Chương 6

Hình mẫu thiết kế - Design Pattern

Giáo viên : ThS. Trần Văn Thọ
Đơn vị : Bộ môn KTHT & MMT

1. Giới thiệu

- **Khái niệm:**
  - + Hình mẫu thiết kế (Design Pattern) là các giải pháp thiết kế chung có thể áp dụng cho nhiều bài toán khác nhau.
  - + Các hình mẫu thiết kế được coi là các giải pháp tối ưu cho nhiều trường hợp trong thực tế, được sinh ra và kiểm nghiệm bởi rất nhiều hệ thống trong thời gian dài.

1. Giới thiệu

- **Phân loại:**
  + Nhóm khởi tạo (Creational Pattern)*: cung cấp các phương thức khởi tạo đối tượng nhằm che giấu việc khởi tạo thật sự (bằng toán tử new): **Abstract Factory, Factory Method, Singleton, Builder, Prototype.**
  + Nhóm cấu trúc (Structural Pattern)*: thiết lập định nghĩa, quan hệ giữa các lớp và đối tượng: **Adapter, Bridge, Composite, Decorator, Facade, Proxy.**
  + Nhóm hành vi(Behavioral Pattern)*: tập trung vào tương tác giữa các đối tượng: **Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor.**

1. Giới thiệu

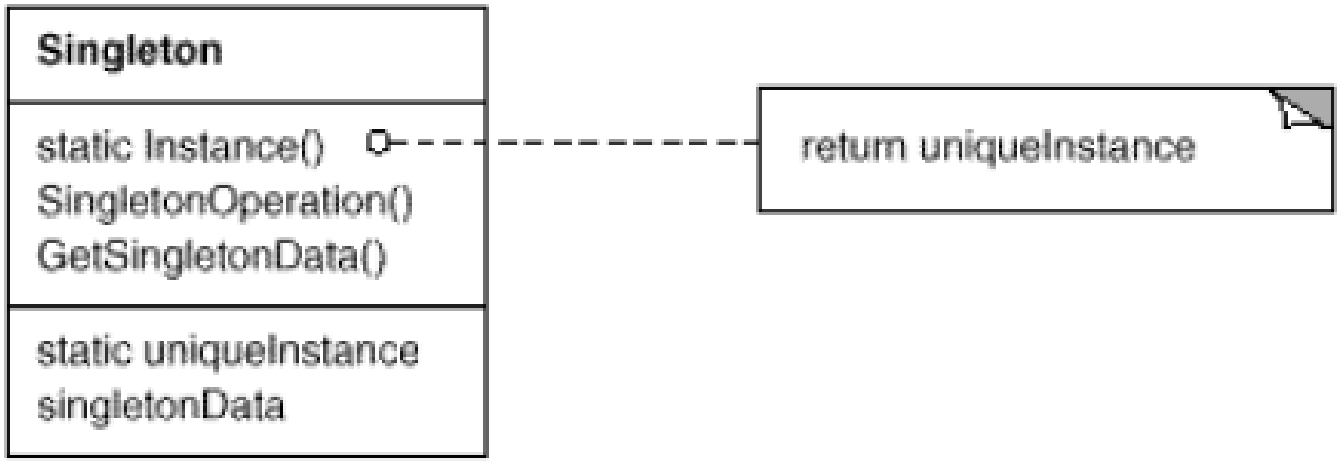
- **Phân loại:**
  + Nhóm kiến trúc (Architecture Pattern) : tập trung vào hình mẫu thiết kế ở mức kiến trúc hệ thống: **MVC, N-tier, Data Access Object (DAO)**,…
  + Nhóm Concurency Pattern: chú trọng đến việc xử lý đồng thời, các hệ thống đa luồng.

2. Singleton Pattern

- Hình mẫu đảm bảo một lớp chỉ có 1 đối tượng duy nhất thuộc lớp được sử dụng trong toàn chương trình.
- Được sử dụng trong các trường hợp hệ thống chỉ cho phép một thể hiện của lớp hoạt động trong 1 thời điểm như lớp phụ trách kết nối cơ sở dữ liệu, kết nối mạng,…
- Cho phép có 1 thể hiện của lớp mang dữ liệu có tính toàn cục trong toàn bộ hệ thống.

2. Singleton Pattern

- Sơ đồ cấu trúc:



Singleton: định nghĩa phương thức Instance() trả về một đối tượng duy nhất để các lớp khác truy cập

2. Singleton Pattern

- **Triển khai:**
  + Sử dụng thành phần dữ liệu tĩnh để lưu đối tượng được tạo.
  + Sử dụng một hàm thành phần tĩnh để khởi tạo đối tượng tĩnh khi nó chưa được khởi tạo, qua đó che dấu đối tượng tĩnh thực sự và lệnh khởi tạo.
  + Các đối tượng bên ngoài khi cần truy cập đến đối tượng duy nhất của lớp có thể truy cập thông qua hàm thành phần tĩnh của lớp Singleton.

2. Singleton Pattern

Ví dụ:

```
public class SingletonTest {
    static SingletonTest _instance;
    int data;
    static public SingletonTest getInstance(){
        if(_instance==null){
            _instance=new SingletonTest();
        }
        return _instance;
    }
    public void doSomething(){
        System.out.println("Singleton object does something:"+data);
    }
    public void setData(int d){
        data=d;
    }
}
```

2. Singleton Pattern

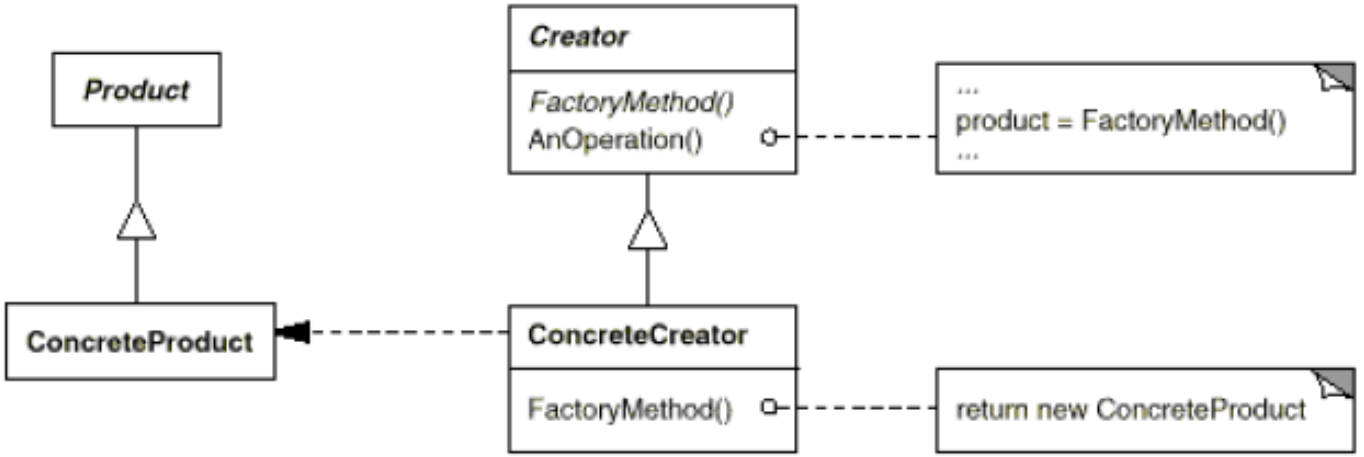
- **Ưu điểm của singleton**:
  + Kiểm soát được truy cập đến đối tượng lớp singleton.
  + Giảm không gian tên: mỗi singleton là một đối tượng toàn cục, nó có thể thay thế cho các biến toàn cục.
  + Singleton có thể dễ dàng mở rộng để có thể tạo ra nhiều đối tượng hơn nhưng vẫn có thể kiểm soát số lượng đối tượng.

3. Factory pattern

- **Mục đích**: định nghĩa giao diện để tạo ra đối tượng, nhưng để lớp con của nó quyết định đối tượng thuộc lớp nào sẽ được tạo ra.
- **Lý do**:
  + Giả sử có 2 lớp trừu tượng là Document và Application. Lớp Application có chứa các phương thức làm việc với lớp Document nhưng không thể khởi tạo đối tượng thuộc lớp này (vì document là lớp trừu tượng) -> tạo ở lớp Application một hàm ảo khởi tạo Document để sử dụng, sau để lớp con của nó định nghĩa lại hàm này và khởi tạo một đối tượng thuộc lớp thực thi của Document.

3. Factory pattern

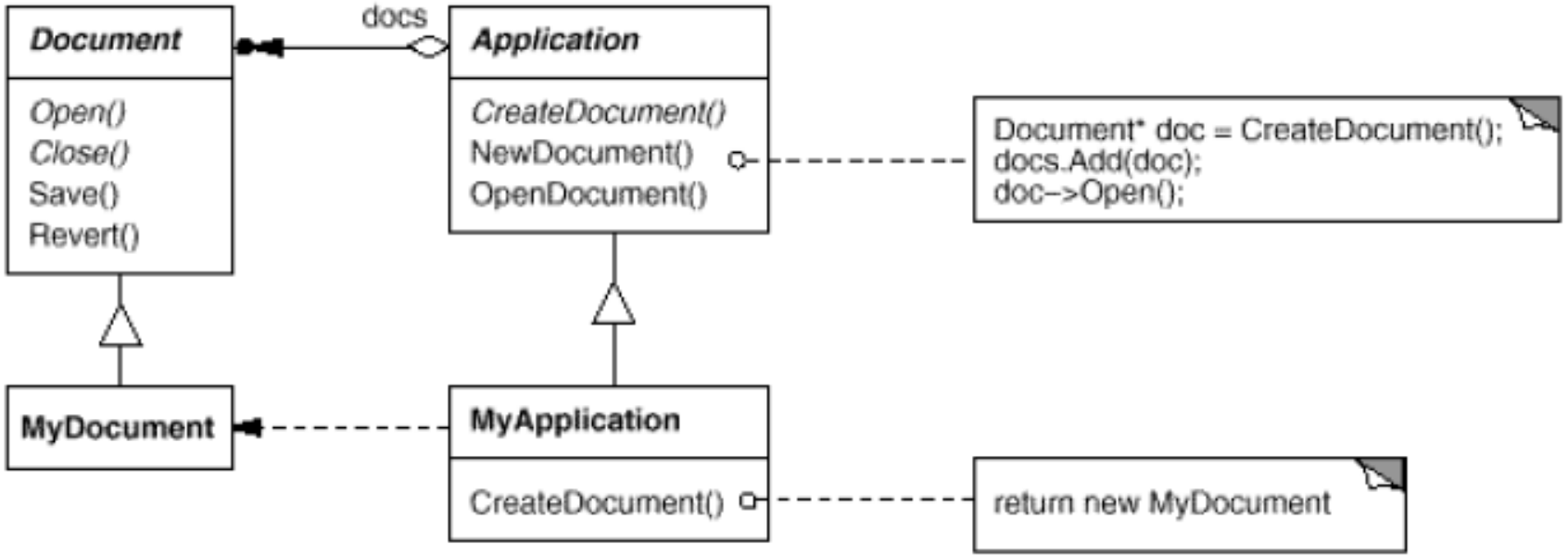
Sơ đồ:



Product: định nghĩa giao diện của các đối tượng cần khởi tạo.
ConcreteProduct: triển khai giao diện Product.
Creator: định nghĩa hàm factory, trả về đối tượng của lớp Product.
ConcreteCreator: định nghĩa lại factory, trả về một đối tượng thuộc lớp ConcreteProduct.

