*Урок 6. Поведенческие паттерны*

Оглавление

[Введение 2](#_Toc90569255)

[Основные 2](#_Toc90569256)

[Цепочка ответственности 2](#_Toc90569257)

[Команда 4](#_Toc90569258)

[Посредник 8](#_Toc90569259)

[Наблюдатель 8](#_Toc90569260)

[Дополнительные 11](#_Toc90569261)

[Итератор 11](#_Toc90569262)

[Интерпретатор 11](#_Toc90569263)

[Хранитель 11](#_Toc90569264)

[Состояние 12](#_Toc90569265)

[Стратегия 12](#_Toc90569266)

[Шаблонный метод 14](#_Toc90569267)

[Посетитель 16](#_Toc90569268)

# Введение

Поведенческие паттерны определяют алгоритмы и способы реализации взаимодействия различных объектов и классов.

Рассмотрим **четыре** основных и **семь** дополнительных.

# Основные

## Цепочка ответственности

Переходим к каталогам паттернов.

*«Дать шанс обработать запрос нескольким участникам, связанным последовательно»*

У нас есть данные, их нужно перебрасывать из класса в класс, чтобы каждый их обрабатывал по-своему.

Встречается довольно редко. В соответствии с данным паттерном нам нужно что-то сделать, потом передать дальше.

Напоминает связный список, когда один элемент в своей структуре имеет ссылку на другой и т.д.

**Листинг 1. Урок 6. Коды к уроку/chain\_of\_responsibility.py**

|  |
| --- |
| **from** abc **import** ABCMeta, abstractmethod **from** random **import** choice, random   **class** Handler(metaclass=ABCMeta):  @abstractmethod  **def** handle(self, request):  **if** self.next **is not None**:  self.next.handle(request)   **def** link(self, next):  self.next = next  **return** self.next   **class** Request:  data = [  **'вопрос по возврату товара'**,  **'вопрос по скидке'**,  **'вопрос по стоимости товара'**,  **'вопрос по дефекту'**,  **'вопрос по новинке'**,  ]   **def** get\_data(self):  *# return random.sample(\_\_class\_\_.data, 1)[0]* **return** choice(\_\_class\_\_.data)   **class** Operator(Handler):  *# вероятность занятости оператора* probability = 0.99   **def** \_\_init\_\_(self, name):  self.name = name   **def** handle(self, request):  **if** self.is\_busy():  print(**f'Оператор {**self.name**} занят'**)  super().handle(request)  **else**:  print(**f'Оператор {**self.name**} обрабатывает: "{**request.get\_data()**}"'**)   **def** is\_busy(self):  **return** random() < \_\_class\_\_.probability   **class** BusyHandler(Handler):  **def** \_\_init\_\_(self):  self.request = **None   def** handle(self, request):  **if** self.request == request:  print(**'Все операторы заняты, пожалуйста подождите'**)  **else**:  self.request = request   super().handle(request)   handler = BusyHandler()  handler.link(Operator(**"#1"**)). \  link(Operator(**"#2"**)). \  link(Operator(**"#3"**)). \  link(Operator(**"#4"**)). \  link(handler)  *# генерируем поток из 3 запросов* **for** \_ **in** range(1):  handler.handle(Request()) |

Здесь **Handler** – абстрактный класс для элемента цепочки. У него есть метод **link()** – это привязка к следующему элементу цепочки. И еще метод **handle()**. Он принимает некоторый запрос и передает его дальше по цепочке.

У нас есть список **data** с некоторыми данными и есть метод **get\_data()**, который случайным образом извлекает элемент из этого списка.

Класс **Operator** имеет атрибут **probability**. Это некоторая вероятность занятости нашего элемента цепочки. В зависимости от этой вероятности оператор либо сам обрабатывает запрос, либо передает его дальше по цепочке.

В классе **Operator** мы переопределяем метод **handle()**.

