Pandit Deendayal Energy University, Gandhinagar

School of Technology

Department of Computer Science & Engineering

Design Pattern Lab(20CP210P)

Name: **Padshala Smit Jagdishbhai**

Enrolment No: **21BCP187**

Semester: **IV Division: 3 (G6)**

Branch: **Computer Science Engineering**

- **Structural Patterns: -**

1. **Composite:** Composes objects into tree structures to represent part-whole hierarchies, and allows clients to treat individual objects and compositions of objects uniformly.
2. **Adapter:** Converts the interface of a class into another interface that clients expect, allowing classes with incompatible interfaces to work together.
3. **Facade:** Provides a unified interface to a set of interfaces in a subsystem, simplifying communication between the subsystem and its clients.
4. **Proxy:** Provides a surrogate or placeholder for another object to control access to it.
5. **Flyweight**: Shares large numbers of fine-grained objects efficiently to reduce memory usage.
6. **Decorator:** Dynamically adds responsibilities to objects by wrapping them in one or more decorator objects.
7. **Bridge**: Decouples an abstraction from its implementation, allowing the two to vary independently.

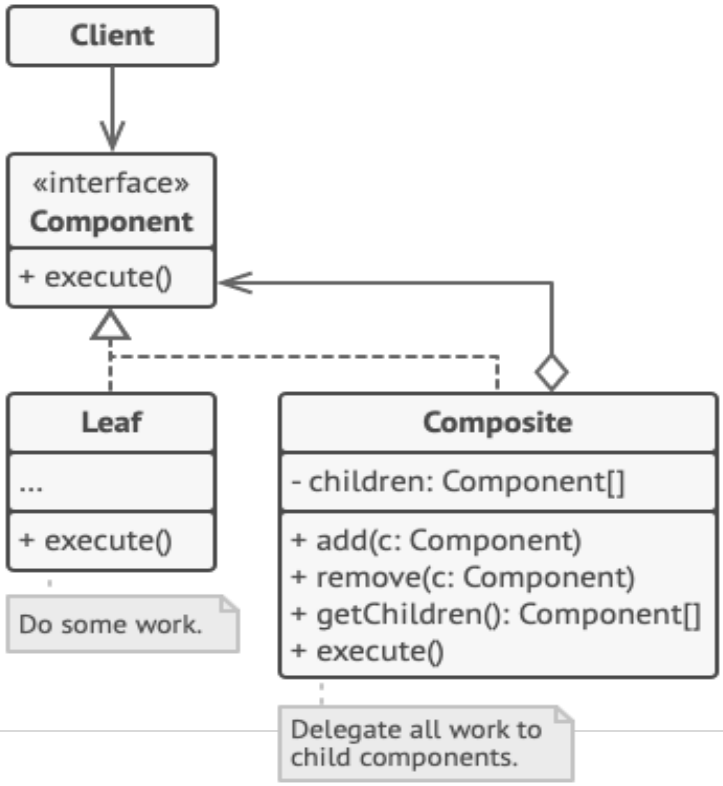
1. Composite Structural Design Pattern: -

The Composite Structural Design Pattern is a design pattern that is used to represent a group of objects in a hierarchical tree structure. It is a structural pattern that defines a class hierarchy that can represent objects as individual entities or as part of a larger composite structure.

In this pattern, a composite object is created by aggregating objects of similar or different types. This composite object can then be treated as an individual object and can be further used to build more complex structures.

Advantages: -

1. Provides a uniform way of representing object hierarchies.
2. Simplifies the addition of new types of components to the hierarchy.
3. Provides a clear distinction between simple and complex components.
4. Allows for the easy creation of recursive hierarchies.



Disadvantages: -

1. Can be more complex than other design patterns.
2. May result in slower execution times due to the recursive nature of the pattern.
3. Can be more difficult to implement and maintain than other design patterns.
4. May lead to an overuse of the pattern, resulting in unnecessary complexity.

Code: -

```
import java.util.ArrayList;
import java.util.List;

// Abstract class that defines the basic structure of a component
abstract class Component{
    // Property to store the name of the component
    protected String name;

    // Constructor that takes in the name of the component
    public Component(String n){
        name = n;
    }

    // Abstract method that needs to be implemented by the concrete classes
    public abstract void describeComponent();
    public abstract int getPrice();
}

// Concrete class for a leaf component
class leaf extends Component{
    int price;
    // Constructor that takes in the name of the leaf component
    public leaf(String n, int price) {
        super(n);
        this.price=price;
    }
```

```
    // Implementation of the describeComponent() method
    @Override
    public void describeComponent() {
        // Prints the name of the leaf component
        System.out.println("Leaf: "+name);
    }

    @Override
    public int getPrice() {
        System.out.println("Leaf: "+name+"\tPrice: "+price);
        return price;
    }
}

// Concrete class for a composite component
class Composite extends Component{
    // List to store the child components (either leaf or composite)
    List<Component> ls = new ArrayList<Component>();

    // Constructor that takes in the name of the composite component
    public Composite(String n) {
        super(n);
    }

    // Method to add a child component to the composite component
    public void addComponent(Component c){
        ls.add(c);
    }

    // Method to remove a child component from the composite component
    public void removeComponent(Component c){
        ls.remove(c);
    }

    // Method to get the list of child components
    public List componenList(){return ls;}

    // Implementation of the describeComponent() method

    @Override
    public void describeComponent() {
        // Prints the name of the composite component
```

```
        System.out.println("Component: "+name);
        // Loops through the list of child components and describes each component
        for(Component c: ls){
            c.describeComponent();
        }
    }

    @Override
    public int getPrice() {
        int p=0;
        for(Component c:ls){
            p += c.getPrice();
        }
        return p;
    }
}

// Client class to use the composite pattern
public class composite_pattern {

    public static void main(String[] args) {
        Component HDD = new leaf("HDD",5000);
        Component RAM = new leaf("RAM",3000);
        Component CPU = new leaf("CPU",49000);
        Component Mouse = new leaf("Mouse",999);
        Component keyboard = new leaf("Keyboard",1499);

        Composite Computer = new Composite("Computer");
        Composite Cabinat = new Composite("Cabinat");
        Composite Peripheral = new Composite("Paripheral");
        Composite Motherboard = new Composite("Motherboard");

        Computer.addComponent(Cabinat);
        Computer.addComponent(Peripheral);

        Cabinat.addComponent(HDD);
        Cabinat.addComponent(Motherboard);

        Peripheral.addComponent(Mouse);
        Peripheral.addComponent(keyboard);

        Motherboard.addComponent(CPU);
```

```
        Motherboard.addComponent(RAM);

        Computer.describeComponent();
        // Computer.getPrice();


    }
}
```