3. Factory pattern

- **Sơ đồ minh họa ví dụ**



3. Factory pattern

- **Triển khai:**
  - **2 dạng của lớp Creator:**
    - Creator là lớp thực thi: Hàm factory trả về luôn đối tượng thuộc một lớp ConcreteProduct dựa vào tham số của hàm.
    - Creator là lớp trừu tượng: Cần tạo thêm các lớp thực thi của creator gắn với các ConcreteProduct tương ứng.

3. Factory pattern

Ví dụ 1:

```
abstract class  Document{
    abstract void open();
    abstract void close();
    abstract void write();
}
abstract class Application{
    abstract Document getDocument();//factory
method
    void doSomething(){
        Document doc= getDocument();
        doc.open();
        doc.write();
        doc.close();
    }
}
```

3. Factory pattern

Ví dụ 1

```
class Notebook extends Document{

    @Override
    void open() {
        System.out.println("open notebook");
    }

    @Override
    void close() {
         System.out.println("close notebook");
    }

    @Override
    void write() {
         System.out.println("write notebook");
    }
}
```

3. Factory pattern

Ví dụ 1

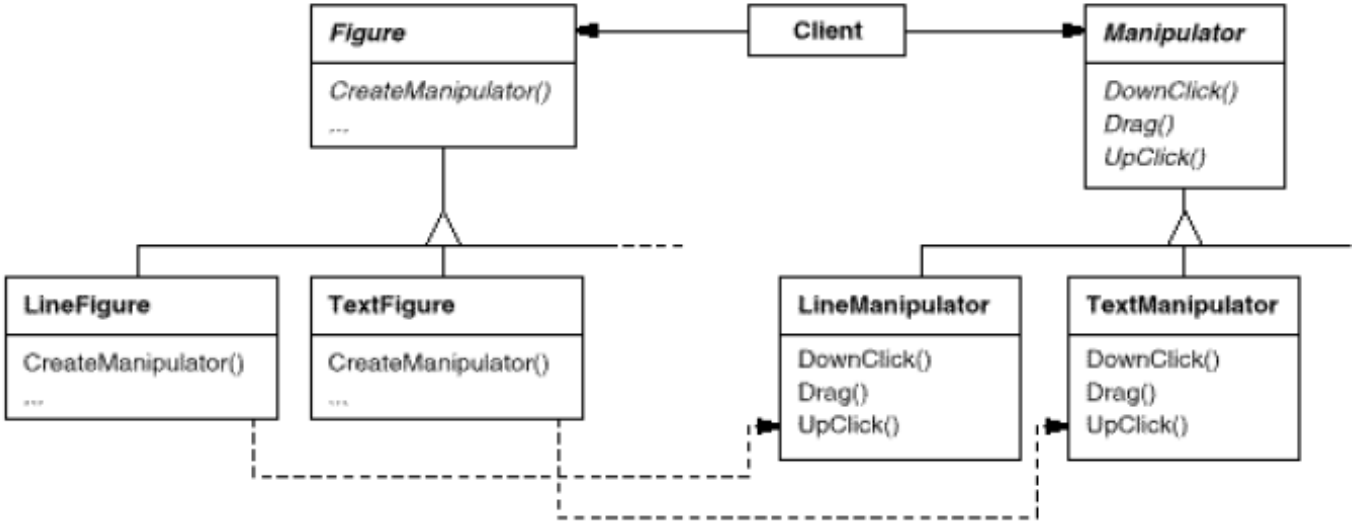
```
class Student extends Application{

    @Override
    Document getDocument() {
        return new Notebook();
    }

}
```

3. Factory pattern

- **Ưu điểm:**
  - + Cung cấp cơ chế kết nối cho các lớp con: nhờ định nghĩa các hàm factory, các lớp con có cơ sở để chồng các hàm này với các lớp thực thi tương ứng.
  - + Kết nối song song giữa các nhánh thừa kế: các hàm ở lớp thuộc nhánh thừa kế này có thể gọi đến các hàm chồng factory ở nhánh kia.

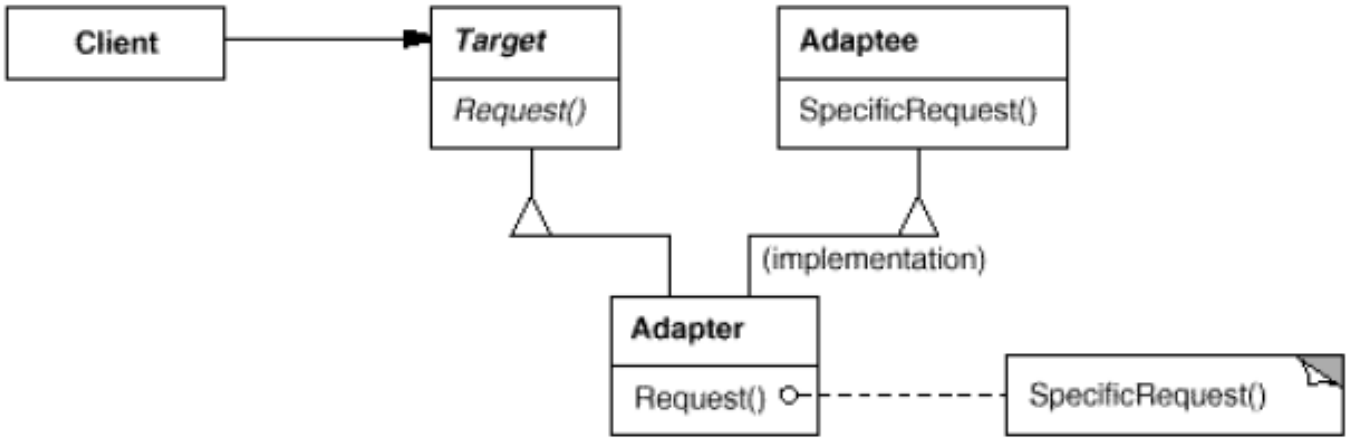


4. Adapter pattern

- **Mục đích**: Chuyển đổi giao diện của một lớp sang một giao diện khác mà client mong muốn. Adapter khiến cho các lớp hợp tác được cùng nhau (điều mà nếu không có nó thì không thể thực hiện được).
- **Lý do**:
  + Đôi khi, một lớp được thiết kế để có thể tái sử dụng không thể áp dụng trong một số trường hợp đặc thù nào đó do không tương thích.
  + VD: lớp Shape định nghĩa giao diện bao gồm các phương thức phục vụ cho việc vẽ hình, thừa kế nó là các lớp vẽ ra các hình cụ thể như Line, Rectange…nhưng để vẽ ra Text lại khá khó khan.
  + -> Sử dụng lại đặc điểm của lớp TextView, nhưng lớp TextView lại không liên quan gì đến Shape.
  + -> Cần tạo một lớp TextShape, trong đó lợi dụng các tính năng của lớp TextView để thực hiện các tính năng của lớp Shape.

4. Adapter pattern

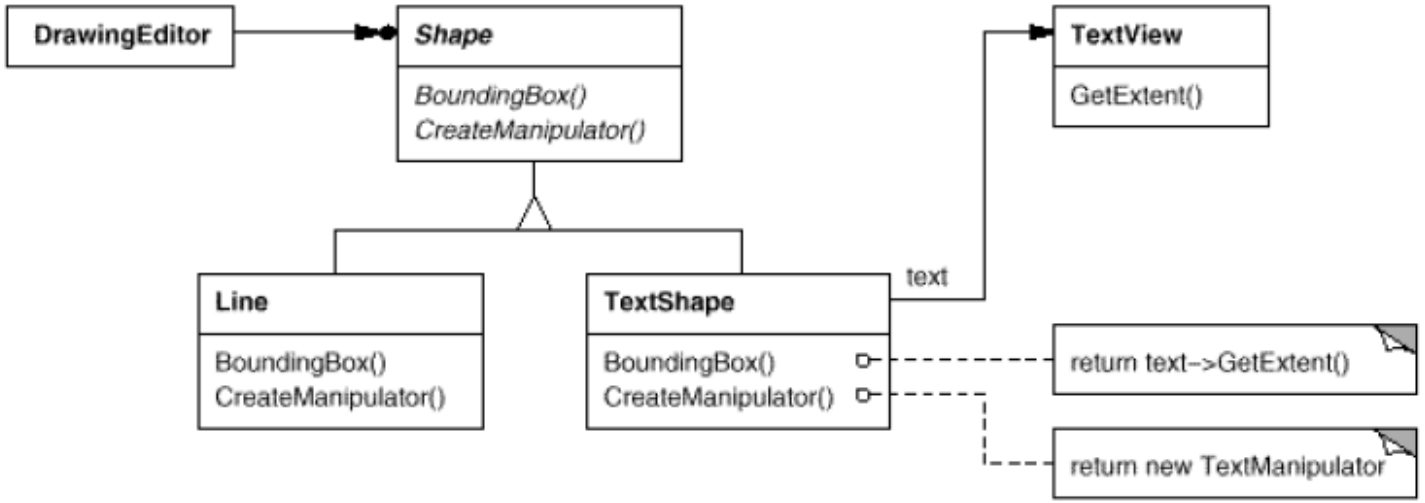
Sơ đồ tổng quát:



Target: định nghĩa giao diện mà **Client** sử dụng.
Client: lớp sử dụng các đối tượng con của **Target,** cụ thể là **Adapter**
Adaptee: chứa các phương thức mà lớp **Adapter** cần sử dụng.
Adapter: chuyển đổi giao diện **Adaptee** sang giao diện **Adapter.t**

4. Adapter pattern

Sơ đồ ví dụ:



Lớp TextShape thừa kế lại lớp Shape nhưng lại sử dụng các phương thức của đối tượng lớp TextView để thực hiện các chức năng của mình -> trở thành một adapter giữa lớp Shape và TextView.

4. Adapter pattern

- **Triển khai**:
  - + Xác định các lớp đóng vai lớp Target và lớp Adaptee
    - Target: lớp (giao diện) mà trong hoàn cảnh sử dụng ta buộc phải dung.
    - Adaptee: lớp (giao diện) chứa các phương thức mà ta cần để bổ sung cho các chức năng của lớp Target.
  - + Xây dựng lớp Adapter : mục tiêu là tạo ra lớp Adapter thừa kế đầy đủ từ lớp Target trong khi lại có một số ưu điểm từ lớp Adaptee, có 2 cách:
    - Thừa kế lại lớp Target, trong đó phương thức nào cần sử dụng tới chức năng của lớp Adaptee thì tạo một đối tượng của lớp Adaptee để khai thác.
    - Đa thừa kế cả lớp Target và Adaptee, sau đó sử dụng chính các phương thức thừa kế từ Adaptee trong các phương thức thừa kế từ Target.

4. Adapter pattern

Ví dụ:

```
class Shape{
    public Shape(){}
    public void BoundingBox(){

System.out.println("shape
method");
    }
}
class TextView{
    public TextView(){}
    public void getExtent(){

System.out.println("textview
method");
    }
}
```

```
class TextShape extends Shape{
    TextView text;
    public TextShape(TextView
tv){

        this.text=tv;
    }
    public void BoundingBox(){
        this.text.getExtent();
    }
}
public class AdapterPattern {
    public static void
main(String[] args) {
        Shape shape=new
TextShape(new TextView());
        shape.BoundingBox();
    }
}
```

