Composite Design Pattern

The Composite Pattern is a structural design pattern that allows you to compose objects into tree structures to represent part-whole hierarchies. This pattern treats individual objects and compositions of objects uniformly. It is particularly useful when you want to treat a group of objects in the same way as a single instance of an object.

Use Case:

Imagine you are designing a graphics drawing application where shapes like circles, squares, and lines can be grouped together to form more complex shapes. You want to treat individual shapes and groups of shapes uniformly, allowing operations to be performed on both without differentiating between them.

Components:

1. Component (Graphic): Declares the interface for objects in the composition.
2. Leaf (Circle, Square, Line): Represents the leaf objects in the composition that do not have any children.
3. Composite (CompositeGraphic): Represents the composite objects that can have children. Implements the component interface and holds child components.

Example: Graphics Drawing Application

1. Component Interface (Graphic):

```java
public interface Graphic {
    void draw();
}
```

2. Leaf (Circle, Square, Line):

```java
public class Circle implements Graphic {
```

```
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

public class Square implements Graphic {
    @Override
    public void draw() {
        System.out.println("Drawing a Square");
    }
}

public class Line implements Graphic {
    @Override
    public void draw() {
        System.out.println("Drawing a Line");
    }
}
```

3. Composite (CompositeGraphic):

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

public class CompositeGraphic implements Graphic {
```

```
    private List<Graphic> childGraphics = new ArrayList<>();

    public void add(Graphic graphic) {
        childGraphics.add(graphic);
    }

    public void remove(Graphic graphic) {
        childGraphics.remove(graphic);
    }

    @Override
    public void draw() {
        for (Graphic graphic : childGraphics) {
            graphic.draw();
        }
    }
}
```

4. Client Code:

```java
public class CompositePatternDemo {
    public static void main(String[] args) {
        // Create leaf objects
        Graphic circle = new Circle();
        Graphic square = new Square();
        Graphic line = new Line();
```

```
        // Create composite objects
        CompositeGraphic composite1 = new CompositeGraphic();
        composite1.add(circle);
        composite1.add(square);

        CompositeGraphic composite2 = new CompositeGraphic();
        composite2.add(composite1);
        composite2.add(line);

        // Draw all graphics
        composite2.draw();
    }
}
```

Key Points:

- Uniformity: The Composite Pattern allows you to treat individual objects and compositions uniformly.
- Recursive Composition: You can create complex tree structures with objects and sub-objects.
- Flexibility: The pattern makes it easier to add new types of components or composites without altering existing code.