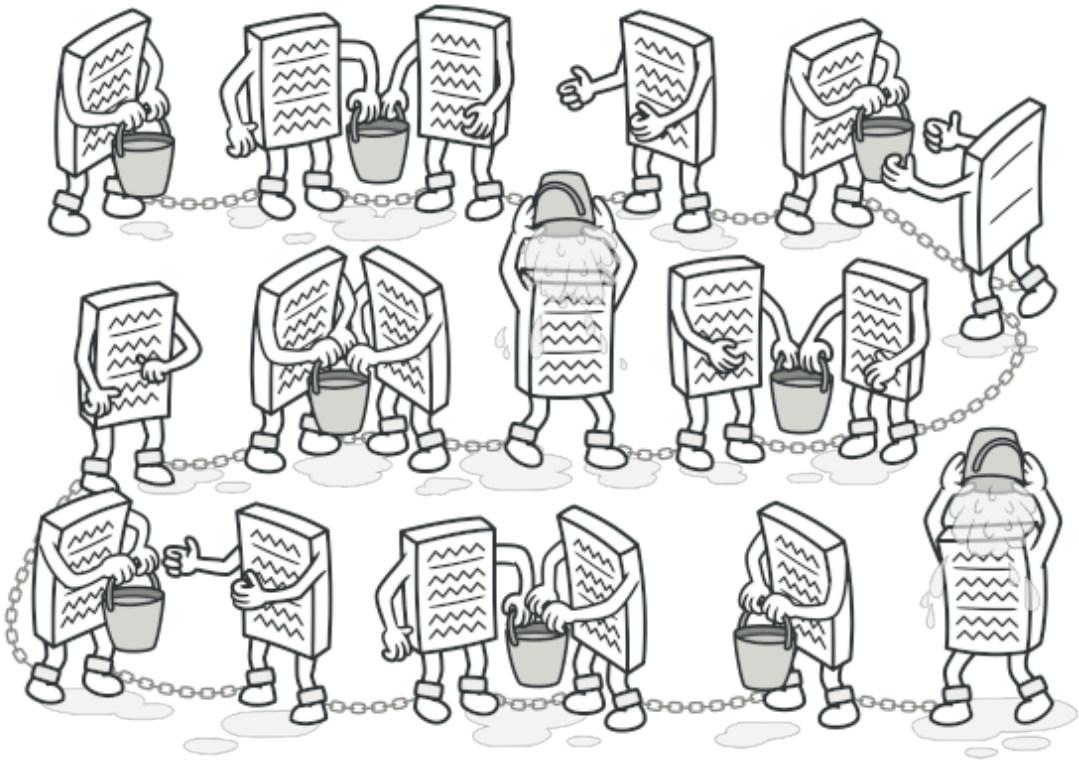
Цепочка обязанностей

Также известен как: CoR, Chain of Command, Chain of Responsibility

Суть паттерна

Цепочка обязанностей — это поведенческий паттерн проектирования, который 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.



