**# Паттерн "Цепочка обязанностей" (Chain of responsibility)**

**## Описание**

Передача запроса одному из нескольких обработчиков в том случае, когда неважно, кто в действительности обработает запрос, и когда количество обработчиков может меняться.

Для этого обработчики выстаиваются в цепочку и запрос передаётся вдоль всей цепочки до тех пор, пока не будет обработан одним из её звеньев, либо не будет достигнут её конец.

**## Участники**

**\*\*Handler\*\*** // Обработчик

    (+) Handle(request: object)

**\*\*ConcreteHandler: Handler\*\*** // Конкретный обработчик

    (-) successor?: Handler // Преемник

    (+) (successor?: Handler)

        .successor = successor

    (+) Handle(request: object)

        use request or .successor.Handle(request)

**\*\*Client\*\*** // Клиент

    (-) Main()

        Handler h1 = new ConcreteHandler1()

        Handler h2 = new ConcreteHandler2(h1)

        object request = new() { … }

        h2.Handle(request)

**## Пример**

Руководитель (Client) передаёт Приказ (Request) вниз по иерархии своих Подчинённых (Handler), каждый из которых имеет ровно по одному Подчинённому (Successor), либо не имеет такового.

Если Подчинённый не может выполнить Приказ, он передаёт его своему Подчинённому.

**# Паттерн "Команда" (Command)**

**## Описание**

Обработка запроса на выполнение определённого действия через посредника, который позволяет разделить две иерархии классов – инициаторов запросов и их обработчиков.

Обработчик запроса выполняет то действие, которое потребует от него посредник, называемый командой.

Обработчик запроса называется получателем команды, а инициатор запроса – инициатором команды.

Команда может позволять не только выполнять действия, но и отменять их, а также может содержать в себе несколько команд, которые выполняются и отменяются по очереди.

Команда, содержащая несколько команд, называется макрокомандой.Она имеет тот же интерфейс, что и другие команды.

**## Участники**

**\*\*Receiver\*\*** // Получатель команды

    (+) ExecuteOperation()

    (+) UndoOperation()

**\*\*Command\*\*** // Команда

    (+) Execute()

    (+) Undo()

**\*\*ConcreteCommand: Command\*\*** // Конкретная команда

    (-) receiver: Receiver

    (+) (receiver: Receiver)

        .receiver = receiver

    (+) Execute()

        .receiver.ExecuteOperation()

    (+) Undo()

        .receiver.UndoOperation()

**\*\*MacroCommand: Command\*\*** // Макрокоманда

    (-) commands: Command[]

    (+) (commands: Command[])

        .commands = commands

    (+) Execute()

        (foreach command in .commands).Execute()

    (+) Undo()

        (foreach command in .commands).Undo()

**\*\*Invoker\*\*** // Инициатор команды

    (-) command: Command

    (+) SetCommand(command: Command)

        .command = command

    (+) Run()

        .command.Execute()

    (+) Cancel()

        .command.Undo()

**\*\*Client\*\*** // Клиент

    (-) Main()

        Receiver receiver = new Receiver()

        Command command = new ConcreteCommand(receiver)

        Invoker invoker = new Invoker()

        invoker.SetCommand(command)

        invoker.Run()

        invoker.Cancel()

        Command macroCommand = new MacroCommand([command])

        invoker.Run()

        invoker.Cancel()

**## Пример**

Кнопка включения/выключения пульта управления (Invoker) может быть настроена для разных моделей телевизоров (Receiver) с помощью инструкции (Command) по работе с конкретной моделью телевизора.

В инструкциях по настройке телевизоров каждой из моделей есть разделы "Включение" и "Выключение", где описывается, какой набор сигналов нужно подать на вход телевизора, чтобы он выполнил указанное действие.

**# Паттерн "Интерпретатор" (Interpreter)**

**## Описание**

Получение результата вычислений в иерархическом дереве выражений.

Выражения, которые размещаются в листовых узлах дерева, называются терминальными (terminal), остальные – нетерминальными (non terminal).

Выражение в каждом узле дерева вычисляет свой результат на основе результатов, полученных от вычисления выражений в его дочерних узлах.

Результат вычисления выражения в корне дерева является конечным результатом вычисления всего дерева выражений.

Значения, необходимые для вычислений, хранятся в контексте – специальном классе, предоставляющий возможность добавления и получения этих значений.

При вычислении результата выражения получают доступ к контексту.