4. Adapter pattern

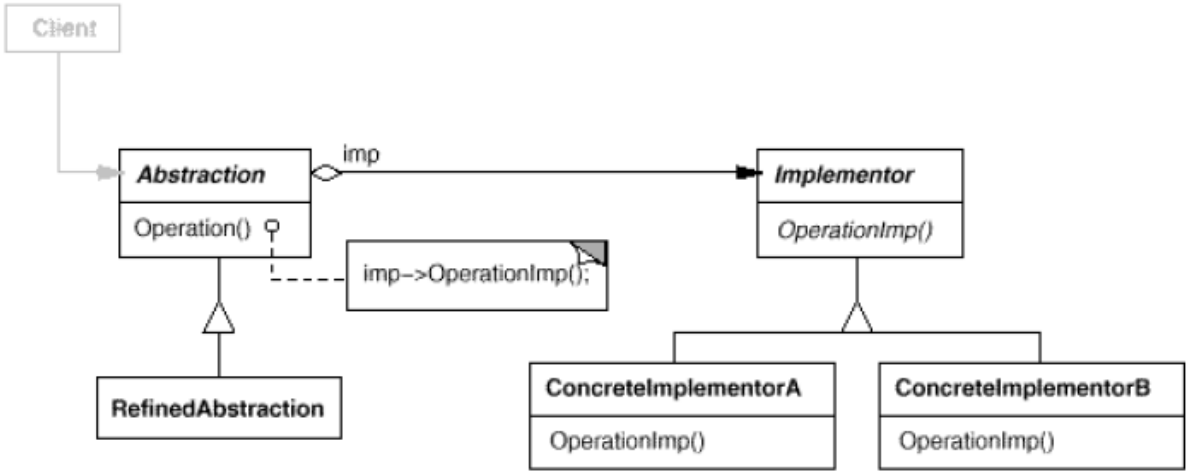
- **Ứng dụng:**
  - + Khi muốn sử dụng một lớp có sẵn nhưng các phương thức của lớp đó (giao diện) lại không đáp ứng yêu cầu.
  - + Khi muốn tạo ra một lớp có thể tái sử dụng là sự kết hợp của nhiều lớp khác nhau và không tương thích nhau (không cùng phân cấp thừa kế).

5. Bridge pattern

- **Mục đích**: tách phần trừu tượng và thực thi của một lớp thành 2 lớp riêng biệt để 2 lớp này có thể sử dụng một cách độc lập.
- **Lý do**:
  + Một khái niệm trừu tượng có thể có nhiều kiểu thực thi khác nhau. Ví dụ như 1 lớp trừu tượng có thể có một số lớp thừa kế thực thi theo nhiều cách. Tuy nhiên cách thừa kế đơn giản này khiến các lớp thực thi bị phụ thuộc hoàn toàn vào lớp trừu tượng, khó thay đổi và mở rộng -> nếu tách phần trừu tượng và thực thi thành 2 nhánh thừa kế khác nhau thì sẽ thuận tiện cho việc sửa đổi và mở rộng hơn nhiều -> liên kết giữa phần trừu tượng và thực thi gọi là Cầu -> Hình mẫu thiết kế Cầu.

5. Bridge pattern

Sơ đồ tổng quát:



Abtraction:định nghĩa giao diện trừu tượng, chứa liên kết đến Implementor.

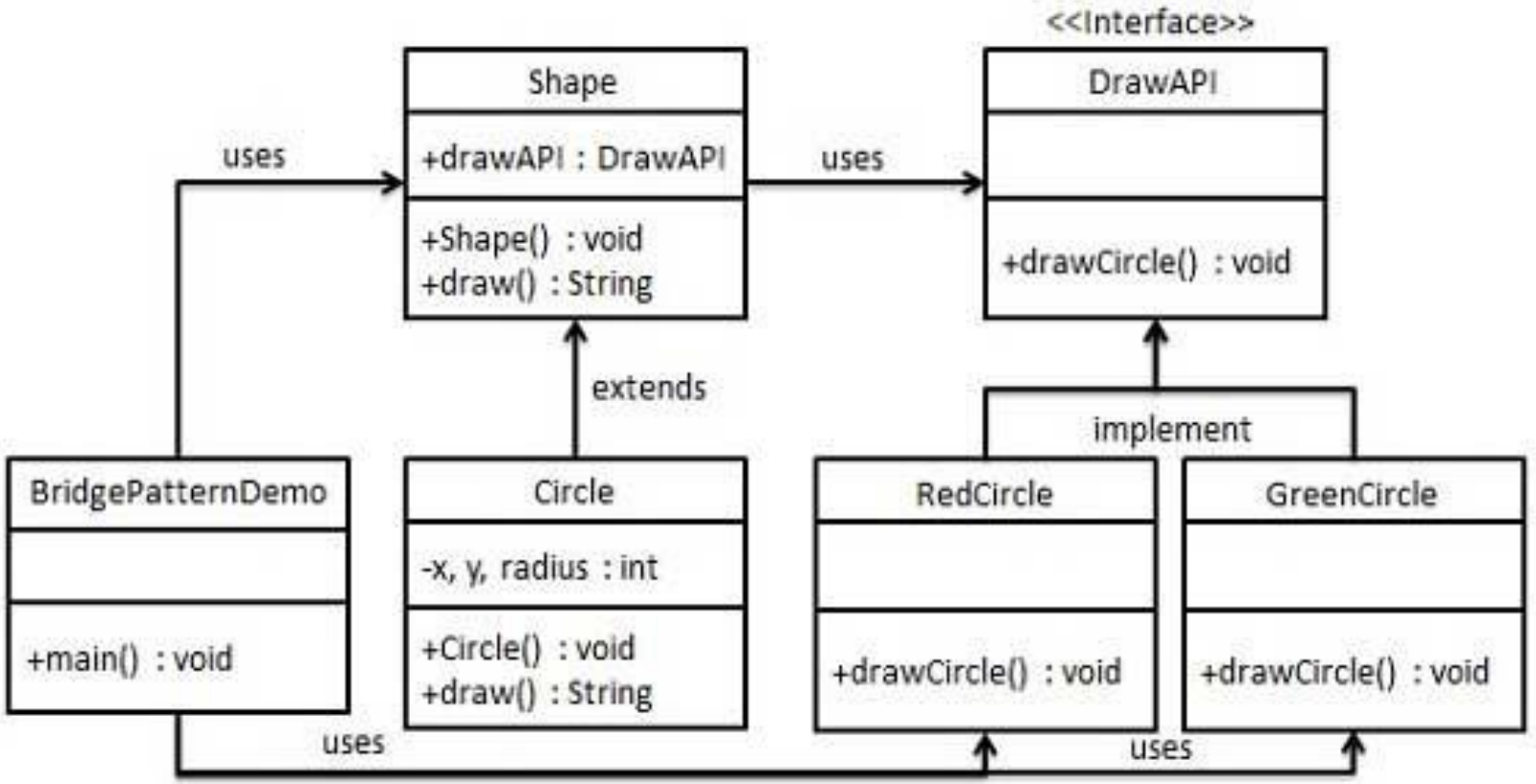
RefinedAbstraction: lớp mở rộng từ Abtraction.

