

# Behavioral Design Patterns

📌 Описывают способы эффективного взаимодействия объектов и распределения ответственности.

## 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:**

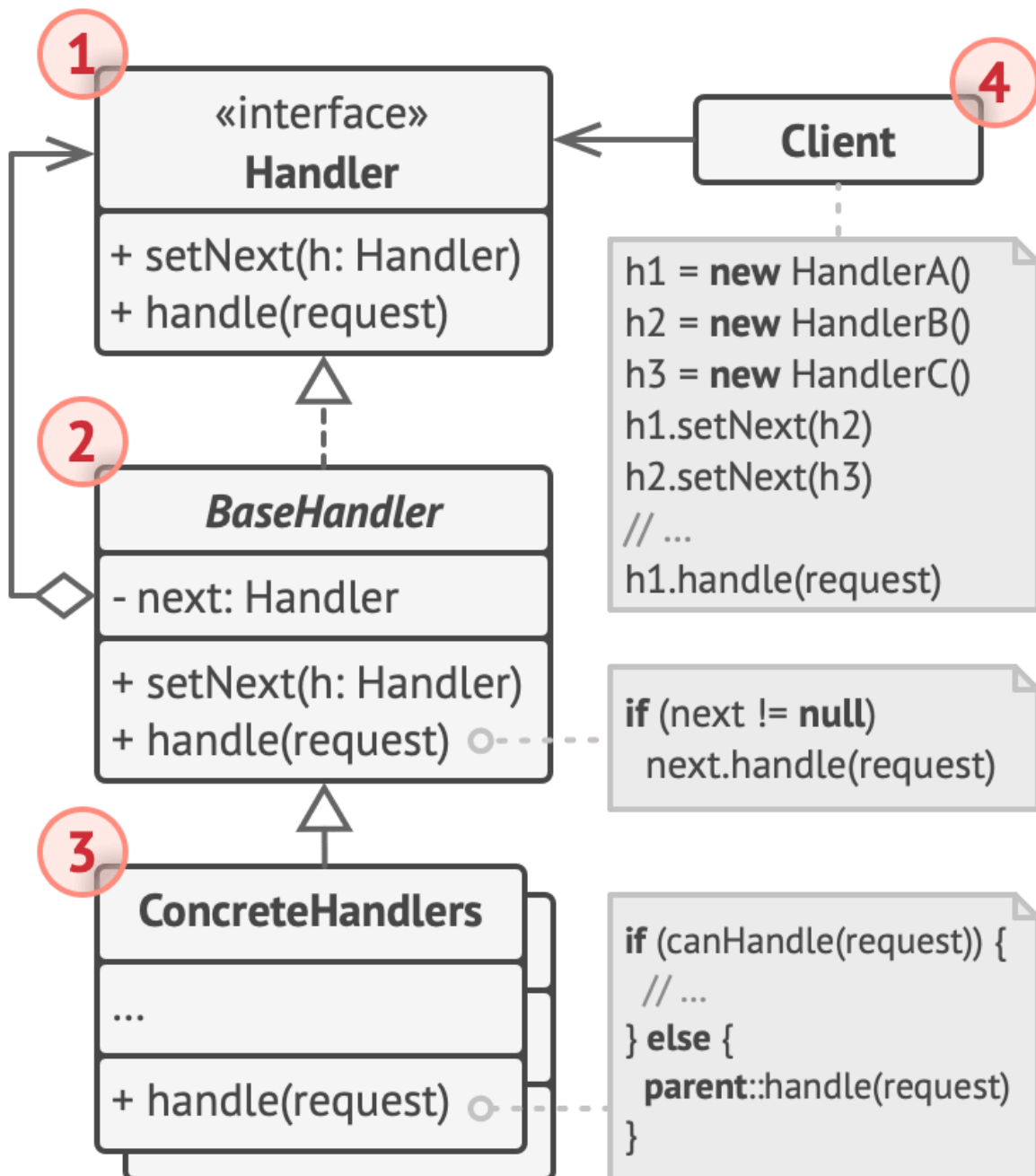
- ✓ Обработка HTTP-запросов (middleware в веб-фреймворках).
- ✓ Фильтры ввода (проверка, авторизация, логирование).

💡 **Преимущества:**

- ✓ Гибкость: можно динамически менять цепочку.
- ✓ Отправитель не знает, кто обрабатывает 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 (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. Хранит ссылку на **Receiver** и вызывает нужные методы.
- **Invoker** – инициирует выполнение команды (кнопка, меню). Хранит команду и вызывает её метод `execute()` при каком-либо событии.
- **Receiver** – объект, который знает, как выполнить действие.