Если оператор занят, то передаем запрос следующему оператору:

**if** self.is\_busy():  
 print(**f'Оператор {**self.name**} занят'**)  
 super().handle(request)

У нас в примере кольцевая реализация и как понять, что мы прошли круг?

Добавляется класс **BusyHandler**, который отвечает за регистрацию прохождения круга.

Он хранит в себе тот самый объект запроса (**request**) и проверяет, если пришел тот же самый **request**, то выпадает сообщение, что:

print(**'Все операторы заняты, пожалуйста подождите'**)

И снова запускает запрос по кругу:

super().handle(request)

А в первый раз он запоминает запрос:

self.request == request

Можно менять вероятность занятости оператора (участника цепочки) и получать разные результаты.

## Команда

*«Действие как объект»*

Позволяет:

* **Передавать как объект**
* **Логировать действия**
* **Ставить в очередь**
* **Откатывать операции**

Очень популярный паттерн.

Первая его часть представляет действие в качестве объекта. Например, «сходить в магазин», мы представляем объектом.

В Python этот паттерн уже реализован. Функция – является объектом.

Но этого мало, чтобы передавать команды. Нам нужны еще дополнительные условия, например, логировать действие, т.е. при выполнении функции писать что-то в лог или ставить в очередь – означает выполнять не сразу, а по очереди, откатывать операции.

Если у нас в программе есть какая-то из этих опций, можем успешно применить паттерн «Команда».

Вначале рассмотрим пример классической реализации этого паттерна.

**Листинг 2. Урок 6. Коды к уроку/command\_classic.py**

|  |
| --- |
| **from** abc **import** ABCMeta, abstractmethod   **class** CommandsInvoker:  **def** \_\_init\_\_(self):  self.\_commands\_list = []   **def** store\_command(self, command):  self.\_commands\_list.append(command)   **def** execute\_commands(self):  **for** command **in** self.\_commands\_list:  command.execute()   **class** Command(metaclass=ABCMeta):  **def** \_\_init\_\_(self, receiver):  self.\_receiver = receiver   @abstractmethod  **def** execute(self):  **pass   class** ActionCommand(Command):  **def** execute(self):  self.\_receiver.action()   **class** PauseCommand(Command):  **def** execute(self):  self.\_receiver.pause()   **class** CommandsReceiver:  **def** action(self):  print(**'action in receiver'**)   **def** pause(self):  print(**'pause in receiver'**)   *# пульт* commands\_invoker = CommandsInvoker()  *# внутренняя логика работа кнопок* commands\_receiver = CommandsReceiver()  *# кнопки* action\_command = ActionCommand(commands\_receiver) pause\_command = PauseCommand(commands\_receiver)  *# добавляем кнопки на пульт* commands\_invoker.store\_command(action\_command) commands\_invoker.store\_command(pause\_command)  commands\_invoker.execute\_commands() |

У нас есть три части.

Первая и самая главная – класс **CommandsInvoker**.

Это ядро, которое собственно и реализует паттерн «Команда». Это пульт, на который мы разместим наши кнопки.

Мы создаем список, где будут храниться наши команды:

self.\_commands\_list = []

То есть наши действия.

Делаем метод для добавления действий в список, т.е. добавления кнопок на пульт.

**def** store\_command(self, command):  
 self.\_commands\_list.append(command)

Делаем метод для выполнения действия, т.е. нажатия всех кнопок одновременно. Т.е. мы выполняем отсроченные команды.

**def** execute\_commands(self):  
 **for** command **in** self.\_commands\_list:  
 command.execute()

Вторая часть – класс **Command**.

У нас была некоторая процедура и мы ее превращаем в объект. Это наши кнопки.

Здесь есть параметр **receiver** – это механизм, связывающий кнопку с ее логикой. Создаем конкретные действия – кнопки.

**class** ActionCommand(Command):  
 **def** execute(self):  
 self.\_receiver.action()  
  
  
**class** PauseCommand(Command):  
 **def** execute(self):  
 self.\_receiver.pause()

Третья часть – класс **Receiver**. Это звено, связывающее кнопки с их механизмом работы.

