Behavioral Design Patterns

Chain of Responsibility

🧠 **Chain of Responsibility (*Цепочка обязанностей*)**

📌 **Назначение:**

Позволяет передавать запрос последовательно по цепочке обработчиков, пока один из них не обработает его.

👥 **Участники:**

- **Handler** – интерфейс обработчика (`handle_request()`).
- **ConcreteHandler** – конкретный обработчик (может передать дальше).
- **Client** – инициирует запрос.

🛠️ **Примеры:**

✔️ Обработка HTTP-запросов (middleware в веб-фреймворках).

✔️ Фильтры ввода (проверка, авторизация, логирование).

💡 **Преимущества:**

✅ Гибкость: можно динамически менять цепочку.

✅ Отправитель не знает, кто обработает запрос.

⚠️ **Недостатки:**

❌ Нет гарантии, что запрос будет обработан.

❌ Может усложнить отладку.

```
abstract class Handler
{
    protected Handler _next;
    public void SetNext(Handler next) => _next = next;
    public abstract void HandleRequest(int request);
```

```
}

class ConcreteHandler1 : Handler
{
    public override void HandleRequest(int request)
    {
        if (request <= 10)
            Console.WriteLine("Handled by Handler1");
        else
            _next?.HandleRequest(request);
    }
}

class ConcreteHandler2 : Handler
{
    public override void HandleRequest(int request)
    {
        Console.WriteLine("Handled by Handler2");
    }
}

// Использование:
var h1 = new ConcreteHandler1();
var h2 = new ConcreteHandler2();
h1.SetNext(h2);
h1.HandleRequest(5);  // Handled by Handler1
h1.HandleRequest(15); // Handled by Handler2
```

Command

🧠 **Command (*Команда*)**

📌 **Назначение:**

Инкапсулирует запрос в виде объекта, позволяя параметризовать клиенты с разными запросами, ставить их в очередь или отменять.

👥 **Участники:**

- **Command** – интерфейс команды (`execute()`, `undo()`).

- **ConcreteCommand** – конкретная команда, связанная с **Receiver**.
- **Invoker** – инициирует выполнение команды (кнопка, меню).
- **Receiver** – объект, который знает, как выполнить действие.

🛠️ **Примеры:**

✔️ Кнопка "Отменить" (Undo) в графическом редакторе.

✔️ Очередь задач (например, отправка отложенных запросов).

💡 **Преимущества:**

✅ Гибкость: команды можно логировать, откладывать, отменять.

✅ Разделение отправителя и получателя.

⚠️ **Недостатки:**

❌ Увеличивает количество классов.

```
interface ICommand
{
    void Execute();
}

class LightOnCommand : ICommand
```

```
{
    public void Execute() => Console.WriteLine("Light is on");
}

class RemoteControl
{
    private ICommand _command;
    public void SetCommand(ICommand command) => _command = command;
    public void PressButton() => _command.Execute();
}

// Использование:
var remote = new RemoteControl();
remote.SetCommand(new LightOnCommand());
remote.PressButton();
```

Iterator

🧠 **Iterator (*Итератор*)**

📌 **Назначение:**

Предоставляет способ последовательного доступа к элементам коллекции, не раскрывая её внутренней структуры.

👥 **Участники:**

- **Iterator** – интерфейс для обхода коллекции (`next()`, `has_next()`).
- **ConcreteIterator** – реализует алгоритм обхода.
- **Aggregate** – интерфейс коллекции для создания итератора.
- **ConcreteAggregate** – конкретная коллекция (список, дерево и т. д.).

🛠️ **Примеры:**

✔️ Цикл `foreach` в языках программирования.

✔️ Обход элементов списка, дерева, графа.

💡 **Преимущества:**

✅ Единый интерфейс для разных коллекций.

✅ Возможность параллельного обхода (несколько итераторов).

⚠ **Недостатки:**

❌ Избыточность для простых коллекций.

```
interface IIterator<T>
{
    bool HasNext();
    T Next();
}

class ConcreteIterator<T> : IIterator<T>
{
    private readonly T[] _collection;
    private int _position;

    public ConcreteIterator(T[] collection) => _collection = collection;

    public bool HasNext() => _position < _collection.Length;
```

```
    public T Next() => _collection[_position++];
}

// Использование:
string[] items = { "A", "B", "C" };
var iterator = new ConcreteIterator<string>(items);
while (iterator.HasNext())
{
    Console.WriteLine(iterator.Next());
}
```

Mediator

🧠 **Mediator (*Посредник*)**

📌 **Назначение:**

Централизует взаимодействие между объектами, уменьшая их прямую связанность.

👥 **Участники:**

- **Mediator** – интерфейс для взаимодействия между **Colleague**.
- **ConcreteMediator** – конкретный посредник, координирующий коллег.
- **Colleague** – объекты, которые общаются через посредника, а не напрямую.

🛠️ **Примеры:**

✔️ Чат-комната (посредник обрабатывает сообщения между пользователями).

✔️ Центр управления воздушным движением (координирует самолёты).

💡 **Преимущества:**

✅ Уменьшает связанность между компонентами.

✅ Упрощает добавление новых участников.

⚠️ **Недостатки:**

❌ Посредник может стать "божественным объектом" (слишком сложным).

```
interface IMediator
{
    void Send(string message, Colleague colleague);
}

class ConcreteMediator : IMediator
{
    public Colleague Colleague1 { get; set; }
    public Colleague Colleague2 { get; set; }

    public void Send(string message, Colleague colleague)
    {
        if (colleague == Colleague1)
            Colleague2.Notify(message);
        else
            Colleague1.Notify(message);
```

```
    }
}

abstract class Colleague
{
    protected IMediator Mediator;
    protected Colleague(IMediator mediator) => Mediator = mediator;
    public abstract void Notify(string message);
    public abstract void Send(string message);
}

class ConcreteColleague1 : Colleague
{
    public ConcreteColleague1(IMediator mediator) : base(mediator) {}

    public override void Notify(string message) => Console.WriteLine($"Colleag

    public override void Send(string message) => Mediator.Send(message, this)
}

// Использование:
var mediator = new ConcreteMediator();
var c1 = new ConcreteColleague1(mediator);
var c2 = new ConcreteColleague2(mediator); // Аналогичный класс

mediator.Colleague1 = c1;
mediator.Colleague2 = c2;

c1.Send("Hello from Colleague1!");
```

Memento

🧠 **Memento** (*Снимок*)

📌 Сохраняет и восстанавливает состояние объекта без нарушения инкапсуляции.

🛠️ **Пример использования:**

Отмена/повтор действий (Undo/Redo) в текстовом редакторе, сохранение состояний игры.

👥 **Участники:**

- **Originator** (*Создатель*) – объект, состояние которого нужно сохранить.
- **Memento** (*Снимок*) – неизменяемый объект, хранящий состояние Originator.
- **Caretaker** (*Опекун*) – управляет историей снимков (сохраняет и восстанавливает их).

💡 **Преимущества:**

✅ Позволяет откатывать состояние объекта.

✅ Не нарушает инкапсуляцию (Originator сам управляет своими снимками).

⚠️ **Недостатки:**

❌ Может потреблять много памяти, если сохранять много состояний.

❌ Усложняет код, если объектов много.

```
class Originator
{
    private string _state;
```

```
    public string State
    {
        get => _state;
        set { _state = value; Console.WriteLine($"State set to: {_state}"); }
    }

    public Memento CreateMemento() => new Memento(_state);

    public void SetMemento(Memento memento) => State = memento.State;
}

class Memento
{
    public string State { get; }
    public Memento(string state) => State = state;
}

class Caretaker
{
    public Memento Memento { get; set; }
}

// Использование:
var originator = new Originator();
var caretaker = new Caretaker();

originator.State = "State1";
caretaker.Memento = originator.CreateMemento();

originator.State = "State2";

originator.SetMemento(caretaker.Memento); // Восстановление State1
```

Observer

🧠 **Observer** (*Наблюдатель*)

📌 **Назначение:**

Определяет зависимость **«один ко многим»** между объектами так, что при изменении состояния одного объекта (**Subject**) все зависящие от него (**Observers**) автоматически уведомляются и обновляются.

👥 **Участники:**

- **Subject** – хранит состояние и уведомляет наблюдателей.
- **Observer** – интерфейс для подписчиков (метод `update()`).
- **ConcreteObserver** – конкретные подписчики, реагирующие на изменения.

🛠️ **Примеры:**

✔️ Система уведомлений (например, email-рассылка).

✔️ Реактивное программирование (RxJS, MobX).

💡 **Преимущества:**

✅ Гибкая подписка/отписка.

✅ Subject не зависит от конкретных Observer.

⚠️ **Недостатки:**

❌ Непредсказуемый порядок уведомлений.

❌ Возможны утечки памяти (если не отписаться).

```
interface IObserver
{
    void Update(string message);
}

class ConcreteObserver : IObserver
{
    public void Update(string message) => Console.WriteLine($"Received: {mes
}

class Subject
{
    private List<IObserver> _observers = new();

    public void AddObserver(IObserver observer) => _observers.Add(observer)
    public void Notify(string message) => _observers.ForEach(o => o.Update(me
}

// Использование:
var subject = new Subject();
subject.AddObserver(new ConcreteObserver());
subject.Notify("Hello Observers!");
```

State

🧠 **State** (*Состояние*)

📌 **Назначение:**

Позволяет объекту менять поведение при изменении внутреннего состояния, создавая иллюзию изменения класса.

👥 **Участники:**

- **Context** – объект, чьё поведение меняется.
- **State** – интерфейс состояния.
- **ConcreteState** – конкретные состояния (например, `Locked`, `Unlocked`).

🛠️ **Примеры:**

✔️ Банкомат (состояния: `CardInserted`, `PinEntered`, `CashWithdrawn`).

✔️ Дверь (`Open`, `Closed`, `Locked`).

💡 **Преимущества:**

✅ Замена громоздких `if/switch` на полиморфизм.

✅ Лёгкое добавление новых состояний.

⚠️ **Недостатки:**

❌ Избыточность для простых случаев.

```
interface IState
{
    void Handle(Context context);
}

class ConcreteStateA : IState
{
```

```
    public void Handle(Context context) => context.State = new ConcreteStateB
}

class ConcreteStateB : IState
{
    public void Handle(Context context) => context.State = new ConcreteStateA
}

class Context
{
    public IState State { get; set; }
    public Context(IState state) => State = state;
    public void Request() => State.Handle(this);
}

// Использование:
var context = new Context(new ConcreteStateA());
context.Request(); // Изменит состояние на ConcreteStateB
```

Strategy

🧠 **Strategy** (*Стратегия*)

📌 **Назначение:**

Инкапсулирует семейство алгоритмов, делая их взаимозаменяемыми. Позволяет менять алгоритм независимо от клиента.

👥 **Участники:**

- **Strategy** – интерфейс стратегии.
- **ConcreteStrategy** – конкретные алгоритмы (например, сортировка по цене/дате).
- **Context** – использует стратегию.

🛠️ **Примеры:**

✔️ Сортировка товаров (по цене, рейтингу).

✔️ Оплата (PayPal, CreditCard, Crypto).

💡 **Преимущества:**

✅ Возможность смены алгоритма **на лету**.

✅ Изоляция кода алгоритмов от клиента.

⚠️ **Недостатки:**

❌ Усложнение кода при малом числе алгоритмов.

```
interface ISortStrategy
{
    void Sort(int[] data);
}

class QuickSort : ISortStrategy
{
    public void Sort(int[] data) => Console.WriteLine("Quick sorting");
}

class BubbleSort : ISortStrategy
{
    public void Sort(int[] data) => Console.WriteLine("Bubble sorting");
}

class Sorter
{
```

```
    private ISortStrategy _strategy;
    public void SetStrategy(ISortStrategy strategy) => _strategy = strategy;
    public void ExecuteSort(int[] data) => _strategy.Sort(data);
}

// Использование:
var sorter = new Sorter();
sorter.SetStrategy(new QuickSort());
sorter.ExecuteSort(new[] {1, 5, 3});
```

Template Method

🧠 **Template Method** (*Шаблонный метод*)

📌 **Назначение:**

Определяет **скелет алгоритма** в базовом классе, позволяя подклассам переопределять отдельные шаги без изменения структуры.

👥 **Участники:**

- **AbstractClass** – определяет шаблонный метод (`templateMethod()`).
- **ConcreteClass** – реализует специфичные шаги.

🛠️ **Примеры:**

✔️ Фреймворки (например, `React.Component` с `render()`).

✔️ Алгоритмы приготовления напитков (кофе/чай).

💡 **Преимущества:**

✅ Исключает дублирование кода.

✅ Контроль над структурой алгоритма.

⚠️ **Недостатки:**

❌ Жёсткая связь через наследование.

```
abstract class Game
{
    public void Play()
    {
        Initialize();
        StartPlay();
        EndPlay();
    }

    protected abstract void Initialize();
    protected abstract void StartPlay();
    protected virtual void EndPlay() => Console.WriteLine("Game ended");
}

class Chess : Game
{
    protected override void Initialize() => Console.WriteLine("Chess initialized");
    protected override void StartPlay() => Console.WriteLine("Chess started");
}

// Использование:
```

```
Game game = new Chess();
game.Play();
```

Visitor

🧠 **Visitor (*Посетитель*)**

📌 **Назначение:**

Позволяет добавлять **новые операции** к объектам, **не изменяя их классы**.

