Поведенческие паттерны. Часть 1

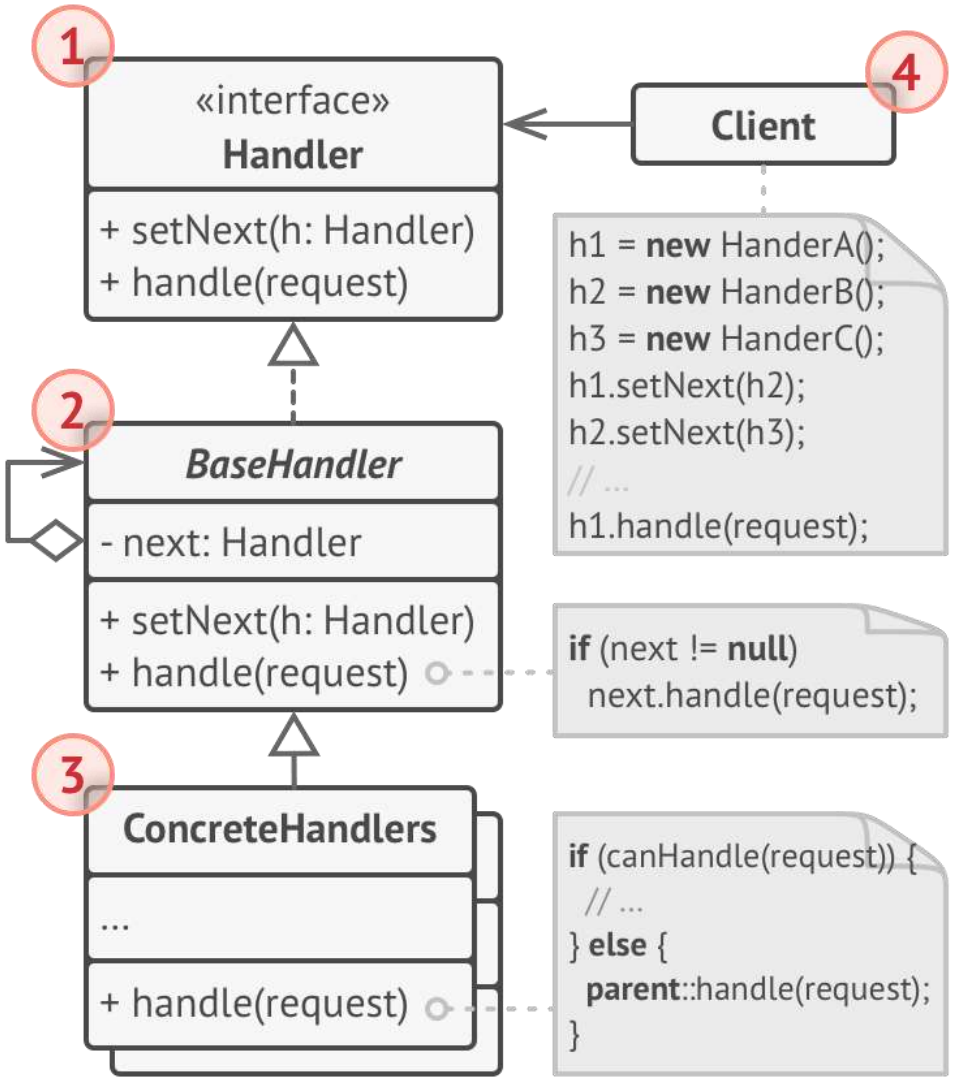
Цепочка обязанностей

Позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Когда применять?

- Когда имеется более одного объекта, который может обработать определенный запрос.
- Когда надо передать запрос на выполнение одному из нескольких объектов, точно не определяя, какому именно объекту.
- Когда набор объектов задается динамически.

Структура


1
«interface»
Handler
+ setNext(h: Handler)
+ handle(request)
4
Client
h1 = new HanderA();
h2 = new HanderB();
h3 = new HanderC();
h1.setNext(h2);
h2.setNext(h3);
// ...
h1.handle(request);
2
BaseHandler
- next: Handler
+ setNext(h: Handler)
+ handle(request)
if (next != null)
next.handle(request);
3
ConcreteHandlers
...
+ handle(request)
if (canHandle(request)) {
// ...
} else {
parent::handle(request);
}

Пример

```
// Интерфейс Обработчика объявляет метод построения цепочки обработчиков. Он
// также объявляет метод для выполнения запроса.
public interface IHandler
{
    IHandler SetNext(IHandler handler);

    object Handle(object request);
}
```

Пример

```
// Поведение цепочки по умолчанию может быть реализовано внутри базового
// класса обработчика.
abstract class AbstractHandler : IHandler
{
    private IHandler _nextHandler;

    public IHandler SetNext(IHandler handler)
    {
        this._nextHandler = handler;

        // Возврат обработчика отсюда позволит связать обработчики простым
        // способом, вот так:
        // monkey.SetNext(squirrel).SetNext(dog);
        return handler;
    }

    public virtual object Handle(object request)
    {
        if (this._nextHandler != null)
        {
            return this._nextHandler.Handle(request);
        }
        else
        {
            return null;
        }
    }
}
```

Пример

```
class MonkeyHandler : AbstractHandler
{
    public override object Handle(object request)
    {
        if ((request as string) == "Banana")
        {
            return $"Monkey: I'll eat the {request.ToString()}.\n";
        }
        else
        {
            return base.Handle(request);
        }
    }
}
```

```
class SquirrelHandler : AbstractHandler
{
    public override object Handle(object request)
    {
        if (request.ToString() == "Nut")
        {
            return $"Squirrel: I'll eat the {request.ToString()}.\n";
        }
        else
        {
            return base.Handle(request);
        }
    }
}
```

```
class DogHandler : AbstractHandler
{
    public override object Handle(object request)
    {
        if (request.ToString() == "MeatBall")
        {
            return $"Dog: I'll eat the {request.ToString()}.\n";
        }
        else
        {
            return base.Handle(request);
        }
    }
}
```

Пример

```
class Client
{
    // Обычно клиентский код приспособлен для работы с единственным
    // обработчиком. В большинстве случаев клиенту даже неизвестно, что этот
    // обработчик является частью цепочки.
    public static void ClientCode(AbstractHandler handler)
    {
        foreach (var food in new List<string> { "Nut", "Banana", "Cup of coffee" })
        {
            Console.WriteLine($"Client: Who wants a {food}?");

            var result = handler.Handle(food);

            if (result != null)
            {
                Console.Write($"   {result}");
            }
            else
            {
                Console.WriteLine($"   {food} was left untouched.");
            }
        }
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        // Другая часть клиентского кода создает саму цепочку.
        var monkey = new MonkeyHandler();
        var squirrel = new SquirrelHandler();
        var dog = new DogHandler();

        monkey.SetNext(squirrel).SetNext(dog);

        // Клиент должен иметь возможность отправлять запрос любому
        // обработчику, а не только первому в цепочке.
        Console.WriteLine("Chain: Monkey > Squirrel > Dog\n");
        Client.ClientCode(monkey);
        Console.WriteLine();

        Console.WriteLine("Subchain: Squirrel > Dog\n");
        Client.ClientCode(squirrel);
    }
}
```

```
Chain: Monkey > Squirrel > Dog

Client: Who wants a Nut?
   Squirrel: I'll eat the Nut.
Client: Who wants a Banana?
   Monkey: I'll eat the Banana.
Client: Who wants a Cup of coffee?
   Cup of coffee was left untouched.

Subchain: Squirrel > Dog

Client: Who wants a Nut?
   Squirrel: I'll eat the Nut.
Client: Who wants a Banana?
   Banana was left untouched.
Client: Who wants a Cup of coffee?
   Cup of coffee was left untouched.
```

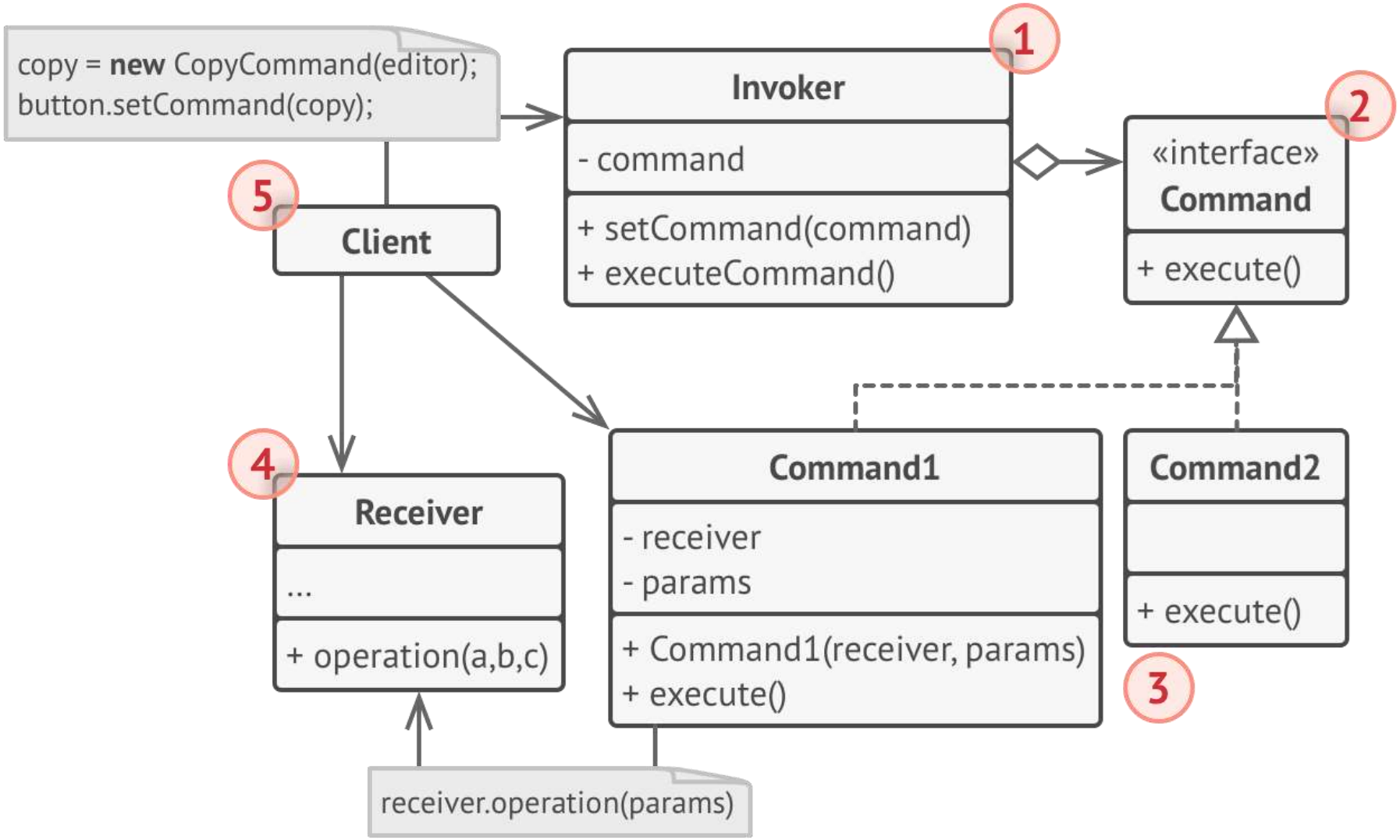
Команда

Преобразует запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

Когда применять?

- Когда необходимо обеспечить выполнение очереди запросов, выполнение их по расписанию или требуется передача по сети.
- Когда необходима возможность отмены действия.
- Когда необходимо параметризовать объекты выполняемым действием.

Структура


copy = new CopyCommand(editor);
button.setCommand(copy);
1
Invoker
- command
+ setCommand(command)
+ executeCommand()
2
«interface»
Command
+ execute()
5
Client
4
Receiver
...
+ operation(a,b,c)
Command1
- receiver
- params
+ Command1(receiver, params)
+ execute()
Command2
+ execute()
3
receiver.operation(params)

Пример

```
// Интерфейс Команды объявляет метод для выполнения команд.
public interface ICommand
{
    void Execute();
}
```

Пример

```
// Некоторые команды способны выполнять простые операции самостоятельно.
class SimpleCommand : ICommand
{
    private string _payload = string.Empty;

    public SimpleCommand(string payload)
    {
        this._payload = payload;
    }

    public void Execute()
    {
        Console.WriteLine($"SimpleCommand: See, I can do simple things like printing ({this._payload})");
    }
}
```

Пример

```
// Некоторые команды способны выполнять простые операции самостоятельно.
class SimpleCommand : ICommand
{
    private string _payload = string.Empty;

    public SimpleCommand(string payload)
    {
        this._payload = payload;
    }

    public void Execute()
    {
        Console.WriteLine($"SimpleCommand: See, I can do simple things like printing ({this._payload})");
    }
}
```

Пример

```
// Но есть и команды, которые делегируют более сложные операции другим
// объектам, называемым «получателями».
class ComplexCommand : ICommand
{
    private Receiver _receiver;

    // Данные о контексте, необходимые для запуска методов получателя.
    private string _a;

    private string _b;

    // Сложные команды могут принимать один или несколько объектов-
    // получателей вместе с любыми данными о контексте через конструктор.
    public ComplexCommand(Receiver receiver, string a, string b)
    {
        this._receiver = receiver;
        this._a = a;
        this._b = b;
    }

    // Команды могут делегировать выполнение любым методам получателя.
    public void Execute()
    {
        Console.WriteLine("ComplexCommand: Complex stuff should be done by a receiver object.");
        this._receiver.DoSomething(this._a);
        this._receiver.DoSomethingElse(this._b);
    }
}
```

Пример

```
// Классы Получателей содержат некую важную бизнес-логику. Они умеют
// выполнять все виды операций, связанных с выполнением запроса. Фактически,
// любой класс может выступать Получателем.
class Receiver
{
    public void DoSomething(string a)
    {
        Console.WriteLine($"Receiver: Working on ({a}.)");
    }

    public void DoSomethingElse(string b)
    {
        Console.WriteLine($"Receiver: Also working on ({b}.)");
    }
}
```

Пример

```
// Отправитель связан с одной или несколькими командами. Он отправляет
// запрос команде.
class Invoker
{
    private ICommand _onStart;

    private ICommand _onFinish;

    // Инициализация команд
    public void SetOnStart(ICommand command)
    {
        this._onStart = command;
    }

    public void SetOnFinish(ICommand command)
    {
        this._onFinish = command;
    }

    // Отправитель не зависит от классов конкретных команд и получателей.
    // Отправитель передаёт запрос получателю косвенно, выполняя команду.
    public void DoSomethingImportant()
    {
        Console.WriteLine("Invoker: Does anybody want something done before I begin?");
        if (this._onStart is ICommand)
        {
            this._onStart.Execute();
        }

        Console.WriteLine("Invoker: ...doing something really important...");

        Console.WriteLine("Invoker: Does anybody want something done after I finish?");
        if (this._onFinish is ICommand)
        {
            this._onFinish.Execute();
        }
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        // Клиентский код может параметризовать отправителя любыми
        // командами.
        Invoker invoker = new Invoker();
        invoker.SetOnStart(new SimpleCommand("Say Hi!"));
        Receiver receiver = new Receiver();
        invoker.SetOnFinish(new ComplexCommand(receiver, "Send email", "Save report"));

        invoker.DoSomethingImportant();
    }
}
```

```
Invoker: Does anybody want something done before I begin?
SimpleCommand: See, I can do simple things like printing (Say Hi!)
Invoker: ...doing something really important...
Invoker: Does anybody want something done after I finish?
ComplexCommand: Complex stuff should be done by a receiver object.
Receiver: Working on (Send email.)
Receiver: Also working on (Save report.)
```

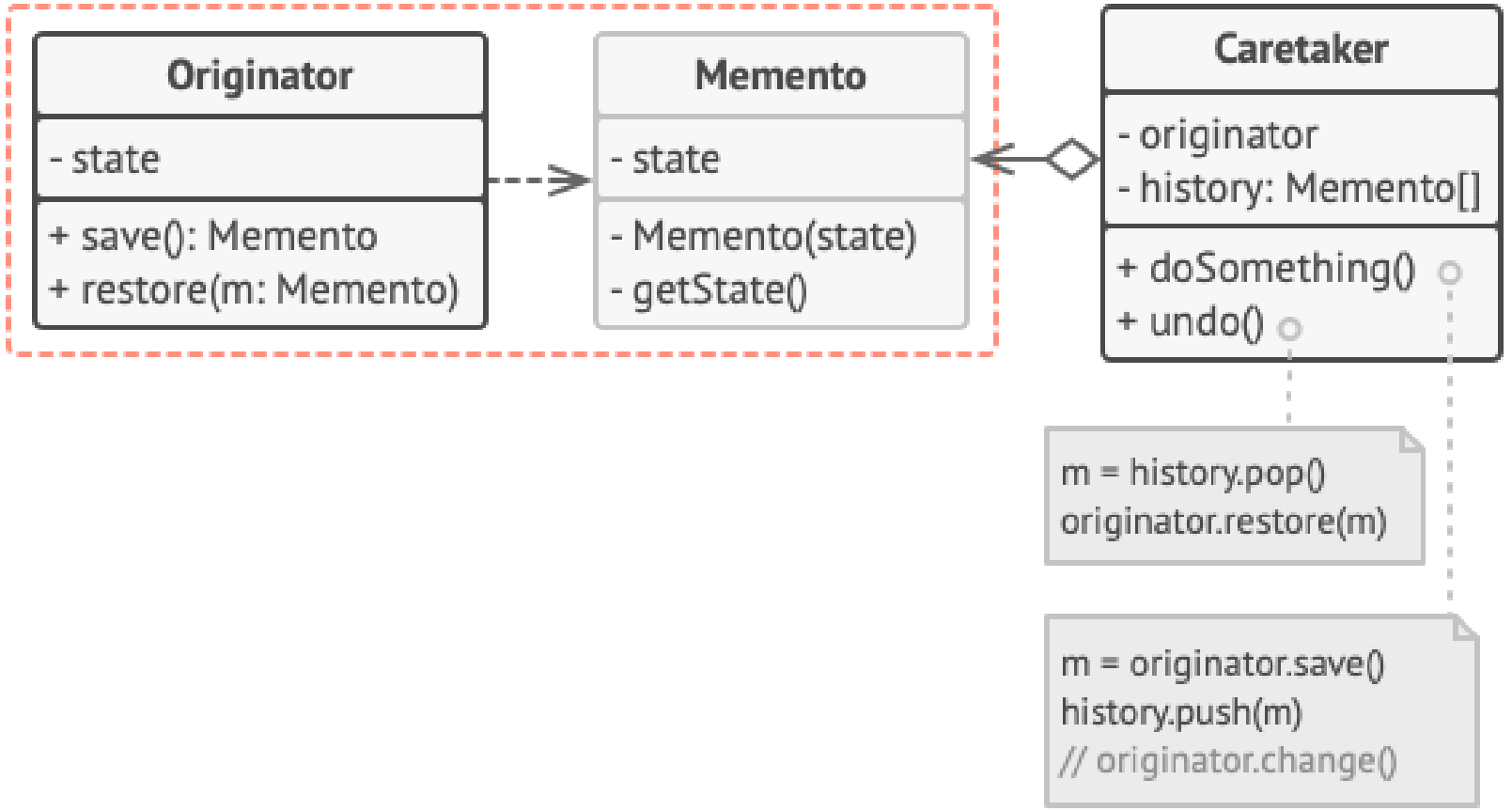
Снимок

Позволяет делать снимки состояния объектов, не раскрывая подробностей их реализации. Затем снимки можно использовать, чтобы восстановить прошлое состояние объектов.

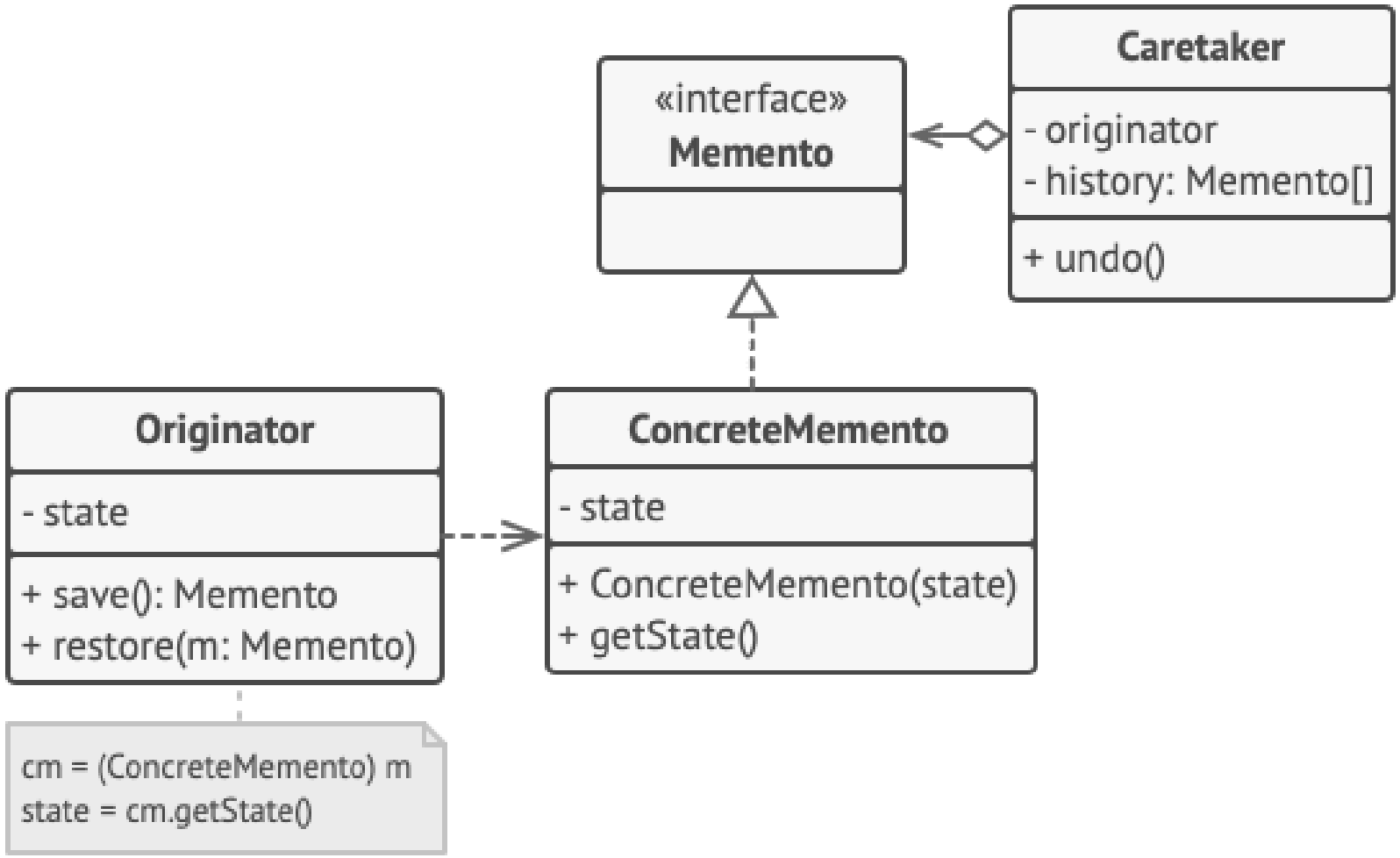
Когда применять?

- Когда нужно сохранить состояние объекта для возможного последующего восстановления.
- Когда сохранение состояния должно проходить без нарушения принципа инкапсуляции.

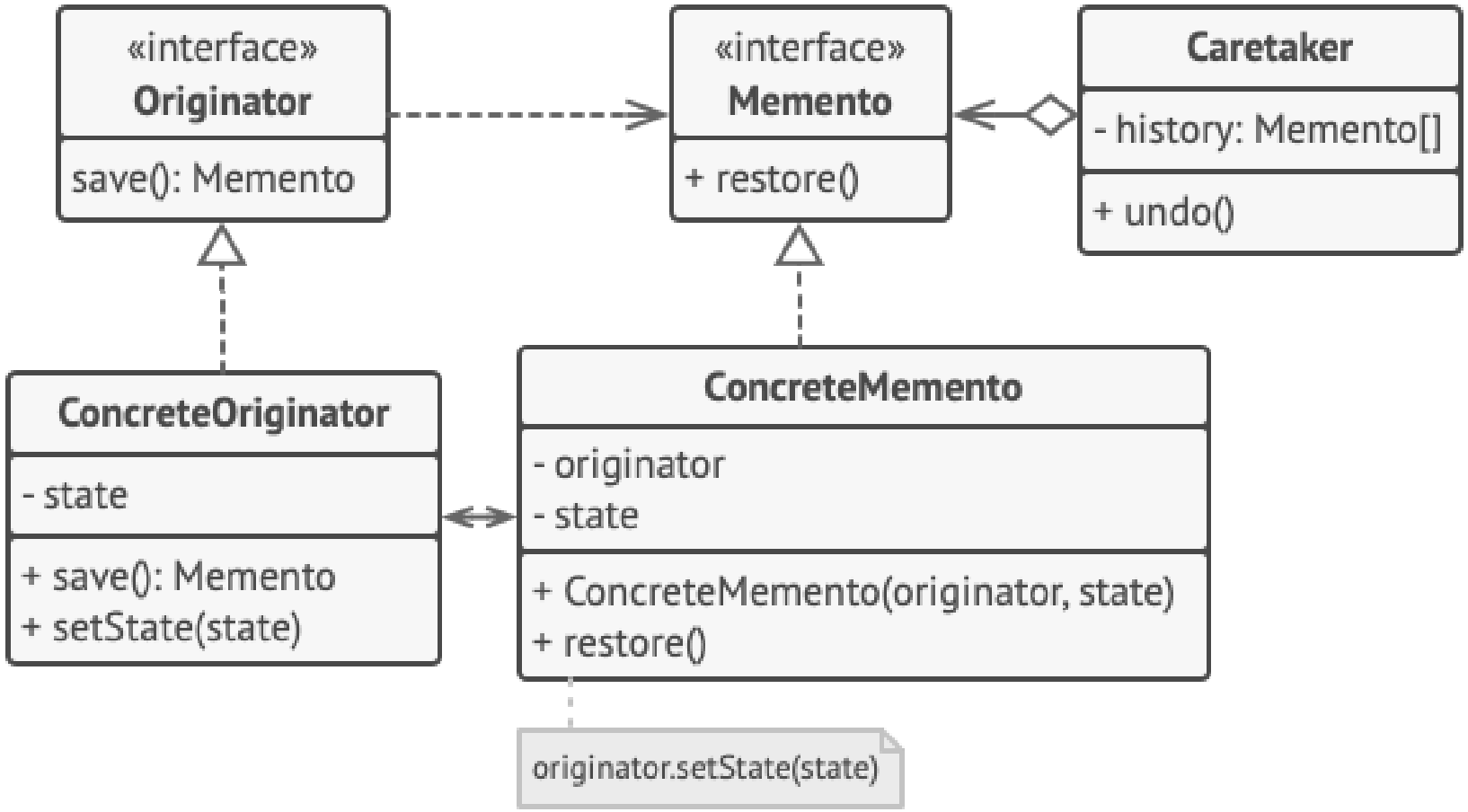
Структура. Вложенные классы



Структура. Пустой интерфейс



Структура. Пустой интерфейс



Пример

```
// Интерфейс Снимка предоставляет способ извлечения метаданных снимка, таких
// как дата создания или название. Однако он не раскрывает состояние
// Создателя.
public interface IMemento
{
    string GetName();

    string GetState();

    DateTime GetDate();
}
```

Пример

```
// Создатель содержит некоторое важное состояние, которое может со временем
// меняться. Он также объявляет метод сохранения состояния внутри снимка и
// метод восстановления состояния из него.
class Originator
{
    // Для удобства состояние создателя хранится внутри одной переменной.
    private string _state;

    public Originator(string state)
    {
        this._state = state;
        Console.WriteLine("Originator: My initial state is: " + state);
    }

    // Бизнес-логика Создателя может повлиять на его внутреннее состояние.
    // Поэтому клиент должен выполнить резервное копирование состояния с
    // помощью метода save перед запуском методов бизнес-логики.
    public void DoSomething()
    {
        Console.WriteLine("Originator: I'm doing something important.");
        this._state = this.GenerateRandomString(30);
        Console.WriteLine($"Originator: and my state has changed to: {_state}");
    }
```

Пример

```
    private string GenerateRandomString(int length = 10)
    {
        string allowedSymbols = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
        string result = string.Empty;

        while (length > 0)
        {
            result += allowedSymbols[new Random().Next(0, allowedSymbols.Length)];

            Thread.Sleep(12);

            length--;
        }

        return result;
    }

    // Сохраняет текущее состояние внутри снимка.
    public IMemento Save()
    {
        return new ConcreteMemento(this._state);
    }

    // Восстанавливает состояние Создателя из объекта снимка.
    public void Restore(IMemento memento)
    {
        if (!(memento is ConcreteMemento))
        {
            throw new Exception("Unknown memento class " + memento.ToString());
        }

        this._state = memento.GetState();
        Console.Write($"Originator: My state has changed to: {_state}");
    }
}
```

Пример

```
// Конкретный снимок содержит инфраструктуру для хранения состояния
// Создателя.
class ConcreteMemento : IMemento
{
    private string _state;

    private DateTime _date;

    public ConcreteMemento(string state)
    {
        this._state = state;
        this._date = DateTime.Now;
    }

    // Создатель использует этот метод, когда восстанавливает своё
    // состояние.
    public string GetState()
    {
        return this._state;
    }

    // Остальные методы используются Опекуном для отображения метаданных.
    public string GetName()
    {
        return $"{this._date} / ({this._state.Substring(0, 9)})...";
    }

    public DateTime GetDate()
    {
        return this._date;
    }
}
```

Пример

```
// Опекун не зависит от класса Конкретного Снимка. Таким образом, он не
// имеет доступа к состоянию создателя, хранящемуся внутри снимка. Он
// работает со всеми снимками через базовый интерфейс Снимка.
class Caretaker
{
    private List<IMemento> _mementos = new List<IMemento>();

    private Originator _originator = null;

    public Caretaker(Originator originator)
    {
        this._originator = originator;
    }

    public void Backup()
    {
        Console.WriteLine("\nCaretaker: Saving Originator's state...");
        this._mementos.Add(this._originator.Save());
    }
```

Пример

```
    public void Undo()
    {
        if (this._mementos.Count == 0)
        {
            return;
        }

        var memento = this._mementos.Last();
        this._mementos.Remove(memento);

        Console.WriteLine("Caretaker: Restoring state to: " + memento.GetName());

        try
        {
            this._originator.Restore(memento);
        }
        catch (Exception)
        {
            this.Undo();
        }
    }

    public void ShowHistory()
    {
        Console.WriteLine("Caretaker: Here's the list of mementos:");

        foreach (var memento in this._mementos)
        {
            Console.WriteLine(memento.GetName());
        }
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        // Клиентский код.
        Originator originator = new Originator("Super-duper-super-puper-super.");
        Caretaker caretaker = new Caretaker(originator);

        caretaker.Backup();
        originator.DoSomething();

        caretaker.Backup();
        originator.DoSomething();

        caretaker.Backup();
        originator.DoSomething();

        Console.WriteLine();
        caretaker.ShowHistory();

        Console.WriteLine("\nClient: Now, let's rollback!\n");
        caretaker.Undo();

        Console.WriteLine("\n\nClient: Once more!\n");
        caretaker.Undo();

        Console.WriteLine();
    }
}
```

```
Originator: My initial state is: Super-duper-super-puper-super.

Caretaker: Saving Originator's state...
Originator: I'm doing something important.
Originator: and my state has changed to: oGyQIIatlDDWNgYYqJATTmdwnnGZQj

Caretaker: Saving Originator's state...
Originator: I'm doing something important.
Originator: and my state has changed to: jBtMDDWogzzRJbTTmEwOOhZrjjBULe

Caretaker: Saving Originator's state...
Originator: I'm doing something important.
Originator: and my state has changed to: exoHyyRkbuuNEXOhhArKccUmexPPHZ

Caretaker: Here's the list of mementos:
12.06.2018 15:52:45 / (Super-dup...)
12.06.2018 15:52:46 / (oGyQIIatl...)
12.06.2018 15:52:46 / (jBtMDDWog...)

Client: Now, let's rollback!

Caretaker: Restoring state to: 12.06.2018 15:52:46 / (jBtMDDWog...)
Originator: My state has changed to: jBtMDDWogzzRJbTTmEwOOhZrjjBULe

Client: Once more!

Caretaker: Restoring state to: 12.06.2018 15:52:46 / (oGyQIIatl...)
Originator: My state has changed to: oGyQIIatlDDWNgYYqJATTmdwnnGZQj
```

Пример

```
using System;

namespace ITMO_OOP
{
    3 references
    public interface IMemento
    {
        2 references
        DateTime CreationDate { get; }
    }
}
```

```
using System;

namespace ITMO_OOP
{
    1 reference
    public class Originator
    {
        private string _state1;
        private int _state2;
        private double _state3;

        3 references
        private class Memento : IMemento
        {
            1 reference
            public Memento(string state1, int state2, double state3)
            {
                State1 = state1;
                State2 = state2;
                State3 = state3;
            }
            2 references
            public string State1 { get; }
            2 references
            public int State2 { get; }
            2 references
            public double State3 { get; }
            2 references
            public DateTime CreationDate { get; } = DateTime.Now;
        }

        1 reference
        public IMemento SaveState() => new Memento(_state1, _state2, _state3);

        1 reference
        public void RestoreState(IMemento memento)
        {
            if (!(memento is Memento ownMemento)) throw new ArgumentException(message: "Not supported Memento!");

            _state1 = ownMemento.State1;
            _state2 = ownMemento.State2;
            _state3 = ownMemento.State3;
        }
    }
}
```

Пример

```
1  using System;
2
3  namespace ITMO_OOP
4  {
       0 references
5      class Program
6      {
           0 references
7          static void Main(string[] args)
8          {
9              var orig = new Originator();
10
11             var memento = orig.SaveState();
12             Console.WriteLine(memento.CreationDate);
13
14             orig.RestoreState(memento);
15         }
16     }
17 }
```

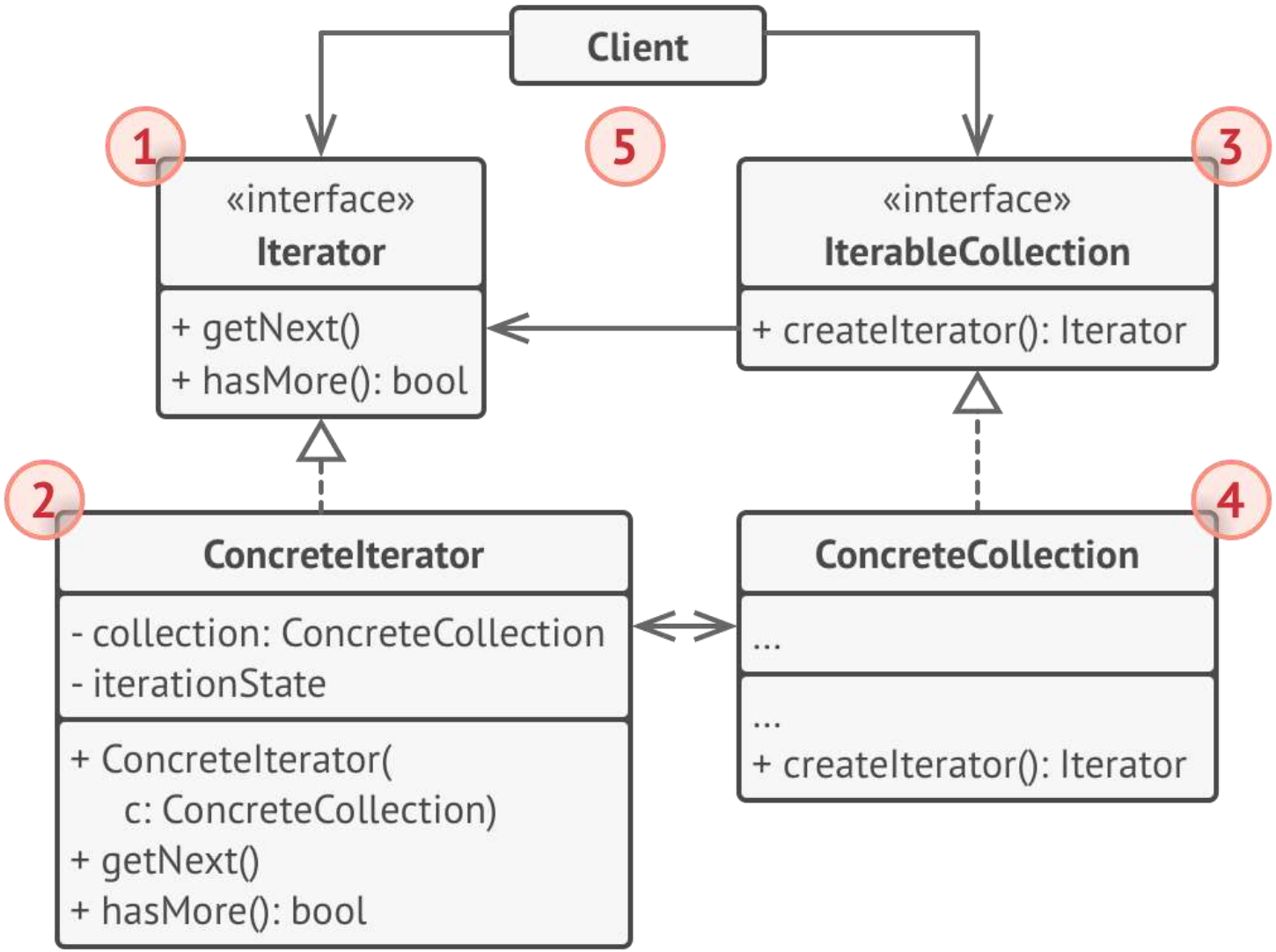
Итератор

Дает возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

Когда применять?

- Когда необходимо осуществить обход объекта без раскрытия его внутренней структуры.
- Когда имеется набор составных объектов, и надо обеспечить единый интерфейс для их перебора.
- Когда необходимо предоставить несколько альтернативных вариантов перебора одного и того же объекта.

Структура


Client
1
5
3
«interface»
Iterator
+ getNext()
+ hasMore(): bool
«interface»
IterableCollection
+ createIterator(): Iterator
2
4
ConcreteIterator
- collection: ConcreteCollection
- iterationState
+ ConcreteIterator(
c: ConcreteCollection)
+ getNext()
+ hasMore(): bool
ConcreteCollection
...
...
+ createIterator(): Iterator

Пример

```
abstract class Iterator : IEnumerator
{
    object IEnumerator.Current => Current();

    // Возвращает ключ текущего элемента
    public abstract int Key();

    // Возвращает текущий элемент.
    public abstract object Current();

    // Переходит к следующему элементу.
    public abstract bool MoveNext();

    // Перематывает Итератор к первому элементу.
    public abstract void Reset();
}
```

```
abstract class IteratorAggregate : IEnumerable
{
    // Возвращает Iterator или другой IteratorAggregate для реализующего
    // объекта.
    public abstract IEnumerator GetEnumerator();
}
```

Пример

```
// Конкретные Коллекции предоставляют один или несколько методов для
// получения новых экземпляров итератора, совместимых с классом коллекции.
class WordsCollection : IteratorAggregate
{
    List<string> _collection = new List<string>();

    bool _direction = false;

    public void ReverseDirection()
    {
        _direction = !_direction;
    }

    public List<string> getItems()
    {
        return _collection;
    }

    public void AddItem(string item)
    {
        this._collection.Add(item);
    }

    public override IEnumerator GetEnumerator()
    {
        return new AlphabeticalOrderIterator(this, _direction);
    }
}
```

Пример

```
// Конкретные Итераторы реализуют различные алгоритмы обхода. Эти классы
// постоянно хранят текущее положение обхода.
class AlphabeticalOrderIterator : Iterator
{
    private WordsCollection _collection;

    // Хранит текущее положение обхода. У итератора может быть множество
    // других полей для хранения состояния итерации, особенно когда он
    // должен работать с определённым типом коллекции.
    private int _position = -1;

    private bool _reverse = false;

    public AlphabeticalOrderIterator(WordsCollection collection, bool reverse = false)
    {
        this._collection = collection;
        this._reverse = reverse;

        if (reverse)
        {
            this._position = collection.getItems().Count;
        }
    }

    public override object Current()
    {
        return this._collection.getItems()[_position];
    }

    public override int Key()
    {
        return this._position;
    }
```

Пример

```
    public override bool MoveNext()
    {
        int updatedPosition = this._position + (this._reverse ? -1 : 1);

        if (updatedPosition >= 0 && updatedPosition < this._collection.getItems().Count)
        {
            this._position = updatedPosition;
            return true;
        }
        else
        {
            return false;
        }
    }

    public override void Reset()
    {
        this._position = this._reverse ? this._collection.getItems().Count - 1 : 0;
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        // Клиентский код может знать или не знать о Конкретном Итераторе
        // или классах Коллекций, в зависимости от уровня косвенности,
        // который вы хотите сохранить в своей программе.
        var collection = new WordsCollection();
        collection.AddItem("First");
        collection.AddItem("Second");
        collection.AddItem("Third");

        Console.WriteLine("Straight traversal:");

        foreach (var element in collection)
        {
            Console.WriteLine(element);
        }

        Console.WriteLine("\nReverse traversal:");

        collection.ReverseDirection();

        foreach (var element in collection)
        {
            Console.WriteLine(element);
        }
    }
}
```

```
Straight traversal:
First
Second
Third

Reverse traversal:
Third
Second
First
```

Спасибо за внимание!