Contents

[Composite Design Pattern 1](#_Toc137242709)

[Applicability 2](#_Toc137242710)

[Pros and Cons 2](#_Toc137242711)

[https://refactoring.guru/design-patterns/composite 2](#_Toc137242712)

[Real World Example: 2](#_Toc137242713)

[Behavioral design patterns: 3](#_Toc137242714)

[Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects 3](#_Toc137242715)

[Chain of Responsibility 3](#_Toc137242716)

[https://refactoring.guru/design-patterns/chain-of-responsibility 3](#_Toc137242717)

[Diagram 3](#_Toc137242718)

[Code Implementation: 3](#_Toc137242719)

[Pros and Cons 4](#_Toc137242720)

[Single Responsibility Principle. 4](#_Toc137242721)

[Open/Closed Principle. 4](#_Toc137242722)

[Cons: 4](#_Toc137242723)

[Some requests may end up unhandled. 4](#_Toc137242724)

[Observer design Pattern: 4](#_Toc137242725)

[https://refactoring.guru/design-patterns/observer 4](#_Toc137242726)

[Code Implementation: 4](#_Toc137242727)

[Abstract Factory 5](#_Toc137242728)

[Intent 5](#_Toc137242729)

[Main code implementation: 5](#_Toc137242730)

[Strategy 6](#_Toc137242731)

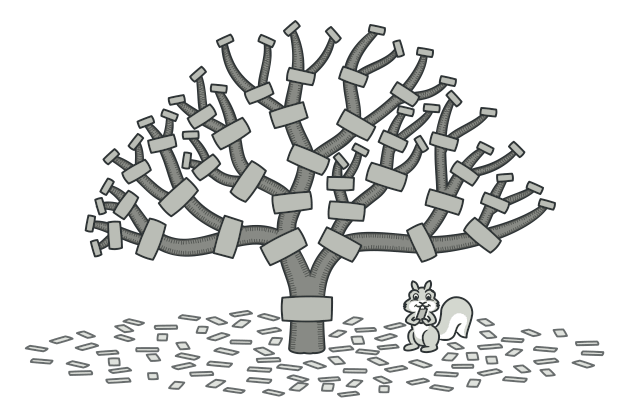
[Intent 6](#_Toc137242732)

[Code Implementation Main 6](#_Toc137242733)

[Applicability 6](#_Toc137242734)

# 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

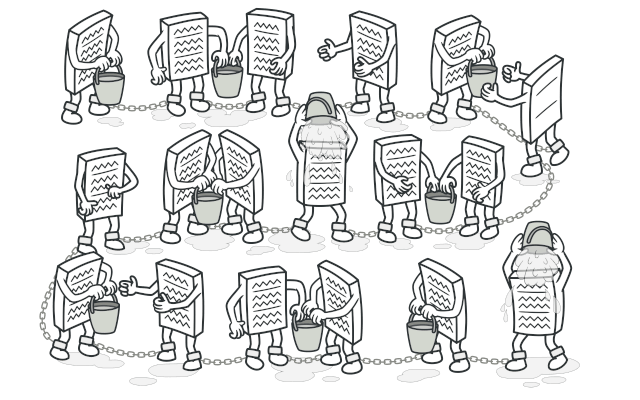
# Behavioral design patterns:

## 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);

# ****Factory Method**** in C++

**Factory method** is a creational design pattern which solves the problem of **creating product objects without specifying their concrete classes.**

class Creater // Factory class

{

public:

    virtual Product \*FactoryMethod() = 0;

    virtual ~Creater()

    {

    }

};

# 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).

    \*/

# Strategy

## Intent

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

## Code Implementation Main

void DriveTheTrasportationStrategies(RouteStrategy \*routes)

{

    Nevigator nv; // this is a context maintains a reference to one of the strategy.

    // objects. The context doesn't know the concrete class of a

    // strategy. It should work with all strategies via the

    // strategy interface.

    nv.SetRouteStrategy(routes); // strategy can be switched at runtime.

    nv.DoSomethingBusinessLogic(); // executes the strategy

}

int main()

{

    RouteStrategy \*routes = new PublicTransportStrategy(); // strategy 1

    DriveTheTrasportationStrategies(routes);

    routes = new RoadStrategy(); // strategy 2

    DriveTheTrasportationStrategies(routes);

    routes = new WalkingStrategy(); // strategy 3

    DriveTheTrasportationStrategies(routes);

    return 0;

}

## Applicability

* Use the Strategy pattern when you want to use **different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.**
* The Strategy pattern lets you **indirectly alter the object’s behavior at runtime**
* Use the Strategy when you have a lot of **similar classes that only differ in the way they execute some behavior.**
* The strategy pattern uses [composition instead of inheritance](https://en.wikipedia.org/wiki/Composition_over_inheritance).
* In the strategy pattern, behaviors are defined as separate interfaces and specific classes that implement these interfaces. This allows better decoupling between the behavior and the class that uses the behavior.
* The behavior can be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes.
* Behaviors can also be changed at run-time as well as at design-time.

# Builder