Proxy

Resources

- Video → 32. All Structural Design Patterns | Decorator, Proxy, Composite, Adapter, Bridge, Facade, FlyWeight
- Video → 13. Proxy Design Pattern Explanation (Hindi) | LLD | System Design Interview Question | Java

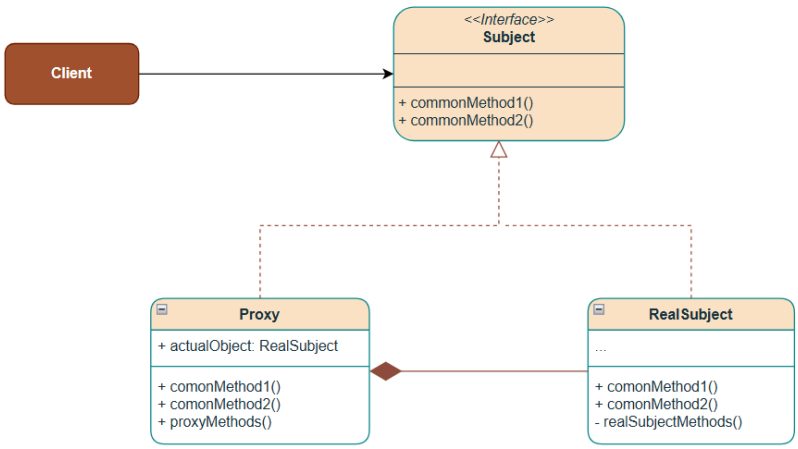
Definition

> ***The Proxy Design Pattern provides a representative or placeholder for another object(real comlpex object) to control access to it.***

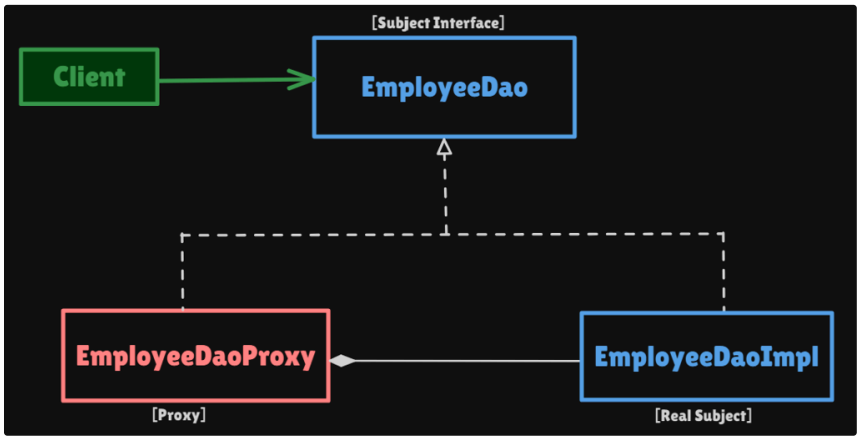
Use cases of the Proxy Pattern

- **Access Control**: Restricts access to sensitive operations based on user permissions.
- **Performance Optimization**: Delays resource-intensive operations like expensive object creation until necessary (lazy loading).
- **Pre/post processing:** Performs additional operations before and after forwarding the request to the actual object, like adding cross-cutting concerns like audit traces (logging & monitoring), without modifying core business logic.
- **Caching**: Can cache results to improve performance.

Class Diagram



Structure of Proxy Pattern



1. **Subject Interface (`EmployeeDao`)**
   - Defines the common interface for both `EmployeeDaoImpl` and `EmployeeDaoProxy`.
   - Ensures the proxy can be used anywhere the real object is expected
2. **Real Subject (`EmployeeDaoImpl`)**
   - Contains the actual business logic and data. Handles core employee operations.
   - Performs expensive operations like database loading.
3. **Proxy Object (`EmployeeDaoProxy`)**
   - Maintains a reference to the Real Subject(`EmployeeDaoImpl`).
   - Controls access through permission checks.
   - Can be modified further to implement lazy loading, logging and etc.
4. **Client (`EmployeeManagement`)**
   - Client is performing operations on the subject without being aware of whether it's a Real/Proxy object

