HowToDoInJava

Composite Design Pattern

Last Updated: August 28, 2021 By: Lokesh Gupta Structural Patterns Design Patterns

Composite design pattern is a **structural pattern** which modifies the structure of an object. This pattern is most suitable in cases where you need to work with **objects which form a tree like hierarchy**. In that tree, each node/object (except root node) is either composite or leaf node. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

Table of Contents

When to use composite design pattern

> Composite design pattern compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- When application has hierarchical structure and needs generic functionality across the structure.

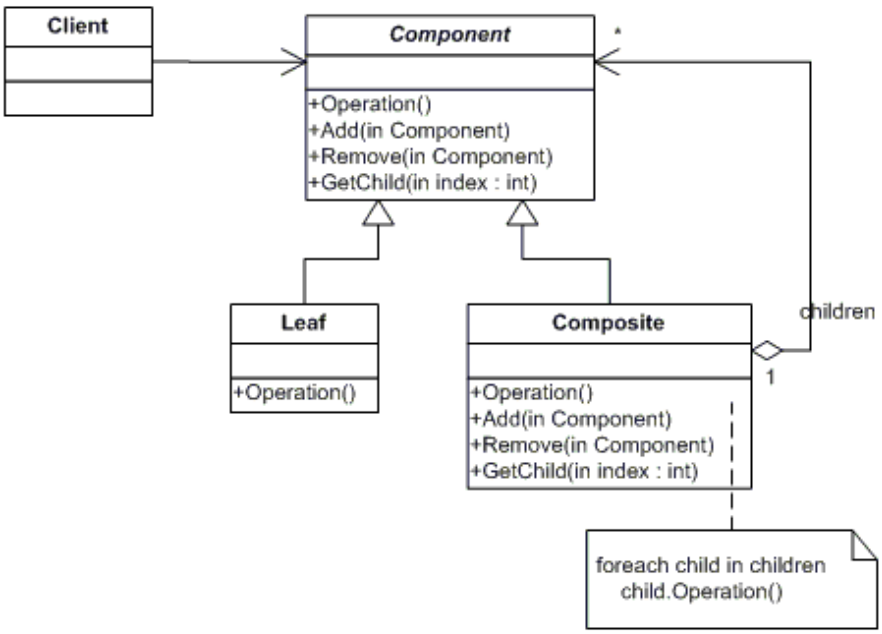
- When application needs to aggregate data across a hierarchy.
- When application wants to treat composite and individual objects uniformly.

Real life example usage of composite design pattern may be:

1. Building consolidate view of a customer's account in bank (i.e. customer's portfolio)
2. Building general ledgers
3. Computer/network monitoring applications
4. Retail and inventory applications
5. Directory structure in file system implementations
6. Menu items in GUI screens

Participants in composite design pattern

Below are the participants in any composite pattern based solution.



Composite design pattern

Where the classes and objects participating in this pattern are:

1. **Component**

    - declares the interface for objects in the composition.
    - implements default behavior for the interface common to all classes, as appropriate.
    - declares an interface for accessing and managing its child components.

2. **Leaf**

    - represents leaf objects in the composition. A leaf has no children.
    - defines behavior for primitive objects in the composition.

3. **Composite**

    - defines behavior for components having children.
    - stores child components.
    - implements child-related operations in the `Component` interface.

4. **Client**

    - manipulates objects in the composition through the `Component` interface.

In above diagram, client uses the `Component` interface to interact with objects in composite hierarchy. Inside hierarchy, if an object is composite then it passes the request to leaf nodes. If object is leaf node, request is handled immediately.

Composite leaves also have a choice of modifying the request/response either before or after the request is handled by leaf node.

Sample problem to solve

Let's suppose we are building a financial application. We have customers with multiple bank accounts. We are asked to prepare a design which can be useful to generate the

customer's consolidated account view which is able to show **customer's total account balance as well as consolidated account statement** after merging all the account statements. So, application should be able to generate:

1) Customer's total account balance from all accounts
2) Consolidated account statement

Solution using composite design pattern

As here we are dealing with account objects which form a tree-like structure in which we will traverse and do some operations on account objects only, so we can apply composite design pattern.

