Composite Design Pattern

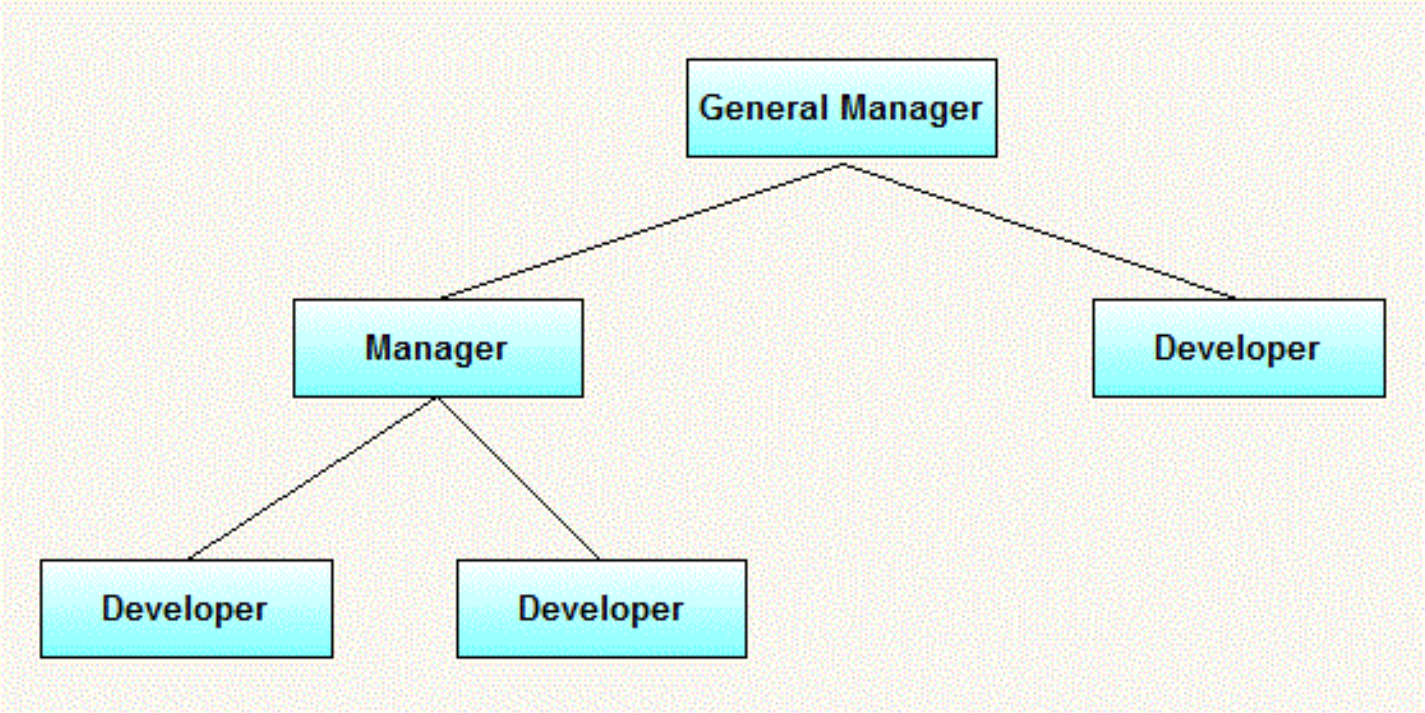
Pantis Vlad
30443

Introduction

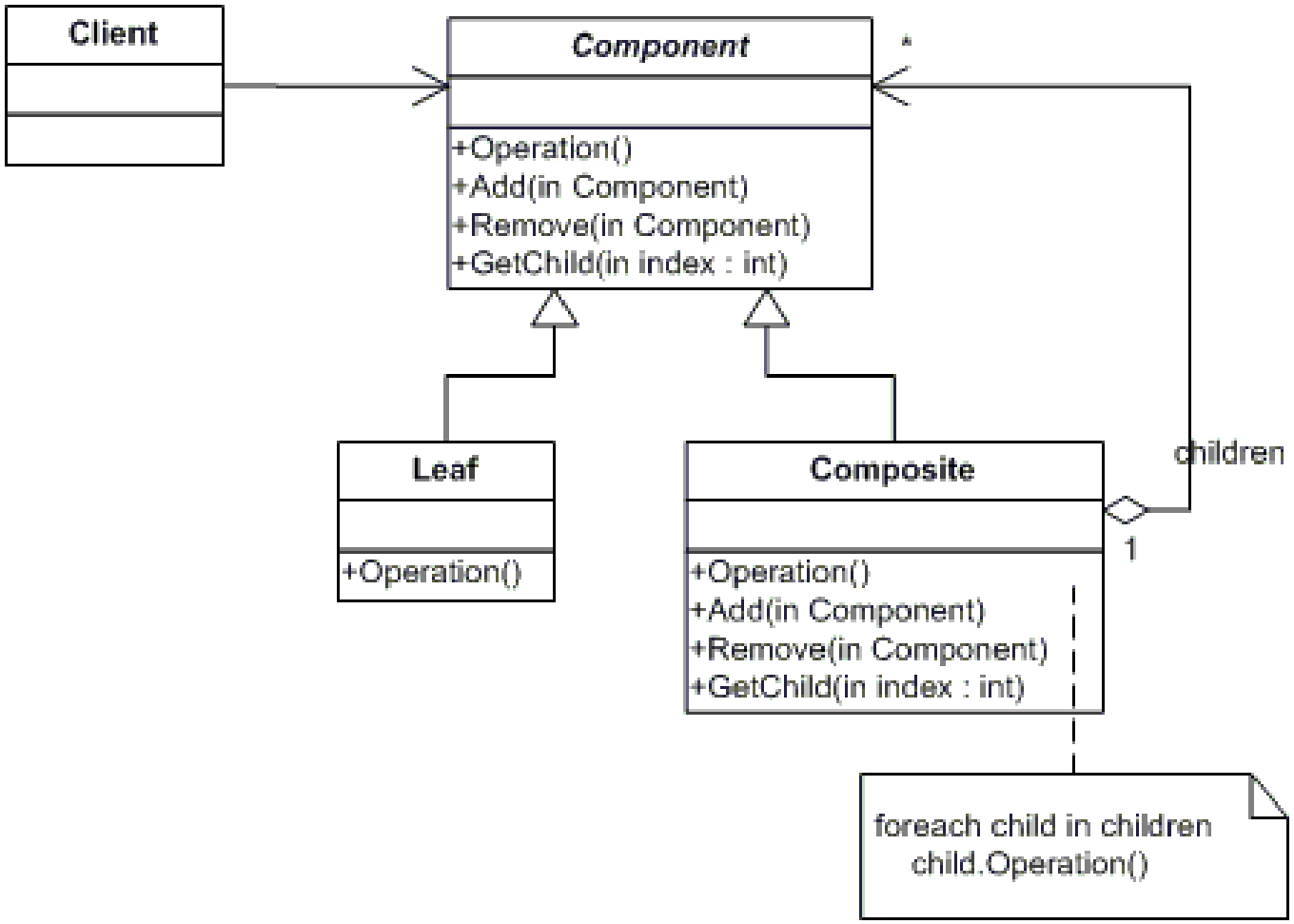
- Composite pattern is a structural pattern, used where we need to treat a group of objects in a similar way as a single object.

Usage

- when you want to represent a hierarchy of objects
- when you want to be able to ignore the difference between compositions of objects and individual objects


General Manager
Manager
Developer
Developer
Developer

Structure


Client
Component
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
*
children
1
Leaf
+Operation()
Composite
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
foreach child in children
child.Operation()

- **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

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

- **Composite**
  - defines behavior 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.

Example

```
public 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 String getName() {
            return name;
    }

    public double getSalary() {
            return salary;
    }

    public void remove(Employee employee) {
            employees.remove(employee);
    }
}
```

```
public 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;
}
public String getName() {
        return name;
}

public double getSalary() {
        return salary;
}

public void remove(Employee employee) {
        //this is leaf node so this method is not applicable to this class.
 }
}
```

Advantages and Disadvantages

- Advantages
  - Simplifies the client
- Disadvantages
  - Once tree structure is defined, composite design makes the tree overly general
  - Leaf class have to create some methods which has to empty