Proxy Design Pattern

Notes By Bhavuk Jain

Definition

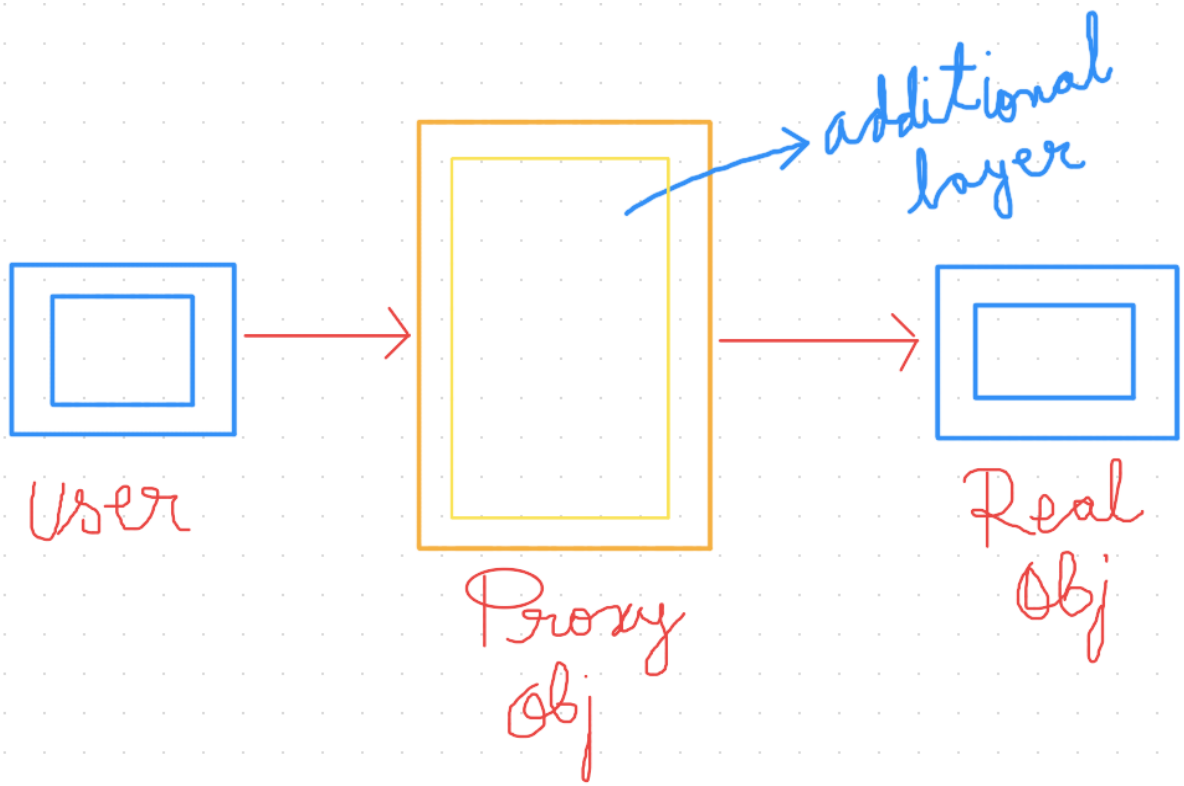
The **Proxy Design Pattern** allows you to create a ***stand-in*** object that controls access to another object. It acts as a **middleman** between a client and the actual object, providing additional functionality or restrictions as needed. The proxy handles requests from the client and can perform tasks like **checking permissions, caching data, or optimizing performance** before passing the request to the real object. Essentially, a proxy helps manage and enhance the interaction with the real object ***without the client directly interacting*** with it.

In-short, the client feels as if he is interacting with the real object but in reality, he is not. **PROXY**, in literal meaning, is kind of an **illusion** where the client is made to think as if he is directly interacting with the real object but actually, there is a **layer** (or there can be ***many layers***) between the client and the real object.

What & Why?

You can imagine it in a way, where a client/user wants to access certain methods of an object to perform some operations. **Instead of directly accessing the real object and use its methods, he communicates with a PROXY of that object**.

Both the ***Proxy*** and ***Real*** objects are implementations of the **same interface** but the ***Proxy introduces additional logic before executing the methods of Real object.*** The **client** calls the methods of the Proxy objects & it is the Proxy object which then calls the methods of Real object after adding some pre-processing or post-processing logic.



Use Cases

- **Control access**: Proxies allow you to control access to objects by adding an additional layer of authorization or authentication. This can help enforce security measures and ensure that only authorized clients can interact with sensitive or restricted resources.

- **Performance optimization**: Proxies can implement techniques such as lazy loading, caching, or request batching to optimize the performance of expensive operations. By deferring the creation or execution of objects until they are actually needed, proxies can improve overall system efficiency.

- **Simplify complex systems**: Proxies can provide a simplified interface to a complex or resource-intensive object. Clients interact with the proxy object, which handles the complexity internally, shielding the client from the underlying implementation details.