Output: -

```
Component: Computer
Component: Cabinat
Leaf: HDD
Component: Motherboard
Leaf: CPU
Leaf: RAM
Component: Paripheral
Leaf: Mouse
Leaf: Keyboard
```

2. Adapter Structural Design Pattern: -

The Adapter Structural Design Pattern is a design pattern that is used to enable the interaction between two incompatible objects. It is a structural pattern that adapts the interface of one class to another interface that is expected by the client.

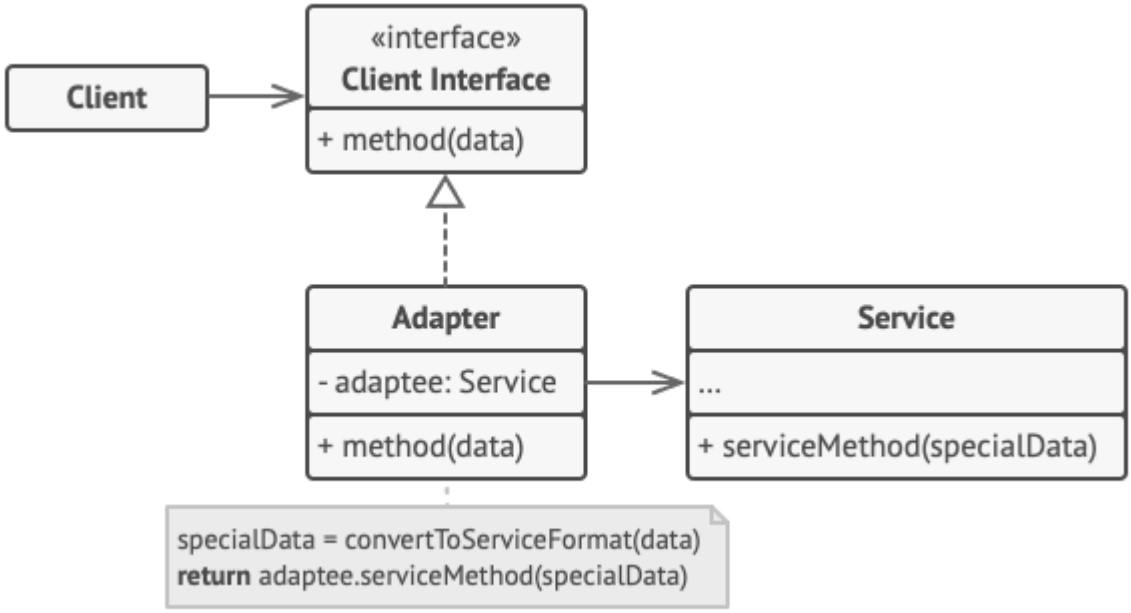
In this pattern, an adapter acts as a mediator between the two incompatible interfaces and translates the requests from one interface to the other. This allows the client to use the incompatible interface as if it were compatible.

Advantages: -

1. Enables the interaction between incompatible objects.
2. Allows for the reuse of existing classes without modifying their source code.
3. Provides a way to convert one interface into another interface.
4. Increases the flexibility of the system by allowing objects to work together that otherwise would not be able to.

Disadvantages: -

1. Can add complexity to the system by introducing additional layers of abstraction.
2. May result in decreased performance due to the additional processing required to translate requests between interfaces.
3. Requires a thorough understanding of the existing classes and interfaces in the system.
4. Can make the code more difficult to read and maintain due to the introduction of adapters.



