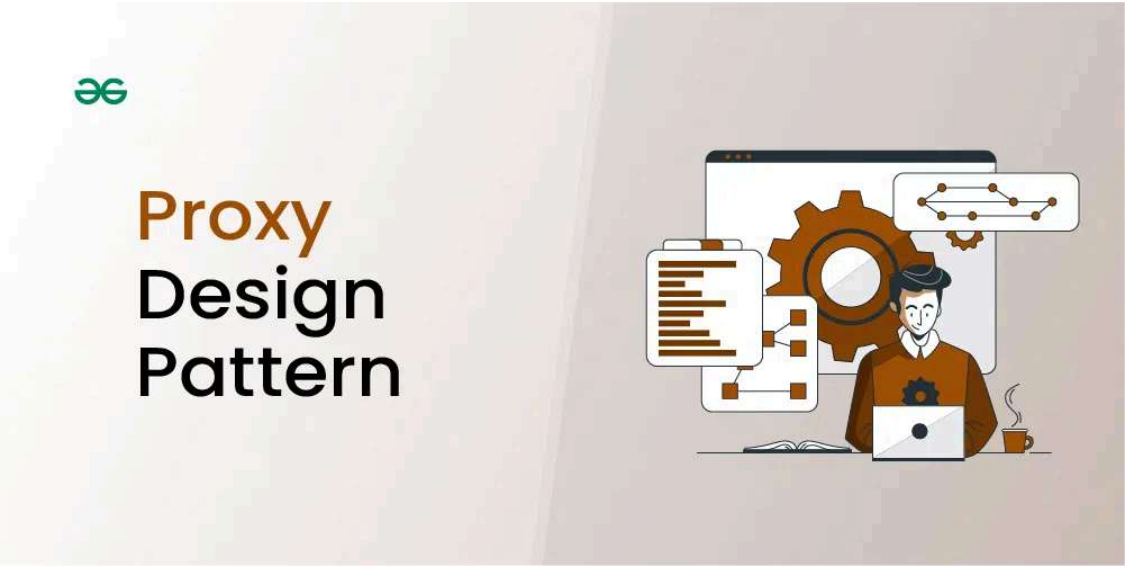
Proxy Design Pattern

Last Updated : 03 Jan, 2025

The Proxy Design Pattern a structural design pattern is a way to use a placeholder object to control access to another object. Instead of interacting directly with the main object, the client talks to the proxy, which then manages the interaction. This is useful for things like controlling access, delaying object creation until it's needed (lazy initialization), logging, or adding security checks.



A real-world example can be a cheque or credit card as a proxy for what is in our bank account. It can be used in place of cash and provides a means of accessing that cash when required.

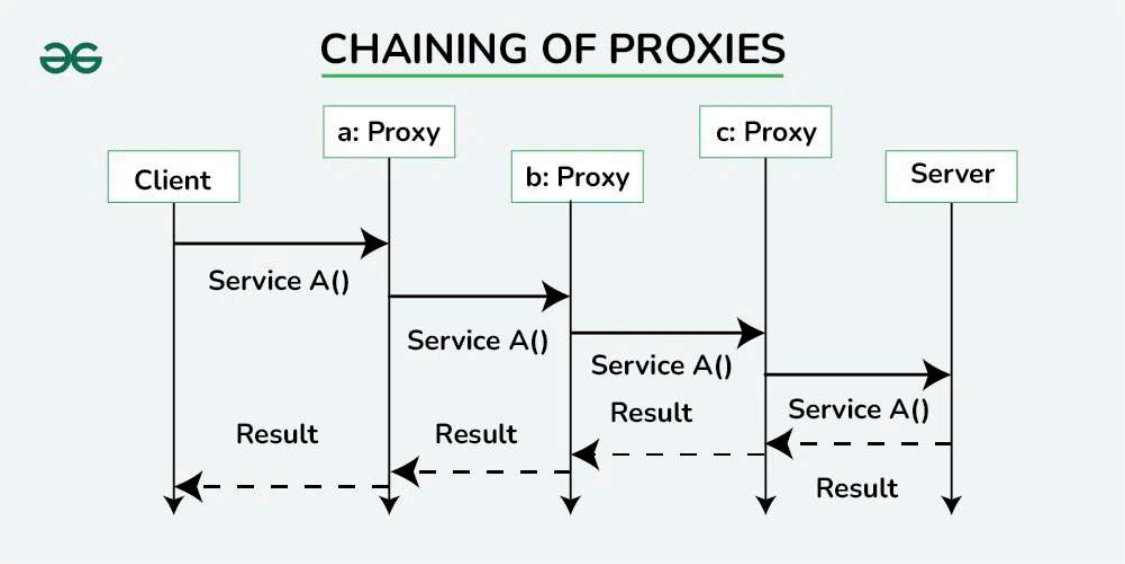
Table of Content

What is Proxy Design Pattern?

