| Part A | |
|---|---|
| **Class B Tech CSE 2nd Year** | **Sub: Design Pattern Lab** |
| **Aim:** Write a program to implement the Composite Design Pattern | |
| **Prerequisite:** Basics of Object Oriented Programming | |
| **Outcome:** To impart knowledge of Design Pattern techniques | |
| **Theory:** Explain about the Composite Design Pattern. | |

## Composite Pattern

The Composite pattern is a structural design pattern used in software engineering. Its main purpose is to treat individual objects and compositions of objects (composites) uniformly, allowing clients to work with both in a consistent manner. This pattern is particularly useful when dealing with hierarchical structures or tree-like relationships among objects.

**Working**:

- Component: This is the base interface or abstract class that defines the common operations for both individual objects and composites. It usually includes methods for adding, removing, accessing, and manipulating child components, as well as any other operations relevant to the hierarchy.

- Leaf: A Leaf is the basic building block of the hierarchy. It represents individual objects that do not have children, meaning they cannot contain other components.

- Composite: A Composite is a container that can hold other components, including both leaves and other composites. It implements the same interface as the Component, allowing clients to treat it the same way they would treat a Leaf. This recursive structure enables the creation of complex hierarchies where individual objects and compositions can be treated uniformly.

- Client: The client is the code that interacts with the components in the hierarchy. It can work with both individual objects and composites interchangeably, thanks to the uniform interface provided by the Component.

**Key components:**

- Transparency: One of the main advantages of the Composite pattern is its transparency. Clients can interact with individual objects and compositions in the same way, without needing to distinguish between them. This simplifies client code and promotes code reuse.

- Recursive Structure: The Composite pattern relies on a recursive structure, where composites can contain other composites as well as individual objects. This enables the creation of complex hierarchies with varying levels of nesting.

- Safety Measures: Depending on the requirements, additional safety measures can be implemented to handle operations that are not applicable to certain types of components. For example, attempting to add a child to a Leaf could result in an exception or be silently ignored, depending on the desired behavior.

- Use Cases: The Composite pattern is commonly used in scenarios involving hierarchical structures, such as file systems, organizational charts, graphical user interfaces (GUIs), and composite graphical objects like shapes and diagrams.

Overall, the Composite pattern provides a flexible and intuitive way to work with hierarchical structures in software. It simplifies client code, promotes code reuse, and enables the construction of complex systems with ease.

| Part B |
| --- |
| **Steps:** Write procedure/code here. |
| **Output:** Paste Output here. |
| **Observation & Learning:** Give your observation and learning about the experiment. |
| **Conclusion:** Give your conclusion for the experiment. |

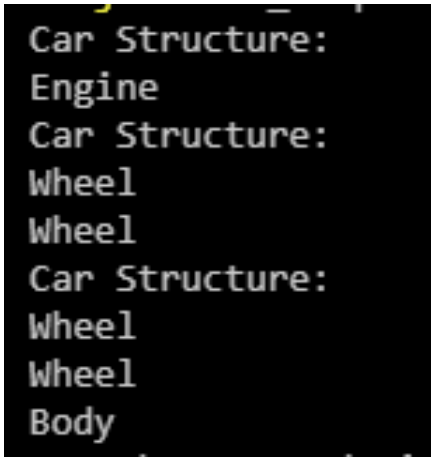
## Code

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

// Component interface
interface CarComponent {
    void display(); // Common method for displaying car components
}

// Leaf class representing the engine
class Engine implements CarComponent {
    @Override
    public void display() {
        System.out.println("Engine");
    }
}

// Leaf class representing a wheel
class Wheel implements CarComponent {
    @Override
    public void display() {
        System.out.println("Wheel");
    }
}

// Leaf class representing the car body
class Body implements CarComponent {
    @Override
    public void display() {
        System.out.println("Body");
    }
}

// Composite class representing the entire car
class CarComposite implements CarComponent {
    private List<CarComponent> components = new ArrayList<>(); // List to
store child components

    // Method to display the entire car structure
    @Override
```

```
    public void display() {
        System.out.println("Car Structure:");
        for (CarComponent component : components) {
            // System.out.println("Car Structure:");
            component.display(); // Delegating display to child components
        }
    }

    // Method to add a component to the car
    public void addComponent(CarComponent component) {
        components.add(component);
    }

    // Method to remove a component from the car
    public void removeComponent(CarComponent component) {
        components.remove(component);
    }
}

// Example usage
public class Pr6_Composite {
    public static void main(String[] args) {
        // Create leaf objects
        Engine engine = new Engine();
        Wheel frontLeftWheel = new Wheel();
        Wheel frontRightWheel = new Wheel();
        Wheel rearLeftWheel = new Wheel();
        Wheel rearRightWheel = new Wheel();
        Body carBody = new Body();

        // Create composite objects
        CarComposite frontAxle = new CarComposite();
        CarComposite rearAxle = new CarComposite();
        CarComposite car = new CarComposite();

        // Build the car structure
        frontAxle.addComponent(frontLeftWheel);
        frontAxle.addComponent(frontRightWheel);

        rearAxle.addComponent(rearLeftWheel);
        rearAxle.addComponent(rearRightWheel);

        car.addComponent(engine);
        car.addComponent(frontAxle);
        car.addComponent(rearAxle);
        car.addComponent(carBody);

        // Display the entire car structure
```

```
        car.display();
    }
}
```

## Output:




## Observation & Learning:

From the experiment, I learned that the Composite pattern is like a toolbox for managing groups of things in software. It helps organize stuff in a tree-like structure, making it easy to handle both individual items and whole groups of items. This pattern simplifies how we write code and allows for better reuse of code.

## Conclusion:

In summary, the experiment showed that the Composite pattern is a handy tool for organizing complex structures in software. It makes code more flexible and easier to work with, especially when dealing with hierarchical relationships. Using this pattern can lead to more efficient and manageable software designs.