Code: -

```
//Client interface
interface MediaPlayer{
   public void play(String type, String fileName);
}

class AudioPlayer implements MediaPlayer{
   @Override
   public void play(String type, String fileName) {
      // if(type.equalsIgnoreCase("vlc")){
      //     System.out.println("Playing "+fileName+"of type "+type);
      // }
      MediaPlayer md = new MediaAdapter();
      md.play(type, fileName);
   }
}

//Service Library->Adaptee
class AdvancedMediaPlayer{

   void playVlc(String fileName){System.out.println("Playing "+fileName+" of type
VLC");};
   void playMp3(String fileName){System.out.println("Playing "+fileName+" of type
mp3");};
```

```
    void playMp4(String fileName){System.out.println("Playing "+fileName+" of type
mp4");};
}

//Adapter
class MediaAdapter implements MediaPlayer{
    //reference of Adaptee
    AdvancedMediaPlayer adv = new AdvancedMediaPlayer();

    @Override
    public void play(String type, String fileName) {
        if(type.equalsIgnoreCase("vlc"))
            adv.playVlc(fileName);
        else if(type.equalsIgnoreCase("mp3"))
            adv.playMp3(fileName);
        else if(type.equalsIgnoreCase("mp4"))
            adv.playMp4(fileName);
    }
}

//Client
public class LAb11AdapterPatternAudio {
        public static void main(String[] args) {
            MediaPlayer m1 = new AudioPlayer();
            m1.play("vlc","Hanuman_Chalisa.vlc");
            m1.play("mp3","Hare_Rama_Hare_Krishna.mp3");
            m1.play("mp4","Mahabharat_S1E1.mp4");
        }
}
```

Output: -

```
Playing Hanuman_Chalisa.vlc of type VLC
Playing Hare_Rama_Hare_Krishna.mp3 of type mp3
Playing Mahabharat_S1E1.mp4 of type mp4
```

3. Facade Structural Design Pattern: -

The Facade Structural Design Pattern is a design pattern that provides a simplified interface to a complex subsystem. It is a structural pattern that provides a unified interface to a set of interfaces in a subsystem.

In this pattern, a facade class is created that provides a simplified interface to a complex subsystem. The facade class acts as an intermediary between the client and the subsystem, shielding the client from the complexity of the subsystem.

The Facade Structural Design Pattern is a useful tool for simplifying the interface to a complex subsystem. It provides a single point of entry to the subsystem, encapsulates its complexity, and reduces coupling between the client and the subsystem. However, it can result in a performance hit and may not be suitable for systems that require a high level of customization or fine-grained control.

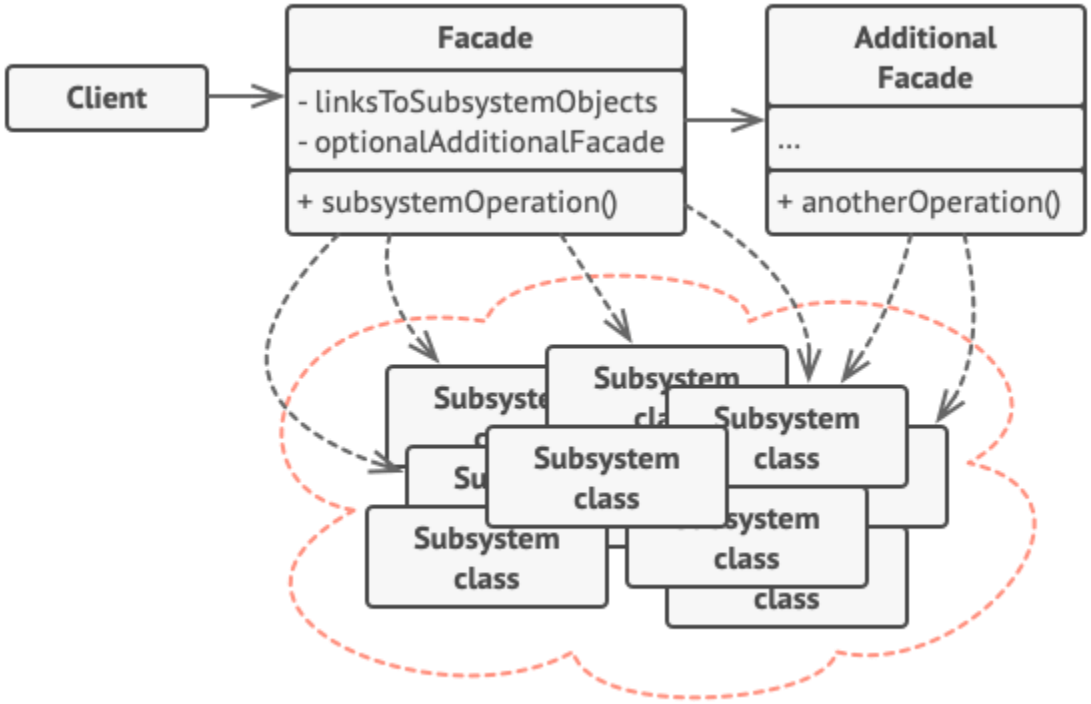
Advantages: -

1. Provides a simplified interface to a complex subsystem, making it easier to use.
2. Increases the flexibility of the system by providing a single point of entry to the subsystem.
3. Encapsulates the complexity of the subsystem, making it easier to modify or replace individual components.
4. Reduces coupling between the client and the subsystem, making it easier to maintain and evolve the system.

Disadvantages: -

1. Can result in a performance hit due to the additional layer of abstraction.
2. May lead to an increase in the amount of code in the system.
3. May not be suitable for systems that require a high level of customization or fine-grained control.

4. Can make it more difficult to understand the interactions between the client and the subsystem.



Code: -

```
// MobileShop interface
interface MobileShop {
    public void modelNo();
    public void price();
}

// Iphone class implementing MobileShop interface
class Iphone implements MobileShop {

    @Override
    public void modelNo() {
        System.out.println("Iphone Model Name: Iphone 14 pro");
    }

    @Override
    public void price() {
```

```
        System.out.println("Iphone 14 pro Price: 1,49,900 Rs.");
    }
}

// Samsung class implementing MobileShop interface
class Samsung implements MobileShop {
    @Override
    public void modelNo() {
        System.out.println("Samsung Model Name: Galaxy S23 Ultra");
    }
    @Override
    public void price() {
        System.out.println("Samsung Price: 1,34,999 Rs.");
    }
}

// Oneplus class implementing MobileShop interface
class Oneplus implements MobileShop {
    @Override
    public void modelNo() {
        System.out.println("Oneplus Model Name: OnePlus 10 Ultra 5G");
    }
    @Override
    public void price() {
        System.out.println("Oneplus Price: 69,999 Rs.");
    }
}

// Shopkeeper class as the Facade
class Shopkeeper {
    private MobileShop iphone;
    private MobileShop samsung;
    private MobileShop Oneplus;

    public Shopkeeper() {
        iphone = new Iphone();
        samsung = new Samsung();
        Oneplus = new Oneplus();
    }

    public void iphoneSale() {
        iphone.modelNo();
```

```
        iphone.price();
    }

    public void samsungSale() {
        samsung.modelNo();
        samsung.price();
    }

    public void OneplusSale() {
        Oneplus.modelNo();
        Oneplus.price();
    }
}

// Facade pattern client
public class FacadepatternClient {
    public static void main(String[] args) {
        Shopkeeper shopkeeper = new Shopkeeper();

        System.out.println("Welcome to our mobile shop!");
        System.out.println("Here are our available mobile phones:");
        System.out.println("----------------------------------");
        shopkeeper.iphoneSale();
        shopkeeper.samsungSale();
        shopkeeper.OneplusSale();
        System.out.println("----------------------------------");
        System.out.println("Thank you for visiting our shop!");
    }
}
```

Output: -

```
Welcome to our mobile shop!
Here are our available mobile phones:
----------------------------------
Iphone Model Name: Iphone 14 pro
Iphone 14 pro Price: 1,49,900 Rs.
Samsung Model Name: Galaxy S23 Ultra
Samsung Price: 1,34,999 Rs.
Oneplus Model Name: OnePlus 10 Ultra 5G
Oneplus Price: 69,999 Rs.
----------------------------------
Thank you for visiting our shop!
```

4. Proxy Structural Design Pattern: -

The Proxy Structural Design Pattern is a design pattern that provides a surrogate or placeholder for another object to control its access. It is a structural pattern that creates a representative object that acts as a substitute for another object.

In this pattern, a proxy object is created that is a stand-in for the real object. The proxy object provides the same interface as the real object, but it controls access to the real object. The proxy can perform additional tasks, such as caching or logging, before or after the real object's methods are called.

The Proxy Structural Design Pattern is a useful tool for controlling access to a real object and providing additional functionality. It can improve the security and performance of the system, as well as provide a way to manage the lifecycle of the real object. However, it can add complexity to the system and may not be suitable for systems that require direct access to the real object.

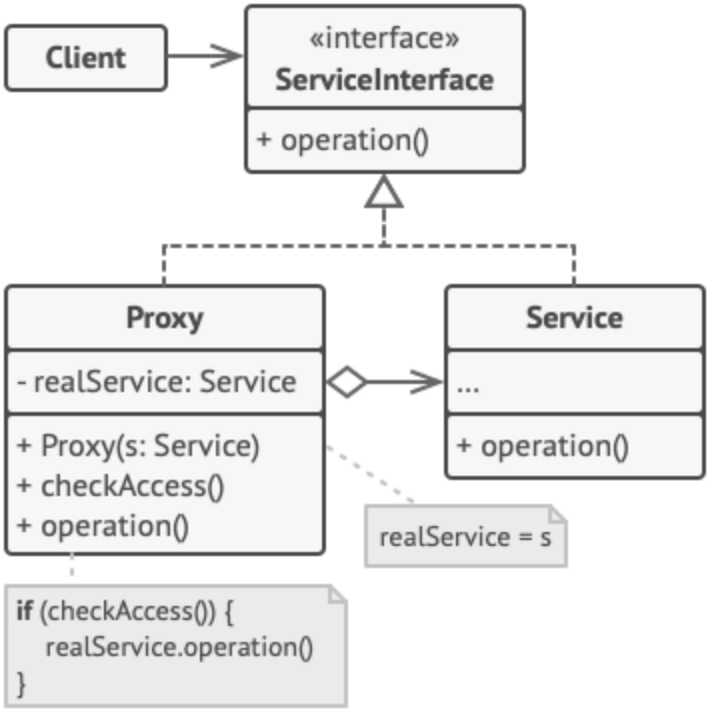
Advantages: -

1. Controls access to the real object, providing additional security and safety.
2. Reduces the overhead of accessing the real object by performing tasks such as caching or logging.
3. Provides a way to manage the lifecycle of the real object, such as by delaying its creation until it is actually needed.
4. Can improve the performance of the system by providing a lightweight alternative to the real object.

Disadvantages: -

1. Can add complexity to the system by introducing additional layers of abstraction.
2. May result in decreased performance due to the additional processing required by the proxy object.
3. Requires additional code to be written to manage the proxy object.

