchronized (lock) { // while we were waiting lock, another thread may have instantiated the object  
 r = **uniqueInstance**;  
 if (r == null) {  
 Random random = new Random();  
 r = new GlobalResourceSingleton(random.nextInt());  
 **uniqueInstance** = r;  
 }  
 }  
 }  
 return r;  
 }  
  
 public void sayHelloWorld() {  
 System.out.println("Hello world!: " + index);  
 }  
}



- Ta đã giải quyết được vấn đề ban đầu.

- Bạn có thể đọc rõ hơn 3 cách Singleton này trong mục Tài liệu tham khảo [5].

4. Decorator Design Pattern

4.1. Phân tích vấn đề

- Đầu tiên, Decorator là cách để chúng ta thêm các hành vi và trạng thái của đối tượng trong runtime. Decorator cho phép thêm hành vì bằng đặt object vào trong một object khác, và cái “object khác” đó là decorator.

- Decorator nên được ứng dụng khi bạn không thể hoặc không muốn thay đổi object nhưng vẫn muốn thêm hành vi.

- Một lần nữa, mình xin nhấn mạnh DESIGN PATTERN sinh ra là để giải quyết các vấn đề có thể say ra ở hiện tại và trong tương lai của tính năng mà ta code. Nếu nhận thấy nó không cần thiết vui lòng không nên cố gắng PHỨC TẠP HÓA VẤN ĐỀ với design pattern.

- Ví dụ mà mình suy nghĩ và lấy ra ở sau chỉ mang tính chất để hiểu về pattern nay, không hẳn trong thực tế sẽ áp dụng. Vui lòng đọc kỹ lại dòng trên. Chuyện quan trọng nói thêm lần nữa, đọc kỹ dòng trên.

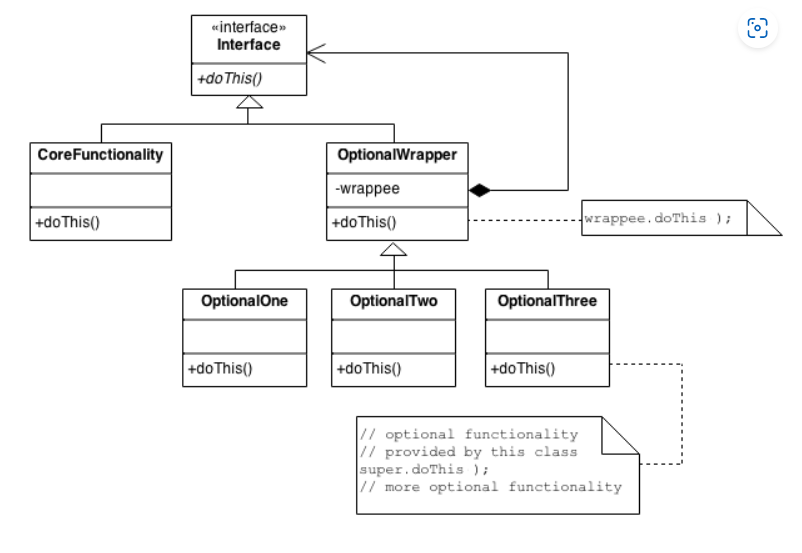
- Ta sẽ lấy ví dụ về Trà sửa đi, một thức uống quen thuộc của mọi người thời nay. Đầu tiên ta có thể đi mua một ly trà sữa, thì lúc đó nhân viên sẽ hỏi “Bạn có muốn thêm topping gì không?”. Đó là vấn đề ta sẽ quan tâm.

- Ta sẽ bắt đầu suy nghĩ thiết kế một hệ thống như vậy

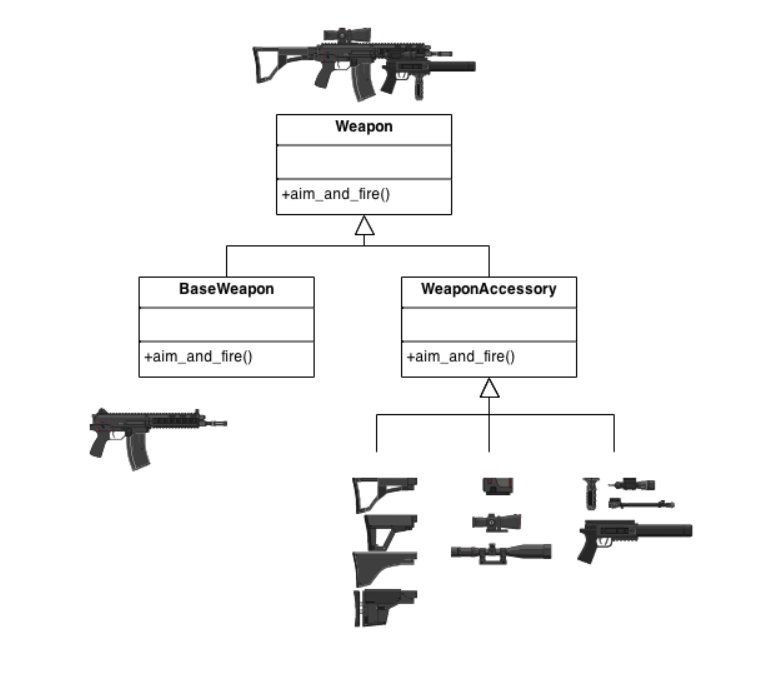
- Đầu tiên ta sẽ có Class MilkTea cho trà sửa. Rồi lúc này ta sẽ tự hỏi, ưm thế một ly trà sửa có topping trân châu đen thì sao ta. Ta sẽ tạo một class BlackBearlMilkTea kế thừa từ MilkTea và có thêm giá để ta override lại phương thức tính tiền + thêm vào. Ơ thế trân châu trắng ta cũng tạo WhitePearlMilkTea, rồi trân châu đường đen, rồi lại các topping khác. Bạn thấy đấy nếu ta muốn một ly trà sửa kết hợp nhiều topping ta thích, thì cách kế thừa của chúng ta sẽ có vấn đề ngay, Vì nếu cứ tạo nhiều class con như vậy thì không biết bao nhiêu class con cho đủ.

- Ta sẽ nghĩ ngay đến các phương hướng khác đó là làm sao để tách các topping ra.

4.2. Kiến trúc



Một ví dụ khá hay cho kiến trúc này



4.3. Ví dụ

- Ví dụ về súng cũng khá hay, nhưng ta sẽ làm trà sữa như ta đã phân tích, để hiểu hơn, mình khuyên bạn sau khi đọc bài của mình cò thể tự tay làm lại ví dụ về súng.

- Đầu tiên ta sẽ tạo một interface chứa các method mà ta cần thêm vào, trong trường hợp của chúng ta, ta cần phương thức tính tiền

package Base;  
  
public interface IMilkTea {  
 int cost();  
}

- Tiếp theo ta sẽ tạo ra lớp MilkTea implement interface

package Base;  
  
public class MilkTea implements IMilkTea {  
 private int price;  
  
 public MilkTea(int *price*) {  
 this.price = *price*;  
 }  
  
 public int getPrice() {  
 return price;  
 }  
  
 public void setPrice(int *price*) {  
 this.price = *price*;  
 }  
  
 *@Override* public int cost() {  
 return getPrice();  
 }  
}

- Sau đó ta tiến hành tạo wrapper, wrapper này là một abtract class sẽ implement interface, trong wrapper ta sẽ có một contructor nhận vào inner (đối tượng bên trong).

package Base;  
  
public abstract class MilkTeaDecorator implements IMilkTea {  
 private IMilkTea milkTea;  
  
 protected MilkTeaDecorator(IMilkTea *inner*) {  
 this.milkTea = *inner*;  
 }  
 *@Override* public int cost() {  
 return milkTea.cost();  
 }  
}

- Tiếp đó ta chỉ cần tạo các topping kế thừa MilkTeaDecorator và override lại cost + thêm tiền các topping

package Decorators;  
  
import Base.IMilkTea;  
import Base.MilkTeaDecorator;  
  
public class BlackPearl extends MilkTeaDecorator {  
 private int price = 5000;  
  
 public BlackPearl(IMilkTea *inner*) {  
 super(*inner*);  
 }  
  
 *@Override* public int cost() {  
 return getPrice() + super.cost();  
 }  
  
 public int getPrice() {  
 return price;  
 }  
  
 public void setPrice(int *price*) {  
 this.price = *price*;  
 }  
}

package Decorators;  
  
import Base.IMilkTea;  
import Base.MilkTeaDecorator;  
  
public class BlackSugar extends MilkTeaDecorator {  
 private int price = 4000;  
 public BlackSugar(IMilkTea *inner*) {  
 super(*inner*);  
 }  
  
