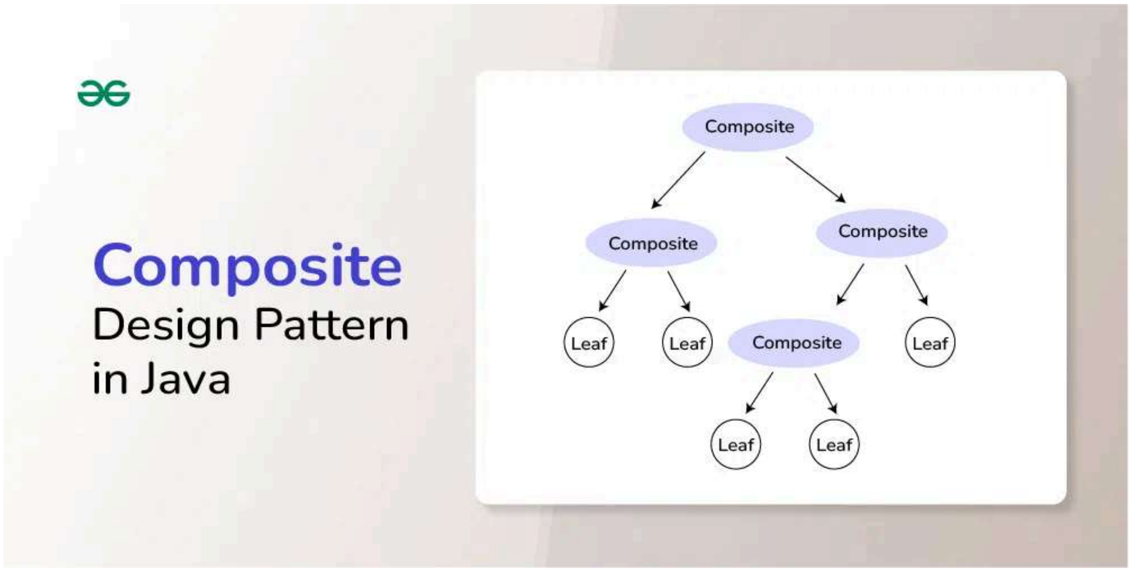
Composite Design Pattern in Java

Last Updated : 03 Jan, 2025

The Composite Design Pattern is a structural design pattern that lets you compose objects into tree-like structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions of objects uniformly. In other words, whether dealing with a single object or a group of objects (composite), clients can use them interchangeably.

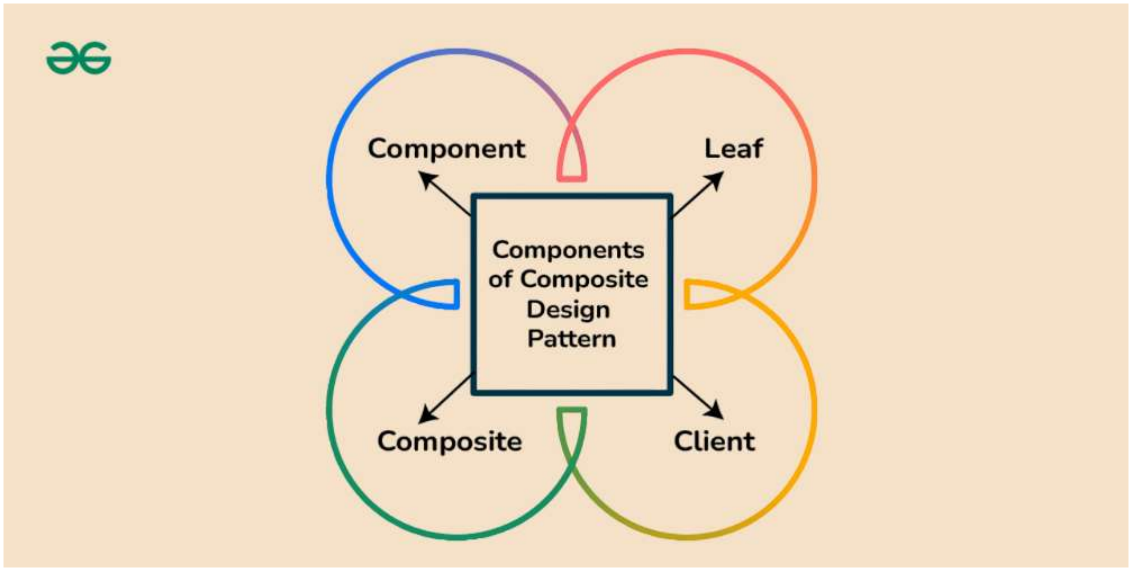
> *As described by the Gang of four, "Compose objects into tree structure to represent **part-whole hierarchies**. Composite lets client treat individual objects and compositions of objects uniformly".*



