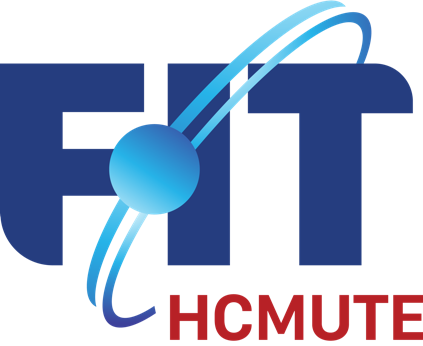
**A white rectangular frame with blue border

Description automatically generatedTRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HCM**

**KHOA CÔNG NGHỆ THÔNG TIN**



**BÁO CÁO**

*Đề tài:*

**VISITOR PATTERN**

**Môn:** Mẫu Thiết Kế Phần Mềm

**Mã lớp:** DEPA330879\_03CLC

**GVHD:** ThS. Nguyễn Minh Đạo

**SVTH:** Nguyễn Hà Quỳnh Giao

**MSSV:** 21110171

*TP. Hồ Chí Minh, tháng 04 năm 2024*

MỤC LỤC

[**MỞ ĐẦU** 1](#_Toc166112993)

[**1.** **Khái niệm:** 1](#_Toc166112994)

[**2.** **Cấu trúc:** 1](#_Toc166112995)

[**3.** **Trường hợp sử dụng:** 1](#_Toc166112996)

[**4.** **Ưu điểm:** 1](#_Toc166112997)

[**5.** **Nhược điểm:** 2](#_Toc166112998)

[**THIẾT KẾ** 3](#_Toc166112999)

[**1.** **Ý tưởng thiết kế:** 3](#_Toc166113000)

[**2.** **Giải pháp sử dụng Visitor Pattern:** 3](#_Toc166113001)

[**3.** **Thiết kế các lớp:** 4](#_Toc166113002)

[**3.1.** **Lớp interface:** 4](#_Toc166113003)

[**3.2.** **Lớp concrete:** 4](#_Toc166113004)

[**4.** **Sử dụng:** 6](#_Toc166113005)

[**5.** **Phân tích:** 6](#_Toc166113006)

[**KẾT LUẬN** 7](#_Toc166113007)

[**TÀI LIỆU THAM KHẢO** 8](#_Toc166113008)

# **MỞ ĐẦU**

1. **Khái niệm:**

Visitor là một mẫu thiết kế hành vi cho phép bạn tách biệt các thuật toán từ các đối tượng mà chúng hoạt động.

1. **Cấu trúc:**

Một mẫu Visitor Pattern bao gồm:

* Lớp interface/abstract:

+ Visitor: khai báo các phương thức, mỗi phương thức tương ứng với một đối tượng cụ thể.

+ Element: đại diện cho các đối tượng, khai báo một phương thức chung là accept.

* Concrete:

+ Visitor: lớp kế thừa lớp interface/abstract Visitor – là các triển khai cụ thể các phương thức được khai báo.

+ Element: lớp kế thừa interface/abstract Element – triển khai cụ thể các đối tượng.

1. **Trường hợp sử dụng:**

* Cần thực hiện một hoạt động trên tất cả các phần tử của một cấu trúc đối tượng phức tạp.
* Làm sạch logic Bussiness của các hành vi phụ trợ.
* Một hành vi chỉ hợp lý trong một số lớp của một cấu trúc lớp, nhưng không phù hợp trong các lớp khác.

1. **Ưu điểm:**

* Nguyên tắc Open/Closed: Bạn có thể thêm một hành vi mới có thể hoạt động với các đối tượng thuộc các lớp khác nhau mà không cần thay đổi các lớp đó.
* Nguyên tắc Single Responsibility: Bạn có thể di chuyển nhiều phiên bản của cùng một hành vi vào cùng một lớp.
* Có thể tích luỹ thông tin hữu ích khi làm việc với các đối tượng khác nhau.

1. **Nhược điểm:**

* Thêm hoặc xóa một lớp trong cấu trúc các phần tử, bạn cần cập nhật tất cả các visitor tương ứng.
* Thiếu quyền truy cập cần thiết vào các trường và phương thức riêng tư của các phần tử mà chúng được dự định làm việc với.

# **THIẾT KẾ**

1. **Ý tưởng thiết kế:**

Trong bài toán này, hệ thống xây dựng để quá lý hình học 2D.

Ngôn ngữ lập trình: Java

1. **Giải pháp sử dụng Visitor Pattern:**

A diagram of a computer program

Description automatically generated with medium confidence

* Lớp interface:

**+** ShapeVisitor: khai báo phương thức, định nghĩa cho từng loại hình.

+ Shape: đại diện cho hình học, chứa khai báo phương thức accept.

* Concrete:

+ AreaCalculator, ShapeDrawer: triển khai các phương thức tương ứng với từng loại hình được khai báo ở Shape Visitor.

