

# MỤC LỤC

CHƯƠNG 1.	CÁC NGUYÊN LÝ THIẾT KẾ.....	2
1.1.	Single Responsibility Principle (Nguyên lý đơn nhiệm) .....	3
1.2.	Open Close Principle (Nguyên lý đóng mở).....	4
1.3.	Liskov Substitution Principle (Nguyên lý thay thế Liskov) .....	6
1.4.	Interface Segregation (Nguyên lý chia tách giao diện).....	9
1.5.	Dependency Inversion Principle (Nguyên lý phụ thuộc đảo).....	10
CHƯƠNG 2.	PATTERN. ....	12
2.1.	Khái niệm. ....	12
2.2.	Đặc điểm.....	13
2.3.	Liệt kê danh sách pattern.....	13
2.4.	Mối quan hệ giữa các Design Pattern.....	17
2.5.	Structural Pattern .....	18
2.5.1.	Adapter Pattern.....	18
2.5.2.	Bridge Pattern.....	24
2.5.3.	Composite Pattern .....	31
2.5.4.	Decorator Pattern.....	34
2.6.	Creational patterns.....	38
2.6.1.	Abstract Factory Method Pattern .....	38
2.6.2.	Builder Pattern.....	42
2.6.3.	Factory Method .....	48
2.6.4.	Prototype .....	52
2.6.5.	Singleton.....	55
2.7.	Behavioral patterns.....	56
2.7.1.	Chain of Responsibility .....	56
2.7.2.	Command Pattern .....	59
2.7.3.	Interpreter Pattern.....	64
2.8.	Model-View-Controller Pattern (MVC Pattern) .....	69

# CHƯƠNG 1. CÁC NGUYÊN LÝ THIẾT KẾ

Ở bài tìm hiểu trước về hướng đối tượng, ta đã hiểu được khái quát về những khái niệm cần thiết trong hướng đối tượng như:

- Abstraction (Tính trừu tượng)
- Encapsulation (Tính bao đóng)
- Inheritance (Tính kế thừa)
- Polymorphism (Tính đa hình)

Để trả lời câu hỏi thế nào là một phần mềm hướng đối tượng? Đây là những cơ sở nền tảng để xây dựng được phần mềm theo tư tưởng hướng đối tượng đúng nghĩa?. Đó là những quy tắc phân tích thiết kế hướng đối tượng cơ bản, mang tính chất khái quát. Do là nguyên lý nên có tính trừu tượng cao chứ không đi vào chi tiết cách thức giải quyết vấn đề cụ thể (việc hiện thực hóa những nguyên lý lập trình hướng đối tượng đòi hỏi chúng ta phải xem xét đến Design Patterns).



5 nguyên tắc cơ bản trong lập trình được gói gọn trong từ “SOLID”. Vậy Solid là gì? Solid được dịch ra tiếng việt có nghĩa là “Cứng” từ đó suy ra cứ áp dụng nguyên lý Solid trong lập trình là cứng?

Thực ra SOLID được ghép bởi 5 nguyên lý sau:



Những nguyên tắc này tuy rất khó nhớ tên nhưng ít nhiều ta đều tiếp xúc trong công việc hàng ngày. Vì vậy nhận ra khi nào ta cần áp dụng nguyên tắc nào sẽ rất có lợi cho việc lập trình của bạn.

### 1.1. Single Responsibility Principle (Nguyên lý đơn nhiệm)

Phát biểu:

“A class should have only one reason to change.”<sup>1</sup>

*Một class chỉ nên giữ 1 trách nhiệm duy nhất*

*(Chỉ có thể sửa đổi class với 1 lý do duy nhất)*

Để hiểu nguyên lý này, ta lấy ví dụ với 1 class **vi phạm nguyên lý**.

```
1 public class ReportManager()  
2 {  
3     public void ReadDataFromDB();  
4     public void ProcessData();  
5     public void PrintReport();  
6 }
```

Class này giữ tới 3 trách nhiệm: Đọc dữ liệu từ DB, xử lý dữ liệu, in kết quả. Do đó, chỉ cần ta thay đổi DB, thay đổi cách xuất kết quả, ... ta sẽ phải sửa đổi class này. Càng về sau class sẽ càng phình to ra. Theo đúng nguyên lý, ta phải tách class này ra làm 3 class riêng. Tuy số lượng class nhiều hơn những việc sửa chữa sẽ đơn giản hơn, class ngắn hơn nên cũng ít bug hơn.

<sup>1</sup> Xem: <http://www.oodeesign.com/single-responsibility-principle.html>

## 1.2. Open Close Principle (Nguyên lý đóng mở)

Phát biểu:

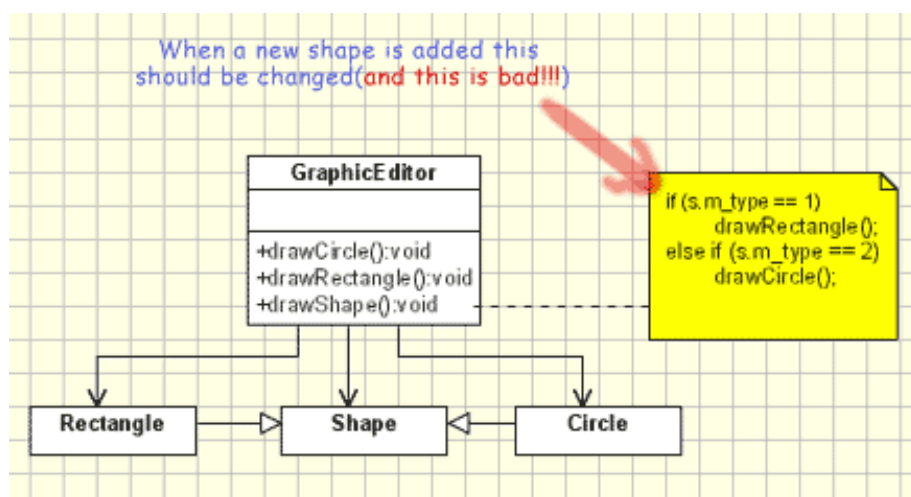
*“Software entities like classes, modules and functions should be **open for extension** but **closed for modifications**.”*

Các thực thể phần mềm (hàm, đơn thể, đối tượng, ...) nên được xây dựng theo hướng mở cho việc mở rộng (be opened for extension) nhưng đóng đối với việc sửa đổi (be closed for modification).

Có thể thoát mái mở rộng (Extend) 1 class, nhưng không được sửa đổi bên trong class đó (open for extension but closed for modification).

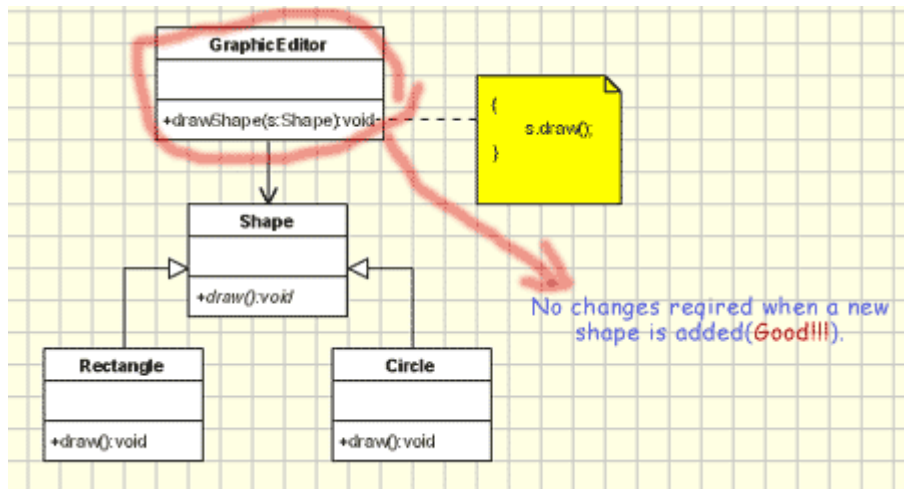
Nội dung:

Các thực thể trong một phần mềm không đứng riêng lẻ mà có sự gắn kết chặt chẽ với nhau. Chúng phối hợp hoạt động để cùng nhau thực hiện các chức năng của phần mềm. Do đó, việc nâng cấp, mở rộng một thực thể nào đó sẽ ảnh hưởng đến những thực thể liên quan. Điều này có thể dẫn đến việc phải nâng cấp, mở rộng cả những thực thể liên quan đó. Với nhu cầu hiện nay, việc phải thường xuyên nâng cấp, mở rộng các thực thể trong phần mềm là điều khó tránh khỏi.



Giả sử ta thêm hình tam giác vào thì GraphicEditor một lần nữa phải thay đổi. Và hãy tưởng tượng đến một lúc nào đó GraphicEditor phình ra thì ta không thể nào kiểm soát nổi, hay một lớp con kế thừa chỉ cần một

phương thức mà lại phải thực thi hết toàn bộ những phương thức đã có thì không hợp lý. Nên chương trình sẽ được viết lại như sau:



```
1 // Open-Close Principle example
2 class GraphicEditor {
3     public void drawShape(Shape s) {
4         s.draw();
5     }
6 }
7
8 public abstract class Shape {
9     abstract void draw();
10 }
11
12 class Rectangle extends Shape {
13     public void draw() {
14         // draw the rectangle
15     }
16 }
17
18 class Circle extends Shape {
19     public void draw() {
20         // draw the Circle
21     }
22 }
```

Việc tuân thủ nguyên lý Open-Closed của một thực thể phần mềm chỉ mang tính tương đối, phụ thuộc vào ngữ cảnh. Có thể trong ngữ cảnh này, thực thể thỏa nguyên lý, nhưng trong một ngữ cảnh khác, thực thể này không còn tuân thủ nguyên lý nữa. Mục tiêu của phân tích thiết kế hướng đối tượng là phải làm sao cho có nhiều thực thể phần mềm nhất tuân thủ nguyên lý

trong ngữ cảnh thường xảy ra nhất của phần mềm, trong đó ưu tiên các thực thể thường xuyên phải nâng cấp, mở rộng thỏa nguyên lý.

Ý nghĩa Nguyên lý Open-Closed là nguyên lý cốt lõi và là một trong bốn nguyên lý cơ bản làm nền tảng cho phân tích thiết kế hướng đối tượng. Nó giúp cho phần mềm dễ bảo trì, nâng cấp và mở rộng.

### 1.3. Liskov Substitution Principle (Nguyên lý thay thế Liskov)

Phát biểu

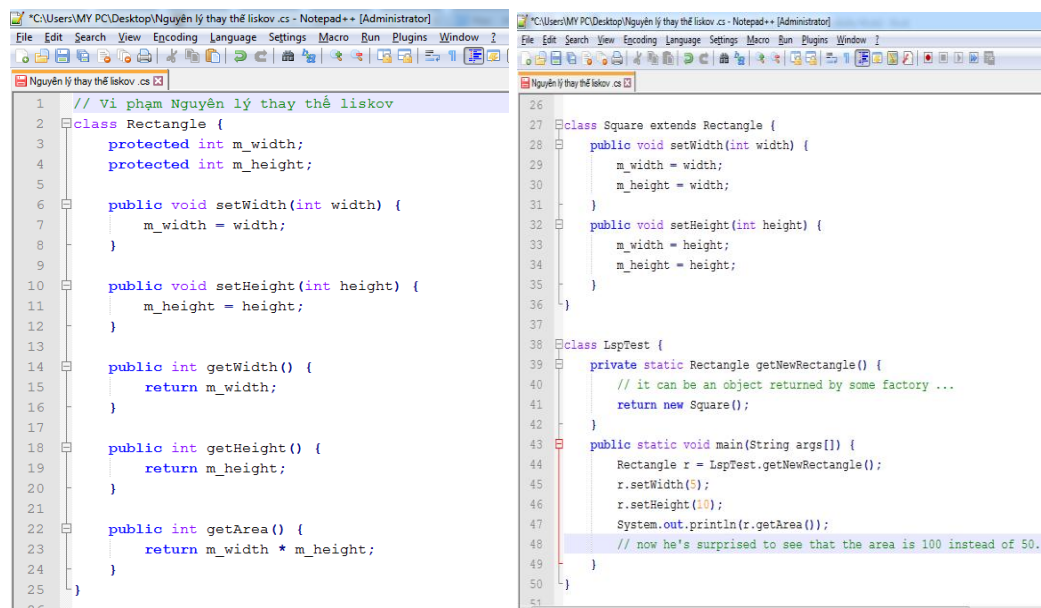
*“Derived types must be completely substitutable for their base types”*

Nội dung

Các chức năng trong hệ thống vẫn thực hiện đúng đắn nếu ta thay đổi bất kỳ đối tượng nào của lớp dẫn xuất bằng đối tượng lớp kế thừa

Ngoài ra nó còn là một phương tiện để kiểm tra xem chương trình hoặc bản thiết kế có thỏa mãn nguyên lý đóng/ mở hay không. Vì sự vi phạm nguyên lý Thay thế Liskov sẽ dẫn đến sự vi phạm nguyên lý Open-Closed.

Chúng ta biết "hình vuông là hình chữ nhật". Nhưng hình chữ nhật có thay thế được hình vuông? Hãy xét mối quan hệ kế thừa đơn giản trong hướng đối tượng giữa hình chữ nhật và hình vuông.



```
// Vi phạm Nguyên lý thay thế liskov
class Rectangle {
    protected int m_width;
    protected int m_height;

    public void setWidth(int width) {
        m_width = width;
    }

    public void setHeight(int height) {
        m_height = height;
    }

    public int getWidth() {
        return m_width;
    }

    public int getHeight() {
        return m_height;
    }

    public int getArea() {
        return m_width * m_height;
    }
}

class Square extends Rectangle {
    public void setWidth(int width) {
        m_width = width;
        m_height = width;
    }

    public void setHeight(int height) {
        m_width = height;
        m_height = height;
    }
}

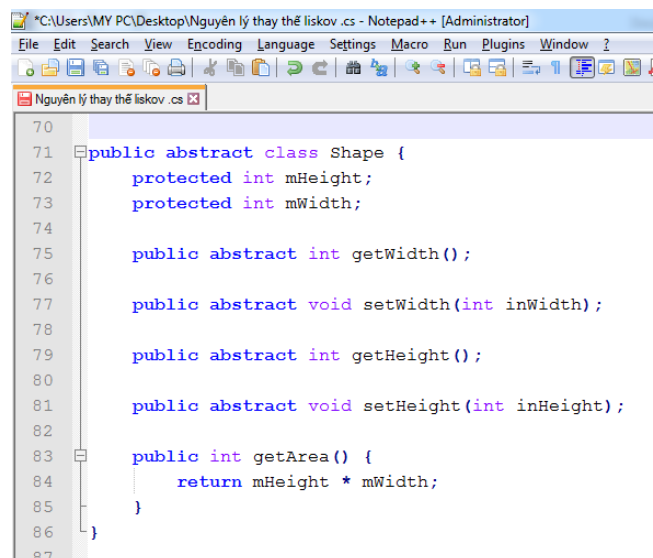
class LspTest {
    private static Rectangle getNewRectangle() {
        // it can be an object returned by some factory ...
        return new Square();
    }

    public static void main(String args[]) {
        Rectangle r = LspTest.getNewRectangle();
        r.setWidth(5);
        r.setHeight(10);
        System.out.println(r.getArea());
        // now he's surprised to see that the area is 100 instead of 50.
    }
}
```

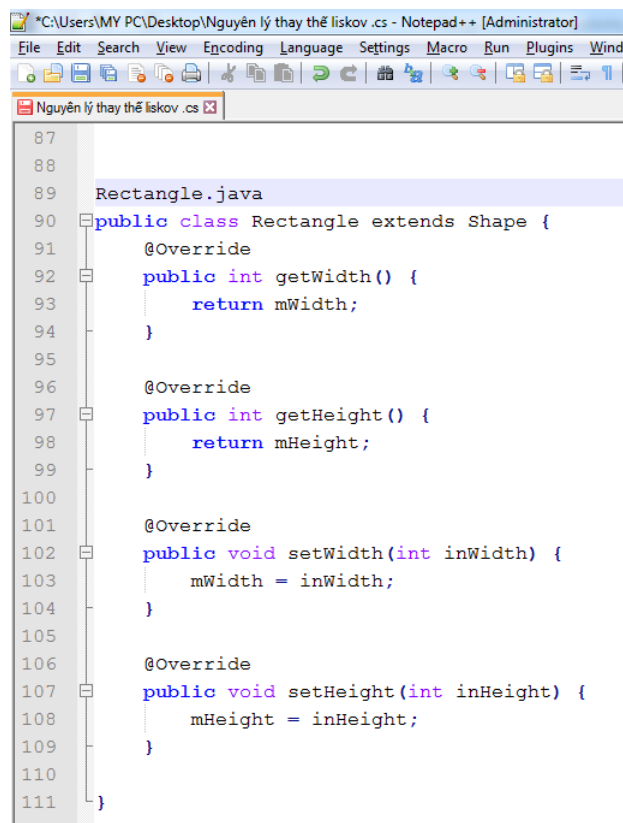
Ở ví dụ trên, chúng ta đã vi phạm nguyên lý thay thế Liskov bởi vì thuộc tính Width và Height đã bị thay đổi ở lớp Square. Vì vậy, khi thay thế Square bằng Rectangle sẽ dẫn đến kết quả không đúng. (Rectangle phải ra  $5 \times 10 = 50$  chứ không phải  $10 \times 10 = 100$ ).