The Proxy Design Pattern is a design pattern in which the client and the actual object are connected by a proxy object. The client communicates with the proxy, which manages access to the real object, rather than the real object directly. Before sending the request to the real object, the proxy can take care of additional tasks like caching, security, logging, and lazy loading.

Chaining of Proxies

Chaining proxies in the Proxy Design Pattern means connecting them in a sequence, where each proxy adds its behavior or checks before passing the request to the next proxy or the real object. It's like forming a chain of guards, each responsible for a specific task.



Components of Proxy Design Pattern

1. Subject

The `Subject` is an interface or an abstract class that defines the common interface shared by the `RealSubject` and `Proxy` classes. It declares the methods that the `Proxy` uses to control access to the `RealSubject`.

- Declares the common interface for both `RealSubject` and `Proxy`.
- Usually includes the methods that the client code can invoke on the `RealSubject` and the `Proxy`.

2. RealSubject

The `RealSubject` is the actual object that the `Proxy` represents. It contains the real implementation of the business logic or the resource that the client code wants to access.

- It Implements the operations declared by the `Subject` interface.
- Represents the real resource or object that the `Proxy` controls access to.

3. Proxy

The `Proxy` acts as a surrogate or placeholder for the `RealSubject`. It controls access to the real object and may provide additional functionality such as lazy loading, access control, or logging.

- Implements the same interface as the `RealSubject` (Subject).
- Maintains a reference to the `RealSubject`.
- Controls access to the `RealSubject`, adding additional logic if necessary.

How to implement Proxy Design Pattern?

Below are the simple steps to implement the Proxy Design Pattern:

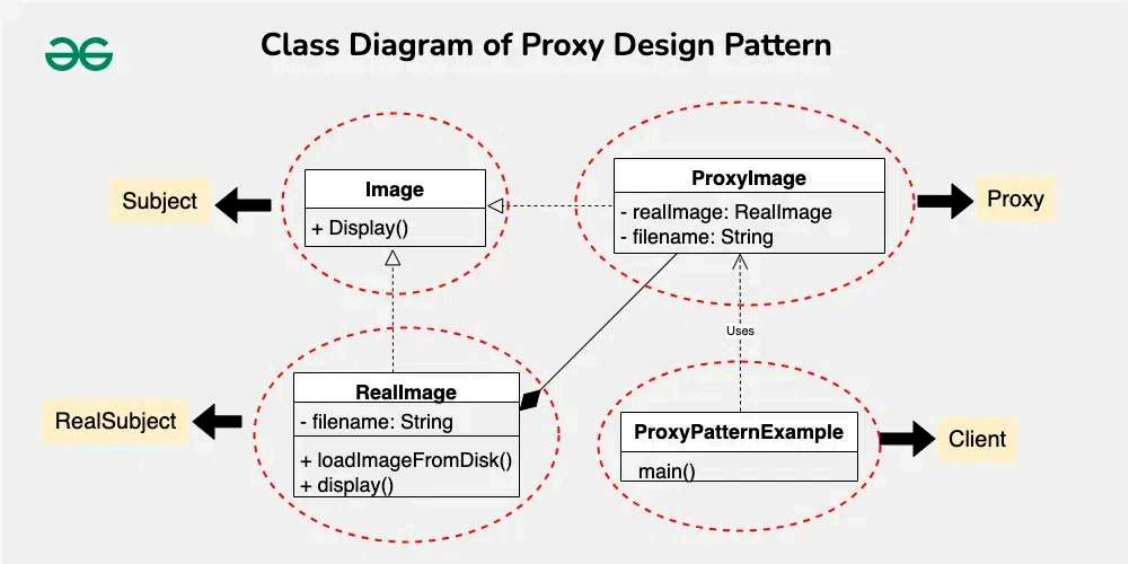
1. **Create the Real Object Interface**: Define an interface or abstract class that represents the operations the real object will provide. Both the real object and proxy will implement this interface.
2. **Create the Real Object**: This class implements the interface and contains the actual logic or operation that the client wants to use.
3. **Create the Proxy Class**: The proxy class also implements the same interface as the real object. It holds a reference to the real object and controls access to it. The proxy can add extra logic like logging, caching, or security checks before calling the real object's methods.
4. **Client Uses the Proxy**: Instead of creating the real object directly, the client interacts with the proxy. The proxy decides when and how to forward the client's request to the real object.

Proxy Design Pattern example (with implementation)

Problem Statement:

> *Consider a scenario where your application needs to load and display images, and you want to optimize the image loading process. Loading images from disk or other external sources can be resource-intensive, especially if the images are large or stored remotely.*

To address this issue, we need to implement the Proxy Design Pattern to control the access and loading of images.



