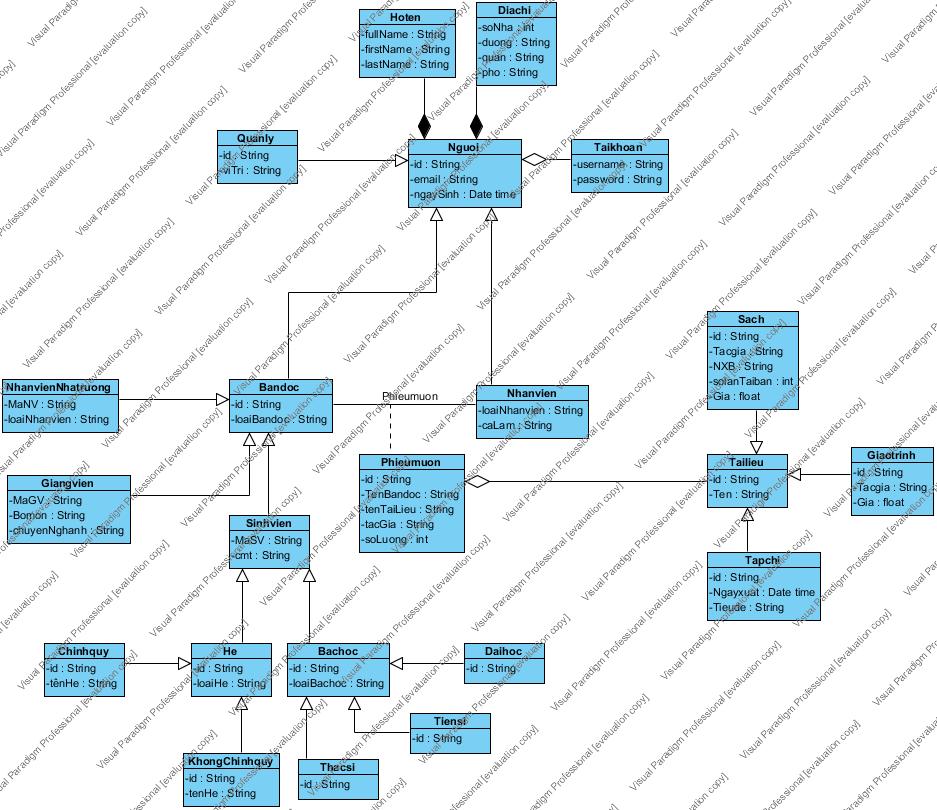
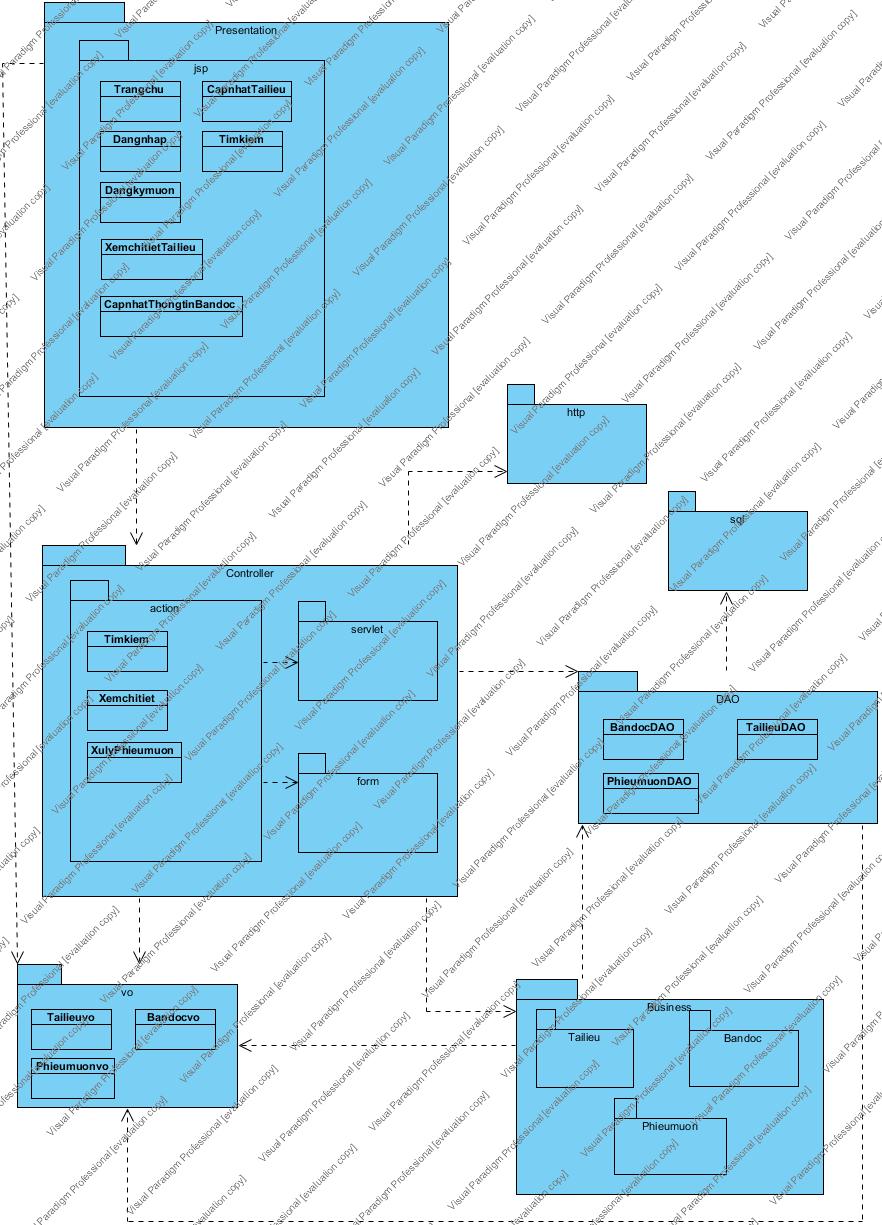
**ĐỀ 4: QL THƯ VIỆN (LibMan)**

