Composite Design Pattern

Composite pattern is a partitioning design pattern and describes a group of objects that is treated the same way as a single instance of the same type of object. The intent of a composite is to “compose” objects into tree structures to represent part-whole hierarchies. It allows you to have a tree structure and ask each node in the tree structure to perform a task.

- As described by Gof, “Compose objects into tree structure to represent **part-whole hierarchies**. Composite lets client treat individual objects and compositions of objects uniformly”.
- When dealing with Tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, error prone. The solution is an **interface** that allows treating complex and primitive objects uniformly.
- In object-oriented programming, a composite is an object designed as a composition of one-or-more similar objects, all exhibiting similar functionality. This is known as a **“has-a”** relationship between objects.

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 a least common denominator relationship.

The Composite Pattern has four participants:

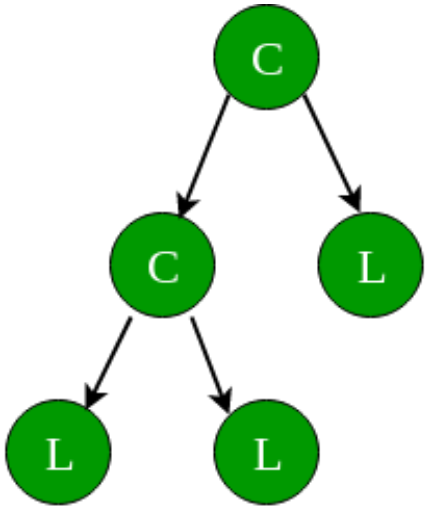
1. **Component –** Component declares the interface for objects in the composition and for accessing and managing its child components. It also implements default behavior for the interface common to all classes as appropriate.
2. **Leaf –** Leaf defines behavior for primitive objects in the composition. It represents leaf objects in the composition.
3. **Composite –** Composite stores child components and implements child related operations in the component interface.
4. **Client –** Client manipulates the objects in the composition through the component interface.

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

Real Life example

In an organization, It have general managers and under general managers, there can be managers and under managers there can be developers. Now you can set a tree structure and ask each node to perform common operation like getSalary().
Composite design pattern treats each node in two ways:
1) **Composite** – Composite means it can have other objects below it.
2) **leaf** – leaf means it has no objects below it.

Tree structure:



Where, C = Composite & L = Leaf

The above figure shows a typical Composite object structure. As you can see, there can be many children to a single parent i.e. Composite, but only one parent per child.

Interface Component.java

```
public interface Employee
{
    public void showEmployeeDetails();
}
```

Leaf.java

```
public class Developer implements Employee
{
    private String name;
    private long empId;
    private String position;
```

```
    public Developer(long empId, String name, String position)
    {
        this.empId = empId;
        this.name = name;
                this.position = position;
    }

    @Override
    public void showEmployeeDetails()
    {
        System.out.println(empId+" " +name+);
    }
}
```

Leaf.java

```
public class Manager implements Employee
{
    private String name;
    private long empId;
        private String position;

    public Manager(long empId, String name, String position)
    {
        this.empId = empId;
        this.name = name;
                this.position = position;
    }

    @Override
    public void showEmployeeDetails()
    {
        System.out.println(empId+" " +name);
    }
}
```

Composite.java

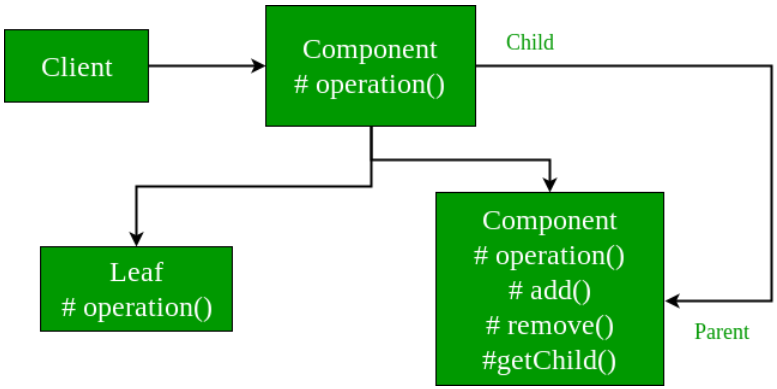
```
import java.util.ArrayList;
import java.util.List;

public class CompanyDirectory implements Employee
{
    private List<Employee> employeeList = new ArrayList<Employee>();

    @Override
    public void showEmployeeDetails()
    {
        for(Employee emp:employeeList)
        {
            emp.showEmployeeDetails();
        }
    }

    public void addEmployee(Employee emp)
    {
        employeeList.add(emp);
    }

    public void removeEmployee(Employee emp)
    {
        employeeList.remove(emp);
    }
}
```

Client.java

```
public class Company
{
    public static void main (String[] args)
    {
        Developer dev1 = new Developer(100, "Lokesh Sharma", "Pro Developer");
        Developer dev2 = new Developer(101, "Vinay Sharma", "Developer");
        CompanyDirectory engDirectory = new CompanyDirectory();
        engDirectory.addEmployee(dev1);
        engDirectory.addEmployee(dev2);

        Manager man1 = new Manager(200, "Kushagra Garg", "SEO Manager");
```

```
        Manager man2 = new Manager(201, "Vikram Sharma ", "Kushagra's Manager");

        CompanyDirectory accDirectory = new CompanyDirectory();
        accDirectory.addEmployee(man1);
        accDirectory.addEmployee(man2);

        CompanyDirectory directory = new CompanyDirectory();
        directory.addEmployee(engDirectory);
        directory.addEmployee(accDirectory);
        directory.showEmployeeDetails();
    }
}
```

UML Diagram for the Composite Design Pattern :



Full Running Code for the above example :

```
// A Java program to demonstrate working of
// Composite Design Pattern with example
// of a company with different
//  employee details
```

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


// A common interface for all employee
interface Employee
{
    public void showEmployeeDetails();
}

class Developer implements Employee
{
    private String name;
    private long empId;
    private String position;

    public Developer(long empId, String name, String position)
    {
        // Assign the Employee id,
        // name and the position
        this.empId = empId;
        this.name = name;
        this.position = position;
    }

    @Override
    public void showEmployeeDetails()
    {
        System.out.println(empId+" " +name+ " " + position );
    }
}

class Manager implements Employee
{
    private String name;
    private long empId;
    private String position;

    public Manager(long empId, String name, String position)
    {
        this.empId = empId;
```

```
            this.name = name;
            this.position = position;
        }

        @Override
        public void showEmployeeDetails()
        {
            System.out.println(empId+" " +name+ " " + position );
        }
    }


    // Class used to get Employee List
    // and do the opertions like
    // add or remove Employee

    class CompanyDirectory implements Employee
    {
        private List<Employee> employeeList = new ArrayList<Employee>();

        @Override
        public void showEmployeeDetails()
        {
            for(Employee emp:employeeList)
            {
                emp.showEmployeeDetails();
            }
        }

        public void addEmployee(Employee emp)
        {
            employeeList.add(emp);
        }

        public void removeEmployee(Employee emp)
        {
            employeeList.remove(emp);
        }
    }

    // Driver class
    public class Company
```

```
{
    public static void main (String[] args)
    {
        Developer dev1 = new Developer(100, "Lokesh Sharma", "Pro Developer");
        Developer dev2 = new Developer(101, "Vinay Sharma", "Developer");
        CompanyDirectory engDirectory = new CompanyDirectory();
        engDirectory.addEmployee(dev1);
        engDirectory.addEmployee(dev2);

        Manager man1 = new Manager(200, "Kushagra Garg", "SEO Manager");
        Manager man2 = new Manager(201, "Vikram Sharma ", "Kushagra's Manager");

        CompanyDirectory accDirectory = new CompanyDirectory();
        accDirectory.addEmployee(man1);
        accDirectory.addEmployee(man2);

        CompanyDirectory directory = new CompanyDirectory();
        directory.addEmployee(engDirectory);
        directory.addEmployee(accDirectory);
        directory.showEmployeeDetails();
    }
}
```

Output :

```
100 Lokesh Sharma Pro Developer
101 Vinay Sharma Developer
200 Kushagra Garg SEO Manager
201 Vikram Sharma  Kushagra's Manager
```

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 homogeneou

1. Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.
2. 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?

1. 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.
2. 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.

This article is contributed by **Saket Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Recommended Posts:

Design Patterns | Set 1 (Introduction)

Design Patterns | Set 2 (Factory Method)

Command Pattern

Observer Pattern | Set 1 (Introduction)

Observer Pattern | Set 2 (Implementation)

Singleton Design Pattern | Implementation

Decorator Pattern | Set 1 (Background)

The Decorator Pattern | Set 2 (Introduction and Design)

Decorator Pattern | Set 3 (Coding the Design)

Strategy Pattern | Set 1 (Introduction)

Strategy Pattern | Set 2 (Implementation)

Adapter Pattern

Iterator Pattern

Curiously recurring template pattern (CRTP)

Flyweight Design Pattern

Article Tags : Design Pattern

Be the First to upvote.

To-do Done

3

Based on **1** vote(s)

Feedback/ Suggest Improvement | Add Notes | Improve Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments

Share this post!