**Proxy Design Pattern Concept [Layman’s Understanding]**

Imagine you have a door that you want to protect. Instead of locking the door itself, you place a guard outside the door. This guard is the "proxy." The guard decides who can access the door (the actual object) based on certain conditions. This is akin to the Proxy Pattern, where the proxy object controls access to the original object, ensuring that it is used correctly, efficiently, and securely.

The Proxy Design Pattern falls under the structural category. It is a method of creating a representative or "proxy" object that controls access to another object, which may be remote, expensive to create, or in need of securing.

## **Implementation Example in Java**

**Real Object Class**

* First, define the RealObject class that the proxy will represent.

public class RealObject {

public void performAction() {

System.out.println("Performing action in the real object");

}

}

**Proxy Class**

* Next, create the ProxyObject class. This class will control access to the RealObject.

public class ProxyObject extends RealObject {

private RealObject realObject;

@Override

public void performAction() {

if (realObject == null) {

realObject = new RealObject();

}

System.out.println("ProxyObject delegating work to the RealObject");

realObject.performAction();

}

}

**Using the Proxy**

Finally, demonstrate how a client would use the ProxyObject.

public class Client {

public static void main(String[] args) {

RealObject proxy = new ProxyObject();

proxy.performAction();

}

}

## **When to use Proxy Pattern**

* **Lazy Initialization:** When an object is resource-intensive and you want to delay its creation until absolutely necessary.
* **Access Control:** To control access to the object, for example, checking user permissions.
* **Logging:** To keep logs when an object is accessed.
* **Caching:** Store results of expensive operations and return cached results for subsequent requests.
* **Reference Counting:** Keep track of how many objects are referencing an object, which is useful in memory management.

## **When not to use Proxy Pattern**

* **Simple Object Access:** If the object access is straightforward and does not require any of the complexities mentioned above.
* **Performance Critical Situations:** If introducing a proxy might add unwanted overhead in scenarios where performance is critical.

## **Pitfalls**

* **Complexity:** Overuse can lead to an overly complex system.
* **Maintenance Difficulty:** Can make the codebase harder to understand and maintain.
* **Performance Overhead:** Introducing an additional layer can impact performance, especially in high-load scenarios.