 *@Override* public int cost() {  
 return getPrice() + super.cost();  
 }  
  
 public int getPrice() {  
 return price;  
 }  
  
 public void setPrice(int *price*) {  
 this.price = *price*;  
 }  
}

package Decorators;  
  
import Base.IMilkTea;  
import Base.MilkTeaDecorator;  
  
public class EggPudding extends MilkTeaDecorator {  
 private int price = 6000;  
 public EggPudding(IMilkTea *inner*) {  
 super(*inner*);  
 }  
  
 *@Override* public int cost() {  
 return getPrice() + super.cost();  
 }  
  
 public int getPrice() {  
 return price;  
 }  
  
 public void setPrice(int *price*) {  
 this.price = *price*;  
 }  
}

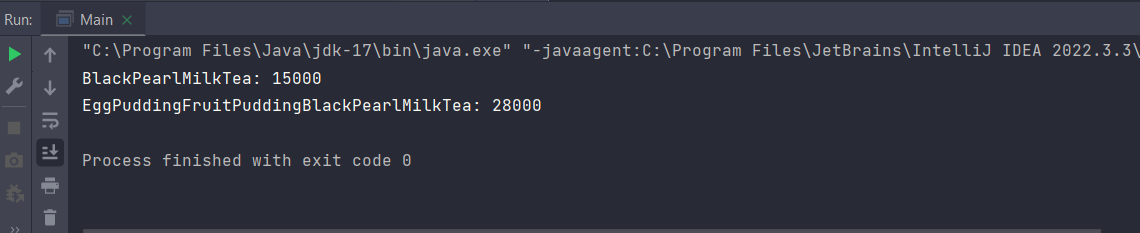
package Decorators;  
  
import Base.IMilkTea;  
import Base.MilkTeaDecorator;  
  
public class FruitPudding extends MilkTeaDecorator {  
 private int price = 7000;  
 public FruitPudding(IMilkTea *inner*) {  
 super(*inner*);  
 }  
  
 *@Override* public int cost() {  
 return getPrice() + super.cost();  
 }  
  
 public int getPrice() {  
 return price;  
 }  
  
 public void setPrice(int *price*) {  
 this.price = *price*;  
 }  
}

package Decorators;  
  
import Base.IMilkTea;  
import Base.MilkTeaDecorator;  
  
public class WhitePearl extends MilkTeaDecorator {  
 private int price = 6000;  
 public WhitePearl(IMilkTea *inner*) {  
 super(*inner*);  
 }  
  
 *@Override* public int cost() {  
 return getPrice() + super.cost();  
 }  
  
 public int getPrice() {  
 return price;  
 }  
  
 public void setPrice(int *price*) {  
 this.price = *price*;  
 }  
}

- Và cuối cùng ta tiến hành test nó trong main,

import Base.MilkTea;  
import Decorators.BlackPearl;  
import Decorators.EggPudding;  
import Decorators.FruitPudding;  
  
public class Main {  
 public static void main(String[] *args*) {  
 var firstMilkTea = new BlackPearl(  
 new MilkTea(10000));  
 System.out.println("BlackPearlMilkTea: " + firstMilkTea.cost());  
  
 var secondMilkTea = new EggPudding(  
 new FruitPudding(  
 new BlackPearl(  
 new MilkTea(10000))));  
 System.out.println("EggPuddingFruitPuddingBlackPearlMilkTea: " + secondMilkTea.cost());  
 }  
}

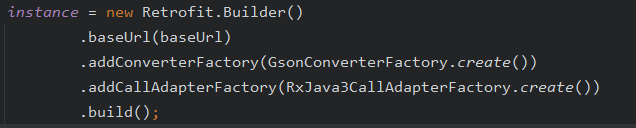
- Ta thu được giá tiền hai ly trà sữa:



5. Builder Design Pattern

- Pattern này là pattern được ứng dụng và xử dụng rất thường xuyên trong thực tế ngày nay.

Ví dụ như



- Đây là một đoạn new instance của Retrofit. Đối tượng này chứa nhiều thuộc tính phức tạp như baseUrl, GsonConverterFactory, RxJava3CallAdapterFactory. Nếu ta tạo constructor thông thường và new sau đó truyền các thuộc tính vào thì bạn thấy có dễ hiểu bằng cách này không.