👥 **Участники:**

- **Visitor** – интерфейс операций (`visitElementA()`, `visitElementB()`).
- **ConcreteVisitor** – конкретные операции.
- **Element** – объекты, принимающие посетителя (`accept(visitor)`).

🛠️ **Примеры:**

✔️ Обход AST (абстрактного синтаксического дерева).

✔️ Экспорт данных в XML/JSON.

💡 **Преимущества:**

✅ Отделение логики операций от структуры объектов.

✅ Удобен для сложных иерархий (например, компиляторы).

⚠️ **Недостатки:**

❌ Нарушает инкапсуляцию (требует `public`-методов).

❌ Сложно добавлять новые типы элементов.

```
interface IVisitor
{
    void Visit(ElementA element);
    void Visit(ElementB element);
}

class ConcreteVisitor : IVisitor
{
    public void Visit(ElementA element) ⇒ element.OperationA();
```

```
    public void Visit(ElementB element) => element.OperationB();
}

interface IElement
{
    void Accept(IVisitor visitor);
}

class ElementA : IElement
{
    public void Accept(IVisitor visitor) => visitor.Visit(this);
    public void OperationA() => Console.WriteLine("Operation A");
}

class ElementB : IElement
{
    public void Accept(IVisitor visitor) => visitor.Visit(this);
    public void OperationB() => Console.WriteLine("Operation B");
}

// Использование:
var elements = new IElement[] { new ElementA(), new ElementB() };
var visitor = new ConcreteVisitor();

foreach (var element in elements)
{
    element.Accept(visitor);
}
```

Behavioral Design Patterns Overview

Pattern	Core Purpose	When to Use	Key Benefit
Chain of Responsibility	Sequential request handling	When multiple objects can handle a request	Loose coupling between sender and receiver
Command	Action encapsulation	For parameterizing and queueing operations	Supports undo, logging, and queuing

Iterator	Collection traversal	When accessing elements without exposing structure	Uniform interface for different collections
Mediator	Centralized communication	With complex object interactions	Reduces dependencies between objects
Memento	State preservation	For implementing undo mechanisms	Maintains encapsulation of object state
Observer	Event notification	When state changes affect multiple objects	Supports one-to-many dependencies
State	Behavior modification	When object behavior depends on state	Simplifies conditional logic
Strategy	Algorithm encapsulation	For interchangeable algorithms	Runtime algorithm switching
Template Method	Algorithm structuring	For defining skeleton with customizable steps	Code reuse with customization points
Visitor	Operation extension	To add operations without changing classes	Separates algorithms from object structure

Patterns by Application Area

User Interface Patterns

- Command - for menu actions, buttons
- Observer - for UI updates
- State - for visual state changes
- Mediator - for component coordination

Event Handling Patterns

- Chain of Responsibility - for event propagation
- Observer - for event notifications
- Command - for event actions

Data Processing Patterns

- Iterator - for collection traversal
- Visitor - for complex operations
- Strategy - for different algorithms

Implementation Complexity

Quick Reference Guide

▼ When to use Chain of Responsibility

- You have multiple handlers for a request
- You don't know which handler should process a request
- You want to decouple sender from receiver
- Order of handling matters

▼ When to use Command

- You need to parameterize objects with operations
- You want to queue, log, or undo operations
- You need to support callbacks

▼ When to use Iterator

- You need to access collection elements without exposing structure

- You want to provide uniform traversal interface for different collections
- You need multiple simultaneous traversals

▼ When to use Mediator

- Many objects communicate in complex ways
- You want to reduce dependencies between objects
- You need to centralize control

▼ When to use Memento

- You need to capture object state for later restoration
- Direct interface would expose implementation details
- You need to implement undo mechanism

▼ When to use Observer

- Changes to one object require changes to others
- You don't know how many objects need to be updated
- You want loose coupling between related objects

▼ When to use State

- Object behavior depends on its state
- Complex conditional statements control behavior
- Transitions between states are well-defined

▼ When to use Strategy

- You need different variants of an algorithm
- You want to avoid exposing complex algorithm details
- You need to switch algorithms at runtime

▼ When to use Template Method

- You have an algorithm with some steps that can vary
- You want to avoid code duplication
- You need to control when subclasses can override steps

▼ When to use Visitor

- You need to perform operations on all elements of complex structure
- You want to add new operations without changing element classes
- Operations need access to state normally hidden from outside

💡 **Pro Tip:** When choosing between behavioral patterns, consider:
- What kind of communication happens between objects?
- What level of coupling is acceptable?
- How likely are behavior changes during runtime?
- Is state preservation important?