1. Subject (Image Interface):

The `Image` interface declares the common methods for displaying images, acting as a blueprint for both the real and proxy objects. In this design, it defines the `display()` method that both `RealImage` and `ProxyImage` must implement. This ensures a uniform interface for clients interacting with image objects.

```
// Subject
interface Image {
    void display();
}
```

2. RealSubject (RealImage Class):

The `RealImage` class represents the real object that the proxy will control access to.

- It implements the `Image` interface, providing concrete implementations for loading and displaying images from disk.
- The constructor initializes the image file name, and the `display()` method is responsible for loading the image if not already loaded and then displaying it.

```
// RealSubject
class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    private void loadImageFromDisk() {
        System.out.println("Loading image: " + filename);
    }

    public void display() {
        System.out.println("Displaying image: " + filename);
    }
}
```

3. Proxy (ProxyImage Class):

The `ProxyImage` class acts as a surrogate for the `RealImage`. It also implements the `Image` interface, maintaining a reference to the real image object.

- The `display()` method in the proxy checks whether the real image has been loaded; if not, it creates a new instance of `RealImage` and delegates the `display()` call to it.
- This lazy loading mechanism ensures that the real image is loaded only when necessary.

```
// Proxy
class ProxyImage implements Image {
    private RealImage realImage;
    private String filename;

    public ProxyImage(String filename) {
        this.filename = filename;
    }

    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}
```

4. Client Code:

The client code (`ProxyPatternExample`) demonstrates the usage of the Proxy Design Pattern. It creates an `Image` object, which is actually an instance of `ProxyImage`.

- The client invokes the `display()` method on the proxy.
- The proxy, in turn, controls access to the real image, ensuring that it is loaded from disk only when needed.
- Subsequent calls to `display()` use the cached image in the proxy, avoiding redundant loading and improving performance.

```
// Client code
public class ProxyPatternExample {
    public static void main(String[] args) {
        Image image = new ProxyImage("example.jpg");

        // Image will be loaded from disk only when
display() is called
        image.display();
```

```
        // Image will not be loaded again, as it has been cached in the Proxy
        image.display();
    }
}
```

5. Complete Code of the above example:

This code demonstrates how the Proxy Pattern efficiently manages the loading and displaying of images by introducing a proxy that controls access to the real image object, providing additional functionality such as lazy loading.

```
// Subject
interface Image {
    void display();
}

// RealSubject
class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    private void loadImageFromDisk() {
        System.out.println("Loading image: " + filename);
    }

    public void display() {
        System.out.println("Displaying image: " + filename);
    }
}

// Proxy
class ProxyImage implements Image {
    private RealImage realImage;
```

```
    private String filename;

    public ProxyImage(String filename) {
        this.filename = filename;
    }

    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}

// Client code
public class ProxyPatternExample {
    public static void main(String[] args) {
        Image image = new ProxyImage("example.jpg");

        // Image will be loaded from disk only when display() is called
        image.display();

        // Image will not be loaded again, as it has been cached in the Proxy
        image.display();
    }
}
```

```
Loading image: example.jpg
Displaying image: example.jpg
Displaying image: example.jpg
```

Why do we need Proxy Design Pattern?

The Proxy Design Pattern is employed to address various concerns and scenarios in software development, providing a way to control access to objects, add functionality, or optimize performance.

- **Lazy Loading:**

    - One of the primary use cases for proxies is lazy loading. In situations where creating or initializing an object is resource-intensive, the proxy delays the creation of the real object until it is actually needed.
    - This can lead to improved performance by avoiding unnecessary resource allocation.
- **Access Control:**
    - Proxies can enforce access control policies.
    - By acting as a gatekeeper to the real object, proxies can restrict access based on certain conditions, providing security or permission checks.
- **Protection Proxy:**
    - Protection proxies control access to a real object by adding an additional layer of security checks.
    - They can ensure that the client code has the necessary permissions before allowing access to the real object.
- **Caching:**
    - Proxies can implement caching mechanisms to store results or resources.
    - This is particularly useful when repeated operations on a real object can be optimized by caching previous results, avoiding redundant computations or data fetching.
- **Logging and Monitoring:**
    - Proxies provide a convenient point to add logging or monitoring functionalities.
    - By intercepting method calls to the real object, proxies can log information, track usage, or measure performance without modifying the real object.

When to use Proxy Design Pattern?

- Use a proxy when you want to postpone the creation of a resource-intensive object until it's actually needed.

- Use a proxy when you need to control and manage access to an object, ensuring that certain conditions or permissions are met before allowing clients to interact with the real object.
- Use a proxy to optimize the utilization of resources, such as caching results or storing previously fetched data. This can lead to performance improvements by avoiding redundant computations or data retrieval.
- Use a proxy when dealing with distributed systems and you want to interact with objects located in different addresses or systems.
- The proxy can handle the communication details, making remote object interaction more seamless.

When not to use Proxy Design Pattern?

- **Overhead for Simple Operations:** Avoid using a proxy for simple objects or operations that don't involve resource-intensive tasks. Introducing a proxy might add unnecessary complexity in such cases.
- **Unnecessary Abstraction:** If your application doesn't require lazy loading, access control, or additional functionalities provided by proxies, introducing proxies may lead to unnecessary abstraction and code complexity.
- **Performance Impact:** If the introduction of a proxy negatively impacts performance rather than improving it, especially in cases where objects are lightweight and creation is not a significant overhead.
- **When Access Control Isn't Needed:** If there are no access control requirements and the client code can directly interact with the real object without any restrictions.
- **When Eager Loading is Acceptable:** If eager loading of objects is acceptable and doesn't affect the performance of the system, introducing a proxy for lazy loading might be unnecessary.

Next Article

Behavioral Design Patterns

Similar Reads

Software Design Patterns Tutorial

Software design patterns are important tools developers, providing proven solutions to common problems encountered during software development. This article will act as tutorial to help you understand the concep...

9 min read

Complete Guide to Design Patterns

Design patterns help in addressing the recurring issues in software design and provide a shared vocabulary for developers to communicate and collaborate effectively. They have been documented and refined over tim...

11 min read

Types of Software Design Patterns

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our...

9 min read

1. Creational Design Patterns

Creational Design Patterns

Creational Design Patterns focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed, and represented. Creational...

4 min read

Types of Creational Patterns

2. Structural Design Patterns

Structural Design Patterns

Structural Design Patterns are solutions in software design that focus on how classes and objects are organized to form larger, functional structures. These patterns help developers simplify relationships betwee...

7 min read

Types of Structural Patterns

Adapter Design Pattern

One structural design pattern that enables the usage of an existing class's interface as an additional interface is the adapter design pattern. To make two incompatible interfaces function together, it serves as a bridge....

8 min read

Bridge Design Pattern

The Bridge design pattern allows you to separate the abstraction from the implementation. It is a structural design pattern. There are 2 parts in Bridge design pattern : AbstractionImplementationThis is a design...

4 min read

Composite Method | Software Design Pattern

Composite Pattern is a structural design pattern that allows you to compose objects into tree structures to represent part-whole hierarchies. The main idea behind the Composite Pattern is to build a tree structure of...

9 min read

Decorator Design Pattern

The Decorator Design Pattern is a structural design pattern that allows behavior to be added to individual objects dynamically, without affecting the behavior of other objects from the same class. It involves creating...

9 min read

Facade Method Design Pattern

Facade Method Design Pattern is a part of the Gang of Four design patterns and it is categorized under Structural design patterns. Before we go into the details, visualize a structure. The house is the facade, it is...

8 min read

Flyweight Design Pattern

The Flyweight design pattern is a structural pattern that optimizes memory usage by sharing a common state among multiple objects. It aims to reduce the number of objects created and to decrease memory...

10 min read

Proxy Design Pattern

The Proxy Design Pattern a structural design pattern is a way to use a placeholder object to control access to another object. Instead of interacting directly with the main object, the client talks to the proxy, which then...

9 min read

3. Behvioural Design Patterns

Behavioral Design Patterns

Behavioral design patterns are a category of design patterns that focus on the interactions and communication between objects. They help define how objects collaborate and distribute responsibility among them, makin...

5 min read

3. Types of Behvioural Patterns

Top Design Patterns Interview Questions [2024]

A design pattern is basically a reusable and generalized solution to a common problem that arises during software design and development. Design patterns are not specific to a particular programming language or...

9 min read

Software Design Pattern in Different Programming Languages

Java Design Patterns Tutorial

Design patterns in Java refer to structured approaches involving objects and classes that aim to solve recurring design issues within specific contexts. These patterns offer reusable, general solutions to common problems...

8 min read

Python Design Patterns Tutorial

Design patterns in Python are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Modern C++ Design Patterns Tutorial

Design patterns in C++ help developers create maintainable, flexible, and understandable code. They encapsulate the expertise and experience of seasoned software architects and developers, making it easier f...

7 min read

JavaScript Design Patterns Tutorial

Design patterns in Javascipt are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Software Design Pattern Books

Software Design Pattern in Development

Some other Popular Design Patterns

Design Patterns in Different Languages

GeeksforGeeks
Sanchhaya Education Private Limited
Corporate & Communications Address:
A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305)

Registered Address:
K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305