- Mục đích của pattern này sẽ giúp cho code của chúng ta dễ hiểu hơn, rõ ràng hơn.

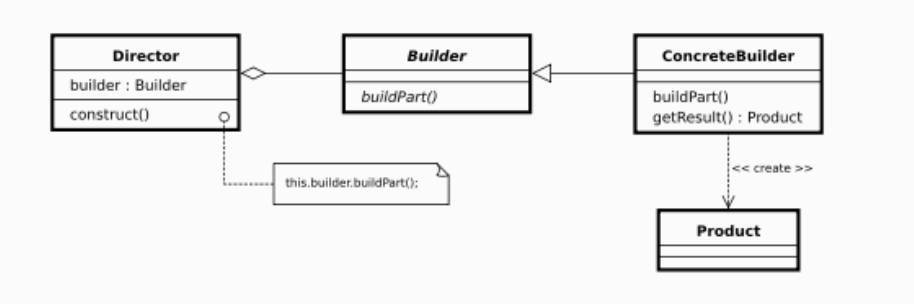
- Chúng ta sẽ lấy một ví dụ: Với một đối tượng phức tạp như Sinh viên chẳng hạn, thì sinh viên sẽ có id, name, age, address, … Ngoài ra còn có các thuộc tính phức tạp khác như Mark( n môn), Class, Major. Như vậy bạn thử tưởng tượng xem khi bạn tiến hành tạo constructor để new một sinh viên, thì sẽ như thế nào. Có phải nó sẽ trông rất chi là dài, và khi đọc các thuộc tính truyền vào, nó là các chuổi string chẳng hạn, bạn có thể hiểu, nhưng những người khác nhìn vào có dễ gây lú cho họ không.

import models.\*;  
  
public class Main {  
 public static void main(String[] *args*) {  
 Student student = new Student(  
 "Tai",  
 21,  
 "HCM",  
 new StudentClass("201101A"),  
 new Major("Software Technology", new Faculty("Information technology")),  
 new Mark(9, 8)  
 );  
 }  
}

- Đây là một đoạn code mình tự tạo cho ví dụ đó, tất nhiên đối tượng sinh viên quá quen thuộc với tất cả chúng ta nên bạn nhìn vào mặt dù thấy cũng hơi loằn ngoằn nhưng vẫn có thể hiểu được.

- Tuy nhiên điều đáng nói ở đây là nếu đối tượng có 10 thuộc tính String và nhiều thuộc tính phức tạp khác thì mọi chuyện nhìn sẽ cực kỳ phức tạp.

- Giờ, ta sẽ tham khảo kiến trúc của Builder Design Pattern để thử sửa lại code của chúng ta.



- Oke, đầy tiên tà tiến hành tạo một IstudentBuilder để khai báo trước cái hàm

package builders;  
  
import models.Major;  
import models.Mark;  
import models.Student;  
import models.StudentClass;  
  
public interface IStudentBuilder {  
 public StudentBuilder name(String *name*);  
 public StudentBuilder age(int *age*);  
 public StudentBuilder address(String *address*);  
 public StudentBuilder addStudentClass(StudentClass *studentClass*);  
 public StudentBuilder addMajor(Major *major*);  
 public StudentBuilder addMark(Mark *mark*);  
 public Student build();  
}

- Sau đó ta tạo một StudentBuilder implements từ interface trên

package builders;  
  
import models.Major;  
import models.Mark;  
import models.Student;  
import models.StudentClass;  
  
public class StudentBuilder implements IStudentBuilder{  
 private String name;  
 private int age;  
 private String address;  
 private StudentClass studentClass;  
 private Major major;  
 private Mark mark;  
  
 *@Override* public StudentBuilder name(String *name*) {  
 this.name = *name*;  
 return this;  
 }  
  
 *@Override* public StudentBuilder age(int *age*) {  
 this.age = *age*;  
 return this;  
 }  
  
 *@Override* public StudentBuilder address(String *address*) {  
 this.address = *address*;  
 return this;  
 }  
  
 *@Override* public StudentBuilder addStudentClass(StudentClass *studentClass*) {  
 this.studentClass = *studentClass*;  
 return this;  
 }  
  
 *@Override* public StudentBuilder addMajor(Major *major*) {  
 this.major = *major*;  
 return this;  
 }  
  
 *@Override* public StudentBuilder addMark(Mark *mark*) {  
 this.mark = *mark*;  
 return this;  
 }  
  
 *@Override* public Student build() {  
 return new Student(name, age, address, studentClass, major, mark);  
 }  
}

- Cuối cùng ta sử dụng nó trong Main

import builders.StudentBuilder;  
import models.\*;  
  
public class Main {  
 public static void main(String[] *args*) {  
 Student student = new Student(  
 "Tai",  
 21,  
 "HCM",  
 new StudentClass("201101A"),  
 new Major("Software Technology", new Faculty("Information technology")),  
 new Mark(9, 8)  
 );  
 System.out.println(student.toString());  
  
 var student2 = new StudentBuilder()  
 .name("Tai")  
 .age(21)  
 .address("HCM")  
 .addStudentClass(new StudentClass("201101A"))  
 .addMajor(new Major("Software Technology", new Faculty("Information technology")))  
 .addMark(new Mark(9, 8))  
 .build();  
  
  
 System.out.println(student2.toString());  
 }  
}

- Và giờ như bạn thấy đấy, code của chúng ta đã trông dễ đọc hơn nhiều dù nó dài hơn. Ngoài ra ta cũng có thể set default = null cho các thuộc tính ví dụ address. Thì ta không cần phải gọi .address thì code vẫn hoạt động bình thường.



- Đấy là cách mà các buider ngày nay sử dụng. Bằng cách set cho các thuộc tính các giá trị mặc định ta có thể chỉ cần truyền các tham số bắt buộc, ngoài ra code vẫn không lỗi. Ứng dụng này của Builder cũng góp phần giải quyết bài toàn yêu cầu quá nhiều constructor trong khi có nhiều thuộc tính.

- Tổng kết lại vẫn là câu nói đó Design Pattern sinh ra để giải quyết vấn đề chứ không làm phức tạp hóa vấn đề, vì vậy nếu nhận thấy cần thiết thì mới dùng.

6. Factory Method Design Pattern

- Define một interface để khởi tạo đối tượng, nhưng để subclass quyết định lớp khởi tạo.

- Factory Method giúp cho thiết kế có thể tùy chỉnh hơn, nhưng đồng thời cũng phức tạp hơn.

- Mọi người sử dụng Factory Method như một cách tiêu chuẩn đeể tạo object, nhưng nó sẽ không cần thiết nếu:

+ Lớp mà instance của nó không thay đổi

+ instance có thể bị ghi đè

- Chúng ta sẽ có một tình huống để mọi người có thể dễ hiểu Design Pattern này.

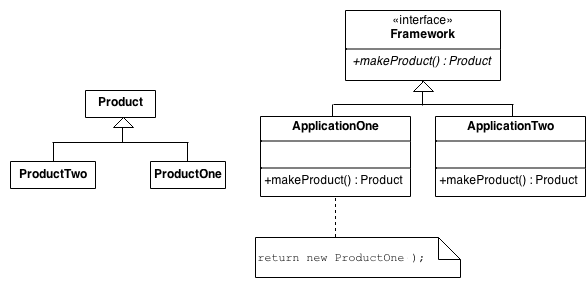
- Ví dụ chúng ta đang có một app tương tự như momo, khi người dùng nạp tiền đến một hạn mức nhất định nào đó thì người dùng sẽ có thẻ đồng, bạc, vàng, kim cương, tương ứng.

- Giả sử như ta có các lớp của thẻ, BronzeCard, SliverCart, GoldCart, DiamondCard.

- Khi người dùng nạp vào 10 tr thì là thẻ đồng, 100tr thì bạc, 1 tỳ thì vàng, 100 tỷ thì kim cương. Vậy trong runtime ta chỉ có số tiền người dùng nộp vào, ta khởi tạo các Object này như thế nào?

- Factory Method có thể giải quyết vấn đề này.

- Ta sẽ đi đến kiến trúc của nó



- Ta sẽ tiến hành code ví dụ ta nói ở trên dựa trên kiến trúc của nó.

