

**TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM  
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG  
KHOA CÔNG NGHỆ THÔNG TIN**



**BÁO CÁO GIỮA KỲ MÔN MẪU THIẾT KẾ**

# **TÌM HIỂU VỀ PROXY VÀ VISITOR DESIGN PATTERN**

*Người hướng dẫn:* **ThS VŨ ĐÌNH HỒNG**

*Người thực hiện:* **LÊ PHAN THẾ VĨ – 52200038**

**NGUYỄN CAO KỲ – 52200056**

**NGUYỄN NAM HOÀNG – 52200079**

**Lớp : 22050201**

**Khoá : 26**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025**

**TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM  
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG  
KHOA CÔNG NGHỆ THÔNG TIN**



**BÁO CÁO GIỮA KỲ MÔN MẪU THIẾT KẾ**

# **TÌM HIỂU VỀ PROXY VÀ VISITOR DESIGN PATTERN**

*Người hướng dẫn:* **ThS VŨ ĐÌNH HỒNG**

*Người thực hiện:* **LÊ PHAN THẾ VĨ – 52200038**

**NGUYỄN CAO KỲ – 52200056**

**NGUYỄN NAM HOÀNG – 52200079**

**Lớp : 22050201**

**Khoá : 26**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025**

## LỜI CẢM ƠN

Đầu tiên, với tình cảm sâu sắc và chân thành nhất, cho phép chúng em được bày tỏ lòng biết ơn đến thầy Vũ Đình Hồng, thầy đã giúp đỡ chúng em trong việc củng cố các kiến thức để chúng em có thể hoàn thành được bài báo cáo giữa kỳ môn ‘Mẫu thiết kế’. Trong thời gian qua, thầy đã dành thời gian để hướng dẫn và trả lời những câu hỏi của chúng em với sự kiên nhẫn và tận tình. Nhờ vậy mà chúng em đã có thể hoàn thành bài báo cáo của mình một cách hiệu quả và tự tin hơn.

Nhờ có sự hướng dẫn của thầy, chúng em đã có thể hiểu rõ hơn về nội dung môn học và hoàn thành bài tập một cách hiệu quả hơn. Chúng em cảm thấy rất hạnh phúc và tự hào khi có cơ hội được học hỏi từ thầy.

Bài báo cáo của chúng em được thực hiện trong khoảng thời gian gần 2 tháng. Bước đầu làm những bài báo cáo nên khả năng của chúng em còn hạn chế và còn nhiều bỡ ngỡ nên không tránh khỏi những thiếu sót, chúng em rất mong nhận được những ý kiến đóng góp quý báu của thầy để kiến thức của chúng em trong lĩnh vực này được hoàn thiện hơn, đồng thời có điều kiện bổ sung, nâng cao ý thức của mình.

Một lần nữa, chúng em xin chân thành cảm ơn về sự giúp đỡ và hướng dẫn tận tình của thầy. Chúng em hy vọng rằng trong tương lai, chúng em vẫn có cơ hội được học tập và được hướng dẫn bởi thầy.

Trân trọng!

## **ĐỒ ÁN ĐƯỢC HOÀN THÀNH TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

Tôi xin cam đoan đây là sản phẩm đồ án của riêng chúng tôi và được sự hướng dẫn của ThS Vũ Đình Hồng. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong đồ án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

**Nếu phát hiện có bất kỳ sự gian lận nào tôi xin hoàn toàn chịu trách nhiệm về nội dung đồ án của mình.** Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

*TP. Hồ Chí Minh, ngày 29 tháng 03 năm 2025*

*Tác giả*



*Lê Phan Thế Vĩ*

*Nguyễn Nam Hoàng*

*Nguyễn Cao Kỳ*

## **PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN**

### **Phần xác nhận của GV hướng dẫn**

---

---

---

---

---

---

---

---

Tp. Hồ Chí Minh, ngày      tháng      năm  
(kí và ghi họ tên)

### **Phần đánh giá của GV chấm bài**

---

---

---

---

---

---

---

---

Tp. Hồ Chí Minh, ngày      tháng      năm  
(kí và ghi họ tên)

## TÓM TẮT

Bài báo cáo bao gồm các nội dung liên quan đến 2 mẫu thiết kế: Proxy và Visitor. Trong bài sẽ trình bày các phần gồm giới thiệu vấn đề, cách giải quyết, các trường hợp áp dụng và code minh họa.

## MỤC LỤC

LỜI CẢM ƠN .....	i
PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN .....	iii
TÓM TẮT .....	iv
MỤC LỤC .....	1
DANH MỤC CÁC BẢNG BIỂU, HÌNH VẼ, ĐỒ THỊ .....	4
CHƯƠNG 1 – PROXY .....	6
1.1 Giới thiệu .....	6
1.1.1 Tổng quát.....	6
1.1.1.1 Sơ đồ lớp tổng quát .....	6
1.1.1.2 Các thành phần.....	6
1.1.1.3 Cách hoạt động.....	7
1.1.2 Trường hợp áp dụng .....	7
1.2 Vấn đề .....	8
1.2.1 Vấn đề với cách tiếp cận truyền thống.....	8
1.2.2 Giải pháp với Proxy Pattern.....	8
1.3 Ví dụ cụ thể.....	8
1.4 Áp mẫu thiết kế vào ví dụ .....	9
1.4.1 Áp dụng vào ví dụ CacheProduct .....	9
1.4.2 Các use case thực tế có thể áp dụng.....	11
1.5 Source code .....	11
1.5.1 Source code với cách áp dụng truyền thống .....	11
1.5.2 Source code áp dụng Proxy.....	13
1.6 Tóm tắt và kết luận.....	15
1.6.1 Tóm tắt .....	15
1.6.2 Kết luận .....	15
CHƯƠNG 2 – VISITOR.....	17

2.1 Giới thiệu .....	17
2.1.1 Tổng quát .....	17
2.1.1.1 Sơ đồ lớp tổng quát.....	17
2.1.1.2 Các thành phần.....	17
2.2.2 Trường hợp áp dụng.....	18
2.2 Vấn đề .....	18
2.2.1 Vấn đề của cách tiếp cận truyền thống .....	18
2.2.2 Giải pháp với Visitor Pattern .....	19
2.3 Ví dụ cụ thể.....	19
2.3.1 Vi phạm nguyên tắc trách Mở/Đóng – OCP.....	20
2.3.2 Vi phạm Single Responsibility Principle (SRP).....	20
2.4 Áp mẫu thiết kế vào ví dụ .....	21
2.4.1 Áp dụng vào ví dụ ShapeApp .....	21
2.4.2 Các use case thực tế có thể áp dụng.....	22
2.5 Source code .....	23
2.5.1 Source code với cách áp dụng truyền thống .....	23
2.5.2 Source code áp dụng Visitor .....	25
2.6 Tóm tắt và kết luận.....	28
2.6.1 Tóm tắt .....	28
2.6.2 Kết luận.....	28



## **DANH MỤC KÍ HIỆU VÀ CHỮ VIẾT TẮT**

**CÁC KÝ HIỆU**

**CÁC CHỮ VIẾT TẮT**

## **DANH MỤC CÁC BẢNG BIỂU, HÌNH VẼ, ĐỒ THỊ**

### **DANH MỤC HÌNH**

#### **CHƯƠNG 1**

Hình 1. 1 Sơ đồ lớp tổng quát .....	6
Hình 1. 2 Sơ đồ lớp cách áp dụng truyền thống.....	9
Hình 1. 3 Sơ đồ lớp cách áp dụng Proxy .....	10
Hình 1. 4 Interface Product .....	11
Hình 1. 5 Lớp RealProduct .....	12
Hình 1. 6 Lớp Test .....	13
Hình 1. 7 Interface Product .....	13
Hình 1. 8 Lớp RealProduct .....	14
Hình 1. 9 Lớp ProductProxy .....	14
Hình 1. 10 Lớp Test .....	15

#### **CHƯƠNG 2**

Hìnhg 2. 1 Sơ đồ tổng quát Visitor .....	17
Hìnhg 2. 2 Sơ đồ lớp ShapeApp với các tiếp cận truyền thống.....	19
Hìnhg 2. 3 Sơ đồ lớp ShapeApp với các tiếp cận truyền thống mở rộng .....	20
Hìnhg 2. 4 Sơ đồ lớp ShapeApp với các tiếp cận Visitor .....	21
Hìnhg 2. 5 Sơ đồ lớp ShapeApp với các tiếp cận Visitor mở rộng.....	22
Hìnhg 2. 6 Interface Shape.....	23
Hìnhg 2. 7 Lớp Circle .....	24
Hìnhg 2. 8 Lớp Square .....	24
Hìnhg 2. 9 Lớp Test .....	24
Hìnhg 2. 10 Interface Shape.....	25
Hìnhg 2. 11 Lớp Circle .....	25
Hìnhg 2. 12 Lớp Square .....	26
Hìnhg 2. 13 Interface ShapeVisitor.....	26

Hình 2. 14 Lớp AreaVisitor .....	26
Hình 2. 15 Lớp DrawVisitor .....	27
Hình 2. 16 Lớp ExportVisitor .....	27
Hình 2. 17 Lớp Test .....	28

## **DANH MỤC BẢNG**

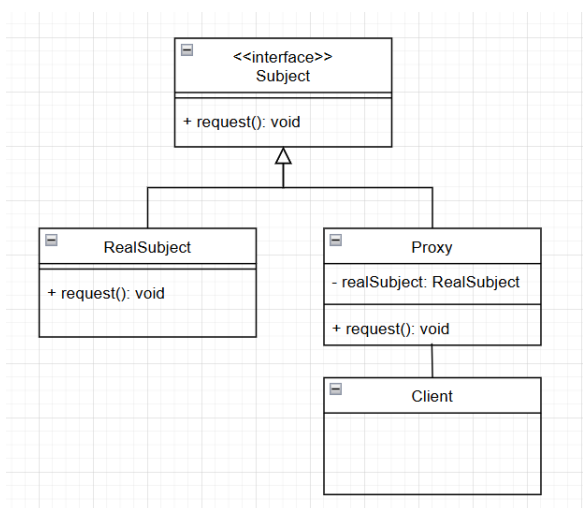
## CHƯƠNG 1 – PROXY

### 1.1 Giới thiệu

#### 1.1.1 Tổng quát

Proxy pattern, hay mẫu đại diện, là một mẫu thiết kế thuộc nhóm cấu trúc, được sử dụng để cung cấp một đối tượng thay thế cho đối tượng thực nhằm kiểm soát truy cập vào nó. Mẫu này cho phép thêm các chức năng bổ sung như tải chậm, kiểm soát quyền truy cập, caching, hoặc logging mà không cần thay đổi giao diện của đối tượng gốc.

##### 1.1.1.1 Sơ đồ lớp tổng quát



Hình 1. 1 Sơ đồ lớp tổng quát

##### 1.1.1.2 Các thành phần

Proxy Pattern bao gồm ba thành phần chính, mỗi thành phần đóng một vai trò quan trọng trong việc quản lý truy cập và mở rộng chức năng của đối tượng:

**Subject:** Đây là giao diện chung mà cả **RealSubject** và **Proxy** đều triển khai. Nó định nghĩa các phương thức mà client có thể gọi, ví dụ như `request()`. Giao diện này đảm bảo rằng client có thể làm việc với cả **Proxy** và **RealSubject** mà không cần biết sự khác biệt.

**RealSubject:** Đây là đối tượng thực sự mà client muốn truy cập. RealSubject chứa logic kinh doanh chính và thực hiện các thao tác thực tế khi được yêu cầu. Nó triển khai giao diện Subject để đảm bảo tính tương thích.

**Proxy:** Đây là đối tượng thay thế cho RealSubject. Proxy cũng triển khai giao diện Subject và giữ một tham chiếu đến RealSubject. Nó có thể kiểm soát truy cập đến RealSubject, đồng thời thêm các chức năng bổ sung như tải chậm (lazy loading), lưu trữ tạm (caching), hoặc kiểm tra quyền truy cập. Khi client gọi phương thức trên Proxy, nó quyết định cách và khi nào chuyển tiếp yêu cầu đến RealSubject.

#### 1.1.1.3 Cách hoạt động

Client tương tác với đối tượng thông qua giao diện Subject. Client không cần biết liệu nó đang làm việc với RealSubject hay Proxy.

Khi client gọi phương thức request() trên Proxy, Proxy có thể thực hiện các hành động bổ sung (ví dụ: kiểm tra quyền, ghi log, hoặc tải dữ liệu khi cần) trước hoặc sau khi chuyển tiếp yêu cầu đến RealSubject.

Nhờ vậy, Proxy giúp kiểm soát và mở rộng hành vi của RealSubject mà không cần thay đổi mã nguồn của RealSubject, tuân theo nguyên tắc Open/Closed trong SOLID.

#### 1.1.2 Trường hợp áp dụng

Mẫu thiết kế Proxy đóng vai trò như một lớp trung gian, giúp kiểm soát và tối ưu hóa quá trình truy cập đối tượng. Một số ứng dụng phổ biến của Proxy Pattern bao gồm:

- Trì hoãn khởi tạo đối tượng nặng cho đến khi thực sự cần thiết, giúp tiết kiệm tài nguyên.
- Bảo mật và kiểm soát truy cập bằng cách sử dụng Protection Proxy để giới hạn quyền truy cập dựa trên vai trò người dùng.
- Giao tiếp từ xa thông qua Remote Proxy, cho phép tương tác với đối tượng trên máy chủ từ xa mà không cần thay đổi mã nguồn.

- Tăng hiệu suất và theo dõi hệ thống bằng cách lưu cache kết quả hoặc ghi log mà không ảnh hưởng đến đối tượng thực.

## 1.2 Vấn đề

Hãy xem xét một ứng dụng thương mại điện tử, nơi thông tin sản phẩm (tên, giá, mô tả) được lấy từ cơ sở dữ liệu.

### 1.2.1 Vấn đề với cách tiếp cận truyền thống

Trong cách tiếp cận thông thường, mỗi lần client yêu cầu thông tin sản phẩm, hệ thống sẽ thực hiện một truy vấn trực tiếp đến cơ sở dữ liệu, dẫn đến các vấn đề sau:

- Giảm hiệu suất: Nếu nhiều khách hàng cùng xem một sản phẩm, hệ thống phải thực hiện nhiều truy vấn lặp lại, gây tốn thời gian và tài nguyên.
- Lãng phí tài nguyên: Việc truy vấn dữ liệu liên tục có thể tạo áp lực lớn lên hệ thống, đặc biệt khi số lượng sản phẩm và người dùng tăng cao.
- Tăng độ trễ: Mỗi lần lấy dữ liệu đều cần giao tiếp với cơ sở dữ liệu, gây độ trễ không cần thiết nếu dữ liệu không thay đổi thường xuyên.

### 1.2.2 Giải pháp với Proxy Pattern

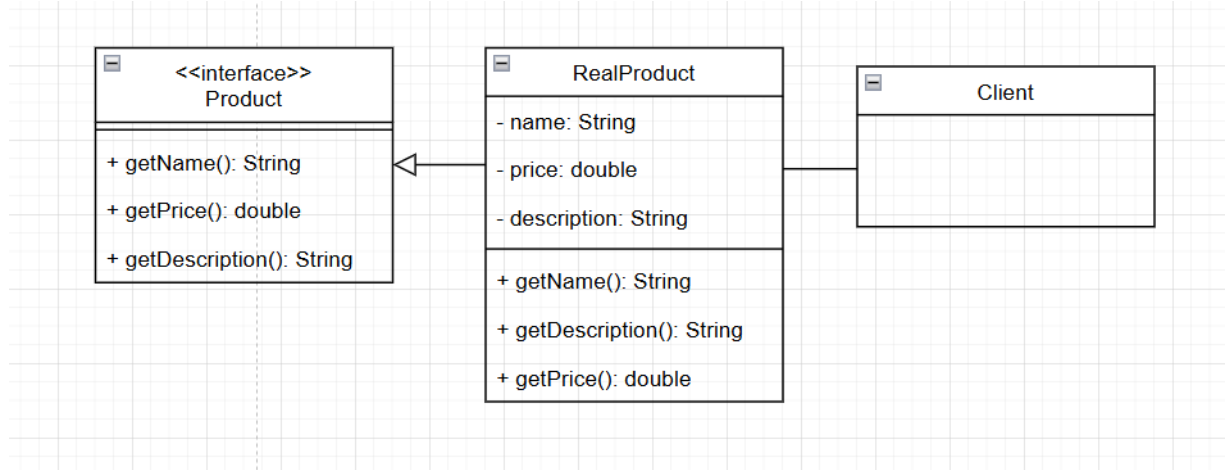
Để tối ưu hiệu suất, chúng ta sử dụng Proxy Pattern bằng cách tạo một lớp Proxy để lưu trữ tạm (cache) thông tin sản phẩm. Cách hoạt động như sau:

- Khi client lần đầu yêu cầu thông tin sản phẩm, Proxy sẽ thực hiện truy vấn từ cơ sở dữ liệu và lưu lại kết quả.
- Với các yêu cầu sau, thay vì truy vấn lại, Proxy sẽ trả về dữ liệu từ cache, giúp giảm tải hệ thống, tiết kiệm tài nguyên và tăng tốc độ phản hồi.

Bằng cách này, Proxy Pattern giúp giảm số lần truy vấn không cần thiết, cải thiện đáng kể hiệu suất và khả năng mở rộng của ứng dụng.

## 1.3 Ví dụ cụ thể

Khi áp dụng cách truyền thống vào, ta có sơ đồ lớp sau:

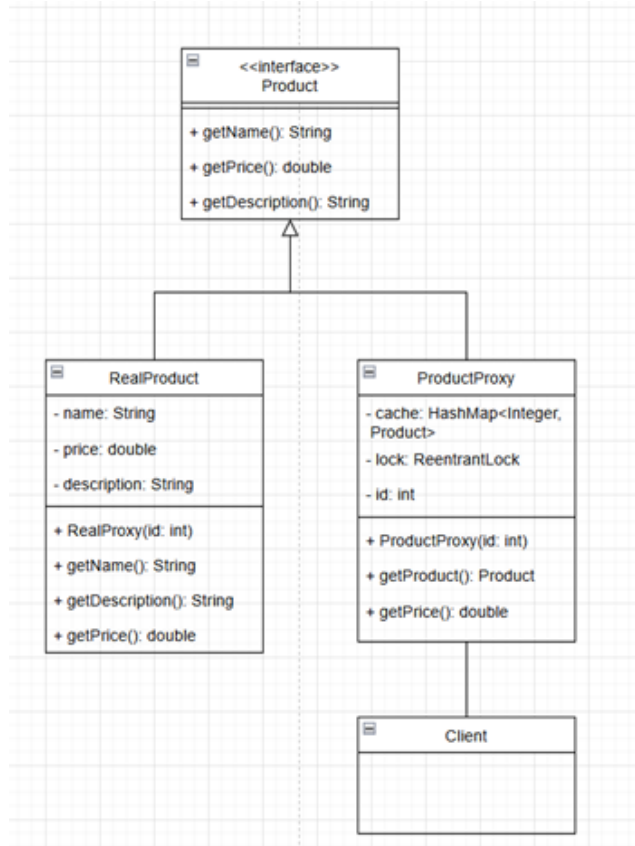


Hình 1. 2 Sơ đồ lớp cách áp dụng truyền thống

Cách tiếp cận truyền thống kém hiệu quả do không có caching, khiến mỗi lần tạo 'RealProduct' đều truy vấn lại cơ sở dữ liệu, dẫn đến thời gian phản hồi chậm, đặc biệt khi có nhiều yêu cầu lặp lại, đồng thời tốn tài nguyên vì tạo mới đối tượng mỗi lần thay vì tái sử dụng, dễ gây quá tải hệ thống. Việc không hỗ trợ lazy loading buộc khởi tạo ngay lập tức dù chưa cần dung cũng gây tốn tài nguyên không cần thiết.

## 1.4 Áp mẫu thiết kế vào ví dụ

### 1.4.1 Áp dụng vào ví dụ *CacheProduct*



Hình 1. 3 Sơ đồ lớp cách áp dụng Proxy

Proxy pattern được áp dụng bằng cách tạo lớp ProductProxy thay thế cho RealProduct. Cả hai cùng triển khai giao diện Product, đảm bảo client có thể sử dụng proxy mà không cần thay đổi code.

Các thành phần chính khi áp dụng Proxy như sau:

- Real Subject (Lớp RealProduct): Thực hiện chức năng thực, mô phỏng truy vấn cơ sở dữ liệu với độ trễ 1 giây
- Proxy (Lớp ProductProxy): Quản lý truy cập vào RealProduct, sử dụng cache để lưu trữ đối tượng đã tạo, giảm số lần truy vấn cơ sở dữ liệu.

Khi client gọi phương thức trên ProductProxy, proxy kiểm tra cache. Nếu sản phẩm chưa có trong cache, proxy tạo mới RealProduct và lưu vào cache. Nếu sản phẩm đã có trong cache, proxy trả về đối tượng đã lưu, tránh truy vấn lại cơ sở dữ liệu.



Client tương tác với ProductProxy như với RealProduct, không nhận ra sự khác biệt. Điều này minh họa cách Proxy pattern tối ưu hóa hiệu suất bằng cách giảm số lần thực hiện thao tác tốn tài nguyên.

#### ***1.4.2 Các use case thực tế có thể áp dụng***

Cache Proxy – Tăng tốc truy vấn dữ liệu: Proxy lưu trữ dữ liệu tạm thời để giảm tải hệ thống và tăng tốc độ phản hồi, thường dùng trong API và thương mại điện tử.

Virtual Proxy – Trì hoãn khởi tạo đối tượng nặng: Chi tài tài nguyên (ảnh, video, mô hình 3D) khi cần, giúp tiết kiệm bộ nhớ và tăng hiệu suất.

Protection Proxy – Kiểm soát quyền truy cập: Hạn chế người dùng truy cập tài nguyên theo quyền hạn, ứng dụng trong hệ thống bảo mật và quản lý tài liệu.

Remote Proxy – Truy cập tài nguyên từ xa: Đại diện cho đối tượng trên máy chủ từ xa, giúp client-server giao tiếp mượt mà, như Java RMI hoặc điều khiển IoT.

Logging Proxy – Ghi log hoạt động hệ thống: Ghi lại tất cả các yêu cầu đến dịch vụ, hỗ trợ theo dõi và phân tích hệ thống, thường dùng trong thanh toán trực tuyến và API monitoring.

