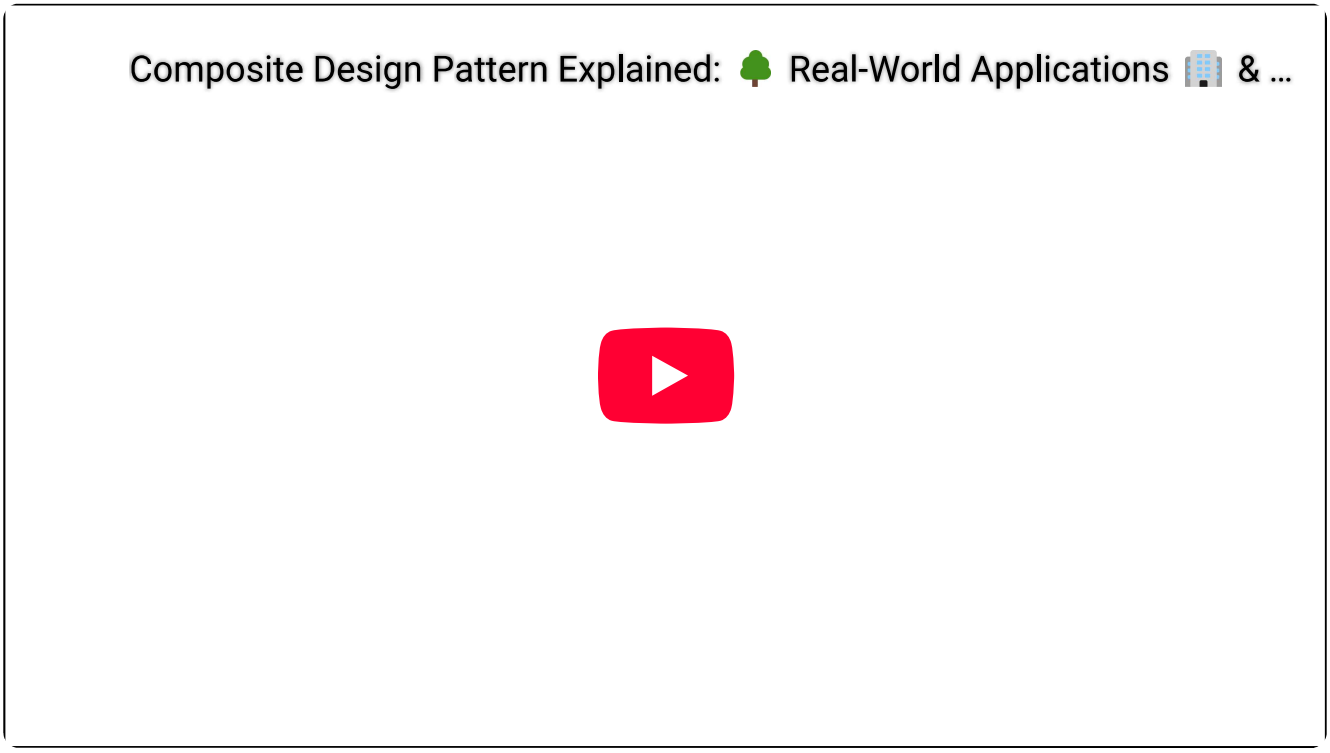
Search articles...

Composite Design Pattern



Topic Tags:

System Design LLD

Problem Statement: Managing Hierarchies of Devices 🏠

Imagine you're designing a smart home system. Your goal is to control various devices such as rooms and individual appliances within a smart home.

For example:
• A Room contains multiple devices like air conditioners, lights, and coffee machines.
• A Floor may contain multiple rooms.
• The House may consist of multiple floors.

You want your SmartHomeController to control devices at any level—whether it's an individual light, an entire room, or the whole house.

The Problem: Managing Hierarchical Structures 🔗

The challenge arises because the devices are organized in a hierarchy. For example:

1. A Room contains multiple devices (e.g., AirConditioner, SmartLight).
2. A Floor contains multiple rooms.
3. The House contains multiple floors.

Your app needs to interact with this hierarchy seamlessly. For example, turning off all devices in the house should automatically turn off devices in every floor, room, and individual appliance.