Let's see the participating classes:

Component.java

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

public abstract class Component
{
    AccountStatement accStatement;

    protected List<Component> list = new ArrayList<>();

    public abstract float getBalance();

    public abstract AccountStatement getStatement();

    public void add(Component g) {
        list.add(g);
    }

    public void remove(Component g) {
        list.remove(g);
    }

    public Component getChild(int i) {
        return (Component) list.get(i);
    }
```

```
}
```

CompositeAccount.java

```
public class CompositeAccount extends Component
{
    private float totalBalance;
    private AccountStatement compositeStmt, individualStmt;

    public float getBalance() {
        totalBalance = 0;
        for (Component f : list) {
            totalBalance = totalBalance + f.getBalance();
        }
        return totalBalance;
    }

    public AccountStatement getStatement() {
        for (Component f : list) {
            individualStmt = f.getStatement();
            compositeStmt.merge(individualStmt);
        }
        return compositeStmt;
    }
}
```

AccountStatement.java

```
public class AccountStatement
{
    public void merge(AccountStatement g)
    {
        //Use this function to merge all account statements
    }
}
```

DepositAccount.java

```
public class DepositAccount extends Component
{
    private String accountNo;
    private float accountBalance;
```

```
    private AccountStatement currentStmt;

    public DepositAccount(String accountNo, float accountBalance) {
        super();
        this.accountNo = accountNo;
        this.accountBalance = accountBalance;
    }

    public String getAccountNo() {
        return accountNo;
    }

    public float getBalance() {
        return accountBalance;
    }

    public AccountStatement getStatement() {
        return currentStmt;
    }
}
```

SavingsAccount.java

```
public class SavingsAccount extends Component
{
    private String accountNo;
    private float accountBalance;

    private AccountStatement currentStmt;

    public SavingsAccount(String accountNo, float accountBalance) {
        super();
        this.accountNo = accountNo;
        this.accountBalance = accountBalance;
    }

    public String getAccountNo() {
        return accountNo;
    }

    public float getBalance() {
        return accountBalance;
    }

    public AccountStatement getStatement() {
        return currentStmt;
    }
}
```

Client.java

```
public class Client
{
    public static void main(String[] args)
    {
        // Creating a component tree
        Component component = new CompositeAccount();

        // Adding all accounts of a customer to component
        component.add(new DepositAccount("DA001", 100));
        component.add(new DepositAccount("DA002", 150));
        component.add(new SavingsAccount("SA001", 200));

        // getting composite balance for the customer
        float totalBalance = component.getBalance();
        System.out.println("Total Balance : " + totalBalance);

        AccountStatement mergedStatement = component.getStatement();
        //System.out.println("Merged Statement : " + mergedStatement);
    }
}

Output:

Total Balance : 450.0
```

Final notes

1. The composite pattern defines class hierarchies consisting of individual objects and composite objects.
2. Clients treat primitive and composite objects uniformly through a component interface which makes client code simple.
3. Adding new components can be easy and client code does not need to be changed since client deals with the new components through the component interface.
4. Composite hierarchy can be traversed with Iterator design pattern.
5. Visitor design pattern can apply an operation over a Composite.

6. Flyweight design pattern is often combined with Composite to implement shared leaf nodes.

Happy Learning !!

Recommended Reading:

1. Decorator Design Pattern in Java
2. Adapter Design Pattern in Java
3. Bridge Design Pattern
4. Facade Design Pattern
5. Flyweight Design Pattern
6. Proxy Design Pattern
7. Prototype design pattern in Java
8. Strategy Design Pattern
9. Memento Design Pattern
0. Observer Design Pattern

Join 7000+ Awesome Developers

Get the latest updates from industry, awesome resources, blog updates and much more.

Email Address

Subscribe

** We do not spam !!*

2 thoughts on “Composite Design Pattern”

Yogesh

July 23, 2019 at 12:56 pm

The description for composite pattern and example code given, does not really match as per me. User having multiple accounts it typical one to many relationship & not a tree structure per say. Directory structure in file system implementations or menu in websites would have been better example as you've earlier suggested.

Reply

srikanth

July 3, 2019 at 11:45 am

please , send a example on both flyweight and composite design patterns with same Account example .

Reply

Leave a Comment

Name *

Email *

Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial