Structural Design Pattern

Composite Design Pattern

Aim:
The aim of implementing the Composite design pattern in this context is to create a computer part system capable of representing hierarchical structures of computer components and composite parts, facilitating the organization and management of complex configurations within a computer assembly.

Description:
The Composite pattern implemented in this code facilitates the creation of hierarchical structures for computer parts, allowing composite parts to contain both individual components and other composite parts. By offering a uniform interface for manipulation, the code simplifies the management of computer assemblies by treating leaf components and composite parts uniformly. This approach enhances extensibility and maintainability as new components can be easily added without modifying existing code. Leveraging recursion, operations such as displaying the price structure are efficiently executed. Overall, the pattern provides an elegant solution for representing and managing complex configurations of computer parts, promoting code reuse and enhancing organizational clarity within computer assembly systems.

Code:

1. ComputerPart

```
package com.deb.composite;

import java.util.*;


interface Component
{
	void showPrice();
}

class Leaf implements Component
{
	int price;
	String name;

	public Leaf(int price, String name)
	{
		this.price = price;
		this.name = name;
	}

	public void showPrice()
	{
		System.out.println(name+": price");
	}
}

class Composite implements Component
```

```
{
	String name;
	List<Component> components=new ArrayList();

	public Composite(String name)
	{
		this.name = name;
	}

	public void addComponent(Component com)
	{
		components.add(com);
	}

	public void showPrice()
	{
		System.out.println(name);
		for(Component c:components)
		{
			c.showPrice();
		}
	}
}

public class ComputerPart {
}
```

2. CompositeTest

```
package com.deb.composite;

public class CompositeTest {
	public static void main(String[] args)
	{
	Component hd = new Leaf(4000, "harddrive");
	Component mouse = new Leaf(4000, "Mouse");
	Component monitor = new Leaf(4000, "Monitor");
	Component ram = new Leaf(4000, "RAM");
	Component cpu = new Leaf(4000, "CPU");
	Composite ph = new Composite("Peri");
	Composite cabinet = new Composite("Cabinet");
	Composite mb = new Composite("MotherBoard");
	Composite computer = new Composite("Computer");

	mb.addComponent(cpu);
	mb.addComponent(ram);

	ph.addComponent(mouse);
	ph.addComponent(monitor);

	cabinet.addComponent(hd);
	cabinet.addComponent(mb);
```

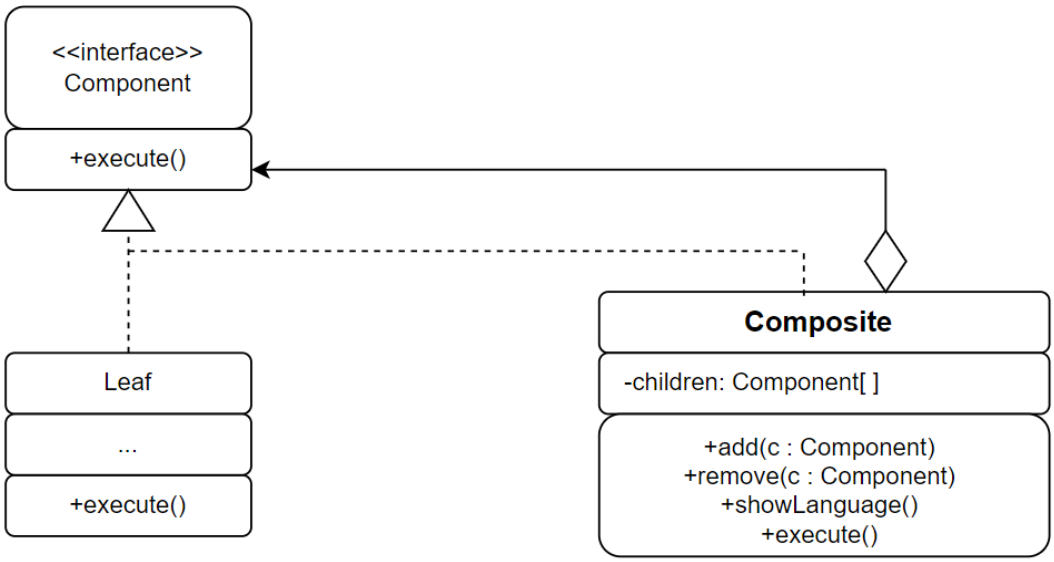
```
        computer.addComponent(ph);
        computer.addComponent(cabinet);

        ram.showPrice();
        ph.showPrice();
        computer.showPrice();
        }
}
```

Output:

```
<terminated> CompositeTest [Java Applic
RAM: price:4000
Peri
Mouse: price:4000
Monitor: price:4000
Computer
Peri
Mouse: price:4000
Monitor: price:4000
Cabinet
harddrive: price:4000
MotherBoard
CPU: price:4000
RAM: price:4000
```

Class Diagram:



Aim:
The aim of implementing the Composite design pattern in this context is to create a computer part system capable of representing hierarchical structures of computer components and composite parts, facilitating the organization and management of complex configurations within a computer assembly.