Implementor: định nghĩa các giao diện thực thi, thường cung cấp các thao tác cụ thể, còn Abstractor định nghĩa các thao tác tổng quan hơn sử dụng các thao tác cụ thể này.

ConcreteImplementor: thừa kế lại Implementor và định nghĩa các thực thi cụ thể của implementor.

5. Bridge pattern

Sơ đồ ví dụ:



5. Bridge pattern

- **Triển khai:**
  - Tạo lớp Implementor chứa các phương thức thực thi cụ thể.
  - Tạo lớp Abstraction chứa đối tượng Implementor để thực thi các phương thức trừu tượng.
  - Tạo các lớp ConcreteImplementor định nghĩa cụ thể các phương thức thừa kế từ Implementor.
  - Tạo các lớp RefinedAbstraction định nghĩa cụ thể các phương thức thừa kế từ Abstraction.
-

5. Bridge pattern

Ví dụ

```
abstract class DrawAPI{
    public abstract void drawCircle();
}
 abstract class Shape{
    DrawAPI api;
    public Shape(){}
    public abstract void draw();
}
```

5. Bridge pattern

Ví dụ:

```
class Circle extends Shape{
    public Circle (DrawAPI api){
        this.api=api;
    }
    @Override
    public void draw() {
        api.drawCircle();
    }
}
class RedCircleAPI extends DrawAPI{

    @Override
    public void drawCircle() {
        System.out.println("draw a red cirlce");
    }
}
class BlueCircleAPI extends DrawAPI{

    @Override
    public void drawCircle() {
        System.out.println("draw a blue cirlce");
    }
}
```

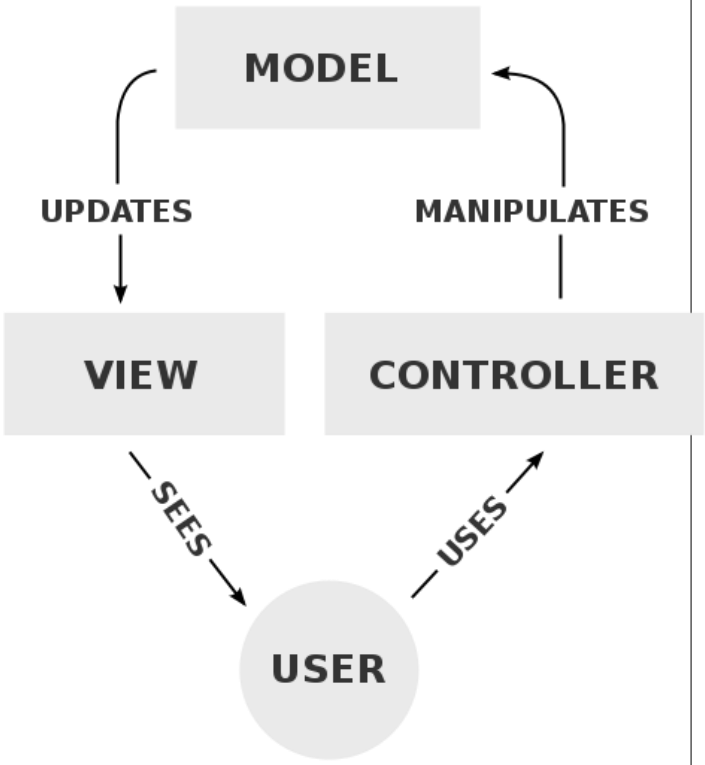
5. Bridge pattern

Ví dụ:

```
public class BridgePattern {
    public static void main(String[] args) {
        Shape redcircle=new Circle(new RedCircleAPI());
        redcircle.draw();
        Shape bluecircle=new Circle(new BlueCircleAPI());
        bluecircle.draw();
    }
}
```

6. MVC pattern

- HÌnh mẫu MVC (Model-View-Controller) nhằm tách biệt ứng dụng thành 3 thành phần kết nối với nhau:
  - **Model**: mô hình thể hiện một đối tượng mang dữ liệu, logic và các luật của ứng dụng.
  - **View**: thể hiện dữ liệu được lưu trữ ở Model.
  - **Controller**: tương tác với cả Model và View, làm nhiệm vụ điều khiển dữ liệu vào các Model và cập nhật View khi dữ liệu thay đổi.