The key concept is that you can manipulate a single instance of the object just as you would manipulate a group of them. The operations you can perform on all the composite objects often have the least common denominator relationship.

Important Topics for the Composite Design Pattern in Java

Components of Composite Design Pattern



1. Component

The component declares the interface for objects in the composition and for accessing and managing its child components. This is like a blueprint that tells us what both individual items (leaves) and groups of items (composites) should be able to do. It lists the things they all have in common.

2. Leaf

Leaf defines behavior for primitive objects in the composition. This is the basic building block of the composition, representing individual objects that don't have any child components. Leaf elements implement the operations defined by the Component interface.

3. Composite

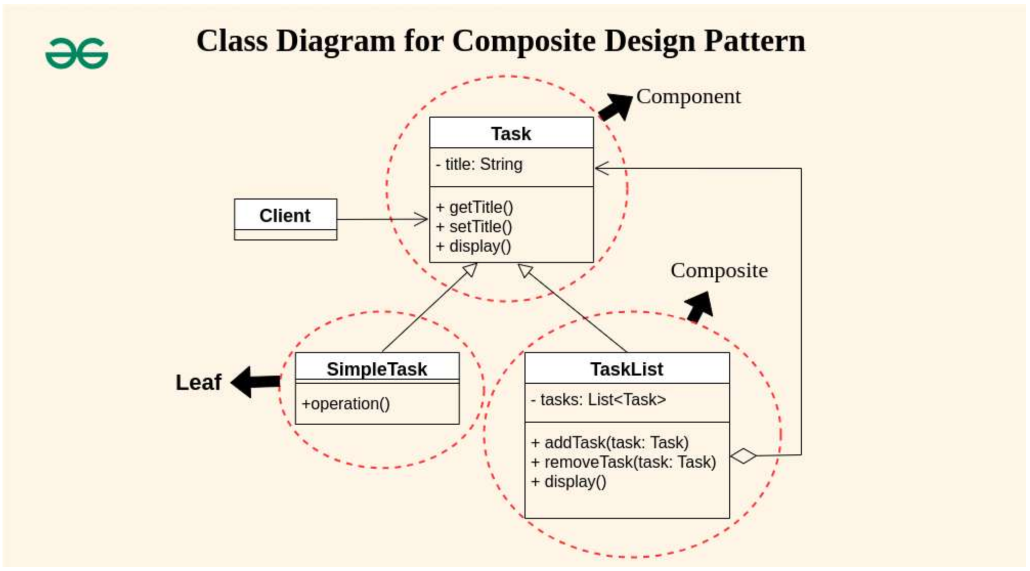
Composite stores child components and implements child-related operations in the component interface. This is a class that has child components, which can be either leaf elements or other composites. A composite class implements the methods declared in the Component interface, often by delegating the operations to its child components.

4. Client

The client manipulates the objects in the composition through the component interface. The client uses the component class interface to interact with objects in the composition structure. If the recipient is a leaf then the request is handled directly. If the recipient is a composite, then it usually forwards the request to its child components, possibly performing additional operations before and after forwarding.

