

The image is a word cloud centered around the words "Design" and "Patterns". The words are in various sizes and colors (black, white, grey) and are arranged around the main title. The words include: Abstract, Behavioral, Singleton, Template, Interpreter, Responsibility, Chain, Composite, Proxy, Structural, Decorator, Bridge, Diagram, Class, Command, Visitor, Object, agile, UNIX, Iterator, Builder, Flyweight, development, Adapter, Interaction, Method, Memento, Creation, Windows, Facade, Strategy, State, Prototype, Mediator, Observer.

NỘI DUNG

- ❑ Design Patterns là gì
 - ❑ Quy tắc của design patterns
 - ❑ Phân loại Design Patterns

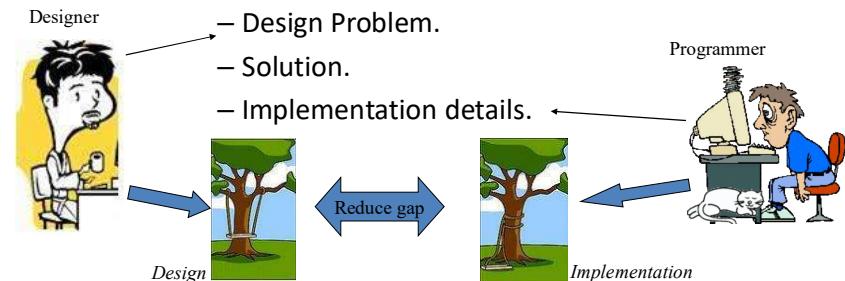
DESIGN PATTERNS

- ❑ Design Patterns (mẫu thiết kế)
 - ❖ là 1 giải pháp, 1 đoạn mô tả, hoặc 1 khuôn mẫu để giải quyết 1 vấn đề chung (thường gặp trong lập trình) một cách tối ưu nhất.
 - ❖ là một kỹ thuật trong lập trình hướng đối tượng
 - không phải là ngôn ngữ lập trình
 - được sử dụng thường xuyên trong các ngôn ngữ OOP.

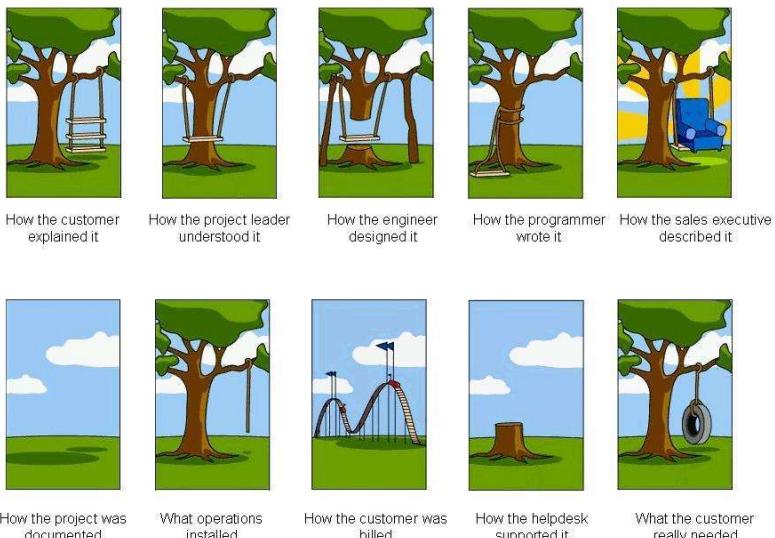
"Mỗi pattern mô tả một vấn đề xảy ra lặp đi lặp lại, và trình bày trọng tâm của giải pháp cho vấn đề đó, theo cách mà bạn có thể dùng đi dùng lại hàng triệu lần mà không cần phải suy nghĩ."

Christopher Wolfgang Alexander

DESIGN PATTERNS



DESIGN PATTERNS



❑ Gang of Four - GoF (bộ tứ)

- ❖ Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides
- ❖ Cuốn sách
"Design Patterns – Elements of Reusable Object-Oriented Software" (1994)
là khởi nguồn của khái niệm design pattern trong lập trình phần mềm.

TẠI SAO PHẢI SỬ DỤNG DESIGN PATTERNS?

- ❑ Design pattern là những giải pháp đã được tối ưu hóa, đã được kiểm chứng để giải quyết các vấn đề trong software engineering.
 - ❖ giúp giải quyết nhanh chóng vấn đề thay vì tự tìm kiếm giải pháp
 - ❖ tránh được các vấn đề tiềm ẩn có thể gây ra những lỗi lớn
 - ❖ giúp chương trình trở nên đơn giản hơn
 - ❖ giúp tái sử dụng mã lệnh
 - ❖ dễ dàng nâng cấp, mở rộng và bảo trì về sau
 - ❖ giúp lập trình viên có thể hiểu code của người khác nhanh chóng
- ❑ Design pattern cung cấp giải pháp ở dạng tổng quát
 - ❖ giúp tăng tốc độ phát triển phần mềm bằng cách đưa ra các mô hình test, mô hình phát triển đã qua kiểm nghiệm.

HỌC & NGHIÊN CỨU DESIGN PATTERNS

- ❑ Tập trung chú ý vào 3 phần:
 - ❖ Nó được sử dụng khi nào, vấn đề mà design pattern đó giải quyết là gì?
 - ❖ Sơ đồ UML mô tả design pattern.
 - ❖ Code minh họa, ứng dụng thực tiễn của nó là gì?
- ❑ Nắm thật vững:
 - ❖ 4 đặc tính của OOP: kế thừa (*inheritance*), đa hình (*polymorphism*), trừu tượng (*abstraction*), bao đóng (*encapsulation*).
 - ❖ 2 khái niệm: *interface* và *abstract*.

QUY TẮC CỦA DESIGN PATTERNS

- ❑ Theo quan điểm của GoF, design pattern chủ yếu được dựa theo những quy tắc sau đây về thiết kế hướng đối tượng.
 - ❖ Lập trình theo interface chứ không phải lập trình để implement interface. (program to interfaces, not implementations)
 - ❖ Ưu tiên object composition hơn là thừa kế. (prefer composition over inheritance)

QUY TẮC CỦA DESIGN PATTERNS

- ❑ Program to interfaces, not implementations

- ❖ Interface
 - chỉ định những hành vi mà một đối tượng có thể làm
 - chỉ định cách thức để gọi thực hiện các hành vi đó của đối tượng
- ❖ Implementation
 - Là cài đặt cụ thể của một hành vi
- ❖ Ví dụ:
 - Interface: void sort();
 - Implementation: thuật toán QuickSort, thuật toán HeapSort,...

QUY TẮC CỦA DESIGN PATTERNS

- ❑ Program to interfaces, not implementations
 - ❖ Lập trình theo cài đặt
 - Vd: khi cần sử dụng chức năng sắp xếp trong "chương trình" thì cài đặt một thuật toán sắp xếp cụ thể trong "chương trình" đó (hoặc trong một lớp khác, và sau đó "chương trình" sẽ gọi phương thức sắp xếp từ lớp đó)
 - Khi cần đổi thuật toán → **phải** viết lại "chương trình"
 - ❖ Lập trình theo interface
 - Vd: khi cần sử dụng chức năng sắp xếp trong "chương trình" thì sử dụng interface (có phương thức) sort, và cài đặt một thuật toán sắp xếp cụ thể trong một lớp implement interface đó, sau đó "chương trình" sẽ thực hiện sắp xếp thông qua phương thức sort của interface.
 - Khi cần đổi thuật toán → tạo lớp khác → **không cần** viết lại "chương trình"

QUY TẮC CỦA DESIGN PATTERNS

- ❑ Program to interfaces, not implementations



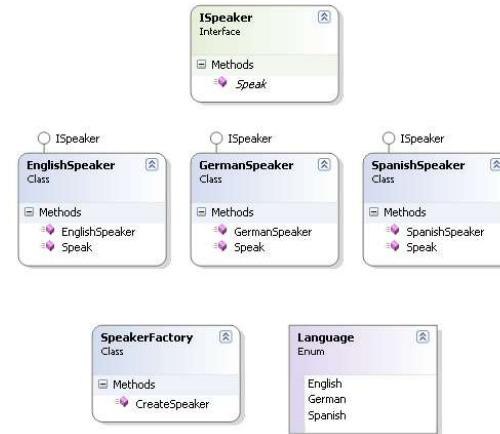
QUY TẮC CỦA DESIGN PATTERNS

- Program to interfaces, not implementations



QUY TẮC CỦA DESIGN PATTERNS

- Program to interfaces, not implementations



QUY TẮC CỦA DESIGN PATTERNS

- Program to interfaces, not implementations

```

public interface ISpeaker {
    void speak();
}

public class EnglishSpeaker implements ISpeaker {
    public EnglishSpeaker() { }

    public void speak() {
        System.out.println("I speak English.");
    }
}
  
```

```

classDiagram
    class ISpeaker {
        <<Interface>>
        <<Methods>>
        <<Speak>>
    }
    class EnglishSpeaker {
        <<ISpeaker>>
        <<Methods>>
        <<EnglishSpeaker>>
        <<Speak>>
    }
    class GermanSpeaker {
        <<ISpeaker>>
        <<Methods>>
        <<GermanSpeaker>>
        <<Speak>>
    }
    class SpanishSpeaker {
        <<ISpeaker>>
        <<Methods>>
        <<SpanishSpeaker>>
        <<Speak>>
    }
    class SpeakerFactory {
        <<Class>>
        <<Methods>>
        <<CreateSpeaker>>
    }
    class Language {
        <<Enum>>
        <<English>>
        <<German>>
        <<Spanish>>
    }
    EnglishSpeaker --> ISpeaker
    GermanSpeaker --> ISpeaker
    SpanishSpeaker --> ISpeaker
    SpeakerFactory --> EnglishSpeaker
    SpeakerFactory --> GermanSpeaker
    SpeakerFactory --> SpanishSpeaker
  
```

QUY TẮC CỦA DESIGN PATTERNS

- Program to interfaces, not implementations

```

public class GermanSpeaker implements ISpeaker {
    public GermanSpeaker() { }

    public void speak() {
        System.out.println("I speak German.");
    }
}

public class SpanishSpeaker implements ISpeaker {
    public SpanishSpeaker() { }

    public void speak() {
        System.out.println("I speak Spanish.");
    }
}
  
```

```

classDiagram
    class ISpeaker {
        <<Interface>>
        <<Methods>>
        <<Speak>>
    }
    class EnglishSpeaker {
        <<ISpeaker>>
        <<Methods>>
        <<EnglishSpeaker>>
        <<Speak>>
    }
    class GermanSpeaker {
        <<ISpeaker>>
        <<Methods>>
        <<GermanSpeaker>>
        <<Speak>>
    }
    class SpanishSpeaker {
        <<ISpeaker>>
        <<Methods>>
        <<SpanishSpeaker>>
        <<Speak>>
    }
    class SpeakerFactory {
        <<Class>>
        <<Methods>>
        <<CreateSpeaker>>
    }
    class Language {
        <<Enum>>
        <<English>>
        <<German>>
        <<Spanish>>
    }
    EnglishSpeaker --> ISpeaker
    GermanSpeaker --> ISpeaker
    SpanishSpeaker --> ISpeaker
    SpeakerFactory --> EnglishSpeaker
    SpeakerFactory --> GermanSpeaker
    SpeakerFactory --> SpanishSpeaker
  
```

QUY TẮC CỦA DESIGN PATTERNS

❑ Program to interfaces, not implementations

```

public enum Language {
    English, German, Spanish;
}

public class SpeakerFactory {
    public static ISpeaker CreateSpeaker(Language language) {
        ISpeaker speaker;
        switch (language) {
            case Language.English:
                speaker = new EnglishSpeaker();
            case Language.German:
                speaker = new GermanSpeaker();
            case Language.Spanish:
                speaker = new SpanishSpeaker();
            default:
                throw new Exception("No speaker can speak such language");
        }
        return speaker;
    }
}

```

QUY TẮC CỦA DESIGN PATTERNS

❑ Program to interfaces, not implementations

❖ “Nâng cấp” 1:

- Khi gọi phương thức “speak()”, các speaker thống nhất nói “Hello” trước khi nói gì sau đó.



❖ “Nâng cấp” 2:

- Khi gọi phương thức “speak()”, các EnglishSpeaker và GermanSpeaker cần nói “Hello” trước khi nói gì sau đó, trong khi SpanishSpeaker thì nói “Hola” trước khi tiếp tục nói gì phía sau.

QUY TẮC CỦA DESIGN PATTERNS

❑ Program to interfaces, not implementations

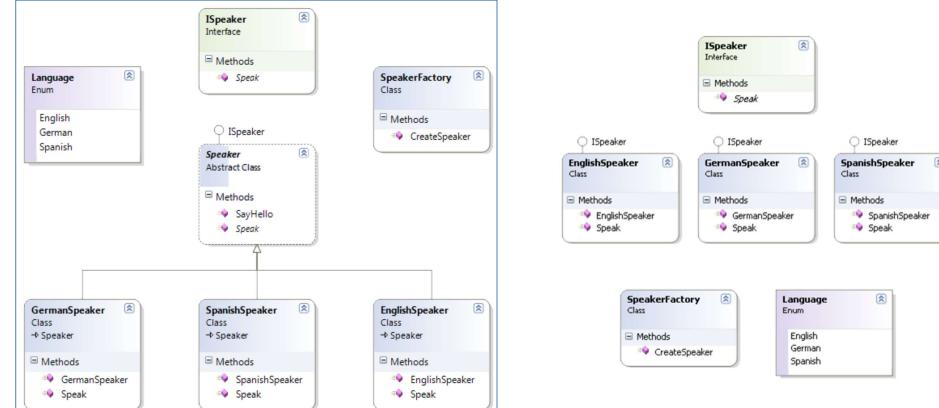
```

public static void main(String[] args) {
    ISpeaker speaker = SpeakerFactory.CreateSpeaker(args[0]);
    speaker.speak();
}

```

QUY TẮC CỦA DESIGN PATTERNS

❑ Program to interfaces, not implementations



QUY TẮC CỦA DESIGN PATTERNS

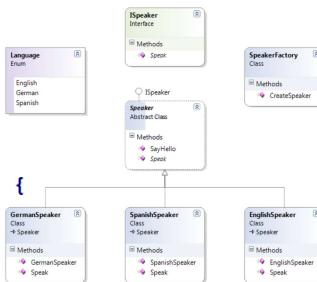
Program to interfaces, not implementations

```
public abstract class Speaker implements ISpeaker{
    public abstract void speak();

    public void sayHello(){
        System.out.println("Hello... ");
    }
}

public class EnglishSpeaker extends Speaker {
    public EnglishSpeaker() { }

    public void speak() {
        sayHello();
        System.out.println("I speak English.");
    }
}
```



QUY TẮC CỦA DESIGN PATTERNS

Program to interfaces, not implementations

```
public class SpanishSpeaker extends Speaker {
    public SpanishSpeaker() { }

    public void sayHello(){
        System.out.println("Holla... ");
    }

    public void speak() {
        sayHello();
        System.out.println("I speak Spanish.");
    }
}
```

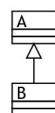
QUY TẮC CỦA DESIGN PATTERNS

Prefer composition over inheritance

❖ Kế thừa

là việc tái sử dụng lại một số thuộc tính, phương thức đã có sẵn từ lớp cơ sở

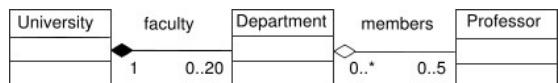
➤ Hầu hết các OOP không hỗ trợ đa kế thừa



❖ "Đối tượng kết hợp"

là đ.tượng được tạo từ việc kết hợp các kiểu dữ liệu và các đ.tượng khác.

➤ "Object composition is a way to combine objects or data types into more complex ones" (wiki).



Prefer composition over inheritance

❖ giúp dễ dàng sửa đổi code về sau

➤ dùng: Dependency Injection, Setters,...

❖ Chú ý: ưu tiên không có nghĩa là luôn luôn

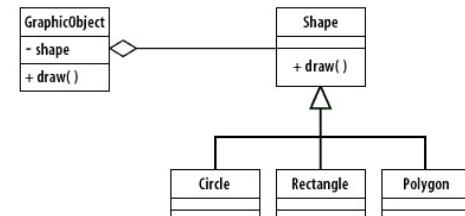
Khi nào?

❖ Kế thừa: **is-a**

➤ Vd: a car *is a* vehicle
a person *is a* mammal

❖ "Đối tượng kết hợp": **has-a**

➤ Vd: a car *has an* engine,
a person *has a* name



QUY TẮC CỦA DESIGN PATTERNS

Prefer composition over inheritance

- ❖ An employee IS A user
- ❖ An employee HAS A userinfo

The goal is to produce smaller code, easier to read, reuse and eventually extend further.

```
public class User
{
    public String UserName;
    public String Password;
    public String FirstName;
    public String LastName;
}

public class Employee : User
{
    public String EmployeeId;
    public String EmployeeCode;
    public String DepartmentId;
}

public class Member : User
{
    public String MemberId;
    public String JoinDate;
    public String ExpiryDate;
}

public class User
{
    public String UserName;
    public String Password;
    public String FirstName;
    public String LastName;
}

public class Employee
{
    public User EmployeeInfo;
    public String EmployeeId;
    public String EmployeeCode;
    public String DepartmentId;
}

public class Member
{
    public User MemberInfo;
    public String MemberId;
    public String JoinDate;
    public String ExpiryDate;
}
```

PHÂN LOẠI DESIGN PATTERNS

23 mẫu chia thành 3 nhóm:

❖ Structural Pattern (nhóm cấu trúc)

- thiết lập, định nghĩa quan hệ giữa các object và các thành phần của object.