**## Участники**

**\*\*Context\*\*** // Контекст

    (-) valueLookup: Dictionary<string, int> = new()

    (+) AddValue(name: string, value: int)

        .valueLookup[name] = value

    (+) GetValue(name: string): int

        return .valueLookup[name]

**\*\*Expression\*\*** // Выражение

    (+) Interpret(context: Context): int

**\*\*TerminalExpression: Expression\*\*** // Терминальное выражение

    (-) name: string

    (+) (name: string)

        .name = name

    (+) Interpret(context: Context): int

        return context.GetValue(.name)

**\*\*NonTerminalExpression: Expression\*\*** // Терминальное выражение

    (-) expression1: Expression

    (-) expression2: Expression

    (+) (expression1: Expression, expression2: Expression)

        .expression1 = expression1

        .expression2 = expression2

    (+) Interpret(context: Context): int

        return .expression1.Interpret(context) + .expression2.Interpret(context)

**\*\*Client\*\*** // Клиент

    Main()

        Context context = new Context()

        context.AddValue(“x”, 1)

        context.AddValue(“y”, 2)

        Expression x = new TerminalExpression(“x”)

        Expression y = new TerminalExpression(“y”)

        Expression x\_plus\_y = new NonTerminalExptrssion(x, y)

        int result = x\_plus\_y.Interpret(context)

**## Пример**

Вычисление арифметического выражения "x + y – z", где переменные "x", "y" и "z" – TerminalExpression, а операции "+" и "-" - NonTerminalExpression.

**# Паттерн "Итератор" (Iterator)**

**## Описание**

Последовательный доступ к элементам составного объекта без раскрытия его внутренней структуры.

Алгоритм перебора сосредоточен в отдельном объекте - итераторе, который имеет доступ к составному объекту - агрегату.

За создание итератора отвечает сам агрегат.

При создании агрегата в него передаётся набор элементов, которые необходимо перебрать.

Перебор этих элементов можно осуществлять разными способами в зависимости от выбранного агрегата.

**## Участники**

**\*\*Iterator\*\*** // Итератор (перечислитель)

    (+) Current(): object

    (+) First(): object

    (+) IsDone(): bool

    (+) Next(): object

**\*\*ConcreteIterator\*\*** // Конкретный итератор

    (-) aggregate: Aggregate

    (-) currentIndex: int

    (+) (items: object[])

        .items = items

        .currentIndex = 0

    (+) Current(): object

        return .aggregate.GetItemByIndex(.currentIndex)

    (+) First(): object

        return .aggregate.GetItemByIndex(0)

    (+) IsDone(): bool

        return .currentIndex == .aggregate.Count - 1

    (+) Next(): object

        return .aggregate.GetItemByIndex(++.currentIndex)

**\*\*Aggregate\*\*** // Агрегат (составной объект)

    (+) get Count: int

    (+) GetItemByIndex(index: int): object

    (+) GetIterator(): Iterator

**\*\*ConcreteAggregate: Aggregate\*\*** // Конкретный агрегат

    (-) items: object[]

    (+) get Count: int

        return items.Length

    (+) (items: object[])

        .items = items

    (+) GetIterator(): Iterator

        return new ConcreteIterator(.)

**\*\*Client\*\*** // Клиент

    (-) Main()

        object[] items = new object [] {...}

        Aggregate aggregate = new ConcreteAggregate(items)

        Iterator iterator = aggregate.GetIterator();

        iterator.Next()

        while (!iterator.IsDone)

        {

            iterator.Next()

        }

        iterator.Current()

**## Пример**

Библиотека (Aggregate) содержит книги (Набор элементов), которые можно перебрать с помощью каталога (Iterator).

**# Паттерн "Посредник" (Mediator)**

**## Описание**

Взаимодействие множества объектов не напрямую, а через специально выделенный объект - посредник.

Посредник принимает сообщения от желающих взаимодействовать друг с другом объектов - коллег.

В зависимости от того, какой коллега прислал сообщение, посредник принимает решение о передаче его другим коллегам.

Это позволяет коллегам не подозревать о существовании друг друга и упростить логику взаимодействия между ними, сосредоточив её в одном месте - в посреднике.

**## Участники**

**\*\*Colleague\*\*** // Коллега

    (+) Notify(message: string)

**\*\*Mediator\*\*** // Посредник

    (+) Send(message: string, colleague: Colleague)

**\*\*ConcreteColleague: Colleague\*\*** // Конкретный коллега

    (-) mediator: Mediator

    (+) (mediator: Mediator)

        .mediator = mediator

    (+) Notify(message: string)

        use message

    (+) Send(message: string)

        .mediator.Send(message, .)

**\*\*ConcreteMediator: Mediator\*\*** // Конкретный посредник

    (+) Colleague1: ConcreteColleague1

    (+) Colleague2: ConcreteColleague1

    (+) Send(message: string, colleague: Colleague)

        (one of .Colleague1 or .Colleague2 which != colleague).Notify(message)

**\*\*Client\*\***

    (-) Main()

        Mediator mediator = new ConcreteMediator()

        Colleague colleague1 = new ConcreteColleague1()

        Colleague colleague2 = new ConcreteColleague2()

        mediator.Colleague1 = colleague1

        mediator.Colleague2 = colleague2

        colleague1.Send("message from colleague1")

        colleague2.Send("message from colleague2")

**## Пример**

Менеджер (Mediator) берёт на себя коммуникации между исполнителями (Colleague), которые не подозревают о существовании друг друга.