Proxy Design Pattern - Complete Java Guide

Overview

The Proxy Design Pattern provides a placeholder or surrogate for another object to control access to it. It acts as an intermediary between the client and the real object, allowing you to perform additional operations before or after requests are forwarded to the real object.

Three Main Types of Proxy Patterns

1. Virtual Proxy

Purpose: Controls access to expensive-to-create objects by deferring their creation until actually needed (lazy loading).

Use Cases:

- Loading large images or files
- Database connections
- Heavy computational objects

Example - Image Loading:

```java
```

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

// Real Subject
class RealImage implements Image {
    private String filename;

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

    private void loadFromDisk() {
        System.out.println("Loading " + filename + " from disk...");
        try {
            Thread.sleep(2000); // Simulate expensive loading operation
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        System.out.println(filename + " loaded successfully!");
    }

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

// Virtual Proxy
class VirtualImageProxy implements Image {
    private String filename;
    private RealImage realImage;

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

    @Override
    public void display() {
        if (realImage == null) {
            System.out.println("First access - creating real image...");
            realImage = new RealImage(filename);
        }
        realImage.display();
```

```
    }
}

// Demo class for Virtual Proxy
class VirtualProxyDemo {
    public static void main(String[] args) {
        System.out.println("=== Virtual Proxy Demo ===");

        // Image is not loaded yet
        Image image = new VirtualImageProxy("large_photo.jpg");
        System.out.println("Proxy created, but image not loaded yet\n");

        // Image gets loaded on first access
        image.display();
        System.out.println();

        // Subsequent calls use already loaded image
        image.display();
    }
}
```

Output:

```
=== Virtual Proxy Demo ===
Proxy created, but image not loaded yet

First access - creating real image...
Loading large_photo.jpg from disk...
large_photo.jpg loaded successfully!
Displaying large_photo.jpg

Displaying large_photo.jpg
```

2. Protection Proxy

Purpose: Controls access to the real object based on access rights, authentication, or authorization.

Use Cases:

- User authentication systems
- Role-based access control
- Resource protection

Example - Document Access Control:

```
java
```

```
// User roles enum
enum UserRole {
    ADMIN, USER, GUEST
}

// Subject interface
interface Document {
    String read();
    String write(String content);
    String delete();
}

// Real Subject
class RealDocument implements Document {
    private String name;
    private String content;

    public RealDocument(String name) {
        this.name = name;
        this.content = "Content of " + name;
    }

    @Override
    public String read() {
        return "Reading: " + content;
    }

    @Override
    public String write(String content) {
        this.content = content;
        return "Writing to " + name + ": " + content;
    }

    @Override
    public String delete() {
        return "Deleting document: " + name;
    }
}

// Protection Proxy
class ProtectionDocumentProxy implements Document {
    private RealDocument document;
    private UserRole userRole;

    public ProtectionDocumentProxy(RealDocument document, UserRole userRole) {
        this.document = document;
```

```
        this.userRole = userRole;
    }

    private boolean hasPermission(String operation) {
        switch (userRole) {
            case ADMIN:
                return true; // Admin has all permissions
            case USER:
                return operation.equals("read") || operation.equals("write");
            case GUEST:
                return operation.equals("read");
            default:
                return false;
        }
    }

    private void checkAccess(String operation) {
        if (!hasPermission(operation)) {
            throw new SecurityException("Access denied: " + userRole + " cannot " + operation);
        }
    }

    @Override
    public String read() {
        checkAccess("read");
        return document.read();
    }

    @Override
    public String write(String content) {
        checkAccess("write");
        return document.write(content);
    }

    @Override
    public String delete() {
        checkAccess("delete");
        return document.delete();
    }
}

// Demo class for Protection Proxy
class ProtectionProxyDemo {
    public static void main(String[] args) {
        System.out.println("=== Protection Proxy Demo ===");
        RealDocument realDoc = new RealDocument("secret_document.txt");
```