By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	<ul style="list-style-type: none"> • Abstract Factory • Builder • Prototype • Singleton 	<ul style="list-style-type: none"> • Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy 	<ul style="list-style-type: none"> • Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

PHÂN LOẠI DESIGN PATTERNS

23 mẫu chia thành 3 nhóm:

❖ Creational Pattern (nhóm khởi tạo)

- cung cấp giải pháp để tạo ra các class/object,
- đảm bảo che giấu được logic của việc tạo (thay vì tạo trực tiếp bằng phương thức new)

By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	<ul style="list-style-type: none"> • Abstract Factory • Builder • Prototype • Singleton 	<ul style="list-style-type: none"> • Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy 	<ul style="list-style-type: none"> • Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

PHÂN LOẠI DESIGN PATTERNS

23 mẫu chia thành 3 nhóm:

❖ Behavioral Pattern: (nhóm hành vi)

- dùng trong thực hiện các hành vi của object, sự giao tiếp giữa các object với nhau.

By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	<ul style="list-style-type: none"> • Abstract Factory • Builder • Prototype • Singleton 	<ul style="list-style-type: none"> • Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy 	<ul style="list-style-type: none"> • Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

CREATIONAL DESIGN PATTERN - SINGLETON

❑ Yêu cầu:

- ❖ muốn có những đối tượng tồn tại duy nhất và có thể truy xuất mọi lúc mọi nơi

❑ Cách thông thường:

- ❖ sử dụng một biến toàn cục (public static final)
→ phá vỡ quy tắc *encapsulation* của OOP

CREATIONAL DESIGN PATTERN - SINGLETON

Singleton is a creational design pattern that lets you ensure that a class has only one instance and provide a global access point to this instance.

- ❖ đảm bảo chỉ duy nhất một *instance* được tạo ra
- ❖ và cung cấp một method cho phép truy xuất *instance* duy nhất đó mọi lúc mọi nơi trong chương trình.
- ❖ Tần suất sử dụng: ★★★★ cao trung bình

CREATIONAL DESIGN PATTERN - SINGLETON

❑ Singleton giải quyết các vấn đề sau:

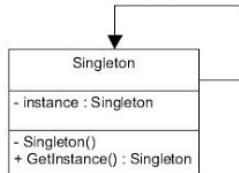
- ❖ Làm thế nào để đảm bảo rằng một lớp chỉ có một instance?
- ❖ Làm thế nào để dễ dàng truy cập instance của một lớp?
- ❖ Làm thế nào một lớp có thể kiểm soát quá trình khởi tạo của nó?
- ❖ Làm thế nào có thể hạn chế số lượng instance của một lớp?

CREATIONAL DESIGN PATTERN - SINGLETON

☐ Cách thực hiện

❖ **private constructor** để

- Đảm bảo không có nhiều hơn 1 instance được tạo ra tại 1 thời điểm
- Đảm bảo chỉ có thể tạo ra instance từ chính lớp này



❖ **Đặt private static final variable** để

- đảm bảo biến là duy nhất
- đảm bảo biến chỉ được khởi tạo và truy cập bên trong lớp.

❖ **Có một method public static** để

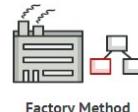
- trả về instance (duy nhất) của lớp đang cần khởi tạo.

CREATIONAL DESIGN PATTERN - SINGLETON

☐ Cài đặt:

```
01. class MySingleton {  
02.     private static final MySingleton instance = null;  
03.     private int something = 10;  
04.  
05.     // private constructor can't be accessed outside the class  
06.     private MySingleton() {}  
07.  
08.     // Factory method to provide the users with instances  
09.     public static MySingleton getInstance() {  
10.         if (instance == null)  
11.             instance = new MySingleton();  
12.         return instance;  
13.     }  
14.  
15.     public static void doSomething() {  
16.         something++;  
17.     }  
18.     public static void display() {  
19.         System.out.println(something);  
20.     }  
21. }  
22. class Main {  
23.     public static void main(String args[]) {  
24.         MySingleton a = MySingleton.getInstance();  
25.         MySingleton b = MySingleton.getInstance();  
26.         a.doSomething();  
27.         a.display(); //11  
28.         b.display(); //11  
29.     }  
30. }  
31. }
```

CREATIONAL DESIGN PATTERN - FACTORY METHOD



Factory Method

Factory Method is a creational design pattern that define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

- ❖ Ẩn chi tiết của khởi tạo đối tượng, quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đối tượng linh hoạt hơn.
- ❖ tham chiếu đến đối tượng mới được tạo ra bằng cách sử dụng một interface chung.
- ❖ Tần suất sử dụng: ★★★★★ cao.

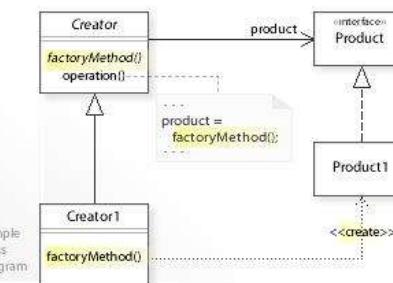
CREATIONAL DESIGN PATTERN - FACTORY METHOD

Cách thực hiện

- ❖ **Super Class:** một supper class trong Factory Pattern có thể là một **interface**, **abstract class** hay một **class** thông thường.
- ❖ **Sub Classes:** các sub class sẽ implement các phương thức của **supper class** theo nghiệp vụ riêng của nó.
- ❖ **Factory Class:** một class chịu trách nhiệm khởi tạo các đối tượng **sub class** dựa theo tham số đầu vào.
 - Lưu ý: lớp này là Singleton hoặc lớp này phải cung cấp một **public static method** cho việc khởi tạo các đối tượng.
 - Factory class sử dụng if-else hoặc switch-case để phân loại tham số đầu vào, từ đó xác định class con đầu ra.

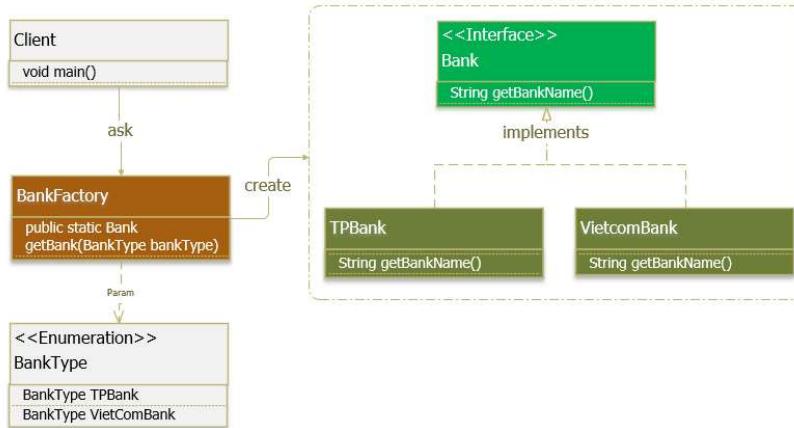
CREATIONAL DESIGN PATTERN - FACTORY METHOD

Cài đặt:



CREATIONAL DESIGN PATTERN - FACTORY METHOD

Cài đặt minh họa:



CREATIONAL DESIGN PATTERN - FACTORY METHOD

Cài đặt minh họa:

```

01. public enum BankType {
02.     VIETCOMBANK, TPBANK;
03. }
04.
05. public interface Bank {
06.     String getBankName();
07. }
08.
09. public class TPBank implements Bank {
10.     public String getBankName() {
11.         return "TPBank";
12.     }
13. }
14. public class VietcomBank implements Bank {
15.     public String getBankName() {
16.         return "VietcomBank";
17.     }
18. }
19.
20. public class BankFactory {
21.     private BankFactory() { }
22.     public static final Bank getBank(BankType bankType) {
23.         switch (bankType) {
24.             case TPBANK: return new TPBank();
25.             case VIETCOMBANK: return new VietcomBank();
26.             default:
27.                 throw new IllegalArgumentException(
28.                     "This bank type is unsupported");
29.         }
30.     }
31.
32.     public class Client {
33.         public static void main(String[] args) {
34.             Bank bank = BankFactory.getBank(BankType.TPBANK);
35.             System.out.println(bank.getBankName()); //TPBank
36.         }
37.     }
38. }
  
```

CREATIONAL DESIGN PATTERN - FACTORY METHOD

☐ Sử dụng khi:

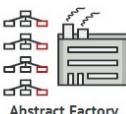
- ❖ có một super class với nhiều sub class
và dựa trên đầu vào, ta cần khởi tạo một sub class cụ thể
→ Mô hình này giúp đưa việc khởi tạo một class từ phía người dùng (client) sang cho Factory.
- ❖ không biết sẽ cần đến những sub class nào trong tương lai
Khi cần mở rộng, thì
tạo 1 sub class mới
và thêm vào factory method việc khởi tạo sub class này.

CREATIONAL DESIGN PATTERN - FACTORY METHOD

☐ Lợi ích của Factory Pattern:

- ❖ Factory Pattern giúp
 - giảm sự phụ thuộc giữa các module (loose coupling): cung cấp 1 hướng tiếp cận với Interface thay thế các implement.
 - giúp chương trình độc lập với những lớp cụ thể mà chúng ta cần tạo.
 - code ở phía client không bị ảnh hưởng khi thay đổi logic ở factory hay sub class.
 - Khởi tạo các Objects mà che giấu đi xử lý logic của việc khởi tạo đấy. Người dùng không biết logic thực sự được khởi tạo bên dưới phương thức factory.
- ❖ Mở rộng code dễ dàng hơn
 - khi cần mở rộng, chỉ việc tạo ra sub class và implement thêm vào factory method.
- ❖ Thống nhất về naming convention: giúp cho các developer có thể hiểu về cấu trúc source code.

CREATIONAL DESIGN PATTERN - ABSTRACT FACTORY



Abstract Factory Method is a creational design pattern that provide an interface for creating families of related or dependent objects without specifying their concrete classes.

- ❖ tạo ra một Super-factory dùng để tạo ra các Factory khác (Factory của các Factory).
- ❖ Tần suất sử dụng: ★★★★★ cao.

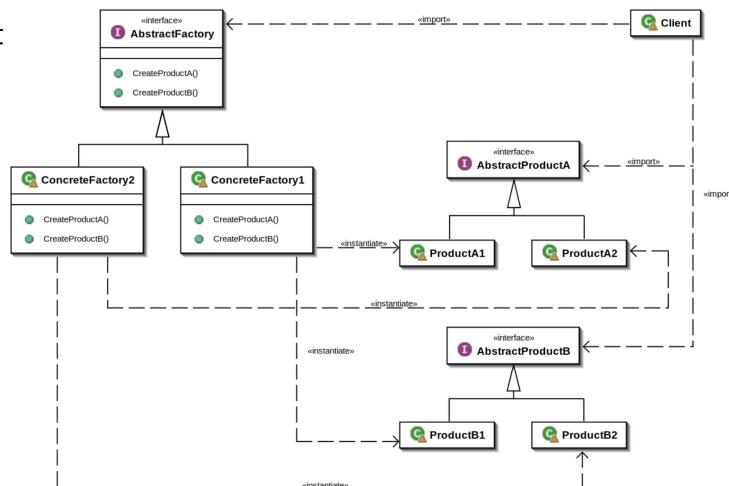
CREATIONAL DESIGN PATTERN - ABSTRACT FACTORY

Cách thực hiện

- ❖ **AbstractFactory**: là một interface hoặc abstract class chứa các phương thức chung của các Factory.
- ❖ **ConcreteFactory**: là cài đặt của một Factory cụ thể (implements/extends từ AbstractFactory).
- ❖ **AbstractProduct**: được khai báo ở dạng interface hoặc abstract class để định nghĩa các thành phần chung của từng nhóm Product (ít nhất 2).
- ❖ **Product**: là cài đặt của một Product cụ thể của một trong các AbstractProduct đã khai báo.
- ❖ **Client**: là đối tượng muốn tạo và sử dụng các Product (thông qua việc sử dụng AbstractFactory và các AbstractProduct).

CREATIONAL DESIGN PATTERN - ABSTRACT FACTORY

Cài đặt



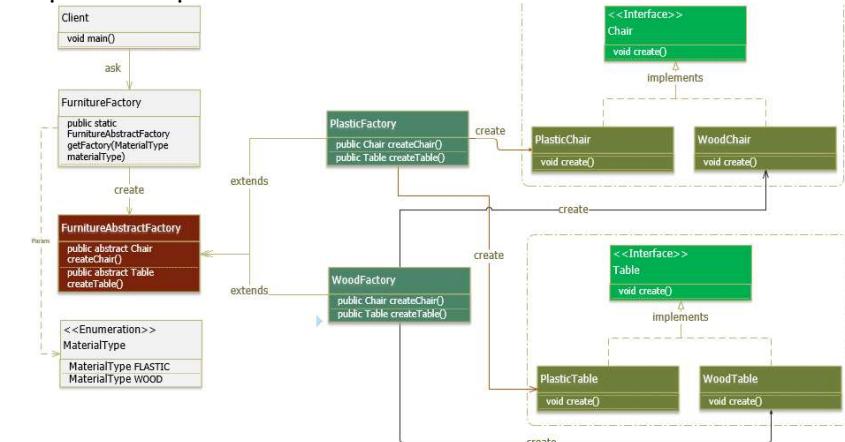
CREATIONAL DESIGN PATTERN - ABSTRACT FACTORY

Cài đặt minh họa:

- ❖ Một nhà máy đồ nội thất chuyên sản xuất ghế (**Chair**)
 - ghế nhựa (**PlasticChair**) và ghế gỗ (**WoodChair**).
- ❖ Kinh doanh thuận lợi nên nhà máy mở rộng thêm sản xuất bàn (**Table**):
 - bàn nhựa (**PlasticTable**) và bàn gỗ (**WoodTable**).
- ❖ Vì quy trình sản xuất ghế/bàn theo từng chất liệu (**MaterialType**) là khác nhau, nên nhà máy (**FurnitureAbstractFactory**) tách ra thành 2 nhà máy con (cả 2 đều có thể sản xuất ghế và bàn):
 - **PlasticFactory**: nhà máy sản xuất vật liệu bằng nhựa
 - **WoodFactory**: nhà máy sản xuất vật liệu bằng gỗ.
- ❖ Khi khách hàng (**Client**) cần mua một món đồ, khách hàng chỉ cần đến cửa hàng (**FurnitureFactory**) để mua.
 - Khi đó ứng với từng loại hàng hóa và vật liệu, đơn hàng sẽ được chuyển về nhà máy con tương ứng để sản xuất (**createXXX**) ra bàn và ghế.

CREATIONAL DESIGN PATTERN - ABSTRACT FACTORY

Cài đặt minh họa:



CREATIONAL DESIGN PATTERN - ABSTRACT FACTORY

☐ Cài đặt minh họa:

```
85. public enum MaterialType {  
86.     FLASTIC, WOOD  
87. }  
88.  
89. public class Client {  
90.     public static void main(String[] args) {  
91.         FurnitureAbstractFactory factory =  
92.             FurnitureFactory.getFactory(MaterialType.FLASTIC);  
93.  
94.         Chair chair = factory.createChair();  
95.         chair.create(); // Create plastic chair  
96.  
97.         Table table = factory.createTable();  
98.         table.create(); // Create plastic table  
99.     }  
100. }
```

CREATIONAL DESIGN PATTERN - ABSTRACT FACTORY

☐ Cài đặt minh họa:

```
01. //-----SuperFactory  
02. public class FurnitureFactory {  
03.     private FurnitureFactory() { }  
04.  
05.     public static FurnitureAbstractFactory  
06.         getFactory(MaterialType materialType) {  
07.             switch (materialType) {  
08.                 case MaterialType.FLASTIC:  
09.                     return new PlasticFactory();  
10.                 case MaterialType.WOOD:  
11.                     return new WoodFactory();  
12.                 default:  
13.                     throw new UnsupportedOperationException  
14.                         ("This furniture is unsupported ");  
15.             }  
16.         }  
17.     }
```

CREATIONAL DESIGN PATTERN - ABSTRACT FACTORY

☐ Cài đặt minh họa:

```
19. //-----AbstractFactory & ConcreteFactory  
20. public abstract class FurnitureAbstractFactory {  
21.     public abstract Chair createChair();  
22.     public abstract Table createTable();  
23.  
24.  
25.     public class PlasticFactory extends FurnitureAbstractFactory {  
26.         @Override  
27.         public Chair createChair() {  
28.             return new PlasticChair();  
29.         }  
30.         @Override  
31.         public Table createTable() {  
32.             return new PlasticTable();  
33.         }  
34.  
35.     public class WoodFactory extends FurnitureAbstractFactory {  
36.         @Override  
37.         public Chair createChair() {  
38.             return new WoodChair();  
39.         }  
40.         @Override  
41.         public Table createTable() {  
42.             return new WoodTable();  
43.         }  
44.     }
```

CREATIONAL DESIGN PATTERN - ABSTRACT FACTORY

☐ Cài đặt minh họa:

```
47. //-----AbstractProduct and Product: Chair  
48. public interface Chair {  
49.     void create();  
50. }  
51.  
52. public class PlasticChair implements Chair {  
53.     @Override  
54.     public void create() {  
55.         System.out.println("Create plastic chair");  
56.     }  
57. }  
58.  
59. public class WoodChair implements Chair {  
60.     @Override  
61.     public void create() {  
62.         System.out.println("Create wood chair");  
63.     }  
64. }  
66. //-----AbstractProduct and Product: Table  
67. public interface Table {  
68.     void create();  
69. }  
70.  
71. public class PlasticTable implements Table {  
72.     @Override  
73.     public void create() {  
74.         System.out.println("Create plastic table");  
75.     }  
76. }  
77.  
78. public class WoodTable implements Table {  
79.     @Override  
80.     public void create() {  
81.         System.out.println("Create wood table");  
82.     }  
83. }
```

STRUCTURAL DESIGN PATTERN - ADAPTER

❑ Tình huống:

- ❖ Đang có một lớp A đã có sẵn (đã cài đặt) toàn bộ (hoặc một phần) dịch vụ mà bạn cần.
- ❖ Lớp B đang dùng hiện tại (cũng đã có sẵn) cần có các dịch vụ mà lớp A đang cung cấp.
- ❖ Lớp B không gọi được các phương thức trong lớp A, vì interface không tương thích.

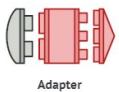
❑ Yêu cầu:

- ❖ Lớp B dùng được các dịch vụ do lớp A cung cấp, mà không được sửa code của lớp A và lớp B

❑ Ví dụ:

```
❖ BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
  > BufferedReader: lớp B, nhận tham số đầu vào là 1 char-stream  
  > System.in: lớp A, trả về 1 byte-stream
```

STRUCTURAL DESIGN PATTERN - ADAPTER



Adapter Pattern is a structural design pattern that convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

- ❖ chuyển đổi interface của một hay nhiều lớp có sẵn thành một interface khác, thích hợp cho lớp đang viết.
- ❖ giữ vai trò trung gian giữa hai lớp, cho phép các lớp có các interface khác nhau có thể giao tiếp với nhau thông qua interface trung gian, mà không cần thay đổi code của lớp có sẵn cũng như lớp đang viết.
- ❖ Tần suất sử dụng: ★★★★ cao trung bình.

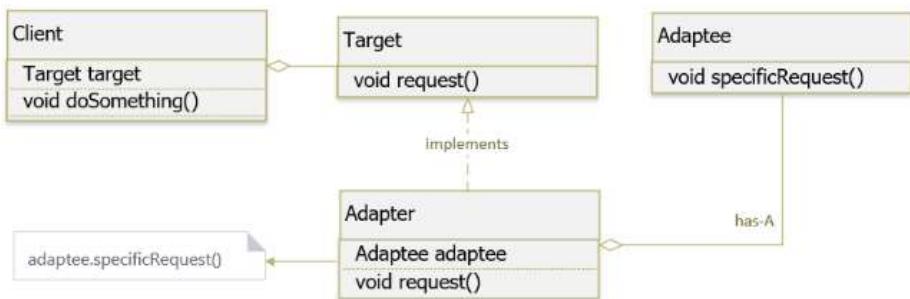
STRUCTURAL DESIGN PATTERN - ADAPTER

❑ Cách thực hiện:

- ❖ **Target:** interface chứa các chức năng được sử dụng bởi Client.
- ❖ **Adaptee:** interface chứa các chức năng hữu ích, cần được tích hợp vào Target, nhưng không tương thích với Target.
- ❖ **Adapter:** lớp trung gian, kế thừa từ Target, đồng thời tích hợp Adaptee, và có các phương thức chuyển đổi giúp Target làm việc được với Adaptee.
- ❖ **Client:** lớp sử dụng các chức năng có trong interface Target.

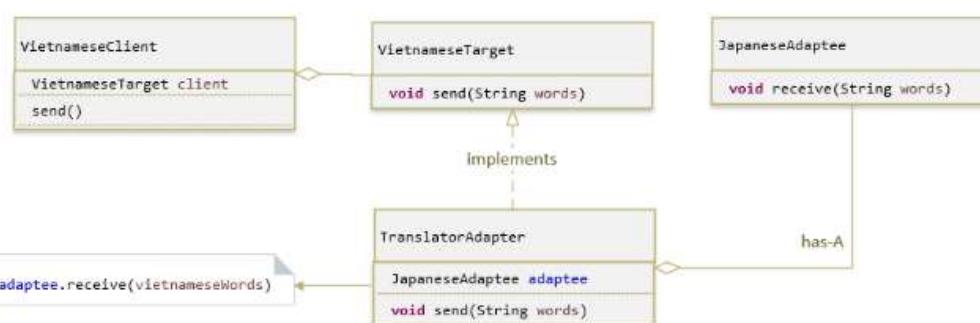
STRUCTURAL DESIGN PATTERN - ADAPTER

Cài đặt



STRUCTURAL DESIGN PATTERN - ADAPTER

Cài đặt minh họa:



STRUCTURAL DESIGN PATTERN - ADAPTER

Cài đặt minh họa:

```
01. public interface VietnameseTarget {
02.     //chi xử lý "words" viết bằng tiếng Việt
03.     void send(String words);
04. }
05.
06. public class JapaneseAdaptee {
07.     //chi xử lý "words" viết bằng tiếng Nhật
08.     public void receive(String words) {
09.         System.out.println("Retrieving words from Adapter ...");
10.         System.out.println(words);
11.     }
12. }
13.
14. public class VietnameseClient {
15.     public static void main(String[] args) {
16.         VietnameseTarget client = new TranslatorAdapter(new JapaneseAdaptee());
17.         client.send("Xin chào");
18.     }
19. }
```

STRUCTURAL DESIGN PATTERN - ADAPTER

Cài đặt minh họa:

```
21. public class TranslatorAdapter implements VietnameseTarget {  
22.     private JapaneseAdaptee adaptee;  
23.  
24.     public TranslatorAdapter(JapaneseAdaptee adaptee) {  
25.         this.adaptee = adaptee;  
26.     }  
27.  
28.     @Override  
29.     public void send(String words) {  
30.         System.out.println("Reading Words ...");  
31.         System.out.println(words);  
32.         String vietnameseWords = this.translate(words);  
33.         System.out.println("Sending Words ...");  
34.         adaptee.receive(vietnameseWords);  
35.     }  
36.  
37.     private String translate(String vietnameseWords) {  
38.         System.out.println("Translated!");  
39.         return "こんにちは";  
40.     }  
41. }
```

STRUCTURAL DESIGN PATTERN - ADAPTER

☐ Cài đặt minh họa:

❖ Output:

```
1 | Reading Words ...
2 | Xin chào
3 | Translated!
4 | Sending Words ...
5 | Retrieving words from Adapter ...
6 | こんにちは
```

STRUCTURAL DESIGN PATTERN - ADAPTER

☐ Cài đặt minh họa 2:

```
01. interface Stack{
02.     void push (Object o);
03.     Object pop ();
04.     Object top ();
05. }
06.
07. class DList { /* DoubleLinkedList */
08.     public void insert (DNode pos, Object o) {...}
09.     public void remove (DNode pos, Object o) {...}
10.
11.     public void insertHead (Object o) {...}
12.     public void insertTail (Object o) {...}
13.
14.     public Object removeHead () {...}
15.     public Object removeTail () {...}
16.
17.     public Object getHead () {...}
18.     public Object getTail () {...}
19. }

21. /* Adapt DList class to Stack interface */
22. class DListStack implements Stack {
23.     private DList _dlist;
24.
25.     public DListStack() { _dlist = new DList(); }
26.     public void push (Object o) {
27.         _dlist.insertTail (o);
28.     }
29.
30.     public Object pop () {
31.         return _dlist.removeTail ();
32.     }
33.
34.     public Object top () {
35.         return _dlist.getTail ();
36.     }
37. }
```

STRUCTURAL DESIGN PATTERN - ADAPTER

☐ Dùng Adapter Pattern trong những trường hợp sau:

- ❖ khi không thể kế thừa lớp A, nhưng muốn một lớp B có những xử lý tương tự như lớp A. Khi đó chúng ta có thể cài đặt B theo Object Adapter, các xử lý của B sẽ gọi những xử lý của A khi cần.
- ❖ Khi muốn sử dụng một lớp đã tồn tại trước đó nhưng interface sử dụng không phù hợp như mong muốn.
- ❖ Khi muốn tạo ra những lớp có khả năng sử dụng lại, chúng phối hợp với các lớp không liên quan hay những lớp không thể đoán trước được và những lớp này không có những interface tương thích.
- ❖ Cần phải có sự chuyển đổi interface từ nhiều nguồn khác nhau.
- ❖ Khi cần đảm bảo nguyên tắc Open/Close trong một ứng dụng.
 - nên hạn chế việc chỉnh sửa bên trong một Class hoặc Module có sẵn, thay vào đó hãy xem xét mở rộng chúng.
 - (Software modules should be closed for modifications but open for extensions)

STRUCTURAL DESIGN PATTERN - FACADE

❑ Tình huống:

- ❖ Đang có sẵn một số lớp A, B, C, D, E cài đặt chức năng hữu ích.
- ❖ Dịch vụ X bao gồm một chuỗi các chức năng được cung cấp bởi A, B, D, E
- ❖ Dịch vụ Y là sự kết hợp của một chuỗi các chức năng được cung cấp bởi A, C, E
- ❖ User1 đang viết lớp F cần cài đặt dịch vụ X và Y.

❑ Yêu cầu:

- ❖ Lớp F vẫn cài đặt được dịch vụ X, Y nhưng User1 không cần biết chi tiết các logic cần thiết để xây dựng X, Y



STRUCTURAL DESIGN PATTERN - FACADE

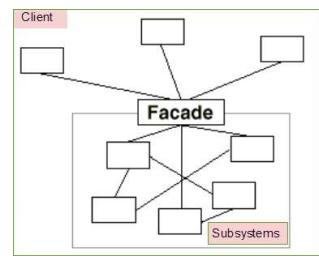
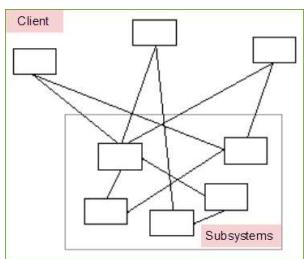
Facade Pattern is a structural design pattern that serves as a front-facing interface masking more complex underlying or structural code.

- ❖ cung cấp một giao diện chung (đơn nhất), đơn giản, ở cấp độ cao hơn thay cho một nhóm các giao diện có trong hệ thống con (subsystem),
- ❖ qua đó che giấu các hoạt động phức tạp bên trong hệ thống con và giúp sử dụng hệ thống con này dễ dàng hơn.
- ❖ Tần suất sử dụng: ★★★★★ cao

STRUCTURAL DESIGN PATTERN - FACADE

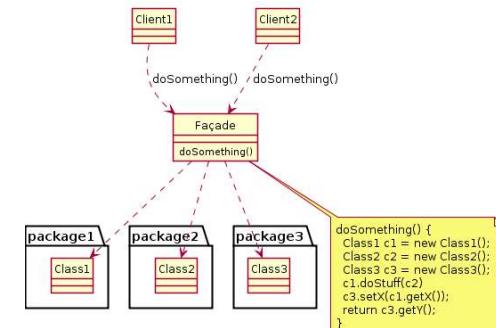
❑ Façade pattern

- ❖ sử dụng khi có một hệ thống rất phức tạp, khó hiểu (bởi vì hệ thống có quá nhiều các lớp phụ thuộc lẫn nhau, hoặc vì không tiếp cận được mã nguồn của hệ thống)
- ❖ giúp ẩn đi sự phức tạp của hệ thống lớn, và cung cấp một giao diện đơn giản cho client để sử dụng hệ thống này.



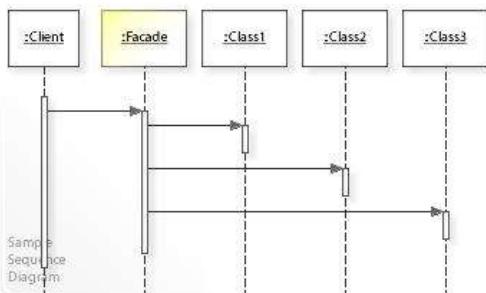
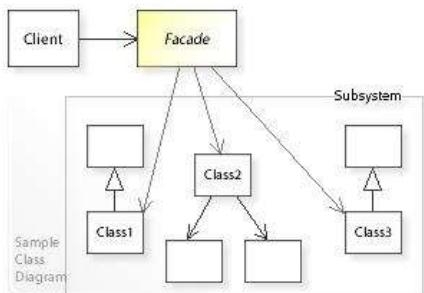
❑ Cách thực hiện

- ❖ Subsystem:
 - chứa các lớp con phụ thuộc lẫn nhau (một cách phức tạp).
 - Các lớp trong Subsystem không tham chiếu đến Facade
- ❖ Facade:
 - chứa các tham chiếu của các lớp con trong subsystem,
 - sử dụng các tham chiếu này cài đặt các dịch vụ cần thiết.
- ❖ Client:
 - Sử dụng subsystem thông qua các dịch vụ cung cấp bởi Facade (không truy cập trực tiếp subsystem).



STRUCTURAL DESIGN PATTERN - FACADE

☐ Cài đặt



STRUCTURAL DESIGN PATTERN - FACADE

☐ Cài đặt minh họa 1

```

18. /* Facade */
19. class ComputerFacade {
20.     private final CPU processor;
21.     private final Memory ram;
22.     private final HardDrive hd;
23.
24.     public ComputerFacade() {
25.         this.processor = new CPU();
26.         this.ram = new Memory();
27.         this.hd = new HardDrive();
28.     }
29.
30.     public void start() {
31.         processor.freeze();
32.         ram.load(BOOT_ADDRESS,
33.             hd.read(BOOT_SECTOR, SECTOR_SIZE));
34.         processor.jump(BOOT_ADDRESS);
35.         processor.execute();
36.     }
37. }
  
```

```

01. /* Complex parts */
02. class CPU {
03.     public void freeze() { ... }
04.     public void jump(long position) { ... }
05.     public void execute() { ... }
06. }
07.
08. class HardDrive {
09.     public byte[] read(long lba, int size)
10.     { ... }
11.
12.
13. class Memory {
14.     public void load(long position, byte[] data)
15.     { ... }
16.
17. /* Client */
18. class You {
19.     public static void main(String[] args) {
20.         ComputerFacade computer =
21.             new ComputerFacade();
22.         computer.start();
23.     }
24. }
  
```

STRUCTURAL DESIGN PATTERN - FACADE

☐ Cài đặt minh họa 2

- ❖ Một công ty bán hàng online cung cấp nhiều lựa chọn cho khách hàng khi mua sản phẩm.
- ❖ Khi một sản phẩm được mua, nó sẽ qua các bước xử lý: lấy thông tin về tài khoản mua hàng, thanh toán, vận chuyển, gửi Email/SMS thông báo.

☐ Cài đặt minh họa 2

- ❖ Hệ thống dùng Facade sẽ như sau:
 - AccountService: lấy thông tin cơ bản của khách hàng thông qua email được cung cấp.
 - PaymentService: có thể thanh toán qua Paypal, Credit Card, E-banking, cash.
 - ShippingService: có thể chọn Free Shipping, Standard Shipping, Express Shipping.
 - EmailService: có thể gửi email thông báo cho khách hàng về tình hình đặt hàng, thanh toán, vận chuyển, nhận hàng.
 - SmsService: có thể gửi SMS thông báo cho khách hàng khi thanh toán online.
 - ShopFacade: bao gồm các dịch vụ cung cấp để Client mua hàng dễ dàng.
 - Tùy vào nghiệp vụ mà nó sẽ sử dụng những dịch tương ứng, chẳng hạn dịch vụ SMS chỉ được sử dụng nếu khách hàng đăng ký mua hàng thông qua hình thức thanh toán online (Paypal, E-banking, ...).
 - Client: là người dùng cuối sử dụng ShopFacade để mua hàng.

STRUCTURAL DESIGN PATTERN - FACADE

☐ Cài đặt minh họa 2

```
01. public class AccountService {  
02.     public void getAccount(String email) {  
03.         System.out.println("Getting the account of "+email);  
04.     }  
05.  
06.     public class PaymentService {  
07.         public void paymentByPaypal() {  
08.             System.out.println("Payment by Paypal");  
09.         }  
10.  
11.         public void paymentByCreditCard() {  
12.             System.out.println("Payment by Credit Card");  
13.         }  
14.  
15.         public void paymentByEbankingAccount() {  
16.             System.out.println("Payment by E-banking account");  
17.         }  
18.  
19.         public void paymentByCash() {  
20.             System.out.println("Payment by cash");  
21.         }  
22.  
23.     }
```

```
25.     public class ShippingService {  
26.         public void freeShipping() {  
27.             System.out.println("Free Shipping");  
28.         }  
29.  
30.         public void standardShipping() {  
31.             System.out.println("Standard Shipping");  
32.         }  
33.  
34.         public void expressShipping() {  
35.             System.out.println("Express Shipping");  
36.         }  
37.  
38.     public class EmailService {  
39.         public void sendMail(String mailTo) {  
40.             System.out.println("Sending an email to "+mailTo);  
41.         }  
42.  
43.     }  
44.     public class SmsService {  
45.         public void sendSMS(String mobilePhone) {  
46.             System.out.println("Sending a sms to "+mobilePhone);  
47.         }  
48.  
49.     }
```

STRUCTURAL DESIGN PATTERN - FACADE

☐ Cài đặt minh họa 2

```
51. public class ShopFacade {  
52.     private static final ShopFacade INSTANCE  
53.             = new ShopFacade();  
54.  
55.     private AccountService accountService;  
56.     private PaymentService paymentService;  
57.     private ShippingService shippingService;  
58.     private EmailService emailService;  
59.     private SmsService smsService;  
60.  
61.     private ShopFacade() {  
62.         accountService = new AccountService();  
63.         paymentService = new PaymentService();  
64.         shippingService = new ShippingService();  
65.         emailService = new EmailService();  
66.         smsService = new SmsService();  
67.     }  
68.  
69.     public static ShopFacade getInstance() {  
70.         return INSTANCE;  
71.     }
```

```
73.     public void buyProductByCashWithFreeShipping  
74.             (String email) {  
75.                 accountService.getAccount(email);  
76.                 paymentService.paymentByCash();  
77.                 shippingService.freeShipping();  
78.                 emailService.sendMail(email);  
79.                 System.out.println("Done\n");  
80.  
81.             }  
82.  
83.     public void buyProductByPaypalWithStandardShipping  
84.             (String email, String mobilePhone) {  
85.                 accountService.getAccount(email);  
86.                 paymentService.paymentByPaypal();  
87.                 shippingService.standardShipping();  
88.                 emailService.sendMail(email);  
89.                 smsService.sendSMS(mobilePhone);  
90.                 System.out.println("Done\n");  
91.             }
```

STRUCTURAL DESIGN PATTERN - FACADE

☐ Cài đặt minh họa 2

```
93. public class Client {  
94.     public static void main(String[] args) {  
95.         ShopFacade.getInstance().  
96.             buyProductByCashWithFreeShipping  
97.                     ("contact@gpcoder.com");  
98.         ShopFacade.getInstance().  
99.             buyProductByPaypalWithStandardShipping  
100.                    ("gpcodervn@gmail.com", "0988.999.999");  
101.  
102.    }
```

STRUCTURAL DESIGN PATTERN - FACADE

☐ Lợi ích của Facade Pattern

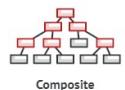
- ❖ Giúp cho hệ thống trở nên đơn giản hơn trong việc hiểu và sử dụng
- ❖ Giảm sự phụ thuộc của các code bên ngoài với cài đặt code bên trong của thư viện → cho phép linh động trong phát triển hệ thống.
- ❖ Đóng gói một tập nhiều hàm API được thiết kế không tốt/phức tạp bằng một hàm API đơn có thiết kế tốt hơn.

