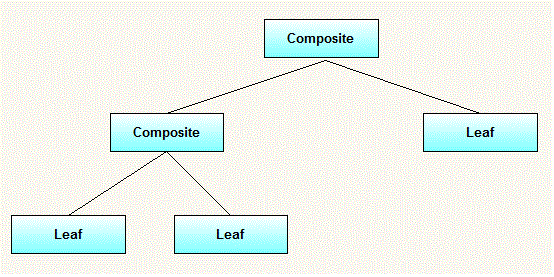
Composite design patten allows you to have a tree structure and ask each node in the tree structure to perform a task.You can take real life example of a organization.It have general managers and under general managers, there can be managers and under managers there can be developers.Now you can set a tree structure and ask each node to perform common operation like getSalary().

As described by Gof:

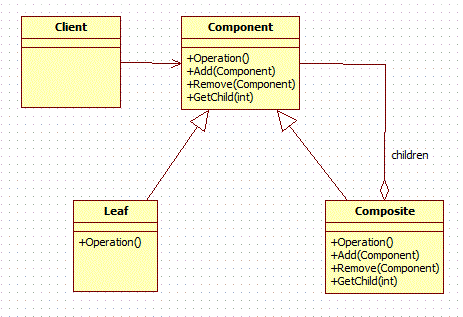
"Compose objects into tree structure to represent part-whole hierarchies.Composite lets client treat individual objects and compositions of objects uniformly".

Composite design pattern treats each node in two ways-**Composite** or **leaf**.Composite means it can have other objects below it.leaf means it has no objects below it.

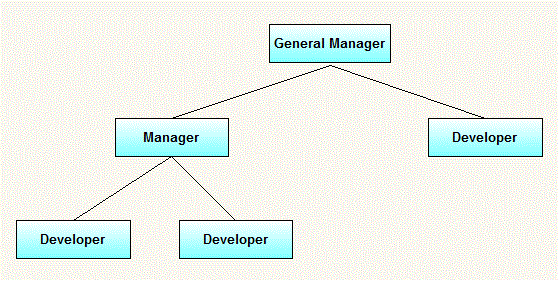


### **When to use it:**

* you want to represent part-whole hierachies of objects.
* you want client to be able to ignore difference between compositions of objects and individual objects.Clients will treat all objects in the composite structure uniformly.



* **Component**
  + declares interface for objects in composition.
  + implements deafault behaviour for the interface common to all classes as appropriate.
  + declares an interface for accessing and managing its child components.
* **Leaf**
  + represents leaf objects in the composition.A leaf has no children.
  + defines behaviour for primitive objects in the composition.
* **Composite**
  + defines behaviour for components having children.
  + stores child components.
  + implements child related operations in the component interface.
* **Client**
  + manipulates objects in the composition through the component interface.



import java.util.List;

import java.util.ArrayList;

import java.util.Iterator;

interface Employee{

public void add(Employee e);

public void remove(Employee e);

public Employee getChild(int i);

public String getName();

public double getSalary();

public void print();

}

class Manager implements Employee{

private String name;

private double salary;

public Manager(String name, double salary) {

this.name = name;

this.salary = salary;

}

List<Employee> employees = new ArrayList<Employee>();

public void add(Employee employee) {

employees.add(employee);

}

public Employee getChild(int i) {

return employees.get(i);

}

public void print() {

System.out.println("-------------------");

System.out.println("name = "+getName());

System.out.println("Salary = "+getSalary());

System.out.println("-------------------");

Iterator<Employee> employeeIterator = employees.iterator();

while(employeeIterator.hasNext()) {

Employee employee = employeeIterator.next();

employee.print();

}

}

public void remove(Employee employee) {

employees.remove(employee);

}

@Override

public String getName() {

// TODO Auto-generated method stub

return name;

}

@Override

public double getSalary() {

// TODO Auto-generated method stub

return salary;

}

}

class Developer implements Employee{

private String name;

private double salary;

public Developer(String name, double salary) {

this.name = name;

this.salary = salary;

}

public void add(Employee employee) {

//this is leaf node so this method is not applicable to this class

}

public Employee getChild(int i) {

//this is leaf node so this method is not applicable to this class

return null;

}

@Override

public String getName() {

// TODO Auto-generated method stub

return name;

}

@Override

public double getSalary() {

// TODO Auto-generated method stub

return salary;

}

public void print() {

System.out.println("----------------------");

System.out.println("Name = "+getName());

System.out.println("salary = "+getSalary());

System.out.println("----------------------");

}

public void remove(Employee employee) {

//this is leaf node so this method is not applicable to this class

}

}

public class TestCompositePattern {

public static void main(String[] args) {

Employee emp1 = new Developer("john",10000);

Employee emp2 = new Developer("David",15000);

Employee manager1 = new Manager("Daniel",25000);

manager1.add(emp1);

manager1.add(emp2);

Employee emp3 = new Developer("Michael",20000);

Manager generalManager = new Manager("Mark", 50000);

generalManager.add(emp3);

generalManager.add(manager1);

generalManager.print();

}

}