**Câu 1:**

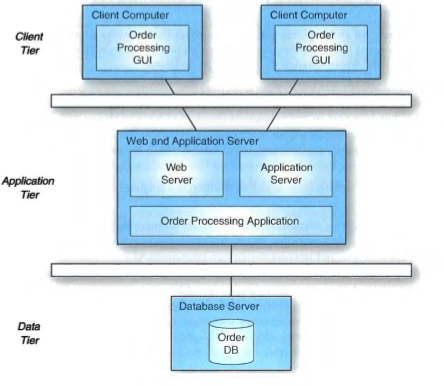
Các lớp thực thể (pakage Models) và quan hệ:

****

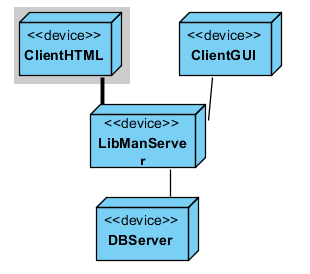
Các gói:

****

Kiến trúc 3 tầng:



Biểu đồ triển khai:



**Câu 2:**

- Các bean thực thể:

Nguoi, Bandoc, Nhanvien, …. (Câu 1)

- Code java của bean thực thể:

***Nguoi.java***

@Entity

@Table(name = "nguoi")

@XmlRootElement

@NamedQueries({

@NamedQuery(name = "Nguoi.findAll", query = "SELECT n FROM Nguoi n"),

@NamedQuery(name = "Nguoi.findById", query = "SELECT n FROM Nguoi n WHERE n.id = :id"),

@NamedQuery(name = "Nguoi.findByEmail", query = "SELECT n FROM Nguoi n WHERE n.email = :email"),

@NamedQuery(name = "Nguoi.findBySdt", query = "SELECT n FROM Nguoi n WHERE n.sdt = :sdt"),

@NamedQuery(name = "Nguoi.findByHoTen", query = "SELECT n FROM Nguoi n WHERE n.hoTen = :hoTen"),

@NamedQuery(name = "Nguoi.findByDiaChi", query = "SELECT n FROM Nguoi n WHERE n.diaChi = :diaChi"),

@NamedQuery(name = "Nguoi.findByNgaySinh", query = "SELECT n FROM Nguoi n WHERE n.ngaysinh = :ngaysinh"),

@NamedQuery(name = "Nguoi.findByAccount", query = "SELECT n FROM Nguoi n WHERE n.account = :account")})

public class Nguoi implements Serializable {

private static final long serialVersionUID = 1L;

@Id

@Basic(optional = false)

@NotNull

@Size(min = 1, max = 255)

@Column(name = "Id")

private String id;

@Basic(optional = false)

@NotNull

@Size(min = 1, max = 255)

@Column(name = "NgaysinhId")

private String ngaysinhId;

@Size(max = 255)

@Column(name = "Email")

private String email;

@Size(max = 255)

@Column(name = "Sdt")

private String sdt;

@Column(name = "HoTen")

private Integer hoTen;

@Column(name = "DiaChi")

private Integer diaChi;

@Column(name = "NgaySinh")

private Integer ngaySinh;

@Column(name = "Account")

private Integer account;

public Nguoi() {

}

Get();

Set();

@Override

public int hashCode() {

int hash = 0;

hash += (id != null ? id.hashCode() : 0);

return hash;

}

}

Các bean khác tương tự

**Câu 3:**

1. *2 loại sesssion bean và ý nghĩa*

- Stateless Session beans: Stateless là design không lưu dữ liệu của client trên server. Có nghĩa là sau khi client gửi dữ liệu lên server, server thực thi xong, trả kết quả thì “quan hệ” giữa client và server bị “cắt đứt” - server không lưu bất cứ dữ liệu gì của client. Như vậy, khái niệm “trạng thái” ở đây được hiểu là dữ liệu và cụm “phi trạng thái” có thể làm nhiều người hiểu sai bản chất vấn đề.

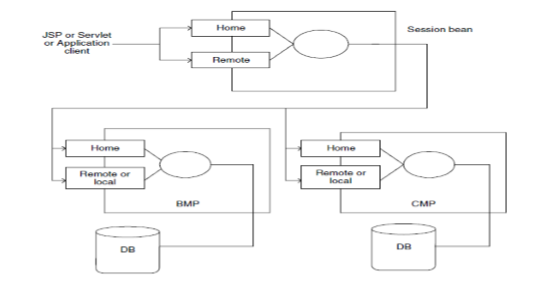
- Stateful Session beans: Stateful là một design ngược với stateless, server cần lưu dữ liệu của client, điều đó đồng nghĩa với việc ràng buộc giữa client và server vẫn được giữ sau mỗi request (yêu cầu) của client. Data được lưu lại phía server có thể làm đầu vào (input parameters) cho lần kế tiếp, hoặc là dữ kiện dùng trong quá trình xử lý hay phục phụ cho bất cứ nhu cầu nào phụ thuộc vào bussiness (nghiệp vụ) cài đặt.

- Ý nghĩa: Nên sử dụng stateless khi không cần lưu giữ trạng thái. Ngược lại thì dùng stateful.