Проблема

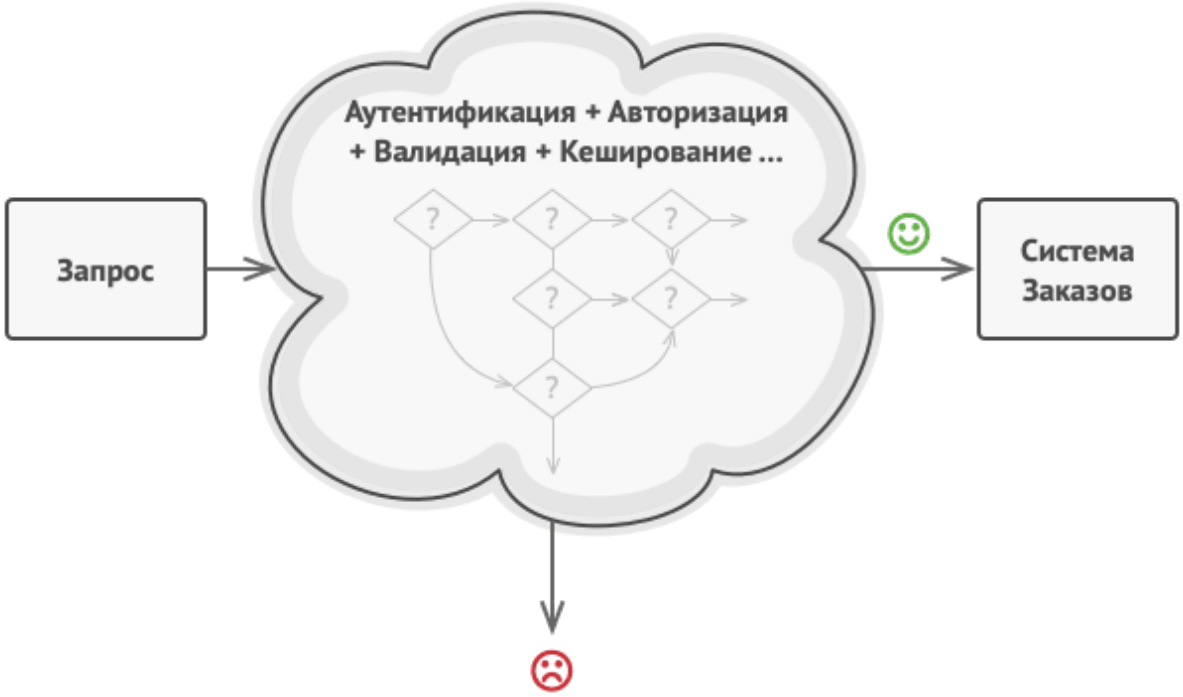
Представьте, что вы делаете систему приёма онлайн-заказов. Вы хотите ограничить к ней доступ так, чтобы только авторизованные пользователи могли создавать заказы. Кроме того, определённые пользователи, владеющие правами администратора, должны иметь полный доступ к заказам.

Вы быстро сообразили, что эти проверки нужно выполнять последовательно. Ведь пользователя можно попытаться «залогинить» в систему, если его запрос содержит логин и пароль. Но если такая попытка не удалась, то проверять расширенные права доступа попросту не имеет смысла.

Запрос проходит ряд проверок перед доступом в систему заказов.

На протяжении следующих нескольких месяцев вам пришлось добавить ещё несколько таких последовательных проверок.

- Кто-то резонно заметил, что неплохо бы проверять данные, передаваемые в запросе перед тем, как вносить их в систему — вдруг запрос содержит данные о покупке несуществующих продуктов.
- Кто-то предложил блокировать массовые отправки формы с одним и тем же логином, чтобы предотвратить подбор паролей ботами.
- Кто-то заметил, что форму заказа неплохо бы доставать из кеша, если она уже была однажды показана.



Со временем код проверок становится всё более запутанным.

С каждой новой «фичей» код проверок, выглядящий как большой клубок условных операторов, всё больше и больше раздувался. При изменении одного правила приходилось трогать код всех проверок. А для того, чтобы применить проверки к другим ресурсам, пришлось продублировать их код в других классах.

Поддерживать такой код стало не только очень хлопотно, но и затратно. И вот в один прекрасный день вы получаете задачу рефакторинга...

Решение

Как и многие другие поведенческие паттерны, Цепочка обязанностей базируется на том, чтобы превратить отдельные поведения в объекты. В нашем случае каждая проверка переедет в отдельный класс с единственным методом выполнения. Данные запроса, над которым происходит проверка, будут передаваться в метод как аргументы.

А теперь по-настоящему важный этап. Паттерн предлагает связать объекты обработчиков в одну цепь. Каждый из них будет иметь ссылку на следующий обработчик в цепи. Таким образом, при получении запроса обработчик сможет не только сам что-то с ним сделать, но и передать обработку следующему объекту в цепочке.

Передавая запросы в первый обработчик цепочки, вы можете быть уверены, что все объекты в цепи смогут его обработать. При этом длина цепочки не имеет никакого значения.

И последний штрих. Обработчик не обязательно должен передавать запрос дальше, причём эта особенность может быть использована по-разному.