Chúng ta phải bảo đảm rằng, khi một lớp con kế thừa từ một lớp khác, nó sẽ không làm thay đổi hành vi của lớp đó. Hay trong ví dụ này là lớp Square không được phép thay đổi hành vi của lớp Rectangle (Width và Height)

Bây giờ hãy tạo ra một thiết kế mới bằng cách áp dụng nguyên lý thay thế Liskov. Chúng ta có 1 lớp Shape sẽ là lớp cơ sở cho cả 2 lớp Rectangle và Square.



```
70
71 public abstract class Shape {
72     protected int mHeight;
73     protected int mWidth;
74
75     public abstract int getWidth();
76
77     public abstract void setWidth(int inWidth);
78
79     public abstract int getHeight();
80
81     public abstract void setHeight(int inHeight);
82
83     public int getArea() {
84         return mHeight * mWidth;
85     }
86 }
87
```



```
87
88
89 Rectangle.java
90 public class Rectangle extends Shape {
91     @Override
92     public int getWidth() {
93         return mWidth;
94     }
95
96     @Override
97     public int getHeight() {
98         return mHeight;
99     }
100
101     @Override
102     public void setWidth(int inWidth) {
103         mWidth = inWidth;
104     }
105
106     @Override
107     public void setHeight(int inHeight) {
108         mHeight = inHeight;
109     }
110
111 }
```

```
*C:\Users\MY PC\Desktop\Nguyễn lý thay thế liskov.cs - Notepad++ [Administrator]
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
Nguyễn lý thay thế liskov.cs
1
2 public class Square extends Shape {
3
4     @Override
5     public int getWidth() {
6         return mWidth;
7     }
8     @Override
9     public void setWidth(int inWidth) {
10         SetWidthAndHeight(inWidth);
11     }
12     @Override
13     public int getHeight() {
14         return mHeight;
15     }
16     @Override
17     public void setHeight(int inHeight) {
18         SetWidthAndHeight(inHeight);
19     }
20     private void SetWidthAndHeight(int inValue) {
21         mHeight = inValue;
22         mWidth = inValue;
23     }
24 }
25
```

```
*C:\Users\MY PC\Desktop\Nguyễn lý thay thế liskov.cs - Notepad++ [Administrator]
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
Nguyễn lý thay thế liskov.cs
55
56 public class Main {
57
58     static final String SQUARE = "Square";
59     static final String RECTANGLE = "Rectangle";
60     public static void main(String[] args) {
61         Shape shape1 = getShape(SQUARE);
62         shape1.setHeight(10);
63         shape1.setWidth(20);
64         System.out.println(SQUARE + "'s area: " + shape1.getArea());
65
66         Shape shape2 = getShape(RECTANGLE);
67         shape2.setHeight(10);
68         shape2.setWidth(20);
69         System.out.println(RECTANGLE + "'s area: " + shape2.getArea());
70     }
71     static Shape getShape(String inShapeType) {
72         if (inShapeType.equals(SQUARE)) {
73             return new Square();
74         }
75         if (inShapeType.equals(RECTANGLE)) {
76             return new Rectangle();
77         }
78         return null;
79     }
80 }
```



## 1.4. Interface Segregation (Nguyên lý chia tách giao diện)

Phát biểu

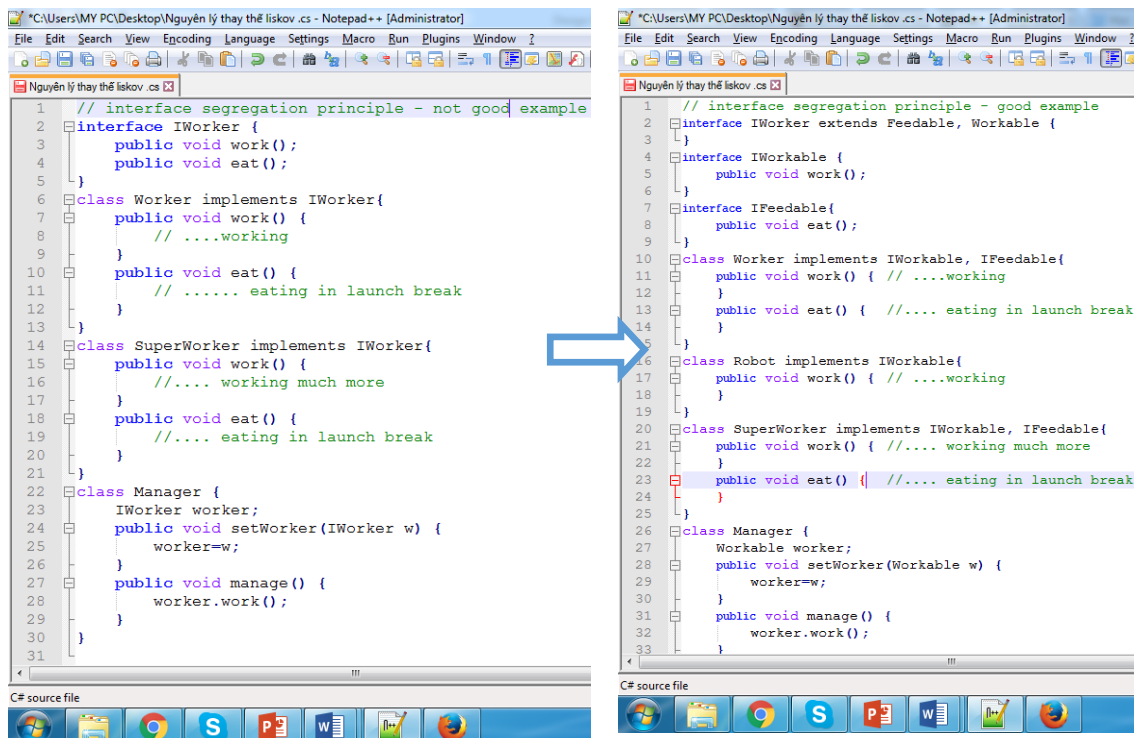
*“Clients should not be forced to depend upon interfaces that they don't use.”*

Không nên buộc các thực thể phần mềm phụ thuộc vào những interface mà chúng không sử dụng đến.

Nội dung

Thay vì dùng 1 interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể.

Nguyên lý này khá dễ hiểu. Hãy tưởng tượng chúng ta có 1 interface lớn, khoảng 100 methods. Việc implements sẽ khá cực khổ, ngoài ra còn có thể dư thừa vì 1 class không cần dùng hết 100 method. Khi tách interface ra thành nhiều interface nhỏ, gồm các method liên quan tới nhau, việc implement + quản lý sẽ dễ hơn.



```
// interface segregation principle - not good example
1 interface IWorker {
2     public void work();
3     public void eat();
4 }
5
6 class Worker implements IWorker{
7     public void work() {
8         // ....working
9     }
10    public void eat() {
11        // ..... eating in launch break
12    }
13 }
14
15 class SuperWorker implements IWorker{
16     public void work() {
17         //..... working much more
18     }
19     public void eat() {
20         //..... eating in launch break
21     }
22 }
23
24 class Manager {
25     IWorker worker;
26     public void setWorker(IWorker w) {
27         worker=w;
28     }
29     public void manage() {
30         worker.work();
31     }
32 }
33
```

```
// interface segregation principle - good example
1 interface IWorker extends Feedable, Workable {
2 }
3
4 interface IWorkable {
5     public void work();
6 }
7
8 interface IFeedable{
9     public void eat();
10 }
11
12 class Worker implements IWorker{
13     public void work() { // ....working
14     }
15     public void eat() { //..... eating in launch break
16     }
17 }
18
19 class Robot implements IWorkable{
20     public void work() { // ....working
21     }
22 }
23
24 class SuperWorker implements IWorkable, IFeedable{
25     public void work() { //..... working much more
26     }
27     public void eat() { //..... eating in launch break
28     }
29 }
30
31 class Manager {
32     Workable worker;
33     public void setWorker(Workable w) {
34         worker=w;
35     }
36     public void manage() {
37         worker.work();
38     }
39 }
```

## 1.5. Dependency Inversion Principle (Nguyên lý phụ thuộc đảo)

Phát biểu

*“High-level modules should not depend on low-level modules.*

*Both should depend on abstractions.*

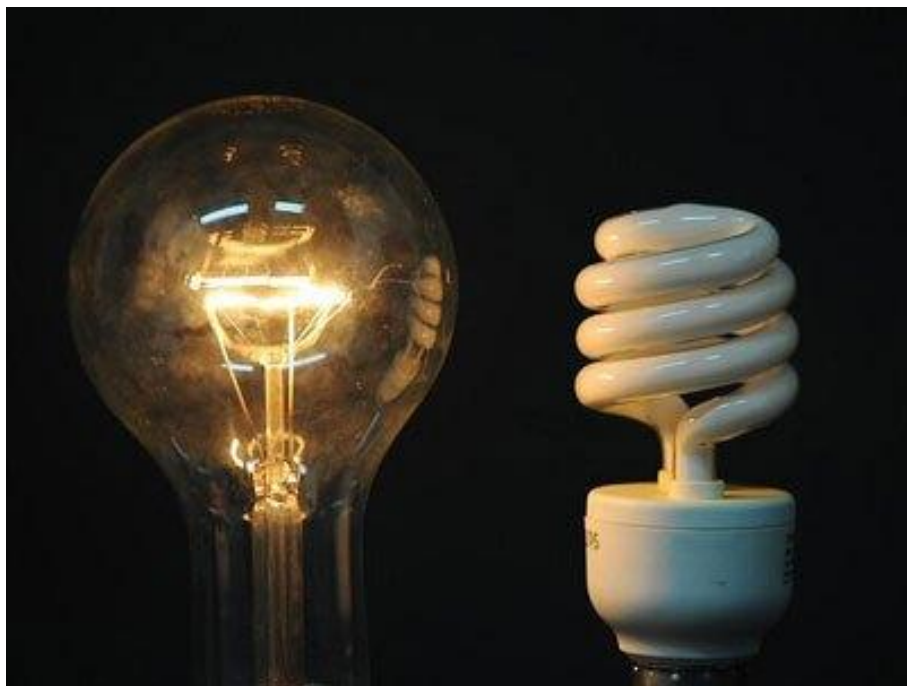
*Abstractions should not depend on details.*

*Details should depend on abstractions.”*

Các thành phần trong phần mềm không nên phụ thuộc vào những cái riêng, cụ thể (details) mà ngược lại nên phụ thuộc vào những cái chung, tổng quát (abstractions) của những cái riêng, cụ thể đó. Những cái chung, tổng quát (abstractions) không nên phụ vào những cái riêng, cụ thể (details). Sự phụ thuộc này nên được đảo ngược lại.

Nội dung

Chúng ta đều biết 2 loại đèn: đèn tròn và đèn huỳnh quang. Chúng cùng có **đuôi tròn**, do đó ta có thể thay thế đèn tròn bằng đèn huỳnh quang cho nhau 1 cách dễ dàng.



Ở đây, interface chính là **đuôi tròn**, implementation là **bóng đèn tròn** và **bóng đèn huỳnh quang**. Ta có thể swap dễ dàng giữa 2 loại bóng vì ổ điện chỉ quan tâm tới interface (đuôi tròn), không quan tâm tới implementation.

Trong code cũng vậy, khi áp dụng Dependency Inverse, ta chỉ cần quan tâm tới interface. Để kết nối tới database, ta chỉ cần gọi hàm Get, Save ... của Interface IDataAccess. Khi thay database, ta chỉ cần thay implementation của interface này.

## CHƯƠNG 2. PATTERN.

### 2.1. Khái niệm.

Trong lĩnh vực kỹ thuật phần mềm, Design Pattern là một giải pháp tổng thể có thể lặp lại cho một vấn đề thường xảy ra trong việc thiết kế phần mềm.



Design Pattern không phải là một thiết kế hoàn chỉnh để chuyển trực tiếp thành code. Nó là một bản mô tả hay cái khung cho cách giải quyết một vấn đề mà có thể được sử dụng trong nhiều tình huống khác nhau.

Design Pattern tuân thủ nghiêm ngặt các nguyên lý thiết kế hướng đối tượng ở trên. Mẫu thiết kế không đơn thuần là một bước nào đó trong các giai đoạn phát triển phần mềm mà nó đóng vai trò là sáng kiến để giải quyết một vấn đề thông dụng nào đó.

Mẫu thiết kế sẽ giúp cho việc giải quyết vấn đề nhanh, gọn và hợp lý hơn. Mẫu thiết kế còn được sử dụng nhằm cô lập các thay đổi trong mã nguồn, từ đó làm cho hệ thống có khả năng tái sử dụng cao.

Như vậy lợi ích của việc sử dụng các mẫu thiết kế là rất thiết thực. Design Pattern giúp bạn tái sử dụng và dễ dàng mở rộng. Giúp bạn giải quyết vấn đề thay vì tự tìm kiếm giải pháp cho một vấn đề đã được chứng minh. 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. Dùng lại design pattern giúp tránh các vấn đề tìm ẩn có thể gây ra những lỗi lớn, dễ dàng nâng cấp và bảo trì về sau. Và để hiểu rõ hơn, chúng ta sẽ tìm hiểu một số mẫu thiết kế dưới đây.

## 2.2. Đặc điểm.

Hệ thống các mẫu design pattern hiện có hơn 20 mẫu được định nghĩa trong cuốn “Design patterns Elements of Reusable Object Oriented Software”. Hệ thống các mẫu này có thể nói là đủ và tối ưu cho việc giải quyết hết các vấn đề của bài toán phân tích thiết kế và xây dựng phần mềm trong thời điểm hiện tại.

Hệ thống các mẫu design pattern được chia thành 3 nhóm: **Creational, Structural, Behavioral**.

## 2.3. Liệt kê danh sách pattern.

**a. Nhóm cấu trúc tĩnh (Structural Pattern):** Liên quan đến vấn đề làm thế nào để các lớp và đối tượng kết hợp với nhau tạo thành các cấu trúc lớn hơn.

STT	Tên	Mục đích
1	<b>Adapter</b>	Do vấn đề tương thích, thay đổi interface của một lớp thành một interface khác phù hợp với yêu cầu người sử dụng lớp.
2	<b>Bridge</b>	Tách rời ngữ nghĩa của một vấn đề khỏi việc cài đặt ; mục đích để cả hai bộ phận (ngữ nghĩa và cài đặt) có thể thay đổi độc lập nhau.
3	<b>Composite</b>	Tổ chức các đối tượng theo cấu trúc phân cấp dạng cây; Tất cả các đối tượng trong cấu trúc được thao tác theo một cách thuần nhất như nhau. Tạo quan hệ thứ bậc bao gộp giữa các đối tượng. Client có thể xem đối tượng bao gộp và bị bao gộp như nhau - > khả năng tổng quát hoá trong code của client -> dễ phát triển, nâng cấp, bảo trì
4	<b>Decorator</b>	Gán thêm trách nhiệm cho đối tượng (mở rộng chức năng) vào lúc chạy (dynamically).
5	<b>Facade</b>	Cung cấp một interface thuần nhất cho một tập hợp các interface trong một “hệ thống con” (subsystem). Nó định nghĩa 1 interface cao hơn các interface có sẵn để làm cho hệ thống con dễ sử dụng hơn

<b>6</b>	<b>Flyweight</b>	Sử dụng việc chia sẻ để thao tác hiệu quả trên một số lượng lớn đối tượng “cỡ nhỏ” (chẳng hạn paragraph, dòng, cột, ký tự...)
<b>7</b>	<b>Proxy</b>	Cung cấp đối tượng đại diện cho một đối tượng khác để hỗ trợ hoặc kiểm soát quá trình truy xuất đối tượng đó. Đối tượng thay thế gọi là proxy

**b. Nhóm cấu thành (Creational Pattern):** Liên quan đến quá trình khởi tạo đối tượng cụ thể từ một định nghĩa trừu tượng (abstract class, interface). Khắc phục các vấn đề khởi tạo đối tượng, hạn chế sự phụ thuộc platform

STT	Tên	Mục đích
<b>1</b>	<b>Abstract Factory</b>	Cung cấp một interface cho việc tạo lập các đối tượng (có liên hệ với nhau) mà không cần qui định lớp khi hay xác định lớp cụ thể (concrete) tạo mỗi đối tượng
<b>2</b>	<b>Builder</b>	Tách rời việc xây dựng (construction) một đối tượng phức tạp khỏi biểu diễn của nó sao cho cùng một tiến trình xây dựng có thể tạo được các biểu diễn khác nhau
<b>3</b>	<b>Factory Method</b>	Định nghĩa Interface để sinh ra đối tượng nhưng để cho lớp con quyết định lớp nào được dùng để sinh ra đối tượng Factory method cho phép một lớp chuyển quá trình khởi tạo đối tượng cho lớp con
<b>4</b>	<b>Prototype</b>	Qui định loại của các đối tượng cần tạo bằng cách dùng một đối tượng mẫu, tạo mới nhờ vào sao chép đối tượng mẫu này.
<b>5</b>	<b>Singleton</b>	Đảm bảo 1 class chỉ có 1 instance và cung cấp 1 điểm truy xuất toàn cục đến nó

**c. Nhóm tương tác động (Behavioral Pattern):** Mô tả cách thức để các lớp hoặc đối tượng có thể giao tiếp với nhau.

Che dấu hiện thực của đối tượng, che dấu giải thuật , hỗ trợ việc thay đổi cấu hình đối tượng một cách linh động.

STT	Tên	Mục đích
1	<b>Chain of Responsibility</b>	Khắc phục việc ghép cặp giữa bộ gửi và bộ nhận thông điệp; Các đối tượng nhận thông điệp được kết nối thành một chuỗi và thông điệp được chuyển dọc theo chuỗi này đến khi gặp được đối tượng xử lý nó. Tránh việc gắn kết cứng giữa phần tử gửi request với phần tử nhận và xử lý request bằng cách cho phép hơn 1 đối tượng có cơ hội xử lý request . liên kết các đối tượng nhận request thành 1 dây chuyền rồi “pass” request xuyên qua từng đối tượng xử lý đến khi gặp đối tượng xử lý cụ thể.
2	<b>Command</b>	Mỗi yêu cầu (thực hiện một thao tác nào đó) được bao bọc thành một đối tượng. Các yêu cầu sẽ được lưu trữ và gọi đi như các đối tượng. Đóng gói request vào trong một Object , nhờ đó có thể nhúng số hoá chương trình nhận request và thực hiện các thao tác trên request: sắp xếp, log, undo...
3	<b>Interpreter</b>	Hỗ trợ việc định nghĩa biểu diễn văn phạm và bộ thông dịch cho một ngôn ngữ.
4	<b>Iterator</b>	Truy xuất các phần tử của đối tượng dạng tập hợp tuần tự (list, array, ...) mà không phụ thuộc vào biểu diễn bên trong của các phần tử.
5	<b>Mediator</b>	Định nghĩa một đối tượng để bao bọc việc giao tiếp giữa một số đối tượng với nhau.
6	<b>Memento</b>	Hiệu chỉnh và trả lại như cũ trạng thái bên trong của đối tượng mà vẫn không vi phạm việc bao bọc dữ liệu.
7	<b>Observer</b>	Định nghĩa sự phụ thuộc <i>một-nhiều</i> giữa các đối tượng sao cho khi một đối tượng thay đổi trạng thái thì tất cả các đối tượng phụ thuộc nó cũng thay đổi theo.
8	<b>State</b>	Cho phép một đối tượng thay đổi hành vi khi trạng thái bên trong của nó thay đổi , ta có cảm giác như