1. *Lựa chọn session bean, biểu đồ kết nối với entity bean, code java*

Chọn stateless vì trong hệ thống không cần lưu trạng thái.

Biểu đồ kết nối:



**Code java cho session bean**:

@Remote

Public interface NguoiRemote {

Void create(Nguoi n);

Void edit(Nguoi n);

Void remove(Nguoi n);

// …

}

@Stateless

Public class NguoiBean implement NguoiRemote {

Private EntityManager em;

@Override

Public void create(Nguoi n) {…}

@Override

Public void edit(Nguoi n) {…}

@Override

Public void remove(Nguoi n) {…}

}

**Câu 4: Pattern (định nghĩa, biểu đồ, ý nghĩa sd, code java)**

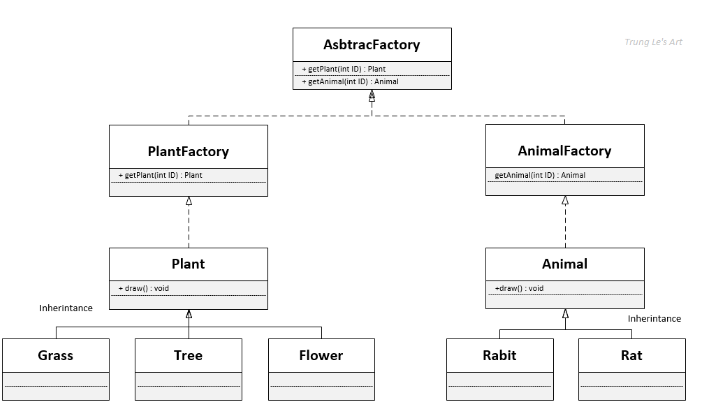
1. **Abstract Factory**

**Định nghĩa:** Abstract Factory, đầy đủ là Abstract Factory pattern, là [thiết kế mẫu](https://vi.wikipedia.org/wiki/M%E1%BA%ABu_thi%E1%BA%BFt_k%E1%BA%BF_(khoa_h%E1%BB%8Dc_m%C3%A1y_t%C3%ADnh)) hướng đối tượng trong việc thiết kế [phần mềm](https://vi.wikipedia.org/wiki/Ph%E1%BA%A7n_m%E1%BB%81m) cho [máy tính](https://vi.wikipedia.org/wiki/M%C3%A1y_t%C3%ADnh), cung cấp một [giao diện lớp](https://vi.wikipedia.org/w/index.php?title=Giao_di%E1%BB%87n_l%E1%BB%9Bp&action=edit&redlink=1) có chức năng tạo ra một [tập hợp](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_h%E1%BB%A3p) các đối tượng liên quan hoặc phụ thuộc lẫn nhau mà không chỉ ra đó là những [lớp](https://vi.wikipedia.org/wiki/L%E1%BB%9Bp) cụ thể nào tại thời điểm [thiết kế](https://vi.wikipedia.org/wiki/Thi%E1%BA%BFt_k%E1%BA%BF). Mẫu thiết kế Abstract Factory đóng gói một nhóm những lớp đóng vai trò "sản xuất" (Factory) trong ứng dụng, đây là những lớp được dùng để tạo lập các đối tượng. Các lớp sản xuất này có chung một giao diện lập trình được kế thừa từ một lớp cha thuần ảo gọi là "lớp sản xuất ảo".

**Ý nghĩa sử dụng**:

* Cho phép khởi tạo các họ các đối tượng mà không nêu rõ một lớp cụ thể
* Sử dụng khi có nhiều đối tượng có nhiều biến đổi trong khoảng thời gian chạy
* Mô hình hóa các đối tượng thông qua một interface chung.

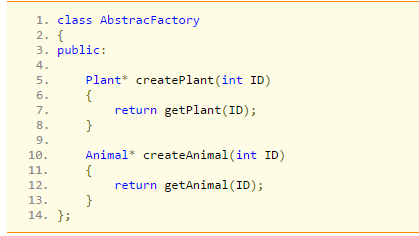
**Biểu đồ UML**

****

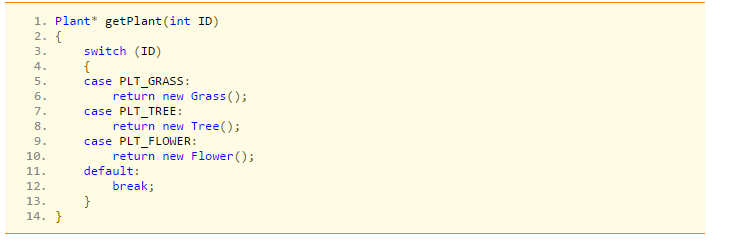
**Code:**

**Khởi tạo các interface Grass, Tree và các , Square,.. implement từ hai interface trên**

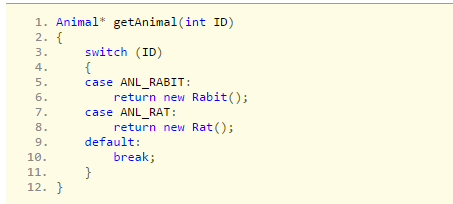
\***AbstractFactory**



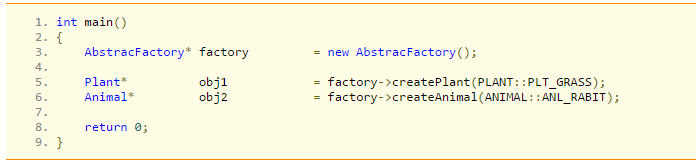
**\*PlantFactory**

****

**\*AnimalFactory**

****

**\*Gọi factory trong hàm main()**



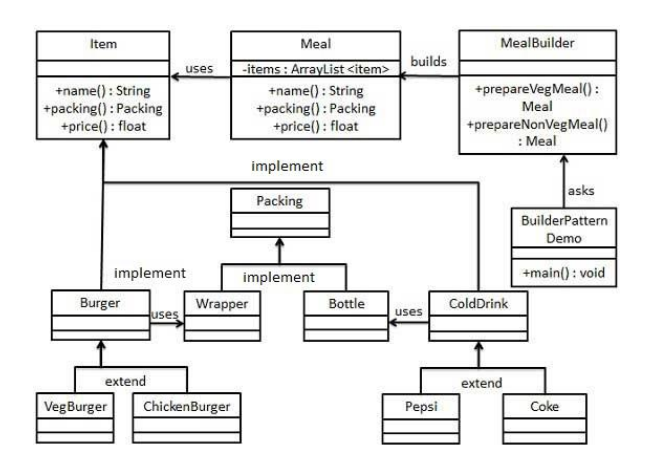
1. **Builder**

**Định nghĩa:** Builder cho phép xây dựng một đối tượng phức tạp bằng việc sử dụng các đối tượng đơn giản và sử dụng cách tiếp cận từng bước một. Mẫu thiết kế được tạo ra theo mô hình creational vì nó cung cấp một trong những cách tốt nhất để tạo ra một đối tượng.

**Ý nghĩa sử dụng**:

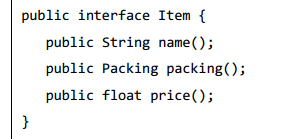
* Xây dựng một đối tượng được tạo thành từ các đối tượng khác
* Các phần được tạo ra là độc lập với đối tượng chính
* Che giấu việc tạo các phần của đối tượng đối với người dùng
* Chỉ có builder hiểu được những đặc điểm kỹ thuật

**Biểu đồ UML**

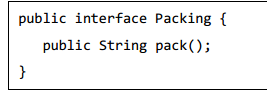
****

**Code java:**

**Item.java:**

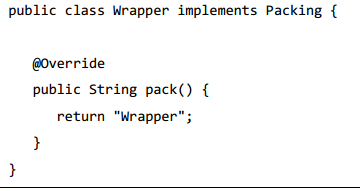
****

**Packing.java**

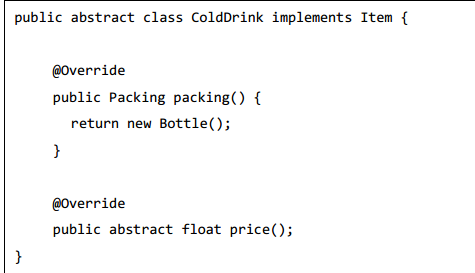
****

**Wrapper.java**

**Tạo các class wrapper, bottle implement Packing:**

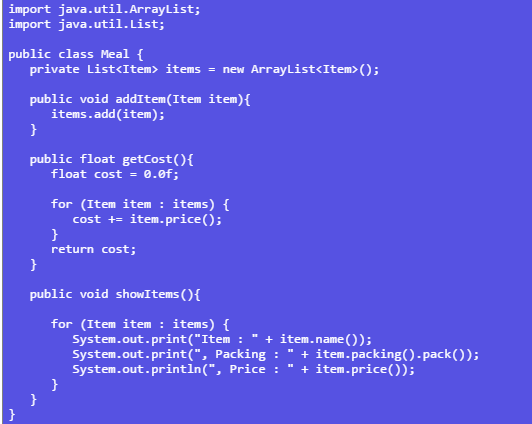
****

**Tạo các class Burger, colddrink implement Item:**

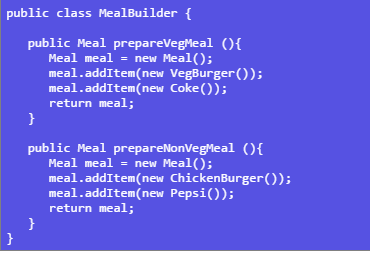
****

**Tương tự tạo các class VegBurger và ChickenBurger extends Burger. Tạo class pessi, coke extends ColdDrink**

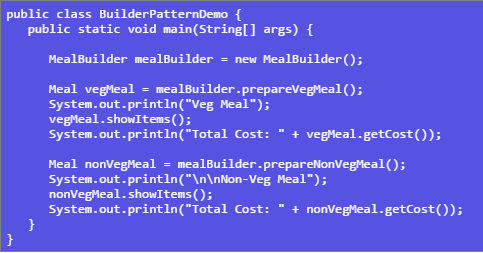
**Tạo Meal:**

****

*MealBuilder.java*

****

*BuilderPatternDemo.java*



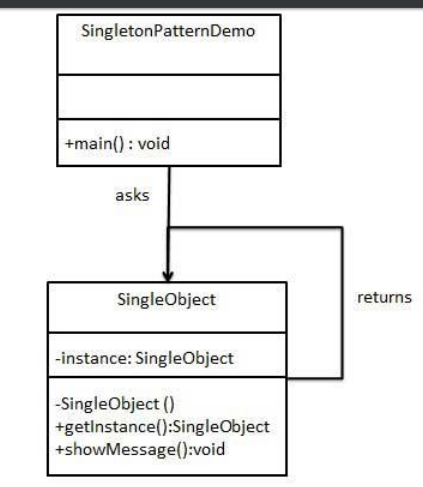
1. **Singleton**

**Định nghĩa:** Singleton Pattern là pattern đảm bảo rằng một lớp chỉ có một thể hiện (instance) duy nhất và trong đó cung cấp một cổng giao tiếp chung nhất để truy cập vào lớp đó.

