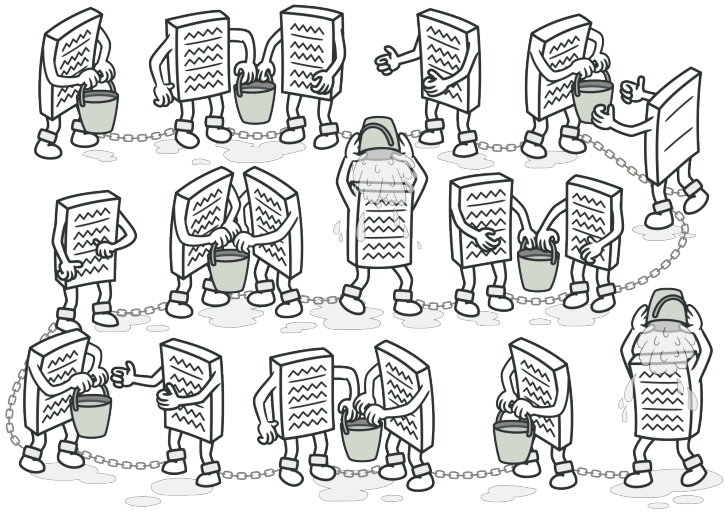


ЦЕПОЧКА ОБЯЗАННОСТЕЙ

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

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 «залогинить» пользователя можно в любой момент, если только запрос содержит логин и пароль. Но если аутентификация не удалась, то проверять права доступа 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 покупку несуществующих продуктов.

- Кто-то предложил фильтровать 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 очень хлопотно, да и затратно. И вот в один прекрасный день вы получаете задачу рефакторинга...

☺ Решение

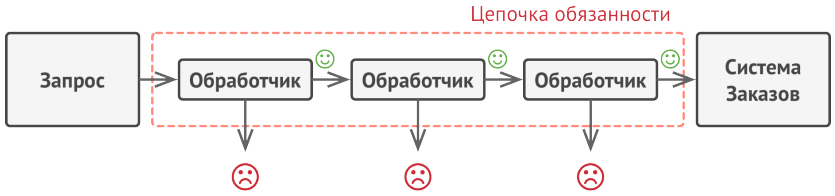
Как и многие другие поведенческие паттерны, Цепочка обязанностей базируется на том, чтобы превратить отдельные поведения в объекты. В нашем случае, каждая проверка переедет в отдельный класс с единственным методом выполнения. Данные запроса, над которым происходит проверка, будут передаваться в метод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

А теперь по-настоящему важный этап. Паттерн предлагает выстроить несколько обработчиков в цепь. Каждый обработчик будет хранить ссылку на следующий обработчик в цепи. А при получении запроса, он не только обработает его, но и передаст следующему объекту.

Таким образом, можно сформировать длинную цепочку обработчиков и передавать запрос в первый из них, зная о том, что вся цепочка сможет его обработать в определённом порядке.

И последний штрих. Обработчик необязательно должен передавать запрос дальше. Причём эта особенность может быть использована по-разному.

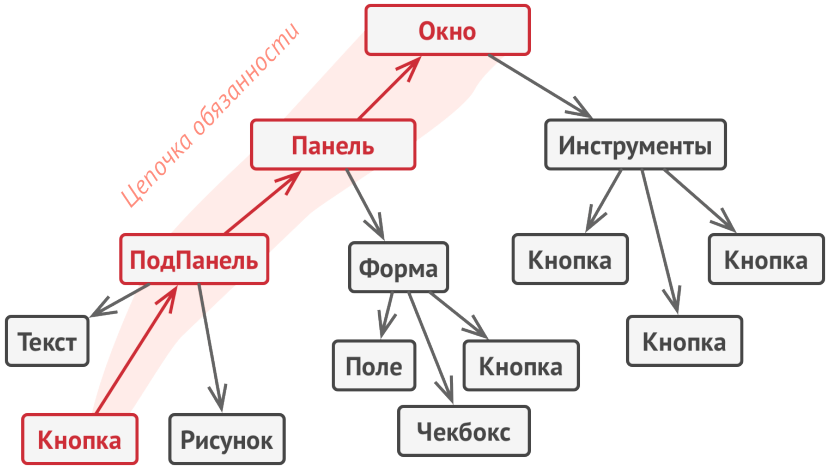
В примере с фильтрацией доступа, обработчики прерывают дальнейшие проверки, если текущая проверка не прошла. Ведь нет смысла тратить попусту ресурсы, если и так понятно, что с запросом что-то не так.