Implementation

```java
// Subject interface - common interface for RealSubject and Proxy
public interface EmployeeDao {

    void getEmployeeInfo();

    void createEmployee();

    void deleteEmployee();

    void updateSalary(double newSalary);

    void displayEmployeeDetails();
}
```

```java
// RealSubject - the actual employee object that does the real work
public class EmployeeDaoImpl implements EmployeeDao {

    int id;
    String name;
    String department;
    Double salary;
    UserRole role;

    public EmployeeDaoImpl(int id, String name, String department,
Double salary, UserRole role) {
```

```
        this.id = id;
        this.name = name;
        this.department = department;
        this.salary = salary;
        this.role = role;
    }

    @Override
    public void displayEmployeeDetails() {
        System.out.println("Employee Details: " + "Id: " + id + ", Name: " + name + ", Department: " + department + ", Salary: Rs" + salary + ", Role: " + role);
    }

    @Override
    public void updateSalary(double newSalary) {
        this.salary = newSalary;
        System.out.println("Salary updated to: Rs. " + newSalary);
    }

    @Override
    public void deleteEmployee() {
        System.out.println("Employee " + name + " has been deleted from the system.");
    }

    @Override
    public void createEmployee() {
        System.out.println("Employee " + name + " has been created in the system.");
    }

    @Override
    public void getEmployeeInfo() {
        System.out.println("Employee Info:" + " Name: " + name + ", Department: " + department);
    }

}
```

```
// Proxy class - controls access to RealEmployee
public class EmployeeDaoProxy implements EmployeeDao {

    // Reference to the RealSubject
    EmployeeDao realEmployee;

    public EmployeeDaoProxy(int id, String name, String department, Double salary, UserRole role) {
        realEmployee = new EmployeeDao(id, name, department, salary, role);
    }

    // Access control for viewing complete employee details
    public void displayEmployeeDetails() {
        if (hasAccessForRole(AccessType.VIEW_ALL_DETAILS, this.realEmployee.role)) {
            this.realEmployee.displayEmployeeDetails();
        } else {
            throw new RuntimeException("Access Denied");
        }
    }

    // Access control for salary updates
    public void updateSalary(double newSalary) {
        if (hasAccessForRole(AccessType.UPDATE, this.realEmployee.role)) {
            this.realEmployee.updateSalary(newSalary);
        } else {
            throw new RuntimeException("Access Denied");
        }
    }
```

```
28
29      // Access control for employee deletion
30      public void deleteEmployee() {
31          if (hasAccessForRole(AccessType.DELETE,
    this.realEmployee.role)) {
32              this.realEmployee.deleteEmployee();
33          } else {
34              throw new RuntimeException("Access Denied");
35          }
36      }
37
38      // Access control for employee creation - resource intensive
39      public void createEmployee() {
40          if (hasAccessForRole(AccessType.CREATE,
    this.realEmployee.role)) {
41              this.realEmployee.createEmployee();
42          } else {
43              throw new RuntimeException("Access Denied");
44          }
45      }
46
47      // All employees can view basic info
48      public void getEmployeeInfo() {
49          if (hasAccessForRole(AccessType.VIEW_BASIC_INFO,
    this.realEmployee.role)) {
50              this.realEmployee.getEmployeeInfo();
51          }
52      }
53
54      // Helper method to simulate different access capabilities for
    different roles
55      private boolean hasAccessForRole(AccessType acesssType, UserRole
    role) {
56          return switch (acesssType) {
57              case AccessType.CREATE, AccessType.DELETE -> role ==
    UserRole.ADMIN;
58              case AccessType.UPDATE -> role == UserRole.ADMIN || role
    == UserRole.HR;
59              case AccessType.VIEW_ALL_DETAILS -> role ==
    UserRole.MANAGER || role == UserRole.HR || role == UserRole.ADMIN;
60              case AccessType.VIEW_BASIC_INFO -> role ==
    UserRole.EMPLOYEE || role == UserRole.MANAGER || role == UserRole.HR
    || role == UserRole.ADMIN;
61              default -> false;
62          };
63      }
64  }
```

```
1   // Enum for user roles
2   enum UserRole {
3       EMPLOYEE, MANAGER, ADMIN, HR
4   }
5
6   // Enum for access types
7   enum AccessType {
8       CREATE, DELETE, UPDATE, VIEW_ALL_DETAILS, VIEW_BASIC_INFO
9   }
10
11  // Client
12  public class EmployeeManagement {
13      public static void main(String[] args) {
14          System.out.println("===== Proxy Design Pattern =====");
15
16          // Demo for ADMIN user
17          EmployeeDao admin = new EmployeeDaoProxy(1, "John", "HR",
    30000.00, UserRole.ADMIN);
18          admin.getEmployeeInfo();
19          admin.displayEmployeeDetails();
20          admin.createEmployee();
21          admin.updateSalary(40000.00);
22          admin.deleteEmployee();
```

```
23
24          // Demo for HR user
25          EmployeeDao hr = new EmployeeDaoProxy(2, "Jane", "HR",
    25000.00, UserRole.HR);
26          hr.getEmployeeInfo();
27          hr.displayEmployeeDetails();
28          // hr.createEmployee(); // Access Denied
29          hr.updateSalary(30000.00);
30          // hr.deleteEmployee(); // Access Denied
31
32          // Demo for MANAGER user
33          EmployeeDao manager = new EmployeeDaoProxy(3, "Bob", "IT",
    20000.00, UserRole.MANAGER);
34          manager.getEmployeeInfo();
35          manager.displayEmployeeDetails();
36          // manager.createEmployee(); // Access Denied
37          // manager.updateSalary(30000.00); // Access Denied
38          // manager.deleteEmployee(); // Access Denied
39
40          // Demo for EMPLOYEE user
41          EmployeeDao employee = new EmployeeDaoProxy(3, "Bob", "IT",
    20000.00, UserRole.EMPLOYEE);
42          employee.getEmployeeInfo();
43          // employee.displayEmployeeDetails(); // Access Denied
44          // employee.createEmployee(); // Access Denied
45          // employee.updateSalary(30000.00); // Access Denied
46          // employee.deleteEmployee(); // Access Denied
47      }
48  }
```

The Proxy pattern is particularly useful in enterprise applications where security, performance, and monitoring are critical concerns.