Composite Design Pattern example in Java

> *Imagine you are building a project management system where tasks can be either simple tasks or a collection of tasks (subtasks) forming a larger task.*



1. Task (Component)

- Represents the common interface for both simple tasks and task lists.
- Defines methods such as `getTitle()`, `setTitle()`, and `display()`.

```
// Component
```

```java

public interface Task {
    String getTitle();
    void setTitle(String title);
    void display();
}
```

2. SimpleTask (Leaf)

- Represents an individual task with a title.
- Implements the `Task` interface.

```java
// Leaf

public class SimpleTask implements Task {
    private String title;

    public SimpleTask(String title) {
        this.title = title;
    }

    @Override
    public String getTitle() {
        return title;
    }

    @Override
    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public void display() {
        System.out.println("Simple Task: " + title);
    }
}
```

3. TaskList (Composite)

- Represents a collection of tasks, which can include both simple tasks and other task lists.
- Implements the `Task` interface but also has a list of tasks (`List<Task>`).
- Defines methods to add, remove, and display tasks.

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

// Composite

public class TaskList implements Task {
    private String title;
    private List<Task> tasks;

    public TaskList(String title) {
        this.title = title;
        this.tasks = new ArrayList<>();
    }

    @Override
    public String getTitle() {
        return title;
    }

    @Override
    public void setTitle(String title) {
        this.title = title;
    }

    public void addTask(Task task) {
        tasks.add(task);
    }

    public void removeTask(Task task) {
        tasks.remove(task);
    }

    @Override
    public void display() {
        System.out.println("Task List: " + title);
        for (Task task : tasks) {
            task.display();
        }
    }
}
```

4. TaskManagementApp (Client)

- Represents the application that uses the Composite Design Pattern to manage tasks.
- It creates a mix of simple tasks and task lists, showcasing how the Composite pattern allows treating both individual tasks and task collections uniformly.
- The created tasks are displayed in a hierarchical structure to illustrate the pattern's flexibility and uniform handling of different task types.

```java
// Client

public class TaskManagementApp {
    public static void main(String[] args) {
        // Creating simple tasks
        Task simpleTask1 = new SimpleTask("Complete Coding");
        Task simpleTask2 = new SimpleTask("Write
Documentation");

        // Creating a task list
        TaskList projectTasks = new TaskList("Project Tasks")
        projectTasks.addTask(simpleTask1);
        projectTasks.addTask(simpleTask2);

        // Nested task list
        TaskList phase1Tasks = new TaskList("Phase 1 Tasks");
        phase1Tasks.addTask(new SimpleTask("Design"));
        phase1Tasks.addTask(new SimpleTask("Implementation")

        projectTasks.addTask(phase1Tasks);

        // Displaying tasks
        projectTasks.display();
    }
}
```

Complete code for the above example:

This code includes the `Task`, `SimpleTask`, `TaskList`, and `TaskManagementApp` classes. It demonstrates the Composite Design Pattern for organizing tasks in a project management system.

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

// Component
interface Task {
    String getTitle();
    void setTitle(String title);
    void display();
}

// Leaf
class SimpleTask implements Task {
    private String title;

    public SimpleTask(String title) {
        this.title = title;
    }

    @Override
    public String getTitle() {
        return title;
    }

    @Override
    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public void display() {
        System.out.println("Simple Task: " + title);
    }
}

// Composite
class TaskList implements Task {
    private String title;
    private List<Task> tasks;

    public TaskList(String title) {
        this.title = title;
        this.tasks = new ArrayList<>();
    }

    @Override
    public String getTitle() {
        return title;
    }

    @Override
    public void setTitle(String title) {
        this.title = title;
    }

    public void addTask(Task task) {
        tasks.add(task);
    }

    public void removeTask(Task task) {
        tasks.remove(task);
    }

    @Override
    public void display() {
        System.out.println("Task List: " + title);
        for (Task task : tasks) {
            task.display();
        }
    }
}

// Client
```

```java
public class TaskManagementApp {
    public static void main(String[] args) {
        // Creating simple tasks
        Task simpleTask1 = new SimpleTask("Complete Coding");
        Task simpleTask2 = new SimpleTask("Write Documentation");

        // Creating a task list
        TaskList projectTasks = new TaskList("Project Tasks")
        projectTasks.addTask(simpleTask1);
        projectTasks.addTask(simpleTask2);

        // Nested task list
        TaskList phase1Tasks = new TaskList("Phase 1 Tasks");
        phase1Tasks.addTask(new SimpleTask("Design"));
        phase1Tasks.addTask(new SimpleTask("Implementation"))

        projectTasks.addTask(phase1Tasks);

        // Displaying tasks
        projectTasks.display();
    }
}
```