Solving It the Traditional Way: A Messy Solution 🛠️

Let's look at how you might solve this problem in a straightforward but inflexible way:

Java

```java
public class SmartHomeController {
  public static void main(String[] args) {
    // Manually managing devices and groups
    AirConditioner airConditioner = new AirConditioner();
    SmartLight smartLight = new SmartLight();
    System.out.println("Turning ON devices in Room 1...");
    airConditioner.turnOn();
    smartLight.turnOn();
    System.out.println("Turning OFF devices in Room 1...");
    airConditioner.turnOff();
    smartLight.turnOff();
    // Manually managing multiple rooms
    System.out.println("Turning ON devices in Floor 1...");
    airConditioner.turnOn();
    smartLight.turnOn();
    airConditioner.turnOn(); // Room 2
    smartLight.turnOn(); // Room 2
    System.out.println("Turning OFF devices in Floor 2...");
    airConditioner.turnOff();
    smartLight.turnOff();
    airConditioner.turnOff(); // Room 3
    smartLight.turnOff(); // Room 3
    System.out.println("Turning ON all devices in the house...");
    airConditioner.turnOn();
    smartLight.turnOn();
    // Add more logic as you scale...
```

```
27     }
28 }
```

The SmartHomeController class creates each device explicitly by calling the constructor of the respective device class. This approach works but becomes cumbersome as the hierarchy grows. Adding new device types or changing the way devices are controlled requires modifying the existing code, leading to tightly coupled and error-prone code.

The Challenge: How Can We Manage Complex Hierarchies Efficiently? 🤔

An interviewer might ask:
1. What happens when you add a new type of component?
For example, a new Garage or Garden that also contains devices.

2. What if the hierarchy changes?
For instance, a new concept like a Zone is introduced, which groups rooms.

The hardcoded logic to traverse the hierarchy becomes unmanageable, especially when new types are introduced or the structure evolves.

Ugly Code: When We Realize the Code Needs Restructuring 🛠️

Let's say the logic for managing devices becomes more complex:
• User Input:
The user decides which level to control (e.g., room, floor, or house).

• Nested Hierarchies:
Each room contains multiple devices, and each floor contains multiple rooms.

• Dynamic Changes:
Rooms, floors, or devices may be added or removed dynamically.
If we manage the hierarchy manually, the code quickly becomes tightly coupled and error-prone, as shown below:

Why Is This Code Problematic? 🤔

1. Hardcoding Logic:

Each level (device, room, floor, house) is managed manually, duplicating logic and making the code difficult to maintain.

2. Fragility:
Adding new components (e.g., a Garage or Garden) or modifying existing ones requires changing the code in multiple places.

3. Tight Coupling:
The controller is tightly coupled to specific devices, making the code less reusable and harder to extend.

4. Scaling Issues:
Managing a real-world hierarchy with hundreds of devices or dozens of rooms would make the code unmanageable.

The Savior: Composite Design Pattern 🦸

> *The Composite Design Pattern is specifically designed to handle hierarchies. It allows you to treat individual objects and groups of objects (composites) uniformly.*
> *In our smart home system, the pattern enables you to control individual devices, rooms, floors, and even the entire house seamlessly through a single interface.*

How the Composite Design Pattern Works 🔧

The Composite Pattern achieves this by defining a common interface for both individual objects and composites (groups). Each composite can hold child components (which could be either individual objects or other composites), forming a tree structure.

For our smart home system:
1. Individual Devices (e.g., AirConditioner, SmartLight) implement the common interface.
2. Groups of Devices (e.g., Room, Floor, House) implement the same interface and delegate actions to their children.

Solving the Problem with Composite Design Pattern 🌳

Here's how we can solve the problem using the Composite Design Pattern:

Step 1: Define a Common Interface

The first step is to define a common interface for all components in the hierarchy.

Java

```java
// SmartComponent.java - Common interface for all components
public interface SmartComponent {
    void turnOn();  // Turn on the component
```

```
    void turnOff(); // Turn off the component
}
```

Step 2: Create Concrete Classes for Individual Devices

Each device implements the SmartComponent interface.

Java

```java
// AirConditioner.java
public class AirConditioner implements SmartComponent {
  @Override
  public void turnOn() {
    System.out.println("Air Conditioner is now ON.");
  }
  @Override
  public void turnOff() {
    System.out.println("Air Conditioner is now OFF.");
  }
}

// SmartLight.java
public class SmartLight implements SmartComponent {
  @Override
  public void turnOn() {
    System.out.println("Smart Light is now ON.");
  }
  @Override
  public void turnOff() {
    System.out.println("Smart Light is now OFF.");
  }
}
```

Step 3: Create Composite Classes for Groups

The composite classes represent groups of components (e.g., Room, Floor, House).

