Contents

[Composite Design Pattern 1](#_Toc136632718)

[Applicability 2](#_Toc136632719)

[Pros and Cons 2](#_Toc136632720)

[https://refactoring.guru/design-patterns/composite 2](#_Toc136632721)

[Real World Example: 2](#_Toc136632722)

[Behavioral design patterns: 3](#_Toc136632723)

[Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects 3](#_Toc136632724)

[Chain of Responsibility 3](#_Toc136632725)

[https://refactoring.guru/design-patterns/chain-of-responsibility 3](#_Toc136632726)

[Diagram 3](#_Toc136632727)

[Code Implementation: 3](#_Toc136632728)

[Pros and Cons 4](#_Toc136632729)

[Single Responsibility Principle. 4](#_Toc136632730)

[Open/Closed Principle. 4](#_Toc136632731)

[Cons: 4](#_Toc136632732)

[Some requests may end up unhandled. 4](#_Toc136632733)

[Observer design Pattern: 4](#_Toc136632734)

[https://refactoring.guru/design-patterns/observer 4](#_Toc136632735)

[Code Implementation: 4](#_Toc136632736)

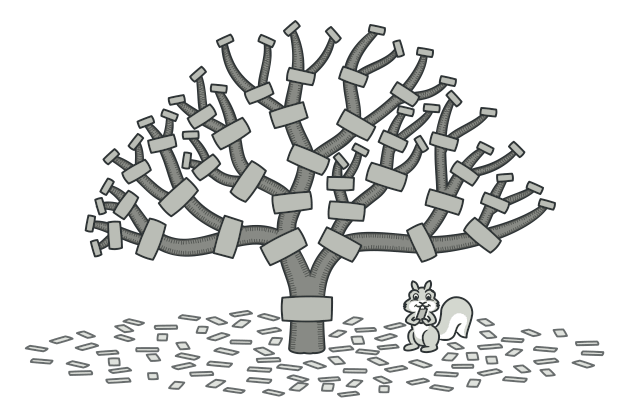
[Abstract Factory 5](#_Toc136632737)

[Intent 5](#_Toc136632738)

[Main code implementation: 5](#_Toc136632739)

# 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

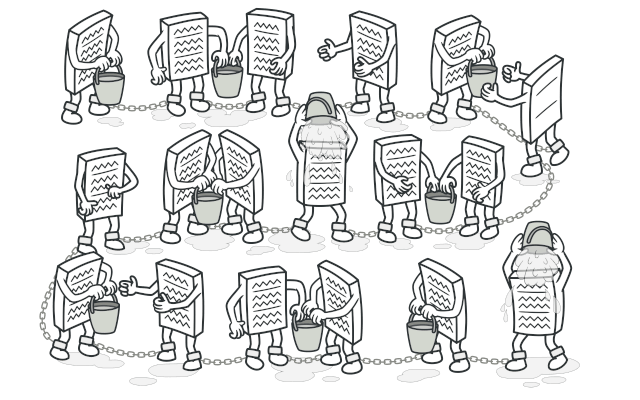
# Behavioral design patterns:

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

# Chain of Responsibility

## https://refactoring.guru/design-patterns/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.

# Observer design Pattern:

Observer is a behavioral design pattern that allows some objects to notify other objects about changes in their state.

## https://refactoring.guru/design-patterns/observer

## Code Implementation:

    Stock \*ibm = new IBM("IBM", 120);

    IInvestor \*i1 = new Investor("Ajay");

    IInvestor \*i2 = new Investor("Gopal");

    IInvestor \*i3 = new Investor("Vinita");

    ibm->attach(i1);

    ibm->attach(i2);

    ibm->attach(i3);

    ibm->setPrice(120.50);

    ibm->setPrice(112.50);

    ibm->setPrice(125.50);

    ibm->detach(i3);

    ibm->setPrice(125.50);

# Abstract Factory

## Intent

**Abstract Factory** is a creational design pattern that lets you **produce families of related objects without specifying their concrete classes.**

## Main code implementation:

    // Icar is the first level abstraction but IVehicleCompany is

// the 2nd level abstraction.

    IVehicleCompany \*comp = new tata();

    Icar \*cars = comp->GetCar(); // we don’t know concrete subclass

    cars->GetCar();

    Itruck \*trucks = comp->GetTruck();

    trucks->GetTruck();

    IVehicleCompany \*comp2 = new Mahindra();

    Icar \*cars2 = comp2->GetCar(); // we don’t know concrete subclass

    cars2->GetCar();

    Itruck \*trucks2 = comp2->GetTruck();

    trucks2->GetTruck();

    /\* conclusion:

    design pattern that lets us produce families of

    related objects (Mahindra, tata) without specifying their concrete

classes (Cars, Trucks).

    \*/