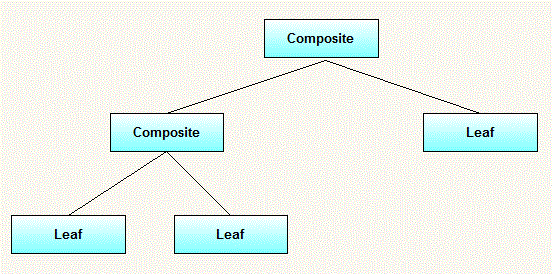
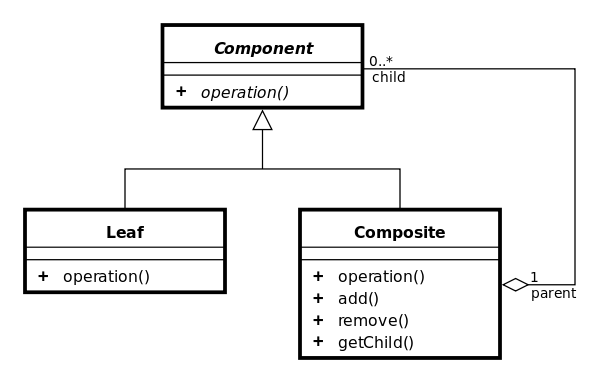
Composite Structural Design Pattern-2024

**GOF : Compose objects into tree structure to represent part-whole hierarchies. Composite lets clients treat individual objects and composition of objects uniformly**.



A pattern to deal with hierarchical, heterogeneous data/objects in a uniform manner.

**The intent behind composite design pattern is to create a common base, for individual objects and collection of objects** which can define operations on both types of objects.

Both **Leaf** and **Composite** are concrete implementations of Component, the only difference is Composite can contain child component ie. Leaf.

It is an attempt to bring homogeneity in heterogeneous objects.

## **Advantages of the Composite Design Pattern**

1. **Uniformity**: This simplifies client code, as it doesn’t need to distinguish between leaf and composite**.**
2. **Hierarchical Structures**: It is particularly useful when dealing with hierarchical structures, organizational hierarchies.
3. **Complex Operations:** Complex operations can be applied to the entire hierarchy.
4. **Encapsulation**: The **Composite** pattern promotes **Encapsulation** by encapsulating the individual objects (**leaves**) and their compositions (**composites**) within a common interface.
5. **Maintenance and Refactoring**: It simplifies maintenance and refactoring efforts. We can modify the structure of individual components or the entire hierarchy without affecting the **Client**’s code.
6. **Clear Abstraction**: The pattern provides a clear abstraction for building complex structures, making the code more understandable and maintainable.

## **Disadvantages of the Composite Design Pattern**

1. **Performance Overhead/ Memory Consumption**: Managing a hierarchy of objects, especially large ones, may introduce performance overhead due to the recursive nature of operations on composite objects. Storing a hierarchy of objects can consume memory, especially if the hierarchy is deep or if there are many objects involved. This can be a concern in resource-constrained environments.
2. **Type Safety**: The pattern uses a common interface or abstract class to represent both **leaf** and **composite** This can lead to a lack of type safety, as **Clients** may attempt operations that are only meaningful for one type of object. Runtime checks may be needed to ensure type safety.
   1. **Limited Leaf Customization**: If individual leaf objects have unique properties or behaviors, the Composite pattern may not be the best choice, as it enforces a uniform interface across all components. In such cases, you may need to resort to other patterns or adaptations.

## **When should we use the Composite Design Pattern?**

A Composite is a pattern that is useful anytime you may need to selectively treat a group of objects **that are part of a hierarchy** as "the same" when they are in fact different.

1. **Hierarchical Structures**: Use the **Composite** pattern when we need to represent hierarchical structures like trees, directories, menus, organizational hierarchies, or any structure where objects can be composed of other objects.
2. **Recursive Operations**: When we have to perform operations on a hierarchy of objects in a recursive manner, the **Composite** pattern is a suitable choice. It simplifies the recursive traversal and application of operations.
3. **Uniformity**: If we want to provide a consistent interface for both individual objects and composite objects, use the **Composite** This uniformity simplifies client code, as it doesn’t need to distinguish between the two.

## **Real-world examples of Composite Design Pattern**

1. **Graphic Design Software**: In graphic design applications like Adobe Photoshop or Illustrator, graphical elements like shapes, lines, and text can be composed into complex designs. The **Composite** pattern is used to represent these elements and the ability to group them into more complex structures.
2. **File Systems**: Operating systems often use the **Composite** pattern to represent files and directories. Directories can contain files or other directories, creating a hierarchical structure that can be navigated and manipulated uniformly.
3. **User Interface (UI) Frameworks**: UI frameworks like Windows Presentation Foundation (WPF) and JavaFX use the **Composite** pattern extensively. UI elements, such as buttons, labels, and panels, can be composed into complex layouts and containers.
4. **Organizational Structures**: Enterprise software often models organizational structures, including departments, teams, and employees. The Composite pattern is used to represent the hierarchy of organizational units and employees within them.
5. **Menu Systems**: In graphical user interfaces, menu systems can be implemented using the **Composite** Menus can contain individual menu items or sub-menus, allowing for nested structures.
6. **Billing and Invoicing Systems**: Billing and invoicing systems may have complex billing structures. The Composite pattern can be used to represent invoices with line items, where line items can be individual charges or sub-invoices.
7. **Financial Modeling**: Financial applications may represent complex financial instruments or portfolios. The **Composite** pattern can be applied to represent these structures, allowing for consistent operations on individual financial instruments and portfolios.

## Composite Design Pattern Java Implementation

**public** **abstract** **class** Employee {

**public** **abstract** **void** printEmpDetails();

}

**public** **class** Developer **extends** Employee {

**private** String name;

**public** Developer(String name) {

**super**();

**this**.name = name;

}

@Override

**public** **void** printEmpDetails() {

System.***out***.println("Name:" + name + " Type:Developer");

}

}

**public** **class** Manager **extends** Employee {

**private** String name;

**public** Manager(String name) {

**super**();

**this**.name = name;

}

@Override

**public** **void** printEmpDetails() {

System.***out***.println("Name:" + name + " Type:Manager");

}

}

**public** **class** Organization **extends** Employee {

**private** List<Employee> empList;

**private** String name;

**public** Organization(String name) {

**this**.name = name;

empList = **new** ArrayList<>();

}

**public** **void** addEmployees(Employee emp) {

empList.add(emp);

}

@Override

**public** **void** printEmpDetails() {

**for**(Employee emp : empList)

emp.printEmpDetails();

}

}

**public** **class** SeniorDeveloper **extends** Developer {

**protected** SeniorDeveloper(String name) {

**super**(name);

}

}

**public** **class** TestOrg {

**public** **static** **void** main(String[] args) {

Organization org = **new** Organization("DDLAB Inc");

Employee e1 = **new** Developer("Shyam");

Employee e2 = **new** Developer("Ram");

Employee e3 = **new** Manager("Hari");

Employee e4 = **new** SeniorDeveloper("Hari11");

org.addEmployees(e1);

org.addEmployees(e2);

org.addEmployees(e3);

org.addEmployees(e4);

org.printEmpDetails();

}

}

Another Example is given below.

Example : Directory and File, both have the ls() method. Java code is given below.

**class** File **implements** AbstractFile {

**private** String name;

**public** File(String name) {

**this**.name = name;

}

**public** **void** ls() {

System.***out***.println(name);

}

}

**import** java.util.ArrayList;

**public** **class** Directory **implements** AbstractFile {

**private** String name;

**private** ArrayList<AbstractFile> files = **new** ArrayList<AbstractFile>();

**public** Directory(String name) {

**this**.name = name;

}

**public** **void** add(AbstractFile f) {

files.add(f);

}

**public** **void** ls() {

System.***out***.println(name);

**for** (AbstractFile file : files) {

file.ls();

}

}

}

**public** **interface** AbstractFile {

**public** **void** ls();

}

**public** **class** Test { 🡸 Test Program

**public** **static** **void** main(String[] args) {

Directory dirOne = **new** Directory("dir111");

Directory dirTwo = **new** Directory("dir222");

File a = **new** File("a");

File b = **new** File("b");

File c = **new** File("c");

File d = **new** File("d");

dirOne.add(a);

dirOne.add(dirTwo);

dirOne.add(b);

dirTwo.add(c);

dirTwo.add(d);

dirOne.ls();

}

}

Using Abstract Class

**public** **abstract** **class** AbstractFile {

**public** **abstract** **void** open();

}

**public** **class** File **extends** AbstractFile {

**private** String name;

**public** File(String name) {

**this**.name = name;

}

@Override

**public** **void** open() {

System.***out***.println("File "+name+" has been opened ...");

}

}

**public** **class** Directory **extends** AbstractFile {

**private** String name;

**private** List<AbstractFile> files = **new** ArrayList<AbstractFile>();

**public** Directory(String name) {

**this**.name = name;

}

**public** **void** add(AbstractFile f) {

files.add(f);

}

@Override

**public** **void** open() {

System.***out***.println("Opening files in directory: "+name);

**for** (AbstractFile file : files) {

file.open();

}

}

}

**public** **class** Check {

**public** **static** **void** main(String[] args) {

Directory dirOne = **new** Directory("dir111");

Directory dirTwo = **new** Directory("dir222");

File a = **new** File("a");

File b = **new** File("b");

File c = **new** File("c");

File d = **new** File("d");

dirOne.add(a);

dirOne.add(dirTwo);

dirOne.add(b);

dirTwo.add(c);

dirTwo.add(d);

dirOne.open();

}

}

References: <https://programmingline.com/software-design-patterns/composite-design-pattern#:~:text=Composite%20Design%20Pattern%20Implementation%20Use%20Case,-If%20we%20are&text=Example%3A%20Developer%2C%20Manager%2C%20etc,treated%20as%20a%20composite%20class>.

<https://www.pentalog.com/blog/design-patterns/composite-design-pattern/#:~:text=Real%2Dworld%20examples%20of%20Composite%20Design%20Pattern&text=The%20Composite%20pattern%20is%20used,to%20represent%20files%20and%20directories>.

<https://stackoverflow.com/questions/5334353/when-should-i-use-composite-design-pattern>

Read Later – For Deeper Structure

**public** **abstract** **class** Employee {

**public** **abstract** **void** showDetails();

}

**public** **class** Developer **extends** Employee {

**private** String name;

**public** Developer(String name) {

**this**.name = name;

}

@Override

**public** **void** showDetails() {

System.***out***.println("Developer Name: "+name);

}

}

**public** **class** Manager **extends** Employee {

**private** List<Employee> empList = **new** ArrayList<>();

**private** String name;

**public** Manager(String name) {

**this**.name = name;

}

**public** **void** addEmployee(Employee emp) {

empList.add(emp);

}

@Override

**public** **void** showDetails() {

System.***out***.println("Emp Reporting to Manager: "+**this**.name);

**for**(Employee emp : empList) {

emp.showDetails();

}

}

}

**public** **class** VP **extends** Employee {

**private** List<Employee> managerList = **new** ArrayList<>();

**private** String name;

**public** VP(String name) {

**this**.name = name;

}

**public** **void** addEmployee(Employee emp) {

managerList.add(emp);

}

@Override

**public** **void** showDetails() {

System.***out***.println("Emp Reporting to VP: "+**this**.name);

**for**(Employee emp : managerList) {

emp.showDetails();

}

}

}

**public** **class** Test11 {

**public** **static** **void** main(String[] args) {

Employee e1 = **new** Developer("Shyam");

Employee e2 = **new** Developer("Hari");

Manager m1 = **new** Manager("Jaga");

m1.addEmployee(e1);

m1.addEmployee(e2);

Manager m2 = **new** Manager("Gaja");

m2.addEmployee(e1);

m2.addEmployee(e2);

VP vp = **new** VP("Somenath");

vp.addEmployee(m1);

vp.addEmployee(m2);

vp.showDetails();

}

}