```
        // Admin user - full access
        Document adminProxy = new ProtectionDocumentProxy(realDoc, UserRole.ADMIN);
        System.out.println("Admin user:");
        System.out.println(adminProxy.read());
        System.out.println(adminProxy.write("Updated by admin"));
        System.out.println(adminProxy.delete());
        System.out.println();

        // Regular user - read and write only
        Document userProxy = new ProtectionDocumentProxy(realDoc, UserRole.USER);
        System.out.println("Regular user:");
        System.out.println(userProxy.read());
        System.out.println(userProxy.write("Updated by user"));
        try {
            System.out.println(userProxy.delete());
        } catch (SecurityException e) {
            System.out.println("Error: " + e.getMessage());
        }
        System.out.println();

        // Guest user - read only
        Document guestProxy = new ProtectionDocumentProxy(realDoc, UserRole.GUEST);
        System.out.println("Guest user:");
        System.out.println(guestProxy.read());
        try {
            System.out.println(guestProxy.write("Trying to write as guest"));
        } catch (SecurityException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

3. Remote Proxy

Purpose: Provides a local representative for an object that exists in a different address space (remote server, different process, etc.).

Use Cases:

- Web services and APIs
- Distributed systems
- Microservices communication

Example - API Service Proxy:

```
java
```

```java
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

// Subject interface
interface DataService {
    Map<String, Object> getUserData(String userId) throws Exception;
    boolean updateUserData(String userId, Map<String, Object> data) throws Exception;
}

// Real Subject (simulates remote service)
class RemoteDataService implements DataService {
    private String serverUrl;
    private Map<String, Map<String, Object>> remoteData;

    public RemoteDataService(String serverUrl) {
        this.serverUrl = serverUrl;
        initializeData();
    }

    private void initializeData() {
        remoteData = new HashMap<>();

        Map<String, Object> user1 = new HashMap<>();
        user1.put("name", "John Doe");
        user1.put("email", "john@example.com");
        remoteData.put("user1", user1);

        Map<String, Object> user2 = new HashMap<>();
        user2.put("name", "Jane Smith");
        user2.put("email", "jane@example.com");
        remoteData.put("user2", user2);
    }

    @Override
    public Map<String, Object> getUserData(String userId) throws Exception {
        System.out.println("Making remote API call to " + serverUrl + "/users/" + userId);
        simulateNetworkDelay();

        if (remoteData.containsKey(userId)) {
            return new HashMap<>(remoteData.get(userId));
        } else {
            throw new Exception("User " + userId + " not found");
        }
    }

    @Override
```

```
    public boolean updateUserData(String userId, Map<String, Object> data) throws Exception {
        System.out.println("Making remote API call to update " + serverUrl + "/users/" + userId);
        simulateNetworkDelay();

        if (remoteData.containsKey(userId)) {
            remoteData.get(userId).putAll(data);
            return true;
        }
        return false;
    }

    private void simulateNetworkDelay() {
        try {
            Thread.sleep(1000); // Simulate network delay
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

// Cache entry class
class CacheEntry {
    private Map<String, Object> data;
    private long timestamp;

    public CacheEntry(Map<String, Object> data) {
        this.data = new HashMap<>(data);
        this.timestamp = System.currentTimeMillis();
    }

    public Map<String, Object> getData() {
        return new HashMap<>(data);
    }

    public boolean isValid(long timeoutMs) {
        return (System.currentTimeMillis() - timestamp) < timeoutMs;
    }

    public void updateData(Map<String, Object> newData) {
        this.data.putAll(newData);
        this.timestamp = System.currentTimeMillis();
    }
}

// Remote Proxy with caching and error handling
class RemoteDataServiceProxy implements DataService {
    private RemoteDataService remoteService;
```

```
    private Map<String, CacheEntry> cache;
    private long cacheTimeoutMs;

    public RemoteDataServiceProxy(String serverUrl) {
        this.remoteService = new RemoteDataService(serverUrl);
        this.cache = new ConcurrentHashMap<>();
        this.cacheTimeoutMs = 30000; // 30 seconds
    }

    @Override
    public Map<String, Object> getUserData(String userId) throws Exception {
        // Check cache first
        CacheEntry cacheEntry = cache.get(userId);
        if (cacheEntry != null && cacheEntry.isValid(cacheTimeoutMs)) {
            System.out.println("Returning cached data for user " + userId);
            return cacheEntry.getData();
        }

        // Cache miss or expired - fetch from remote
        try {
            Map<String, Object> data = remoteService.getUserData(userId);
            // Update cache
            cache.put(userId, new CacheEntry(data));
            System.out.println("Data cached for user " + userId);
            return data;
        } catch (Exception e) {
            System.out.println("Remote call failed: " + e.getMessage());
            // Return cached data if available, even if expired
            if (cacheEntry != null) {
                System.out.println("Returning stale cached data for user " + userId);
                return cacheEntry.getData();
            }
            throw e;
        }
    }

    @Override
    public boolean updateUserData(String userId, Map<String, Object> data) throws Exception {
        try {
            boolean success = remoteService.updateUserData(userId, data);
            if (success) {
                // Update cache with new data
                CacheEntry cacheEntry = cache.get(userId);
                if (cacheEntry != null) {
                    cacheEntry.updateData(data);
                    System.out.println("Cache updated for user " + userId);
                }
```

```
            }
            return success;
        } catch (Exception e) {
            System.out.println("Update failed: " + e.getMessage());
            return false;
        }
    }
}

// Demo class for Remote Proxy
class RemoteProxyDemo {
    public static void main(String[] args) {
        System.out.println("=== Remote Proxy Demo ===");

        // Create proxy
        DataService service = new RemoteDataServiceProxy("https://api.example.com");

        try {
            // First call - will hit remote service
            System.out.println("First call:");
            Map<String, Object> userData = service.getUserData("user1");
            System.out.println("Retrieved: " + userData);
            System.out.println();

            // Second call - will use cache
            System.out.println("Second call (within cache timeout):");
            userData = service.getUserData("user1");
            System.out.println("Retrieved: " + userData);
            System.out.println();

            // Update data
            System.out.println("Updating user data:");
            Map<String, Object> updateData = new HashMap<>();
            updateData.put("email", "newemail@example.com");
            boolean success = service.updateUserData("user1", updateData);
            System.out.println("Update successful: " + success);
            System.out.println();

            // Verify update
            System.out.println("Verifying update (should use cache):");
            userData = service.getUserData("user1");
            System.out.println("Retrieved: " + userData);

        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
        }
```

```
    }
}
```