#### 🔧 Примеры:

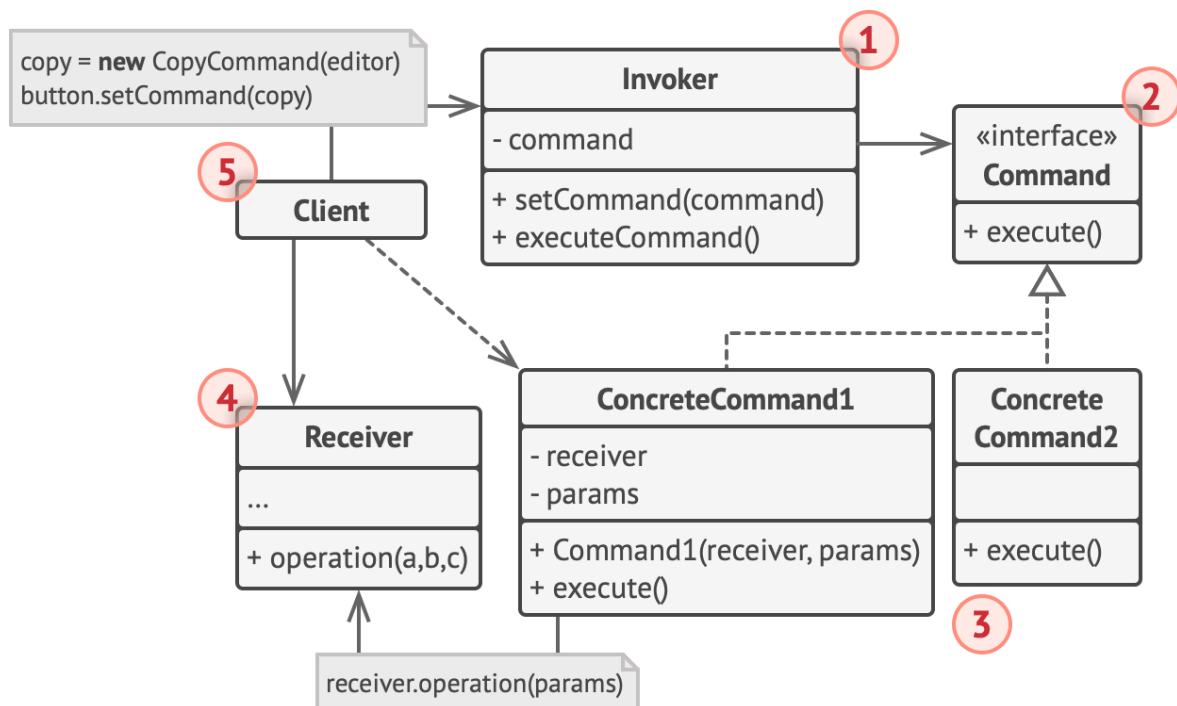
- ✓ Кнопка "Отменить" (Undo) в графическом редакторе.
- ✓ Очередь задач (например, отправка отложенных запросов).

#### 💡 Преимущества:

- ✓ Гибкость: команды можно логировать, откладывать, отменять.
- ✓ Разделение отправителя и получателя.

#### ⚠ Недостатки:

- ✗ Увеличивает количество классов.



```

// Получатель команды - знает КАК выполнить действие
public class Light
{
    public void TurnOn() ⇒ Console.WriteLine("Свет включен");
}

```

```

    public void TurnOff() ⇒ Console.WriteLine("Свет выключен");
}

public class MusicPlayer
{
    public void Play() ⇒ Console.WriteLine("Музыка включена");
    public void Stop() ⇒ Console.WriteLine("Музыка выключена");
}

// Интерфейс команды - ЧТО можно сделать
public interface ICommand
{
    void Execute();
    void Undo(); // Для отмены действия
}

// Конкретные команды - связывают действие с получателем
public class LightOnCommand : ICommand
{
    private Light _light;

    public LightOnCommand(Light light) ⇒ _light = light;

    public void Execute() ⇒ _light.TurnOn();
    public void Undo() ⇒ _light.TurnOff();
}

public class MusicPlayCommand : ICommand
{
    private MusicPlayer _player;

    public MusicPlayCommand(MusicPlayer player) ⇒ _player = player;

    public void Execute() ⇒ _player.Play();
    public void Undo() ⇒ _player.Stop();
}

// Пульт управления - знает КОГДА выполнить команду

```

```

public class RemoteControl
{
    private ICommand _command;

    public void SetCommand(ICommand command) ⇒ _command = command;
    public void PressButton() ⇒ _command?.Execute();
    public void PressUndo() ⇒ _command?.Undo();
}

class Program
{
    static void Main()
    {
        // 1. Создаем устройства (Receivers)
        var livingRoomLight = new Light();
        var kitchenPlayer = new MusicPlayer();

        // 2. Создаем команды
        var lightOn = new LightOnCommand(livingRoomLight);
        var musicPlay = new MusicPlayCommand(kitchenPlayer);

        // 3. Настраиваем пульт (Invoker)
        var remote = new RemoteControl();

        // Управляем светом
        remote.SetCommand(lightOn);
        remote.PressButton(); // Включаем свет
        remote.PressUndo();  // Выключаем свет

        // Управляем музыкой
        remote.SetCommand(musicPlay);
        remote.PressButton(); // Включаем музыку
        remote.PressUndo();  // Выключаем музыку
    }
}