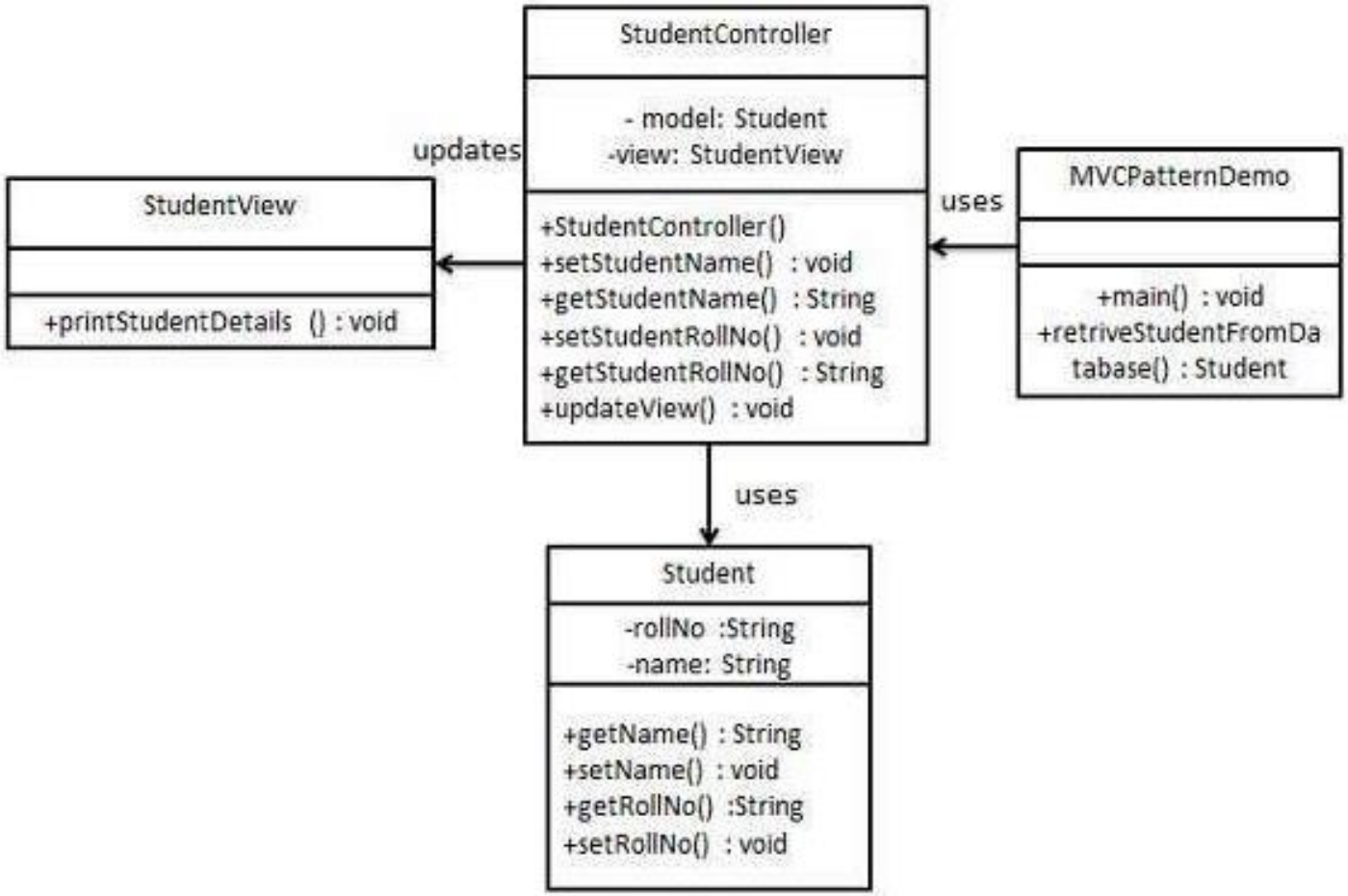


6. MVC pattern

- **Triển khai**:
  + Phân tích hệ thống để xác định được đặc điểm dữ liệu và xử lý của bài toán.
  + Tạo các lớp xử lý luồng dữ liệu
    - Model:Tạo các lớp mang dữ liệu, các lớp này thường chỉ chứa thuộc tính dữ liệu, các phương thức truy cập dữ liệu.
    - View: tạo các lớp có các phương thức thể hiện dữ liệu, ví dụ như in dữ liệu ra màn hình, vẽ biểu đồ, máy in…
    - Controller: các lớp có các phương thức thao tác dữ liệu đầu vào trên Model và cập nhật lên các View.

6. MVC pattern

Ví dụ:

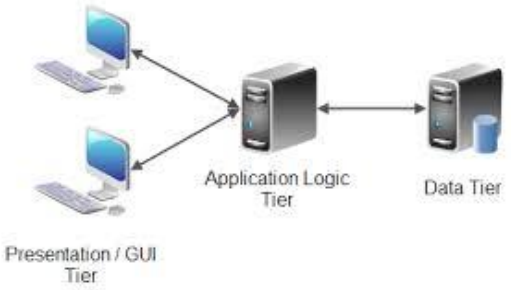


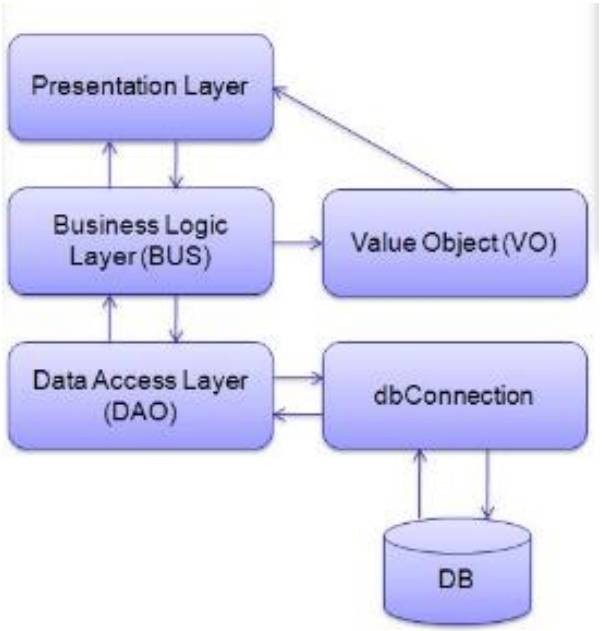
6. MVC pattern

- **Ứng dụng:**
  - + Mô hình MVC được ứng dụng rộng rãi trên nhiều loại hình ứng dụng, đặc biệt là trên các ứng dụng Web.
  - + Hiện nay có nhiều biến thể từ MVC : HMVC,MVVM
  - + Hầu hết các framework hiện nay đều hỗ trợ mô hình MVC: Spring (Java), ASP.NET MVC, …
- **Ưu điểm:**
  - + Các thành phần hiển thị, chứa dữ liệu và logic, điều khiển luồng dữ liệu là tách biệt, việc sửa đổi thành phần này ít ảnh hưởng đến thành phần khác.
  - + Dễ bảo trì và mở rộng ứng dụng.
  - + Cho phép phát triển song song nhiều thành phần ứng dụng.
  - + Cho phép tách biệt 2 công việc có đặc thù khác nhau: trình bày dữ liệu và xử lý truy vấn dữ liệu.

7. N-Tiers pattern

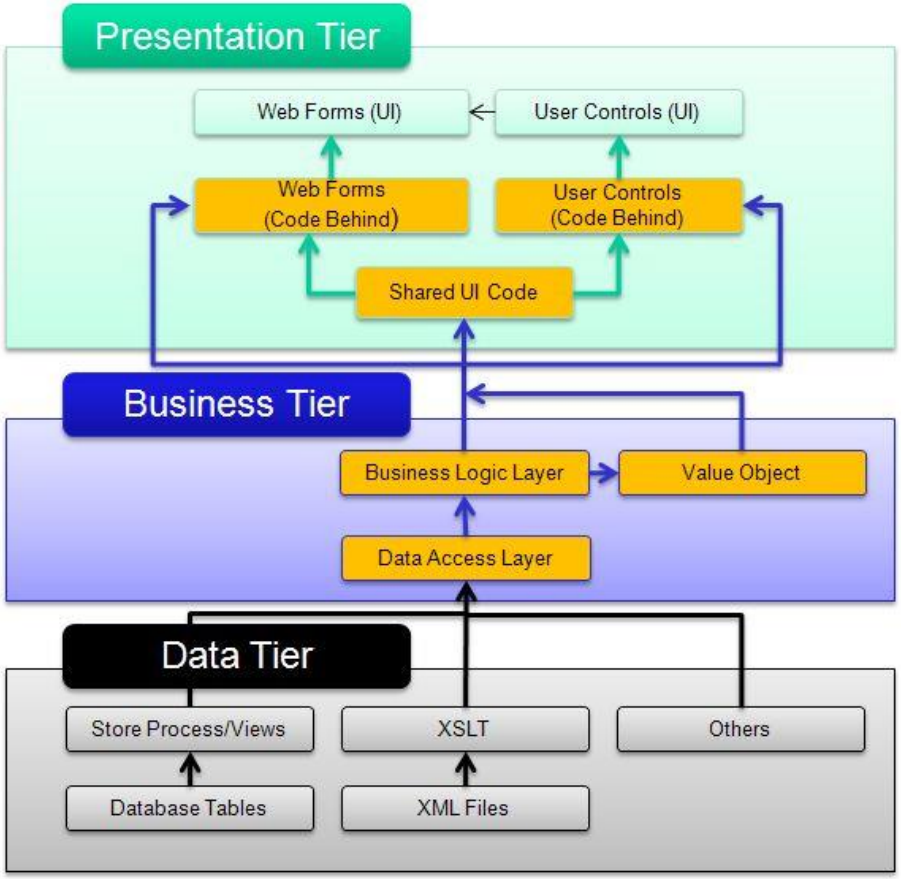
- Kiến trúc đa tầng (N-tiers) :
  + Hệ thống được chia thành các tầng vật lý nhằm tách rời các công đoạn thao tác CSDL, xử lý dữ liệu và phản hồi thao tác người dùng.
  + Được sử dụng phổ biến nhất là mô hình 3 tầng.
- Mô hình 3 lớp (3-layer):
  + Thường được dùng để mô tả kiến trúc đa tầng trong phần mềm.
  + Bao gồm 3 lớp (layer):
    - Truy cập dữ liệu.
    - Xử lý nghiệp vụ.
    - Giao diện.





7. Mô hình 3 lớp

- **Tầng truy cập dữ liệu** (Data Access Layer):
  - + Bao gồm CSDL và các thành phần thao tác trực tiếp với CSDL.
  - + Cung cấp các hàm truy xuất dữ liệu cho các tầng trên.
- **Tầng nghiệp vụ** (Business Layer):
  - + Chứa các thành phần xử lý nghiệp vụ.
  - + Sử dụng các hàm truy vấn dữ liệu ở tầng dưới và cung cấp các hàm nghiệp vụ cho tầng trên.
- **Tầng Giao diện** (Presentation Layer):
  - + Chứa giao diện người dùng.
  - + Nhận thao tác của người dùng, gọi đến các hàm nghiệp vụ để xử lý yêu cầu và trả lời người dùng.



7. Mô hình 3 lớp

- Triển khai (xây dựng từ tầng thấp nhất lên tầng trên):
  + Tầng truy xuất dữ liệu:
    - CSDL: tạo lập CSDL và các storedprocedure.
    - Tạo các thành phần kết nối CSDL: các lớp kết nối, lớp chứa các hàm truy xuất csdl cho mỗi bảng.
  + Tầng nghiệp vụ:
    - Tạo các lớp mang dữ liệu (Object Info).
    - Tạo các lớp chứa các hàm xử lý nghiệp vụ, sử dụng các hàm truy xuất dữ liệu và thao tác với các lớp mang dữ liệu.
  + Tầng giao diện:
    - Xây dựng giao diện người dùng.
    - Xử lý thao tác người dùng bằng cách gọi các hàm xử lý nghiệp vụ ở tầng dưới.

7. Mô hình 3 lớp

- Ứng dụng:
  + Mô hình đa tầng được sử dụng phổ biến trong nhiều kiểu hệ thống khác nhau: Web, Desktop application,...
- Ưu điểm của mô hình 3 lớp:
  + Dễ hiểu, dễ triển khai.
  + Có tính linh hoạt cao: hoạt động của các tầng là tương đối độc lập, có thể thay đổi tầng nào đó mà không cần phải chỉnh sửa ở các tầng khác.
  + Dễ phân chia công việc.