**Ý nghĩa sử dụng**:

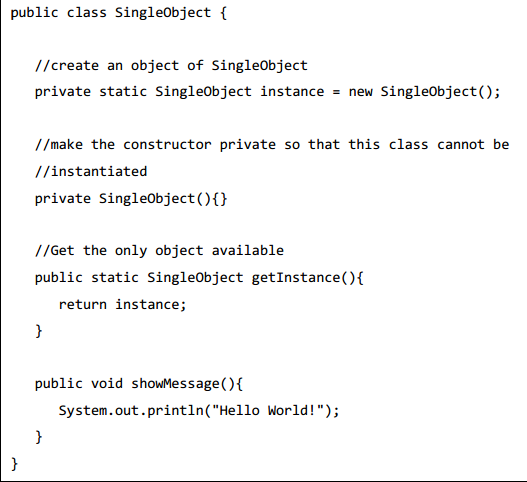
* Trong trường hợp chỉ cần một thể hiện duy nhất của một lớp.
* Khi thể hiện duy nhất khả mở thông qua việc kế thừa, người dùng có thể sử dụng thể hiện kế thừa đó mà không cần thay đổi các đoạn mã của chương trình.

**Biểu Đồ UML:**

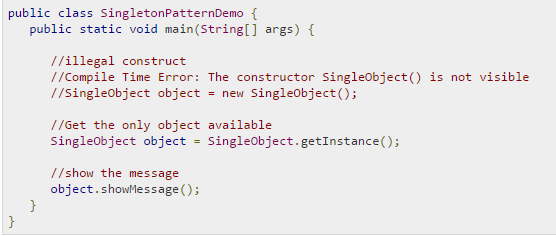
****

**Code java:**

**SingleObject.java**

****

**SingletonPatternDemo.java**

****

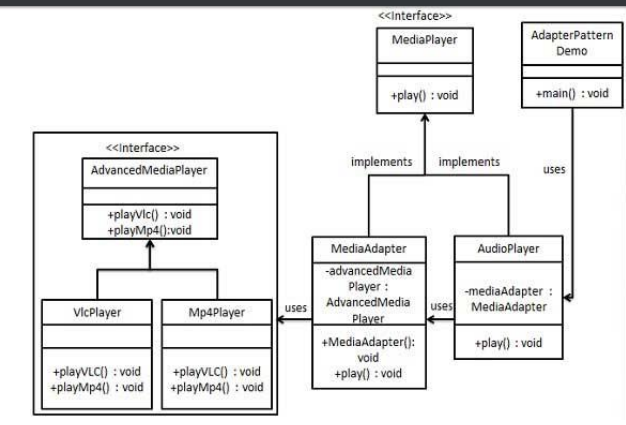
1. **Adapter**

**Định nghĩa:** Mẫu adapter chuyển đổi giao diện thành một giao diện khác mà phù hợp với yêu cầu. Giúp kết nối các lớp có giao diện không tương thích để làm việc với nhau

**Ý nghĩa sử dụng:**

* Khi ta muốn chuyển đổi một lớp với một giao diện thành giao diện mà ta mong muốn.
* Xậy dựng, mở rộng các phương thức của lớp có sẵn phù hợp với yêu cầu.
* Tái sử dụng giao diện cũ. Giảm thiểu việc viết lại mã lệnh.

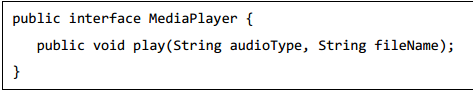
**Biểu đồ UML:**

****

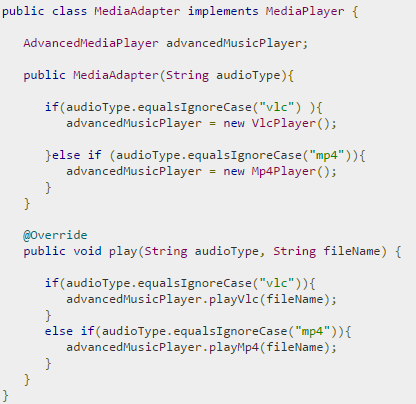
**Code Java:**

Tạo interfaces Media Player and Advanced Media Player:

**MediaPlayer.java**

****

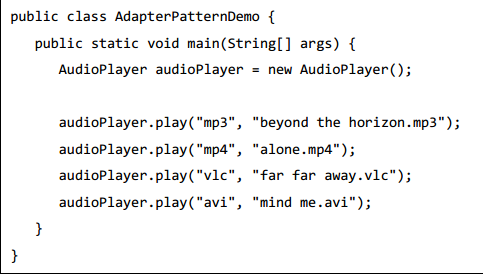
**MediaAdapter.java**

****

*AudioPlayer.java*

****

**AdapterPatternDemo.java**

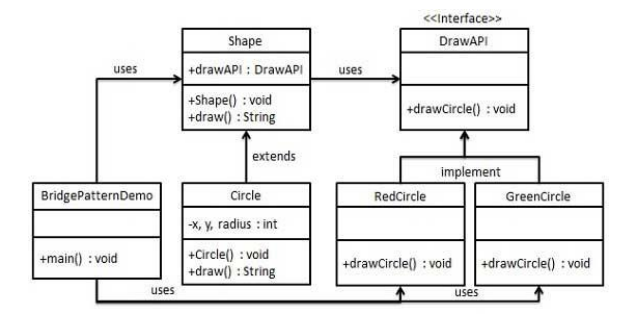
****

1. **Bridge**

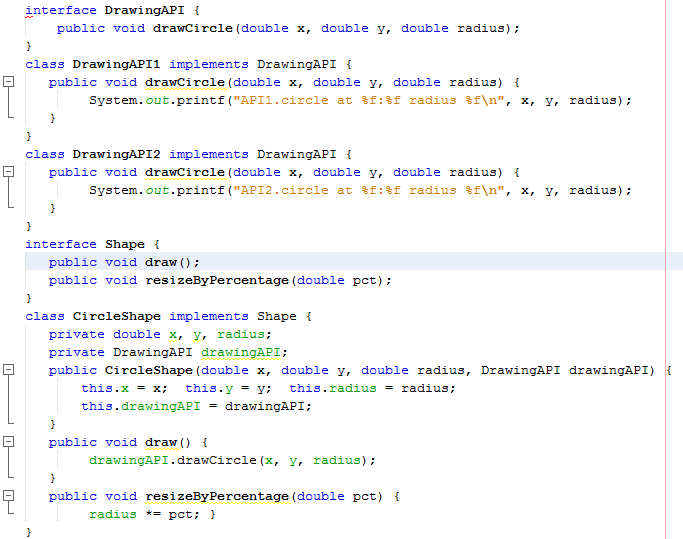
**Định nghĩa:** Bridge được sử dụng khi chúng ta cần phải tách riêng các thành phần trừu tượng từ việc thực hiện để cả hai có thể khác nhau một cách độc lập. Đây là loại design pattern đi kèm theo mô hình cấu trúc (structural pattern) như mô hình này tách riêng lớp thực hiện và lớp trừu tượng bằng cách cung cấp một cấu trúc cầu nối giữa chúng.  
Mô hình này bao gồm một giao diện hoạt động như một cầu nối mà làm cho các chức năng của các lớp cụ thể độc lập từ các lớp giao diện người thực hiện. Cả hai loại lớp có thể được thay đổi cấu trúc mà không ảnh hưởng lẫn nhau.

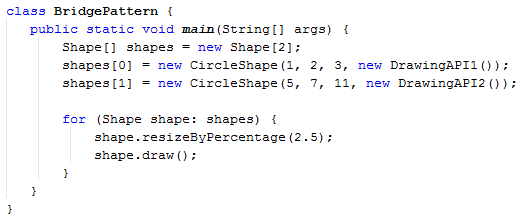
**Ý nghĩa sử dụng:**

**Biểu đồ UML:**

****

**Code java:**

****

****

1. **Façade**

**Định nghĩa:**

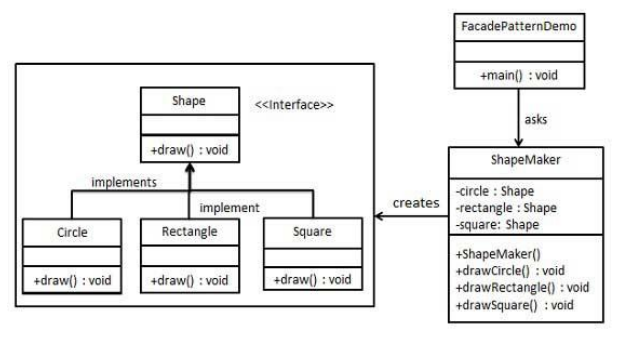
Facade là pattern cung cấp một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem). Façade Pattern định nghĩa một giao diện ở một cấp độ cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này vì chỉ cần giao tiếp với một giao diện chung duy nhất.

Façade Pattern cho phép các đối tượng truy cập vào hệ thống con bằng cách sử dụng giao diện chung này để giao tiếp với các giao diện có trong hệ thống con. Mục tiêu là che giấu các hoạt động trong hệ thống con.

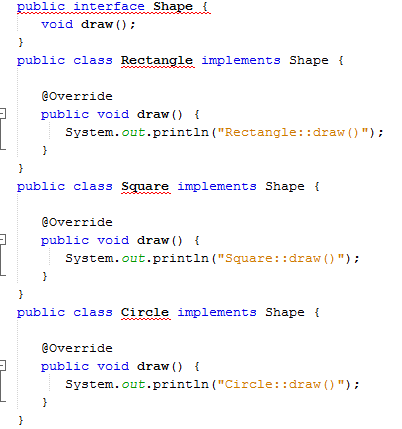
**Ý nghĩa sử dụng:**

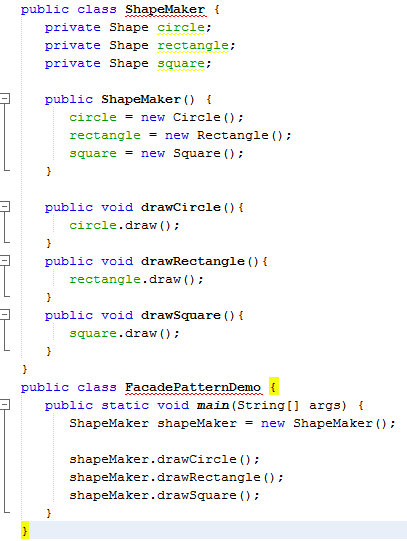
* Khi việc thiết kế các pattern sinh ra rất nhiều lớp làm người sử dụng rất khó để có thể hiểu được quy trình xử lý của chương trình. Và khi có rất nhiều hệ thống con mà mỗi hệ thống con đó lại có những giao diện riêng lẻ của nó nên rất khó cho việc sử dụng phối hợp.
* Khi người lập trình muốn đưa ra một giao diện đơn giản cho người sử dụng một hệ thống phức tạp.
* Khi người sử dụng phụ thuộc nhiều vào các lớp cài đặt. Việc áp dụng Façade Pattern sẽ tách biệt hệ thống con của người dùng và các hệ thống con khác, do đó tăng khả năng độc lập và khả chuyển của hệ thống con, dễ chuyển đổi nâng cấp trong tương lai.
* Khi người lập trình muốn phân lớp các hệ thống con. Dùng Façade Pattern để định nghĩa cổng giao tiếp chung cho mỗi hệ thống con, do đó giúp đơn giản hóa sự phụ thuộc của các hệ thống con vì các hệ thống này chỉ giao tiếp với nhau thông qua các cổng giao diện chung đó.