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, могущий его обработать. Очень часто такой подход используется для передачи событий в классах графического интерфейса.

Например, когда пользователь кликает по кнопке, выстраивается цепочка из самой кнопки, и всех её родительских элементов, заканчивающаяся окном всего приложения. 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.



Цепочку можно выделить даже из дерева объектов.

Очень важно, чтобы все объекты цепочки имели общий интерфейс. Это сделает связку объектов гибкой и позволит формировать её на лету из разнообразных объектов, не привязываясь к конкретным классам. Каждому конкретному обработчику будет важно знать только то, что следующий объект в цепи имеет метод `выполнить`.

Аналогия из жизни

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

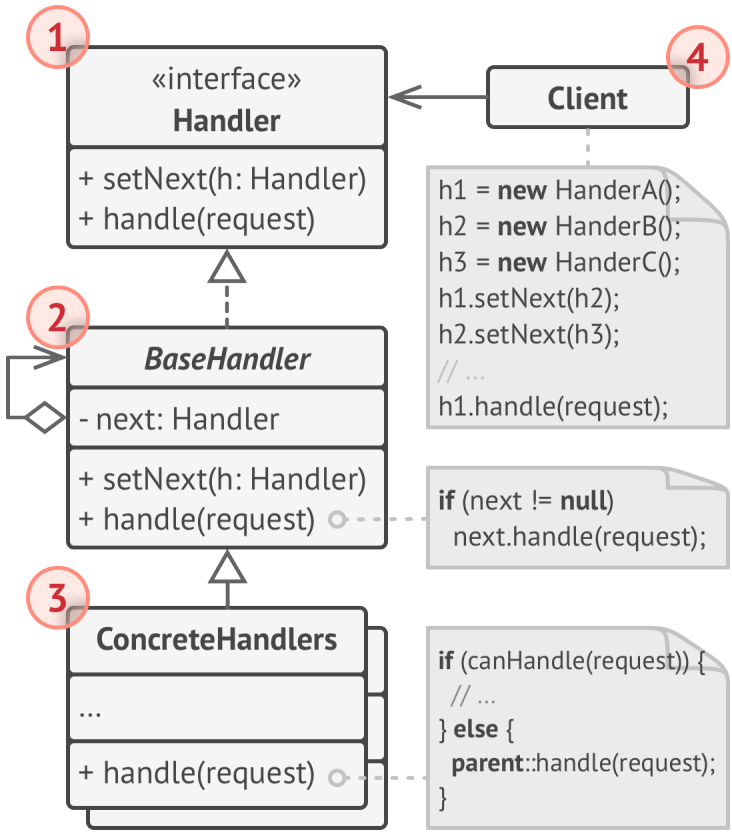
вариантов не подходит, и робот соединяет вас с живым оператором.

Пример общения с поддержкой.

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, где и как вы можете 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 определён 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 в конструктор или сеттер, определённый здесь. Здесь можно реализовать и метод обработки, который бы просто перенаправлял запрос следующему объекту, проверив его наличие.

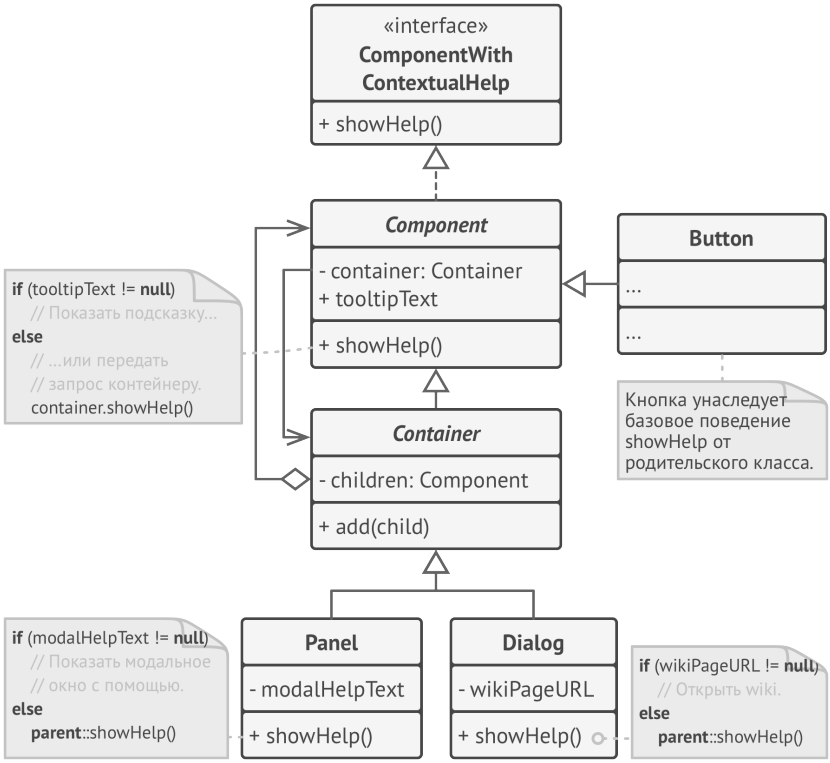
3. **Конкретные обработчики** содержат код обработки запросов. При получении запроса каждый обработчик решает, может ли он обработать запрос или нет, а также стоит ли передать его следующему объекту.

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

4. **Клиент** составляет цепочки обработчиков один раз или динамически, в зависимости от логики программы. Клиент может отправить запрос любому из объектов цепочки, причём это не всегда первый объект в цепочке.

Псевдокод

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

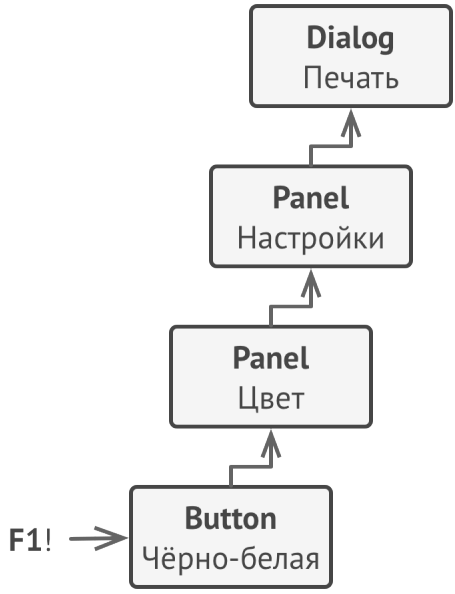


Классы UI построены с помощью компоновщика, но каждый элемент «знает» о своём контейнере. Цепочку можно выстроить, пройдясь по всем контейнерам, в которые вложен элемент.

Графический интерфейс приложения обычно структурирован в виде дерева компонентов. Класс `Диалог` — это корень дерева, отображающий всё окно приложения. Диалог содержит `Панели`, которые, в свою очередь, могут

содержать либо другие вложенные панели, либо простые компоненты вроде `Кнопок`.

Простые `Компоненты` могут показывать небольшие подсказки, если к ним привязан текст помощи. Но есть и более сложные компоненты, для которых этот способ слишком прост. Они определяют собственный способ отображения помощи.



Пример вызова контекстной помощи в цепочке объектов UI.

Когда пользователь наводит указатель мыши на компонент и жмёт клавишу `F1`, приложение берёт компонент под курсором и шлёт ему запрос на показ помощи. Запрос путешествует по всем родителям компонента, пока не находится компонент, способный показать помощь.

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


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

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

  // Компонент показывает всплывающую подсказку, если
  // задан текст подсказки. В обратном случае, он
  // перенаправляет запрос 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 унаследует метод showHelp от своего родителя.
abstract class Container extends Component is
  protected field children: array of Component

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

```

```
// Примитивные компоненты может устраивать поведение помощи
// по умолчанию...
class Button extends Component is
  // ...

// Но сложные компоненты могут переопределять метод помощь
// по-своему. Но если помощь не может быть предоставлена,
// компонент вызовет базовую реализацию (см.
// класс Component)
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.wikiPage = "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 a 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 содержит несколько объектов, способных обработать тот или иной запрос, однако заранее неизвестно какой запрос придёт и какой обработчик понадобится.

Вы связываете потенциальных обработчиков в одну цепь и поочерёдно спрашиваете, хочет ли данный объект обработать запрос. Если нет, двигаетесь дальше по цепочке.

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

Цепочка обязанностей позволяет запускать обработчики последовательно один за другим в определённом порядке.

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 здесь 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, обратившись 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 он обработать запрос или нет?
   - Следует 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 собирают фабричные объекты исходя из конфигурации приложения или текущего окружения.

5. Клиент может посылать запросы любому обработчику в цепи, а не только первому. Запрос будет передаваться по цепочке 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 подписывать 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.