Proxy Pattern

Nhóm 1
Phan Hoàng Minh Quân – 20520713
Trần Gia Bảo – 21521863
Trần Đông Đông - 21521957

Nội dung

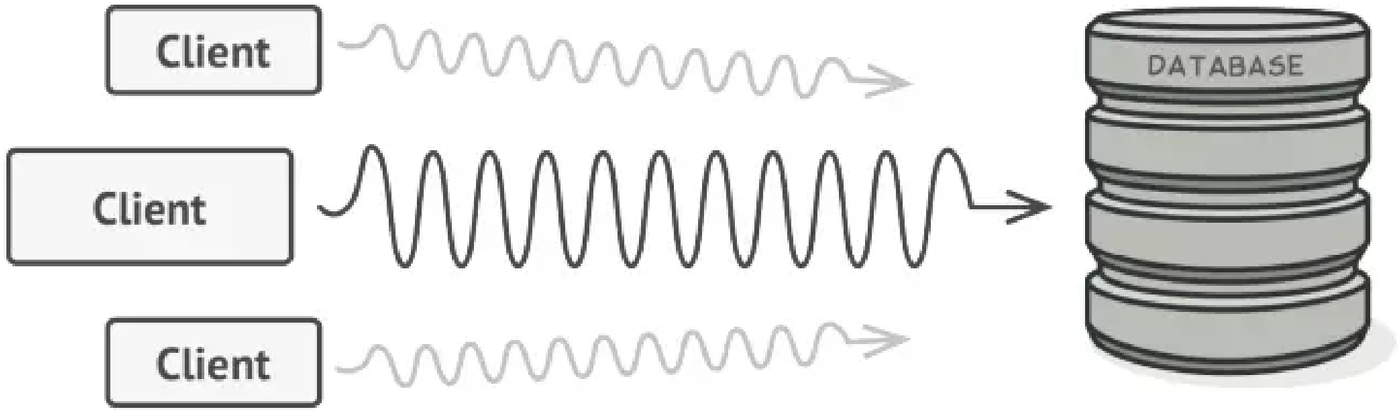
1. Tổng quan
   - Tên
   - Mô tả ngắn về mẫu
   - Phân loại
2. Motivation
3. SOLID principle in Builder
4. Cấu trúc mẫu và mô tả + ví dụ minh họa
5. Các bước hiện thực mẫu + code minh họa cho ví dụ trên
6. Ưu điểm
7. Nhược điểm
8. Áp dụng thực tiễn
9. Các mẫu liên quan
10. Link source code ví dụ và nguồn tài liệu

1. Tổng quan

- Tên mẫu: Proxy
- Phân Loại: Structural Pattern
- Mô tả:
  - Proxy Pattern là mẫu thiết kế mà ở đó tất cả các truy cập trực tiếp đến một đối tượng nào đó sẽ được chuyển hướng vào một đối tượng trung gian (Proxy Class). Mẫu Proxy (người đại diện) đại diện cho một đối tượng khác thực thi các phương thức, phương thức đó có thể được định nghĩa lại cho phù hợp với múc đích sử dụng.