```

## 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:

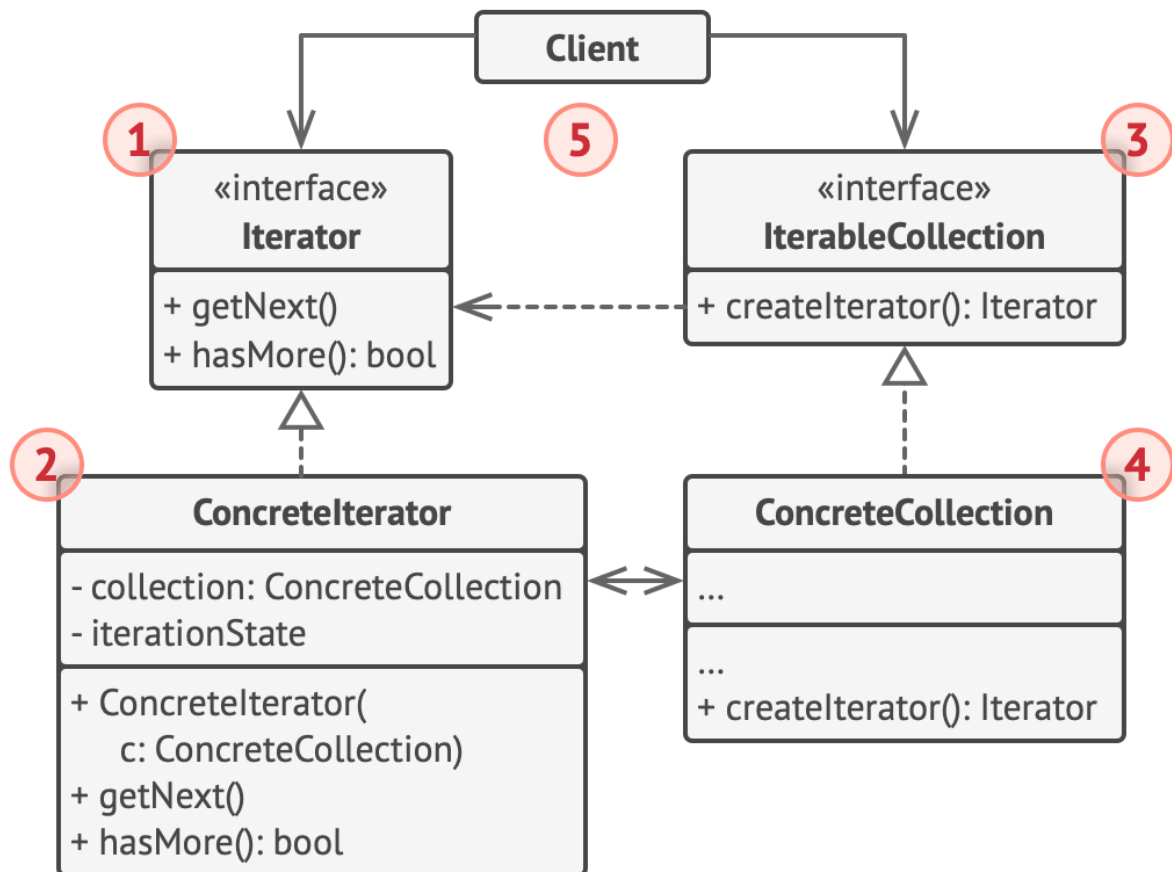
- ✓ Цикл `foreach` в языках программирования.
- ✓ Обход элементов списка, дерева, графа.

### 💡 Преимущества:

- ✓ Единый интерфейс для разных коллекций.
- ✓ Возможность параллельного обхода (несколько итераторов).

### ⚠ Недостатки:

- ✗ Избыточность для простых коллекций.



```

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

class ConcreteIterator<T> : Iterator<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, определяющий способ **взаимодействия между Colleague**.
- **ConcreteMediator** – конкретная реализация посредника, знает о коллегах и **координирует их взаимодействие**. Управляет логикой маршрутизации сообщений/действий между **Colleague**.
- **Colleague** – объекты, которые **общаются через посредника**, а не напрямую.

#### Примеры:

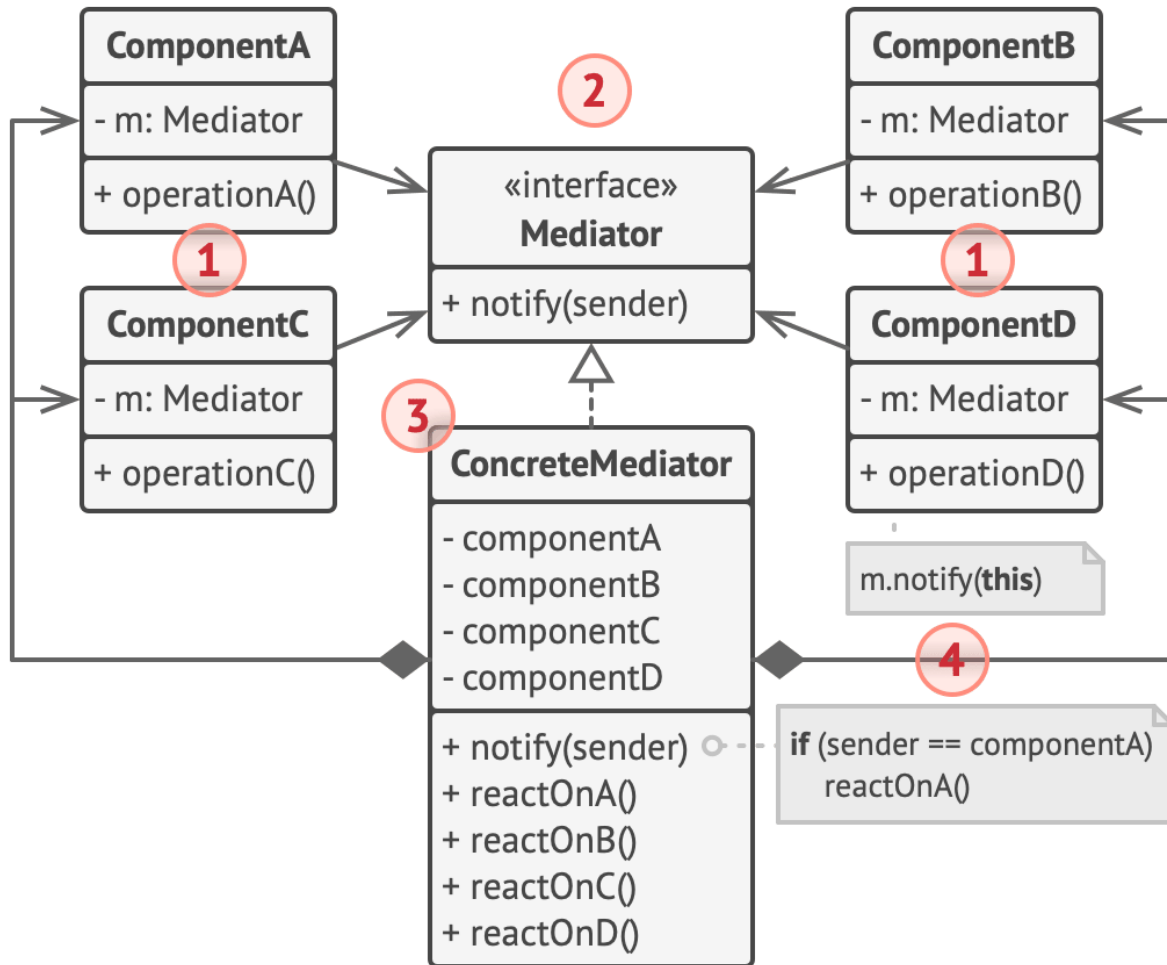
- ✓ Чат-комната (посредник обрабатывает сообщения между пользователями).
- ✓ Центр управления воздушным движением (координирует самолёты).

#### 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($"Colleague1: {message}");

    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 игры.

#### 👤 Участники:

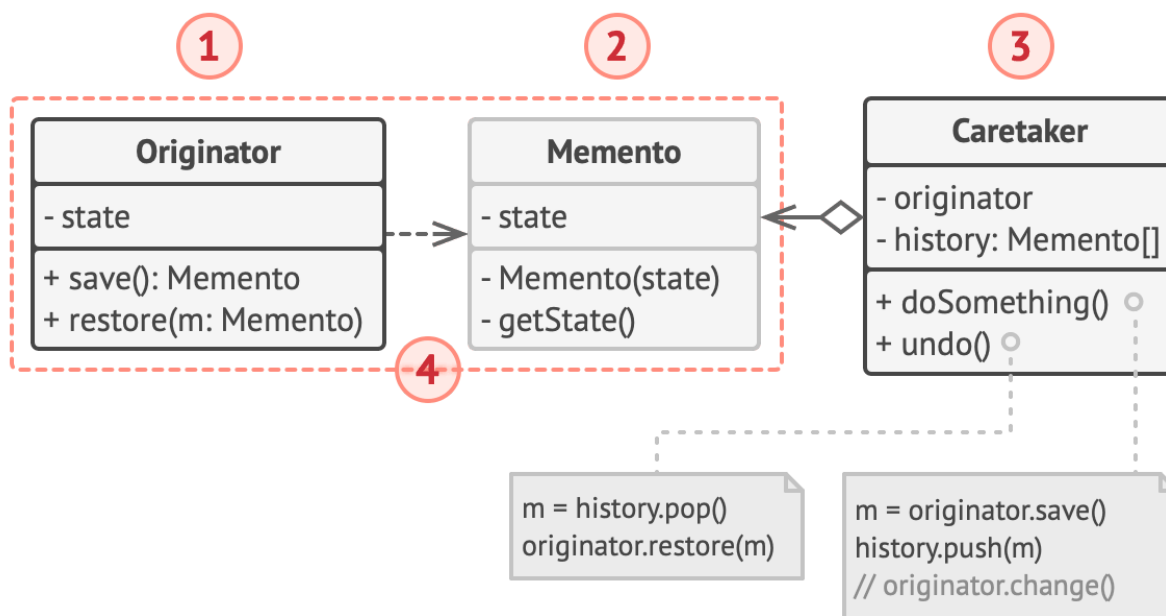
- **Originator** (Создатель) – объект, состояние которого нужно сохранить.
- **Memento** (Снимок) – неизменяемый объект, хранящий состояние Originator.
- **Caretaker** (Опекун) – управляет историей снимков (сохраняет и восстанавливает их).

#### 💡 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:

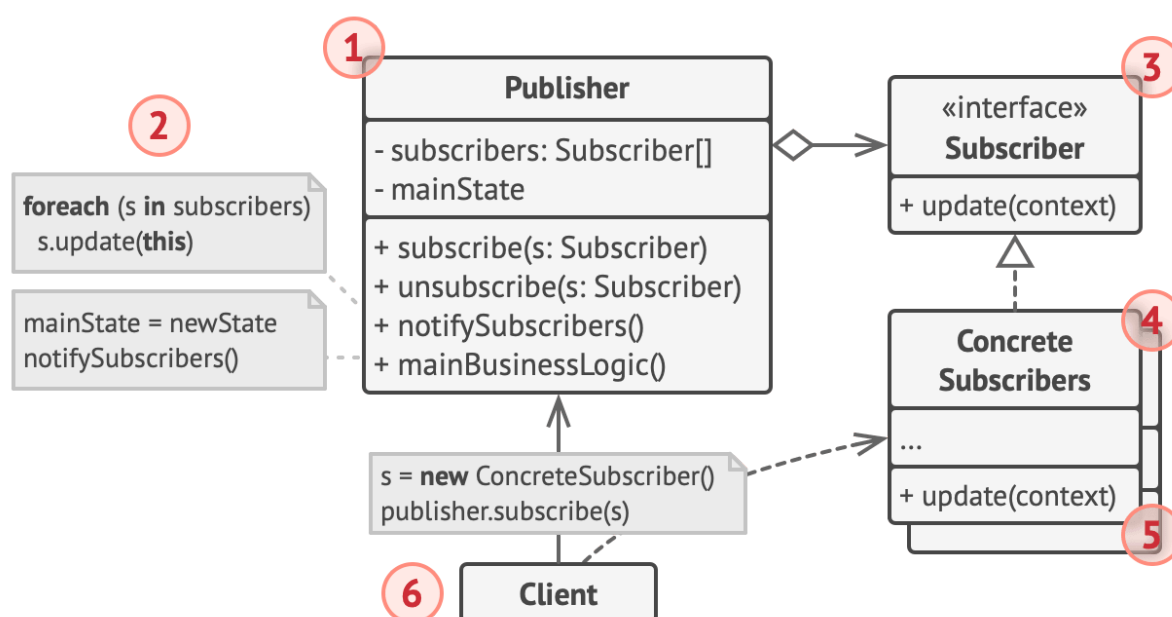
- ✓ Система уведомлений (например, email-рассылка).
- ✓ Реактивное программирование (RxJS, MobX).

#### 💡 Преимущества:

- ✓ Гибкая подписка/отписки.
- ✓ Subject не зависит от конкретных Observer.

#### ⚠ Недостатки:

- ✗ Непредсказуемый порядок уведомлений.
- ✗ Возможны утечки памяти (если не отписаться).



```

public interface IObservable {
    void Update(string message);
}

public class NewsPublisher {
    private List<IObservable> _observers = new List<IObservable>();


    public void Subscribe(IObservable observer) ⇒ _observers.Add(observer);
    public void Notify(string news) ⇒ _observers.ForEach(o ⇒ o.Update(news))
}

public class Subscriber : IObservable {
    public void Update(string message) ⇒ Console.WriteLine($"Получено: {message}");
}

// Использование:
var publisher = new NewsPublisher();
publisher.Subscribe(new Subscriber());
publisher.Notify("Новая статья!");

```

## 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`).

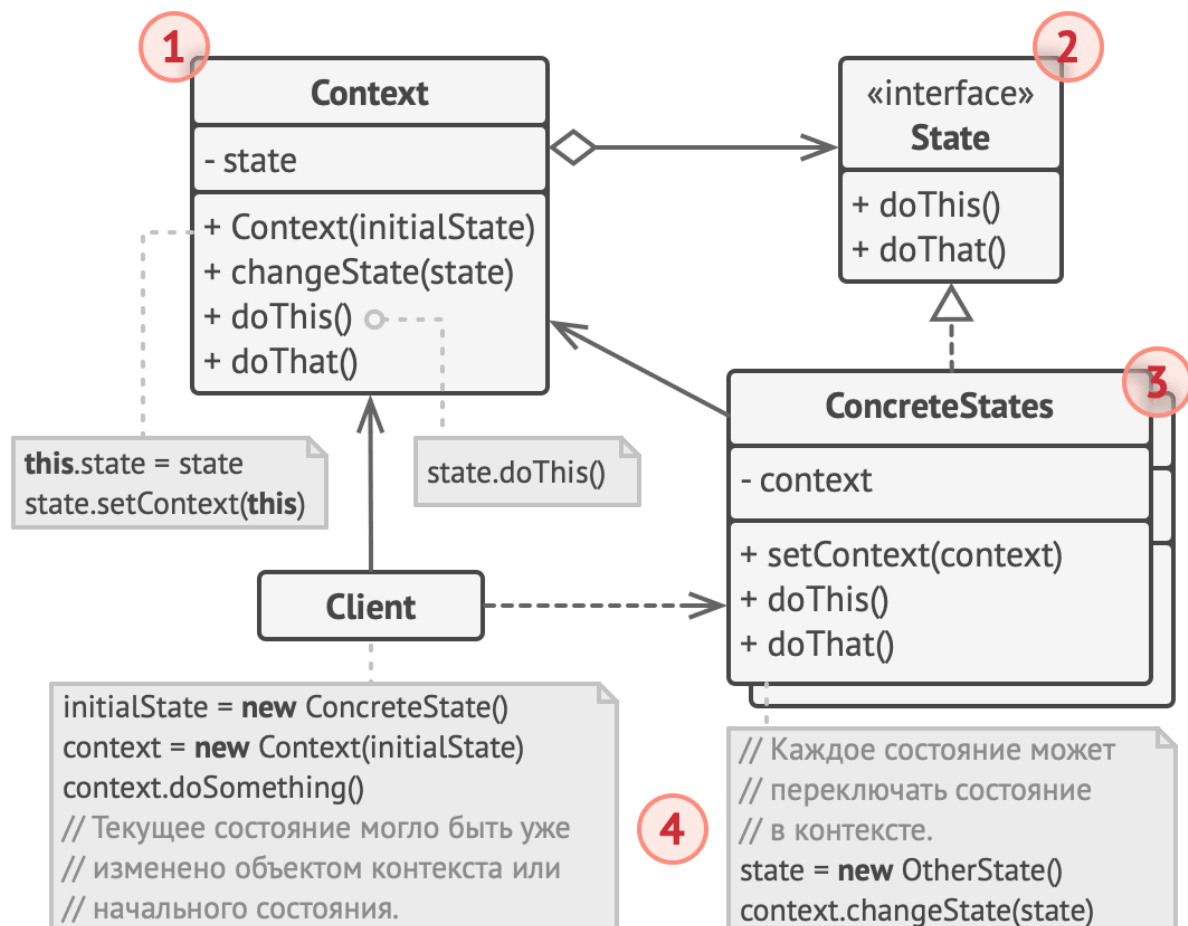
 **Преимущества:**

✓ 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:**

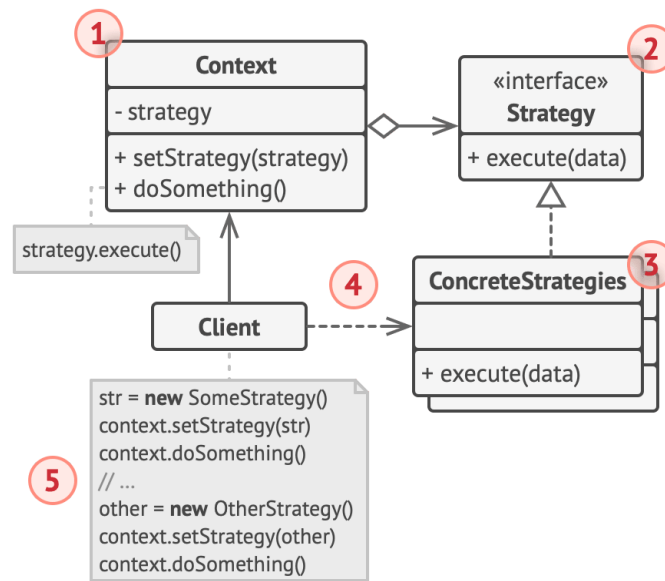
- ✓ Сортировка товаров (по цене, рейтингу).
- ✓ Оплата (PayPal, CreditCard, Crypto).

 **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:**

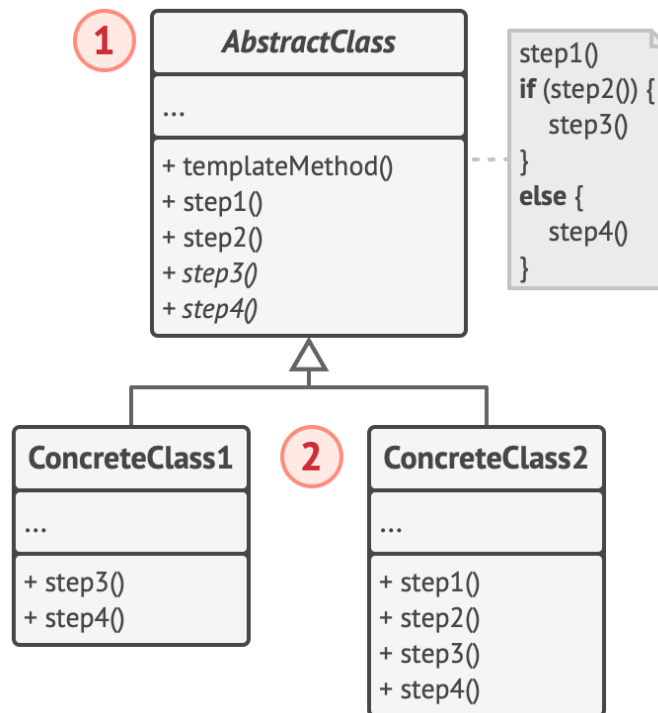
- ✓ Фреймворки (например, `React.Component` с `render()` ).
- ✓ Алгоритмы приготовления напитков (кофе/чай).

 **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:

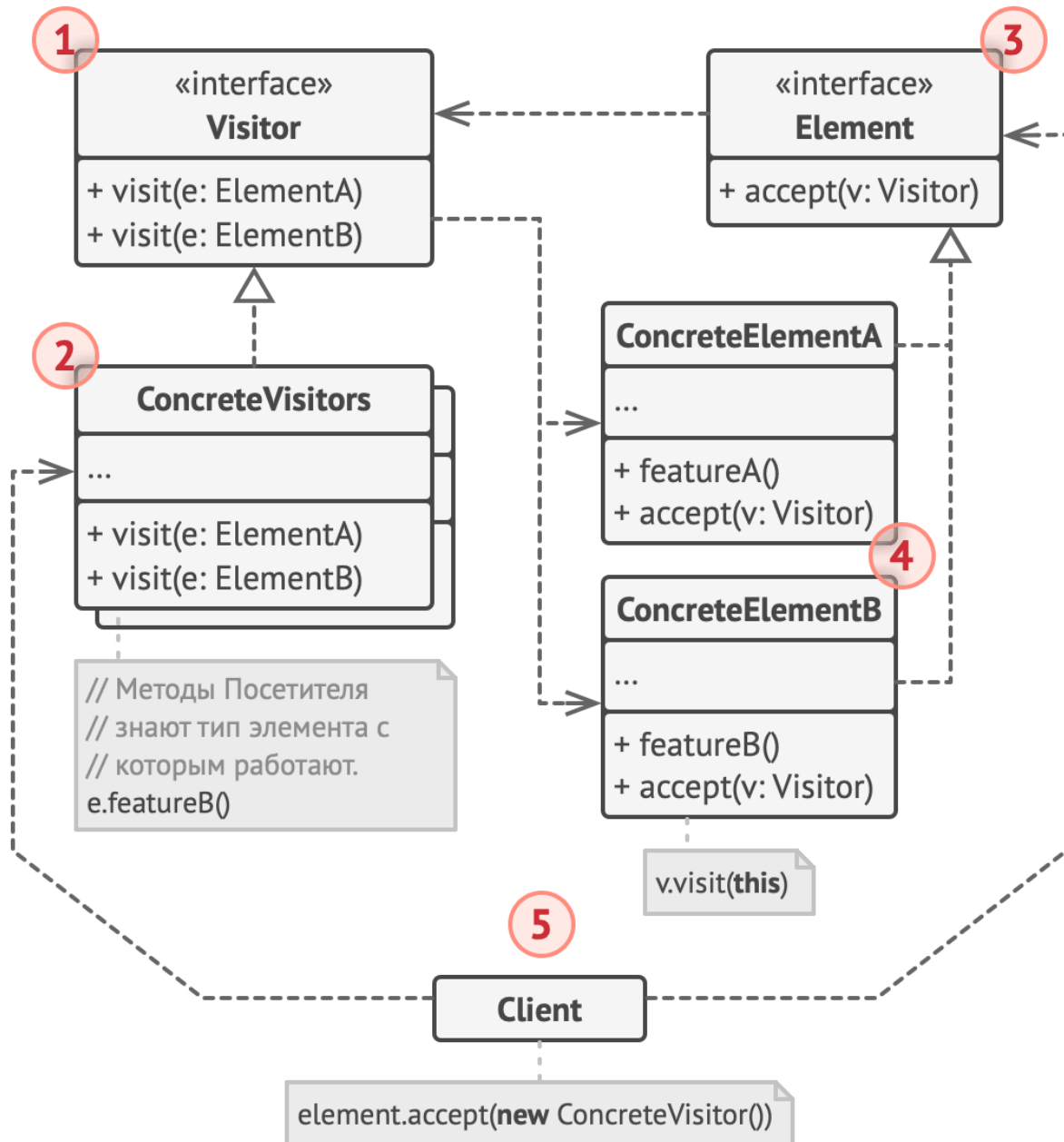
- ✓ Обход AST (абстрактного синтаксического дерева).
- ✓ Экспорт данных в XML/JSON.

#### Преимущества:

- ✓ Отделение логики операций от структуры объектов.
- ✓ Удобен для сложных иерархий (например, компиляторы).

#### Недостатки:

- ✗ Нарушает инкапсуляцию (требуется `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);
}

```



## Сравнение в одной таблице

Паттерн	Основное назначение	Когда использовать	Ключевое преимущество
---------	---------------------	--------------------	-----------------------

<b>Chain of Responsibility</b> (Цепочка обязанностей)	Последовательная обработка запросов	Когда несколько объектов могут обработать запрос	Снижение связанности между отправителем и получателем
<b>Command</b> (Команда)	Инкапсуляция действий	Для параметризации и постановки операций в очередь	Поддержка отмены, логирования и очередей
<b>Iterator</b> (Итератор)	Обход коллекций	Когда нужен доступ к элементам без раскрытия структуры	Единый интерфейс для разных коллекций
<b>Mediator</b> (Посредник)	Централизованное взаимодействие	При сложных взаимодействиях объектов	Уменьшает зависимости между объектами
<b>Memento</b> (Снимок)	Сохранение состояния	Для реализации механизма отмены	Сохраняет инкапсуляцию состояния объекта
<b>Observer</b> (Наблюдатель)	Уведомление о событиях	Когда изменения состояния влияют на многие объекты	Поддержка отношений "один-ко-многим"
<b>State</b> (Состояние)	Изменение поведения	Когда поведение зависит от состояния	Упрощает условную логику
<b>Strategy</b> (Стратегия)	Инкапсуляция алгоритмов	Для взаимозаменяемых алгоритмов	Возможность переключения алгоритмов в runtime
<b>Template Method</b> (Шаблонный метод)	Структурирование алгоритма	Для определения скелета с настраиваемыми шагами	Повторное использование кода с точками гибкости
<b>Visitor</b> (Посетитель)	Расширение операций	Для добавления операций без изменения классов	Разделяет алгоритмы и структуру объектов





## Как выбрать паттерн?

1. **Chain of Responsibility** → Когда **несколько объектов могут обработать запрос**, но неизвестно, кто именно
2. **Command** → Когда нужно **инкапсулировать действие в объект** (например, для отмены операций)
3. **Iterator** → Когда нужно **последовательно обходить элементы коллекции**, не раскрывая её структуру
4. **Mediator** → Когда **много объектов общаются между собой**, и нужно убрать прямые зависимости
5. **Memento** → Когда нужно **сохранять и восстанавливать состояние объекта** (например, для "Undo")
6. **Observer** → Когда **одни объекты должны реагировать на изменения других** (событийная модель)
7. **State** → Когда **поведение объекта зависит от его состояния** и меняется динамически
8. **Strategy** → Когда **нужно выбирать алгоритм на лету**
9. **Template Method** → Когда **есть общий алгоритм, но шаги могут отличаться**
10. **Visitor** → Когда **нужно добавить операции к объектам, не меняя их классы**



## Как запомнить?

1. **Chain of Responsibility** → **Официанты в ресторане** (просят одного, но кто-то один в итоге принесёт)
2. **Command** → **Заказ в кафе** (официант передаёт повару «объект-запрос», а не кричит на кухню)
3. **Iterator** → **Путешествие гида по музею** (ходит по залам, но не знает, как они устроены)
4. **Mediator** → **Диспетчер такси** (водители и клиенты не звонят друг другу напрямую)
5. **Memento** → **Сохранение в игре** (F5 — сохранил, F9 — откатился)
6. **Observer** → **Подписка на YouTube** (автор выложил видео — все подписчики получили уведомление)
7. **State** → **Термометр** (вода может быть льдом, жидкостью или паром — поведение зависит от состояния)
8. **Strategy** → **Навигатор** (можно выбрать маршрут «быстрый», «короткий» или «экономный»)
9. **Template Method** → **Готовка борща** (общий алгоритм: «варим бульон → кладём овощи»), но свеклу можно жарить или нет)
10. **Visitor** → **Экскурсовод в зоопарке** (Он рассказывает о львах, зебрах и обезьянах, но не меняет самих животных.)

### ▼ Когда использовать Chain of Responsibility

- У вас несколько обработчиков для одного запроса
- Вы не знаете, какой обработчик должен обработать запрос
- Нужно разделить отправителя и получателя
- Порядок обработки важен

### ▼ Когда использовать Command

- Нужно параметризовать объекты операциями

- Требуется ставить операции в очередь, логировать или отменять
- Необходима поддержка обратных вызовов

#### ▼ Когда использовать **Iterator**

- Нужен доступ к элементам коллекции без раскрытия структуры
- Требуется единый интерфейс обхода для разных коллекций
- Необходимо несколько одновременных обходов

#### ▼ Когда использовать **Mediator**

- Многие объекты взаимодействуют сложным образом
- Нужно уменьшить зависимости между объектами
- Требуется централизованное управление

#### ▼ Когда использовать **Memento**

- Требуется сохранить состояние объекта для возможного восстановления
- Прямой доступ раскроет детали реализации
- Требуется реализовать механизм отмены действий

#### ▼ Когда использовать **Observer**

- Изменения одного объекта влияют на другие
- Неизвестно, сколько объектов нужно обновить
- Требуется слабая связанность между объектами

#### ▼ Когда использовать **State**

- Поведение объекта зависит от его состояния
- Поведение управляется сложными условиями
- Переходы между состояниями четко определены

#### ▼ Когда использовать Стратегию (**Strategy**)

- Нужны разные варианты алгоритма
- Хочется скрыть сложные детали алгоритмов
- Требуется переключать алгоритмы во время выполнения

#### ▼ Когда использовать **Template Method**

- Алгоритм содержит изменяемые шаги
- Нужно избежать дублирования кода
- Требуется контроль над переопределением шагов

#### ▼ Когда использовать Visitor

- Нужно выполнить операцию над всеми элементами сложной структуры
- Требуется добавлять операции без изменения классов элементов
- Операциям нужен доступ к скрытым данным