**Biểu đồ UML**

****

**Code Java:**

****

****

1. **Chain Of Responsibility**

**Định nghĩa:**

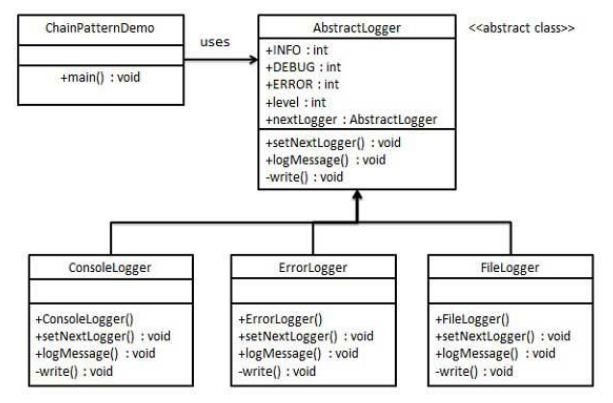
Chain of Responsibility kết nối người gửi một yêu cầu đến nơi nhận yêu cầu của nó bằng cách cho nhiều hơn một đối tượng một cơ hội để xử lý các yêu cầu. Chuỗi các đối tượng tiếp nhận và truyền các yêu cầu theo chuỗi cho đến khi một đối tượng tiếp nhận xử lý nó.  
Khởi tạo và chạy lại yêu cầu với một đường ống xử lý duy nhất có chứa nhiều xử lý khả thi.

Trong mô hình này, thông thường mỗi đối tượng tiếp nhận yêu cầu có chứa tham chiếu đến đối tượng tiếp nhận yêu cầu khác.Nếu một đối tượng không thể xử lý các yêu cầu, nó sẽ gửi yêu cầu đó thông qua message đến đối tượng tiếp theo

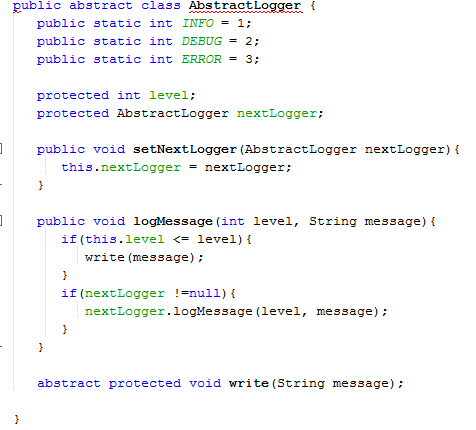
**Ý nghĩa sử dụng:**

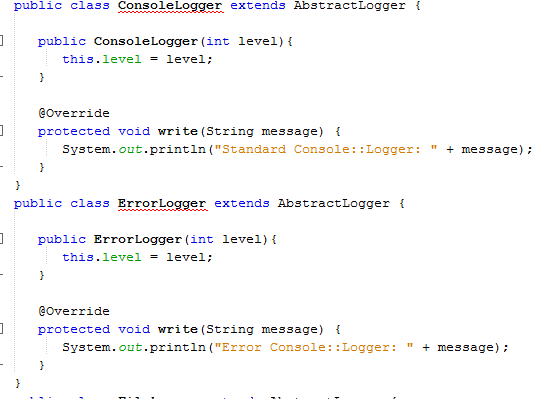
* Có nhiều hơn một đối tượng có khả thực xử lý một yêu cầu trong khi đối tượng cụ thể nào xử lý yêu cầu đó lại phụ thuộc vào ngữ cảnh sử dụng.
* Muốn gửi yêu cầu đến một trong số vài đối tượng nhưng không xác định đối tượng cụ thể nào sẽ xử lý yêu cầu đó.
* Tập các đối tượng có khả năng xử lý yêu cầu là một tập biến đổi.