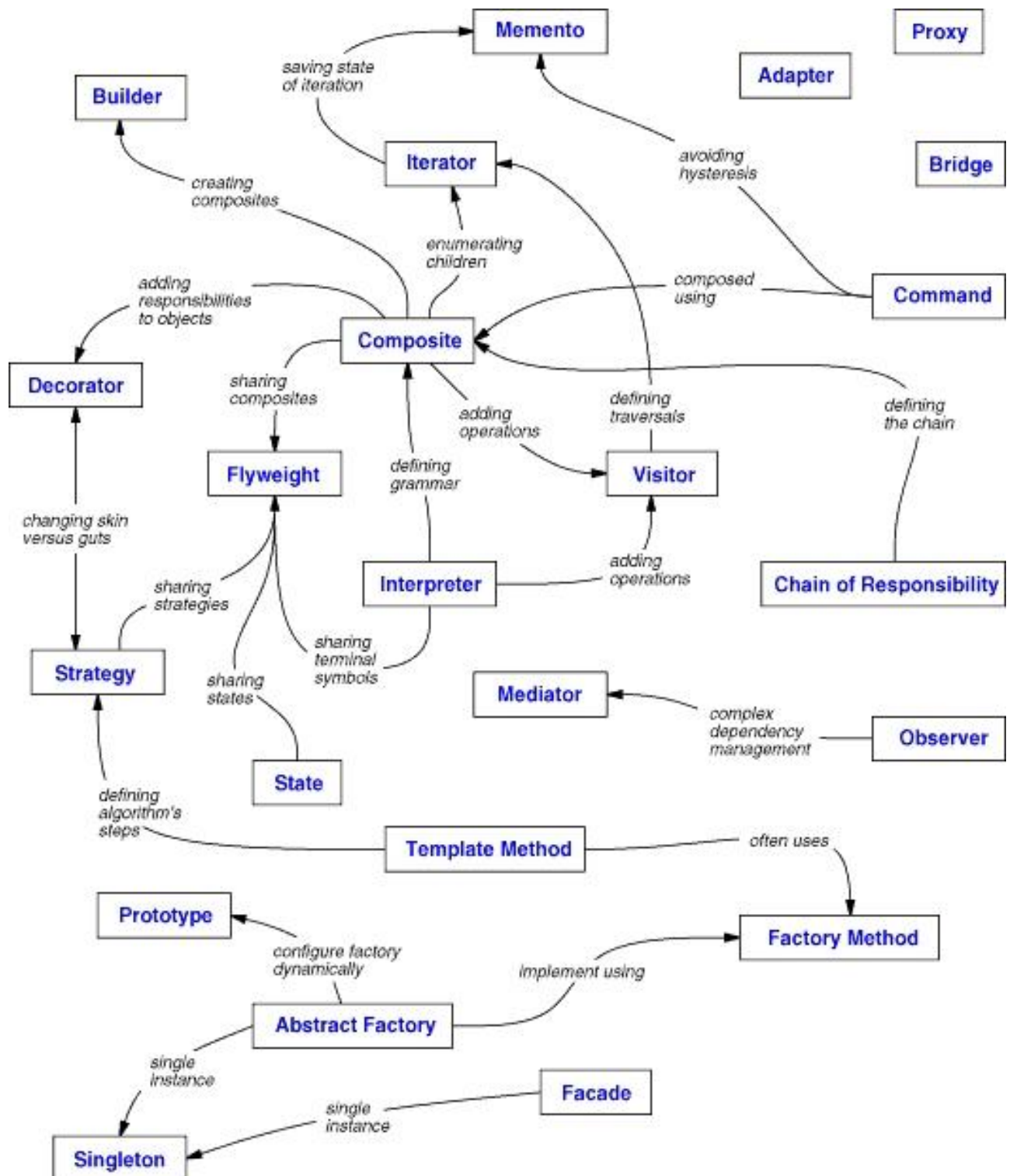
		class của đối tượng bị thay đổi
<b>9</b>	<b>Strategy</b>	Bao bọc một họ các thuật toán bằng các lớp đối tượng để thuật toán có thể thay đổi độc lập đối với chương trình sử dụng thuật toán. Cung cấp một họ giải thuật cho phép client chọn lựa linh động một giải thuật cụ thể khi sử dụng
<b>10</b>	<b>Template method</b>	Định nghĩa phần khung của một thuật toán, tức là một thuật toán tổng quát gọi đến một số phương thức chưa được cài đặt trong lớp cơ sở; việc cài đặt các phương thức được ủy nhiệm cho các lớp kế thừa.
<b>11</b>	<b>Visitor</b>	Cho phép định nghĩa thêm phép toán mới tác động lên các phần tử của một cấu trúc đối tượng mà không cần thay đổi các lớp định nghĩa cấu trúc đó.

## Design pattern catalog

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



## 2.4. Môi quan hệ giữa các Design Pattern



## 2.5. Structural Pattern

### 2.5.1. Adapter Pattern

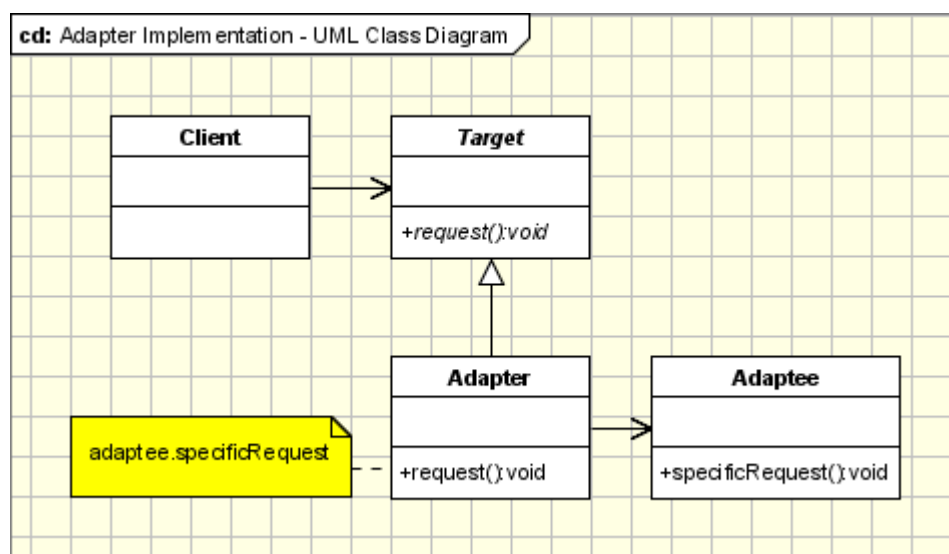
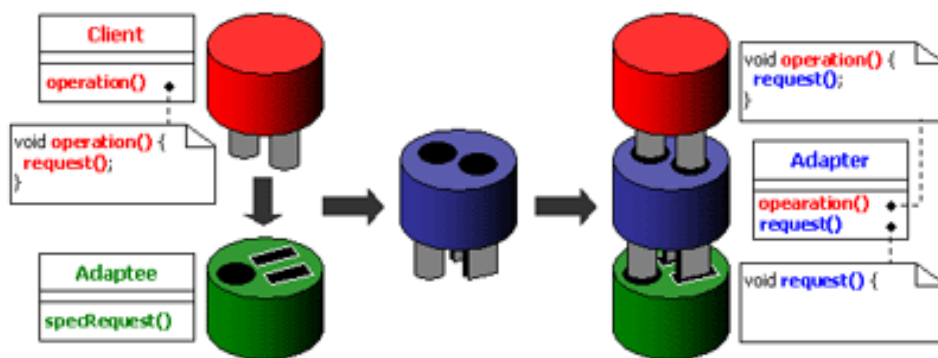
- Ý nghĩa

Cũng giống như Adapter trong thế giới thực, nó được sử dụng như một bộ biến điện, bộ ổn áp tăng hoặc giảm điện thế giúp thiết bị 110v sử dụng được nguồn điện 220v...

Trong lập trình, Adapter tạo thành một bộ chuyển đổi trung gian để cho các lớp làm việc cùng nhau do không tương thích với nhau về Interface.

Chuyển đổi interface của một lớp thành interface của một lớp khác như mong muốn.

- Cấu trúc mẫu



Trong đó:

Target là một interface định nghĩa chức năng, yêu cầu mà Client cần sử dụng

Adaptee định nghĩa một interface đã tồn tại sẵn

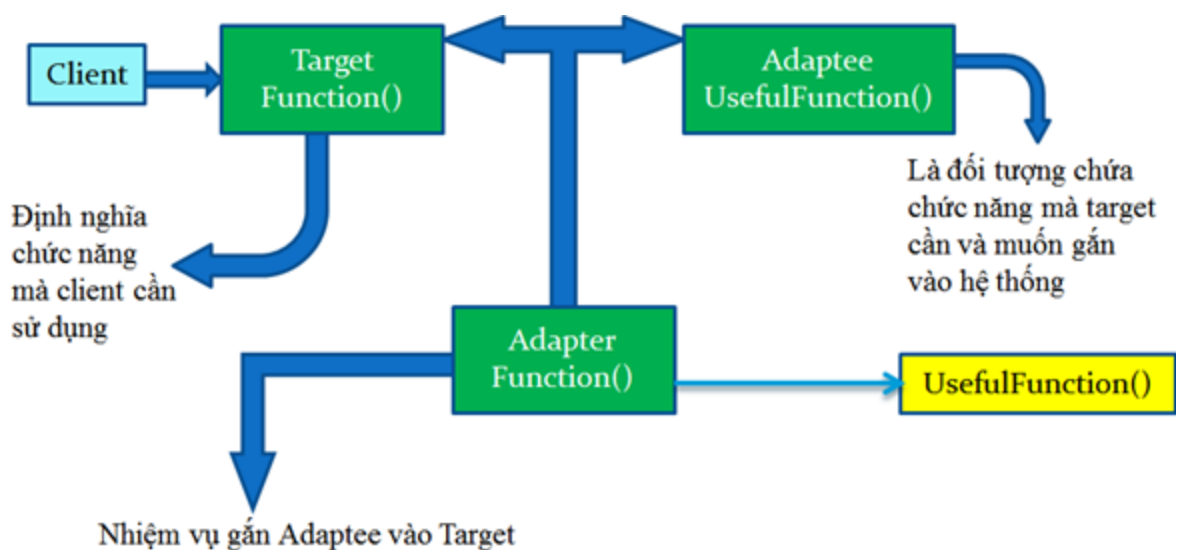
Adapter thực là cầu nối giữa Adaptee và Target, Adapter có nhiệm vụ gắn kết Adaptee vào Target để có được chức năng mà Client mong muốn.

Client cộng tác với những đối tượng phù hợp với Target interface

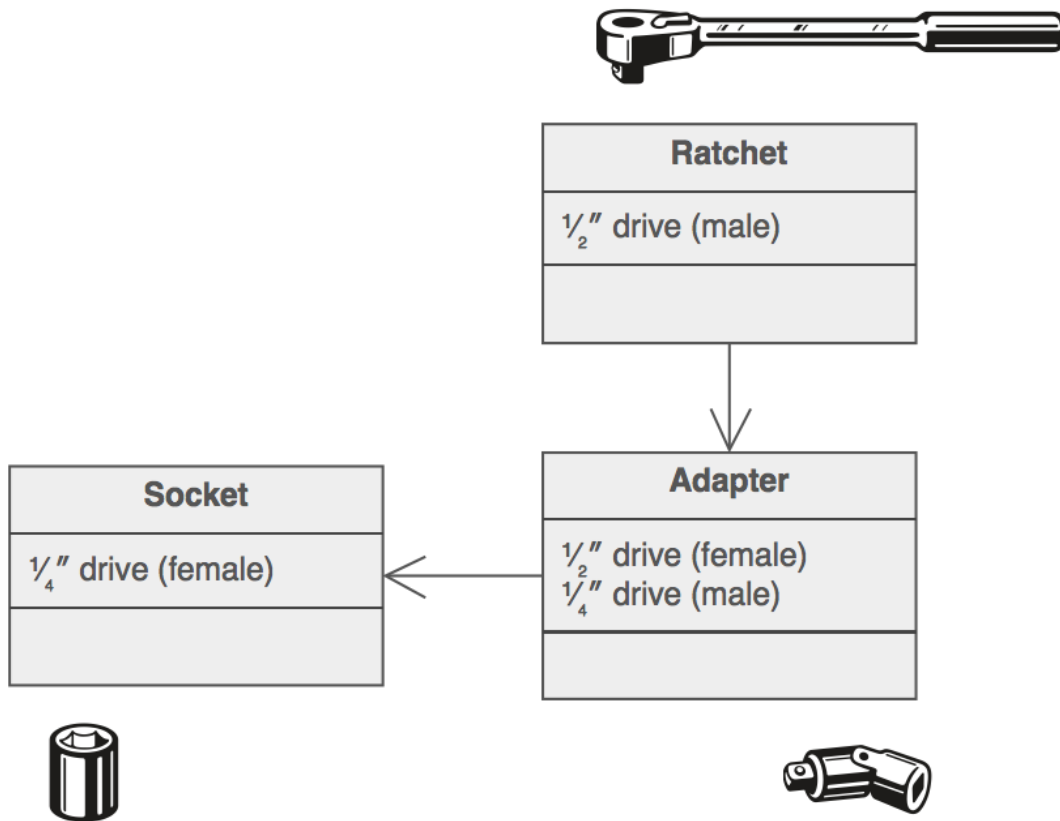
- Trường hợp ứng dụng

Muốn sử dụng 1 lớp có sẵn nhưng giao tiếp của nó không tương thích với yêu cầu hiện tại

Muốn tạo 1 lớp có thể sử dụng lại mà lớp này có thể làm việc được với những lớp khác không liên hệ gì với nó, và là những lớp không cần thiết tương thích trong giao diện.



Ví dụ:



Ví dụ 2:

Tạo một đối tượng là Device1, đây là một Interface với 2 hàm setName và getName

```
package adapterPattern;
```

```
public interface Device1_Interface {  
    void setName(String n);  
    String getName();  
}
```

Đối tượng Device1 sẽ được tạo ra từ lớp Device1\_Class, lớp này hiện thực giao diện trên

```
package adapterPattern;
```

```
public class Device1_Class implements Device1_Interface{  
    String name;
```

```

@Override
public void setName(String n) {
    name = n;
}

@Override
public String getName() {
    return name;
}
}
package adapterPattern;

```

Đây là tất cả những gì cần có trong thiết bị cũ. Tuy nhiên hiện tại, chúng ta đang muốn chuyển qua một thiết bị mới cho phù hợp với công việc hiện tại, nhưng không thể xóa bỏ hết cái cũ mà làm lại từ đầu vậy nên ta tạo đối tượng Device 2 với 4 hàm setCompanyName, setBranchName, getCompanyName, getBranchName. Đây là Device2\_Interface

```

public interface Device2_Interface {

    void setCompanyName(String c);
    void setBanchName(String b);
    String getCompanyName();
    String getBanchName();
}

```

Đối tượng Device2 được tạo từ lớp Device2\_Class, hiện thực Device2\_Interface như sau:

```

package adapterPattern;

public class Device2_Class implements Device2_Interface{

    String companyName;
    String branchName;

    @Override
    public void setCompanyName(String c) {
        companyName = c;
    }
}

```

```

@Override
public void setBanchName(String b) {
    branchName = b;
}
@Override
public String getCompanyName() {
    return companyName;
}
@Override
public String getBanchName() {
    return branchName;
}
}

```

Tới lúc này, bạn đã có đối tượng Device1, và đối tượng Device2. Bây giờ bạn cần một bộ chuyển đổi để gắn kết đối tượng Device1 vào đối tượng Device2.

```
package adapterPattern;
```

```
public class Device1ToDevice2_Adapter implements
Device2_Interface{
```

```

    Device1_Class device1Object;
    String companyName;
    String branchName;

```

```

public Device1ToDevice2_Adapter(Device1_Class d)
{
    device1Object = d;
    String[] split = device1Object.getName().split(" ");
    companyName = split[0];
    branchName = split[1];
}

```

```

@Override
public void setCompanyName(String c) {
    companyName = c;
}
@Override
public void setBanchName(String b) {
    branchName = b;
}

```

```

@Override
public String getCompanyName() {
    return companyName;
}

@Override
public String getBanchName() {
    return branchName;
}
}

```

Chạy thử bộ chuyển đổi bạn sẽ nhận được kết quả. Đó là những gì bạn mong muốn thông qua bộ chuyển đổi

```

package adapterPattern;
public class DemoAdapter {

    public static void main(String[] args) {

        Device1_Class device1Object = new Device1_Class();
        device1Object.setName("Samsung GalaxyS4");
        Device1ToDevice2_Adapter adapter = new
Device1ToDevice2_Adapter(device1Object);
        System.out.println("Company name: "+
adapter.getCompanyName());
        System.out.println("Branch name: "+
adapter.getBanchName());

    }
}

```

Kết quả:

```

Company name: Samsung
Branch name: GalaxyS4

```

## 2.5.2. Bridge Pattern

### - Ý nghĩa

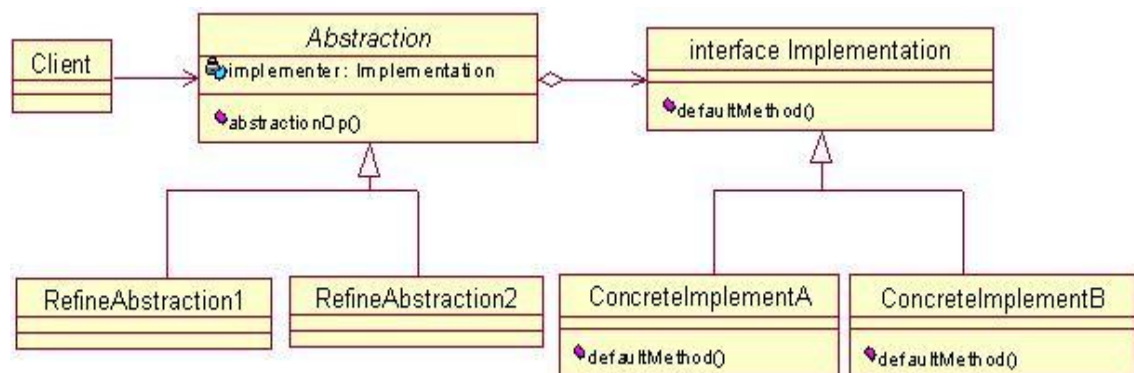
Một thành phần trong OOP thường có 2 phần:

Phần ảo – định nghĩa các chức năng

Phần thực thi – thực thi các chức năng được định nghĩa trong phần ảo. Hai phần này liên hệ với nhau qua quan hệ kế thừa. Những thay đổi trong phần ảo dẫn đến các thay đổi trong phần thực thi.

Mẫu Bridge được sử dụng để tách thành phần ảo và thành phần thực thi riêng biệt, do đó các thành phần này có thể thay đổi độc lập và linh động. Thay vì liên hệ với nhau bằng quan hệ kế thừa hai thành phần này liên hệ với nhau thông qua quan hệ “chứa trong”.

### - Cấu trúc mẫu



### Trong đó:

**Abstraction:** là lớp trừu tượng khai báo các chức năng và cấu trúc cơ bản, trong lớp này có 1 thuộc tính là 1 thể hiện của giao tiếp **Implementation**, thể hiện này bằng các phương thức của mình sẽ thực hiện các chức năng `abstractionOp()` của lớp **Abstraction**

**Implementation:** là giao tiếp thực thi của lớp các chức năng nào đó của **Abstraction**

**RefineAbstraction:** là định nghĩa các chức năng mới hoặc các chức năng đã có trong **Abstraction**.



ConcreteImplement: là các lớp định nghĩa tường minh các thực thi trong lớp giao tiếp Implementation

#### **- Trường hợp ứng dụng**

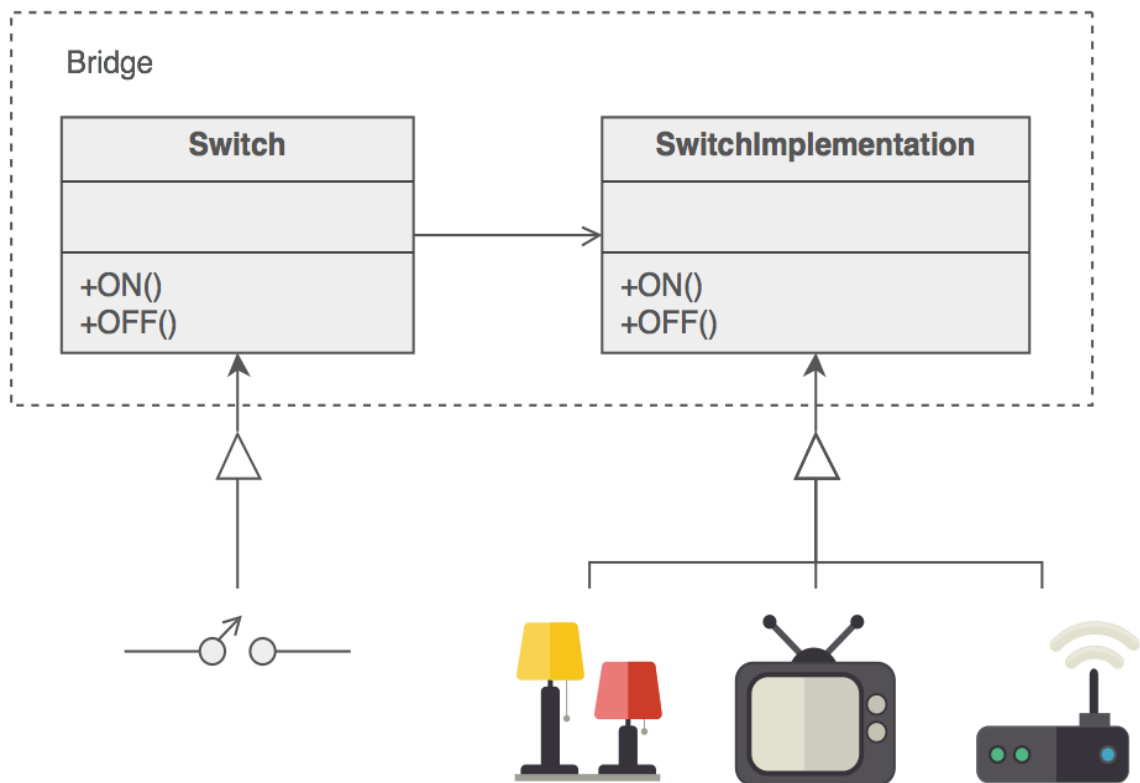
Khi bạn muốn tạo ra sự mềm dẻo giữa 2 thành phần ảo và thực thi của một thành phần, và tránh đi mối quan hệ tĩnh giữa chúng

Khi bạn muốn những thay đổi của phần thực thi sẽ không ảnh hưởng đến client

Bạn định nghĩa nhiều thành phần ảo và thực thi.

