Composite Pattern

Michal Moravik, SD20w2

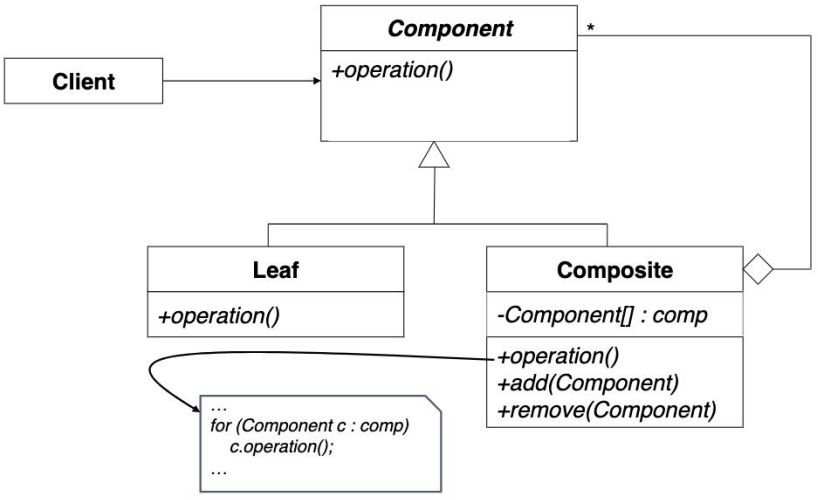
Name: Composite Pattern

Category: Structural

Intent: Lets us compose objects into three structures and then work with the structures as if they were individual objects

Motivation: Imagine that we end up with a model which can be represented as a tree. For example, in a box you can have a new speaker and a new iPhone box which includes charger, stickers, and iPhone itself. Then this box can be represented as a tree. In order to calculate the total price, you would need to go through all items inside of the box. Instead of doing this, you can add a common interface which declares a method for calculating the total price.

UML:



- Leaf in the UML represents the basic element which does not have sub-elements.
- Component is an interface describing operations common to both simple and complex elements of the tree.
- Composite is an element that has sub-elements. It does not not concrete classes of its children (e.g. items).

Implementation:

```java
public interface IEmployee {
    public void showEmployeeDetails();
}
```

```java
public class Manager implements IEmployee {
    private String name;
    private String position;

    public Manager(String name, String position)
    {
        this.name = name;
        this.position = position;
    }

    @Override
    public void showEmployeeDetails() { System.out.println(name + " is a " + position); }
}
```

```java
public class Developer implements IEmployee {
    private String name;
    private String position;

    public Developer(String name, String position)
    {
        this.name = name;
        this.position = position;
    }

    @Override
    public void showEmployeeDetails() { System.out.println(name + " is a " + position); }
}
```

```java
public class CompanyDepartment implements IEmployee {
    private List<IEmployee> employeeList = new ArrayList<IEmployee>();

    @Override
    public void showEmployeeDetails()
    {
        for(IEmployee emp:employeeList)
        {
            emp.showEmployeeDetails();
        }
    }

    public void addEmployee(IEmployee emp)
    {
        employeeList.add(emp);
    }
}
```

```java
public class Client {
    public static void main(String[] args) {
        IEmployee dev1 = new Developer("Michal M", "Tech lead");
        IEmployee dev2 = new Developer("Daniel K", "Senior developer");
        CompanyDepartment productDevelopmentDepartment = new CompanyDepartment();
        productDevelopmentDepartment.addEmployee(dev1);
        productDevelopmentDepartment.addEmployee(dev2);

        IEmployee man1 = new Manager("Patrick H", "Project manager");
        IEmployee man2 = new Manager("Ivan M", "Lead project manager");
        CompanyDepartment productManagementDepartment = new CompanyDepartment();
        productManagementDepartment.addEmployee(man1);
        productManagementDepartment.addEmployee(man2);

        // thanks to composite, we can create more departments and
        // add the employees using the same process - by addingEmployee() but
        // now instead of adding employees separately, we add the whole department
        CompanyDepartment productDepartment = new CompanyDepartment();
        productDepartment.addEmployee(productDevelopmentDepartment);
        productDepartment.addEmployee(productManagementDepartment);
        // moreover, we can call the same method. In this case it will print out all
        // of the employees' method showEmployeeDetails()
        productDepartment.showEmployeeDetails();
    }
}
```

IEmployee - Component
CompanyDepartment - Composite (an elements which has sub-elements inside)
Manager, Developer - Leafs

Consequences:

Pros:

- Using polymorphism and recursion (solution depends on the smaller elements).
- It is open/closed so we can add new element types into the app without breaking the code

Cons:

- Might be difficult to provide a common interface for classes whose functionality differs too much.

Known uses:

- Tree-like structures
- When you want client to treat both simple and complex objects the same way