Output

```
Task List: Project Tasks
Simple Task: Complete Coding
Simple Task: Write Documentation
Task List: Phase 1 Tasks
Simple Task: Design
Simple Task: Implementation
```

Why do we need Composite Design Pattern?

The Composite Design Pattern was created to address specific challenges related to the representation and manipulation of hierarchical structures in a uniform way. Here are some points that highlight the need for the Composite Design Pattern:

1. **Uniform Interface:**

- The Composite Pattern provides a uniform interface for both individual objects and compositions.
- This uniformity simplifies client code, making it more intuitive and reducing the need for conditional statements to differentiate between different types of objects.
- Other design patterns may not offer the same level of consistency in handling individual and composite objects.

2. **Hierarchical Structures:**
    - The primary focus of the Composite Pattern is to deal with hierarchical structures where objects can be composed of other objects.
    - While other patterns address different types of problems, the Composite Pattern specifically targets scenarios involving tree-like structures.

3. **Flexibility and Scalability:**
    - The Composite Pattern allows for dynamic composition of objects, enabling the creation of complex structures.
    - It promotes flexibility and scalability, making it easier to add or remove elements from the hierarchy without modifying the client code.

4. **Common Operations:**
    - By defining common operations at the component level, the Composite Pattern reduces code duplication and promotes a consistent approach to handling both leaf and composite objects.
    - Other design patterns may not provide the same level of support for common operations within hierarchical structures.

5. **Client Simplification:**
    - The Composite Pattern simplifies client code by providing a unified way to interact with individual and composite objects. This simplification is particularly valuable when working with complex structures, such as graphical user interfaces or organizational hierarchies.

When to use Composite Design Pattern?

Composite Pattern should be used when clients need to ignore the difference between compositions of objects and individual objects. If programmers find that they are using multiple objects in the same way, and often have nearly identical code to handle each of them, then composite is a good choice, it is less complex in this situation to treat primitives and composites as homogeneous.

- Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.
- Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.

When not to use Composite Design Pattern?

Composite Design Pattern makes it harder to restrict the type of components of a composite. So it should not be used when you don't want to represent a full or partial hierarchy of objects.

- Composite Design Pattern can make the design overly general.
- It makes harder to restrict the components of a composite.
- Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you.
- Instead you'll have to use run-time checks.

Comment More info Advertise with us

Next Article

Decorator Method Design Pattern in Java

Similar Reads

Java Design Patterns Tutorial

Design patterns in Java refer to structured approaches involving objects and classes that aim to solve recurring design issues within specific contexts. These patterns offer reusable, general solutions to common problems...

8 min read

Creational Software Design Patterns in Java

Factory Method Design Pattern in Java

It is a creational design pattern that talks about the creation of an object. The factory design pattern says to define an interface (A java interface or an abstract class) for creating the object and let the subclasses decid...

6 min read

Builder Method Design Pattern in Java

Method Chaining: In java, Method Chaining is used to invoke multiple methods on the same object which occurs as a single statement. Method-chaining is implemented by a series of methods that return the this...

5 min read

Builder, Fluent Builder, and Faceted Builder Method Design Pattern in Java

Builder Pattern is defined as a creational design pattern that is used to construct a complex object step by step. It separates the construction of an object from its representation, allowing us to create different...