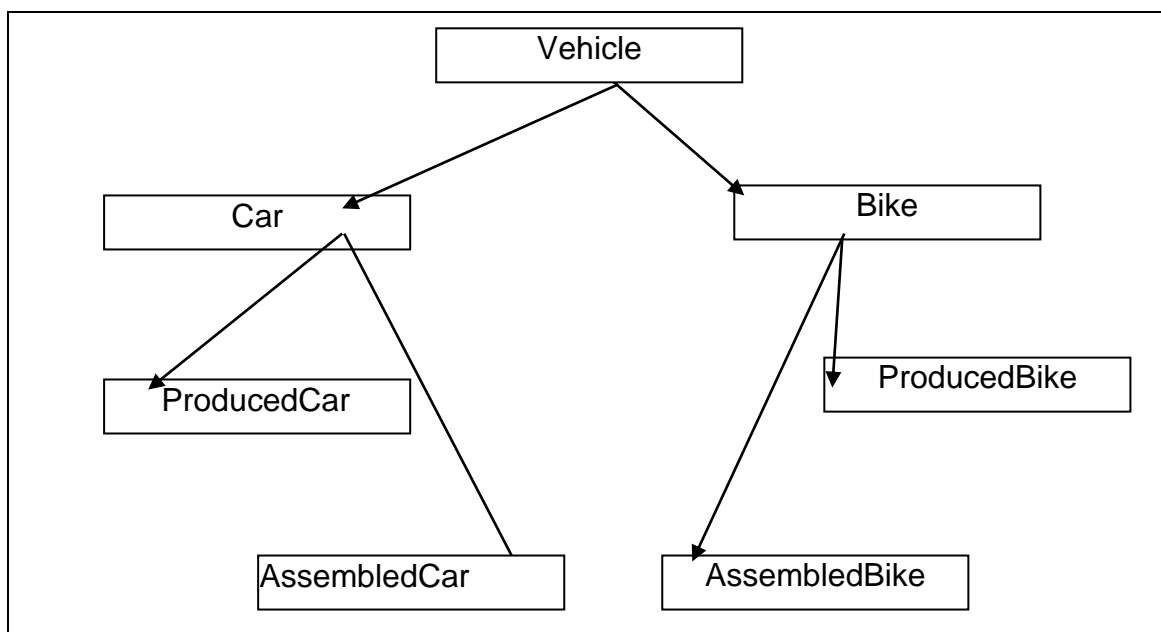
Phân lớp con một cách thích hợp, nhưng bạn muốn quản lý 2 thành phần của hệ thống một cách riêng biệt

Qua ví dụ sau bạn sẽ hiểu rõ hơn về Bridge Pattern, trong một hộ gia đình, vì mới đầu khi xây dựng thợ điện thiết kế cho gia đình chỉ có 1 bảng điện để chứa cả công tắc lẫn ổ điện. Ban đầu thiết kế đó được cho là hợp lý, tại là vợ chồng trẻ mới cưới nên nhu cầu sử dụng các đồ dùng điện trong nhà không có nhiều. Nhưng một thời gian sau, vì bận rộn với công việc nên 2 vợ chồng quyết định mua thêm máy giặt để giặt đồ, mua thêm tủ lạnh để đi siêu thị một ngày ăn nhiều ngày, 2 vợ chồng cũng cần 2 cái laptop để làm việc...như vậy là những thiết bị điện đã tăng lên đáng kể. Chưa tính đến là khi thiết kế chỉ lắp có một công tắc cho 2 bóng đèn ở phòng khách, bật thì sáng thật đấy nhưng nếu như không có khách mà bật cả 2 bóng cùng lúc thì thật là tốn kém...sự việc bắt đầu thấy có vẻ căng. Người chồng quyết định sẽ thay đổi hệ thống điện của gia đình, anh dùng dây điện kéo thêm vài bảng điện nằm ở nhiều vị trí khác nhau, mỗi bóng là một công tắc, mỗi vị trí lại có một ổ điện thuận tiện cho việc giặt đồ, chạy tủ lạnh, sử dụng laptop...và nhờ đó mà công việc được thuận tiện hơn

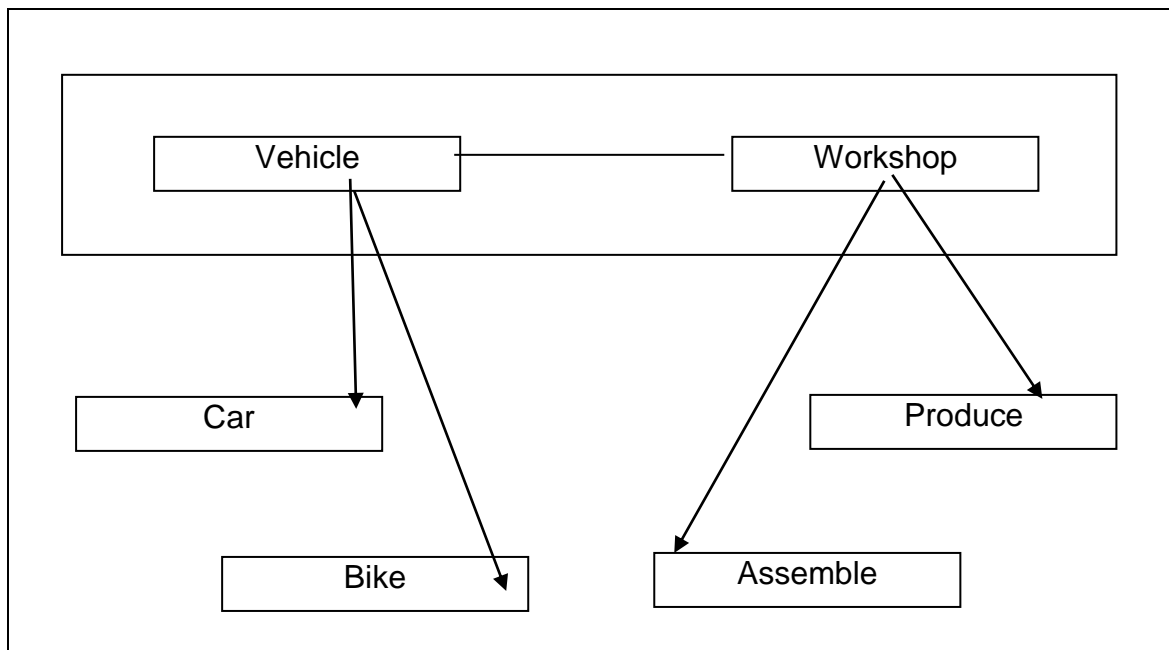


Ví dụ 2:

Trước khi sử dụng mẫu bridge



## Dùng mẫu Bridge



Đầu tiên tạo ra 2 phân cấp khác nhau, một là phân trừu tượng, hai là phân thực thi

```
package bridgePattern;

//Implementor for bridge pattern
public interface Workshop {
    public abstract void work();
}
```

Phân trừu tượng

```
package bridgePattern;

//abstraction in bridge pattern

public abstract class Vehicle {

    protected Workshop workShop1;
    protected Workshop workShop2;
```

```

        protected Vehicle(Workshop workShop1, Workshop
workShop2) {
            this.workShop1 = workShop1;
            this.workShop2 = workShop2;
        }

        public abstract void manufacture();
    }

```

Phần thực thi ta có 2 công việc là lắp ráp và sản xuất

```

package bridgePattern;

//Concrete implementation 2 for bridge pattern

public class Assemble implements Workshop{

    @Override
    public void work() {
        System.out.println("assembled ");
    }

}

```

Đây là công việc sản xuất

```

package bridgePattern;

//Concrete implementation 1 for bridge pattern

public class Produce implements Workshop{

    @Override
    public void work() {
        System.out.print("produced ");
    }

}

```

Cả 2 công việc này đều được mở rộng từ Workshop

Trong phần trừu tượng ta có 2 đối tượng đó là Car và Bike, Car và Bike đều được mở rộng từ Vehicle

```
package bridgePattern;
```

```
public class Car extends Vehicle{

    protected Car(Workshop workShop1, Workshop workShop2)
    {
        super(workShop1, workShop2);
        // TODO Auto-generated constructor stub
    }

    @Override
    public void manufacture() {
        System.out.print("Car ");
        workShop1.work();
        workShop2.work();
    }
}
```

```
package bridgePattern;
```

```
//Refine abstraction 2 in bridge pattern
```

```
public class Bike extends Vehicle{

    protected Bike(Workshop workShop1, Workshop workShop2)
    {
        super(workShop1, workShop2);
        // TODO Auto-generated constructor stub
    }

    @Override
    public void manufacture() {
        System.out.print("Bike ");
        workShop1.work();
        workShop2.work();
    }
}
```

Như vậy, mẫu Bridge được sử dụng để tách thành phần ảo và thành phần thực thi riêng biệt, do đó các thành phần này có thể thay đổi độc lập và linh động.

```
package bridgePattern;

public class DemoBridgePattern {

    public static void main(String[] args) {

        Vehicle vehicle1 = new Car(new Produce(), new
        Assemble());
        vehicle1.manufacture();
        Vehicle vehicle2 = new Bike(new Produce(), new
        Assemble());
        vehicle2.manufacture();

    }

}
```

**Output :**

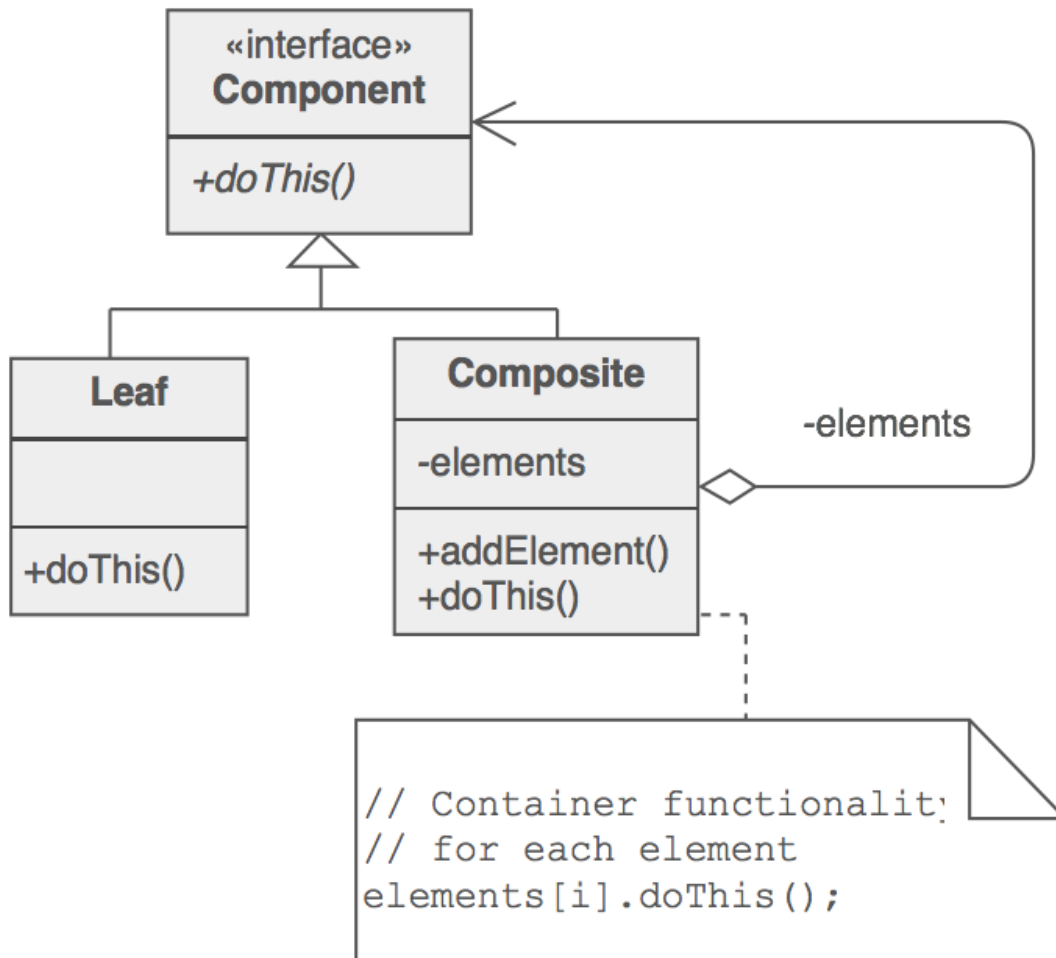
```
Car produced assembled
Bike produced assembled
```

### 2.5.3. Composite Pattern

#### - Ý nghĩa

Mẫu này nhằm gom các đối tượng vào trong một cấu trúc cây để thể hiện được cấu trúc tổng quát của nó. Trong khi đó cho phép mỗi phần tử của cấu trúc cây có thể thực hiện một chức năng theo một giao tiếp chung

#### - Mô hình mẫu



#### Trong đó:

**Component:** là một Interface định nghĩa các phương thức cho tất cả các phần của cấu trúc cây. Component có thể được thực thi như một lớp trừu tượng khi bạn cần cung cấp các hành vi cho tất cả các kiểu con. Bình thường, các Component không có các thể hiện, các lớp con hoặc các lớp thực thi của nó, gọi là các nốt, có thể có thể hiện và được sử dụng để tạo nên cấu trúc cây

Composite: là lớp được định nghĩa bởi các thành phần mà nó chứa. Composite chứa một nhóm động các Component, vì vậy nó có các phương thức để thêm vào hoặc loại bỏ các thể hiện của Component trong tập các Component của nó. Những phương thức được định nghĩa trong Component được thực thi để thực hiện các hành vi đặc tả cho lớp Composite và để gọi lại phương thức đó trong các nốt của nó. Lớp Composite được gọi là lớp nhánh hay lớp chứa

Leaf: là lớp thực thi từ Interface Component. Sự khác nhau giữa lớp Leaf và Composite là lớp Leaf không chứa các tham chiếu đến các Component khác, lớp Leaf đại diện cho mức thấp nhất của cấu trúc cây

#### **- Trường hợp ứng dụng**

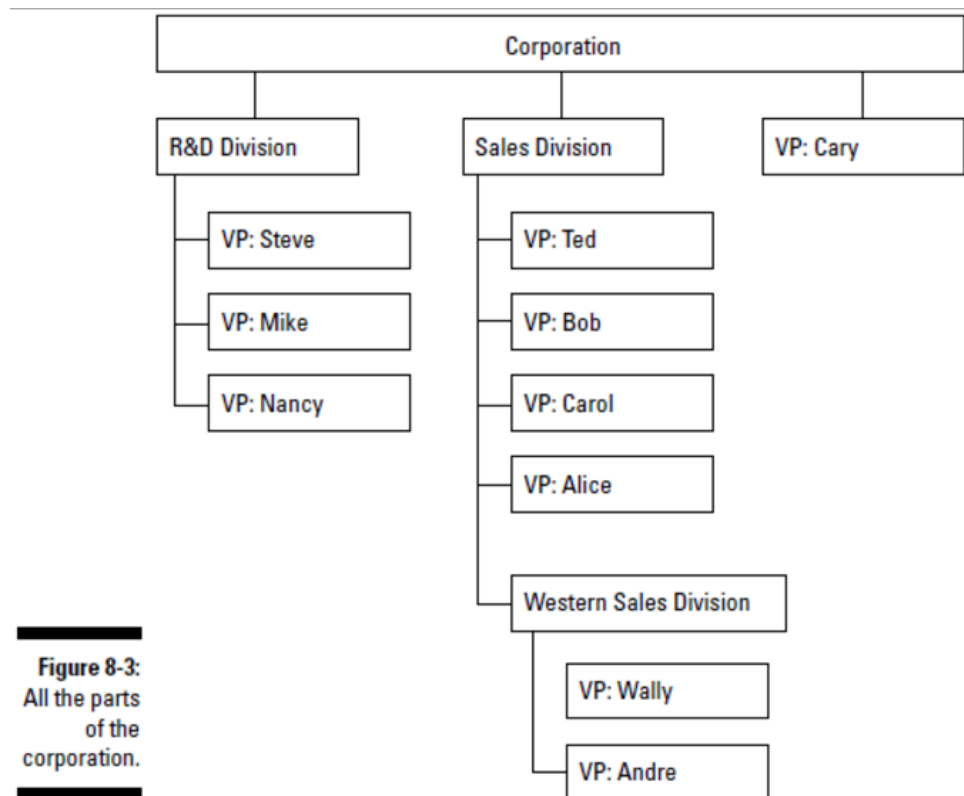
Khi có một mô hình thành phần với cấu trúc nhánh – lá, toàn bộ – bộ phận, ...

Khi cấu trúc có thể có vài mức phức tạp và động

Bạn muốn thăm cấu trúc thành phần theo một cách qui chuẩn, sử dụng các thao tác chung thông qua mối quan hệ kế thừa

**Ví dụ:** Nếu bạn là một thư ký, khi được xếp phân công cho việc tập hợp lại thông tin về nhân sự trong công ty, thì đó là việc làm thật khó khăn, nếu chỉ một hai phòng thôi thì họa may. Trong trường hợp này việc thu thập thông tin chính là một Composite và những tác vụ như thu tập thông tin lý lịch lương lậu là những Leaf



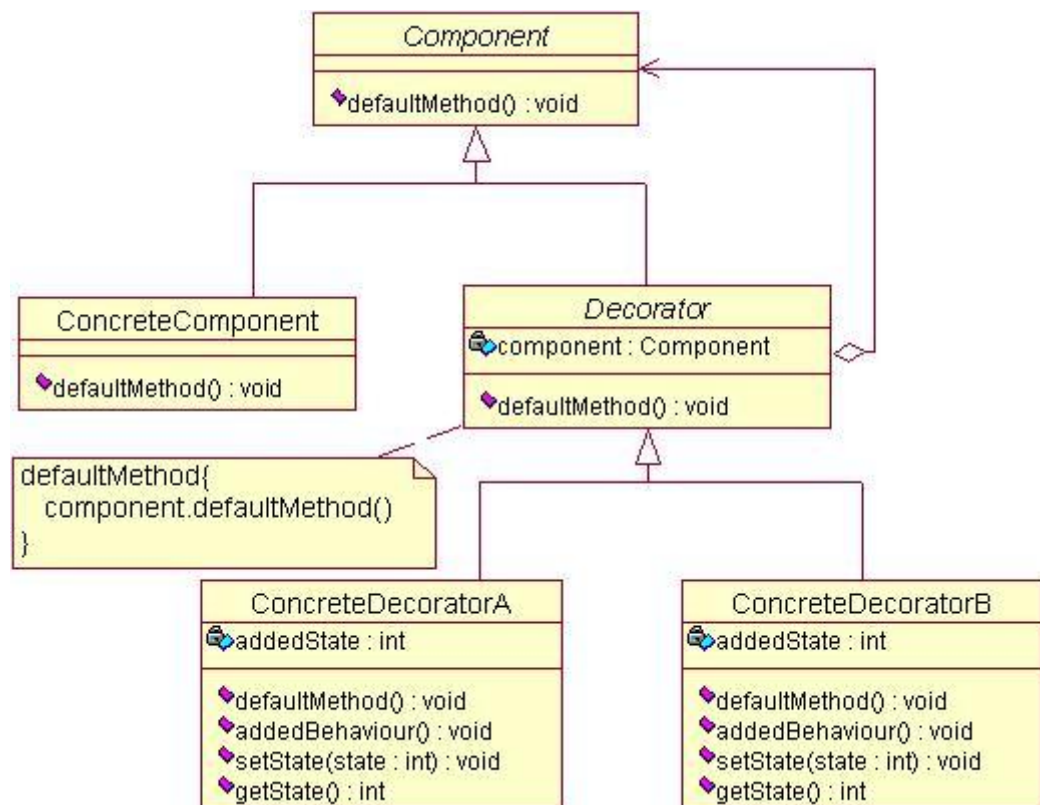


## 2.5.4. Decorator Pattern

### - Ý nghĩa

Bổ sung trách nhiệm cho đối tượng tại thời điểm thực thi. Đây được xem là sự thay thế hiệu quả cho phương pháp kế thừa trong việc bổ sung trách nhiệm cho đối tượng và mức tác động là ở mức đối tượng thay vì ở mức lớp như phương pháp kế thừa.

### - Mô hình mẫu



### Trong đó:

**Component:** là một interface chứa các phương thức ảo (ở đây là `defaultMethod`)

**ConcreteComponent:** là một lớp kế thừa từ **Component**, cài đặt các phương thức cụ thể (`defaultMethod` được cài đặt tường minh)

**Decorator:** là một lớp ảo kế thừa từ **Component** đồng thời cũng chứa 1 thể hiện của **Component**, phương thức `defaultMethod` trong **Decorator** sẽ được thực hiện thông qua thể hiện này.

ConcreteDecoratorX: là các lớp kế thừa từ Decorator, khai báo tường minh các phương thức, đặc biệt trong các lớp này khai báo tường minh các “trách nhiệm” cần thêm vào khi run-time

### - Trường hợp ứng dụng

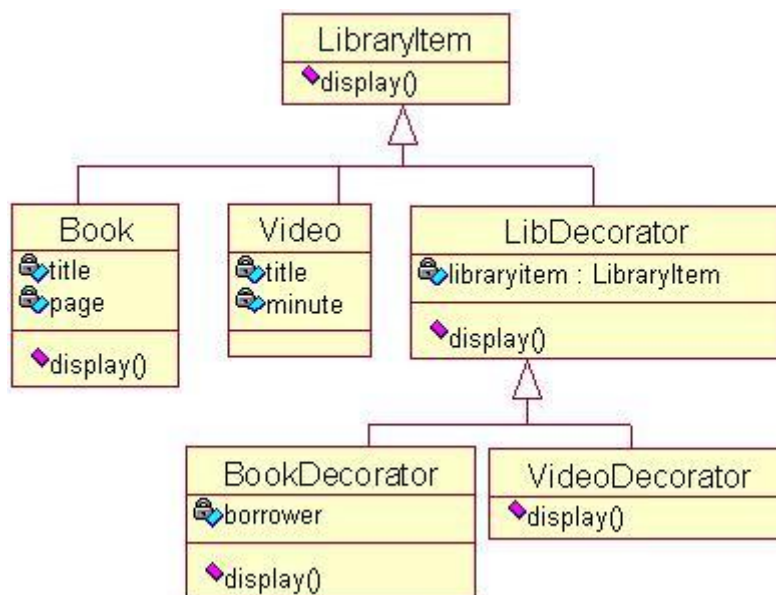
Khi bạn muốn thay đổi động mà không ảnh hưởng đến người dùng, không phụ thuộc vào giới hạn các lớp con

Khi bạn muốn thành phần có thể thêm vào hoặc rút bỏ đi khi hệ thống đang chạy

Có một số đặc tính phụ thuộc mà bạn muốn ứng dụng một cách động và bạn muốn kết hợp chúng vào trong một thành phần

### - Ví dụ

Giả sử trong thư viện có các tài liệu: sách, video... Các loại tài liệu này có các thuộc tính khác nhau, phương thức hiển thị của giao tiếp LibraryItem sẽ hiển thị các thông tin này. Giả sử , ngoài các thông tin thuộc tính trên, đôi khi ta muốn hiển thị độc giả mượn một cuốn sách (chức năng hiển thị độc giả này không phải lúc nào cũng muốn hiển thị), hoặc muốn xem đoạn video (không thường xuyên).



Giao tiếp LibraryItem định nghĩa phương thức display() cho tất cả các tài liệu của thư viện

```
public interface LibraryItem{
    public void display();           // đây là
    defaultMethod
}
```

Các lớp tài liệu

```
public class Book implements LibraryItem{
    private String title;
    private int page;
    public Book(String s, int p){
        title = s;
        page = p;
    }
    public void display(){
        System.out.println("Title: " + title);
        System.out.println("Page number: " +
        page);
    }
}

public class Video implements LibraryItem{
    private String title;
    private int minutes;
    public Video(String s, int m){
        title = s;
        minutes = m;
    }
    public void display(){
        System.out.println("Title: " + title);
        System.out.println("Time: " + minutes);
    }
}
```

Lớp ảo Decorator thư viện

```
public abstract class LibDecorator implements
LibraryItem{
    private LibraryItem libraryitem;
    public LibDecorator(LibraryItem li){
        libraryitem = li;
    }
    public void display(){
        libraryitem.display();
    }
}
```

Các lớp Decorator cho mỗi tài liệu thư viện cần bổ sung trách nhiệm ở thời điểm run-time

```

public class BookDecorator extends LibDecorator{
    private String borrower;
    public BookDecorator(LibraryItem li, String
b) {
        super(li);
        borrower = b;
    }
    public void display(){
        super.display();
        System.out.println("Borrower:      "      +
borrower);
    }
}
public class VideoDecorator extends LibDecorator{
    public VideoDecorator(LibraryItem li){
        super(li);
    }
    public void display(){
        super.display();
        play(); //phương thức play video
    }
}

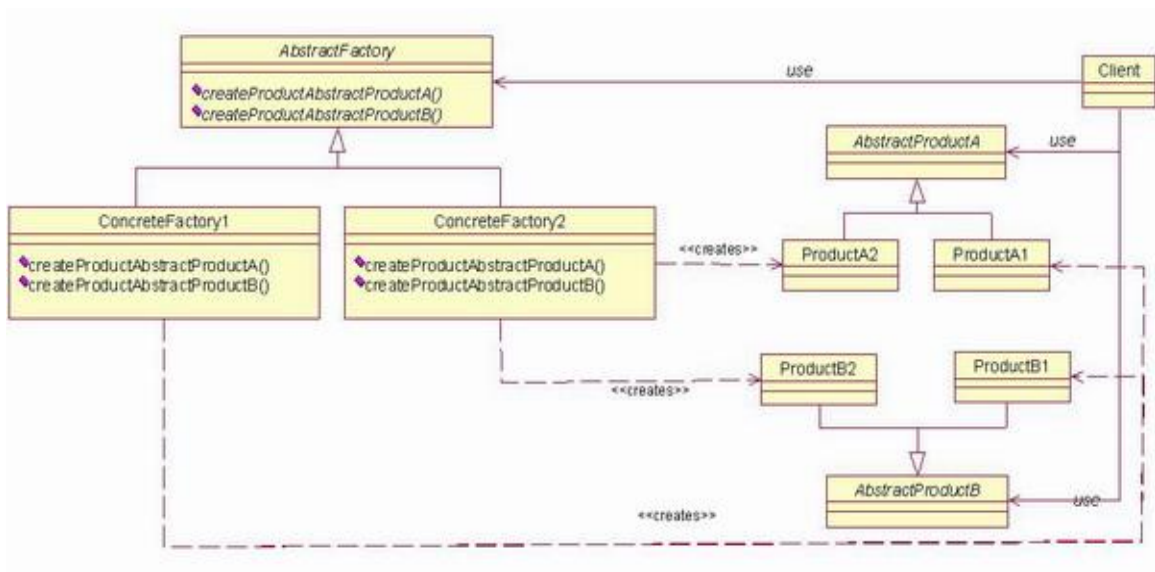
```

## 2.6. Creational patterns

### 2.6.1. Abstract Factory Method Pattern

- Ý nghĩa: Đóng gói một nhóm những lớp đóng vai trò “sản xuất” (Factory) trong ứng dụng, đây là những lớp được dùng để tạo lập các đối tượng. Các lớp sản xuất này có chung một giao diện lập trình được kế thừa từ một lớp cha thuận ảo gọi là “lớp sản xuất ảo”

- Cấu trúc mẫu



Trong đó:

**AbstractFactory**: là lớp trừu tượng, tạo ra các đối tượng thuộc 2 lớp trừu tượng là: **AbstractProductA** và **AbstractProductB**

**ConcreteFactoryX**: là lớp kế thừa từ **AbstractFactory**, lớp này sẽ tạo ra một đối tượng cụ thể

**AbstractProduct**: là các lớp trừu tượng, các đối tượng cụ thể sẽ là các thể hiện của các lớp dẫn xuất từ lớp này.

- Tình huống áp dụng

Phía trình khách sẽ không phụ thuộc vào việc những sản phẩm được tạo ra như thế nào.

Ứng dụng sẽ được cấu hình với một hoặc nhiều họ sản phẩm.

Các đối tượng cần phải được tạo ra như một tập hợp để có thể tương thích với nhau.

Chúng ta muốn cung cấp một tập các lớp và chúng ta muốn thể hiện các ràng buộc, các mối quan hệ giữa chúng mà không phải là các thực thi của chúng(interface).

- Ví dụ:

Tạo Interface Animal có phương thức là breathe

```
package abstractFactoryPattern;  
  
public interface Animal {  
    public void breathe();  
}
```

Tạo nhà máy trừu tượng của động vật AnimalFactory có phương thức sản xuất ra động vật createAnimal

```
package abstractFactoryPattern;  
  
public interface AnimalFactory {  
    public Animal createAnimal();  
}
```

Lớp kế thừa từ AnimalFactory, lớp này sẽ tạo ra một đối tượng cụ thể là Shark

```
package abstractFactoryPattern;  
  
public class SeaFactory implements AnimalFactory{  
  
    @Override  
    public Animal createAnimal() {  
        return new Shark();  
    }  
  
}
```

Lớp kế thừa từ AnimalFatory, lớp này sẽ tạo ra một đối tượng cụ thể là Monkey

```
package abstractFactoryPattern;

public class LandFactory implements AnimalFactory{

    @Override
    public Animal createAnimal() {
        return new Monkey();
    }

}
```

Từ Nhà máy SeaFactory nhờ phương thức createAnimal ta tạo ra được Shark

```
package abstractFactoryPattern;

public class Shark implements Animal {

    @Override
    public void breathe() {
        System.out.println("I breathe in water");
    }

}
```

Từ Nhà máy LandFactory nhờ phương thức createAnimal ta tạo ra được Monkey

```
package abstractFactoryPattern;

public class Monkey implements Animal {

    @Override
    public void breathe() {
        System.out.println("I breathe with my nose. this
year is Monkey year.kaka");
    }

}
```



```
package abstractFactoryPattern;
```

```
public class Wonderland {  
    public Wonderland(AnimalFactory factory){  
        Animal animal = factory.createAnimal();  
        animal.breathe();  
    }  
}
```

Test ta xem kết quả

```
package abstractFactoryPattern;
```

```
public class DemoAbstractFactory {  
  
    public static void main(String[] args) {  
        new Wonderland(createAnimalFactory("water"));  
    }  
  
    private static AnimalFactory  
createAnimalFactory(String string) {  
        if("water".equals(string))  
            return new SeaFactory();  
        else {  
            return new LandFactory();  
        }  
    }  
}
```

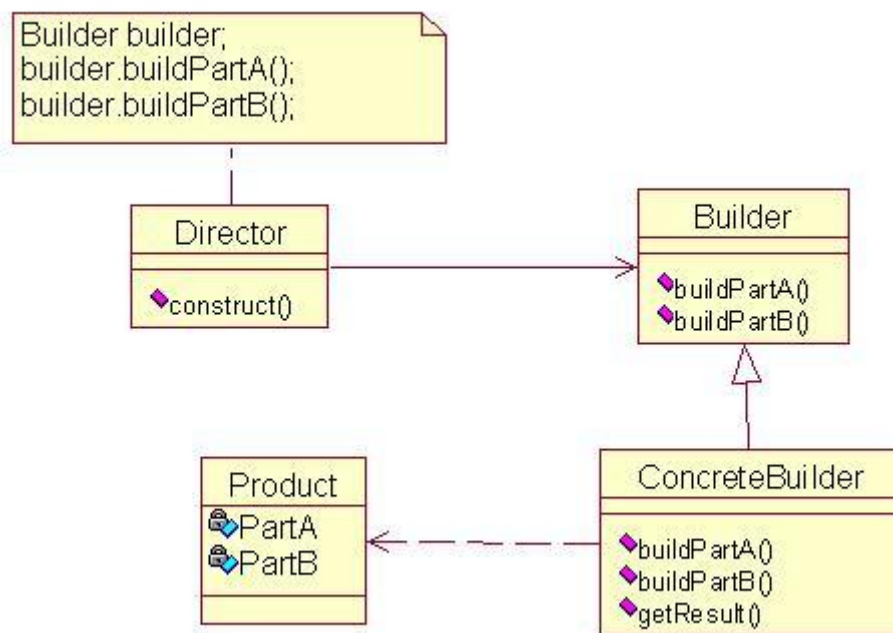
**Output:** I breathe in water

### 2.6.2. Builder Pattern

#### - Ý nghĩa

Phân tách những khởi tạo các thành phần của một đối tượng phức hợp, để có thể cùng một khởi tạo mà có thể tạo nên nhiều định dạng khác nhau.

#### - Cấu trúc mẫu



**Trong đó:** Director: là lớp điều khiển tạo ra một đối tượng Product

Builder: là lớp trừu tượng cho phép tạo ra đối tượng Product từ các phương thức nhỏ khởi tạo từng thành phần của Product

ConcreteBuilder: là lớp dẫn xuất của Builder, khởi tạo từng đối tượng cụ thể, lớp này sẽ khởi tạo đối tượng.

#### - Tình huống áp dụng

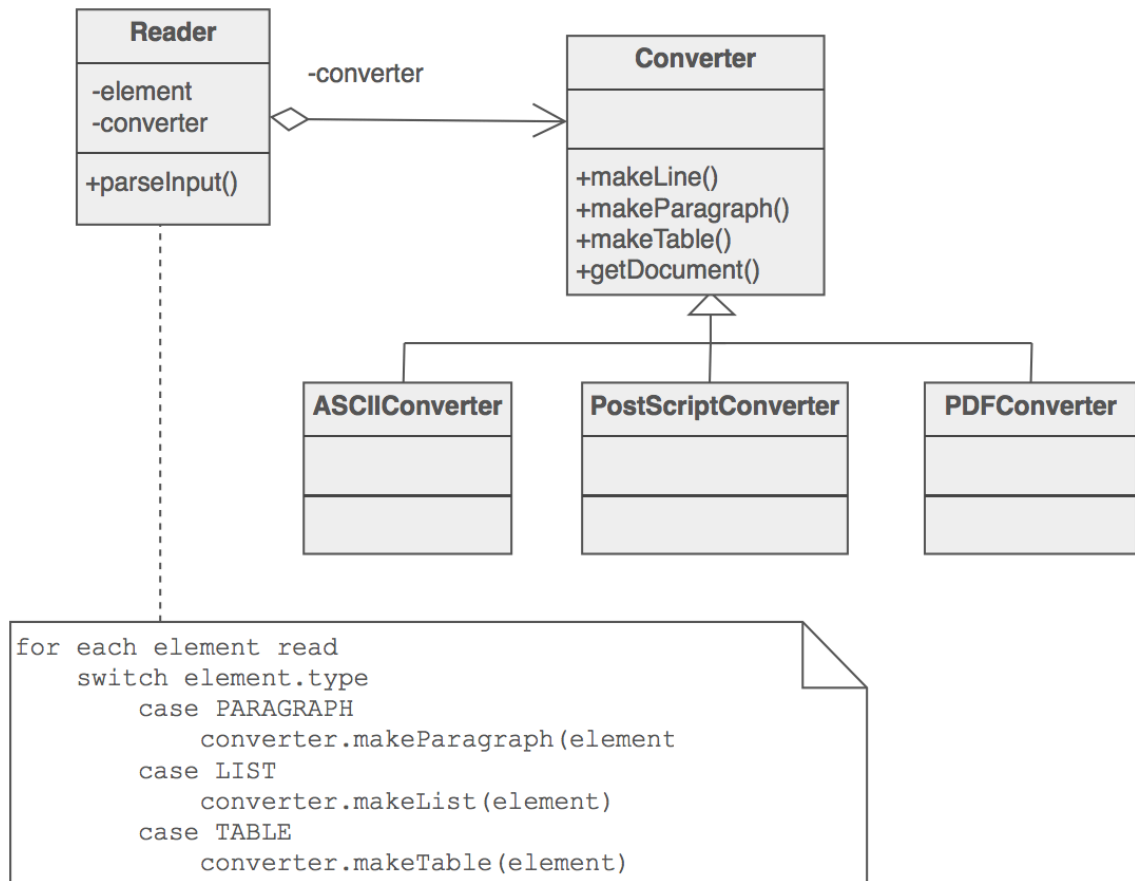
Có cấu trúc bên trong phức tạp

Có các thuộc tính phụ thuộc vào các thuộc tính khác

Sử dụng các đối tượng khác trong hệ thống mà có thể khó khởi tạo hoặc khởi tạo phức tạp

### Ví dụ:

Lớp Reader có cấu trúc phức tạp, nhiều thành phần. Nó bao gói nhiều phân tích mang tính chung chung của đầu vào. Vậy nên hệ thống phân cấp Builder sẽ thực hiện công việc phân tách ra nhiều đại diện đặc trưng cũng như mục đích khác nhau trong lớp Reader. Nhìn vào hình ví dụ dưới đây ta sẽ thấy và hiểu được nguyên lý hoạt động của Builder Pattern

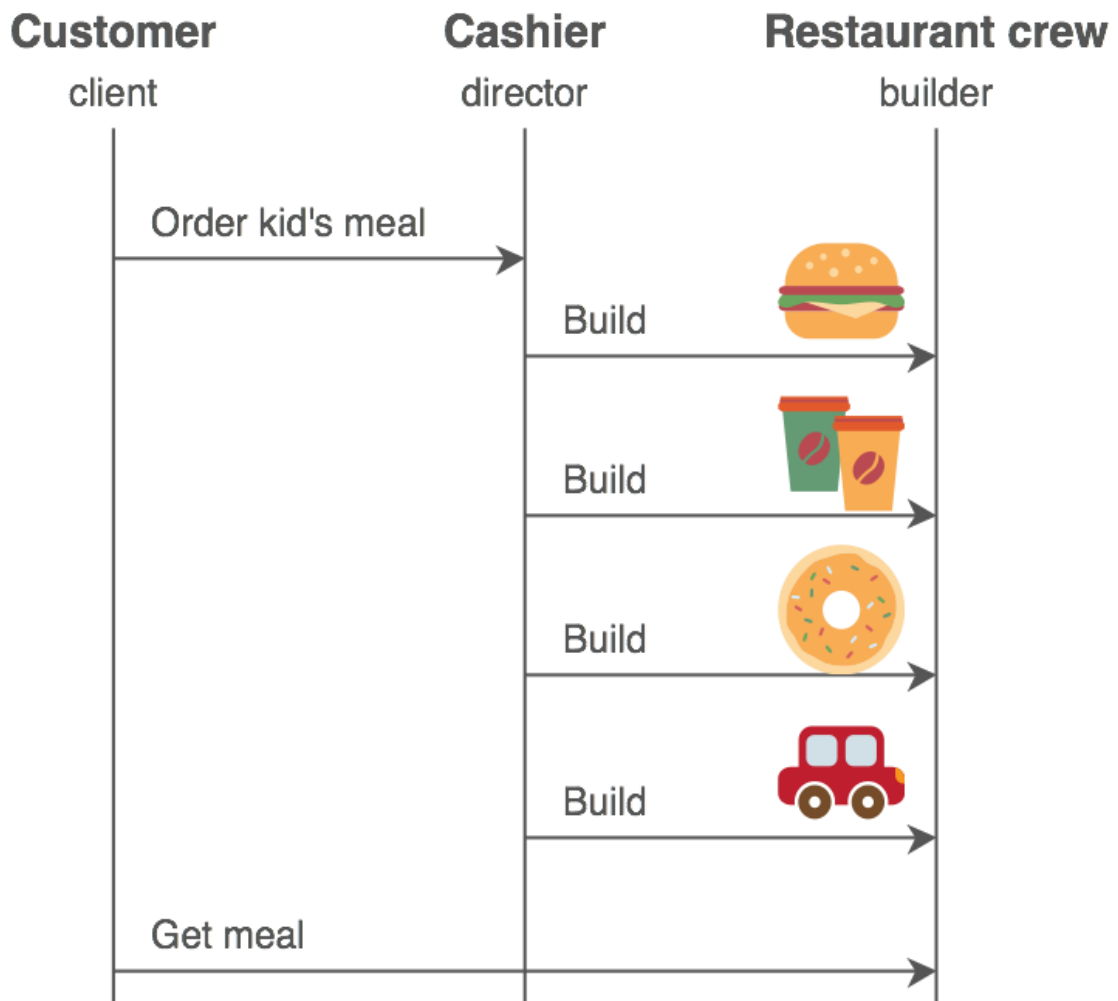


Và cuối cùng, sau khi Converter thực hiện thì ta đã phân tách được cái phức tạp thành cái đơn giản với những mục đích và đặc tính khác nhau.

Ngoài ra ta cũng có thể thấy mô hình này được sử dụng như một cửa hàng thức ăn nhanh cung cấp thức ăn cho các bé trong trường cấp 1.

Bữa ăn của các bé bao gồm một món chính, một món phụ, đồ uống và đồ chơi (ví dụ như một phần hamburger, khoai tây chiên, coca cola, và một chiếc xe đồ chơi). Sau khi phụ huynh order món cho con mình xong thì nhà hàng có nhiệm vụ chuẩn bị các phần thức ăn đó cho các bé. Thay vì tất

cả các món sẽ được đặt trong một cái khay trong đó có cả đồ ăn, cả thức uống, cả đồ chơi... thì nhân viên nhà hàng sẽ phân ra, món chính sẽ được đặt trong một hộp đựng đồ ăn, nước sẽ được đặt trong ly đựng nước và đồ chơi sẽ đưa cho em sau khi ăn xong...như vậy quá trình xây dựng thực đơn vẫn được diễn ra một cách trôi chảy mà việc sử dụng các món ăn sẽ dễ dàng và tiện gọn hơn rất nhiều



Ví dụ 2:

```
package builderPattern;
```

```
/* "Product" */  
public class MilkCoffee {  
    private String milk = "";  
    private String coffee = "";  
    private String ice = "";  
  
    public void setMilk(String milk){  
        this.milk = milk;  
    }  
    public void setCoffee(String coffee){  
        this.coffee = coffee;  
    }  
    public void setIce(String ice){  
        this.ice = ice;  
    }  
}
```

```
package builderPattern;
```

```
/* "Abstract Builder" */  
public abstract class MilkCoffeeBuilder {  
    protected MilkCoffee milkCoffee;  
  
    public MilkCoffee getMilkCoffee(){  
        return milkCoffee;  
    }  
    public void createNewMilkCoffeeProduct(){  
        milkCoffee = new MilkCoffee();  
    }  
  
    public abstract void buildMilk();  
    public abstract void builCoffee();  
    public abstract void buildIce();  
}
```

```

package builderPattern;

/* "ConcreteBuilder" */
public class G7CoffeeBuilder extends MilkCoffeeBuilder{

    @Override
    public void buildMilk() {
        milkCoffee.setMilk("Mr Thọ");
    }

    @Override
    public void builCoffee() {
        milkCoffee.setCoffee("G7 Coffee");
    }

    @Override
    public void buildIce() {
        milkCoffee.setIce("Mini Ice");
    }

}

```

```

package builderPattern;

/* "ConcreteBuilder" */
public class NetCafeBuilder extends MilkCoffeeBuilder{
    @Override
    public void buildMilk() {
        milkCoffee.setMilk("5 Stars");
    }

    @Override
    public void builCoffee() {
        milkCoffee.setCoffee("Net Cafe'");
    }

    @Override
    public void buildIce() {
        milkCoffee.setIce("Mini Ice");
    }

}

```

```

package builderPattern;

/* "Director" */
public class Waiter {
    private MilkCoffeeBuilder milkCoffeeBuilder;

    public void setMilkCoffeeBuilder(MilkCoffeeBuilder
builder){
        milkCoffeeBuilder = builder;
    }
    public MilkCoffee getMilkCoffee(){
        return milkCoffeeBuilder.getMilkCoffee();
    }
    public void constructMilkCoffee(){
        milkCoffeeBuilder.createNewMilkCoffeeProduct();
        milkCoffeeBuilder.builCoffee();
        milkCoffeeBuilder.buildIce();
        milkCoffeeBuilder.buildMilk();
    }
}

```

```

package builderPattern;

/* A customer ordering a cup of coffee. */
public class DemoBuilderPattern {

    public static void main(String[] args) {
        Waiter waiter = new Waiter();
        MilkCoffeeBuilder g7coffee = new G7CoffeeBuilder();
        MilkCoffeeBuilder netCafe = new NetCafeBuilder();

        waiter.setMilkCoffeeBuilder(g7coffee);
        waiter.setMilkCoffeeBuilder(netCafe);

        MilkCoffee coffee = waiter.getMilkCoffee();
    }
}

```

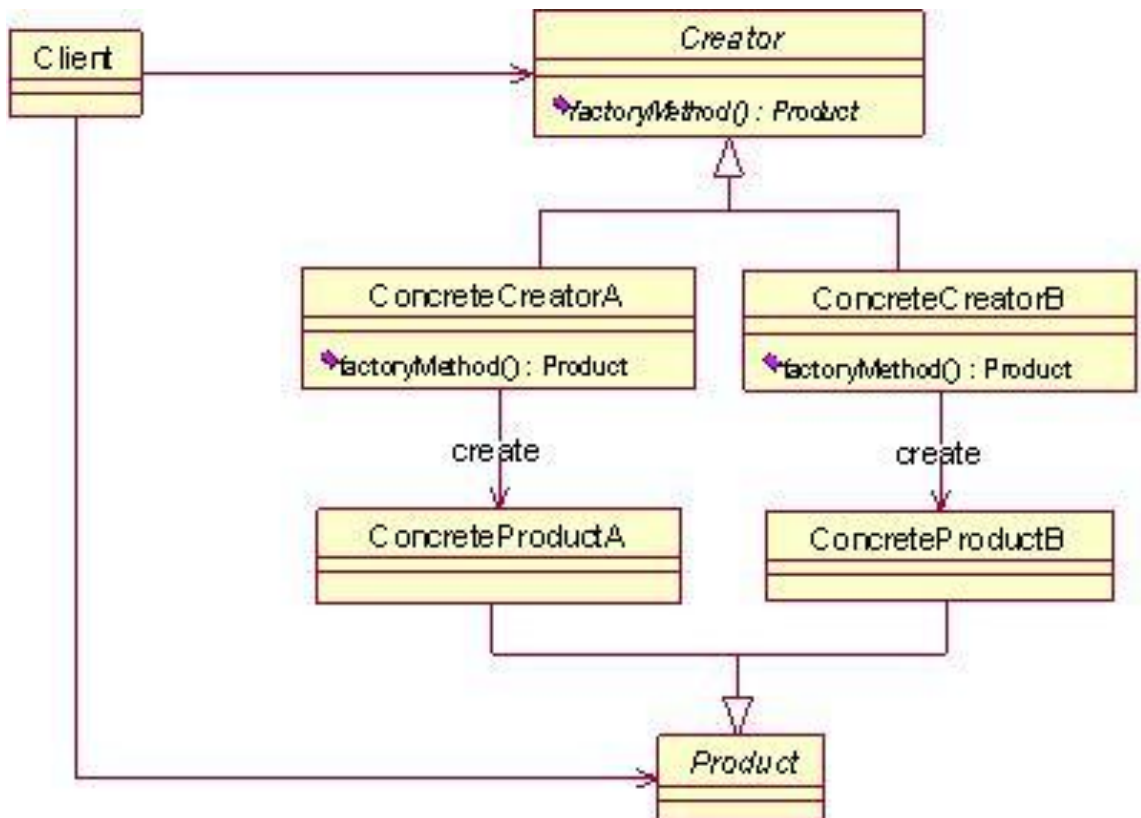
\* Abstract Factory và Builder đều được sử dụng để xây dựng một đối tượng phức tạp. Nhưng Builder tập trung nhấn mạnh vào từng bước một, mô hình có nhiều bước nhỏ. Như vậy đối tượng phát triển theo từng giai đoạn

### 2.6.3. Factory Method

#### - Ý nghĩa

Định nghĩa một phương thức chuẩn để khởi tạo đối tượng, như là một phần của phương thức tạo, nhưng việc quyết định kiểu đối tượng nào được tạo ra thì phụ thuộc vào các lớp con

#### - Cấu trúc mẫu



#### Trong đó:

**Creator** là lớp trừu tượng, khai báo phương thức `factoryMethod()` nhưng không cài đặt

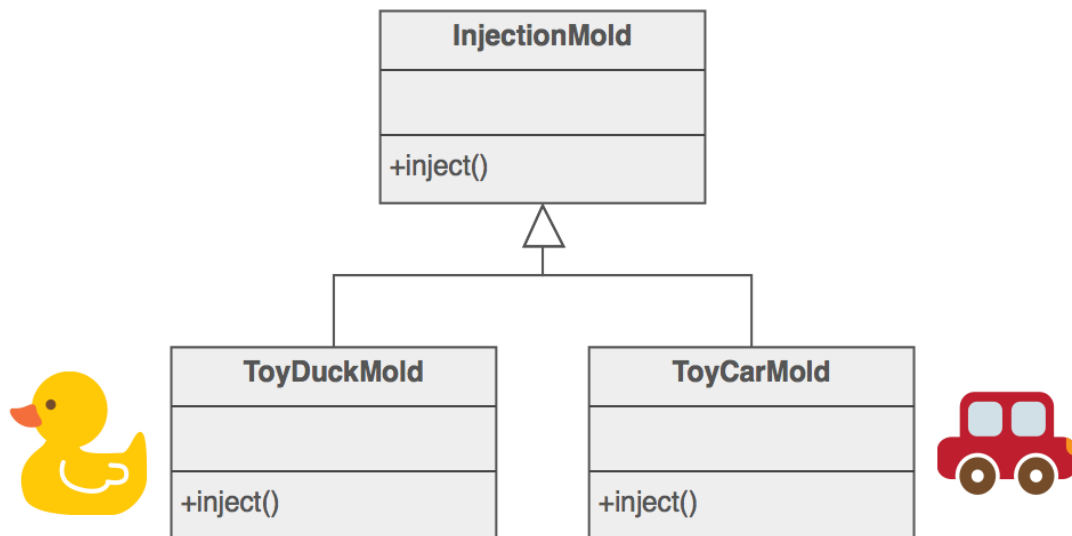
**Product** cũng là lớp trừu tượng

**ConcreteCreatorA** và **ConcreteCreatorB** là 2 lớp kế thừa từ lớp **Creator** để tạo ra các đối tượng riêng biệt

**ConcreteProductA** và **ConcreteProductB** là các lớp kế thừa của lớp **Product**, các đối tượng của 2 lớp này sẽ do 2 lớp **ConcreteCreatorA** và **ConcreteCreatorB** tạo ra



Ví dụ sau sẽ giúp bạn hiểu hơn về Factory Pattern



Đầu tiên sẽ định nghĩa một Interface cho việc tạo ra các đối tượng, nhưng cho phép các lớp con có quyền quyết định kiểu đối tượng nào được tạo ra, giống như một cái máy làm đồ chơi bằng nhựa. Nhân công sẽ dùng một phương thức chung đã được định nghĩa sẵn, đó là bơm nhựa đã được nung nóng chảy vào mỗi khuôn có hình dạng định trước như con vịt, con gà, cái xe, cái búa...như vậy tuy cùng một phương thức được định sẵn nhưng sản phẩm được tạo thành thì lại phụ thuộc vào khuôn. Đó chính là đặc tính của Factory Pattern.

Ví dụ 2:

Phương thức chuẩn

```
package factoryMethod;

public interface Pet {
    public String speak();
}
```

Lớp con thứ nhất

```
package factoryMethod;

public class Dog implements Pet{
```

```

@Override
public String speak() {
    return "Go Go...";
}
}

```

Lớp con thứ 2

```

package factoryMethod;

public class Cat implements Pet{

    @Override
    public String speak() {
        return "Meo...Meo...";
    }

}

```

Lớp Factory

```

package factoryMethod;

public class PetFactory {
    public Pet getPet(String petSpeak){
        Pet pet = null;

        if("Go".equals(petSpeak))
            pet = new Dog();
        else if ("Meo".equals(petSpeak)) {
            pet = new Cat();
        }
        return pet;
    }
}

```

Demo

```
package factoryMethod;

//using the factory method pattern
public class DemoFactoryMethod {

    public static void main(String[] args) {

        //creating the factory
        PetFactory petFactory = new PetFactory();

        //factory instantiates an object
        Pet pet = petFactory.getPet("Go");

        //you don't know which object factory created
        System.out.println(pet.speak());

    }
}
```

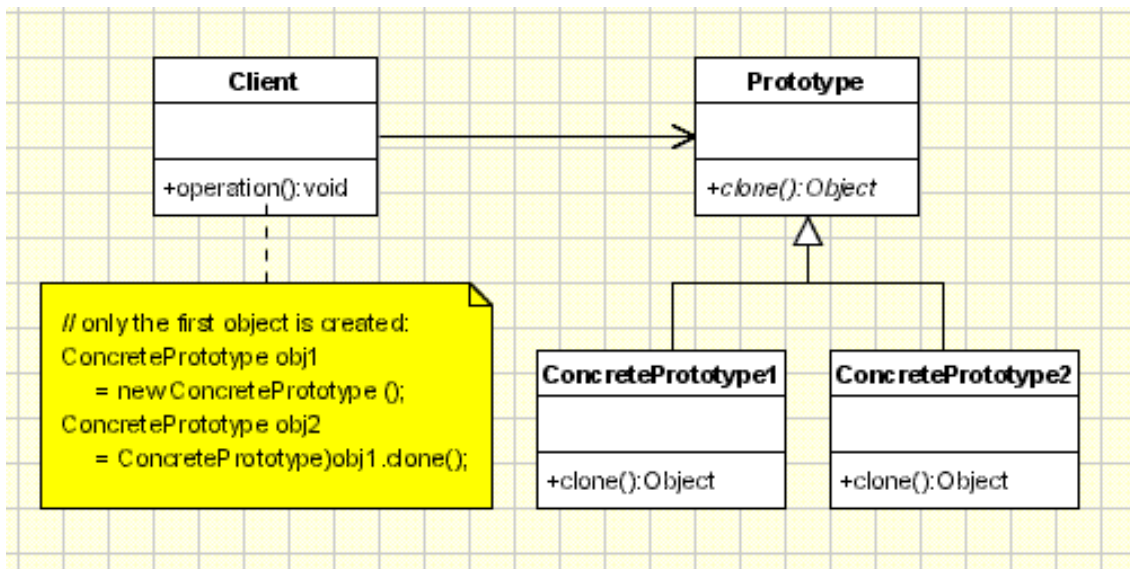
Output: Go Go...

## 2.6.4. Prototype

### - Ý nghĩa

Giúp khởi tạo đối tượng bằng cách copy một đối tượng khác đã tồn tại (đối tượng này là “prototype” – nguyên mẫu).

### - Cấu trúc mẫu



Trong đó:

Client tạo ra một đối tượng mới bằng cách yêu cầu 1 prototype copy lại chính bản thân cái đã có sẵn

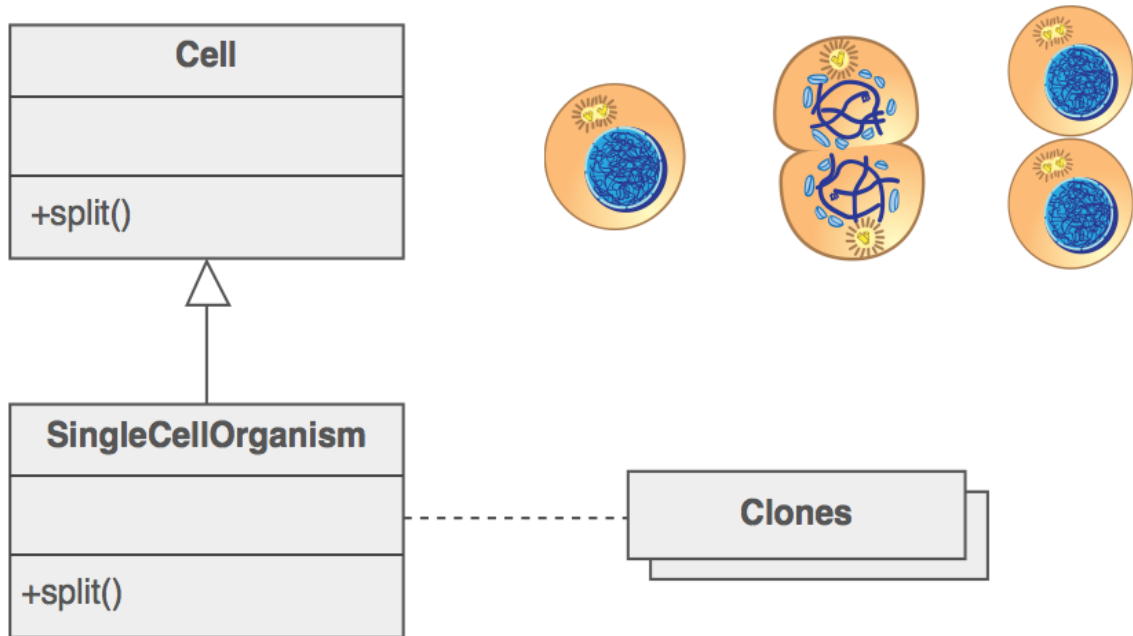
Prototype là lớp trừu tượng cài đặt phương thức Clone() là phương thức copy bản thân đối tượng đã tồn tại.

ConcretePrototype1 và ConcretePrototype2 là các lớp kế thừa lớp Prototype.

### - Tình huống áp dụng

Khi bạn muốn khởi tạo một đối tượng bằng cách sao chép từ một đối tượng đã tồn tại

Ví dụ về việc phân chia nguyên phân của một tế bào - kết quả là hai tế bào giống hệt nhau – đó là một ví dụ về Prototype, nó đóng một vai trò tích cực trong việc sao chép chính nó.



Ví dụ 2:

```
package prototypePattern;

public class Bike implements Cloneable{
    private int price;
    private String bikeType;
    private String model;
    public Bike() {
        bikeType = "Standard";
        model = "Leopard";
        price = 4;
    }

    public Bike clone(){
        return new Bike();
    }

    public void makeAdvanced(){
        bikeType = "Advanced";
        model = "Sport";
        price = 6;
    }
}
```

```

    public String getModel(){
        return model;
    }

    public String getBikeType(){
        return bikeType;
    }

    public int getPrice(){
        return price;
    }
}

```

```

package prototypePattern;

```

```

public class Workshop {

    public Bike makeSport(Bike basicBike){
        basicBike.makeAdvanced();
        return basicBike;
    }

    public static void main(String[] args) {
        Bike bike = new Bike();
        Bike basicBike = bike.clone();
        Workshop workshop = new Workshop();
        Bike advancedBike = workshop.makeSport(basicBike);
        System.out.println("Prototype Design Pattern : "
            + advancedBike.getModel()+ " / "
            + advancedBike.getBikeType()+ " / "
            + advancedBike.getPrice());
    }

}

```

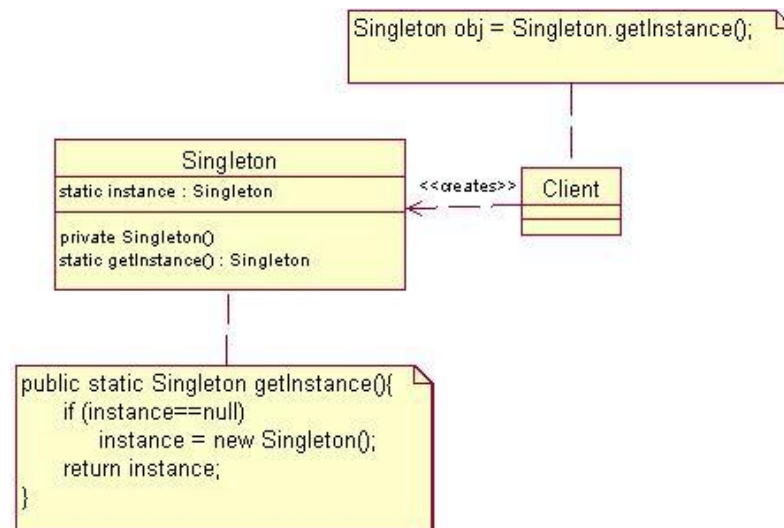
Output: Prototype Design Pattern : Sport / Advanced / 6

### 2.6.5. Singleton

#### - Ý nghĩa

Mẫu này được thiết kế để đảm bảo cho một lớp chỉ có thể tạo ra duy nhất một thể hiện của nó.

#### - Cấu trúc mẫu



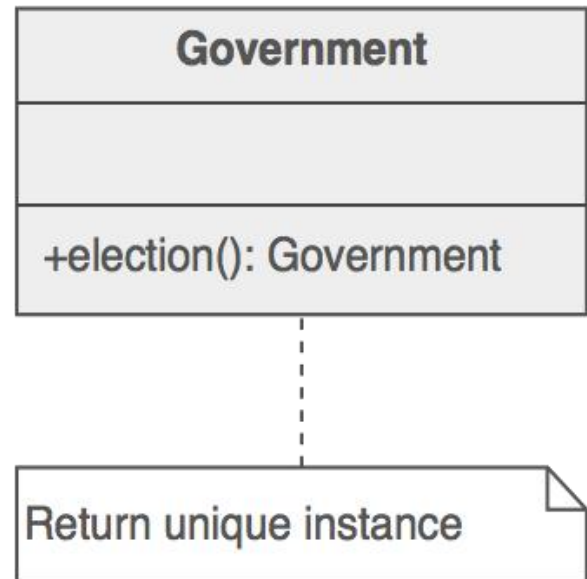
Trong đó:

Singleton cung cấp một phương thức tạo private, duy trì một thuộc tính tĩnh để tham chiếu đến một thể hiện của lớp Singleton này, và nó cung cấp thêm một phương thức tĩnh trả về thuộc tính tĩnh này

#### - Tình huống áp dụng

Khi bạn muốn lớp chỉ có 1 thể hiện duy nhất và nó có hiệu lực ở mọi nơi

Ví dụ sau sẽ cho bạn thấy một cái nhìn tổng quan hơn



Mẫu Singleton được định nghĩa là một tập hợp có chứa một phần tử. Bạn lấy văn phòng của Tổng Thống Hoa Kỳ ra làm một ví dụ cho Singleton. Hiến pháp Hoa kỳ quy định kết quả bầu cử chỉ được lấy ra một tổng thống. Tổng thống sẽ được ở tại Nhà Trắng. Như vậy có thể có duy nhất một tổng thống đang hoạt động và làm việc ở bất kỳ thời điểm nào.

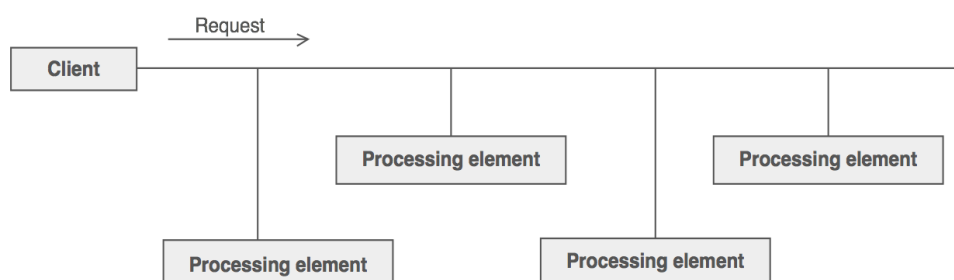
## 2.7. Behavioral patterns.

### 2.7.1. Chain of Responsibility

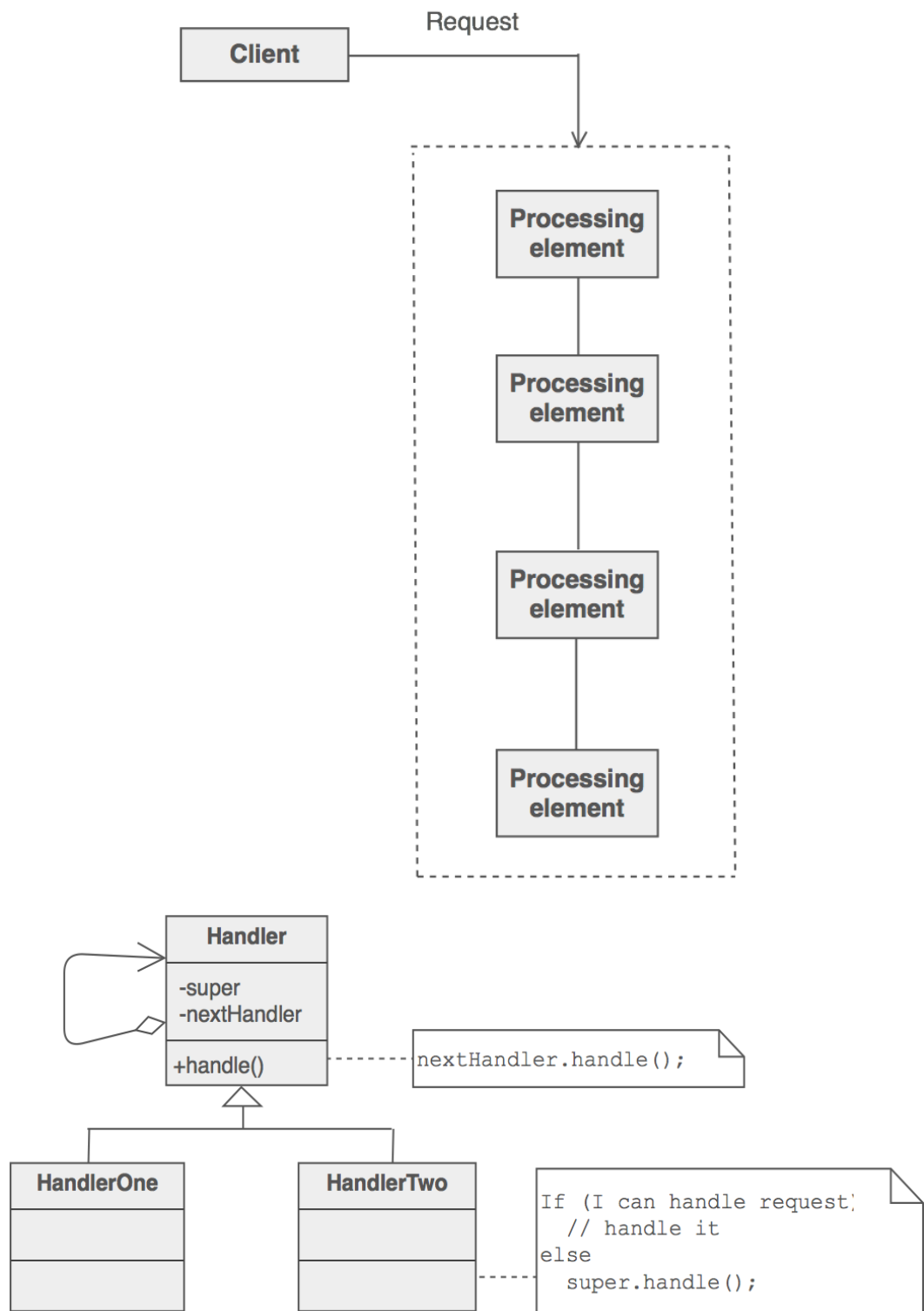
#### - Ý nghĩa

Mẫu này thiết lập một chuỗi bên trong một hệ thống, nơi mà các thông điệp hoặc có thể được thực hiện ở tại một mức, nơi mà nó được nhận lần đầu hoặc là được chuyển đến một đối tượng mà có thể thực hiện điều đó

#### - Mô hình mẫu







### Trong đó:

**Handler:** là một giao tiếp định nghĩa phương thức sử dụng để chuyển thông điệp qua các lần thực hiện tiếp theo.

**ConcreteHandler:** là một thực thi của giao tiếp **Handler**. Nó giữ một tham chiếu đến một **Handler** tiếp theo. Việc thực thi phương thức

handleMessage có thể xác định làm thế nào để thực hiện phương thức và gọi một handlerMethod, chuyển tiếp thông điệp đến cho Handler tiếp theo hoặc kết hợp cả hai

#### - Trường hợp ứng dụng

Có một nhóm các đối tượng trong một hệ thống có thể đáp ứng tất cả các loại thông điệp giống nhau

Các thông điệp phải được thực hiện bởi một vài các đối tượng trong hệ thống

Các thông điệp đi theo mô hình “thực hiện – chuyển tiếp”, một vài sự kiện có thể được thực hiện tại mức mà chúng được nhận hoặc tại ra, trong khi số khác phải được chuyển tiếp đến một vài đối tượng khác

Ví dụ: Việc rút tiền của máy ATM sẽ tuần tự, giải quyết các request của khách hàng

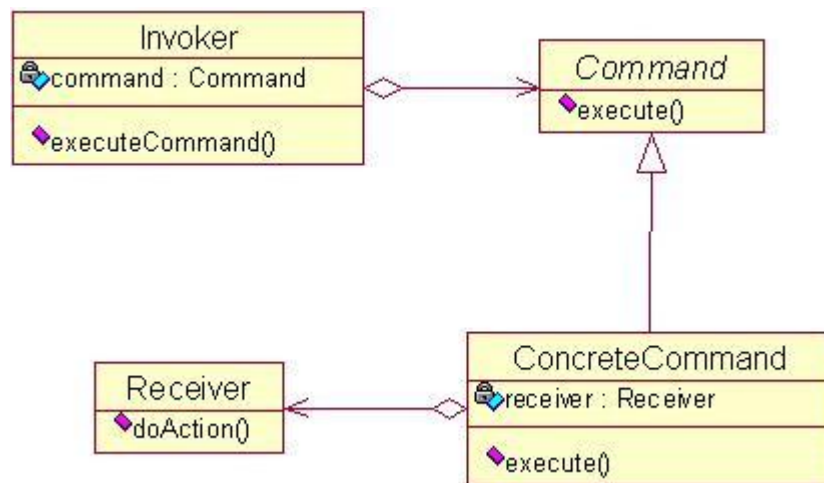


## 2.7.2. Command Pattern

### - Ý nghĩa

Gói một mệnh lệnh vào trong một đối tượng mà nó có thể được lưu trữ, chuyển vào các phương thức và trả về một vài đối tượng khác

### - Cấu trúc mẫu



### Trong đó:

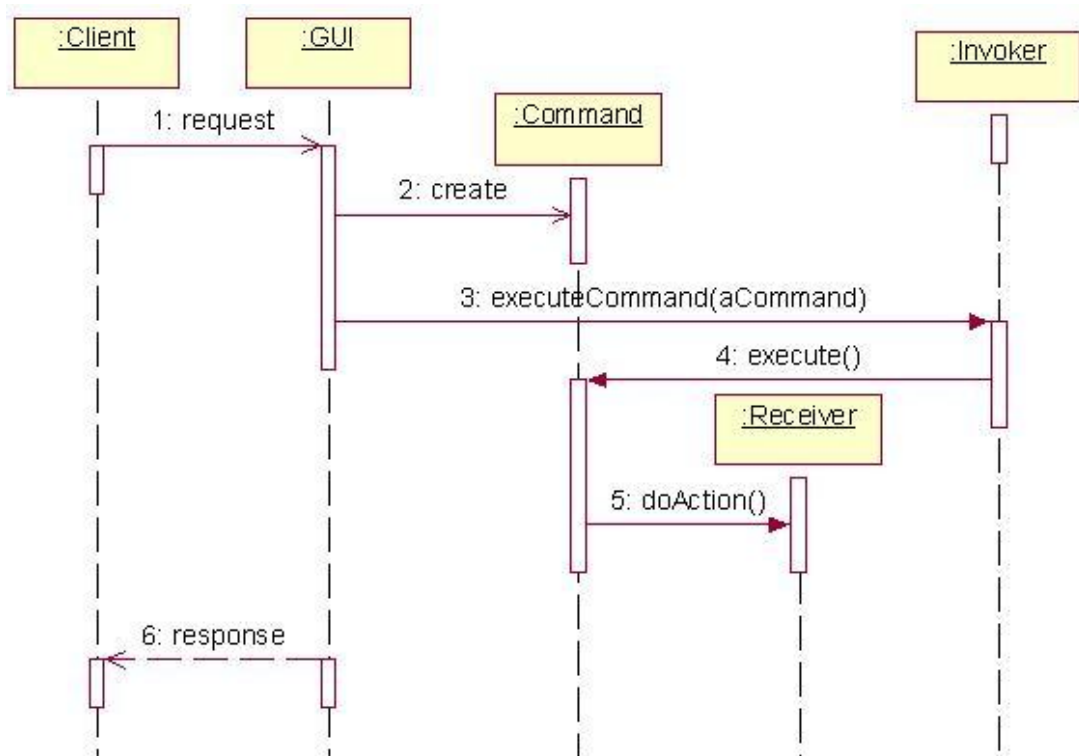
**Command:** là một giao tiếp định nghĩa các phương thức cho **Invoker** sử dụng

**Invoker:** lớp này thực hiện các phương thức của đối tượng **Command**

**Receiver:** là đích đến của **Command** và là đối tượng thực hiện hoàn tất yêu cầu, nó có tất cả các thông tin cần thiết để thực hiện điều này

**ConcreteCommand:** là một thực thi của giao tiếp **Command**. Nó lưu giữ một tham chiếu **Receiver** mong muốn

Luồng thực thi của mẫu **Command** như sau:



Client gửi yêu cầu đến GUI của ứng dụng

Ứng dụng khởi tạo một đối tượng Command thích hợp cho yêu cầu đó (đối tượng này sẽ là các ConcreteCommand)

Sau đó ứng dụng gọi phương thức executeCommand() với tham số là đối tượng Command vừa khởi tạo

Invoker khi được gọi thông qua phương thức executeCommand() sẽ thực hiện gọi phương thức execute() của đối tượng Command tham số

Đối tượng Command này sẽ gọi tiếp phương thức doAction() của thành phần Receiver của nó, được khởi tạo từ đầu, doAction() chính là phương thức chính để hoàn tất yêu cầu của Client

### - Trường hợp áp dụng

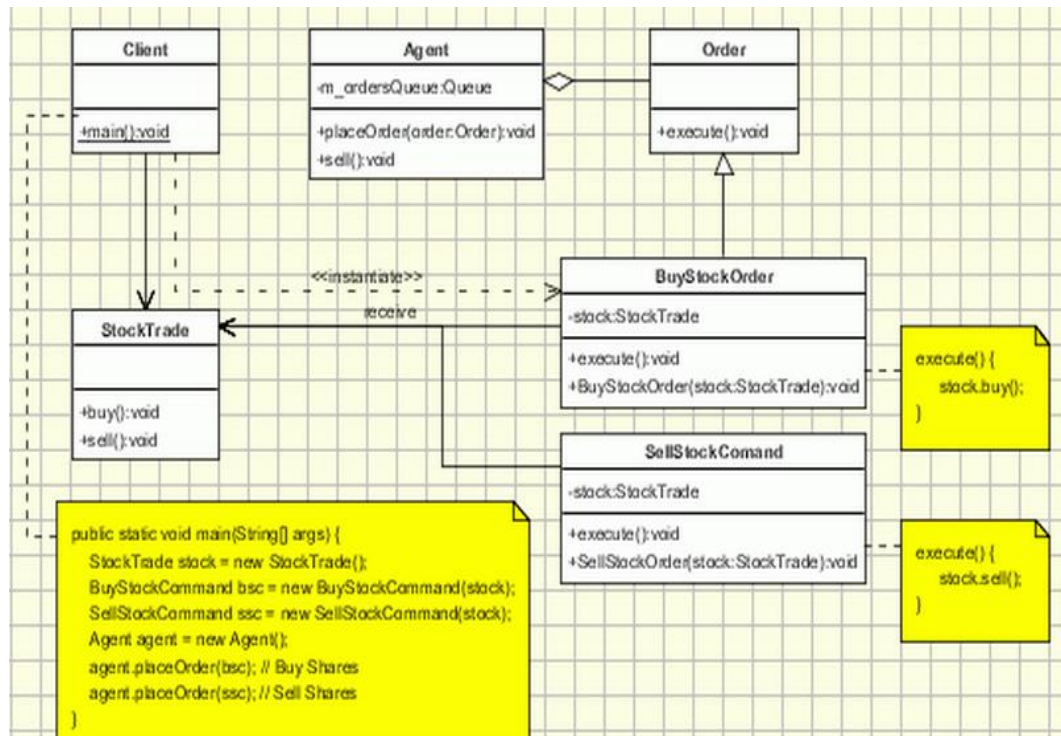
Hỗ trợ undo, logging hoặc transaction

Thực hiện hàng đợi lệnh và thực hiện lệnh tại các thời điểm khác nhau

Hạn chế sự chặt chẽ của yêu cầu với đối tượng thực hiện hoàn tất yêu cầu

### - Ví dụ mẫu

```
public interface Command{
    public void execute();
}
public class ConcreteCommand implements Command{
    private Receiver receiver;
    public void setReceiver(Receiver receiver){
        this.receiver = receiver;
    }
    public Receiver getReceiver(){
        return this.receiver;
    }
    public void execute (){
        receiver.doAction();
    }
}
public class Receiver{
    private String name;
    public Receiver(String name){
        this.name = name;
    }
    public void doAction(){
        System.out.print(this.name + " fulfill
request!");
    }
}
public class Invoker{
    public void executeCommand(Command command){
        command.execute();
    }
}
public class Run{
    public static void main(String[] args){
        Command command = new ConcreteCommand();
        command.setReceiver(new
Receiver("NguyenD"));
        Invoker invoker = new Invoker();
        Invoker.executeCommand(command);
    }
}
```



Vi dụ về mua bán trên cơ phiếu trên thị trường chứng khoán

```

//Command
public interface Order {
    public abstract void execute();
}

// Receiver
class StockTrade {
    public void buy() {
        Console.WriteLine("You want to buy stocks");
    }
    public void sell() {
        Console.WriteLine("You want to sell stocks ");
    }
}

// Invoker
class Agent {
    private ArrayList ordersQueue = new ArrayList();
    public Agent() {
    }
    public void placeOrder(Order order)
    {
        ordersQueue.Add(order);
    }
}
  
```

```

        // proccess
        ((Order)ordersQueue[0]).execute();
        //Log, undo
        ordersQueue.RemoveAt(0);
    }
}

//ConcreteCommand
class BuyStockOrder : Order {
    private StockTrade stock;
    public BuyStockOrder(StockTrade st) {
        stock = st;
    }
    public void execute() {
        stock.buy();
    }
}

//ConcreteCommand
class SellStockOrder : Order {
    private StockTrade stock;
    public SellStockOrder(StockTrade st) {
        stock = st;
    }
    public void execute() {
        stock.sell();
    }
}

// Client
class Program
{
    public static void main(String[] args) {
        StockTrade stock = new StockTrade();
        BuyStockOrder bsc = new BuyStockOrder(stock);
        SellStockOrder ssc = new SellStockOrder(stock);
        Agent agent = new Agent();
        agent.placeOrder(bsc);
        agent.placeOrder(ssc);
    }
}

```

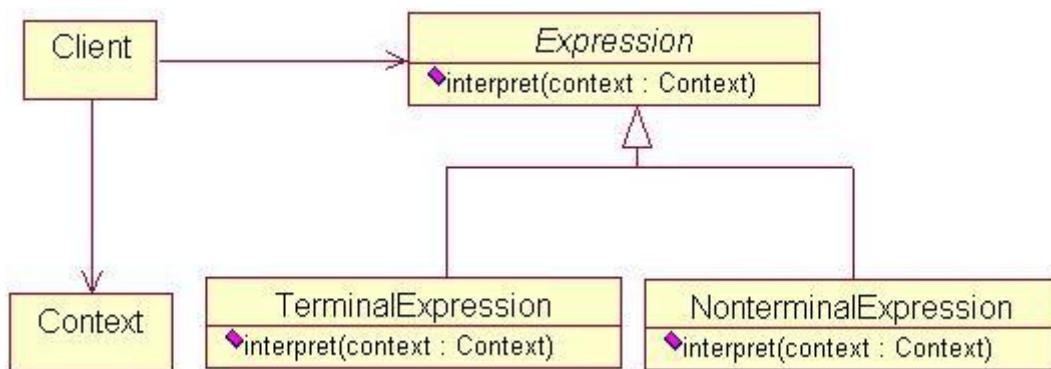
### 2.7.3. Interpreter Pattern

#### - Ý nghĩa

Ý tưởng chính của Interpreter là triển khai ngôn ngữ máy tính đặc tả để giải quyết nhanh một lớp vấn đề được định nghĩa. Ngôn ngữ đặc tả thường làm cho vấn đề được giải quyết nhanh hơn ngôn ngữ thông thường từ một cho đến vài trăm lần

Ý tưởng tương tự như vậy biểu diễn các biểu thức tính toán theo cú pháp Ba Lan

#### - Mô hình mẫu



Trong đó:

Expression: là một giao tiếp mà thông qua nó, client tương tác với các biểu thức

TerminalExpression: là một thực thi của giao tiếp Expression, đại diện cho các nốt cuối trong cây cú pháp

NonterminalExpression: là một thực thi khác của giao tiếp Expression, đại diện cho các nút chưa kết thúc trong cấu trúc của cây cú pháp. Nó lưu trữ một tham chiếu đến Expression và triệu gọi phương thức diễn giải cho mỗi phần tử con

Context: chứa thông tin cần thiết cho một vài vị trí trong khi diễn giải. Nó có thể phục vụ như một kênh truyền thông cho các thể hiện của Expression



Client: hoặc là xây dựng hoặc là nhận một thể hiện của cây cú pháp ảo. Cây cú pháp này bao gồm các thể hiện của TerminalExpression và NonterminalExpression để tạo nên câu đặc tả. Client triệu gọi các phương thức diễn giải với ngữ cảnh thích hợp khi cần thiết

- Trường hợp ứng dụng

Có một ngôn ngữ đơn giản để diễn giải vấn đề

Các vấn đề lặp lại có thể được diễn giải nhanh bằng ngôn ngữ đó

- Ví dụ mẫu

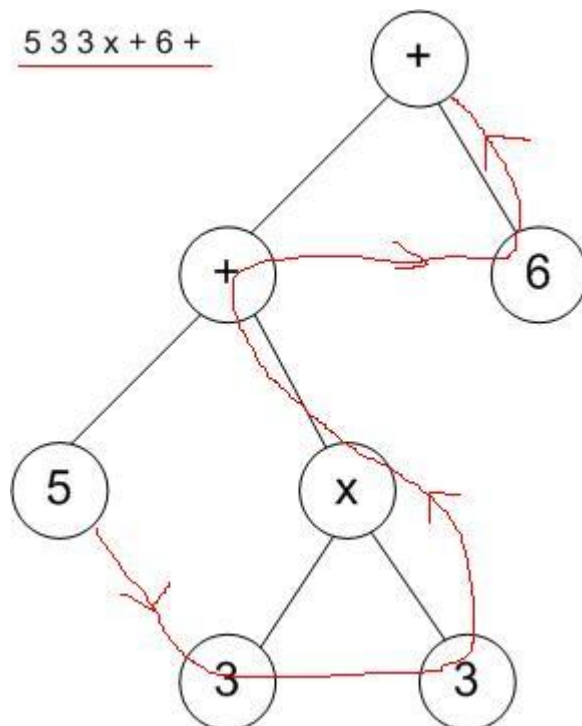
Xét biểu thức  $5 + 3 \times 3 + 6$ , với bài toán này ta có thể chia thành các bài các bài toán nhỏ hơn

-Tính  $3 \times 3 = a$

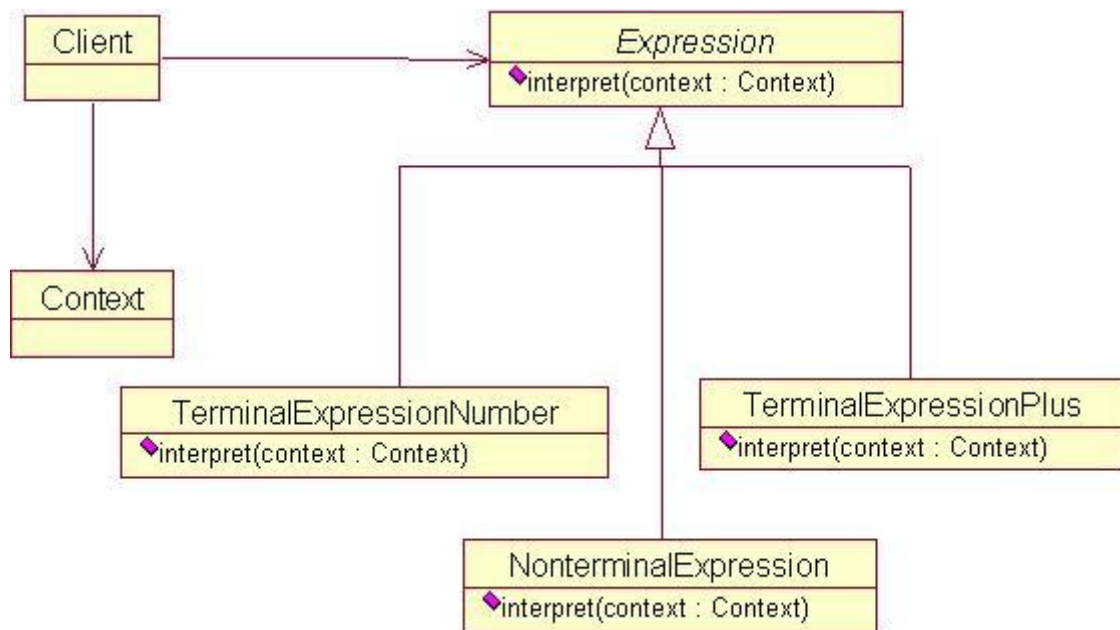
-Sau đó tính  $5 + a = b$

-Sau đó tính  $b + 6$

Ta biểu diễn bài toán thành cấu trúc cây và duyệt cây theo Ba Lan (hay Ba Lan đảo gì đó không còn nhớ nữa)



Mã nguồn cho ví dụ này như sau



```

public class Context extends Stack<Integer>{
}
public interface Expression {
    public void interpret(Context context);
}
public class TerminalExpressionNumber implements
Expression {
    private int number;
    public TerminalExpressionNumber(int number){
        this.number = number;
    }
    public void interpret(Context context) {
        context.push(this.number);
    }
}
public class TerminalExpressionPlus implements
Expression {
    public void interpret(Context context) {
        //Cong 2 phan tu phia tren dinh Stack
        context.push(context.pop())
        context.pop();
    }
}
public class TerminalExpressionMutil implements
Expression{
    public void interpret(Context context) {
        //Nhan 2 phan tu phia tren dinh Stack
        context.push(context.pop())
        context.pop();
    }
}

```

```

public class NonterminalExpression implements
Expression {
    private ArrayList<Expression>
expressions; //tham chieu den mang Expression con
    public ArrayList<Expression> getExpressions()
    {
        return expressions;
    }
    public void
setExpressions(ArrayList<Expression> expressions)
    {
        this.expressions = expressions;
    }
    public void interpret(Context context) {
        if (expressions != null){
            int size = expressions.size();
            for (Expression e : expressions){
                e.interpret(context);
            }
        }
    }
}

public class Client {
    public static void main(String[] args){
        Context context = new Context();
        // 3 3 *
        ArrayList<Expression> treeLevel1 = new
ArrayList<Expression>();
        treeLevel1.add(new
TerminalExpressionNumber(3));
        treeLevel1.add(new
TerminalExpressionNumber(3));
        treeLevel1.add(new
TerminalExpressionMutil());
        // 5 (3 3 *) +
        ArrayList<Expression> treeLevel2 = new
ArrayList<Expression>();
        treeLevel2.add(new
TerminalExpressionNumber(5));
        Expression nonexpLevel1 = new
NonterminalExpression();
        ((NonterminalExpression) nonexpLevel1).set
Expressions(treeLevel1);
        treeLevel2.add(nonexpLevel1);
        treeLevel2.add(new
TerminalExpressionPlus());
        // (5 (3 3 *) +) 6 +
        ArrayList<Expression> treeLevel3 = new
ArrayList<Expression>();
        Expression nonexpLevel2 = new
NonterminalExpression();

```

```

        ((NonterminalExpression) nonexpLevel2).set
Expressions (treeLevel2);
        treeLevel3.add(nonexpLevel2);
        treeLevel3.add(new
TerminalExpressionNumber(6));
        treeLevel3.add(new
TerminalExpressionPlus());
        for(Expression e : treeLevel3){
            e.interpret(context);
        }
        if (context != null)
            System.out.print("Ket    qua:    "    +
context.pop());
    }
}

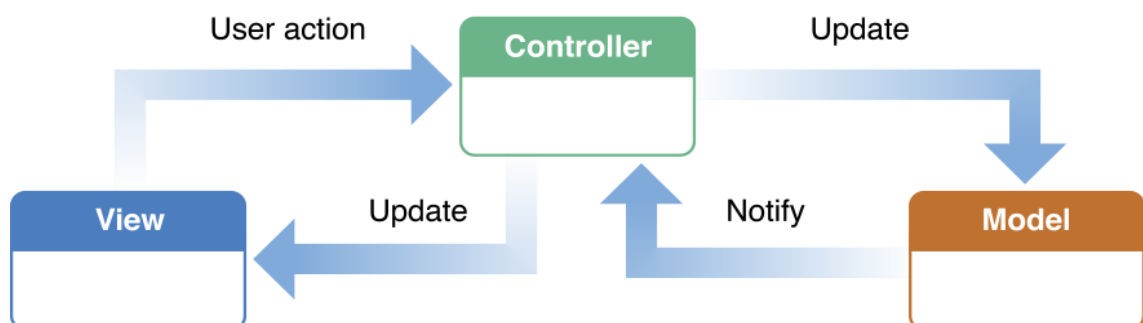
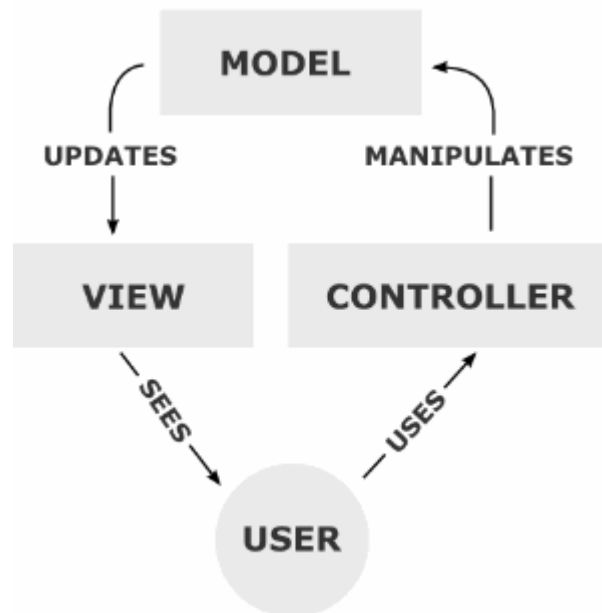
```

## 2.8. Model-View-Controller Pattern (MVC Pattern)

- Ý nghĩa:

MVC là chữ viết tắt của **Model - View - Controller**, Là một trong những design pattern, đây là một mô hình kiến trúc phần mềm được tạo ra với mục đích quản lý và xây dựng dự án phần mềm có hệ thống hơn. Mô hình này được dùng khá rộng rãi, hầu hết trong các ngôn ngữ.

MVC được vận hành để tách mã lệnh thành 3 phần riêng biệt. Ở mỗi phần MVC sẽ có những chức năng đặc thù. Để xử lý các tác vụ mà request gửi tới. MVC làm cho mã lệnh trở nên trong sáng, dễ phát triển và dễ nâng cấp theo thời gian.



### **Trong mô hình này thì:**

- Model: có nhiệm vụ thao tác với cơ sở dữ liệu, nghĩa là nó sẽ chứa tất cả các hàm, các phương thức truy vấn trực tiếp với dữ liệu và controller sẽ thông qua các hàm, phương thức đó để lấy dữ liệu rồi gửi qua View

- View: có nhiệm vụ tiếp nhận dữ liệu từ controller và hiển thị thông tin tương phản khi gửi và nhận request, bạn có thể hiểu nôm na đây người ta còn gọi là thành phần giao diện.

- Controller: đóng vai trò trung gian giữa Model và View. Nó có nhiệm vụ tiếp nhận yêu cầu từ user sau đó xử lý request, load model tương ứng và gửi data qua view tương ứng rồi trả kết quả về cho user

Nhìn vào mô hình này các bạn thấy giữa model và view không hề có mối liên hệ mà nó sẽ thông qua controller để giao tiếp với nhau

### **Ưu khuyết điểm của MVC**

#### **Ưu Điểm:**

MVC làm cho ứng dụng trở nên trong sáng, giúp lập trình viên phân tách ứng dụng thành ba lớp một cách rõ ràng. Điều này sẽ rất giúp ích cho việc phát triển những ứng dụng xét về mặt lâu dài cho việc bảo trì và nâng cấp hệ thống.

MVC hiện đang là mô hình lập trình tiên tiến bậc nhất hiện nay, điều mà các framework vẫn đang nỗ lực để hướng tới sự đơn giản và yếu tố lâu dài cho người sử dụng.

#### **Khuyết Điểm:**

Mặc dù, MVC tỏ ra lợi thế hơn nhiều so với cách lập trình thông thường. Nhưng MVC luôn phải nạp, load những thư viện đồ sộ để xử lý dữ liệu. Chính điều này làm cho mô hình trở nên chậm chạp hơn nhiều so với việc code tay thuần túy.

MVC đòi hỏi người tiếp cận phải biết qua OOP, có kinh nghiệm tương đối cho việc thiết lập và xây dựng một ứng dụng hoàn chỉnh. Sẽ rất khó khăn nếu OOP của người sử dụng còn yếu.

MVC tận dụng mảng là thành phần chính cho việc truy xuất dữ liệu. Nhất là với việc sử dụng active record để viết ứng dụng. Chúng luôn cần người viết phải nắm vững mô hình mảng đa chiều.