- Đầu tiên ta định nghĩa một interface Card, ta ta khai báo phương thức getName để tiện test.

package card;  
  
public interface Card {  
 public void getName();  
}

- Tiếp theo ta tạo các loại card implement từ Card

package card;  
  
public class BasicCard implements Card{  
 *@Override* public void getName() {  
 System.out.println("Basic Card");  
 }  
}

package card;  
  
public class BronzeCard implements Card{  
 *@Override* public void getName() {  
 System.out.println("BronzeCard");  
 }  
}

package card;  
  
public class SliverCard implements Card{  
 *@Override* public void getName() {  
 System.out.println("SliverCard");  
 }  
}

package card;  
  
public class GoldCard implements Card{  
 *@Override* public void getName() {  
 System.out.println("GoldCard");  
 }  
}

package card;  
  
public class DiamondCard implements Card{  
 *@Override* public void getName() {  
 System.out.println("DiamondCard");  
 }  
}

- Sau đó ta lại tạo một interface factory, và định nghĩa phương thức createCard

package factory;  
  
import card.Card;  
  
public interface ICardFactory {  
 Card createCard(Long *price*);  
}

- Ta sẽ tạo một BasicCardFactory

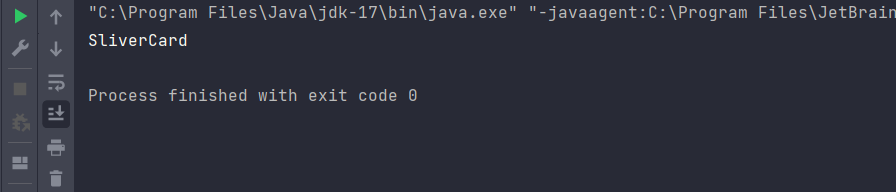
package factory;  
  
import card.\*;  
  
public class BasicCardFactory implements ICardFactory {  
  
 *@Override* public Card createCard(Long *price*) {  
 if (*price* >= 100000000000L) {  
 return new DiamondCard();  
 } else if (*price* >= 1000000000L) {  
 return new GoldCard();  
 } else if (*price* >= 100000000L) {  
 return new SliverCard();  
 } else if (*price* >= 10000000L) {  
 return new BronzeCard();  
 }  
 return new BasicCard();  
 }  
}

- Đây là cách thông thường, ngoài ra nếu có chương trình đặc biệt gì đó, khuyến mãi chẳng hạng ví dụ như chỉ cần 1 tỷ là lên thẻ kim cương, thì ta sẽ tạo một SpecialCardFactory implements IcardFactory và định nghĩa createCard như ta mong muốn.

- Ta tiếp tục vào main để test

import card.Card;  
import factory.BasicCardFactory;  
  
public class Main {  
 public static void main(String[] *args*) {  
 BasicCardFactory basicCardFactory = new BasicCardFactory();  
 Card card = basicCardFactory.createCard(100000000L);  
 card.getName();  
 }  
}

- Ta sẽ thu được kết quả mong muốn:



- Để hiểu hơn, bạn có thể làm thêm một Factory Special như mình đề cập ở trên.

7. Abstract Factory Design Pattern

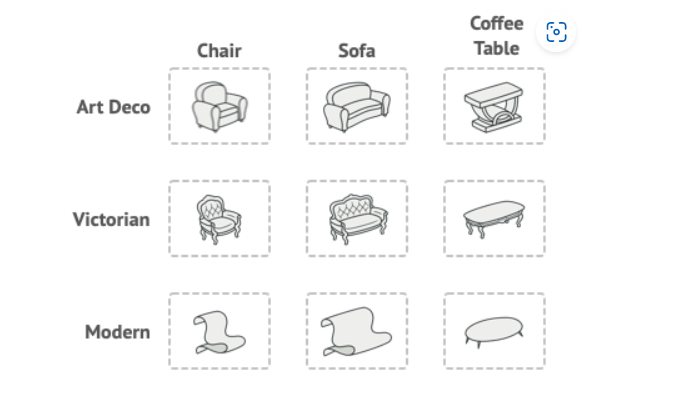
- Abstract Factory là một design pattern thuộc nhóm Creational Pattern – những mẫu thiết kế cho việc khởi tạp đối tượng.

- Nó được xây dựng dựa trên Factory Method ở phần trước, và được xem là một factory cao nhất trong hệ thống

- Ta tiến hành phân tích một ví dụ trên viblo [9].

- Ta có một trang bán nội thất với các sản phẩm Sofar, Chair, CoffeeTable.

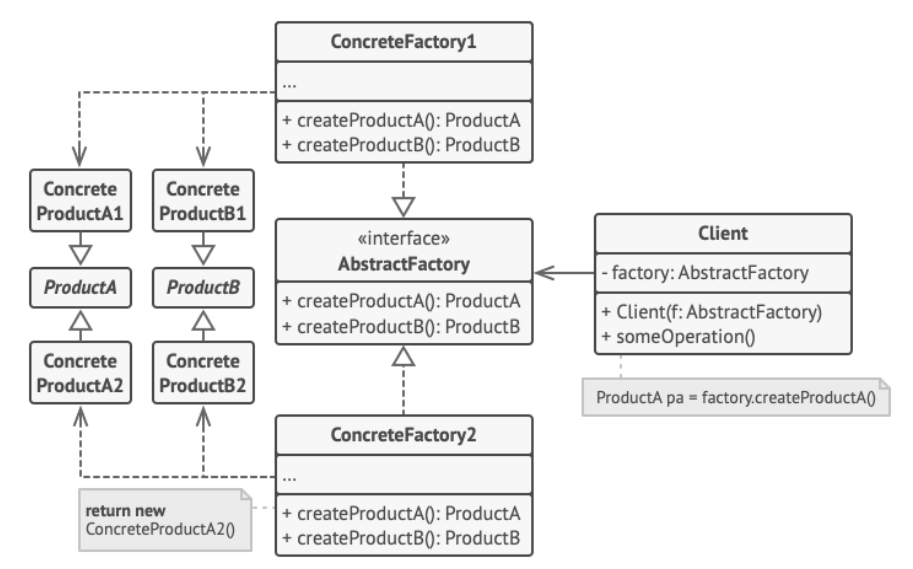
- Ta cũng có các biến thể của nó như : ArtDeco, Victorian, Modern



- Mục đích của ta là tạo ra một đồ nội thất đơn lẽ và phải phù hợp với các đồ vật khác: ví dụ tôi muốn 2 chiếc ghế trong nhóm Victorian, và tôi muốn thêm một chiếc sofa.

- Tôi sẽ không hài lòng nếu bạn gửi cho tôi 1 chiếc soa trong nhóm Arrt Deco

- Chúng ta sẽ tiến hành code thử giải pháp theo kiến trúc của Abstract Factory



- Việc đầu tiên ta sẽ tiến hành khai báo interface và tạo các biến thể của sản phẩm.

package Product;  
  
public interface Chair {  
 void getName();  
}

package Product;  
  
public interface Sofa {  
 void getName();  
}

package Product;  
  
public class ArtDecoChair implements Chair{  
  
 *@Override* public void getName() {  
 System.out.println("I'm ArtDecoChair");  
 }  
}

package Product;  
  
public class ModernChair implements Chair{  
  
 *@Override* public void getName() {  
 System.out.println("I'm ModernChair");  
 }  
}

package Product;  
  
public class VictorianChair implements Chair{  
 *@Override* public void getName() {  
 System.out.println("I'm VictorianChair");  
 }  
}

package Product;  
  
public class ArtDecoSofa implements Sofa{  
 *@Override* public void getName() {  
 System.out.println("I'm ArtDecoSofa");  
 }  
}

package Product;  
  
public class ModernSofa implements Sofa{  
 *@Override* public void getName() {  
 System.out.println("I'm ModernSofa");  
 }  
}

package Product;  
  
public class VictorianSofa implements Sofa{  
 *@Override* public void getName() {  
 System.out.println("I'm VictorianSofa");  
 }  
}

- Tiếp theo ta tạo một FurnitureFactory

package Factory;  
  
import Product.Chair;  
import Product.Sofa;  
  
public interface FurnitureFactory {  
 Chair createChair();  
 Sofa createSofa();  
}

- Sau đó ta tạo các factory dựa trên interface trên

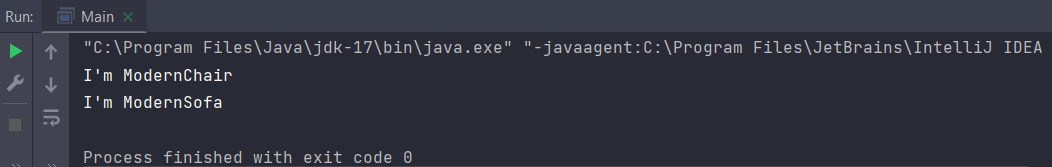
package Factory;  
  
import Product.ArtDecoChair;  
import Product.ArtDecoSofa;  
import Product.Chair;  
import Product.Sofa;  
  
public class ArtDecoFurnitureFactory implements FurnitureFactory{  
  
 *@Override* public Chair createChair() {  
 return new ArtDecoChair();  
 }  
  
 *@Override* public Sofa createSofa() {  
 return new ArtDecoSofa();  
 }  
}

package Factory;  
  
import Product.Chair;  
import Product.ModernChair;  
import Product.ModernSofa;  
import Product.Sofa;  
  
public class ModernFurnitureFactory implements FurnitureFactory{  
  
 *@Override* public Chair createChair() {  
 return new ModernChair();  
 }  
  
 *@Override* public Sofa createSofa() {  
 return new ModernSofa();  
 }  
}

package Factory;  
  
import Product.Chair;  
import Product.Sofa;  
import Product.VictorianChair;  
import Product.VictorianSofa;  
  
public class VictorianFurnitureFactory implements FurnitureFactory{  
 *@Override* public Chair createChair() {  
 return new VictorianChair();  
 }  
  
 *@Override* public Sofa createSofa() {  
 return new VictorianSofa();  
 }  
}

- Trong Main ta tạo để test

import Factory.ModernFurnitureFactory;  
import Product.Chair;  
import Product.Sofa;  
  
public class Main {  
 public static void main(String[] *args*) {  
 ModernFurnitureFactory modernFurnitureFactory = new ModernFurnitureFactory();  
 Chair chair = modernFurnitureFactory.createChair();  
 chair.getName();  
 Sofa sofa = modernFurnitureFactory.createSofa();  
 sofa.getName();  
 }  
}



- Với cách làm này, ta có thể dễ dàng thêm một sản phẩm mới vào cửa hàng nội thất mà không tác động đến các sản phẩm khác.

- Cách này sẽ phù hợp với ứng dụng có các đối tượng thuộc cùng nhóm cùng họ với nhau, như trong ví dụ này có các ChairModern và SofaModern cùng nhóm.

TÀI LIỆU THAM KHẢO

[1]. Ông Dev. (2019). Design Pattern. Link: <https://www.youtube.com/@ongdev>

[2]. Design Patterns and Refactoring. (2023). Retrieved May 4, 2023, from Sourcemaking.com website: <https://sourcemaking.com/design_patterns/strategy>

[3]. Design Patterns and Refactoring. (2023). Retrieved May 5, 2023, from Sourcemaking.com website: <https://sourcemaking.com/design_patterns/observer>

[4]. Design Patterns and Refactoring. (2023). Retrieved May 10, 2023, from Sourcemaking.com website: <https://sourcemaking.com/design_patterns/singleton>

[5]. RickDavis. (2012, June 23). Java Singleton and Synchronization. Retrieved May 10, 2023, from Stack Overflow website: <https://stackoverflow.com/questions/11165852/java-singleton-and-synchronization>

[6]. Design Patterns and Refactoring. (2023). Retrieved May 11, 2023, from Sourcemaking.com website: <https://sourcemaking.com/design_patterns/decorator>

[7]. Gupta, L. (2014, May 9). Builder Design Pattern - HowToDoInJava. Retrieved May 13, 2023, from HowToDoInJava website: <https://howtodoinjava.com/design-patterns/creational/builder-pattern-in-java/>

[8]. Design Patterns and Refactoring. (2023). Retrieved May 13, 2023, from Sourcemaking.com website: <https://sourcemaking.com/design_patterns/builder>

‌[9]. Ren. (2022, March 8). Design Pattern - Abstract Factory - Viblo. Retrieved May 20, 2023, from Viblo website: <https://viblo.asia/p/design-pattern-abstract-factory-gAm5yEvV5db>

‌[10]. Hoàng Đinh. (2021, November 21). Tổng quan về Abstract Factory Design Pattern. Retrieved May 20, 2023, from Viblo website: <https://viblo.asia/p/abstract-factory-design-pattern-tro-thu-dac-luc-cua-developers-maGK7B4M5j2>

‌

[11]. Design Patterns and Refactoring. (2023). Retrieved May 20, 2023, from website: <https://sourcemaking.com/design_patterns/abstract_factory>

‌[12].

‌