STRUCTURAL DESIGN PATTERN - FACADE

Sử dụng Facade Pattern khi

- ❖ Hệ thống phức tạp, có nhiều lớp phụ thuộc lẫn nhau → khó hiểu, khó sử dụng
- ❖ Hệ thống có nhiều hệ thống con, mỗi hệ thống con lại có các "giao tiếp" khác nhau → khó hiểu, khó sử dụng phối hợp.
- ❖ Khi muốn tách biệt/giảm phụ thuộc giữa chức năng đang cài đặt với các hệ thống con → tăng khả năng độc lập và khả chuyển của hệ thống con, dễ chuyển đổi nâng cấp trong tương lai.
- ❖ Khi muốn phân lớp các hệ thống con → dùng Façade Pattern để định nghĩa cổng giao tiếp chung cho mỗi hệ thống con, làm giảm sự phụ thuộc vào các hệ thống con vì các hệ thống này chỉ giao tiếp với nhau thông qua các giao diện chung đó.
- ❖ Khi muốn bao bọc, che giấu tính phức tạp trong các hệ thống con đối với phía Client.

STRUCTURAL DESIGN PATTERN - COMPOSITE



Composite Pattern is a structural design pattern that describes a group of objects that is treated the same way as a single instance of the same type of object.

The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies.

- ❖ tổng hợp và cấu trúc thành cây những thành phần có quan hệ với nhau để tạo ra thành phần lớn hơn.
- ❖ cho phép xử lý một nhóm object tương tự theo cách xử lý 1 object.

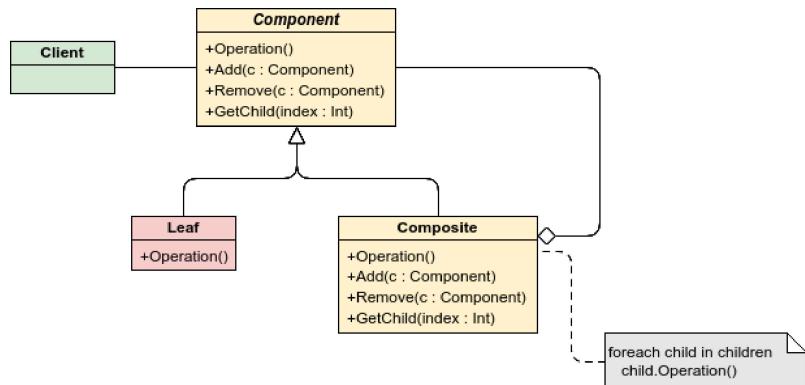
- ❖ Tần suất sử dụng: ★★★★ cao trung bình

Cách thực hiện

- ❖ Base Component: một interface hoặc abstract class quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.
- ❖ Leaf: là lớp implement các phương thức của Component.
- ❖ Composite: là lớp chứa tập hợp các Leaf và cài đặt các phương thức của Base Component bằng cách ủy nhiệm cho các thành phần con (Leaf) xử lý.
- ❖ Client: sử dụng Base Component để làm việc với các đối tượng trong Composite.

STRUCTURAL DESIGN PATTERN - COMPOSITE

Cài đặt



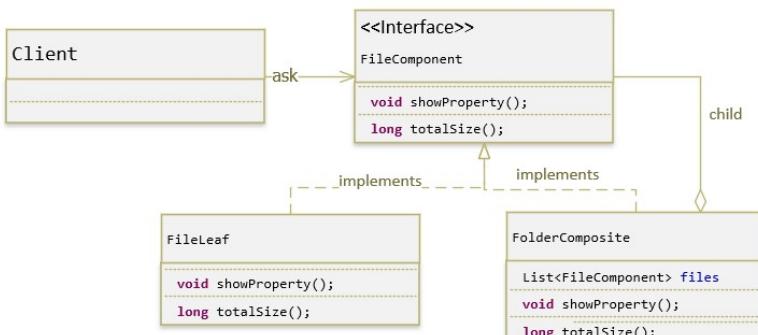
STRUCTURAL DESIGN PATTERN - COMPOSITE

Cài đặt minh họa

- ❖ 2 đối tượng: **File** và **Folder**
 - ❖ Cả 2 đối tượng đều có 2 phương thức:
 - `long totalSize()`: trả về dung lượng của đối tượng
 - `void showProperty()`: hiển thị các thuộc tính của đối tượng
 - ❖ Một chương trình **Client** quản lý danh sách 2 đối tượng trên và thực hiện các câu lệnh hiển thị ra màn hình tổng dung lượng của 1 đối tượng bất kỳ trong danh sách

STRUCTURAL DESIGN PATTERN - COMPOSITE

Cài đặt minh họa



STRUCTURAL DESIGN PATTERN - COMPOSITE

Cài đặt minh họa

```
01. public interface FileComponent {  
02.     void showProperty();  
03.     long totalSize();  
04. }  
05.  
06. public class FileLeaf implements FileComponent {  
07.     private String name;  
08.     private long size;  
09.  
10.    public FileLeaf(String name, long size) {  
11.        super();  
12.        this.name = name;  
13.        this.size = size;  
14.    }  
15.  
16.    @Override  
17.    public long totalSize() {  
18.        return size;  
19.    }  
20.  
21.    @Override  
22.    public void showProperty() {  
23.        System.out.println("FileLeaf [name=" +  
24.            + name + ", size=" + size + "]");  
25.    }  
26.  
27.  
28. public class FolderComposite implements FileComponent {  
29.     private List<FileComponent> files = new ArrayList<>();  
30.  
31.    public FolderComposite(List<FileComponent> files) {  
32.        this.files = files;  
33.    }  
34.  
35.    @Override  
36.    public void showProperty() {  
37.        for (FileComponent file : files) {  
38.            file.showProperty();  
39.        }  
40.    }  
41.  
42.    @Override  
43.    public long totalSize() {  
44.        long total = 0;  
45.        for (FileComponent file : files) {  
46.            total += file.totalSize();  
47.        }  
48.        return total;  
49.    }  
50. }
```

STRUCTURAL DESIGN PATTERN - COMPOSITE

☐ Cài đặt minh họa

```

52. public class Client {
53.
54.     public static void main(String[] args) {
55.         FileComponent file1 = new FileLeaf("file 1", 10);
56.         FileComponent file2 = new FileLeaf("file 2", 5);
57.         FileComponent file3 = new FileLeaf("file 3", 12);
58.
59.         List<FileComponent> files = Arrays.asList(file1, file2, file3);
60.         FileComponent folder = new FolderComposite(files);
61.         folder.showProperty();
62.         System.out.println("Total Size: " + folder.totalSize());
63.     }
64. }
```

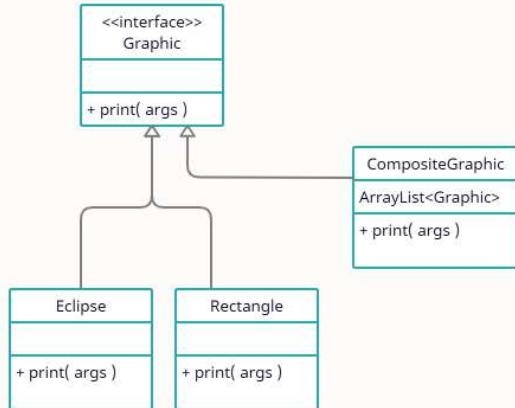
STRUCTURAL DESIGN PATTERN - COMPOSITE

☐ Cài đặt minh họa 2

- ❖ Trong ứng dụng vẽ/xử lý ảnh, mỗi hình (eclipse, rectangle,...) là một đối tượng (lớp) thuộc nhóm hình (Graphic).
- ❖ Ta muốn thực hiện một số thao tác (vd: print()) đối với từng nhóm hình. (chứ không muốn thực hiện với từng hình)

STRUCTURAL DESIGN PATTERN - COMPOSITE

☐ Cài đặt minh họa 2



STRUCTURAL DESIGN PATTERN - COMPOSITE

☐ Cài đặt minh họa 2

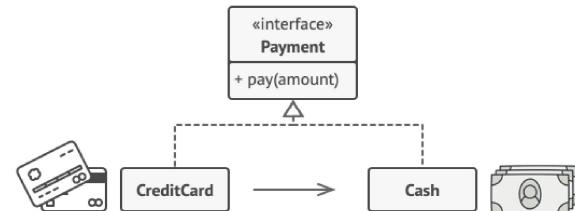
```

01. /**
02.  * Component
03.  */
04. interface Graphic {
05.     //Prints the graphic.
06.     public void print();
07.
08. /**
09.  * Composite
09.  */
10. class CompositeGraphic implements Graphic {
11.     //Collection of child graphics.
12.     private final ArrayList<Graphic> childGraphics
13.         = new ArrayList<>();
14.
15.     //Adds the graphic to the composition.
16.     public void add(Graphic graphic) {
17.         childGraphics.add(graphic);
18.     }
19.
20.     @Override
21.     public void print() {
22.         for (Graphic graphic : childGraphics) {
23.             graphic.print(); //Delegation
24.         }
25.     }
26.
27. /**
28.  * Leaf
28.  */
29. class Ellipse implements Graphic {
30.     @Override
31.     public void print() {
32.         System.out.println("Ellipse");
33.     }
34.
35. /**
36.  * Client
36.  */
37. public class CompositeDemo {
38.     public static void main(String[] args) {
39.         //Initialize four ellipses
40.         Ellipse ellipse1 = new Ellipse();
41.         Ellipse ellipse2 = new Ellipse();
42.         Ellipse ellipse3 = new Ellipse();
43.         Ellipse ellipse4 = new Ellipse();
44.
45.         //Creates two composites containing the ellipses
46.         CompositeGraphic graphic2 = new CompositeGraphic();
47.         graphic2.add(ellipse1);
48.         graphic2.add(ellipse2);
49.         graphic2.add(ellipse3);
50.
51.         CompositeGraphic graphic3 = new CompositeGraphic();
52.         graphic3.add(ellipse4);
53.
54.         //Create another graphics that contains two graphics
55.         CompositeGraphic graphic1 = new CompositeGraphic();
56.         graphic1.add(graphic2);
57.         graphic1.add(graphic3);
58.
59.         //Prints the complete graphic (4 string "Ellipse").
60.         graphic1.print();
61.     }
62. }
```

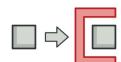
STRUCTURAL DESIGN PATTERN - PROXY

❑ Tình huống thực tế:

- ❖ Thẻ tín dụng (CreditCard) đóng vai trò như một proxy cho lượng tiền mặt (cash) một người đang có trong tài khoản ngân hàng.



STRUCTURAL DESIGN PATTERN - PROXY



Proxy Pattern is a structural design pattern that is a class functioning as an interface to something else. i.e. It plays as a wrapper or agent object that is being called by the client to access the real serving object behind the scenes.

- ❖ Là mẫu thiết kế mà tất cả các truy cập trực tiếp đến một đối tượng nào đó sẽ được chuyển hướng vào một đối tượng trung gian (Proxy)
- ❖ Nó có thể là "interface" cho bất kỳ cái gì: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.
 - Các chức năng bổ sung có thể được thêm bên trong Proxy
- ❖ Tần suất sử dụng: ★★★★ cao trung bình

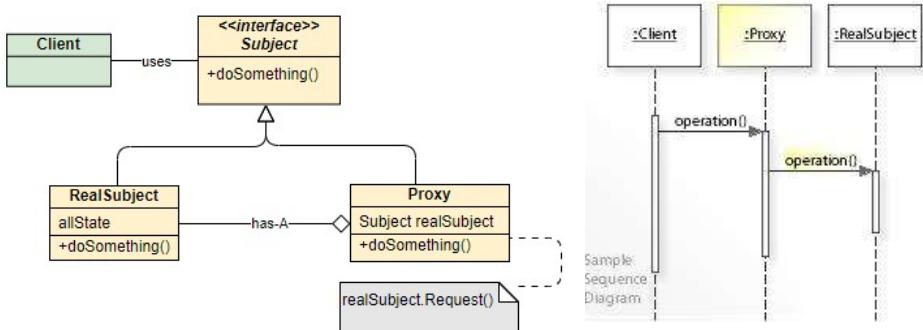
STRUCTURAL DESIGN PATTERN - PROXY

❑ Cách thực hiện

- ❖ RealSubject: là class service, là lớp cài đặt thực sự các chức năng.
 - Đây là đối tượng mà proxy sẽ đại diện.
- ❖ Client: là đối tượng cần sử dụng RealSubject (thông qua Proxy).
- ❖ Subject: là interface định nghĩa các phương thức cho phép Client thực hiện (gọi) các chức năng của RealSubject (thông qua Proxy).
- ❖ Proxy: là class
 - chứa 1 tham chiếu của RealSubject
 - sử dụng tham chiếu RealSubject này để implement các phương thức trong interface Subject để thực hiện các chức năng của RealSubject.
 - bổ sung thêm (nếu cần) các bước kiểm tra trước khi gọi hàm của RealSubject.

STRUCTURAL DESIGN PATTERN - PROXY

Cài đặt



STRUCTURAL DESIGN PATTERN - PROXY

Cài đặt minh họa 1

Đối tượng Image:

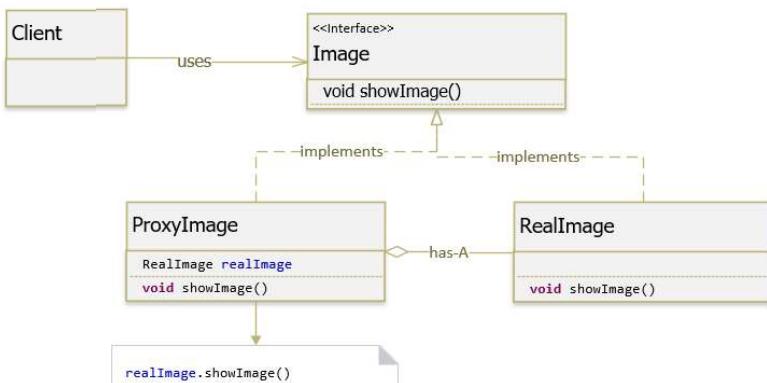
- đại diện cho 1 ảnh có dung lượng lớn
- Phương thức **showImage()** sẽ hiển thị ảnh này lên giao diện của ứng dụng.

Yêu cầu:

- Nếu đối tượng ảnh xuất hiện (được hiển thị) nhiều hơn 1 lần trên giao diện thì chỉ 1 đối tượng image được tạo ra, và đối tượng này sẽ được tái sử dụng nhiều lần.

STRUCTURAL DESIGN PATTERN - PROXY

Cài đặt minh họa 1



Cài đặt minh họa 1

```

01. public interface Image {
02.     void showImage();
03. }
04.
05. public class RealImage implements Image {
06.     private String url_img;
07.
08.     public RealImage(String url_img) {
09.         this.url_img = url_img;
10.         System.out.println("Image loaded: " + this.url_img);
11.     }
12.
13.     @Override
14.     public void showImage() {
15.         System.out.println("Image showed: " + this.url_img);
16.     }
17.
18. }
19.

21. public class ProxyImage implements Image {
22.     private Image realImage;
23.     private String url_img;
24.
25.     public ProxyImage(String url_img) {
26.         this.url_img = url_img;
27.         System.out.println("Image unloaded: " + this.url_img);
28.     }
29.
30.     @Override
31.     public void showImage() {
32.         if (realImage == null) {
33.             realImage = new RealImage(this.url_img);
34.         } else {
35.             System.out.println("Image already existed: " + this.url_img);
36.         }
37.         realImage.showImage();
38.     }
39.
40. }
41.

```

STRUCTURAL DESIGN PATTERN - PROXY

☐ Cài đặt minh họa 1

```
43. public class Client {  
44.     public static void main(String[] args) {  
45.         ProxyImage image1 = new ProxyImage("http://abc.com/a1.jpg");  
46.         ProxyImage image2 = new ProxyImage("http://abc.com/a2.jpg");  
47.  
48.         image1.showImage();  
49.         image1.showImage();  
50.         image2.showImage();  
51.         image1.showImage();  
52.         image2.showImage();  
53.     }  
54. }
```

Image unloaded: http://abc.com/a1.jpg
Image unloaded: http://abc.com/a2.jpg
Image loaded: http://abc.com/a1.jpg
Image showed: http://abc.com/a1.jpg
Image already existed: http://abc.com/a1.jpg
Image showed: http://abc.com/a1.jpg
Image loaded: http://abc.com/a2.jpg
Image showed: http://abc.com/a2.jpg
Image already existed: http://abc.com/a1.jpg
Image showed: http://abc.com/a1.jpg
Image already existed: http://abc.com/a2.jpg
Image showed: http://abc.com/a2.jpg

STRUCTURAL DESIGN PATTERN - PROXY

☐ Phân loại proxy

- ❖ **Remote Proxy:** cho phép Client truy cập vào một RealSubject từ xa (vd: ở máy khác) qua Remote Proxy
 - Vd: ứng dụng trong RMI, Webservice
- ❖ **Cache Proxy:** cho phép lưu trữ tạm RealSubject hoặc các kết quả trả về từ RealSubject, dùng để tái sử dụng khi cần.
 - Loại Proxy này hoạt động tương tự như Flyweight Pattern.
- ❖ **Synchronization Proxy:** đảm bảo nhiều client có thể truy cập vào cùng một RealSubject mà không gây ra xung đột.
 - Chú ý: nếu một client nào đó khóa RealSubject thời gian dài sẽ dẫn đến hiện tượng "tắc nghẽn" (số lượng các client trong danh sách đợi cứ tăng lên) → hoạt động của hệ thống bị ngừng trệ.
- ❖ **Firewall Proxy:** bảo vệ đối tượng RealSubject trước các yêu cầu đến từ các client không tin cậy.

STRUCTURAL DESIGN PATTERN - PROXY

☐ Phân loại proxy

- ❖ **Virtual Proxy:** trong lúc thực thi ứng dụng, proxy chỉ tạo đối tượng RealSubject khi có yêu cầu → làm tăng hiệu suất của ứng dụng.
 - (nói chung) proxy tốn ít tài nguyên để khởi tạo và duy trì hơn so với RealSubject
 - Lazy Loading: trì hoãn việc tải các đối tượng cho đến thời điểm thực sự cần dùng nó.
- ❖ **Protection Proxy:** cho phép kiểm tra quyền truy cập của client đến RealSubject
 - phạm vi (quyền) truy cập của các client khác nhau có thể khác nhau
- ❖ **Monitor Proxy:** cho phép
 - thiết lập các bảo mật bổ sung trên RealSubject,
 - ngăn không cho client truy cập một số trường quan trọng của RealSubject;
 - hoặc theo dõi, giám sát, ghi log việc truy cập, sử dụng RealSubject.

STRUCTURAL DESIGN PATTERN - PROXY

☐ Phân loại proxy

- ❖ **Copy-On-Write Proxy:** đảm bảo rằng sẽ không có client nào phải chờ vô thời hạn.
 - Copy-On-Write Proxy thường có thiết kế rất phức tạp.
- ❖ **Smart Reference Proxy:** cho phép bổ sung một số thao tác trước khi gọi tới RealSubject → tăng cường kiểm soát hoạt động truy cập RealSubject
 - Vd: cache RealSubject trong lần khởi tạo đầu tiên để tái sử dụng;
lock RealSubject trong ứng dụng Multi-thread;
etc.

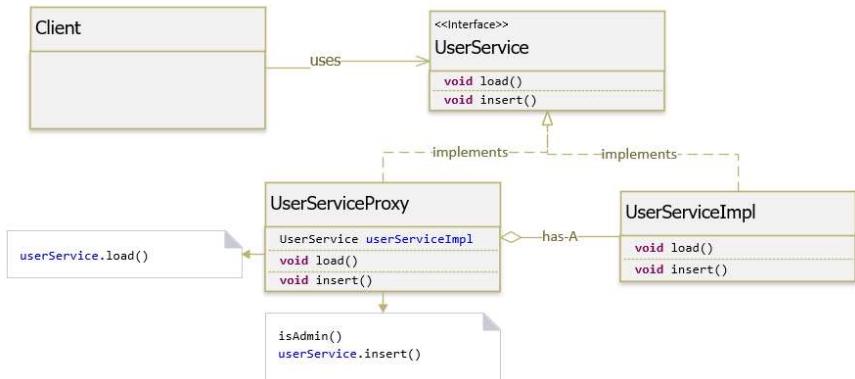
STRUCTURAL DESIGN PATTERN - PROXY

Cài đặt minh họa 2 (protection proxy)

- ❖ Một ứng dụng cung cấp 2 dịch vụ load() và insert()
 - Lập trình viên không thể tiếp cận được code của 2
- ❖ Trong đó bất kỳ Client nào cũng có thể sử dụng (gọi) được load(), nhưng chỉ có Client là admin mới sử dụng được insert()

STRUCTURAL DESIGN PATTERN - PROXY

Cài đặt minh họa 2 (protection proxy)



STRUCTURAL DESIGN PATTERN - PROXY

Cài đặt minh họa 2 (protection proxy)

```

01. public interface UserService {
02.     void load();
03.     void insert();
04. }
05.
06. public class UserServiceImpl
07.     implements UserService {
08.     private String name;
09.
10.     public UserServiceImpl(String name) {
11.         this.name = name;
12.     }
13.
14.     @Override
15.     public void load() {
16.         System.out.println(name
17.                             + " loaded");
18.     }
19.
20.     @Override
21.     public void insert() {
22.         System.out.println(name
23.                             + " inserted");
24.     }
25. }
26.
27. public class UserServiceProxy implements UserService {
28.     private String role;
29.     private UserService userService;
30.
31.     public UserServiceProxy(String name, String role) {
32.         this.role = role;
33.         userService = new UserServiceImpl(name);
34.     }
35.
36.     @Override
37.     public void load() {
38.         userService.load();
39.     }
40.
41.     @Override
42.     public void insert() {
43.         if (isAdmin()) {
44.             userService.insert();
45.         } else {
46.             throw new IllegalAccessException("Access denied");
47.         }
48.     }
49.
50.     private boolean isAdmin() {
51.         return "admin".equalsIgnoreCase(this.role);
52.     }
53. }
  
```

STRUCTURAL DESIGN PATTERN - PROXY

Cài đặt minh họa 2 (protection proxy)

```

55. public class Client {
56.     public static void main(String[] args) {
57.         UserService admin = new UserServiceProxy("quantri", "admin");
58.         admin.load();
59.         admin.insert();
60.
61.         UserService customer = new UserServiceProxy("customer", "guest");
62.         customer.load();
63.         customer.insert();
64.     }
65. }

  
```

The code shows the main method of the Client class. It creates two instances of UserServiceProxy: one for the user 'quantri' with role 'admin' and another for 'customer' with role 'guest'. It then calls the load() and insert() methods on both instances. The output of the program is:

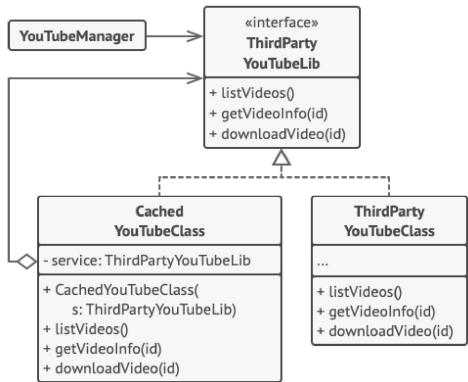
```

quantri loaded
quantri inserted
customer loaded
Exception in thread "main"
java.lang.IllegalAccessException: Access denied
at UserServiceProxy.insert(UserServiceProxy.java:46)
at Client.main(Client.java:63)
  
```

STRUCTURAL DESIGN PATTERN - PROXY

❑ Cài đặt minh họa 3:

- ❖ lazy initialization & caching



STRUCTURAL DESIGN PATTERN - PROXY

❑ Sử dụng Proxy Pattern khi nào?

- ❖ Khi muốn bảo vệ quyền truy xuất vào các phương thức của RealSubject.
- ❖ Khi cần một số thao tác bổ sung trước khi thực hiện các phương thức của RealSubject.
- ❖ Khi tạo đối tượng ban đầu là theo yêu cầu hoặc hệ thống yêu cầu sự trì hoãn khi tải một số tài nguyên nhất định (lazy loading).
- ❖ Khi có nhiều truy cập vào RealSubject có chi phí khởi tạo ban đầu lớn.
- ❖ Khi RealSubject tồn tại trong môi trường từ xa (remote).
- ❖ Khi RealSubject nằm trong một hệ thống cũ hoặc thư viện của bên thứ ba.
- ❖ Khi muốn theo dõi trạng thái và vòng đời của RealSubject.

STRUCTURAL DESIGN PATTERN - DECORATOR

❑ Yêu cầu:

- ❖ Mở rộng chức năng của 1 đối tượng

❖ Nhưng:

- Không dùng kế thừa
- Khi lớp của đối tượng được khai báo final
- Khi chương trình đã được biên dịch và thực thi

STRUCTURAL DESIGN PATTERN - DECORATOR



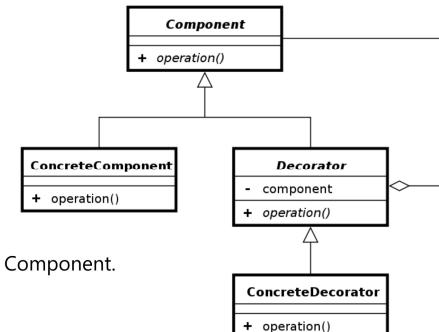
Decorator Pattern is a structural design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class.

- ❖ cho phép thêm chức năng mới vào đối tượng hiện tại mà không ảnh hưởng đến các đối tượng khác (của cùng lớp đó)
- ❖ khi cần thêm tính năng mới, đối tượng hiện có được wrap trong một đối tượng mới (decorator class).
 - Decorator pattern còn được gọi là Wrapper hay Smart Proxy.
- ❖ Tần suất sử dụng: ★★★ trung bình

STRUCTURAL DESIGN PATTERN - DECORATOR

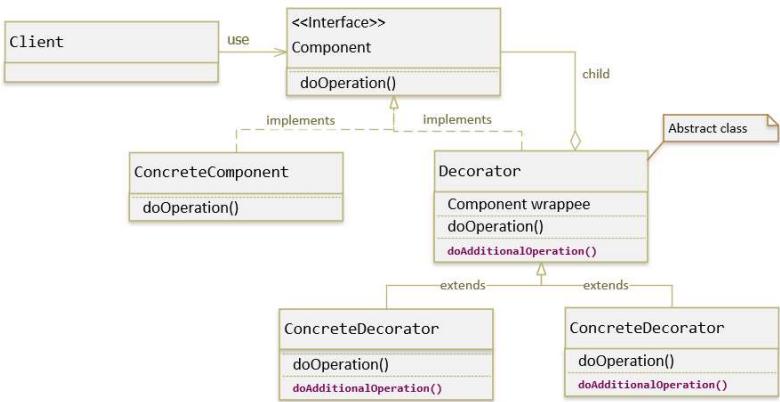
Cách thực hiện

- ❖ Component:
 - là một interface quy định các method chung cần phải có cho tất cả các thành phần tham gia vào pattern này.
- ❖ ConcreteComponent:
 - là lớp implements các phương thức của interface Component.
- ❖ Decorator:
 - là một lớp abstract có chứa một tham chiếu của đối tượng Component, và implement các phương thức của interface Component.
- ❖ ConcreteDecorator:
 - là lớp kế thừa lớp abstract Decorator, đồng thời cài đặt thêm các tính năng mới cho Component.
- ❖ Client: đối tượng sử dụng Component.



STRUCTURAL DESIGN PATTERN - DECORATOR

Cài đặt



STRUCTURAL DESIGN PATTERN - DECORATOR

Cài đặt minh họa

- ❖ Trong 1 dự án (phát triển ứng dụng di động Android), các nhân viên (**employee**) có thể làm việc với các vai trò khác nhau:
 - Thành viên nhóm (**team member**): chịu trách nhiệm thực hiện các nhiệm vụ được giao và phối hợp với các thành viên khác để hoàn thành nhiệm vụ nhóm.
 - Trưởng nhóm (**team leader**): quản lý và phối hợp với các thành viên trong nhóm của mình, lập kế hoạch nhiệm vụ cho họ.
 - Quản lý (**manager**): có trách nhiệm đối với trưởng nhóm như quản lý yêu cầu dự án, tiến độ, phân công công việc.

STRUCTURAL DESIGN PATTERN - DECORATOR

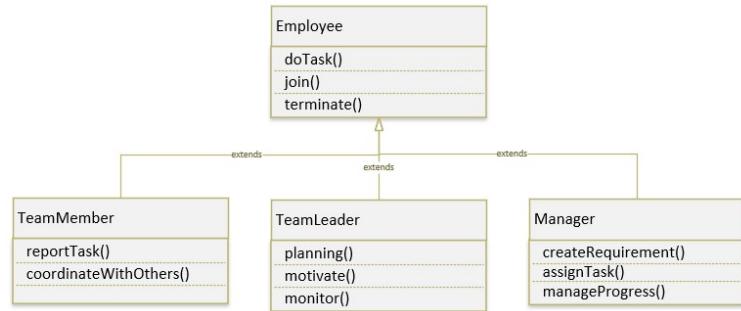
☐ Cài đặt minh họa

❖ Phân tích “behavior” của các đối tượng trên:

- **Employee:**
 - ✓ tham gia vào dự án (**join**),
 - ✓ thực hiện công việc (**doTask**),
 - ✓ rời khỏi dự án (**terminate**).
- **Team member:**
 - ✓ báo cáo task được giao (**reportTask**),
 - ✓ cộng tác với các thành viên khác (**coordinateWithOthers**).
- **Team leader:**
 - ✓ lên kế hoạch (**planning**),
 - ✓ hỗ trợ các thành viên phát triển (**motivate**),
 - ✓ theo dõi chất lượng công việc và thời gian (**monitor**).
- **Manager:**
 - ✓ tạo các yêu cầu dự án (**createRequirement**),
 - ✓ giao nhiệm vụ cho thành viên (**assignTask**),
 - ✓ quản lý tiến độ dự án (**progressManagement**).

STRUCTURAL DESIGN PATTERN - DECORATOR

☐ Cài đặt minh họa



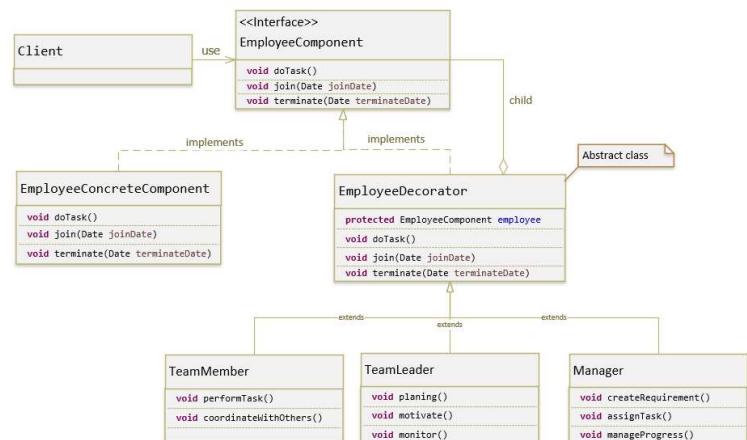
STRUCTURAL DESIGN PATTERN - DECORATOR

☐ Cài đặt minh họa

- ❖ Vì phát sinh thêm phiên bản dành cho iOS (bên cạnh phiên bản cho Android) trong quá trình thực hiện dự án, nên 1 TeamMember được đẩy lên làm TeamLeader
 - Tạo đối tượng TeamLeader mới thế nào? Đối tượng TeamMember cũ xử lý thế nào?
- ❖ Vì quản lí dự án nhảy việc trong quá trình thực hiện dự án, nên 1 TeamLeader được đẩy lên làm Manager
 - Tạo đối tượng Manager mới thế nào? Đối tượng TeamLeader cũ xử lý thế nào?
- ❖ Vì đội code thiếu người, nên anh TeamLeader phải kiêm luôn TeamMember
(Vì đội iOS thiếu người, nên anh Manager phải kiêm luôn TeamLeader)
 - Cần tạo 2 đối tượng cho cùng 1 employee (?)

STRUCTURAL DESIGN PATTERN - DECORATOR

☐ Cài đặt minh họa



STRUCTURAL DESIGN PATTERN - DECORATOR

☐ Cài đặt minh họa

```
01. public interface EmployeeComponent {  
02.     String getName();  
03.     void doTask();  
04.     void join(Date joinDate);  
05.     void terminate(Date terminateDate);  
06.  
07.     default String formatDate(Date theDate) {  
08.         DateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");  
09.         return sdf.format(theDate);  
10.    }  
11.  
12.    default void showBasicInformation() {  
13.        System.out.println("-----");  
14.        System.out.println("The basic information of "+getName());  
15.        join(Calendar.getInstance().getTime());  
16.        Calendar terminateDate = Calendar.getInstance();  
17.        terminateDate.add(Calendar.MONTH, 6);  
18.        terminate(terminateDate.getTime());  
19.    }  
20. }
```

STRUCTURAL DESIGN PATTERN - DECORATOR

☐ Cài đặt minh họa

```
22. public class EmployeeConcreteComponent implements EmployeeComponent {  
23.     private String name;  
24.  
25.     public EmployeeConcreteComponent (String name) {  
26.         this.name = name;  
27.     }  
28.  
29.     public String getName() {  
30.         return name;  
31.     }  
32.  
33.     public void join(Date joinDate) {  
34.         System.out.println(this.getName()  
35.             + " joined on " + formatDate(joinDate));  
36.     }  
37.  
38.     public void terminate(Date terminateDate) {  
39.         System.out.println(this.getName()  
40.             + " terminated on " + formatDate(terminateDate));  
41.     }  
42.  
43.     public void doTask() {  
44.         // do something  
45.     }  
46.  
47. }
```

```
49. public abstract class EmployeeDecorator implements EmployeeComponent {  
50.     protected EmployeeComponent employee;  
51.  
52.     protected EmployeeDecorator (EmployeeComponent employee) {  
53.         this.employee = employee;  
54.     }  
55.  
56.     public String getName() {  
57.         return employee.getName();  
58.     }  
59.  
60.     public void join(Date joinDate) {  
61.         employee.join(joinDate);  
62.     }  
63.  
64.     public void terminate(Date terminateDate) {  
65.         employee.terminate(terminateDate);  
66.     }  
67.  
68. }
```

STRUCTURAL DESIGN PATTERN - DECORATOR

☐ Cài đặt minh họa

```
71. public class TeamMember extends EmployeeDecorator {  
72.     protected TeamMember(EmployeeComponent employee) {  
73.         super(employee);  
74.     }  
75.  
76.     public void reportTask() {  
77.         System.out.println(this.employee.getName()  
78.             + " is reporting his assigned tasks.");  
79.     }  
80.  
81.     public void coordinateWithOthers() {  
82.         System.out.println(this.employee.getName()  
83.             + " is coordinating with other members.");  
84.     }  
85.  
86.     @Override  
87.     public void doTask() {  
88.         employee.doTask();  
89.         reportTask();  
90.         coordinateWithOthers();  
91.     }  
92. }
```

STRUCTURAL DESIGN PATTERN - DECORATOR

☐ Cài đặt minh họa

```
94. public class TeamLeader extends EmployeeDecorator {  
95.     protected TeamLeader(EmployeeComponent employee) {  
96.         super(employee);  
97.     }  
98.  
99.     public void planing() {  
100.        System.out.println(this.employee.getName()  
101.            + " is planing.");  
102.    }  
103.  
104.    public void motivate() {  
105.        System.out.println(this.employee.getName()  
106.            + " is motivating his members.");  
107.    }  
108.  
109.    public void monitor() {  
110.        System.out.println(this.employee.getName()  
111.            + " is monitoring his members.");  
112.    }  
113.  
114.    @Override  
115.    public void doTask() {  
116.        employee.doTask();  
117.        planing();  
118.        motivate();  
119.        monitor();  
120.    }  
121. }
```

```
123. public class Manager extends EmployeeDecorator {  
124.     protected Manager(EmployeeComponent employee) {  
125.         super(employee);  
126.     }  
127.  
128.     public void createRequirement() {  
129.         System.out.println(this.employee.getName()  
130.             + " is create requirements.");  
131.     }  
132.  
133.     public void assignTask() {  
134.         System.out.println(this.employee.getName()  
135.             + " is assigning tasks.");  
136.     }  
137.  
138.     public void manageProgress() {  
139.         System.out.println(this.employee.getName()  
140.             + " is managing the progress.");  
141.     }  
142.  
143.     @Override  
144.     public void doTask() {  
145.         employee.doTask();  
146.         createRequirement();  
147.         assignTask();  
148.         manageProgress();  
149.     }  
150. }
```

STRUCTURAL DESIGN PATTERN - DECORATOR

☐ Cài đặt minh họa

```
152. public class Client {
153.     public static void main(String[] args) {
154.         System.out.println("NORMAL EMPLOYEE: ");
155.         EmployeeComponent employee = new
156.             EmployeeConcreteComponent("Ronando");
157.         employee.showBasicInformation();
158.         employee.doTask();
159.
160.         System.out.println("\nTEAM LEADER: ");
161.         EmployeeComponent teamLeader = new TeamLeader(employee);
162.         teamLeader.showBasicInformation();
163.         teamLeader.doTask();
164.
165.         System.out.println("\nMANAGER: ");
166.         EmployeeComponent manager = new Manager(employee);
167.         manager.showBasicInformation();
168.         manager.doTask();
169.
170.         System.out.println("\nTEAM LEADER AND MANAGER: ");
171.         EmployeeComponent teamLeaderAndManager = new Manager(teamLeader);
172.         teamLeaderAndManager.showBasicInformation();
173.         teamLeaderAndManager.doTask();
174.     }
175. }
```

STRUCTURAL DESIGN PATTERN - DECORATOR

☐ Cài đặt minh họa

NORMAL EMPLOYEE:

The basic information of Ronando
Ronando joined on 04/11/2018
Ronando terminated on 04/05/2019

TEAM LEADER:

The basic information of Ronando
Ronando joined on 04/11/2018
Ronando terminated on 04/05/2019
Ronando is planing.
Ronando is motivating his members.
Ronando is monitoring his members.

MANAGER:

The basic information of Ronando
Ronando joined on 04/11/2018
Ronando terminated on 04/05/2019
Ronando is create requirements.
Ronando is assigning tasks.
Ronando is managing the progress.

TEAM LEADER AND MANAGER:

The basic information of Ronando
Ronando joined on 04/11/2018
Ronando terminated on 04/05/2019
Ronando is planing.
Ronando is motivating his members.
Ronando is monitoring his members.
Ronando is create requirements.
Ronando is assigning tasks.
Ronando is managing the progress.

STRUCTURAL DESIGN PATTERN - DECORATOR

☐ Lợi ích của Decorator Pattern

- ❖ Tăng cường khả năng mở rộng của đối tượng, vì những thay đổi được thực hiện bằng cách thực hiện trên các lớp mới.
- ❖ Client sẽ không nhận thấy sự khác biệt khi bạn truyền vào 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).

STRUCTURAL DESIGN PATTERN - DECORATOR

☐ Sử dụng Decorator Pattern khi:

- ❖ Muốn thêm tính năng mới cho các đối tượng mà không muốn ảnh hưởng đến các đối tượng khác của cùng lớp này.
- ❖ Khi không thể mở rộng chức năng của một đối tượng bằng cách kế thừa
 - Vd: một class sử dụng từ khóa final
→ muốn mở rộng đối với class này thì phải sử dụng decorator (cách duy nhất).
- ❖ Trong một số trường hợp mà việc sử dụng kế thừa sẽ mất nhiều công sức.

STRUCTURAL DESIGN PATTERN - SO SÁNH DECORATOR VÀ ADAPTER

❑ Giống:

- ❖ đều là structural pattern
- ❖ đều sử dụng composition để cài đặt

❑ Khác:

- ❖ Decorator cho phép thêm một tính năng mới vào một object nhưng không sử dụng thừa kế. Nó cho phép thay đổi lúc thực thi (run-time)
- ❖ Decorator có xu hướng hoạt động trên một đối tượng.

- ❖ Adapter được sử dụng khi ta có một interface, và muốn ánh xạ interface đó đến một đối tượng khác có vai trò chức năng tương tự, nhưng nó có một interface khác.
- ❖ Adapter có xu hướng hoạt động trên nhiều đối tượng (có thể wrap nhiều interface)

STRUCTURAL DESIGN PATTERN - SO SÁNH DECORATOR VÀ PROXY

❑ Giống:

- ❖ đều là structural pattern
- ❖ có cấu trúc tương tự nhau (đều Wrap một đối tượng thực bên trong nó)

❑ Khác (ở mục đích sử dụng):

- ❖ Decorator sử dụng để có thể thêm tính năng động vào một đối tượng có trước.
- ❖ Proxy cho phép ta tạo ra một đại diện cho một đối tượng khác.

STRUCTURAL DESIGN PATTERN - BRIDGE

❑ Tình huống

- ❖ Một thành phần trong OOP thường có 2 phần:
 - Phần trừu tượng (abstraction) định nghĩa các chức năng
 - Phần thực thi (implementation) cài đặt cụ thể các chức năng được định nghĩa trong phần trừu tượng.
- ❖ Hai phần này thường liên hệ với nhau thông qua quan hệ kế thừa.
- ❖ → Những thay đổi trong phần trừu tượng dẫn đến các thay đổi trong phần thực thi.

STRUCTURAL DESIGN PATTERN - BRIDGE



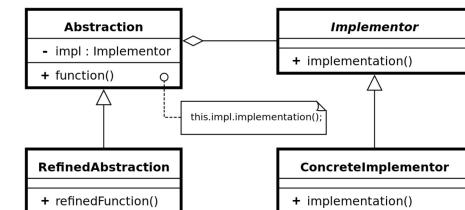
Bridge Pattern is a structural design pattern that is meant to "decouple an abstraction from its implementation so that the two can vary independently".

- ❖ dùng để tách thành phần trừu tượng (abstraction) và thành phần thực thi (implementation) riêng biệt.
 - các thành phần này có thể thay đổi một cách độc lập mà không ảnh hưởng đến các thành phần khác.
- ❖ hai thành phần này sẽ liên hệ với nhau thông qua quan hệ "chứa trong" (object composition), thay vì bằng quan hệ kế thừa.
- ❖ Tần suất sử dụng: ★★★ trung bình

STRUCTURAL DESIGN PATTERN - BRIDGE

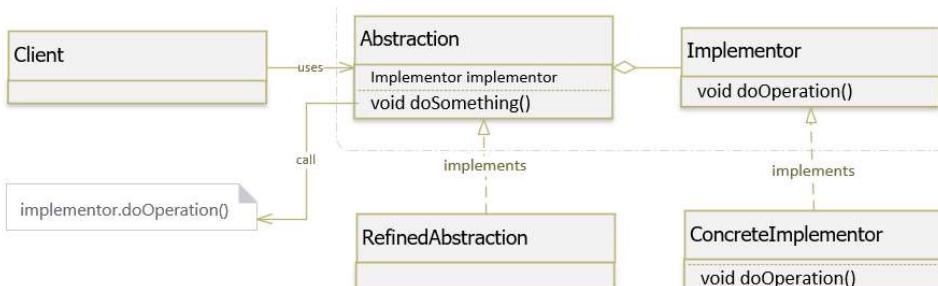
Cách thực hiện

- ❖ Abstraction:
 - là một lớp abstract quản lý tham chiếu của Implementor.
- ❖ RefinedAbstraction:
 - Là một lớp (normal class) implement các phương thức trong interface Abstraction bằng cách dùng tham chiếu của đối tượng Implementor.
- ❖ Implementor:
 - Là interface của "phần cài đặt". Thông thường nó là interface định ra các tác vụ nào đó của Abstraction.
- ❖ ConcreteImplementor:
 - Là một lớp (normal class) implement các phương thức của interface Implementor.
- ❖ Client:
 - là lớp sử dụng các chức năng thông qua interface Abstraction.



STRUCTURAL DESIGN PATTERN - BRIDGE

Cài đặt



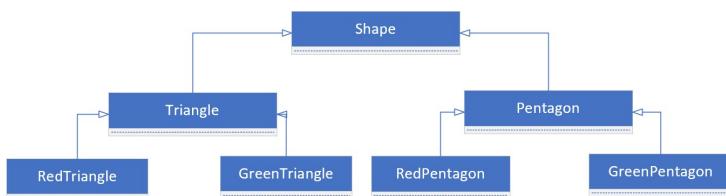
STRUCTURAL DESIGN PATTERN - BRIDGE

Cài đặt minh họa 1

- ❖ Có một class/interface **Shape** và 2 subclass **Triangle** và **Pentagon**
- ❖ Sau đó phát sinh thêm nhu cầu tô màu cho các hình trên: Red và Green
 - Tô 2 màu cho 2 loại hình thì sẽ thu được 4 đối tượng mới: RedTriangle, RedPentagon, GreenTriangle, GreenPentagon.
- ❖ Cách tổ chức thông thường?
 - ❖ Trường hợp phát sinh thêm loại hình khác (vd: Rectangle)?
 - ❖ Trường hợp phát sinh thêm màu khác (vd: blue)?

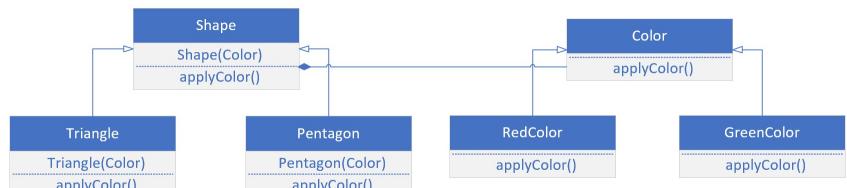
STRUCTURAL DESIGN PATTERN - BRIDGE

❑ Cài đặt minh họa 1



STRUCTURAL DESIGN PATTERN - BRIDGE

❑ Cài đặt minh họa 1



STRUCTURAL DESIGN PATTERN - BRIDGE

❑ Cài đặt minh họa 1

```

01. public interface Color {
02.     public void applyColor();
03. }
04.
05. public class RedColor implements Color{
06.     public void applyColor(){
07.         System.out.println("red.");
08.     }
09. }
10.
11. public class GreenColor implements Color{
12.     public void applyColor(){
13.         System.out.println("green.");
14.     }
15. }
  
```

STRUCTURAL DESIGN PATTERN - BRIDGE

❑ Cài đặt minh họa 1

```

17. public abstract class Shape {
18.     protected Color color;
19.
20.     public Shape(Color c){
21.         this.color=c;
22.     }
23.
24.     abstract public void applyColor();
25. }

27. public class Triangle extends Shape{
28.     public Triangle(Color c) {
29.         super(c);
30.     }
31.
32.     @Override
33.     public void applyColor() {
34.         System.out.print("Triangle with ");
35.         color.applyColor();
36.     }
37. }

39. public class Pentagon extends Shape{
40.     public Pentagon(Color c) {
41.         super(c);
42.     }
43.
44.     @Override
45.     public void applyColor() {
46.         System.out.print("Pentagon with ");
47.         color.applyColor();
48.     }
49. }
  
```

STRUCTURAL DESIGN PATTERN - BRIDGE

❑ Cài đặt minh họa 1

```
51. public class BridgePatternTest {  
52.     public static void main(String[] args) {  
53.         Shape tri = new Triangle(new RedColor());  
54.         tri.applyColor();  
55.  
56.         Shape pent = new Pentagon(new GreenColor());  
57.         pent.applyColor();  
58.     }  
59. }
```

Triangle with red.
Pentagon with green.

STRUCTURAL DESIGN PATTERN - BRIDGE

❑ Cài đặt minh họa 2

- ❖ Ta có 3 chương trình xem ảnh tương ứng với 3 loại file: JPG, PNG và GIF
 - showImage();
- ❖ Trên mỗi hệ điều hành khác nhau (vd: 3 hệ điều hành là Windows, Linux, MacOS) lại cần các chương trình khác nhau để xem 3 loại file trên.
 - Khi chạy trên Windows thì:
 - ✓ hiển thị giao diện của windows (vd: in ra dòng "hiển thị cửa sổ giao diện windows")
 - ✓ showImage()
 - Tương tự với Linux và MacOS
- ❖ Dùng Brigde pattern để cài đặt bài toán trên.

STRUCTURAL DESIGN PATTERN - BRIDGE

❑ Cài đặt minh họa 3

- ❖ Xem lại code cài đặt của Framework Ta có 3 chương trình xem ảnh tương ứng với 3 loại file: JPG, PNG và GIF
 - showImage();
- ❖ Trên mỗi hệ điều hành khác nhau (vd: 3 hệ điều hành là Windows, Linux, MacOS) lại cần các chương trình khác nhau để xem 3 loại file trên.
 - Khi chạy trên Windows thì:
 - ✓ hiển thị giao diện của windows (vd: in ra dòng "hiển thị cửa sổ giao diện windows")
 - ✓ showImage()
 - Tương tự với Linux và MacOS
- ❖ Dùng Brigde pattern để cài đặt bài toán trên.

STRUCTURAL DESIGN PATTERN - BRIDGE

❑ Lợi ích của Bridge Pattern

- ❖ Giảm sự phức tạp giữa abstraction và implementation (loose coupling)
 - cho phép chọn implementation phù hợp lúc runtime.
- ❖ Giảm số lượng những lớp con không cần thiết → code sẽ gọn gàng hơn và kích thước ứng dụng sẽ nhỏ hơn.
- ❖ Dễ bảo trì hơn: các Abstraction và Implementation sẽ dễ dàng thay đổi lúc runtime cũng như có thể được thay đổi bớt trong tương lai.
- ❖ Dễ dàng mở rộng về sau
- ❖ Cho phép ẩn các chi tiết implement từ client: do abstraction và implementation hoàn toàn độc lập nên chúng ta có thể thay đổi một thành phần mà không ảnh hưởng đến phía Client.

STRUCTURAL DESIGN PATTERN - BRIDGE

❑ 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.
- ❖ Một Implementation được chia sẻ giữa nhiều Object
- ❖ Có sự “bung nở” các lớp khi bổ sung các Implementation cho các Abstraction

STRUCTURAL DESIGN PATTERN - SO SÁNH BRIDGE VÀ ADAPTER

❑ Giống:

- ❖ đều là structural pattern
- ❖ đều “nhờ” một lớp khác để thực hiện một số xử lý nào đó

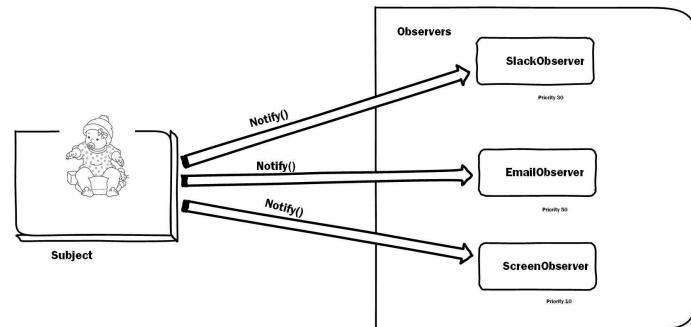
❑ Khác:

- ❖ Bridge dùng để tách thành phần trừu tượng (abstraction) và thành phần thực thi (implementation) riêng biệt.
- ❖ Bridge được dùng trong khi thiết kế hệ thống (trước khi phát triển) để Abstraction và Implementation có thể thực hiện một cách độc lập.
- ❖ Adapter dùng để biến đổi một class/interface sang một dạng khác có thể sử dụng được
 - giúp các lớp không tương thích hoạt động cùng nhau mà bình thường là không thể
- ❖ Adapter làm cho mọi thứ có thể hoạt động với nhau sau khi chúng đã được thiết kế (đã tồn tại).

BEHAVIORAL DESIGN PATTERN - OBSERVER

❑ Tình huống

- ❖ Khi có một sự kiện xảy ra trong hệ thống thì cần gửi thông báo đến nhiều thành phần khác trong hệ thống



BEHAVIORAL DESIGN PATTERN - OBSERVER

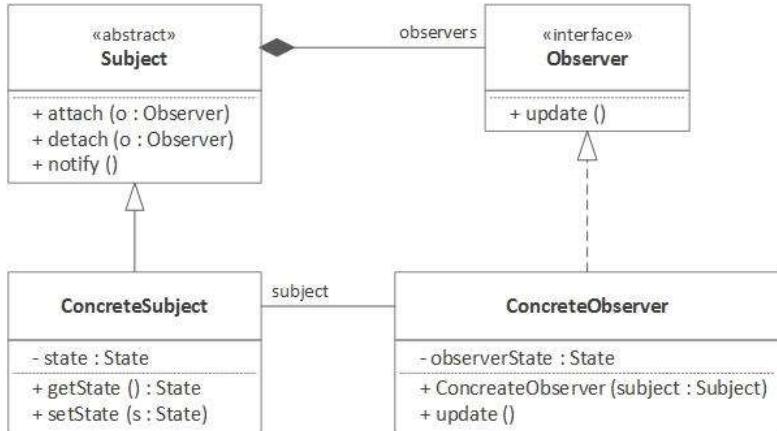


Observer Pattern is a behavioral design pattern that define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- ❖ dùng để gửi thông báo và thực hiện cập nhật một cách tự động tất cả các thành phần phụ thuộc của một đối tượng khi đối tượng đó có sự thay đổi trạng thái.
- ❖ còn gọi là Dependents, Publish/Subscribe hoặc Source/Listener.
- ❖ Tần suất sử dụng: ★★★★☆ cao

BEHAVIORAL DESIGN PATTERN - OBSERVER

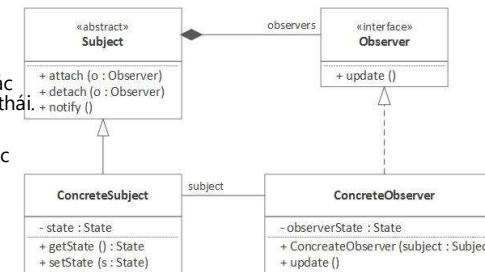
Cài đặt



BEHAVIORAL DESIGN PATTERN - OBSERVER

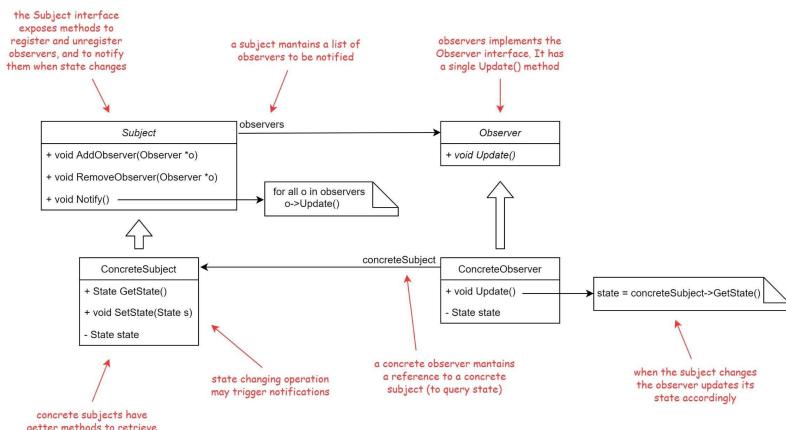
Cách thực hiện

- ❖ Subject:
 - là một lớp abstract (hoặc interface) chứa danh sách các observer, và các phương thức để có thể thêm và loại bỏ observer.
- ❖ Observer:
 - là một interface (hoặc lớp abstract), chứa một tham chiếu đến đối tượng Subject, và định nghĩa một phương thức update() cho các đối tượng sẽ được subject thông báo đến khi có sự thay đổi trạng thái.
- ❖ ConcreteSubject:
 - cài đặt các phương thức của Subject, lưu trữ trạng thái danh sách các ConcreteObserver, gửi thông báo đến các observer của nó khi có sự thay đổi trạng thái.
- ❖ ConcreteObserver:
 - là một lớp kế thừa từ Observer, cài đặt các phương thức của Observer, lưu trữ trạng thái của subject, thực thi việc cập nhật để giữ cho trạng thái của nó đồng nhất với thông báo do subject gửi đến.



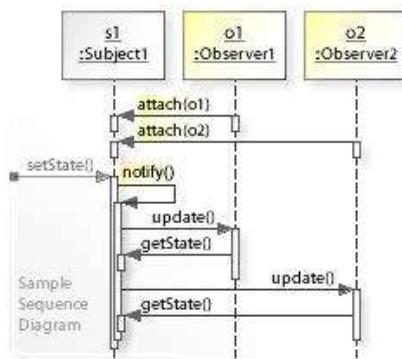
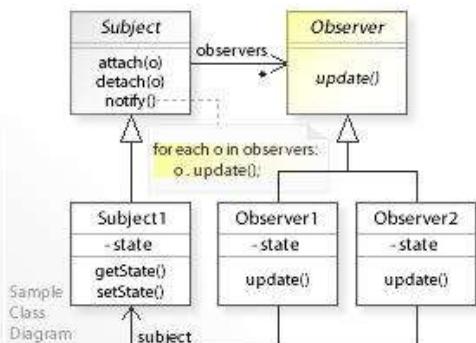
BEHAVIORAL DESIGN PATTERN - OBSERVER

Cài đặt



BEHAVIORAL DESIGN PATTERN - OBSERVER

Cài đặt



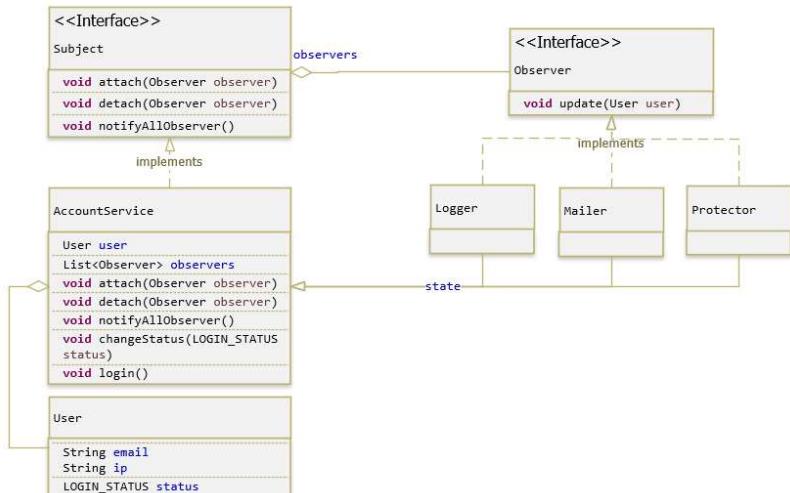
BEHAVIORAL DESIGN PATTERN - OBSERVER

Cài đặt minh họa

Hệ thống theo dõi tài khoản của người dùng Chat (Account)

- mọi thao tác của người dùng đều cần được ghi lại (**Logger**)
(vd: **login**, **changeStatus**)
- thực hiện gửi mail (**Mailer**) thông báo khi tài khoản hết hạn (**expired**)
- thực hiện chặn người dùng (**Protector**) nếu truy cập không hợp lệ (**invalid**)

BEHAVIORAL DESIGN PATTERN - OBSERVER



BEHAVIORAL DESIGN PATTERN - OBSERVER

Cài đặt minh họa

```

01. enum LoginStatus {
02.     SUCCESS, FAILURE, INVALID, EXPIRED
03. }
04.
05. @Data
06. class User {
07.     private String email;
08.     private String ip;
09.     private LoginStatus status;
10. }
11.
12. public interface Subject {
13.     void attach(Observer observer);
14.     void detach(Observer observer);
15.     void notifyAllObserver();
16. }
17.
18. public interface Observer {
19.     void update(User user);
20. }
21.
22. public class Mailer implements Observer {
23.     @Override
24.     public void update(User user) {
25.         if (user.getStatus() == LoginStatus.EXPIRED) {
26.             System.out.println("Mailer: User "
27.                 + user.getEmail()
28.                 + " is expired. An email was sent!");
29.         }
30.     }
31. }
32.
33. public class Protector implements Observer {
34.     @Override
35.     public void update(User user) {
36.         if (user.getStatus() == LoginStatus.INVALID) {
37.             System.out.println("Protector: User "
38.                 + user.getEmail() + " is invalid. "
39.                 + "IP " + user.getIp() + " is blocked");
40.         }
41.     }
42. }
43.
44. public class Logger implements Observer {
45.     @Override
46.     public void update(User user) {
47.         System.out.println("Logger: " + user);
48.     }
49. }
  
```

BEHAVIORAL DESIGN PATTERN - OBSERVER

❑ Cài đặt minh họa

```
51. public class AccountService implements Subject {  
52.     private User user;  
53.     private List<Observer> observers = new ArrayList<>();  
54.  
55.     public AccountService(String email, String ip) {  
56.         user = new User();  
57.         user.setEmail(email);  
58.         user.setIp(ip);  
59.     }  
60.  
61.     public void attach(Observer observer) {  
62.         if (!observers.contains(observer))  
63.             observers.add(observer);  
64.     }  
65.  
66.     public void detach(Observer observer) {  
67.         if (observers.contains(observer)) {  
68.             observers.remove(observer);  
69.         }  
70.     }  
71.  
72.     public void notifyAllObserver() {  
73.         for (Observer observer : observers) {  
74.             observer.update(user);  
75.         }  
76.     }  
77.  
78.     public void changeStatus(LoginStatus status) {  
79.         user.setStatus(status);  
80.         System.out.println("Status is changed");  
81.         this.notifyAllObserver();  
82.     }  
83.  
84.     public void login() {  
85.         if (!this.isValidIP()) {  
86.             user.setStatus(LoginStatus.INVALID);  
87.         } else if (this.isValidEmail()) {  
88.             user.setStatus(LoginStatus.SUCCESS);  
89.         } else {  
90.             user.setStatus(LoginStatus.FAILURE);  
91.         }  
92.         System.out.println("Login is handled");  
93.         this.notifyAllObserver();  
94.     }  
95.  
96.     private boolean isValidIP() {  
97.         return "127.0.0.1".equals(user.getIp());  
98.     }  
99.  
100.    private boolean isValidEmail() {  
101.        return "spam@mail.com"  
102.            .equalsIgnoreCase(user.getEmail());  
103.    }  
104. }
```

BEHAVIORAL DESIGN PATTERN - OBSERVER

❑ Cài đặt minh họa

```
106. public class ObserverPatternExample {  
107.     public static void main(String[] args) {  
108.         AccountService account1 = createAccount("spam@mail.com", "127.0.0.1");  
109.         account1.login();  
110.         account1.changeStatus(LoginStatus.EXPIRED);  
111.  
112.         System.out.println("---");  
113.         AccountService account2 = createAccount("spam@mail.com", "116.108.77.231");  
114.         account2.login();  
115.     }  
116.  
117.     private static AccountService createAccount(String email, String ip) {  
118.         AccountService account = new AccountService(email, ip);  
119.         account.attach(new Logger());  
120.         account.attach(new Mailer());  
121.         account.attach(new Protector());  
122.         return account;  
123.     }  
124. }
```

BEHAVIORAL DESIGN PATTERN - OBSERVER

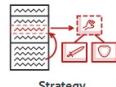
❑ Cài đặt minh họa

```
126. Login is handled  
127. Logger: User(email=spam@mail.com, ip=127.0.0.1, status=SUCCESS)  
128. Status is changed  
129. Logger: User(email=spam@mail.com, ip=127.0.0.1, status=EXPIRED)  
130. Mailer: User spam@mail.com is expired. An email was sent!  
131. ---  
132. Login is handled  
133. Logger: User(email=spam@mail.com, ip=116.108.77.231, status=INVALID)  
134. Protector: User spam@mail.com is invalid. IP 116.108.77.231 is blocked
```

BEHAVIORAL DESIGN PATTERN - STRATEGY

❑ Tình huống

- ❖ Cần cài đặt nhiều hình thức thanh toán cho chức năng thanh toán
- ❖ Cần cài đặt nhiều thuật toán sắp xếp khác nhau cho chức năng sắp xếp
- ❖ Cho phép nhiều hình thức xác thực cho chức năng đăng nhập



Strategy Pattern is a behavioral design pattern that Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

- ❖ dùng để tách rời phần xử lý một chức năng cụ thể ra khỏi đối tượng.
- ❖ tạo ra một tập hợp các thuật toán để xử lý chức năng đó
- ❖ cho phép lựa chọn thuật toán khi thực thi chương trình.
→ Mẫu thiết kế này thường được sử dụng để thay thế cho sự kế thừa, khi muốn chấm dứt việc theo dõi và chỉnh sửa một chức năng qua nhiều lớp con.
- ❖ Tần suất sử dụng: ★★★★ cao trung bình

BEHAVIORAL DESIGN PATTERN - STRATEGY

❑ Cách thực hiện

❖ Strategy:

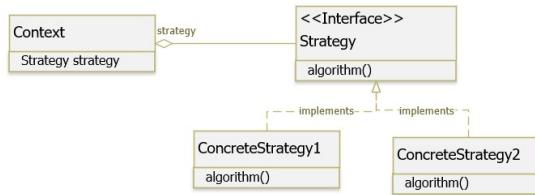
- định nghĩa các hành vi có thể có của một Strategy.

❖ ConcreteStrategy:

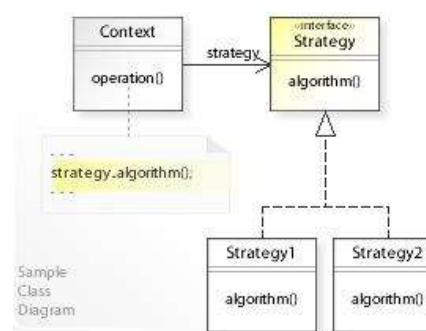
- cài đặt các hành vi cụ thể của Strategy.

❖ Context:

- chứa một tham chiếu đến đối tượng Strategy và nhận các yêu cầu từ Client, các yêu cầu này sau đó được ủy quyền cho Strategy thực hiện.

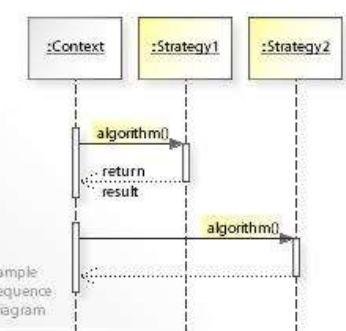


❑ Cài đặt



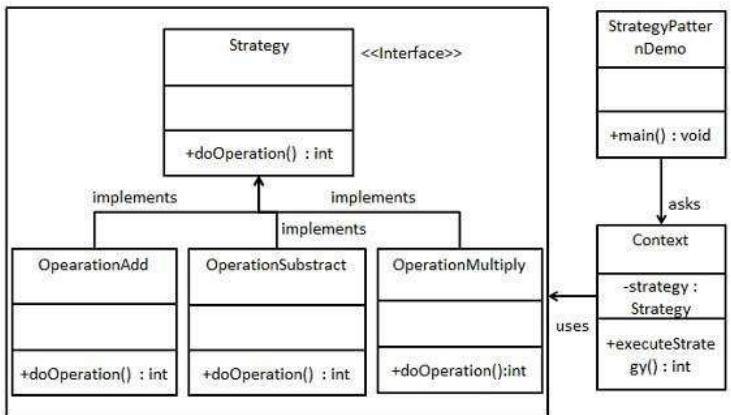
BEHAVIORAL DESIGN PATTERN - STRATEGY

❑ Cài đặt



BEHAVIORAL DESIGN PATTERN - STRATEGY

❑ Ví dụ minh họa



BEHAVIORAL DESIGN PATTERN - STRATEGY

❑ Ví dụ minh họa

```

01. public interface Strategy {
02.     public int doOperation(int num1, int num2);
03. }
04.
05. public class OperationAdd implements Strategy{
06.     @Override
07.     public int doOperation(int num1, int num2) {
08.         return num1 + num2;
09.     }
10.
11. public class OperationSubtract implements Strategy{
12.     @Override
13.     public int doOperation(int num1, int num2) {
14.         return num1 - num2;
15.     }
16.
17. }
18.
19. public class OperationMultiply implements Strategy{
20.     @Override
21.     public int doOperation(int num1, int num2) {
22.         return num1 * num2;
23.     }
24.
25. public class Context {
26.     private Strategy strategy;
27.
28.     public Context(Strategy strategy){
29.         this.strategy = strategy;
30.     }
31.
32.     public int executeStrategy(int num1, int num2){
33.         return strategy.doOperation(num1, num2);
34.     }
35.
36. }
37.
38.
39.
40.
41.
42.
43.
44.
45.
46.
47.
48.
49.
  
```

BEHAVIORAL DESIGN PATTERN - STRATEGY

❑ Ví dụ minh họa

```

38. public class StrategyPatternDemo {
39.     public static void main(String[] args) {
40.         Context context = new Context(new OperationAdd());
41.         System.out.println("10 + 5 = " + context.executeStrategy(10, 5));
42.
43.         context = new Context(new OperationSubtract());
44.         System.out.println("10 - 5 = " + context.executeStrategy(10, 5));
45.
46.         context = new Context(new OperationMultiply());
47.         System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
48.
49.     }
  
```

```

10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
  
```

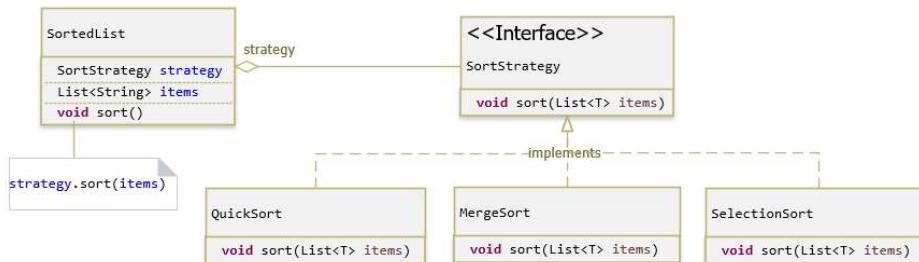
BEHAVIORAL DESIGN PATTERN - STRATEGY

❑ Ví dụ minh họa 2

- ❖ Cài đặt module thực hiện chức năng sắp xếp bằng nhiều thuật toán sắp xếp khác nhau (vd: Quick Sort, Merge sort, Selection sort,...)

BEHAVIORAL DESIGN PATTERN - STRATEGY

❑ Ví dụ minh họa 2



BEHAVIORAL DESIGN PATTERN - STRATEGY

❑ Ví dụ minh họa

```

01. public interface SortStrategy {
02.     public <T> void sort(List<T> items);
03. }
04.
05. public class QuickSort implements SortStrategy {
06.     @Override
07.     public <T> void sort(List<T> items) {
08.         System.out.println("Quick sort");
09.     }
10.
11. public class MergeSort implements SortStrategy {
12.     @Override
13.     public <T> void sort(List<T> items) {
14.         System.out.println("Merge sort");
15.     }
16.
17. public class SelectionSort implements SortStrategy {
18.     @Override
19.     public <T> void sort(List<T> items) {
20.         System.out.println("Selection sort");
21.     }
22.
23.
24.
25.
26. public class Sortedlist {
27.     private SortStrategy strategy;
28.     private List<String> items = new ArrayList<>();
29.
30.     public void setSortStrategy(SortStrategy strategy) {
31.         this.strategy = strategy;
32.     }
33.
34.     public void add(String name) {
35.         items.add(name);
36.     }
37.
38.     public void sort() {
39.         strategy.sort(items);
40.     }
41. }
  
```

BEHAVIORAL DESIGN PATTERN - STRATEGY

❑ Ví dụ minh họa

```

43. public class StrategyPatternExample {
44.     public static void main(String[] args) {
45.         SortedList sortedList = new SortedList();
46.         sortedList.add("Mot");
47.         sortedList.add("Hai");
48.         sortedList.add("Ba");
49.
50.         sortedList.setSortStrategy(new QuickSort());
51.         sortedList.sort();
52.
53.         sortedList.setSortStrategy(new MergeSort());
54.         sortedList.sort();
55.
56.     }
  
```

Quick sort
Merge sort

BEHAVIORAL DESIGN PATTERN - STRATEGY

❑ Lợi ích của Strategy Pattern

- ❖ Đảm bảo nguyên tắc Single responsibility principle (SRP)
 - một lớp định nghĩa nhiều hành vi và chúng xuất hiện dưới dạng với nhiều câu lệnh có điều kiện.
 - Thay vì nhiều điều kiện, chúng ta sẽ chuyển các nhánh có điều kiện liên quan vào lớp Strategy riêng lẻ của nó.
- ❖ Đảm bảo nguyên tắc Open/Closed Principle (OCP):
 - chúng ta dễ dàng mở rộng và kết hợp hành vi mới mà không thay đổi ứng dụng.
- ❖ Cung cấp một sự thay thế cho kế thừa.

BEHAVIORAL DESIGN PATTERN - STRATEGY

❑ Sử dụng Strategy Pattern

- ❖ Khi muốn có thể thay đổi các thuật toán được sử dụng bên trong một đối tượng tại thời điểm run-time.
- ❖ Khi có một đoạn mã dễ thay đổi, và muốn tách chúng ra khỏi chương trình chính để dễ dàng bảo trì.
- ❖ Khi cần che dấu sự phức tạp, cấu trúc bên trong của thuật toán.
- ❖ Tránh sự rắc rối, khi phải hiện thực một chức năng nào đó qua quá nhiều lớp con.

BEHAVIORAL DESIGN PATTERN - COMMAND

❑ Tình huống

- ❖ Lớp A (đã sẵn) cung cấp một số chức năng hữu ích cơ bản
- ❖ Lớp B (đang viết) cần thực hiện một số "công việc" (có thể phức tạp) thông qua sử dụng các chức năng cơ bản của A.

- ❖ Yêu cầu:
 - Muốn giảm sự phụ thuộc giữa B vào A
 - ✓ Chuyển từ: B cài đặt và gọi thực thi "công việc" thành: B gọi thực thi "công việc" (ra lệnh)
 - Đóng gói các "công việc" thành các "đối tượng"
 - Muốn có khả năng "undo/redo" các "công việc"
 - Muốn thực hiện một số "công việc" theo dạng transaction



Command Pattern is a behavioral design pattern that encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

- ❖ cho phép chuyển Request thành đối tượng độc lập, có thể được sử dụng để tham số hóa các đối tượng với các yêu cầu khác nhau.
 - những Request gửi đến object được lưu trữ trong chính object đó dưới dạng một object Command - một class trung gian được tạo ra để lưu trữ các câu lệnh và trạng thái của object tại một thời điểm nào đó.

- ❖ Tần suất sử dụng: ★★★★ cao trung bình

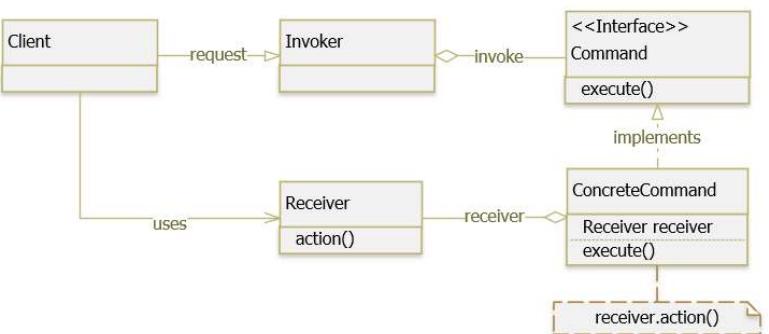
BEHAVIORAL DESIGN PATTERN - COMMAND

Cách thực hiện

- ❖ Command
 - là một interface hoặc abstract class,
 - chứa một phương thức trừu tượng thực thi (execute) một hành động (operation).
- ❖ ConcreteCommand
 - là (các) implementation của Command.
 - Thực thi execute() bằng việc gọi operation đang "chờ" trên Receiver.
- ❖ Client
 - là một lớp tiếp nhận request từ 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
 - Là một lớp cài đặt thực sự business logic cho case request.
 - Trong phương thức execute() của ConcreteCommand chúng ta sẽ gọi method thích hợp trong Receiver.

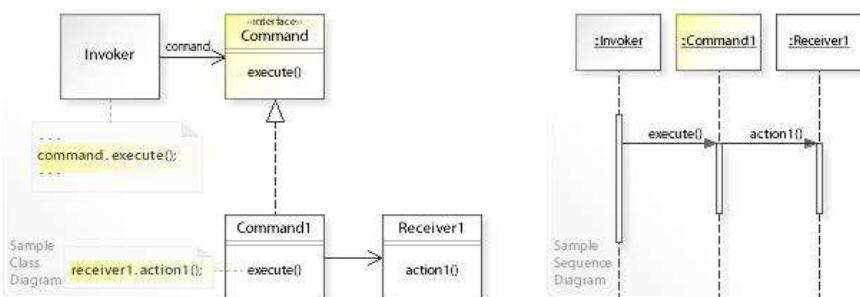
BEHAVIORAL DESIGN PATTERN - COMMAND

Cài đặt



BEHAVIORAL DESIGN PATTERN - COMMAND

Cài đặt



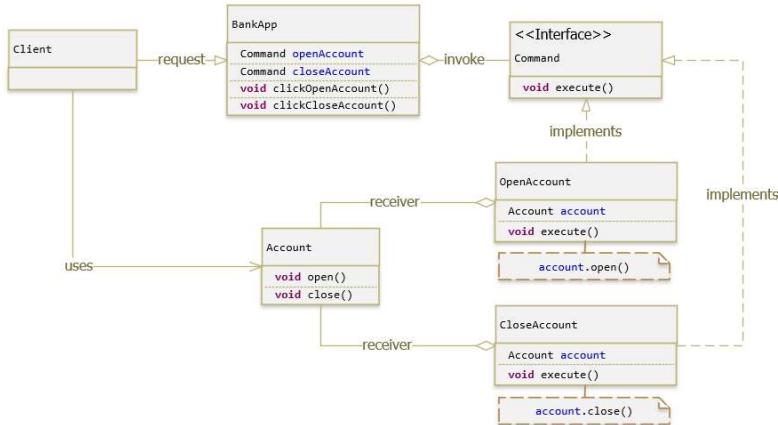
BEHAVIORAL DESIGN PATTERN - COMMAND

Cài đặt minh họa

- ❖ Ngân hàng cung cấp ứng dụng cho phép khách hàng (client) có thể mở (open) hoặc đóng (close) tài khoản trực tuyến.
- ❖ Hệ thống này được thiết kế theo dạng module, mỗi module sẽ thực hiện một nhiệm vụ: module mở tài khoản (OpenAccount), module đóng tài khoản (CloseAccount).
- ❖ Khi có yêu cầu từ client, hệ thống sẽ đóng gói yêu cầu và gọi module tương ứng xử lý.

BEHAVIORAL DESIGN PATTERN - COMMAND

☐ Cài đặt minh họa



BEHAVIORAL DESIGN PATTERN - COMMAND

☐ Cài đặt minh họa

```

23. public class OpenAccount implements Command {
24.     private Account account;
25.
26.     public OpenAccount(Account account) {
27.         this.account = account;
28.     }
29.
30.     @Override
31.     public void execute() {
32.         account.open();
33.     }
34.
35.     public class CloseAccount implements Command {
36.         private Account account;
37.
38.         public CloseAccount(Account account) {
39.             this.account = account;
40.         }
41.
42.         @Override
43.         public void execute() {
44.             account.close();
45.         }
46.
47.     }
  
```

Java code implementation of the Command design pattern. It defines Command, OpenAccount, CloseAccount, and Account classes. The Command interface has an execute() method. OpenAccount and CloseAccount implement Command and have a receiver relationship to Account (account.open() and account.close() respectively). Account also implements Command.

BEHAVIORAL DESIGN PATTERN - COMMAND

☐ Cài đặt minh họa

```

49. public class BankApp {
50.     private Command openAccount;
51.     private Command closeAccount;
52.
53.     public BankApp(Command openAccount,
54.                     Command closeAccount) {
55.         this.openAccount = openAccount;
56.         this.closeAccount = closeAccount;
57.     }
58.
59.     public void clickOpenAccount() {
60.         System.out.println("open an account");
61.         openAccount.execute();
62.     }
63.
64.     public void clickCloseAccount() {
65.         System.out.println("close an account");
66.         closeAccount.execute();
67.     }
68. }
  
```

```

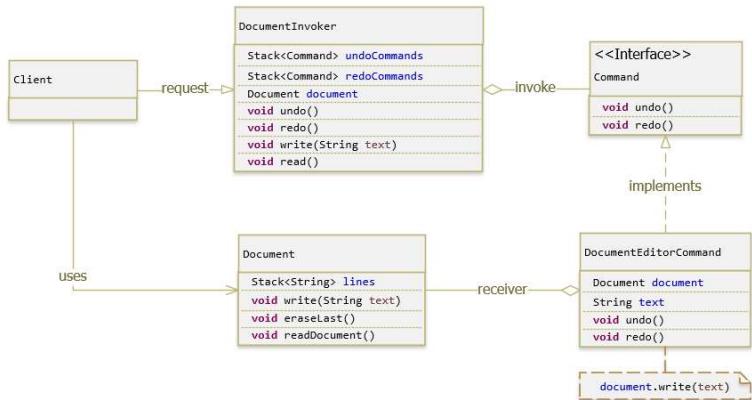
70. public class Client {
71.     public static void main(String[] args) {
72.         Account account = new Account("QNU");
73.
74.         Command open = new OpenAccount(account);
75.         Command close = new CloseAccount(account);
76.         BankApp bankApp = new BankApp(open, close);
77.
78.         bankApp.clickOpenAccount();
79.         bankApp.clickCloseAccount();
80.     }
81. }
  
```

Java code implementation of the Command design pattern. It defines BankApp, Client, Account, OpenAccount, CloseAccount, and Command classes. BankApp holds Command objects. Client performs actions like clickOpenAccount and clickCloseAccount, which invoke the execute() methods of the Command objects.

BEHAVIORAL DESIGN PATTERN - COMMAND

☐ Cài đặt minh họa

❖ Xây dựng ứng dụng văn bản hỗ trợ undo/redo



BEHAVIORAL DESIGN PATTERN - COMMAND

❑ Cài đặt minh họa

```
01. public class Document {  
02.     private Stack<String> lines = new Stack<>();  
03.  
04.     public void write(String text) {  
05.         lines.push(text);  
06.     }  
07.  
08.     public void eraseLast() {  
09.         if (!lines.isEmpty()) {  
10.             lines.pop();  
11.         }  
12.     }  
13.  
14.     public void readDocument() {  
15.         for (String line : lines) {  
16.             System.out.println(line);  
17.         }  
18.     }  
19. }
```

BEHAVIORAL DESIGN PATTERN - COMMAND

❑ Cài đặt minh họa

```
21. public interface Command {  
22.     void undo();  
23.     void redo();  
24. }  
25.  
26. public class DocumentEditorCommand implements Command {  
27.     private Document document;  
28.     private String text;  
29.  
30.     public DocumentEditorCommand(Document doc, String text) {  
31.         this.document = doc;  
32.         this.text = text;  
33.         this.document.write(text);  
34.     }  
35.  
36.     public void undo() {  
37.         document.eraseLast();  
38.     }  
39.  
40.     public void redo() {  
41.         document.write(text);  
42.     }  
43. }
```

BEHAVIORAL DESIGN PATTERN - COMMAND

❑ Cài đặt minh họa

```
45. public class DocumentInvoker {  
46.     private Stack<Command> undoCmds = new Stack<>();  
47.     private Stack<Command> redoCmds = new Stack<>();  
48.     private Document doc = new Document();  
49.  
50.     public void read() {  
51.         doc.readDocument();  
52.     }  
53.  
54.     public void write(String text) {  
55.         Command cmd = new  
56.             DocumentEditorCommand(doc, text);  
57.         undoCmds.push(cmd);  
58.         redoCmds.clear();  
59.     }  
60.  
61.     public void undo() {  
62.         if (!undoCmds.isEmpty()) {  
63.             Command cmd = undoCmds.pop();  
64.             cmd.undo(); //xoá dòng vừa viết  
65.             redoCmds.push(cmd);  
66.         } else {  
67.             System.out.println("Nothing to undo");  
68.         }  
69.     }  
70.  
71.     public void redo() {  
72.         if (!redoCmds.isEmpty()) {  
73.             Command cmd = redoCmds.pop();  
74.             cmd.redo(); //viet lai dòng ra  
75.             undoCmds.push(cmd);  
76.         } else {  
77.             System.out.println("Nothing to redo");  
78.         }  
79.     }  
80. }
```

BEHAVIORAL DESIGN PATTERN - COMMAND

❑ Cài đặt minh họa

```
82. public class Client {  
83.     public static void main(String[] args) {  
84.         DocumentInvoker instance = new DocumentInvoker();  
85.         instance.write("Dong dau tien");  
86.         instance.undo();  
87.         instance.read(); //EMPTY  
88.  
89.         instance.redo();  
90.         instance.read(); //Dong dau tien.  
91.  
92.         instance.write("Dong thu 2");  
93.         instance.write("Dong thu 3");  
94.         instance.read(); //Dong dau tien. Dong thu 2. Dong thu 3.  
95.         instance.undo(); //Dong dau tien. Dong thu 2.  
96.         instance.undo(); //Dong dau tien.  
97.         instance.undo(); //EMPTY  
98.         instance.undo(); //Nothing to undo  
99.     }  
100. }
```

BEHAVIORAL DESIGN PATTERN - COMMAND

❑ Lợi ích của Command Pattern

- ❖ Dễ dàng thêm các Command mới trong hệ thống mà không cần thay đổi trong các lớp hiện có. Đảm bảo Open/Closed Principle.
- ❖ Tách đối tượng gọi operation "ra khỏi" đối tượng thực sự thực hiện operation. Giảm kết nối giữa Invoker và Receiver.
- ❖ Cho phép tham số hóa các yêu cầu khác nhau bằng một hành động để thực hiện.
- ❖ Cho phép lưu các yêu cầu trong hàng đợi.
- ❖ Đóng gói một yêu cầu trong một đối tượng. Dễ dàng chuyển dữ liệu dưới dạng đối tượng giữa các thành phần hệ thống.

BEHAVIORAL DESIGN PATTERN - COMMAND

❑ Sử dụng Command Pattern

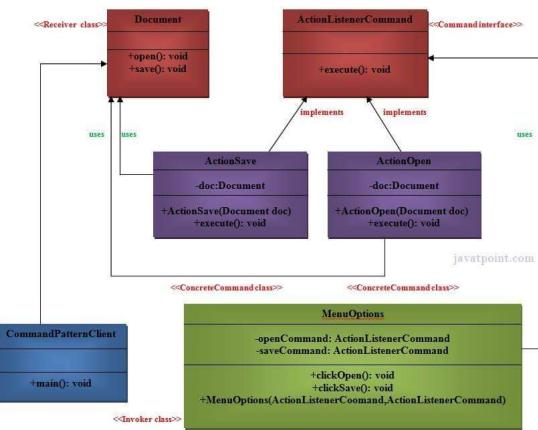
- ❖ Khi cần tham số hóa các đối tượng theo một hành động thực hiện.
- ❖ Khi cần tạo và thực thi các yêu cầu vào các thời điểm khác nhau.
- ❖ Khi cần hỗ trợ tính năng undo, log, callback hoặc transaction.

BEHAVIORAL DESIGN PATTERN - COMMAND

❑ Bài tập:

- ❖ 1 đối tượng Document cung cấp 2 chức năng Open và Save
- ❖ Muốn cài đặt 1 chương trình có hệ thống menu chứa 2 chức năng này

❑ Bài tập



BEHAVIORAL DESIGN PATTERN - COMMAND

❑ Bài tập

```
01. public class Document {  
02.     public void open(){  
03.         System.out.println("Open...");  
04.     }  
05.     public void save(){  
06.         System.out.println("Save...");  
07.     }  
08. }  
09.  
10. public interface ActionListenerCommand {  
11.     public void execute();  
12. }  
  
14. public class CommandPatternClient {  
15.     public static void main(String[] args) {  
16.         Document doc = new Document();  
17.  
18.         ActionListenerCommand clickOpen  
19.             = new ActionOpen(doc);  
20.         ActionListenerCommand clickSave  
21.             = new ActionSave(doc);  
22.  
23.         MenuOptions menu =  
24.             new MenuOptions(clickOpen, clickSave);  
25.  
26.         menu.clickOpen();  
27.         menu.clickSave();  
28.     }  
29. }
```

BEHAVIORAL DESIGN PATTERN - COMMAND

❑ Bài tập

```
31. public class ActionOpen implements  
32.     ActionListenerCommand{  
33.         private Document doc;  
34.         public ActionOpen(Document doc) {  
35.             this.doc = doc;  
36.         }  
37.         @Override  
38.         public void execute() {  
39.             doc.open();  
40.         }  
41.     }  
42.  
43. public class ActionSave implements  
44.     ActionListenerCommand{  
45.         private Document doc;  
46.         public ActionSave(Document doc) {  
47.             this.doc = doc;  
48.         }  
49.         @Override  
50.         public void execute() {  
51.             doc.save();  
52.         }  
53.     }  
  
55. public class MenuOptions {  
56.     private ActionListenerCommand openCommand;  
57.     private ActionListenerCommand saveCommand;  
58.  
59.     public MenuOptions(ActionListenerCommand open,  
60.                         ActionListenerCommand save) {  
61.         this.openCommand = open;  
62.         this.saveCommand = save;  
63.     }  
64.     public void clickOpen(){  
65.         openCommand.execute();  
66.     }  
67.     public void clickSave(){  
68.         saveCommand.execute();  
69.     }  
70. }
```