### **1.5 Source code**

Link: <https://github.com/dukelw/visitor-proxy-design-pattern>

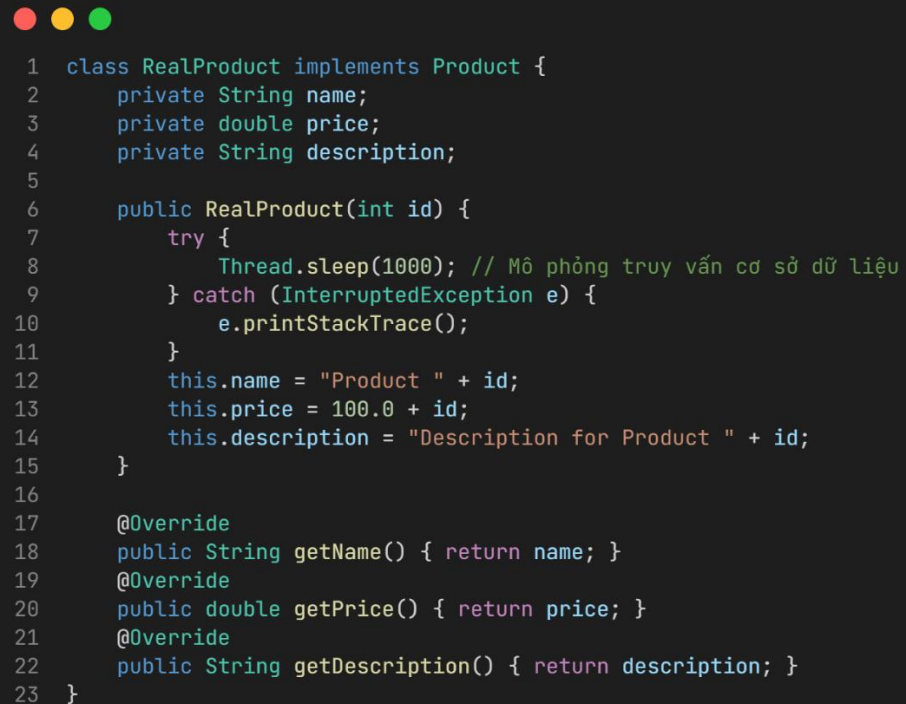
#### ***1.5.1 Source code với cách áp dụng truyền thống***



```

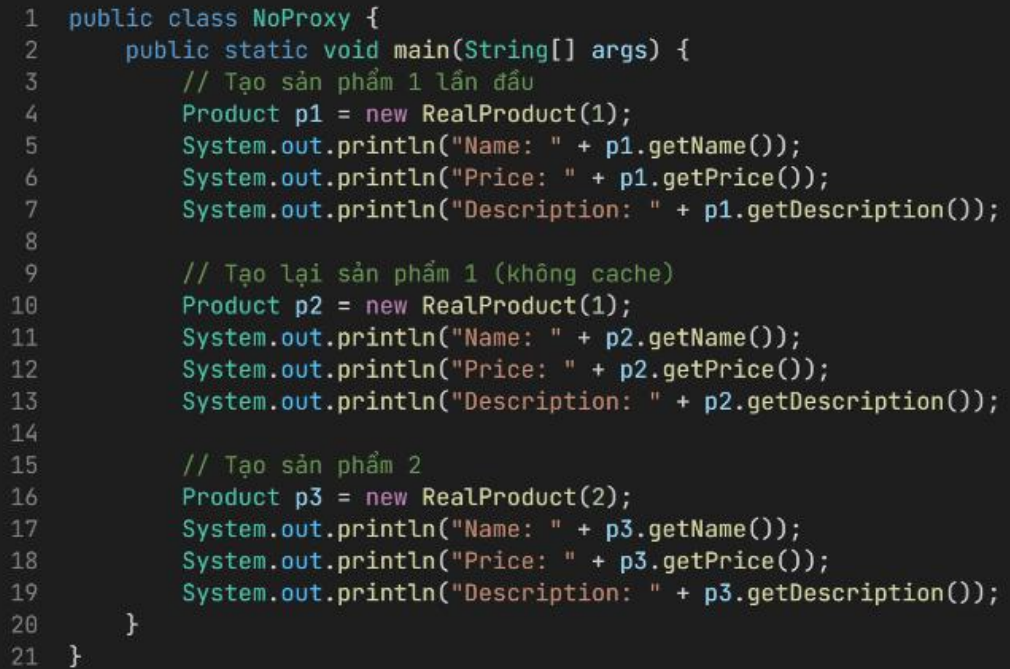
1 interface Product {
2     String getName();
3     double getPrice();
4     String getDescription();
5 }
  
```

Hình 1. 4 Interface Product



```
1 class RealProduct implements Product {
2     private String name;
3     private double price;
4     private String description;
5
6     public RealProduct(int id) {
7         try {
8             Thread.sleep(1000); // Mô phỏng truy vấn cơ sở dữ liệu
9         } catch (InterruptedException e) {
10             e.printStackTrace();
11         }
12         this.name = "Product " + id;
13         this.price = 100.0 + id;
14         this.description = "Description for Product " + id;
15     }
16
17     @Override
18     public String getName() { return name; }
19     @Override
20     public double getPrice() { return price; }
21     @Override
22     public String getDescription() { return description; }
23 }
```

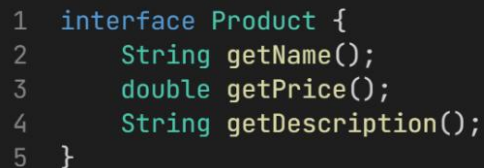
Hình 1. 5 Lớp RealProduct



```
1 public class NoProxy {
2     public static void main(String[] args) {
3         // Tạo sản phẩm 1 lần đầu
4         Product p1 = new RealProduct(1);
5         System.out.println("Name: " + p1.getName());
6         System.out.println("Price: " + p1.getPrice());
7         System.out.println("Description: " + p1.getDescription());
8
9         // Tạo lại sản phẩm 1 (không cache)
10        Product p2 = new RealProduct(1);
11        System.out.println("Name: " + p2.getName());
12        System.out.println("Price: " + p2.getPrice());
13        System.out.println("Description: " + p2.getDescription());
14
15        // Tạo sản phẩm 2
16        Product p3 = new RealProduct(2);
17        System.out.println("Name: " + p3.getName());
18        System.out.println("Price: " + p3.getPrice());
19        System.out.println("Description: " + p3.getDescription());
20    }
21 }
```

Hình 1. 6 Lớp Test

### 1.5.2 Source code áp dụng Proxy



```
1 interface Product {
2     String getName();
3     double getPrice();
4     String getDescription();
5 }
```

Hình 1. 7 Interface Product

```

1  class RealProduct implements Product {
2      private String name;
3      private double price;
4      private String description;
5
6      public RealProduct(int id) {
7          try {
8              Thread.sleep(1000); // Mô phỏng truy vấn cơ sở dữ liệu
9          } catch (InterruptedException e) {
10             e.printStackTrace();
11          }
12             this.name = "Product " + id;
13             this.price = 100.0 + id;
14             this.description = "Description for Product " + id;
15         }
16
17         @Override
18         public String getName() { return name; }
19         @Override
20         public double getPrice() { return price; }
21         @Override
22         public String getDescription() { return description; }
23     }

```

Hình 1. 8 Lớp RealProduct

```

1  class ProductProxy implements Product {
2      private static final HashMap<Integer, Product> cache = new HashMap<>();
3      private static final ReentrantLock lock = new ReentrantLock();
4      private int id;
5
6      public ProductProxy(int id) {
7          this.id = id;
8      }
9
10     @Override
11     public String getName() { return getProduct().getName(); }
12     @Override
13     public double getPrice() { return getProduct().getPrice(); }
14     @Override
15     public String getDescription() { return getProduct().getDescription(); }
16
17     private Product getProduct() {
18         lock.lock();
19         try {
20             if (!cache.containsKey(id)) {
21                 cache.put(id, new RealProduct(id));
22             }
23             return cache.get(id);
24         } finally {
25             lock.unlock();
26         }
27     }
28 }
29