+ Circle, Triangle, Square: triển khai cụ thể từng loại hình học, và cung cấp phương thức accept.

1. **Thiết kế các lớp:**
   1. **Lớp interface:**

Shape.java

package org.example;  
  
public interface Shape {  
 public void accept(ShapeVictor shapeVictor);  
}

ShapeVisitor.java

package org.example;  
  
public interface ShapeVictor {  
 public void visitCircle(Circle circle);  
 public void visitSquare(Square square);  
 public void visitTriangle(Triangle triangle);  
}

* 1. **Lớp concrete:**

Circle.java

package org.example;  
  
public class Circle implements Shape{  
 protected double radius;  
  
 public Circle(double radius){  
 this.radius = radius;  
 }  
  
 @Override  
 public void accept(ShapeVictor shapeVictor) {  
 shapeVictor.visitCircle(this);  
 }  
}

Square.java

package org.example;  
  
public class Square implements Shape{  
  
 protected double side;  
  
 public Square(double side){  
 this.side = side;  
 }  
  
 @Override  
 public void accept(ShapeVictor shapeVictor) {  
 shapeVictor.visitSquare(this);  
 }  
}

Triangle.java

package org.example;  
  
public class Triangle implements Shape{  
 protected double base;  
 protected double height;  
  
 public Triangle(double base, double height){  
 this.base = base;  
 this.height = height;  
 }  
  
 @Override  
 public void accept(ShapeVictor shapeVictor) {  
 shapeVictor.visitTriangle(this);  
 }  
}

AreaCalculator.java

package org.example;  
  
public class AreaCalculator implements ShapeVictor{  
 @Override  
 public void visitCircle(Circle circle) {  
 double area = Math.*pow*(circle.radius, 2) \* Math.*PI*;  
 System.*out*.println("Area of Circle is " + area);  
 }  
  
 @Override  
 public void visitSquare(Square square) {  
 double area = Math.*pow*(square.side, 2);  
 System.*out*.println("Area of Square is " + area);  
 }  
  
 @Override  
 public void visitTriangle(Triangle triangle) {  
 double area = 0.5 \* triangle.base \* triangle.height;  
 System.*out*.println("Area of Square is " + area);  
 }  
}

ShapeDrawer.java

package org.example;  
  
public class ShapeDrawer implements ShapeVictor{  
 @Override  
 public void visitCircle(Circle circle) {  
 System.*out*.println("Drawing circle...");  
 }  
  
 @Override  
 public void visitSquare(Square square) {  
 System.*out*.println("Drawing square...");  
 }  
  
 @Override  
 public void visitTriangle(Triangle triangle) {  
 System.*out*.println("Drawing triangle...");  
 }  
}

1. **Sử dụng:**

Main.java

package org.example;  
  
public class Main {  
 public static void main(String[] args) {  
 ShapeVictor area = new AreaCalculator();  
 ShapeVictor draw = new ShapeDrawer();  
  
 Shape circle = new Circle(4);  
 Shape square = new Square(4);  
 Shape triangle = new Triangle(4, 4);  
  
 circle.accept(area);  
 square.accept(area);  
 triangle.accept(area);  
  
 circle.accept(draw);  
 square.accept(draw);  
 triangle.accept(draw);  
 }  
}

1. **Phân tích:**

Với ví dụ trên, ta thấy rằng mẫu Visitor cho phép chúng ta thêm các phương thức hoặc tính năng mới cho các lớp đối tượng mà không cần phải thay đổi cấu trúc của chúng. Bằng cách sử dụng một interface Shape và ShapeVisitor, các lớp hình học như Circle, Square và Triangle có thể chấp nhận và tương tác với các ConcreteVisitor như AreaCalculator và ShapeDrawer để tính toán diện tích và vẽ hình một cách linh hoạt và dễ dàng bảo trì. Điều này giúp mã nguồn trở nên linh hoạt hơn và dễ mở rộng, mà không làm ảnh hưởng đến cấu trúc của các lớp đối tượng.

# **KẾT LUẬN**

Mẫu thiết kế Visitor cung cấp một phương pháp linh hoạt để thêm các phương thức hoặc tính năng mới cho các lớp đối tượng mà không làm thay đổi cấu trúc của chúng. Bằng cách sử dụng một interface Visitor và các lớp ConcreteVisitor, mẫu thiết kế này cho phép chúng ta thực hiện các hành động khác nhau trên các đối tượng khác nhau một cách dễ dàng và hiệu quả. Điều này giúp tách biệt logic xử lý với các lớp đối tượng, làm cho mã nguồn trở nên dễ bảo trì, mở rộng và tái sử dụng.

# **TÀI LIỆU THAM KHẢO**

**[1]** Refactoring. Visitor. Refactoring ruru.https://refactoring.guru/design-patterns/visitor **.** Ngày truy cập: 09-05-2024.