Java

```java
import java.util.ArrayList;
import java.util.List;
// Composite class for groups of components
public class CompositeSmartComponent implements SmartComponent {
  private List<SmartComponent> components = new ArrayList<>();
public void addComponent(SmartComponent component) {
    components.add(component);
  }
  public void removeComponent(SmartComponent component) {
    components.remove(component);
  }
  @Override
  public void turnOn() {
    for (SmartComponent component : components) {
      component.turnOn();
    }
  }
  @Override
  public void turnOff() {
    for (SmartComponent component : components) {
      component.turnOff();
    }
  }
}
```

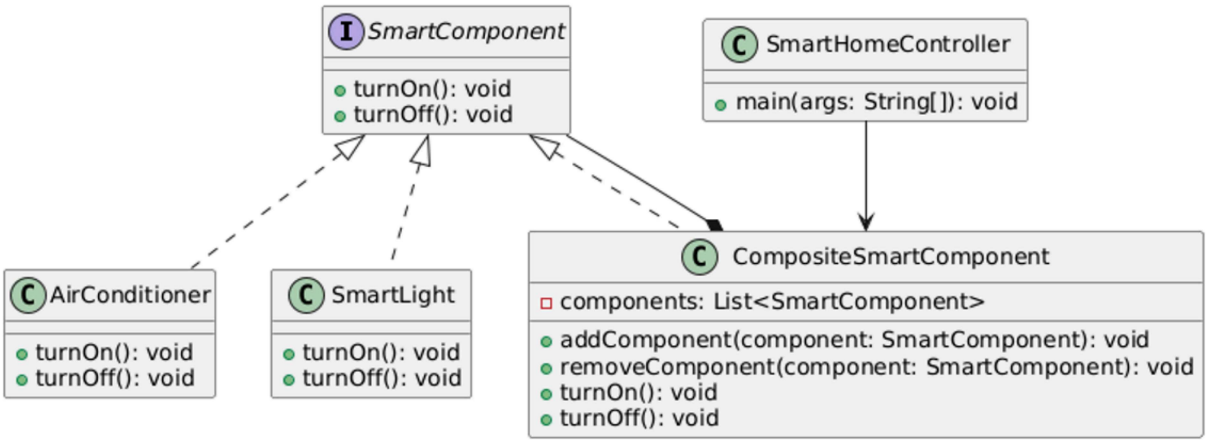
Step 4 : Build the Hierarchy and Use :

Here's how you can build the smart home hierarchy and control it:

Java

```java
public class SmartHomeController {
  public static void main(String[] args) {
    // Create individual devices
    SmartComponent airConditioner = new AirConditioner();
    SmartComponent smartLight = new SmartLight();
    // Create a room and add devices
    CompositeSmartComponent room1 = new CompositeSmartComponent();
    room1.addComponent(airConditioner);
```

```
        room1.addComponent(smartLight);
        // Add more rooms for demonstration
        CompositeSmartComponent room2 = new CompositeSmartComponent();
        room2.addComponent(new AirConditioner());
        room2.addComponent(new SmartLight());
        // Create a floor and add rooms
        CompositeSmartComponent floor = new CompositeSmartComponent();
        floor.addComponent(room1);
        floor.addComponent(room2);
        // Create the house and add floors
        CompositeSmartComponent house = new CompositeSmartComponent();
        house.addComponent(floor);
        // Control the entire house
        System.out.println("Turning ON all devices in the house:");
        house.turnOn();
        System.out.println("nTurning OFF all devices in the house:");
        house.turnOff();
        // Control a single floor
        System.out.println("nTurning ON all devices on the first floor:");
        floor.turnOn();
        System.out.println("nTurning OFF all devices on the first floor:");
        floor.turnOff();
        // Control a single room
        System.out.println("nTurning ON all devices in Room 1:");
        room1.turnOn();
        System.out.println("nTurning OFF all devices in Room 1:");
        room1.turnOff();
    }
}
```



Advantages of Using the Composite Design Pattern 🏆

1. Uniformity:
Treat individual devices and groups (rooms, floors, etc.) uniformly using the SmartComponent interface.

2. Scalability:
Easily add new components (e.g., Garage, Garden) without changing the existing code.

3. Decoupling:
The controller is decoupled from the specific structure of the hierarchy, making the system more modular.

4. Flexibility:
Changes in the hierarchy (e.g., adding or removing components) are easily handled by the composite structure.

Real-Life Use Cases and Examples of the Composite Pattern 🌎

1. File Systems:

Files and directories in an operating system follow the composite pattern, where directories contain files or other directories.

2. Graphics Rendering:
GUI frameworks use the composite pattern for rendering graphical components, where containers (e.g., panels) hold other components (e.g., buttons, labels).

3. Organization Hierarchies:

Companies use the composite pattern to model organizational structures, where departments contain employees or other sub-departments.

Conclusion 🎯

The Composite Design Pattern simplifies managing hierarchical structures by allowing you to treat individual objects and composites uniformly.
In our SmartHomeController example, it enables seamless control of individual devices, rooms, floors, and the entire house, making the system clean, scalable, and easy to maintain.
The Composite Pattern is essential for building systems where hierarchies are a natural fit, ensuring flexibility and reducing complexity. 🌟