Complete Example with All Three Patterns

```java
// Unified demo showing all three patterns
public class ProxyPatternDemo {
    public static void main(String[] args) {
        System.out.println("============================================");
        System.out.println("       PROXY DESIGN PATTERN DEMO       ");
        System.out.println("============================================\n");

        // Virtual Proxy Demo
        VirtualProxyDemo.main(args);
        System.out.println("\n" + "=".repeat(50) + "\n");

        // Protection Proxy Demo
        ProtectionProxyDemo.main(args);
        System.out.println("\n" + "=".repeat(50) + "\n");

        // Remote Proxy Demo
        RemoteProxyDemo.main(args);
    }
}
```

Key Benefits of Proxy Pattern

1. Lazy Loading (Virtual Proxy)

- Improves performance by deferring expensive object creation
- Reduces memory usage for unused objects
- Faster application startup times

2. Access Control (Protection Proxy)

- Implements security and authorization
- Centralized access control logic
- Easy to modify permissions without changing core objects

3. Remote Access (Remote Proxy)

- Abstracts network complexity
- Provides caching and error handling

- Improves performance with local caching

When to Use Proxy Pattern

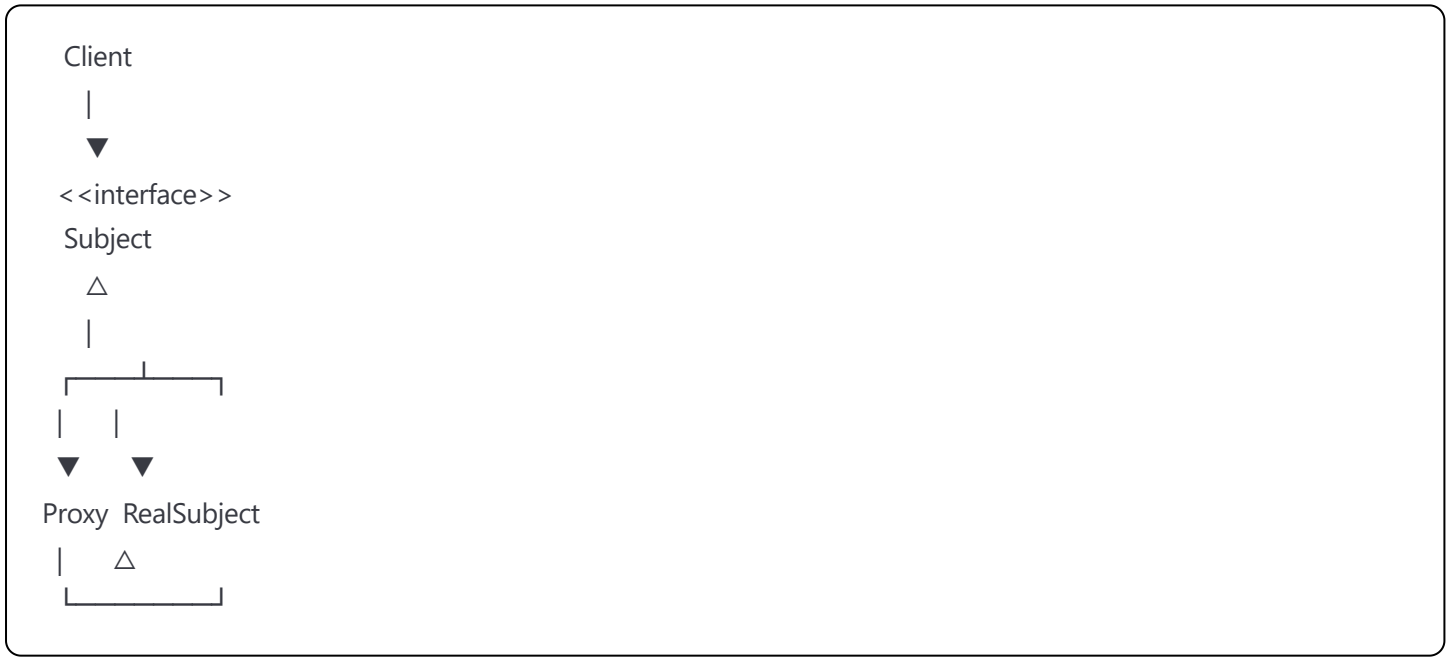
✅ **Use Proxy When:**

- You need lazy initialization of expensive objects
- Access control or security is required
- Working with remote objects or services
- You want to add logging, caching, or monitoring
- Need to control the lifecycle of objects

❌ **Avoid Proxy When:**

- Simple direct access is sufficient
- The overhead of proxy outweighs benefits
- Real-time performance is critical and caching isn't suitable

UML Structure



Best Practices

1. **Interface Consistency**: Proxy should implement the same interface as RealSubject
2. **Transparent Access**: Client shouldn't know it's working with a proxy
3. **Error Handling**: Implement proper exception handling in proxies
4. **Thread Safety**: Consider concurrent access in multi-threaded environments
5. **Resource Management**: Properly manage resources (connections, files, etc.)

This comprehensive guide covers all three main types of Proxy design patterns with practical Java examples that you can run and modify for your specific use cases.