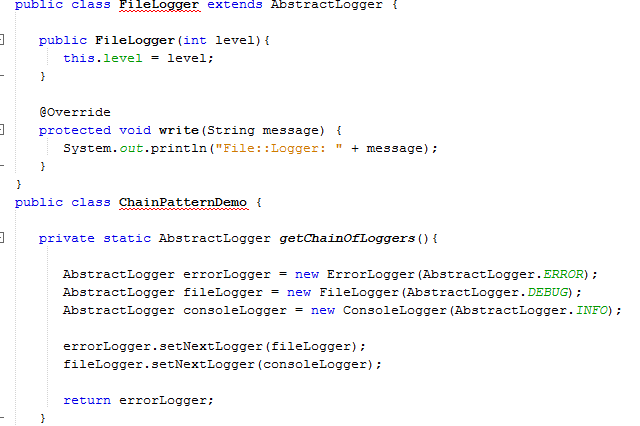
**Biểu đồ UML**

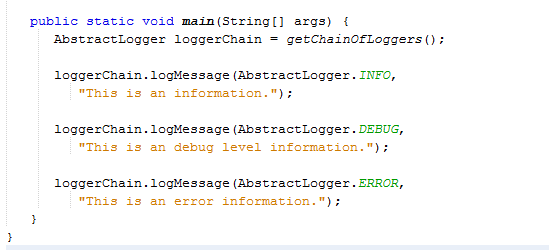
****

**Code Java:**

****

****

****

****

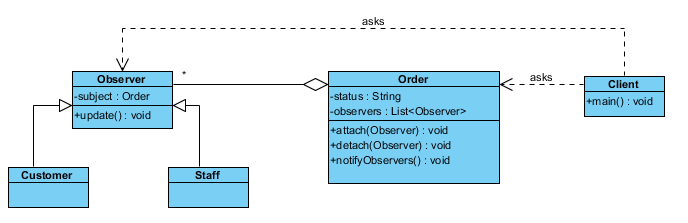
1. **Observer**

**Định nghĩa:** Observer cho phép các đối tượng có thể lắng nghe và phản ứng khi có thông báo từ một đối tượng khác. Tức là khi một đối tượng gửi một thông báo, các đối tượng lắng nghe nó có thể phản ứng lại với thông báo đó.

**Ý nghĩa sử dụng:**

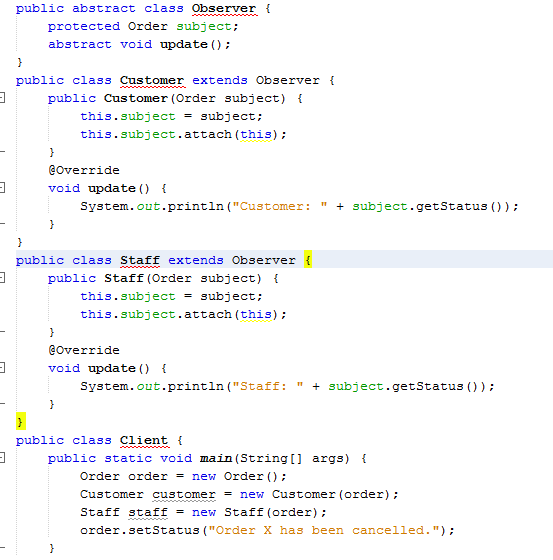
Khi bạn muốn các đối tượng liên lạc với nhau. Khi đối tượng này gửi 1 thông điệp thì các đối tượng đăng ký lắng nghe thông điệp sẽ phản ứng lại với thông điệp đó. Đối tượng gửi thông điệp sẽ không cần biết nó sẽ gửi cho ai và đối tượng nhận thông điệp sẽ không cần biết ai gửi thông điệp đó.

**Biểu đồ UML:**

****

**Code Java:**

****

****

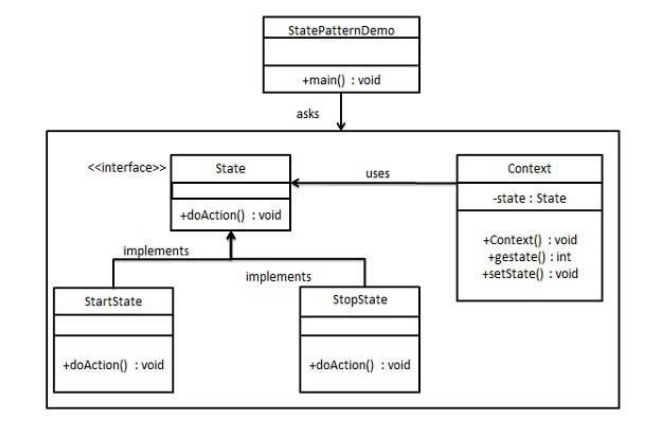
1. **State**

**Định nghĩa:**

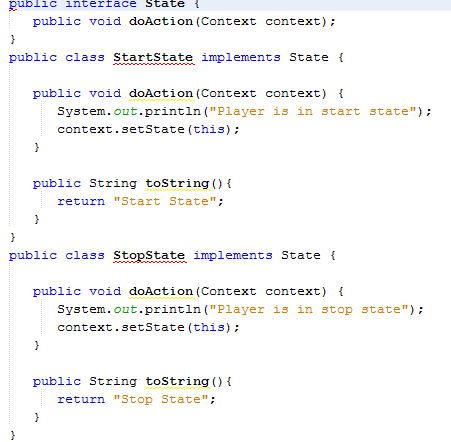
hành vi một class được thay đổi dựa trên trạng thái của nó. Đây là loại mẫu thiết kế thuộc mô hình hành vi

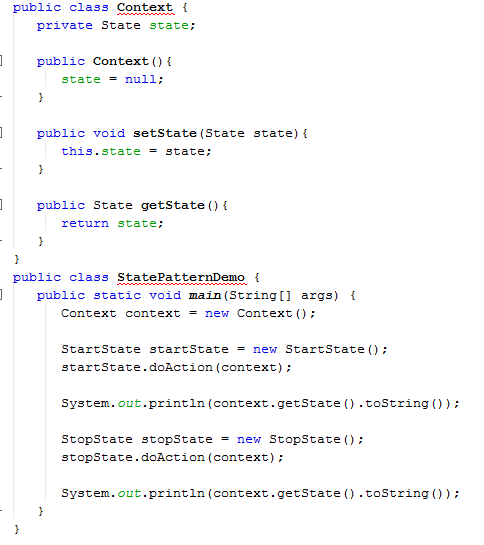
**Ý nghĩa sử dụng:**

**Biểu đồ UML:**

****

**Code Java:**

****

****

**Câu 5: 2 pattern (trình bày và giải thích)**