Description:
The Composite pattern implemented in this code facilitates the creation of hierarchical structures for computer parts, allowing composite parts to contain both individual components and other composite parts. By offering a uniform interface for manipulation, the code simplifies the management of computer assemblies by treating leaf components and composite parts uniformly. This approach enhances extensibility and maintainability as new components can be easily added without modifying existing code. Leveraging recursion, operations such as displaying the price structure are efficiently executed. Overall, the pattern provides an elegant solution for representing and managing complex configurations of computer parts, promoting code reuse and enhancing organizational clarity within computer assembly systems.

Code:

1. MenuExample.java

```
package Personal;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

// Component interface
interface MenuComponent {
    void display();
}

// Leaf class representing a single menu item
class MenuItem implements MenuComponent {
    private String name;

    public MenuItem(String name) {
        this.name = name;
    }

    public void display() {
        System.out.println(name);
    }}

// Composite class representing a menu containing submenus or menu items
class Menu implements MenuComponent {
    private String name;
    private List<MenuComponent> menuComponents = new ArrayList<>();

    public Menu(String name) {
        this.name = name;
    }
```

```
    public void add(MenuComponent component) {
        menuComponents.add(component);
    }

    public void remove(MenuComponent component) {
        menuComponents.remove(component);}

    public void display() {
        System.out.println("Menu: " + name);
        for (MenuComponent component : menuComponents) {
            component.display();
        }}}

public class MenuExample {
    public static void main(String[] args) {
        // Create main menu
        Menu mainMenu = new Menu("Main Menu");

        // Create submenus
        Menu fileMenu = new Menu("File");
        Menu editMenu = new Menu("Edit");
        Menu viewMenu = new Menu("View");

        // Add submenus to the main menu
        mainMenu.add(fileMenu);
        mainMenu.add(editMenu);
        mainMenu.add(viewMenu);

        // Create menu items for the submenus
        MenuItem newItem = new MenuItem("New");
        MenuItem openItem = new MenuItem("Open");
        MenuItem saveItem = new MenuItem("Save");
        MenuItem exitItem = new MenuItem("Exit");
        MenuItem cutItem = new MenuItem("Cut");
        MenuItem copyItem = new MenuItem("Copy");
        MenuItem pasteItem = new MenuItem("Paste");
        MenuItem zoomInItem = new MenuItem("Zoom In");
        MenuItem zoomOutItem = new MenuItem("Zoom Out");

        // Add menu items to the submenus
        fileMenu.add(newItem);
        fileMenu.add(openItem);
        fileMenu.add(saveItem);
        fileMenu.add(exitItem);
        editMenu.add(cutItem);
        editMenu.add(copyItem);
        editMenu.add(pasteItem);

        viewMenu.add(zoomInItem);
        viewMenu.add(zoomOutItem);
```

```
        // Display the main menu
        mainMenu.display();
        // Simulate user interaction by selecting a submenu
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the submenu to display (File/Edit/View): ");
        String selectedMenu = scanner.nextLine().trim().toLowerCase();

        // Display the selected submenu
        switch (selectedMenu) {
            case "file":
                fileMenu.display();
                break;
            case "edit":
                editMenu.display();
                break;
            case "view":
                viewMenu.display();
                break;
            default:
                System.out.println("Invalid submenu selection.");
        }
        scanner.close();
    }}
```

Output:

```
(base) go-d-code@gem-zangetsu:~/Roger/College/Design_lab_Codes/Composite Design/Personal$ cd "/home/go-d-code/Roger/College/Design_lab_Codes/Composite Design/Personal/" && javac MenuExample.java && java MenuExample
Menu: Main Menu
Menu: File
New
Open
Save
Exit
Menu: Edit
Cut
Copy
Paste
Menu: View
Zoom In
Zoom Out
Enter the submenu to display (File/Edit/View): File
Menu: File
New
Open
Save
Exit
```

Class Diagram: