Поведенческие паттерны. Часть 2

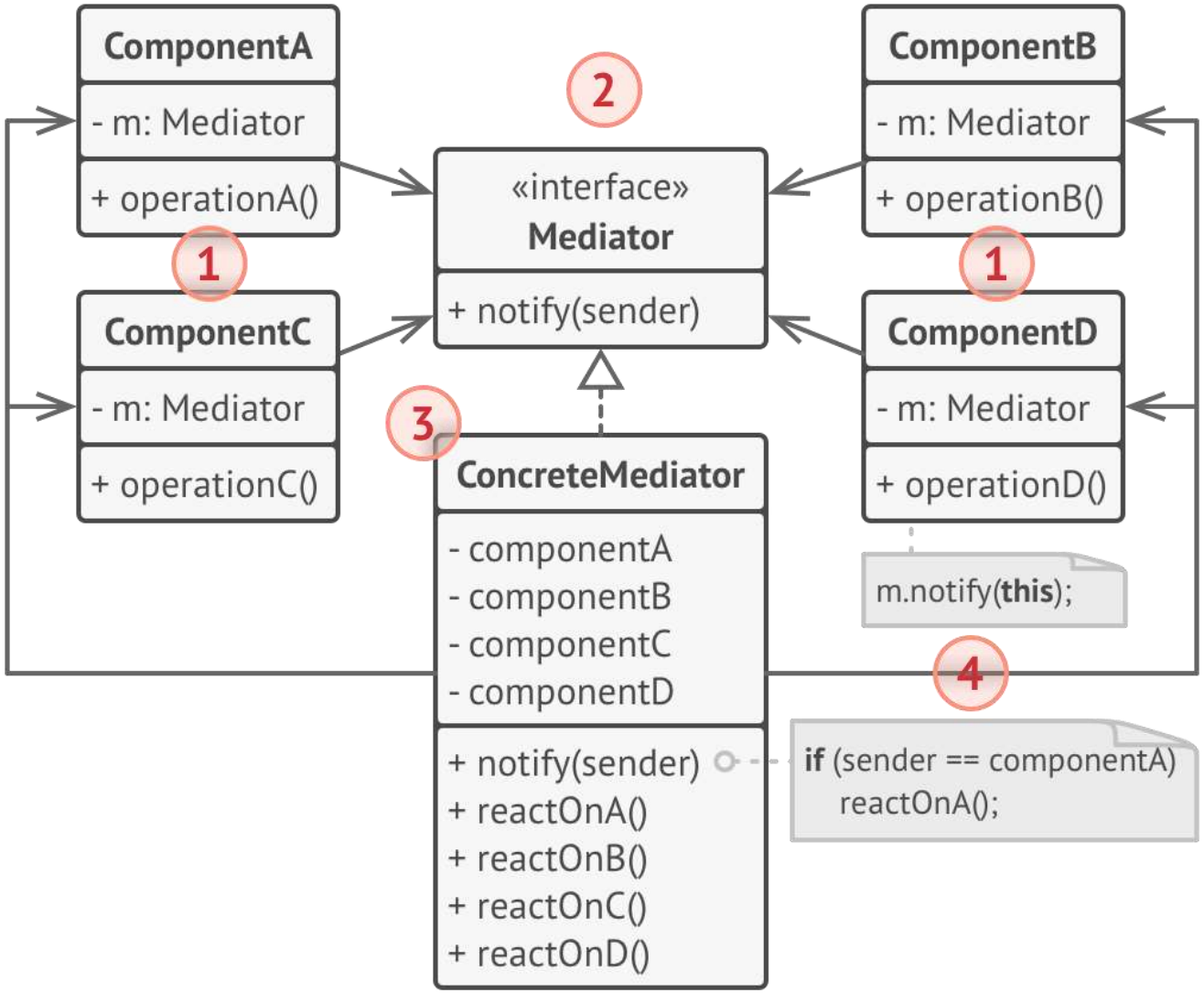
Посредник

Позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

Когда применять?

- Когда имеется множество взаимосвязанных объектов, связи между которыми сложны и запутаны.
- Когда необходимо повторно использовать объект, однако повторное использование затруднено в силу сильных связей с другими объектами.

Структура


ComponentA
- m: Mediator
+ operationA()
ComponentB
- m: Mediator
+ operationB()
2
«interface»
Mediator
+ notify(sender)
1
1
ComponentC
- m: Mediator
+ operationC()
ComponentD
- m: Mediator
+ operationD()
3
ConcreteMediator
- componentA
- componentB
- componentC
- componentD
+ notify(sender)
+ reactOnA()
+ reactOnB()
+ reactOnC()
+ reactOnD()
m.notify(this);
4
if (sender == componentA)
reactOnA();

Пример

```
// Интерфейс Посредника предоставляет метод, используемый компонентами для
// уведомления посредника о различных событиях. Посредник может реагировать
// на эти события  и передавать исполнение другим компонентам.
public interface IMediator
{
    void Notify(object sender, string ev);
}
```

Пример

```
// Базовый Компонент обеспечивает базовую функциональность хранения
// экземпляра посредника внутри объектов компонентов.
class BaseComponent
{
    protected IMediator _mediator;

    public BaseComponent(IMediator mediator = null)
    {
        this._mediator = mediator;
    }

    public void SetMediator(IMediator mediator)
    {
        this._mediator = mediator;
    }
}
```

Пример

```
// Конкретные Компоненты реализуют различную функциональность. Они не
// зависят от других компонентов. Они также не зависят от каких-либо
// конкретных классов посредников.
class Component1 : BaseComponent
{
    public void DoA()
    {
        Console.WriteLine("Component 1 does A.");

        this._mediator.Notify(this, "A");
    }

    public void DoB()
    {
        Console.WriteLine("Component 1 does B.");

        this._mediator.Notify(this, "B");
    }
}
```

Пример

```
class Component2 : BaseComponent
{
    public void DoC()
    {
        Console.WriteLine("Component 2 does C.");

        this._mediator.Notify(this, "C");
    }

    public void DoD()
    {
        Console.WriteLine("Component 2 does D.");

        this._mediator.Notify(this, "D");
    }
}
```

Пример

```
// Конкретные Посредники реализуют совместное поведение, координируя
// отдельные компоненты.
class ConcreteMediator : IMediator
{
    private Component1 _component1;

    private Component2 _component2;

    public ConcreteMediator(Component1 component1, Component2 component2)
    {
        this._component1 = component1;
        this._component1.SetMediator(this);
        this._component2 = component2;
        this._component2.SetMediator(this);
    }

    public void Notify(object sender, string ev)
    {
        if (ev == "A")
        {
            Console.WriteLine("Mediator reacts on A and triggers folowing operations:");
            this._component2.DoC();
        }
        if (ev == "D")
        {
            Console.WriteLine("Mediator reacts on D and triggers following operations:");
            this._component1.DoB();
            this._component2.DoC();
        }
    }
}
```

Пример

```
// Конкретные Посредники реализуют совместное поведение, координируя
// отдельные компоненты.
class ConcreteMediator : IMediator
{
    private Component1 _component1;

    private Component2 _component2;

    public ConcreteMediator(Component1 component1, Component2 component2)
    {
        this._component1 = component1;
        this._component1.SetMediator(this);
        this._component2 = component2;
        this._component2.SetMediator(this);
    }

    public void Notify(object sender, string ev)
    {
        if (ev == "A")
        {
            Console.WriteLine("Mediator reacts on A and triggers folowing operations:");
            this._component2.DoC();
        }
        if (ev == "D")
        {
            Console.WriteLine("Mediator reacts on D and triggers following operations:");
            this._component1.DoB();
            this._component2.DoC();
        }
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        // Клиентский код.
        Component1 component1 = new Component1();
        Component2 component2 = new Component2();
        new ConcreteMediator(component1, component2);

        Console.WriteLine("Client triggets operation A.");
        component1.DoA();

        Console.WriteLine();

        Console.WriteLine("Client triggers operation D.");
        component2.DoD();
    }
}
```

```
Client triggers operation A.
Component 1 does A.
Mediator reacts on A and triggers following operations:
Component 2 does C.

Client triggers operation D.
Component 2 does D.
Mediator reacts on D and triggers following operations:
Component 1 does B.
Component 2 does C.
```

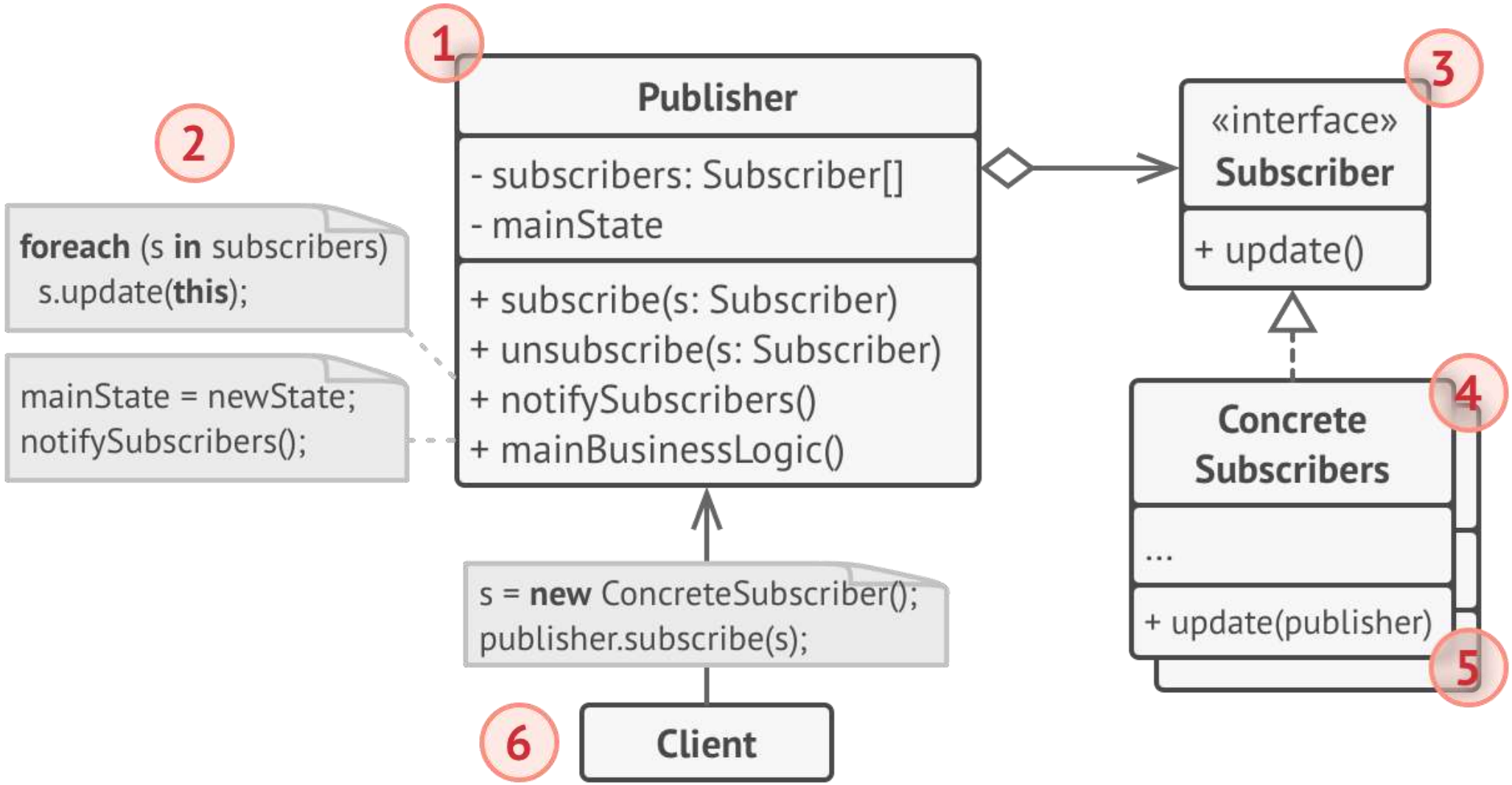
Наблюдатель

Создает механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Когда применять?

- Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях.
- Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения, другая получает сообщения и реагирует на них.

Структура


1
Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
2
foreach (s in subscribers)
s.update(this);
mainState = newState;
notifySubscribers();
3
«interface»
Subscriber
+ update()
4
Concrete
Subscribers
...
+ update(publisher)
5
s = new ConcreteSubscriber();
publisher.subscribe(s);
6
Client

Пример

```
public interface IObserver
{
    // Получает обновление от издателя
    void Update(ISubject subject);
}
```

```
public interface ISubject
{
    // Присоединяет наблюдателя к издателю.
    void Attach(IObserver observer);

    // Отсоединяет наблюдателя от издателя.
    void Detach(IObserver observer);

    // Уведомляет всех наблюдателей о событии.
    void Notify();
}
```

Пример

```
// Издатель владеет некоторым важным состоянием и оповещает наблюдателей о
// его изменениях.
public class Subject : ISubject
{
    // Для удобства в этой переменной хранится состояние Издателя,
    // необходимое всем подписчикам.
    public int State { get; set; } = -0;

    // Список подписчиков. В реальной жизни список подписчиков может
    // храниться в более подробном виде (классифицируется по типу события и
    // т.д.)
    private List<IObserver> _observers = new List<IObserver>();

    // Методы управления подпиской.
    public void Attach(IObserver observer)
    {
        Console.WriteLine("Subject: Attached an observer.");
        this._observers.Add(observer);
    }

    public void Detach(IObserver observer)
    {
        this._observers.Remove(observer);
        Console.WriteLine("Subject: Detached an observer.");
    }
```

Пример

```
    // Запуск обновления в каждом подписчике.
    public void Notify()
    {
        Console.WriteLine("Subject: Notifying observers...");

        foreach (var observer in _observers)
        {
            observer.Update(this);
        }
    }

    // Обычно логика подписки – только часть того, что делает Издатель.
    // Издатели часто содержат некоторую важную бизнес-логику, которая
    // запускает метод уведомления всякий раз, когда должно произойти что-то
    // важное (или после этого).
    public void SomeBusinessLogic()
    {
        Console.WriteLine("\nSubject: I'm doing something important.");
        this.State = new Random().Next(0, 10);

        Thread.Sleep(15);

        Console.WriteLine("Subject: My state has just changed to: " + this.State);
        this.Notify();
    }
}
```

Пример

```
// Конкретные Наблюдатели реагируют на обновления, выпущенные Издателем, к
// которому они прикреплены.
class ConcreteObserverA : IObserver
{
    public void Update(ISubject subject)
    {
        if ((subject as Subject).State < 3)
        {
            Console.WriteLine("ConcreteObserverA: Reacted to the event.");
        }
    }
}
```

```
class ConcreteObserverB : IObserver
{
    public void Update(ISubject subject)
    {
        if ((subject as Subject).State == 0 || (subject as Subject).State >= 2)
        {
            Console.WriteLine("ConcreteObserverB: Reacted to the event.");
        }
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        // Клиентский код.
        var subject = new Subject();
        var observerA = new ConcreteObserverA();
        subject.Attach(observerA);

        var observerB = new ConcreteObserverB();
        subject.Attach(observerB);

        subject.SomeBusinessLogic();
        subject.SomeBusinessLogic();

        subject.Detach(observerB);

        subject.SomeBusinessLogic();
    }
}
```

```
Subject: Attached an observer.
Subject: Attached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 2
Subject: Notifying observers...
ConcreteObserverA: Reacted to the event.
ConcreteObserverB: Reacted to the event.

Subject: I'm doing something important.
Subject: My state has just changed to: 1
Subject: Notifying observers...
ConcreteObserverA: Reacted to the event.
Subject: Detached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 5
Subject: Notifying observers...
```

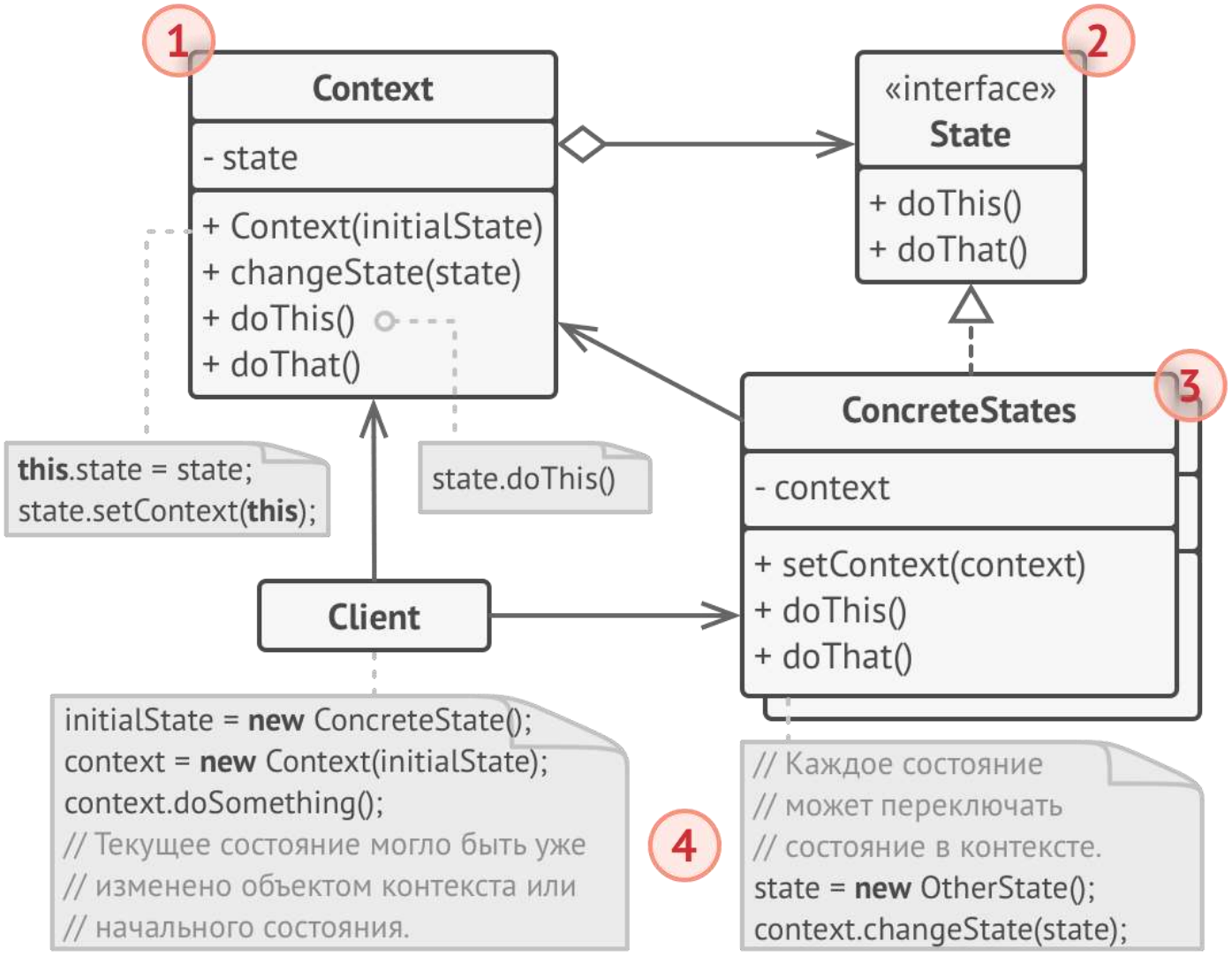
Состояние

Позволяет объектам менять поведение в зависимости от своего состояния. Извне создается впечатление, что изменился класс объекта.

Когда применять?

- Когда поведение объекта должно зависеть от его состояния и может изменяться динамически во время выполнения.

Структура


1
Context
- state
+ Context(initialState)
+ changeState(state)
+ doThis()
+ doThat()
2
«interface»
State
+ doThis()
+ doThat()
3
ConcreteStates
- context
+ setContext(context)
+ doThis()
+ doThat()
this.state = state;
state.setContext(this);
state.doThis()
Client
initialState = new ConcreteState();
context = new Context(initialState);
context.doSomething();
// Текущее состояние могло быть уже
// изменено объектом контекста или
// начального состояния.
4
// Каждое состояние
// может переключать
// состояние в контексте.
state = new OtherState();
context.changeState(state);

Пример

```
// Базовый класс Состояния объявляет методы, которые должны реализовать все
// Конкретные Состояния, а также предоставляет обратную ссылку на объект
// Контекст, связанный с Состоянием. Эта обратная ссылка может
// использоваться Состояниями для передачи Контекста другому Состоянию.
abstract class State
{
    protected Context _context;

    public void SetContext(Context context)
    {
        this._context = context;
    }

    public abstract void Handle1();

    public abstract void Handle2();
}
```

Пример

```
// Контекст определяет интерфейс, представляющий интерес для клиентов. Он
// также хранит ссылку на экземпляр подкласса Состояния, который отображает
// текущее состояние Контекста.
class Context
{
    // Ссылка на текущее состояние Контекста.
    private State _state = null;

    public Context(State state)
    {
        this.TransitionTo(state);
    }

    // Контекст позволяет изменять объект Состояния во время выполнения.
    public void TransitionTo(State state)
    {
        Console.WriteLine($"Context: Transition to {state.GetType().Name}.");
        this._state = state;
        this._state.SetContext(this);
    }

    // Контекст делегирует часть своего поведения текущему объекту
    // Состояния.
    public void Request1()
    {
        this._state.Handle1();
    }

    public void Request2()
    {
        this._state.Handle2();
    }
}
```

Пример

```
// Конкретные Состояния реализуют различные модели поведения, связанные с
// состоянием Контекста.
class ConcreteStateA : State
{
    public override void Handle1()
    {
        Console.WriteLine("ConcreteStateA handles request1.");
        Console.WriteLine("ConcreteStateA wants to change the state of the context.");
        this._context.TransitionTo(new ConcreteStateB());
    }

    public override void Handle2()
    {
        Console.WriteLine("ConcreteStateA handles request2.");
    }
}
```

Пример

```
class ConcreteStateB : State
{
    public override void Handle1()
    {
        Console.Write("ConcreteStateB handles request1.");
    }

    public override void Handle2()
    {
        Console.WriteLine("ConcreteStateB handles request2.");
        Console.WriteLine("ConcreteStateB wants to change the state of the context.");
        this._context.TransitionTo(new ConcreteStateA());
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        // Клиентский код.
        var context = new Context(new ConcreteStateA());
        context.Request1();
        context.Request2();
    }
}
```

```
Context: Transition to ConcreteStateA.
ConcreteStateA handles request1.
ConcreteStateA wants to change the state of the context.
Context: Transition to ConcreteStateB.
ConcreteStateB handles request2.
ConcreteStateB wants to change the state of the context.
Context: Transition to ConcreteStateA.
```

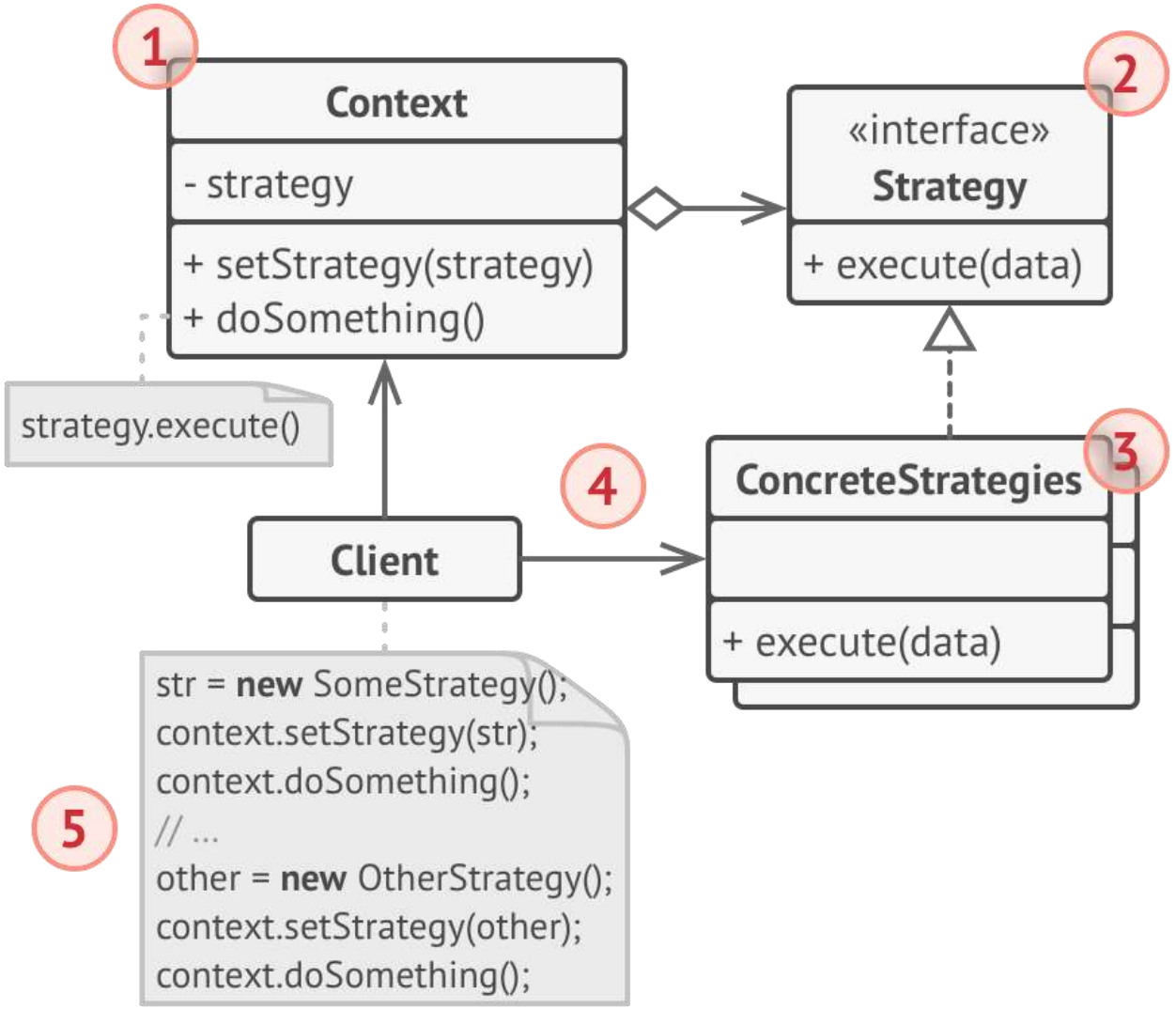
Стратегия

Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс. После чего, алгоритмы можно взаимозаменять прямо во время исполнения программы.

Когда применять?

- Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий.
- Когда необходимо менять поведение объектов на стадии выполнения программы.
- Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации.

Структура


1
Context
- strategy
+ setStrategy(strategy)
+ doSomething()
2
«interface»
Strategy
+ execute(data)
strategy.execute()
4
3
ConcreteStrategies
Client
+ execute(data)
5
str = new SomeStrategy();
context.setStrategy(str);
context.doSomething();
// ...
other = new OtherStrategy();
context.setStrategy(other);
context.doSomething();

Пример

```
// Интерфейс Стратегии объявляет операции, общие для всех поддерживаемых
// версий некоторого алгоритма.
//
// Контекст использует этот интерфейс для вызова алгоритма, определённого
// Конкретными Стратегиями.
public interface IStrategy
{
    object DoAlgorithm(object data);
}
```

Пример

```
// Контекст определяет интерфейс, представляющий интерес для клиентов.
class Context
{
    // Контекст хранит ссылку на один из объектов Стратегии. Контекст не
    // знает конкретного класса стратегии. Он должен работать со всеми
    // стратегиями через интерфейс Стратегии.
    private IStrategy _strategy;

    public Context()
    { }

    // Обычно Контекст принимает стратегию через конструктор, а также
    // предоставляет сеттер для её изменения во время выполнения.
    public Context(IStrategy strategy)
    {
        this._strategy = strategy;
    }

    // Обычно Контекст позволяет заменить объект Стратегии во время
    // выполнения.
    public void SetStrategy(IStrategy strategy)
    {
        this._strategy = strategy;
    }
```

Пример

```
    // Вместо того, чтобы самостоятельно реализовывать множественные версии
    // алгоритма, Контекст делегирует некоторую работу объекту Стратегии.
    public void DoSomeBusinessLogic()
    {
        Console.WriteLine("Context: Sorting data using the strategy (not sure how it'll do it)");
        var result = this._strategy.DoAlgorithm(new List<string> { "a", "b", "c", "d", "e" });

        string resultStr = string.Empty;
        foreach (var element in result as List<string>)
        {
            resultStr += element + ",";
        }

        Console.WriteLine(resultStr);
    }
}
```

Пример

```
    // Вместо того, чтобы самостоятельно реализовывать множественные версии
    // алгоритма, Контекст делегирует некоторую работу объекту Стратегии.
    public void DoSomeBusinessLogic()
    {
        Console.WriteLine("Context: Sorting data using the strategy (not sure how it'll do it)");
        var result = this._strategy.DoAlgorithm(new List<string> { "a", "b", "c", "d", "e" });

        string resultStr = string.Empty;
        foreach (var element in result as List<string>)
        {
            resultStr += element + ",";
        }

        Console.WriteLine(resultStr);
    }
}
```

Пример

```
// Конкретные Стратегии реализуют алгоритм, следуя базовому интерфейсу
// Стратегии. Этот интерфейс делает их взаимозаменяемыми в Контексте.
class ConcreteStrategyA : IStrategy
{
    public object DoAlgorithm(object data)
    {
        var list = data as List<string>;
        list.Sort();

        return list;
    }
}
```

Пример

```
class ConcreteStrategyB : IStrategy
{
    public object DoAlgorithm(object data)
    {
        var list = data as List<string>;
        list.Sort();
        list.Reverse();

        return list;
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        // Клиентский код выбирает конкретную стратегию и передаёт её в
        // контекст. Клиент должен знать о различиях между стратегиями,
        // чтобы сделать правильный выбор.
        var context = new Context();

        Console.WriteLine("Client: Strategy is set to normal sorting.");
        context.SetStrategy(new ConcreteStrategyA());
        context.DoSomeBusinessLogic();

        Console.WriteLine();

        Console.WriteLine("Client: Strategy is set to reverse sorting.");
        context.SetStrategy(new ConcreteStrategyB());
        context.DoSomeBusinessLogic();
    }
}
```

```
Client: Strategy is set to normal sorting.
Context: Sorting data using the strategy (not sure how it'll do it)
a,b,c,d,e

Client: Strategy is set to reverse sorting.
Context: Sorting data using the strategy (not sure how it'll do it)
e,d,c,b,a
```

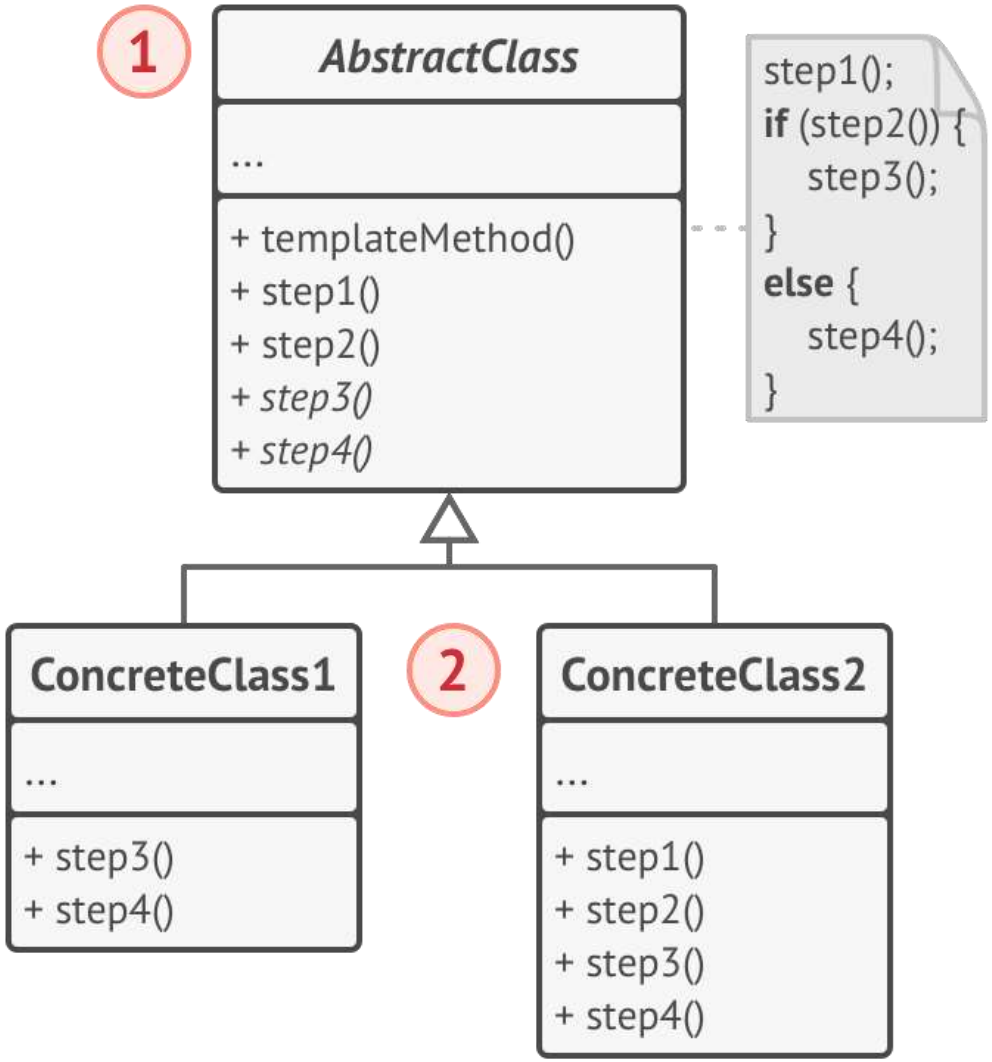
Шаблонный метод

Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

Когда применять?

- Когда планируется, что в будущем подклассы должны будут переопределять различные этапы алгоритма без изменения его структуры.
- Когда в классах, реализующих схожий алгоритм, происходит дублирование кода. Вынесение общего кода в шаблонный метод уменьшит его дублирование в подклассах.

Структура


1
AbstractClass
...
+ templateMethod()
+ step1()
+ step2()
+ step3()
+ step4()
step1();
if (step2()) {
step3();
}
else {
step4();
}
2
ConcreteClass1
...
+ step3()
+ step4()
ConcreteClass2
...
+ step1()
+ step2()
+ step3()
+ step4()

Пример

```
// Абстрактный Класс определяет шаблонный метод, содержащий скелет
// некоторого алгоритма, состоящего из вызовов (обычно) абстрактных
// примитивных операций.
//
// Конкретные подклассы должны реализовать эти операции, но оставить сам
// шаблонный метод без изменений.
abstract class AbstractClass
{
    // Шаблонный метод определяет скелет алгоритма.
    public void TemplateMethod()
    {
        this.BaseOperation1();
        this.RequiredOperations1();
        this.BaseOperation2();
        this.Hook1();
        this.RequiredOperation2();
        this.BaseOperation3();
        this.Hook2();
    }
```

Пример

```
// Эти операции уже имеют реализации.
protected void BaseOperation1()
{
    Console.WriteLine("AbstractClass says: I am doing the bulk of the work");
}

protected void BaseOperation2()
{
    Console.WriteLine("AbstractClass says: But I let subclasses override some operations");
}

protected void BaseOperation3()
{
    Console.WriteLine("AbstractClass says: But I am doing the bulk of the work anyway");
}
```

Пример

```
    // А эти операции должны быть реализованы в подклассах.
    protected abstract void RequiredOperations1();

    protected abstract void RequiredOperation2();

    // Это «хуки». Подклассы могут переопределять их, но это не обязательно,
    // поскольку у хуков уже есть стандартная (но пустая) реализация. Хуки
    // предоставляют дополнительные точки расширения в некоторых критических
    // местах алгоритма.
    protected virtual void Hook1() { }

    protected virtual void Hook2() { }
}
```

Пример

```
// Конкретные классы должны реализовать все абстрактные операции базового
// класса. Они также могут переопределить некоторые операции с реализацией
// по умолчанию.
class ConcreteClass1 : AbstractClass
{
    protected override void RequiredOperations1()
    {
        Console.WriteLine("ConcreteClass1 says: Implemented Operation1");
    }

    protected override void RequiredOperation2()
    {
        Console.WriteLine("ConcreteClass1 says: Implemented Operation2");
    }
}
```

Пример

```
// Обычно конкретные классы переопределяют только часть операций базового
// класса.
class ConcreteClass2 : AbstractClass
{
    protected override void RequiredOperations1()
    {
        Console.WriteLine("ConcreteClass2 says: Implemented Operation1");
    }

    protected override void RequiredOperation2()
    {
        Console.WriteLine("ConcreteClass2 says: Implemented Operation2");
    }

    protected override void Hook1()
    {
        Console.WriteLine("ConcreteClass2 says: Overridden Hook1");
    }
}
```

Пример

```
class Client
{
    // Клиентский код вызывает шаблонный метод для выполнения алгоритма.
    // Клиентский код не должен знать конкретный класс объекта, с которым
    // работает, при условии, что он работает с объектами через интерфейс их
    // базового класса.
    public static void ClientCode(AbstractClass abstractClass)
    {
        // ...
        abstractClass.TemplateMethod();
        // ...
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Same client code can work with different subclasses:");

        Client.ClientCode(new ConcreteClass1());

        Console.Write("\n");

        Console.WriteLine("Same client code can work with different subclasses:");
        Client.ClientCode(new ConcreteClass2());
    }
}
```

Пример

```
Same client code can work with different subclasses:
AbstractClass says: I am doing the bulk of the work
ConcreteClass1 says: Implemented Operation1
AbstractClass says: But I let subclasses override some operations
ConcreteClass1 says: Implemented Operation2
AbstractClass says: But I am doing the bulk of the work anyway

Same client code can work with different subclasses:
AbstractClass says: I am doing the bulk of the work
ConcreteClass2 says: Implemented Operation1
AbstractClass says: But I let subclasses override some operations
ConcreteClass2 says: Overridden Hook1
ConcreteClass2 says: Implemented Operation2
AbstractClass says: But I am doing the bulk of the work anyway
```

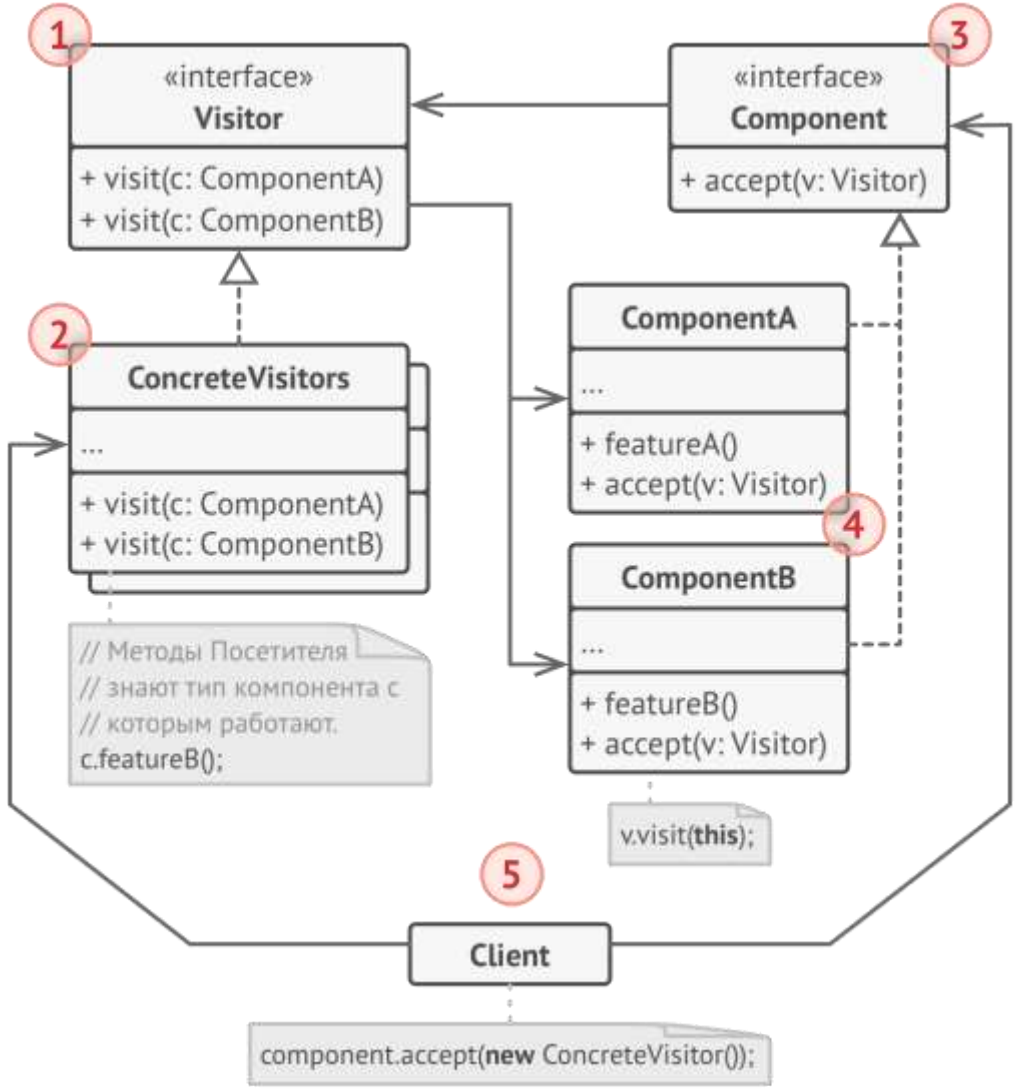
Посетитель

Позволяет создавать новые операции, не меняя классы объектов, над которыми эти операции могут выполняться.

Когда применять?

- Когда нужно выполнить операцию над всеми элементами сложной структуры объектов.
- Когда над объектами сложной структуры объектов надо выполнять некоторые, не связанные между собой операций, но нет смысла «засорять» классы такими операциями.
- Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии.

Структура


1
«interface»
Visitor
+ visit(c: ComponentA)
+ visit(c: ComponentB)
3
«interface»
Component
+ accept(v: Visitor)
2
ConcreteVisitors
...
+ visit(c: ComponentA)
+ visit(c: ComponentB)
// Методы Посетителя
// знают тип компонента с
// которым работают.
c.featureB();
ComponentA
...
+ featureA()
+ accept(v: Visitor)
4
ComponentB
...
+ featureB()
+ accept(v: Visitor)
v.visit(this);
5
Client
component.accept(new ConcreteVisitor());

Пример

```
// Интерфейс Посетителя объявляет набор методов посещения, соответствующих
// классам компонентов. Сигнатура метода посещения позволяет посетителю
// определить конкретный класс компонента, с которым он имеет дело.
public interface IVisitor
{
    void VisitConcreteComponentA(ConcreteComponentA element);

    void VisitConcreteComponentB(ConcreteComponentB element);
}
```

```
// Интерфейс Компонента объявляет метод accept, который в качестве аргумента
// может получать любой объект, реализующий интерфейс посетителя.
public interface IComponent
{
    void Accept(IVisitor visitor);
}
```

Пример

```
// Каждый Конкретный Компонент должен реализовать метод Accept таким
// образом, чтобы он вызывал метод посетителя, соответствующий классу
// компонента.
public class ConcreteComponentA : IComponent
{
    // Обратите внимание, мы вызываем VisitConcreteComponentA, что
    // соответствует названию текущего класса. Таким образом мы позволяем
    // посетителю узнать, с каким классом компонента он работает.
    public void Accept(IVisitor visitor)
    {
        visitor.VisitConcreteComponentA(this);
    }

    // Конкретные Компоненты могут иметь особые методы, не объявленные в их
    // базовом классе или интерфейсе. Посетитель всё же может использовать
    // эти методы, поскольку он знает о конкретном классе компонента.
    public string ExclusiveMethodOfConcreteComponentA()
    {
        return "A";
    }
}
```

Пример

```
public class ConcreteComponentB : IComponent
{
    // То же самое здесь: VisitConcreteComponentB => ConcreteComponentB
    public void Accept(IVisitor visitor)
    {
        visitor.VisitConcreteComponentB(this);
    }

    public string SpecialMethodOfConcreteComponentB()
    {
        return "B";
    }
}
```

Пример

```
public class ConcreteComponentB : IComponent
{
    // То же самое здесь: VisitConcreteComponentB => ConcreteComponentB
    public void Accept(IVisitor visitor)
    {
        visitor.VisitConcreteComponentB(this);
    }

    public string SpecialMethodOfConcreteComponentB()
    {
        return "B";
    }
}
```

Пример

```
// Конкретные Посетители реализуют несколько версий одного и того же
// алгоритма, которые могут работать со всеми классами конкретных
// компонентов.
//
// Максимальную выгоду от паттерна Посетитель вы почувствуете, используя его
// со сложной структурой объектов, такой как дерево Компоновщика. В этом
// случае было бы полезно хранить некоторое промежуточное состояние
// алгоритма при выполнении методов посетителя над различными объектами
// структуры.
class ConcreteVisitor1 : IVisitor
{
    public void VisitConcreteComponentA(ConcreteComponentA element)
    {
        Console.WriteLine(element.ExclusiveMethodOfConcreteComponentA() + " + ConcreteVisitor1");
    }

    public void VisitConcreteComponentB(ConcreteComponentB element)
    {
        Console.WriteLine(element.SpecialMethodOfConcreteComponentB() + " + ConcreteVisitor1");
    }
}
```

Пример

```
class ConcreteVisitor2 : IVisitor
{
    public void VisitConcreteComponentA(ConcreteComponentA element)
    {
        Console.WriteLine(element.ExclusiveMethodOfConcreteComponentA() + " + ConcreteVisitor2");
    }

    public void VisitConcreteComponentB(ConcreteComponentB element)
    {
        Console.WriteLine(element.SpecialMethodOfConcreteComponentB() + " + ConcreteVisitor2");
    }
}
```

Пример

```
public class Client
{
    // Клиентский код может выполнять операции посетителя над любым набором
    // элементов, не выясняя их конкретных классов. Операция принятия
    // направляет вызов к соответствующей операции в объекте посетителя.
    public static void ClientCode(List<IComponent> components, IVisitor visitor)
    {
        foreach (var component in components)
        {
            component.Accept(visitor);
        }
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        List<IComponent> components = new List<IComponent>
        {
            new ConcreteComponentA(),
            new ConcreteComponentB()
        };

        Console.WriteLine("The client code works with all visitors via the base Visitor interface:");
        var visitor1 = new ConcreteVisitor1();
        Client.ClientCode(components,visitor1);

        Console.WriteLine();

        Console.WriteLine("It allows the same client code to work with different types of visitors:");
        var visitor2 = new ConcreteVisitor2();
        Client.ClientCode(components, visitor2);
    }
}
```

Пример

```
The client code works with all visitors via the base Visitor interface:
A + ConcreteVisitor1
B + ConcreteVisitor1

It allows the same client code to work with different types of visitors:
A + ConcreteVisitor2
B + ConcreteVisitor2
```

Спасибо за внимание!