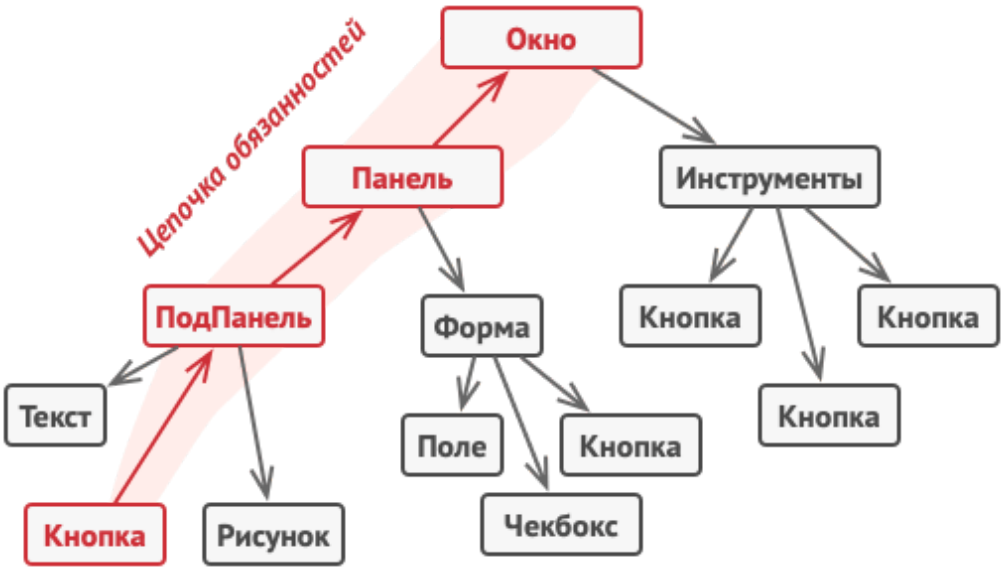
В примере с фильтрацией доступа обработчики прерывают дальнейшие проверки, если текущая проверка не прошла. Ведь нет смысла тратить попусту ресурсы, если и так понятно, что с запросом что-то не так.

Обработчики следуют в цепочке один за другим.

Но есть и другой подход, при котором обработчики прерывают цепь только когда они *могут* обработать запрос. В этом случае запрос движется по цепи, пока не найдётся обработчик, который может его обработать. Очень часто такой подход используется для передачи событий, создаваемых классами графического интерфейса в результате взаимодействия с пользователем.

Например, когда пользователь кликает по кнопке, программа выстраивает цепочку из объекта этой кнопки, всех её родительских элементов и общего окна приложения на конце. Событие клика передаётся по этой цепи до тех пор, пока не найдётся объект, способный его

обработать. Этот пример примечателен ещё и тем, что цепочку всегда можно выделить из древовидной структуры объектов, в которую обычно и свёрнуты элементы пользовательского интерфейса.



Цепочку можно выделить даже из дерева объектов.

Очень важно, чтобы все объекты цепочки имели общий интерфейс. Обычно каждому конкретному обработчику достаточно знать только то, что следующий объект в цепи имеет метод `выполнить`. Благодаря этому связи между объектами цепочки будут более гибкими. Кроме того, вы сможете формировать цепочки на лету из разнообразных объектов, не привязываясь к конкретным классам.

Аналогия из жизни

Пример общения с поддержкой.

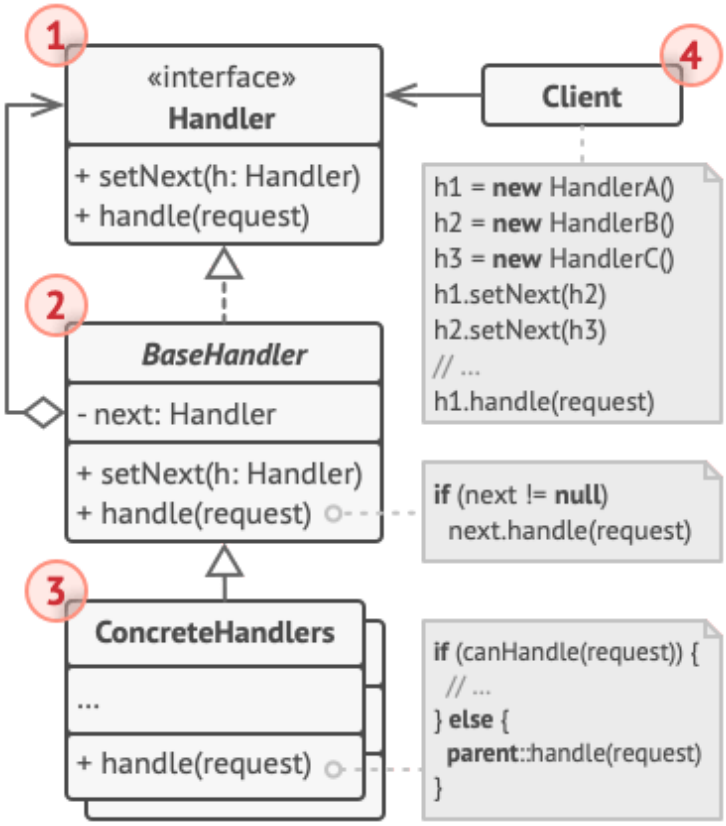
Вы купили новую видеокарту. Она автоматически определилась и заработала под Windows, но в вашей любимой Ubuntu «завести» её не удалось. Со слабой надеждой вы звоните в службу поддержки.

Первым вы слышите голос автоответчика, предлагающий выбор из десятка стандартных решений. Ни один из вариантов не подходит, и робот соединяет вас с живым оператором.

Увы, но рядовой оператор поддержки умеет общаться только заученными фразами и давать шаблонные ответы. После очередного предложения «выключить и включить компьютер» вы просите связать вас с настоящими инженерами.

Оператор перебрасывает звонок дежурному инженеру, изнывающему от скуки в своей каморке. Уж он-то знает, как вам помочь! Инженер рассказывает вам, где скачать подходящие драйвера и как настроить их под Ubuntu. Запрос удовлетворён. Вы кладёте трубку.

Структура



1. **Обработчик** определяет общий для всех конкретных обработчиков интерфейс. Обычно достаточно описать единственный метод обработки запросов, но иногда здесь может быть объявлен и метод выставления следующего обработчика.

2. **Базовый обработчик** — опциональный класс, который позволяет избавиться от дублирования одного и того же кода во всех конкретных обработчиках.

   Обычно этот класс имеет поле для хранения ссылки на следующий обработчик в цепочке. Клиент связывает обработчики в цепь, подавая ссылку на следующий обработчик через

конструктор или сеттер поля. Также здесь можно реализовать базовый метод обработки, который бы просто перенаправлял запрос следующему обработчику, проверив его наличие.

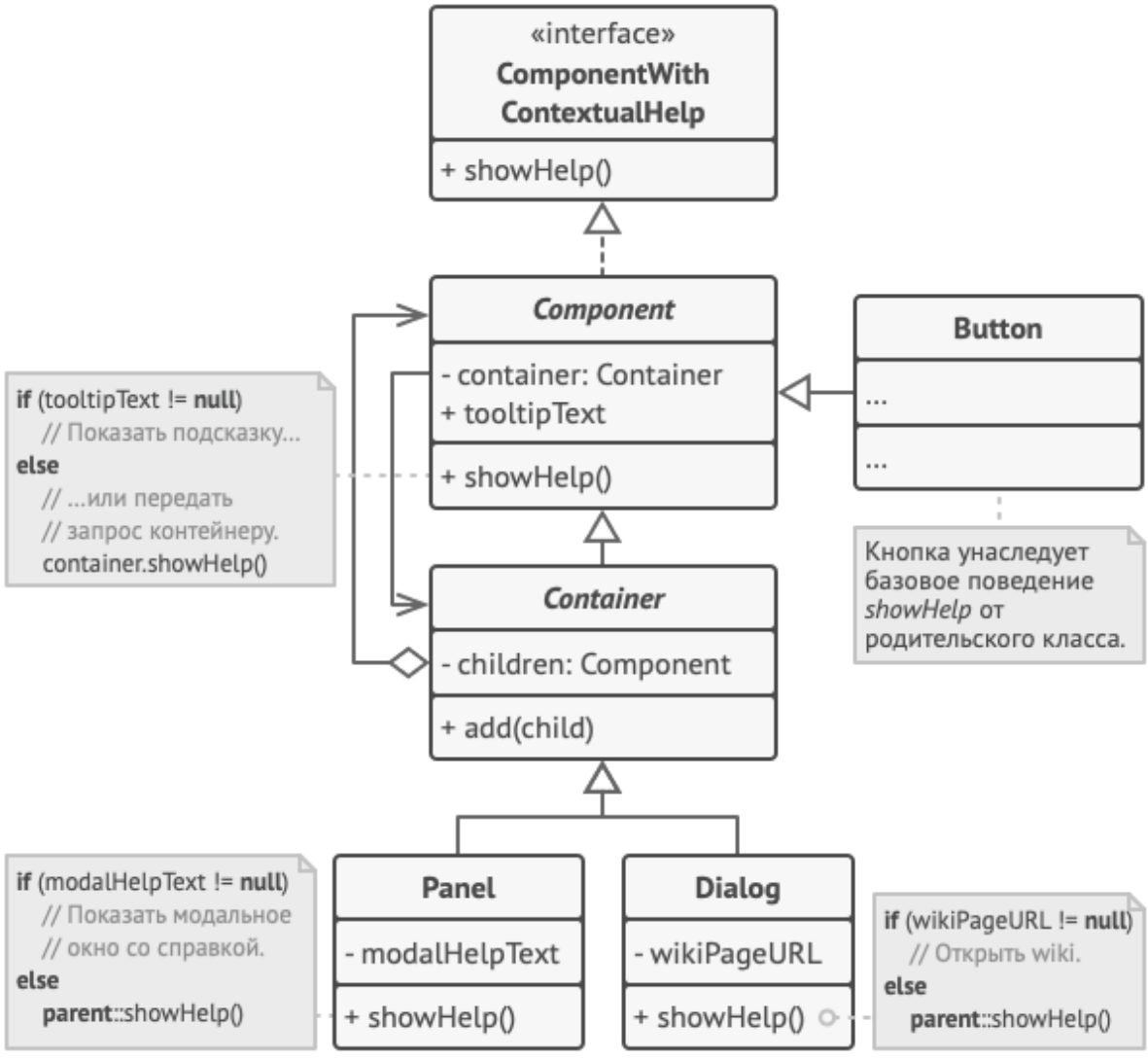
3. **Конкретные обработчики** содержат код обработки запросов. При получении запроса каждый обработчик решает, может ли он обработать запрос, а также стоит ли передать его следующему объекту.

   В большинстве случаев обработчики могут работать сами по себе и быть неизменяемыми, получив все нужные детали через параметры конструктора.

4. **Клиент** может либо сформировать цепочку обработчиков единожды, либо перестраивать её динамически, в зависимости от логики программы. Клиент может отправлять запросы любому из объектов цепочки, не обязательно первому из них.

Псевдокод

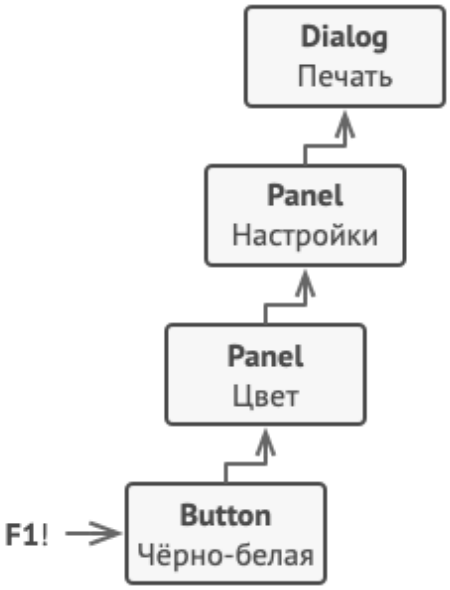
В этом примере **Цепочка обязанностей** отвечает за показ контекстной помощи для активных элементов пользовательского интерфейса.



Графический интерфейс построен с помощью компоновщика, где у каждого элемента есть ссылка на свой элемент-контейнер. Цепочку можно выстроить, пройдясь по всем контейнерам, в которые вложен элемент.

Графический интерфейс приложения обычно структурирован в виде дерева. Класс `Диалог`, отображающий всё окно приложения — это корень дерева. Диалог содержит `Панели`, которые, в свою очередь, могут содержать либо другие вложенные панели, либо простые элементы, вроде `Кнопок`.

Простые элементы могут показывать небольшие подсказки, если для них указан текст помощи. Но есть и более сложные компоненты, для которых этот способ демонстрации помощи слишком прост. Они определяют собственный способ отображения контекстной помощи.



Пример вызова контекстной помощи в цепочке объектов UI.

Когда пользователь наводит указатель мыши на элемент и жмёт клавишу `F1`, приложение шлёт этому элементу запрос на показ помощи. Если он не содержит никакой справочной информации, запрос путешествует далее по списку контейнера элемента, пока не находится тот, который способен отобразить помощь.

```
// Интерфейс обработчиков.
interface ComponentWithContextualHelp is
    method showHelp()


// Базовый класс простых компонентов.
abstract class Component implements ComponentWithContextualHelp is
    field tooltipText: string

    // Контейнер, содержащий компонент, служит в качестве
    // следующего звена цепочки.
    protected field container: Container
```

```
    // Базовое поведение компонента заключается в том, чтобы
    // показать всплывающую подсказку, если для неё задан текст.
    // В обратном случае – перенаправить запрос своему
    // контейнеру, если тот существует.
    method showHelp() is
        if (tooltipText != null)
            // Показать подсказку.
        else
            container.showHelp()


// Контейнеры могут включать в себя как простые компоненты, так
// и другие контейнеры. Здесь формируются связи цепочки. Класс
// контейнера унаследует метод showHelp от своего родителя –
// базового компонента.
abstract class Container extends Component is
    protected field children: array of Component

    method add(child) is
        children.add(child)
        child.container = this


// Большинство примитивных компонентов устроит базовое поведение
// показа помощи через подсказку, которое они унаследуют из
// класса Component.
class Button extends Component is
    // ...

// Но сложные компоненты могут переопределять метод показа
// помощи по-своему. Но и в этом случае они всегда могут
// вернуться к базовой реализации, вызвав метод родителя.
class Panel extends Container is
    field modalHelpText: string

    method showHelp() is
        if (modalHelpText != null)
            // Показать модальное окно с помощью.
        else
            super.showHelp()

// ...то же, что и выше...
class Dialog extends Container is
    field wikiPageURL: string

    method showHelp() is
        if (wikiPageURL != null)
            // Открыть страницу Wiki в браузере.
        else
            super.showHelp()
```

```
// Клиентский код.
class Application is
    // Каждое приложение конфигурирует цепочку по-своему.
    method createUI() is
        dialog = new Dialog("Budget Reports")
        dialog.wikiPageURL = "http://..."
        panel = new Panel(0, 0, 400, 800)
        panel.modalHelpText = "This panel does..."
        ok = new Button(250, 760, 50, 20, "OK")
        ok.tooltipText = "This is an OK button that..."
        cancel = new Button(320, 760, 50, 20, "Cancel")
        // ...
        panel.add(ok)
        panel.add(cancel)
        dialog.add(panel)

    // Представьте, что здесь произойдёт.
    method onF1KeyPress() is
        component = this.getComponentAtMouseCoords()
        component.showHelp()
```

Применимость

Когда программа должна обрабатывать разнообразные запросы несколькими способами, но заранее неизвестно, какие конкретно запросы будут приходить и какие обработчики для них понадобятся.

С помощью Цепочки обязанностей вы можете связать потенциальных обработчиков в одну цепь и при получении запроса поочерёдно спрашивать каждого из них, не хочет ли он обработать запрос.

