Composite Design Pattern

The key idea behind the **Composite** Design Pattern is to create a common interface or base class that both **Leaf** objects (individual objects) and **Composite** objects (collections of objects) implement.

This common interface, or **Component**, defines operations that may be performed on both types of objects, allowing clients to treat them interchangeably.

A **Leaf** is a concrete implementation of the **Component** interface. It represents individual objects that do not have any children. In the context of a tree structure, leaves are the end nodes.

A **Composite** is also a concrete implementation of the **Component** interface, but it can hold child components. It can be used to represent complex objects that are composed of smaller objects or other **Composites**.

Example:

The **OrganizationalUnit**class is a container that can comprise any number of **Employees**, and other **OrganizationalUnits**. An **OrganizationalUnit** and a simple **Employee** implements the same methods.

## Advantages of the Composite Design Pattern

1. **Uniformity**: One of the primary advantages is that it allows us to treat both individual objects and compositions of objects uniformly. 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, such as trees or organizational hierarchies.
3. **Complex Operations**: It enables us to define complex operations that can be applied to the entire hierarchy. These operations can be implemented recursively, making it easier to work with deeply nested structures.
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. **Reuse**: Components can be reused in different parts of the application, which promotes code reuse and modularity
7. **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. **Complexity**: While the pattern simplifies client code, it can introduce complexity in the implementation of **composite** objects (composites) that need to manage a list of child components. This complexity can make the code harder to maintain.
2. **Performance Overhead**: Managing a hierarchy of objects, especially large ones, may introduce performance overhead due to the recursive nature of operations on composite objects. Depending on the use case, this overhead may not be acceptable.
3. **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.
4. **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.
5. **Memory Consumption**: 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.

## When should we use the Composite Design Pattern?

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.
4. **Flexibility**: When we want **Clients** to be able to work with objects at various levels of the hierarchy, from individual **leaves** to entire **compositions**, the **Composite** pattern provides this flexibility.

## 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.

Ref: 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.

## Composite Design Pattern Java Implementation

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

{

**public** **abstract** **void** printEmployeeDetails();

}

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

{

**protected** String name;

**protected** **long** empID;

**protected** String designation;

**public** Manager(String name, **long** empID)

    {

**this**.name = name;

**this**.empID = empID;

    }

    @Override

**public** **void** printEmployeeDetails()

    {

        System.out.println("Name:" + name + " empID:" + empID + " Designation:Manager");

    }

}

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

{

**protected** String name;

**protected** **long** empID;

**protected** String designation;

**public** Developer(String name, **long** empID)

    {

**this**.name = name;

**this**.empID = empID;

    }

    @Override

**public** **void** printEmployeeDetails()

    {

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

    }

**public** **class** CompanyDirectory **extends** Employee

{

**private** List employeeList;

**public** CompanyDirectory()

    {

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

    }

    @Override

**public** **void** printEmployeeDetails()

    {

**for** (Employee employee : employeeList)

        {

            employee.printEmployeeDetails();

        }

    }

**public** **void** addEmployee(Employee employee)

    {

        employeeList.add(employee);

    }

}

**public** **class** CompanyDetails **extends** CompanyDirectory

{

**private** String companyName;

**public** CompanyDetails(String companyName)

    {

**this**.companyName=companyName;

    }

**public** **void** showCompanyName()

    {

        System.out.println(**this**.companyName);

    }

}

**public** **class** Company

{

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

    {

        Employee d1 = **new** Developer("Lenin", 10);

        Employee d2 = **new** Developer("Chenchu", 11);

        Employee d3 = **new** Developer("Raja", 12);

        Employee m1 = **new** Manager("Rakesh", 1);

        CompanyDetails companyDetails = **new** CompanyDetails("XYZ");

        companyDetails.addEmployee(m1);

        companyDetails.addEmployee(d1);

        companyDetails.addEmployee(d2);

        companyDetails.addEmployee(d3);

        companyDetails.printEmployeeDetails();

    }

}

Ref: <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://stackoverflow.com/questions/5334353/when-should-i-use-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. Typically the examples used talk in terms of treating leaves and nodes the same, but the pattern can also be extended to heterogeneous lists.

The Composite allows you to treat hierarchical, heterogeneous data in a uniform manner without having to cast objects, evaluate their type, and perform conditionals to see if they contain other objects.

Common Interface for individual Objects and Composite Objects

interface Data{

public void doubleClick();

}

Individual Object Implementation

class File implements Data {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public void doubleClick() {

System.out.println(this.getName()+" file is Opened in a Program ");

}

}

Composite Implementation

class Folder implements Data {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

private List<Data> folder = new ArrayList<Data>();

@Override

public void doubleClick() {

System.out.println(this.getName() + " folder is Opened");

for(Data data : folder) {

data.doubleClick();

}

}

public void add(Data data) {

folder.add(data);

}

public void remove(Data data) {

folder.remove(data);

}

}

Client Program

public class CompositePattern {

public static void main(String[] args) {

Folder f1 = new Folder();f1.setName("Folder 1");

Folder f2 = new Folder();f2.setName("Folder 2");

Folder f3 = new Folder();f3.setName("Folder 3");

File file1 = new File();file1.setName("File 1");

File file2 = new File();file2.setName("File 2");

File file3 = new File();file3.setName("File 3");

File file4 = new File();file4.setName("File 4");

f1.add(file1);

f2.add(file2);

f3.add(f2);

f3.add(file3);

f3.add(file4);

f1.doubleClick();f2.doubleClick();f3.doubleClick();

}

}