- **Remote communication**: Proxies are often used in distributed systems or remote communication scenarios. A proxy can act as a representative or surrogate for a remote object, allowing the client to interact with the remote object as if it were a local object. The proxy can handle tasks like network communication, serialization, or error handling transparently.

- **Logging and auditing**: Proxies can be used to log or audit interactions with objects, providing a way to record important information about method calls, parameters, and results. This can be valuable for debugging, monitoring, or compliance purposes.

Code Snippets

(Go through the code comments as well for better perspective)

1.) **Document Interface** which has two abstract methods namely, **view()** & **edit().**

```
package proxydesignpattern;

public interface Document {
   /**
    * Method to view document
    */
   void view();

   /**
    * Method to edit document
    */
   void edit();
}
```

2.) **RealDocument** class implements the **Document Interface** and provides implementation to its abstract methods. **(RealDocument is-a Document)**

```
package proxydesignpattern;

public class RealDocument implements Document{
   /**
    * Implementation of edit()
    */
   public void edit(){
      // Operations to be done when editing a document.
      System.out.println("Document edited successfully!");
   }

   /**
    * Implementation of view()
    */
   public void view(){
      // Operations to be done when viewing a document.
      System.out.println("Document viewed successfully!\n");
   }
}
```

3.) **ProxyDocument** class is also an implementation of **Document Interface (is-a),** which also includes a **RealDocument** type of object with it **(has-a)**. Along with it, it has a **parameterized constructor** which initializes the value of **User.**
Therefore, **ProxyDocument** provides the implementation for the abstract methods of **Document Interface** along with some *additionalchecks*.

In this case, the additional logic is checking if a type of user has **permissions to view/edit** the document or not. Based on these permissions, it then **calls the methods of RealDocument object**.

```
package proxydesignpattern;

public class ProxyDocument implements Document{
   // RealDocument Instance
   private RealDocument realDocument = new RealDocument();

   // Type of User
   private String user;

   // Constructor to initialize the type of user -> "admin" | "user" | "unknown"
   public ProxyDocument(String user) { this.user = user; }

   // Implementing view() of Document Interface
   public void view(){
   if(hasPermission( operation: "view")){
      realDocument.view();    // Calling the RealDocument's view() after additional checks!
   }
   else{
      System.out.println("No Permission To View.");    // Operations in case of no permission
   }
   }
```

```
    // Implementing edit() of Document Interface
    public void edit(){
    if(hasPermission( operation: "edit")){
        realDocument.edit();     // Calling the RealDocument's edit() after additional checks!
    }
    else{
        System.out.println("No Permission To Edit.");    // Operations in case of no permission
    }
    }

    /*
    * Helper method to check if a user has a specified permission or not!
    * "admin" - has both view() and edit() permissions
    * "user" - has only view() permission
    * "unknown" - has no permissions at all.
     */
    private boolean hasPermission(String operation){
        if(user.equals("admin")){
            return true;
        }
        else if(user.equals("user")){
            return operation.equals("view");
        }
        else {
            return false;
        }
    }
}
```

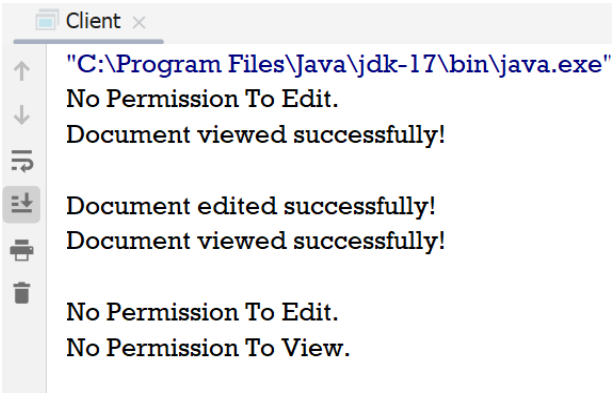
Execution

We create three objects of **ProxyDocument**. All three objects are created by passing a different kind of User i.e. ***Normal User***, ***Admin*** **and any other** ***Unknown*** **user.** Then, the ***view()***and ***edit()***methods are called using those objects.

```java
package proxydesignpattern;

public class Client {
    public static void main(String[] args) {
        /*
         * Normal User - Only View Permission
         */
        ProxyDocument proxyDocumentUser = new ProxyDocument( user: "user");
        proxyDocumentUser.edit();
        proxyDocumentUser.view();

        /**
         * Admin - User with special permissions
         */
        ProxyDocument proxyDocumentAdmin = new ProxyDocument( user: "admin");
        proxyDocumentAdmin.edit();
        proxyDocumentAdmin.view();

        /**
         * Unknown User - No Permissions
         */
        ProxyDocument proxyDocumentUnknownUser = new ProxyDocument( user: "unknown");
        proxyDocumentUnknownUser.edit();
        proxyDocumentUnknownUser.view();
    }
}
```

Output

For Normal User : There is no Edit Permission but only View Permission
For Admin : There are both the permissions to Edit & View The Document
For Unknown : There is no permission for View/Edit



Thanks For Reading!
❤️ Bhavuk Jain