# Design Pattern TT - 02

**Time: 45 mins** **Total Mark: 20**

---

1. You are developing a video streaming application that needs to stream videos from remote servers. To improve performance and security, you need to implement caching and access control mechanisms for the remote video resources. How would you design the application to provide an intermediary object that acts between the client and the remote video objects, handling caching, access control, and other optimizations? (10)
2. How does the Decorator pattern allow you to add responsibilities to objects dynamically? (5)
3. Explain the Chain of Responsibility pattern with a real-world example and draw the UML diagram for the Chain of Responsibility design pattern. (1+4)

For the first question regarding the design of a video streaming application that includes caching and access control mechanisms, we can use the **Proxy Design Pattern**. The Proxy pattern provides a surrogate or placeholder for another object to control access to it. Here's how you can implement this pattern in the context of your video streaming application.

## Proxy Design Pattern for Video Streaming Application

### Components and Roles

1. **Subject Interface**: Declares common operations that both the RealSubject (actual video server) and Proxy will implement.
2. **RealSubject**: The actual remote video server that contains the video resources.
3. **Proxy**: Acts as an intermediary between the client and the RealSubject. The Proxy handles caching and access control before forwarding requests to the RealSubject if necessary.
4. **Client**: Interacts with the Proxy to request video resources.

### Structure

1. **Subject Interface**: Defines the common interface for the RealSubject and Proxy.
2. **RealSubject**: Implements the Subject interface to provide the actual video streaming.
3. **Proxy**: Implements the Subject interface to add caching and access control functionalities.

### UML Diagram

```plaintext
+-----------------+     +----------------------+
|     Client      |     |      Subject         |
+-----------------+     +----------------------+
        |                         |
        v                         v
+-----------------+     +----------------------+
|     Proxy       |---->|     RealSubject      |
+-----------------+     +----------------------+
| - cache         |
| - accessControl |
+-----------------+
```

## Implementation

1. **Subject Interface**

java

```java
public interface VideoService {
    Video streamVideo(String videoId, User user);
}
```

2. **RealSubject**

java

```java
public class RealVideoService implements VideoService {
    @Override
    public Video streamVideo(String videoId, User user) {
        // Fetch video from remote server
        return fetchVideoFromServer(videoId);
    }
    private Video fetchVideoFromServer(String videoId) {
        // Logic to fetch video from remote server
    }
}
```

3. **Proxy**

java Copy co

```java
import java.util.HashMap;
import java.util.Map;

public class ProxyVideoService implements VideoService {
    private RealVideoService realVideoService = new RealVideoService();
    private Map<String, Video> cache = new HashMap<>();
    private AccessControlService accessControlService = new AccessControlService();

    @Override
    public Video streamVideo(String videoId, User user) {
        // Access Control Check
        if (!accessControlService.isUserAuthorized(user, videoId)) {
            throw new UnauthorizedAccessException("User is not authorized to view this vid
        }

        // Check Cache
        if (cache.containsKey(videoId)) {
            return cache.get(videoId);
        }

        // Fetch from real service
        Video video = realVideoService.streamVideo(videoId, user);

        // Cache the video
        cache.put(videoId, video);

        return video;
    }
}
```

4. **Access Control Service**

java

```java
public class AccessControlService {
    public boolean isUserAuthorized(User user, String videoId) {
        // Logic to check if the user has access to the video
    }
}
```

5. **Client**

java

```java
public class Client {
    public static void main(String[] args) {
        VideoService videoService = new ProxyVideoService();
        User user = new User("john_doe");
        try {
            Video video = videoService.streamVideo("video123", user);
            // Play video
        } catch (UnauthorizedAccessException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

## Explanation

- **Client** interacts with the **ProxyVideoService** to request video streaming.
- **ProxyVideoService**:
  - **Access Control**: Checks if the user is authorized to access the requested video.
  - **Caching**: Checks if the video is already in the cache. If yes, it returns the cached video.
  - If the video is not in the cache, it fetches the video from the **RealVideoService** and caches it for future requests.

## Decorator Pattern

The Decorator pattern allows you to dynamically add responsibilities to objects without altering their structure. This is achieved by creating a set of decorator classes that are used to wrap concrete components.

**How it Works:**

1. **Component Interface**: Defines the interface for objects that can have responsibilities added to them.
2. **Concrete Component**: The original object to which additional responsibilities can be added.
3. **Decorator**: Implements the component interface and contains a reference to a component object. The decorator class forwards requests to the component and may add additional behavior before or after forwarding.

**Example:**

In a video streaming app, you might have a `VideoStream` class (Concrete Component) and decorators such as `` `CachingDecorator` `` and `` `AccessControlDecorator` `` that add caching and access control functionalities.

## Chain of Responsibility Pattern

The Chain of Responsibility pattern allows multiple objects to handle a request without coupling the sender class to the receiver classes. This pattern creates a chain of receiver objects.

**Real-world Example:**

Consider a tech support system:

1. **Client**: Submits a support ticket.
2. **Level 1 Support**: First point of contact, handles basic issues.
3. **Level 2 Support**: Handles more complex issues if Level 1 cannot resolve.
4. **Level 3 Support**: Handles the most complex issues, such as those requiring specialized knowledge.

If Level 1 cannot handle the issue, it forwards the request to Level 2, and so on.

## UML Diagram:

plaintext

```
+-----------------+
|  Client         |
+--------+--------+
         |
         v
+--------+--------+       +---------+         +---------+
| Handler         +------>+ Handler  +-------->+ Handler  +------> Null
| (abstract class) |       | Level 2 |         | Level 3 |
+--------+--------+       +---------+         +---------+
         |
 +-------+-------+
 | Handler Level 1 |
 +-----------------+
```

**Explanation:**

1. **Handler (Abstract Class)**: Defines an interface for handling requests and an optional method for setting the next handler.
2. **Concrete Handlers (Level 1, Level 2, Level 3)**: Implement the handling logic and forward unhandled requests to the next handler in the chain.

By using these design patterns, the video streaming application can efficiently manage caching and access control, while maintaining flexibility and scalability.