4. May not be suitable for systems that require direct access to the real object.



Code: -

```
import java.io.*;
import java.util.*;
import java.text.SimpleDateFormat;

// CustomerServiceDatabase interface to be implemented by CustomerService and
CustomerServiceProxy
interface CustomerServiceDatabase{
   public Customer getCustomerDetails(String custId);
}

// Customer class to store customer details
class Customer {
   String custId, custName, custEmail, custCity;
   public Customer(String custId, String custName, String custEmail, String custCity) {
      this.custId = custId;
      this.custName = custName;
      this.custEmail = custEmail;
      this.custCity = custCity;
```

```
    }

    public String toString() {
        return "Customer Details:\n[custId: " + custId + ",   custName: " + custName +
",   custEmail: " + custEmail + ",   custCity: "+ custCity + "]";
    }
}

// CustomerService class to read customer details from a file
class CustomerService implements CustomerServiceDatabase {
    public Customer getCustomerDetails(String custId){
        try {
            //Read the file customerinfo.txt
            BufferedReader reader = new BufferedReader(new
FileReader("customerinfo.txt"));
            String line = null;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(" ");
                if (parts[0].equals(custId)) {
                    //Search for the custId if exist return Customer Details.
                    Customer c = new Customer(parts[0], parts[1], parts[2], parts[3]);
                    return c;
                }
            }
            //else Record not found.
             // If custId doesn't exist, print a message and return null
            System.out.println("Record not found for custId: " + custId);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
}


// CustomerServiceProxy class to intercept requests and log user access and verify user
role
class CustomerServiceProxy implements CustomerServiceDatabase {
    // Instantiate a CustomerService object to forward the request
    CustomerService cs = new CustomerService();
    // Create a SimpleDateFormat object to format the date
    SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
```

```
    // Create a Date object to store the current date and time
    Date date = new Date();
    // Store the user ID
    private String userId;
    // Constructor to initialize the userId variable
    CustomerServiceProxy(String userId){
        this.userId=userId;
    }

    public Customer getCustomerDetails(String custId){
        try {
            //Write user information to the logCustomerDatabase.txt(user who is accessing
the information)
            BufferedWriter writer = new BufferedWriter(new
FileWriter("logcustomerdatabase.txt", true));
            writer.append("User " + getCurrentUserId() + " accessed customer details at " +
formatter.format(date) + "\n");
            writer.close();

            //If user role is "Admin" then forward the request
            if (getCurrentUserRole().equals("Admin")) {
                return cs.getCustomerDetails(custId);
            }
            //Else message "Access is denied"
            System.out.println("Access is denied for user: " + getCurrentUserId());
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }

    // Method to return the current user ID
    private String getCurrentUserId() {
        //Implementation for getting current user ID
        return userId;
    }

    // This method reads the userinfo.txt file and returns the role of the user with the
matching user ID
    private String getCurrentUserRole() {
        try {
            // Read the file userinfo.txt
```

```
            BufferedReader reader = new BufferedReader(new FileReader("userinfo.txt"));
            String line = null;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(", ");
                if (parts[0].equals(getCurrentUserId())) {
                    // Matched user found, return their role
                    return parts[1];
                }
            }
            // User not found in the file
            System.out.println("User not found!!");
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
}

// The main method of the ClientProxyPattern2 class
public class ClientProxyPattern2 {

    // This method reads the userinfo.txt file and checks if the user ID and password are
correct
    public static boolean checker(String userId, String password) {
        String x=null, y=null;
        try {
            // Read the file userinfo.txt
            BufferedReader reader = new BufferedReader(new FileReader("userinfo.txt"));
            String line = null;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(", ");
                if (parts[0].equals(userId)) {
                    // Matched user found, return their role
                    x=parts[0];
                    y=parts[2];
                }
            }
            // UserId not found in the file
            if(x==null){
                System.out.println("user not found");
            }
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
        // Check if the user ID and password are correct
        if(userId.equals(x) && password.equals(y)){
            return true;
        }
        else{
            return false;
        }
    }

  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter your user ID: ");
    String userId = scanner.nextLine();
    System.out.print("Enter your password: ");
    String password = scanner.nextLine();
    // checker the client using the user ID and password
    if (checker(userId, password)) {
        // Access customer details if checkerd
        CustomerServiceDatabase cs = new CustomerServiceProxy(userId);
        System.out.print("Enter Customer Id: ");
        String custid = scanner.nextLine();
        Customer c1 = cs.getCustomerDetails(custid);
        // Print customer details
        if (c1 != null) {
            System.out.println(c1);
        }
    } else {
        System.out.println("Incorrect user ID or password.");
    }
  }
}
```

Output: -

```
Enter your user ID: user1
Enter your password: abcd1234
Enter Customer Id: cust1
Customer Details:
[custId: cust1,   custName: Smit,   custEmail: smit187@gmail.com,   custCity: A'bad]
```

Custmerinfo.txt

lab 13 Proxy Pattern > customerinfo.txt

```
1  cust1 Smit smit187@gmail.com A'bad
2  cust2 Nisarg nisarg191@gmail.com A'bad
3  cust3 Tej tej190@gmail.com Rajkot
4  cust4 Shlok shlok201@gmail.com A'bad
5  cust5 Bhavya bhavya203@gmail.com Rajkot
6  cust6 Denil denil205@gmail.com bhuj
```

Userinfo.txt

lab 13 Proxy Pattern > userinfo.txt

```
1  user1, Admin, abcd1234
2  user2, Student, password
3  user3, Admin, 12345678
4  user4, Student, 00000000
```

Logcustomerdatabase.txt

lab 13 Proxy Pattern > logcustomerdatabase.txt

```
1  User user1 accessed customer details at 22/02/2023 16:44:50
2  User user1 accessed customer details at 22/02/2023 16:45:33
3  User user2 accessed customer details at 22/02/2023 16:47:27
4  User user1 accessed customer details at 22/02/2023 17:18:31
5  User user1 accessed customer details at 22/02/2023 17:19:28
6  User with ID cust1 accessed customer database.
7  User user1 accessed customer details at 20/03/2023 11:51:37
8
```

5. Fly Weight Structural Design Pattern: -

The Flyweight Structural Design Pattern is a design pattern that is used to minimize the memory usage of an application by sharing data that is common to multiple objects. It is a structural pattern that allows objects to share data, thus reducing the memory usage of the application.

