Contents

[Composite Design Pattern 1](#_Toc136461100)

[Applicability 2](#_Toc136461101)

[Pros and Cons 2](#_Toc136461102)

[Real World Example: 2](#_Toc136461103)

[Behavioral design patterns: 2](#_Toc136461104)

[Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects 2](#_Toc136461105)

[Chain of Responsibility 2](#_Toc136461106)

[Diagram 3](#_Toc136461107)

[Code Implementation: 3](#_Toc136461108)

[Pros and Cons 3](#_Toc136461109)

[Single Responsibility Principle. 3](#_Toc136461110)

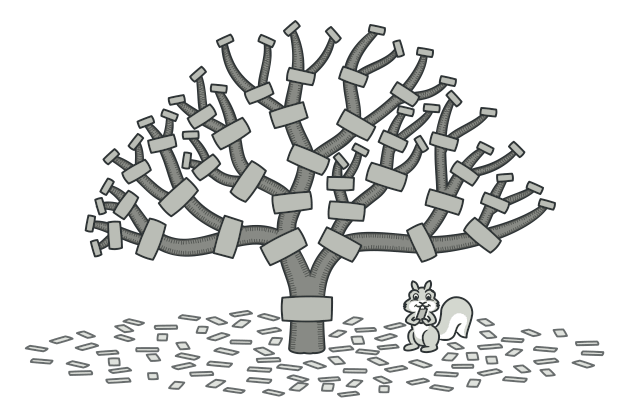
[Open/Closed Principle. 3](#_Toc136461111)

[Cons: 4](#_Toc136461112)

[Some requests may end up unhandled. 4](#_Toc136461113)

# Composite Design Pattern

**Also known as:**Object Tree



## **Applicability**

 Use the Composite pattern when you have to implement **a tree-like object structure.**

a **common interface: simple leaves and complex containers.**

**A container can be composed of both leaves and other containers.** This lets you construct a nested recursive object structure that resembles a tree.

**Use the pattern when you want the client code to treat both simple and complex elements uniformly.**

 All elements defined by the Composite pattern share a **common interface.**

## **Pros and Cons**

* You can work with complex tree structures more conveniently**: use polymorphism and recursion to your advantage.**
* **Open/Closed Principle**. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

<https://refactoring.guru/design-patterns/composite>

## Real World Example:

**Interface IEmployee**

Director’s vector<IEmployee>

|

Managers vector<IEmployee>

|

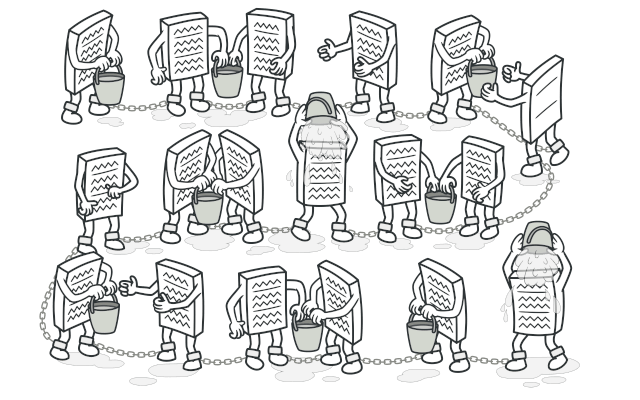
Employees

# Behavioral design patterns:

## Behavioral design patterns are concerned with **algorithms and the assignment of responsibilities between objects**

# Chain of Responsibility

Chain of Responsibility is a behavioral design pattern that lets you pass requests **along a chain of handlers**. Upon receiving a request, each handler decides either to **process the request or to pass it to the next handler** in the chain.



## Diagram

## Code Implementation:

    int request[13] = {1, 3, 23, 12, 5, 6, 16, 13, 11, 23, 22, 27, 3};

    Handler \*h1 = new ConcreteHandle1();

    Handler \*h2 = new ConcreteHandle2();

    Handler \*h3 = new ConcreteHandle3();

    h1->SetSuccessor(h2);

    h2->SetSuccessor(h3);

    for (int i = 0; i < 13; i++)

    {

        h1->HandleRequest(request[i]);

    }

## Pros and Cons

You can control the order of request handling.

Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform operations.

Open/Closed Principle. You can introduce new handlers into the app without breaking the existing client code.

## Cons:

### Some requests may end up unhandled.