2. Motivation

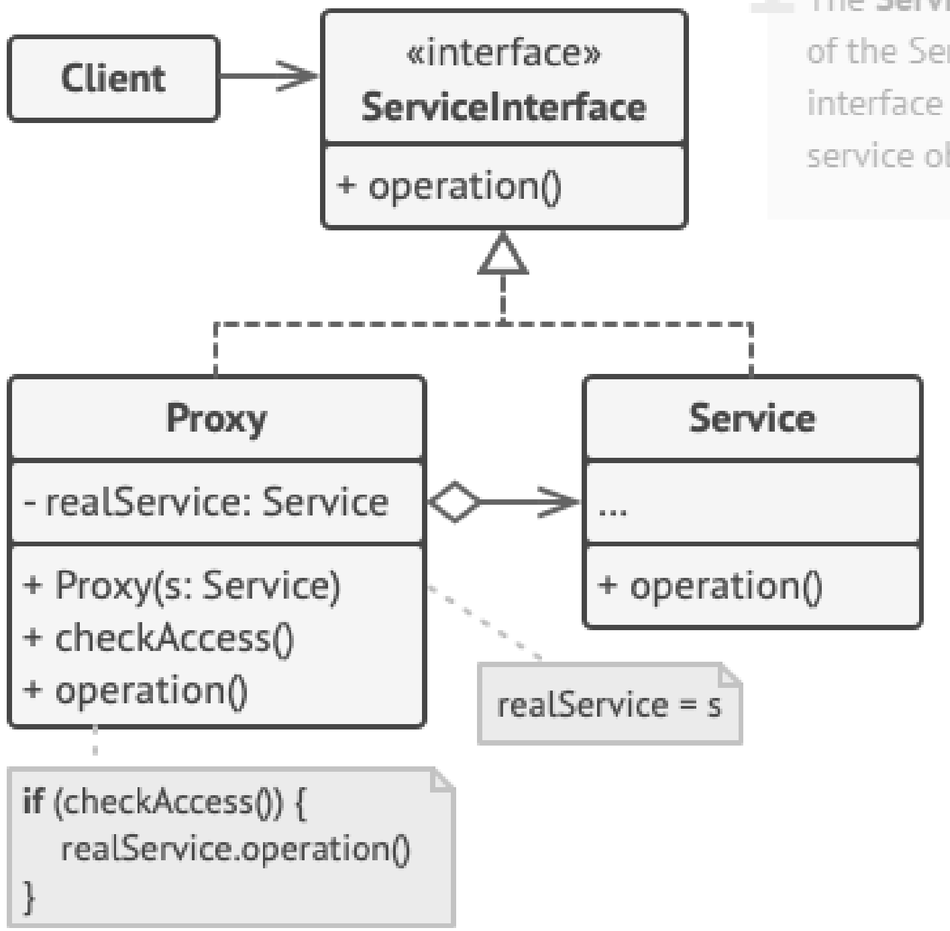
- Giả sử ta có một bài toán truy cập vào 1 object lớn. Object này chiếm nhiều tài nguyên hệ thống. Ta cần nó thường xuyên, nhưng không phải luôn luôn. Ví dụ như khi ta truy vấn cơ sở dữ liệu.
- Ta có thể implement lazy initialization, tức là chỉ tạo khi cần. Khi đó client muốn truy cập đều phải chạy qua đoạn code này, tuy nhiên vấn đề phát sinh là sẽ khiến code duplicate



- Trong lí tưởng, ta muốn đưa đoạn code này vào thẳng các lớp đối tượng, nhưng không phải lúc nào cũng thuận lợi. Ví dụ như khi lớp này có thể là một phần của thư viện đóng do bên thứ 3 cung cấp

4. Cấu trúc

4 The **Client** should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.



1 The **Service Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.

3 The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.

Usually, proxies manage the full lifecycle of their service objects.

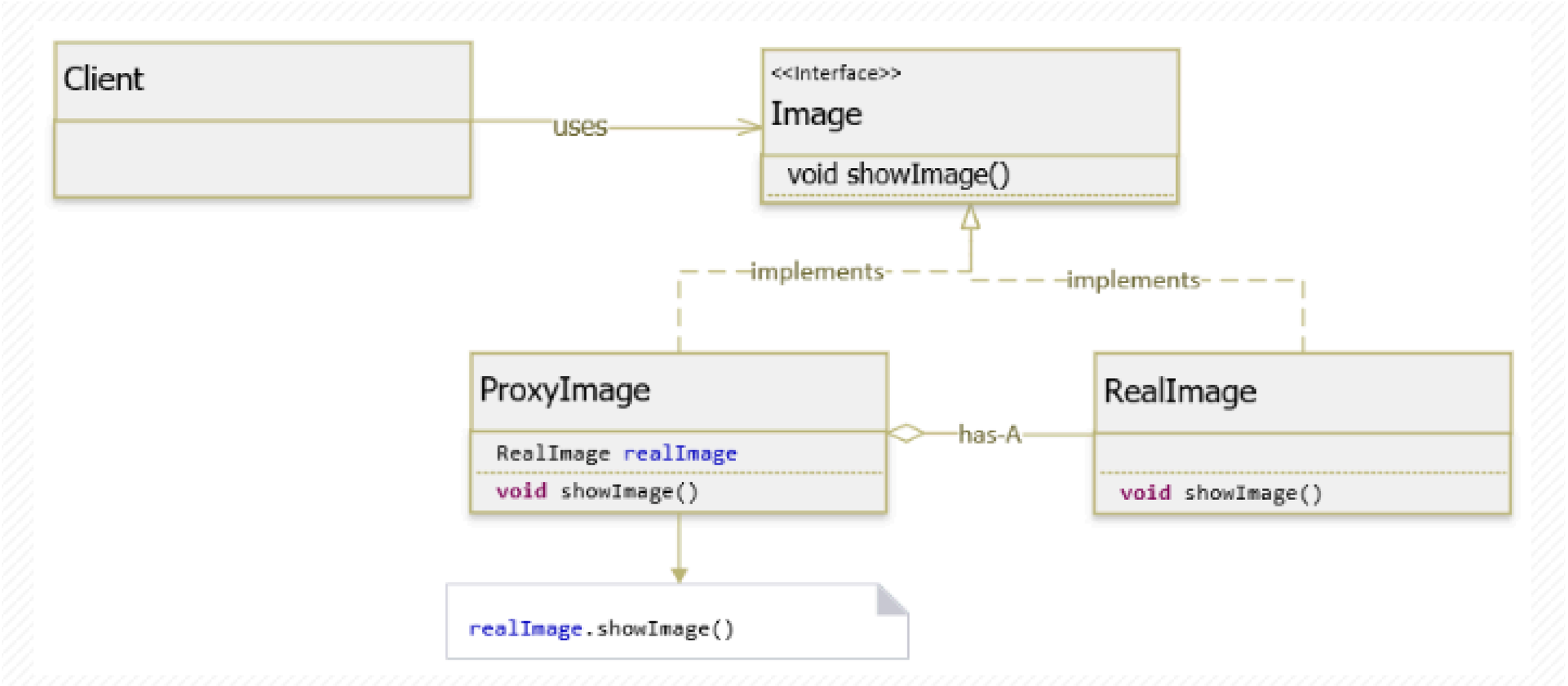
2 The **Service** is a class that provides some useful business logic.

4. Cấu trúc

Một Proxy Pattern bao gồm các thành phần sau:

- **Service Interface**: Đây là một interface chung (còn được gọi là ServiceInterface) mà cả Proxy và Service đều tuân theo. Interface này khai báo một phương thức operation(). Việc này cho phép Proxy ngụy trang mình như một đối tượng dịch vụ.
- **Service**: Đây là lớp cung cấp các logic nghiệp vụ thực sự. Service hiện thực hóa ServiceInterface và thực hiện phương thức operation().
- **Proxy**: Đây là lớp trung gian giữa Client và Service. Proxy có một trường tham chiếu (realService) trỏ đến một đối tượng dịch vụ (Service). Khi client gọi operation(), proxy sẽ thực hiện các xử lý bổ sung như kiểm tra quyền truy cập (checkAccess()), ghi log, hoặc khởi tạo dịch vụ một cách lười biếng (lazy initialization). Sau đó, nó sẽ chuyển tiếp yêu cầu tới đối tượng dịch vụ thực nếu các điều kiện thỏa mãn.
- **Client**: Là thành phần làm việc với cả Service và Proxy thông qua ServiceInterface. Điều này cho phép client tương tác với Proxy như thể nó là Service.
-

5. Code Ví dụ (Cấu trúc)



5. Code Ví dụ

```java
public interface Image {
    void showImage();
}
```

5. Code Ví dụ

```java
public class RealImage implements Image {
    private String url;

    public RealImage(String url) {
        this.url = url;
        System.out.println("Image loaded: " + this.url);
    }

    @Override
    public void showImage() {
        System.out.println("Image showed: " + this.url);
    }
}
```

5. Code Ví dụ

```java
public class ProxyImage implements Image {
    private Image realImage;
    private String url;

    public ProxyImage(String url) {
        this.url = url;
        System.out.println("Image unloaded: " + this.url);
    }

    @Override
    public void showImage() {
        if (realImage == null) {
            realImage = new RealImage(this.url);
        } else {
            System.out.println("Image already existed: " + this.url)
        }
        realImage.showImage();
    }
}
```

5. Code Ví dụ

```
public class Main {
    public static void main(String[] args) {

        System.out.println("Init proxy image: ");
        ProxyImage proxyImage = new ProxyImage("favicon.icon");

        System.out.println("---");
        System.out.println("Call real service 1st: ");
        proxyImage.showImage();

        System.out.println("---");
        System.out.println("Call real service 2nd: ");
        proxyImage.showImage();
    }
}
```

5. Code ví dụ (Kết quả)

```
Init proxy image:
Image unloaded: favicon.icon
---
Call real service 1st:
Image loaded: favicon.icon
Image showed: favicon.icon
---
Call real service 2nd:
Image already existed: favicon.icon
Image showed: favicon.icon
```

6. Ưu điểm

- Cãi thiện Performance thông qua lazy loading, chỉ tải các tài nguyên khi chúng được yêu cầu.
- Nó cung cấp sự bảo vệ cho đối tượng thực từ thế giới bên ngoài.
- Giảm chi phí khi có nhiều truy cập vào đối tượng có chi phí khởi tạo ban đầu lớn.
- Dễ nâng cấp, bảo trì.

7. Nhược điểm

- Độ phức tạp tổng thể của mã tăng lên vì bạn cần xây dựng nhiều class mới.
- Tốc độ phản hồi có thể bị ảnh hưởng do gọi service qua một class trung gian

8. Áp dụng mẫu trong thực tiễn

Proxy với Lazy Initialization

- Trong Spring, proxy cũng được sử dụng để hỗ trợ tải lười biếng (lazy loading). Khi bạn sử dụng @Lazy, Spring sẽ không khởi tạo bean ngay lập tức, mà thay vào đó sẽ tạo một proxy. Chỉ khi nào bean này được truy cập lần đầu, proxy mới thực hiện việc khởi tạo thực tế.

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Component;

@Component
public class OrderService {

    private final PaymentService paymentService;

    @Autowired
    public OrderService(@Lazy PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    public void placeOrder() {
        System.out.println("Placing order...");
        paymentService.processPayment();
    }
}

@Component
public class PaymentService {

    public void processPayment() {
        System.out.println("Processing payment...");
    }
}
```

9. Các mẫu liên quan

- Với **Adapter**, bạn truy cập một đối tượng hiện có thông qua một giao diện khác. Với Proxy, giao diện vẫn giữ nguyên. Với Decorator, bạn truy cập đối tượng thông qua một giao diện được mở rộng.
- **Facade** tương tự như Proxy ở chỗ cả hai đều tạo bộ đệm cho một thực thể phức tạp và tự khởi tạo nó. Khác với Facade, Proxy có cùng giao diện với đối tượng dịch vụ của nó, điều này giúp chúng có thể hoán đổi cho nhau.
- **Decorator** và Proxy có cấu trúc tương tự nhau, nhưng mục đích rất khác nhau. Cả hai mẫu đều dựa trên nguyên lý hợp thành (composition principle), nơi mà một đối tượng sẽ ủy quyền một phần công việc cho một đối tượng khác. Sự khác biệt là Proxy thường tự quản lý vòng đời của đối tượng dịch vụ của nó, trong khi thành phần của Decorators luôn được điều khiển bởi client.

10. Link source code ví dụ và nguồn tài liệu

- Link: phmquan/design-pattern (github.com)
- Document
  - Dive into Design Patterns
  - https://refactoring.guru/design-patterns/proxy
  - https://gpcoder.com/4644-huong-dan-java-design-pattern-proxy/