Когда важно, чтобы обработчики выполнялись один за другим в строгом порядке.

Цепочка обязанностей позволяет запускать обработчиков последовательно один за другим в том порядке, в котором они находятся в цепочке.

Когда набор объектов, способных обработать запрос, должен задаваться динамически.

В любой момент вы можете вмешаться в существующую цепочку и переназначить связи так, чтобы убрать или добавить новое звено.

Шаги реализации

1. Создайте интерфейс обработчика и опишите в нём основной метод обработки.

   Продумайте, в каком виде клиент должен передавать данные запроса в обработчик. Самый гибкий способ — превратить данные запроса в объект и передавать его целиком через параметры метода обработчика.

2. Имеет смысл создать абстрактный базовый класс обработчиков, чтобы не дублировать реализацию метода получения следующего обработчика во всех конкретных обработчиках.

   Добавьте в базовый обработчик поле для хранения ссылки на следующий объект цепочки. Устанавливайте начальное значение этого поля через конструктор. Это сделает объекты обработчиков неизменяемыми. Но если программа предполагает динамическую перестройку цепочек, можете добавить и сеттер для поля.

   Реализуйте базовый метод обработки так, чтобы он перенаправлял запрос следующему объекту, проверив его наличие. Это позволит полностью скрыть поле-ссылку от подклассов, дав им возможность передавать запросы дальше по цепи, обращаясь к родительской реализации метода.

3. Один за другим создайте классы конкретных обработчиков и реализуйте в них методы обработки запросов. При получении запроса каждый обработчик должен решить:

   - Может ли он обработать запрос или нет?
   - Следует ли передать запрос следующему обработчику или нет?

4. Клиент может собирать цепочку обработчиков самостоятельно, опираясь на свою бизнес-логику, либо получать уже готовые цепочки извне. В последнем случае цепочки собираются фабричными объектами, опираясь на конфигурацию приложения или параметры окружения.

5. Клиент может посылать запросы любому обработчику в цепи, а не только первому. Запрос будет передаваться по цепочке до тех пор, пока какой-то обработчик не откажется передавать его дальше, либо когда будет достигнут конец цепи.

6. Клиент должен знать о динамической природе цепочки и быть готов к таким случаям:

   - Цепочка может состоять из единственного объекта.
   - Запросы могут не достигать конца цепи.
   - Запросы могут достигать конца, оставаясь необработанными.

Преимущества и недостатки

Уменьшает зависимость между клиентом и обработчиками.

Реализует *принцип единственной обязанности*.

Реализует *принцип открытости/закрытости*.

Запрос может остаться никем не обработанным.

Отношения с другими паттернами

- **Цепочка обязанностей**, **Команда**, **Посредник** и **Наблюдатель** показывают различные способы работы отправителей запросов с их получателями:

  - *Цепочка обязанностей* передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.
  - *Команда* устанавливает косвенную одностороннюю связь от отправителей к получателям.
  - *Посредник* убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.
  - *Наблюдатель* передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.

- **Цепочку обязанностей** часто используют вместе с **Компоновщиком**. В этом случае запрос передаётся от дочерних компонентов к их родителям.

- Обработчики в **Цепочке обязанностей** могут быть выполнены в виде **Команд**. В этом случае множество разных операций может быть выполнено над одним и тем же контекстом, коим является запрос.

  Но есть и другой подход, в котором сам запрос является *Командой*, посланной по цепочке объектов. В этом случае одна и та же операция может быть выполнена над множеством разных контекстов, представленных в виде цепочки.

- **Цепочка обязанностей** и **Декоратор** имеют очень похожие структуры. Оба паттерна базируются на принципе рекурсивного выполнения операции через серию связанных объектов. Но есть и несколько важных отличий.

  Обработчики в *Цепочке обязанностей* могут выполнять произвольные действия, независимые друг от друга, а также в любой момент прерывать дальнейшую передачу по цепочке. С другой стороны *Декораторы* расширяют какое-то определённое действие, не ломая интерфейс базовой операции и не прерывая выполнение остальных декораторов.