In this pattern, a Flyweight Factory is created that manages a pool of Flyweight objects. Each Flyweight object represents a unique state in the system. When an object is requested, the Flyweight Factory checks if it already exists in the pool. If it does, it returns the existing object; if not, it creates a new object and adds it to the pool.

The Flyweight Structural Design Pattern is a useful tool for minimizing the memory usage of an application and improving its performance. It can reduce the amount of data that needs to be stored in memory and provide a way to manage large amounts of objects in the system. However, it can add complexity to the system and may not be suitable for systems that require a high level of customization or fine-grained control.

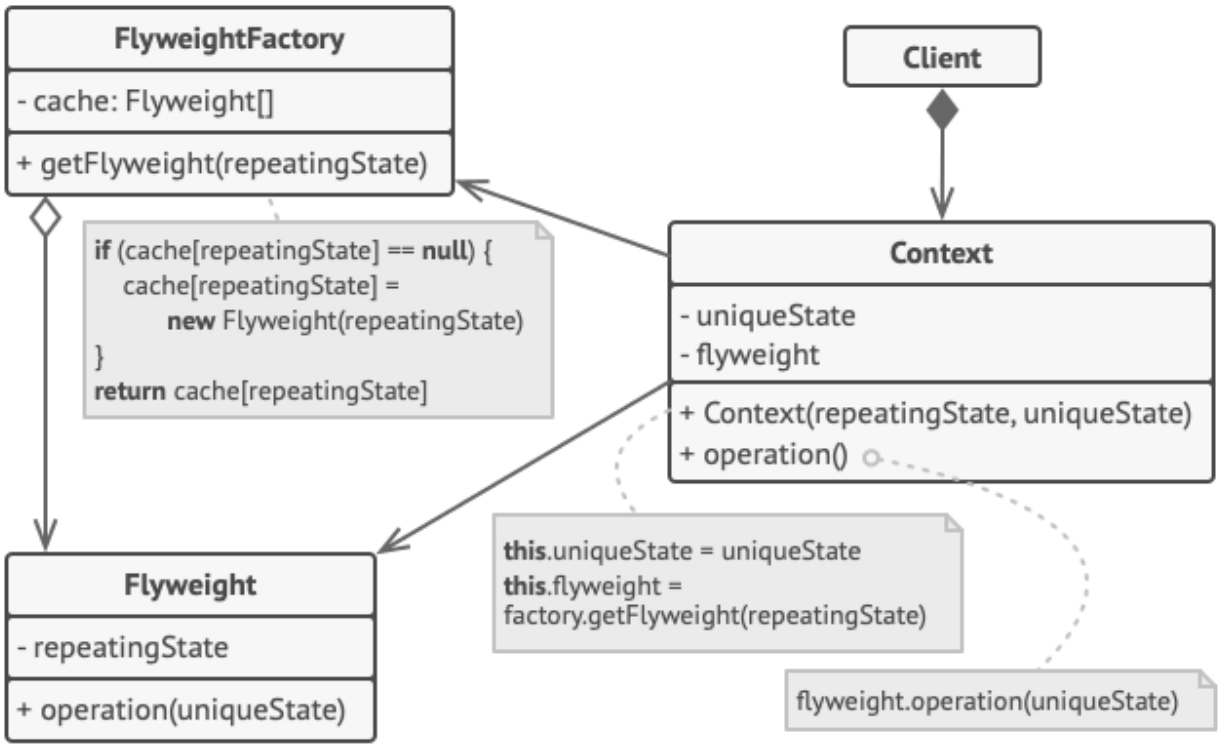
Advantages: -

1. Minimizes the memory usage of the application by sharing data that is common to multiple objects.
2. Improves performance by reducing the number of objects that need to be created.
3. Reduces the amount of data that needs to be stored in memory, which can improve the application's speed and reduce its storage requirements.
4. Provides a way to manage large amounts of objects in the system.

Disadvantages: -

1. Can add complexity to the system by introducing additional layers of abstraction.

2. May result in decreased performance due to the additional processing required to manage the Flyweight objects.
3. May not be suitable for systems that require a high level of customization or fine-grained control.
4. Requires a thorough understanding of the data that is common to multiple objects in the system.



Code: -

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.Scanner;

class TreeFlyWeight{
    String type;
```

```
    String color;
    public TreeFlyWeight(String t, String c){
        type = t;
        color = c;
    }
}

class Tree{
    TreeFlyWeight tf;
    int x;
    int y;
    public Tree(TreeFlyWeight t, int x, int y){
        System.out.println("Tree Object created!!!");
        // type = t;
        // color = c;
        tf = t;
        this.x = x;
        this.y = y;
    }

    public void plantTree(int x, int y){
        this.x = x;
        this.y = y;
        System.out.println("Tree type - "+tf.type+" planted at (x, y) : ("+x+", "+y+")");
    }

}
class TreeFactory{
    Map<String, TreeFlyWeight> mp = new HashMap<String, TreeFlyWeight>();
    TreeFlyWeight tf = null;
    public TreeFlyWeight createTree(String type){
        if(mp.containsKey(type))
            tf = mp.get(type);
        else{
            if(type.equalsIgnoreCase("Neem-Tree")){
                tf = new TreeFlyWeight("Neem-Tree", "Green");
            } else if(type.equalsIgnoreCase("Oak-Tree")){
                tf = new TreeFlyWeight("Oak-Tree", "Orange");
            } else{
                System.out.println("Tree-type not available");
            }
            mp.put(type, tf);
```

```
        }
        return tf;
    }
}
public class lab14_FlyWeight {
    public static void main(String[] args) {
        // for loop for N times
        TreeFactory factory = new TreeFactory();
        List<Tree> forest = new ArrayList<Tree>();
        String[] treelist = new String[] {"Neem-tree", "Oak-tree"};
        Random random = new Random();
        Scanner sc= new Scanner(System.in);
        System.out.println("Enter number of Tree you want");
        int n = sc.nextInt();
        for(int i=0; i<n; i++){
            TreeFlyWeight tfw = factory.createTree(treelist[random.nextInt(treelist.length)]);
            int a =random.nextInt(5);
            int b =random.nextInt(5);
            Tree tree = new Tree(tfw, a, b);
            tree.plantTree(a, b);
            forest.add(tree);

        }
    }
}
```

Output: -

```
Enter number of Tree you want
5
Tree Object created!!!
Tree type - Oak-Tree planted at (x, y) : (0, 4)
Tree Object created!!!
Tree type - Oak-Tree planted at (x, y) : (1, 4)
Tree Object created!!!
Tree type - Neem-Tree planted at (x, y) : (3, 3)
Tree Object created!!!
Tree type - Oak-Tree planted at (x, y) : (1, 0)
Tree Object created!!!
Tree type - Oak-Tree planted at (x, y) : (1, 1)
```

6. Decorate Design Pattern: -

A Decorator Pattern says that just "attach a flexible additional responsibility to an object dynamically".

In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

The Decorator Pattern is also known as Wrapper.

The Decorator Design Pattern is a structural design pattern that allows behaviour to be added to an individual object, dynamically, without affecting the behaviour of other objects from the same class. The pattern is useful when you need to extend the functionality of a class, but want to avoid subclassing or making permanent changes to the class.

Advantages: -

The pattern allows for the dynamic addition of behaviour to an individual object, without affecting other objects of the same class.

It allows you to add behaviour to an object at runtime, rather than at compile time.

The pattern promotes the Open/Closed principle, which states that classes should be open for extension, but closed for modification.

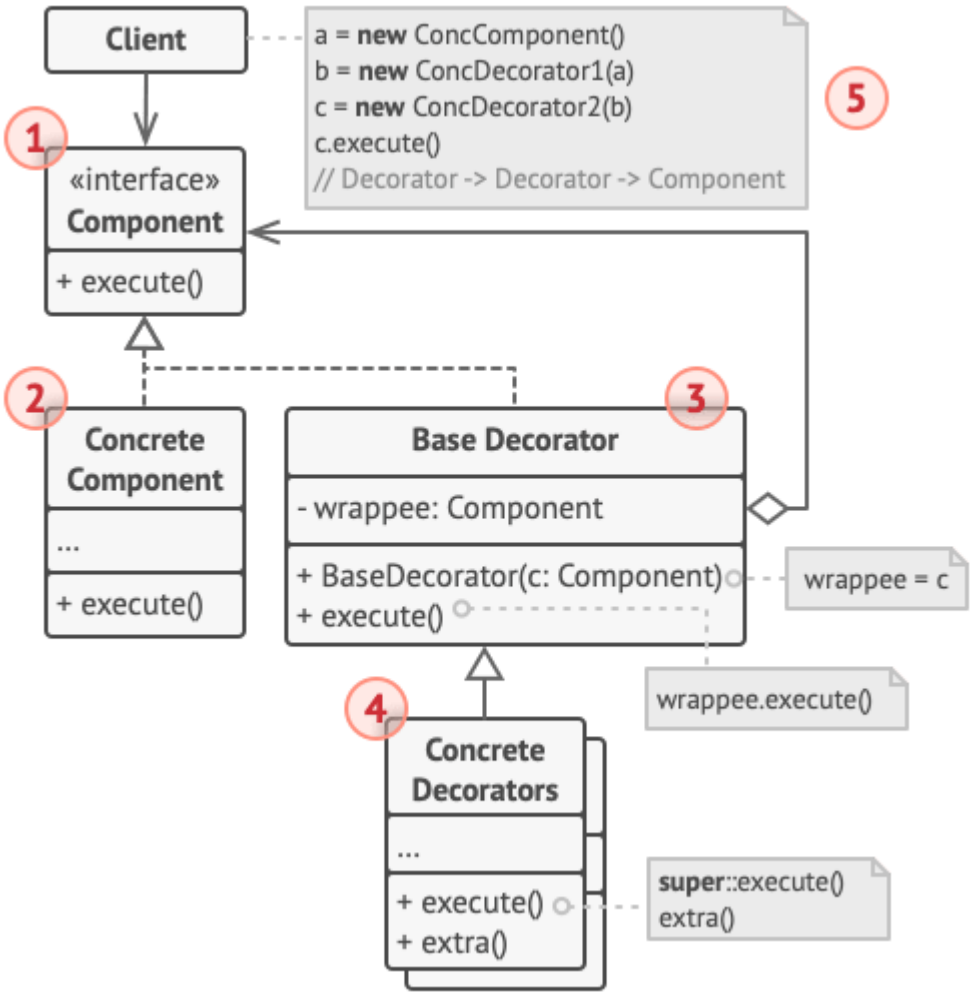
The pattern allows for a large number of combinations of behaviours, without creating a large number of subclasses.

Disadvantages: -

The pattern can lead to an increase in the number of objects created, due to the use of decorators.

It can be difficult to remove a specific behaviour from an object, as it may be buried within multiple decorators.

The pattern can be complex to implement, as it requires the use of multiple classes and interfaces.



Code: -

```
import java.util.Scanner;

// Component Interface
interface IceDish {
    public double cost();
    public String getDescription();
}

// Concrete Component
class BasicIceDish implements IceDish {
    public double cost() {
        return 50.0; // basic ice dish cost
    }

    public String getDescription() {
        return "Basic Ice Dish";
    }
```

```
}

// Decorator Class
abstract class IceDishDecorator implements IceDish {
    protected IceDish iceDish;

    public IceDishDecorator(IceDish iceDish) {
        this.iceDish = iceDish;
    }

    public double cost() {
        return iceDish.cost();
    }

    public String getDescription() {
        return iceDish.getDescription();
    }
}

// Concrete Decorator Classes
class Flavor extends IceDishDecorator {
    public Flavor(IceDish iceDish) {
        super(iceDish);
    }

    public double cost() {
        return iceDish.cost() + 10.0; // additional cost of flavor
    }

    public String getDescription() {
        return iceDish.getDescription() + ", with added flavor";
    }
}

class DryFruit extends IceDishDecorator {
    public DryFruit(IceDish iceDish) {
        super(iceDish);
    }

    public double cost() {
        return iceDish.cost() + 25.0; // additional cost of dry fruit
    }
```

```
    public String getDescription() {
        return iceDish.getDescription() + ", with added dry fruit";
    }
}

class MalaiCreame extends IceDishDecorator {
    public MalaiCreame(IceDish iceDish) {
        super(iceDish);
    }

    public double cost() {
        return iceDish.cost() + 25.0; // additional cost of malai creame
    }

    public String getDescription() {
        return iceDish.getDescription() + ", with added malai creame";
    }
}

class ChocolateSyrup extends IceDishDecorator {
    public ChocolateSyrup(IceDish iceDish) {
        super(iceDish);
    }

    public double cost() {
        return iceDish.cost() + 30.0; // additional cost of chocolate syrup
    }

    public String getDescription() {
        return iceDish.getDescription() + ", with added chocolate syrup";
    }
}

class CoconutGrinded extends IceDishDecorator {
    public CoconutGrinded(IceDish iceDish) {
        super(iceDish);
    }

    public double cost() {
        return iceDish.cost() + 15.0; // additional cost of grinded coconut
    }
```

```
    public String getDescription() {
        return iceDish.getDescription() + ", with added grinded coconut";
    }
}

public class lab15_Decorator_icedish {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        IceDish iceDish = new BasicIceDish();
        boolean done = false;

        while (!done) {
            System.out.println("What would you like to add to your ice dish?");
            System.out.println("1. Flavor (+10 Rs.)");
            System.out.println("2. Dry fruit (+25 Rs.)");
            System.out.println("3. Malai creame (+25 Rs.)");
            System.out.println("4. Chocolate syrup (+30 Rs.)");
            System.out.println("5. Grinded coconut (+15 Rs.)");
            System.out.println("6. Done");

            int choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    iceDish = new Flavor(iceDish);
                    break;
                case 2:
                    iceDish = new DryFruit(iceDish);
                    break;
                case 3:
                    iceDish = new MalaiCreame(iceDish);
                    break;
                case 4:
                    iceDish = new ChocolateSyrup(iceDish);
                    break;
                case 5:
                    iceDish = new CoconutGrinded(iceDish);
                    break;
                case 6:
                    done = true;
                    break;
```

```
                default:
                    System.out.println("Invalid choice, please try again.");
                    break;
            }
        }

        System.out.println("Your ice dish has been created:");
        System.out.println(iceDish.getDescription());
        System.out.println("Total cost: " + iceDish.cost() + " Rs.");
    }

}
```

Output: -

```
What would you like to add to your ice dish?
1. Flavor (+10 Rs.)
2. Dry fruit (+25 Rs.)
3. Malai creame (+25 Rs.)
4. Chocolate syrup (+30 Rs.)
5. Grinded coconut (+15 Rs.)
6. Done
2
What would you like to add to your ice dish?
1. Flavor (+10 Rs.)
2. Dry fruit (+25 Rs.)
3. Malai creame (+25 Rs.)
4. Chocolate syrup (+30 Rs.)
5. Grinded coconut (+15 Rs.)
6. Done
3
```

```
What would you like to add to your ice dish?
1. Flavor (+10 Rs.)
2. Dry fruit (+25 Rs.)
3. Malai creame (+25 Rs.)
4. Chocolate syrup (+30 Rs.)
5. Grinded coconut (+15 Rs.)
6. Done
1
What would you like to add to your ice dish?
1. Flavor (+10 Rs.)
2. Dry fruit (+25 Rs.)
3. Malai creame (+25 Rs.)
4. Chocolate syrup (+30 Rs.)
5. Grinded coconut (+15 Rs.)
6. Done
6
```

```
Your ice dish has been created:
Basic Ice Dish, with added dry fruit, with added malai creame, with added flavor
Total cost: 110.0 Rs.
```