8 min read

Singleton Design Pattern in Java

Singleton Design Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to it. This pattern is particularly useful when exactly one object is needed t...

5 min read

Structural Software Design Patterns in Java

Composite Design Pattern in Java

The Composite Design Pattern is a structural design pattern that lets you compose objects into tree-like structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions o...

8 min read

Decorator Method Design Pattern in Java

A structural design pattern called the Decorator Design Pattern enables the dynamic addition of functionality to specific objects without changing the behavior of other objects in the same class. To wrap concrete...

10 min read

Design Patterns in Java - Iterator Pattern

A design pattern is proved solution for solving the specific problem/task. We need to keep in mind that design patterns are programming language independent for solving the common object-oriented design problems. I...

5 min read

Flyweight Method Design Pattern in Java

A flyweight design pattern or flyweight method is defined as a structural pattern that is used to minimize memory usage or computational expenses by sharing as much as possible with other similar objects. The ke...

9 min read

Behavioural Software Design Patterns in Java

Mediator Design Pattern in Java

The mediator design pattern defines an object that encapsulates how a set of objects interact. The Mediator is a behavioral pattern (like the Observer or the Visitor pattern) because it can change the program's running...

4 min read

Observer Method Design Pattern in Java

Observer Design Pattern is a behavioral design pattern where an object, known as the subject, maintains a list of its dependents, called observers, that are notified of any changes in the subject's state. This pattern is ofte...

11 min read

Strategy Method Design Pattern in Java

Strategy method or Strategy Design Pattern is a behavioral design pattern in Java that defines a family of algorithms, encapsulates each algorithm, and makes them interchangeable. It lets the client algorithm vary...

11 min read

Template Method Design Pattern in Java

Template Design Pattern or Template Method is the behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its...

10 min read

Null Object Design Pattern

The Null Object Design Pattern is a behavioral design pattern that is used to provide a consistent way of handling null or non-existing objects. It is particularly useful in situations where you want to avoid explicit n...

7 min read

Software Design Pattern in other Programming Languages

Java Design Patterns Tutorial

Design patterns in Java refer to structured approaches involving objects and classes that aim to solve recurring design issues within specific contexts. These patterns offer reusable, general solutions to common problems...

8 min read

Python Design Patterns Tutorial

Design patterns in Python are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

JavaScript Design Patterns Tutorial

Design patterns in Javascipt are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Data Structures
Algorithms
DSA for Beginners
Basic DSA Problems
DSA Roadmap
Top 100 DSA Interview Problems
DSA Roadmap by Sandeep Jain
All Cheat Sheets

Data Science With Python
Data Science For Beginner
Machine Learning
ML Maths
Data Visualisation
Pandas
NumPy
NLP
Deep Learning

Web Technologies
HTML
CSS
JavaScript
TypeScript
ReactJS
NextJS
Bootstrap
Web Design

Python Tutorial
Python Programming Examples
Python Projects
Python Tkinter
Web Scraping
OpenCV Tutorial
Python Interview Question
Django

Computer Science
Operating Systems
Computer Network
Database Management System
Software Engineering
Digital Logic Design
Engineering Maths
Software Development
Software Testing

DevOps
Git
Linux
AWS
Docker
Kubernetes
Azure
GCP
DevOps Roadmap

System Design
High Level Design
Low Level Design
UML Diagrams
Interview Guide
Design Patterns
OOAD
System Design Bootcamp
Interview Questions

Inteview Preparation
Competitive Programming
Top DS or Algo for CP
Company-Wise Recruitment Process
Company-Wise Preparation
Aptitude Preparation
Puzzles

School Subjects
Mathematics
Physics
Chemistry
Biology
Social Science
English Grammar
Commerce

GeeksforGeeks Videos
DSA
Python
Java
C++
Web Development
Data Science
CS Subjects

World GK