```

Hình 1. 9 Lớp ProductProxy



Hình 1. 10 Lớp Test

## 1.6 Tóm tắt và kết luận

### 1.6.1 Tóm tắt

Proxy pattern là một công cụ mạnh mẽ và linh hoạt, mang lại giải pháp hiệu quả cho các nhà phát triển trong việc cải thiện hiệu suất và bảo mật của ứng dụng mà không làm phức tạp hóa mã nguồn. Có thể sử dụng Proxy nhằm tăng hiệu suất thông qua các kỹ thuật như caching và tải chậm, hỗ trợ bảo trì và mở rộng dễ dàng nhờ tách biệt logic quản lý truy cập khỏi đối tượng thực, hệ thống trở nên dễ bảo trì và linh hoạt hơn khi cần mở rộng cũng như đặc biệt hữu ích trong các tình huống như kiểm soát quyền truy cập, thêm chức năng bổ sung, hay xử lý tải chậm mà không cần thay đổi giao diện gốc của đối tượng.

### 1.6.2 Kết luận

Proxy Pattern là một mẫu thiết kế hữu ích khi cần một lớp trung gian để kiểm soát truy cập, tối ưu hiệu suất hoặc bảo vệ tài nguyên. Nó giúp trì hoãn khởi tạo đối tượng tốn tài nguyên, caching dữ liệu, kiểm soát quyền truy cập và hỗ trợ giao tiếp từ xa mà không làm thay đổi đối tượng gốc.

Với các ứng dụng thực tế như caching, bảo mật, logging và phân tán hệ thống, Proxy Pattern không chỉ cải thiện hiệu suất mà còn giúp quản lý tài nguyên hiệu quả hơn, duy trì mã nguồn linh hoạt và dễ mở rộng.

## CHƯƠNG 2 – VISITOR

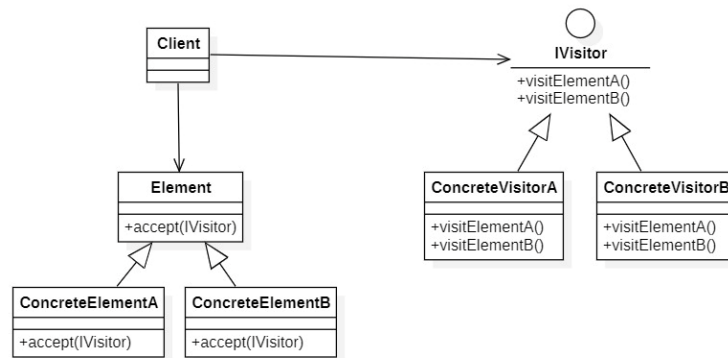
### 2.1 Giới thiệu

#### 2.1.1 Tổng quát

Visitor Pattern là một mẫu thiết kế hành vi (Behavioral Design Pattern) cho phép định nghĩa một hoạt động mới mà không cần thay đổi cấu trúc của các lớp hiện có. Điều này giúp duy trì nguyên tắc Open/Closed Principle (OCP) - mở rộng mà không làm thay đổi mã nguồn gốc.

Mẫu này thường được sử dụng khi có một cấu trúc dữ liệu phức tạp (ví dụ: cây phân cấp các đối tượng) và ta muốn thực hiện nhiều thao tác khác nhau trên các đối tượng này mà không làm thay đổi định nghĩa của chúng.

##### 2.1.1.1 Sơ đồ lớp tổng quát



Hình 2. 1 Sơ đồ tổng quát Visitor

##### 2.1.1.2 Các thành phần

**Client:** Đây là thành phần giao tiếp trực tiếp với cấu trúc đối tượng (Element) và mẫu thiết kế Visitor. Client chịu trách nhiệm tạo các Visitor và truyền chúng vào các phần tử cần xử lý.

**Element:** Thành phần này đóng vai trò như gốc của cấu trúc dữ liệu dạng cây, nơi Visitor sẽ được áp dụng. Thông thường, nó là một interface chứa phương thức accept, mà tất cả các phần tử trong cấu trúc đều phải triển khai.

**ConcreteElement:** Đây là các đối tượng con trong cấu trúc. Một hệ thống có thể chứa nhiều phần tử cụ thể, và tất cả chúng đều cần phải triển khai phương thức `accept`.

**IVisitor:** Đây là giao diện xác định các phương thức mà `Visitor` cần có. Mỗi loại phần tử trong cấu trúc cần một phương thức tương ứng trong `Visitor` để xử lý.

**ConcreteVisitor:** Là các lớp triển khai `Visitor`, thực hiện các thao tác cụ thể trên phần tử. Có thể có nhiều `ConcreteVisitor` khác nhau để thực hiện các hành động khác nhau trên cùng một tập hợp đối tượng.

### ***2.2.2 Trường hợp áp dụng***

Khi cần thực hiện nhiều phép toán trên một tập hợp đối tượng mà không muốn thay đổi lớp của các đối tượng đó.

Khi cấu trúc đối tượng hiếm khi thay đổi nhưng hành vi hoặc thao tác trên chúng lại thay đổi thường xuyên.

Khi cần tách biệt thuật toán khỏi cấu trúc dữ liệu để dễ dàng mở rộng.

## **2.2 Vấn đề**

Ta xét bài toán: Giả sử ta có một hệ thống với nhiều loại đối tượng khác nhau, chẳng hạn như Hình tròn (`Circle`), Hình vuông (`Square`), Hình tam giác (`Triangle`). Mỗi đối tượng có thể cần nhiều chức năng như:

- Vẽ (`draw`)
- Tính diện tích (`calculateArea`)

### ***2.2.1 Vấn đề của cách tiếp cận truyền thống***

Cách tiếp cận thông thường là thêm các phương thức này trực tiếp vào từng lớp. Tuy nhiên, điều này gây ra một số vấn đề như:

- Vi phạm nguyên tắc OCP: Nếu muốn thêm một thao tác mới, ta phải chỉnh sửa tất cả các lớp liên quan.
- Làm cho lớp trở nên cồng kềnh: Mỗi lớp sẽ chứa rất nhiều phương thức khác nhau.



- Khó mở rộng và bảo trì: Khi có nhiều thao tác mới, mã nguồn trở nên khó đọc và dễ bị lỗi.

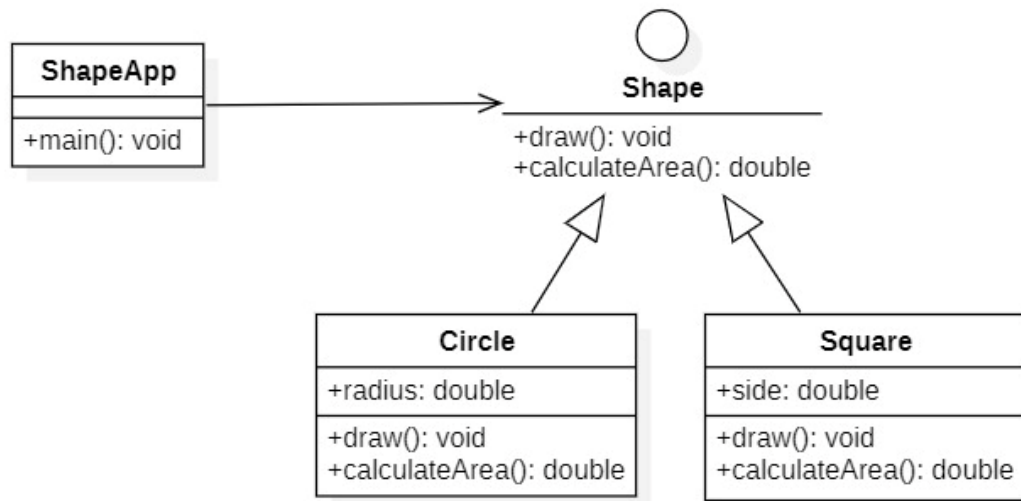
### 2.2.2 Giải pháp với Visitor Pattern

Visitor Pattern giải quyết vấn đề trên bằng cách đưa các hành vi ra khỏi các lớp đối tượng và đặt vào một lớp Visitor riêng. Khi đó:

- Các lớp đối tượng (Circle, Square, Triangle) không thay đổi khi thêm hành vi mới.
- Các lớp Visitor có thể được mở rộng tùy ý mà không ảnh hưởng đến cấu trúc dữ liệu.
- Code dễ bảo trì hơn và tuân thủ nguyên tắc Single Responsibility Principle (SRP).

### 2.3 Ví dụ cụ thể

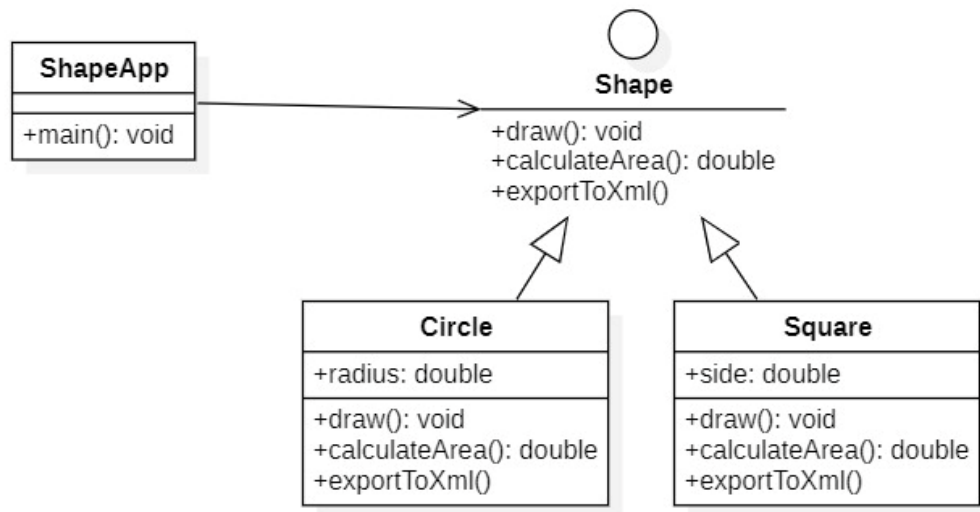
Khi áp dụng cách xử lý truyền thống vào, ta có sơ đồ lớp như sau:



Hình 2. 2 Sơ đồ lớp ShapeApp với các tiếp cận truyền thống

Vấn đề với cách áp dụng truyền thống sẽ hiện ra khi ta được yêu cầu thêm chức năng `exportToXml()` cho các lớp **Circle** và **Square**, khi ấy ta sẽ vi phạm 2 nguyên tắc

thiết kế Open/Closed Principle (Nguyên tắc Mở/Đóng - OCP) và Single Responsibility Principle (Nguyên tắc Trách nhiệm Đơn lẻ - SRP).



Hình 2. 3 Sơ đồ lớp ShapeApp với các tiếp cận truyền thống mở rộng

### 2.3.1 Vi phạm nguyên tắc trách Mở/Đóng – OCP

Nguyên tắc: Một lớp nên mở rộng được (open for extension) nhưng không được sửa đổi (closed for modification).

Với cách tiếp cận ban đầu, các lớp **Circle** và **Square** đã có sẵn các phương thức `draw()` và `calculateArea()`. Khi cần thêm chức năng `exportToXml()`, ta buộc phải chỉnh sửa interface **Shape** để thêm phương thức mới, rồi sau đó phải cập nhật tất cả các lớp con (**Circle**, **Square**) để hiện thực phương thức này.

Điều này vi phạm nguyên tắc OCP, vì mỗi lần có chức năng mới, ta lại phải sửa đổi tất cả các lớp hiện có, thay vì chỉ mở rộng hệ thống một cách độc lập.

### 2.3.2 Vi phạm Single Responsibility Principle (SRP)

Nguyên tắc: Mỗi lớp chỉ nên có một lý do để thay đổi.

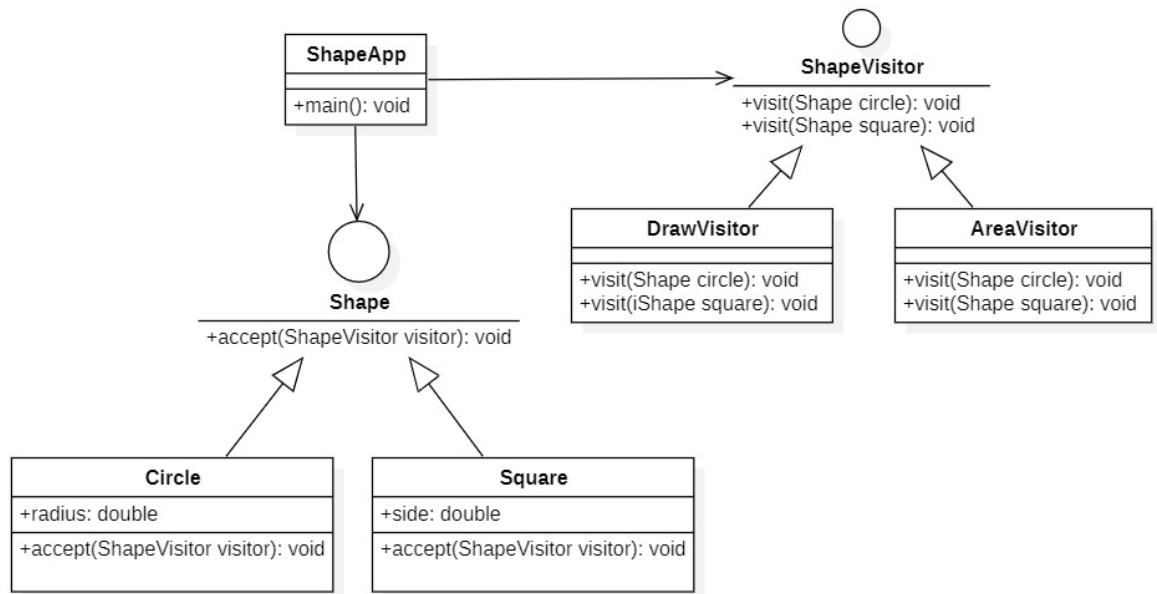
Trong cách tiếp cận ban đầu, các lớp Circle và Square vừa chịu trách nhiệm vẽ hình, tính diện tích, và bây giờ còn phải xuất dữ liệu sang XML. Điều này khiến các lớp trở nên cồng kềnh và khó bảo trì.

Nếu ta cần thay đổi cách vẽ, cách tính diện tích, hoặc định dạng xuất dữ liệu, ta phải sửa đổi trực tiếp trong Circle và Square, dẫn đến nguy cơ ảnh hưởng đến các phần khác của chương trình. Điều này vi phạm nguyên tắc SRP vì một lớp chỉ có một lý do để thay đổi mà phải thay đổi khi có bất kỳ yêu cầu mới nào liên quan đến hình học hoặc xuất dữ liệu.

## 2.4 Áp mẫu thiết kế vào ví dụ

### 2.4.1 Áp dụng vào ví dụ ShapeApp

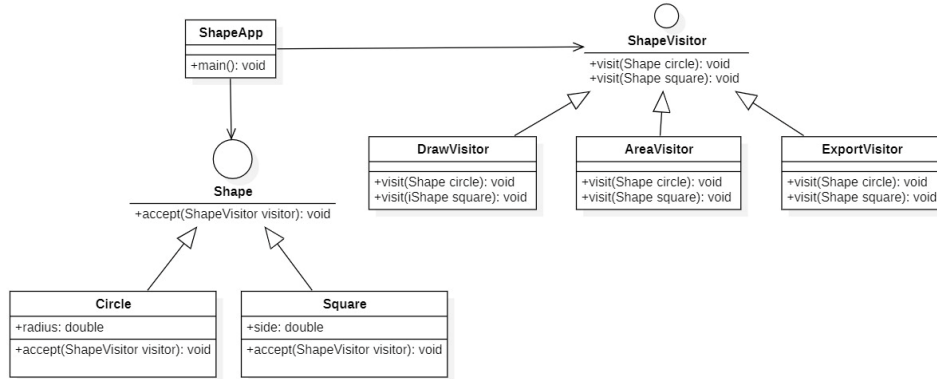
Ta có thể áp dụng Visitor vào yêu cầu trên:



Hình 2. 4 Sơ đồ lớp ShapeApp với các tiếp cận Visitor

Quan sát sơ đồ lớp, ta thấy Circle và Square chỉ chịu trách nhiệm chứa dữ liệu, không phải lo về cách vẽ, tính toán. Đồng thời, mỗi Visitor chỉ làm một nhiệm vụ duy nhất (đúng theo SRP).

Khi cần thêm chức năng mới, chỉ cần thêm một Visitor mới mà không động vào Circle hay Square. Chẳng hạn khi ta thêm ExportVisitor thì hệ thống sẽ như sau:



Hình 2. 5 Sơ đồ lớp ShapeApp với các tiếp cận Visitor mở rộng

Lúc này Circle/Square chỉ còn chức năng lưu trữ các thuộc tính, các thao tác vẽ, tính diện tích, xuất file được chuyển sang các Visitor cụ thể. Khi mở rộng ta chỉ cần thêm 1 Visitor mới, điều này giúp ta không còn vi phạm các nguyên tắc thiết kế

#### 2.4.2 Các use case thực tế có thể áp dụng

Visitor Pattern hữu ích khi cần áp dụng nhiều hành vi lên một cấu trúc dữ liệu phức tạp mà không muốn sửa đổi lớp gốc, ta có thể áp dụng vào:

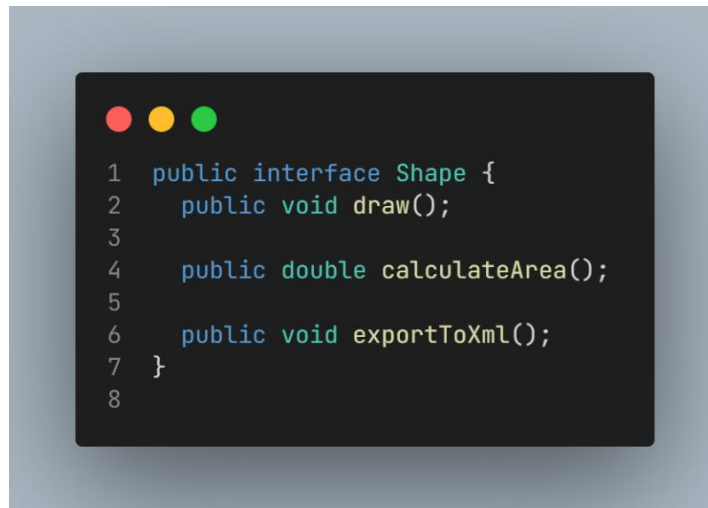
- Trình biên dịch & phân tích cú pháp: Visitor Pattern được sử dụng để duyệt cây cú pháp (AST) và thực hiện các thao tác như kiểm tra lỗi, tối ưu hóa mã, hoặc sinh mã máy.
- Hệ thống xử lý tài liệu: Dùng để xuất nội dung sang nhiều định dạng khác nhau như PDF, HTML, XML mà không cần thay đổi lớp tài liệu gốc.
- Quản lý phần tử trong game: Mô hình hóa các đối tượng trong game như quái vật, người chơi, vật phẩm, giúp dễ dàng thêm các hành vi như tính sát thương, kiểm tra va chạm, hoặc cập nhật trạng thái.

- Công cụ kiểm tra mã nguồn (Static Analysis Tools): Dùng để phân tích mã nguồn, tìm lỗi, hoặc tính toán độ phức tạp của mã mà không cần sửa đổi cấu trúc mã gốc.
- Quản lý hệ thống tệp (File System Visitor): Duyệt qua thư mục và tệp để thực hiện các thao tác như tính tổng kích thước, tìm kiếm, hoặc áp dụng quyền truy cập.

## 2.5 Source code

Link: <https://github.com/dukelw/visitor-proxy-design-pattern>

### 2.5.1 Source code với cách áp dụng truyền thống

A screenshot of a code editor with a dark background and light-colored text. The editor shows a Java interface named 'Shape'. The code is as follows:

```
1 public interface Shape {  
2     public void draw();  
3  
4     public double calculateArea();  
5  
6     public void exportToXml();  
7 }  
8
```

Hình 2. 6 Interface Shape

```
1 public class Circle implements Shape {
2     double radius;
3
4     Circle(double radius) {
5         this.radius = radius;
6     }
7
8     public void draw() {
9         System.out.println("Drawing a circle with radius: " + radius);
10    }
11
12    public double calculateArea() {
13        return Math.PI * radius * radius;
14    }
15
16    public void exportToXml() {
17        System.out.println("Exporting a circle with radius: " + radius);
18    }
19 }
```

Hình 2. 7 Lớp Circle

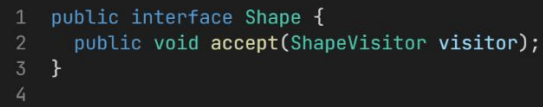
```
1 public class Square implements Shape {
2     double side;
3
4     Square(double side) {
5         this.side = side;
6     }
7
8     public void draw() {
9         System.out.println("Drawing a square with side: " + side);
10    }
11
12    public double calculateArea() {
13        return side * side;
14    }
15
16    public void exportToXml() {
17        System.out.println("Exporting a square with side: " + side);
18    }
19 }
```

Hình 2. 8 Lớp Square

```
1 public class TraditionalApproach {
2     public static void main(String[] args) {
3         Shape circle = new Circle(5);
4         Shape square = new Square(4);
5
6         circle.draw();
7         System.out.println("Area of circle: " + circle.calculateArea());
8
9         square.draw();
10        System.out.println("Area of square: " + square.calculateArea());
11
12        circle.exportToXml();
13        square.exportToXml();
14    }
15 }
```

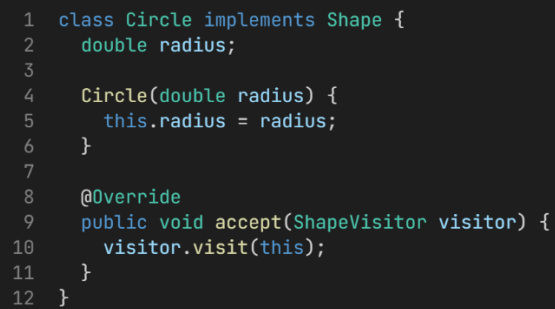
Hình 2. 9 Lớp Test

### 2.5.2 Source code áp dụng Visitor



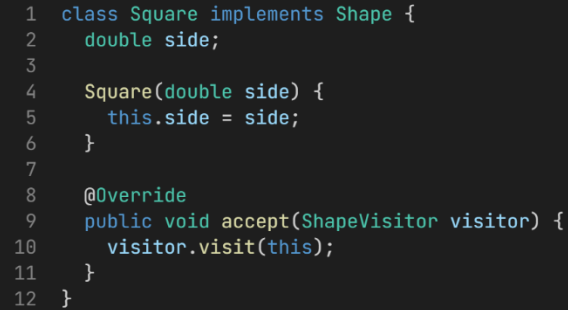
```
1 public interface Shape {  
2     public void accept(ShapeVisitor visitor);  
3 }  
4
```

Hình 2. 10 Interface Shape



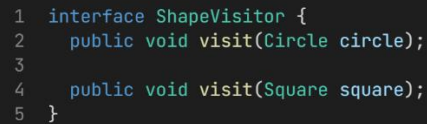
```
1 class Circle implements Shape {  
2     double radius;  
3  
4     Circle(double radius) {  
5         this.radius = radius;  
6     }  
7  
8     @Override  
9     public void accept(ShapeVisitor visitor) {  
10         visitor.visit(this);  
11     }  
12 }
```

Hình 2. 11 Lớp Circle



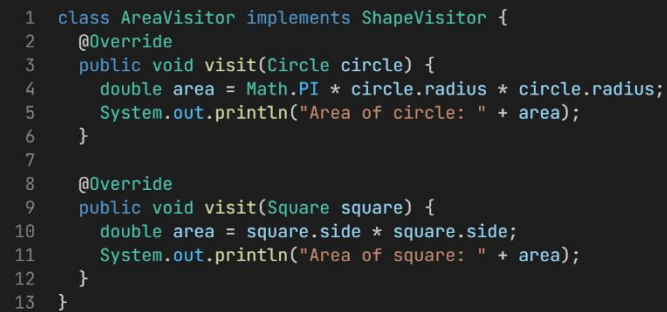
```
1 class Square implements Shape {
2     double side;
3
4     Square(double side) {
5         this.side = side;
6     }
7
8     @Override
9     public void accept(ShapeVisitor visitor) {
10        visitor.visit(this);
11    }
12 }
```

Hình 2. 12 Lớp Square



```
1 interface ShapeVisitor {
2     public void visit(Circle circle);
3
4     public void visit(Square square);
5 }
```

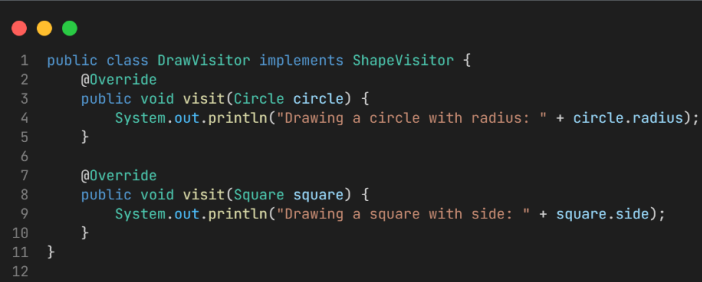
Hình 2. 13 Interface ShapeVisitor



```
1 class AreaVisitor implements ShapeVisitor {
2     @Override
3     public void visit(Circle circle) {
4         double area = Math.PI * circle.radius * circle.radius;
5         System.out.println("Area of circle: " + area);
6     }
7
8     @Override
9     public void visit(Square square) {
10        double area = square.side * square.side;
11        System.out.println("Area of square: " + area);
12    }
13 }
```

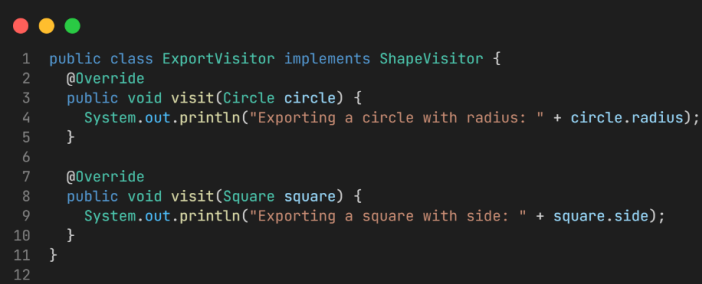
Hình 2. 14 Lớp AreaVisitor





```
1 public class DrawVisitor implements ShapeVisitor {
2     @Override
3     public void visit(Circle circle) {
4         System.out.println("Drawing a circle with radius: " + circle.radius);
5     }
6
7     @Override
8     public void visit(Square square) {
9         System.out.println("Drawing a square with side: " + square.side);
10    }
11 }
12
```

Hình 2. 15 Lớp DrawVisitor



```
1 public class ExportVisitor implements ShapeVisitor {
2     @Override
3     public void visit(Circle circle) {
4         System.out.println("Exporting a circle with radius: " + circle.radius);
5     }
6
7     @Override
8     public void visit(Square square) {
9         System.out.println("Exporting a square with side: " + square.side);
10    }
11 }
12
```

Hình 2. 16 Lớp ExportVisitor



Hinhg 2. 17 Lớp Test

## 2.6 Tóm tắt và kết luận

### 2.6.1 Tóm tắt

Visitor Pattern là một mẫu thiết kế hành vi giúp tách biệt logic xử lý khỏi cấu trúc dữ liệu, cho phép dễ dàng mở rộng các thao tác mà không cần chỉnh sửa các lớp hiện có, giúp khắc phục vấn đề của cách tiếp cận truyền thống như vi phạm Open/Closed Principle (OCP) và Single Responsibility Principle (SRP) bằng cách định nghĩa một interface Visitor để nhóm các hành vi cần thực hiện trên các đối tượng, các lớp dữ liệu chỉ cần triển khai phương thức `accept(visitor)` mà không cần thay đổi khi có hành vi mới từ đó giúp dễ dàng mở rộng bằng cách thêm Visitor mới mà không cần sửa đổi cấu trúc dữ liệu hiện tại.

### 2.6.2 Kết luận

Visitor Pattern là một giải pháp hữu ích khi cần thêm nhiều thao tác trên một tập hợp đối tượng mà không làm thay đổi cấu trúc của chúng. Nó giúp tách biệt logic, tăng

tính mở rộng, và bảo trì dễ dàng hơn. Tuy nhiên, mẫu này cũng có nhược điểm là tăng độ phức tạp do cần tạo nhiều lớp Visitor, nên chỉ nên sử dụng khi thực sự cần thiết.

## TÀI LIỆU THAM KHẢO

### Tiếng Việt

1. Visitor Design Pattern - Trợ thủ đắc lực của Developers - <https://s.pro.vn/S2vi>. Truy cập: 29/03/2025.
2. Proxy Design Pattern - Trợ thủ đắc lực của Developers - <https://s.pro.vn/DhMc>. Truy cập: 29/03/2025.

### Tiếng Anh

1. Visitor design pattern - <https://short.com.vn/jKcB>. Truy cập: 29/03/2025.
2. Visitor pattern - <https://s.pro.vn/I3Po>. Truy cập: 29/03/2025.
3. Visitor - <https://short.com.vn/01u6>. Truy cập: 29/03/2025.
4. Visitor - <https://s.pro.vn/7rbZ>. Truy cập: 29/03/2025.
5. Proxy Design Pattern - <https://s.pro.vn/K5gy>. Truy cập: 29/03/2025.
6. Proxy - <https://short.com.vn/Uk9e>. Truy cập: 29/03/2025.