**class** CommandsReceiver:  
 **def** action(self):  
 print(**'action in receiver'**)  
  
 **def** pause(self):  
 print(**'pause in receiver'**)

По сути здесь мы реализуем отсроченное выполнение команд – сразу всех вместе, одновременно.

Теперь рассмотрим и Python-реализацию.

**Листинг 3. Урок 6. Коды к уроку/command\_python.py**

|  |
| --- |
| **class** CommandsInvoker:  **def** \_\_init\_\_(self):  self.\_commands\_list = []   **def** store\_command(self, command):  self.\_commands\_list.append(command)   **def** execute\_commands(self):  **for** command **in** self.\_commands\_list:  print(**'execute'**, dir(command))  command()   **def** action():  print(**'action in receiver'**)   **def** pause():  print(**'pause in receiver'**)   **def** param\_command(param):  print(**f'console {**param**}'**)   **def** param\_close():  param\_command(**'my param'**)   **class** ParamClass:   **def** \_\_init\_\_(self, param):  self.param = param   **def** \_\_call\_\_(self, \*args, \*\*kwargs):  print(**f'console {**self.param**}'**)   command\_invoker = CommandsInvoker()  command\_invoker.store\_command(action) command\_invoker.store\_command(pause) command\_invoker.store\_command(param\_close) command\_invoker.store\_command(**lambda**: param\_command(**'my param'**))  command\_invoker.execute\_commands() |

В Python функции уже являются объектами, поэтому нет необходимости каждую функцию-команду представлять классом.

## Посредник

*«Определяет и координирует взаимодействие объектов»*

Это когда вместо двух классов вы делаете три и один из них – посредник.

Например, у нас есть два класса, они взаимодействуют между собой.

Если хотим сделать более гибкую систему – добавляем класс-посредник.

Примерно так отрабатывает концепция MVT.

## Наблюдатель

*«Создает механизм оповещения объектов об изменениях в других интересующих их объектах»*

Достаточно сложный паттерн.

Есть некоторый класс. В зависимости от изменения его состояния, могу поменяться другие объекты. Нужно как-то сообщить этим объектам, что им необходимо измениться. Компоненты могут не знать друг о друге и зависеть только от указанного выше состояния.

Нам нужно сделать так, чтобы можно было добавлять новые объекты, и они отслеживали бы изменение этого состояния.

**Листинг 4. Урок 6. Коды к уроку/observer.py**

|  |
| --- |
| **from** abc **import** ABCMeta, abstractmethod   **class** Subject:  **def** \_\_init\_\_(self):  self.\_observers = set()  self.\_subject\_state = **None   def** attach(self, observer):  observer.\_subject = self  self.\_observers.add(observer)   **def** detach(self, observer):  observer.\_subject = **None** self.\_observers.discard(observer)   **def** \_notify(self):  **for** observer **in** self.\_observers:  observer.update(self.\_subject\_state)   **class** Observer(metaclass=ABCMeta):  **def** \_\_init\_\_(self):  self.\_subject = **None** self.\_observer\_state = **None** @abstractmethod  **def** update(self, arg):  **pass   class** Sensor(Subject):  @property  **def** t(self):  **return** self.\_subject\_state   @t.setter  **def** t(self, t):  self.\_subject\_state = t  self.\_notify()   **class** SmsNotifier(Observer):   **def** update(self, arg):  **if** arg > 50:  print(**'send sms'**, **'куда так горячо!'**)   **class** DisplayObserver(Observer):  **def** update(self, arg):  print(**f'{**self.\_\_class\_\_.\_\_name\_\_**} temperature {**arg**}'**)   **class** HeaterObserver(Observer):  **def** \_\_init\_\_(self, low\_threshold, step):  super().\_\_init\_\_()  self.low\_threshold = low\_threshold  self.step = step   **def** update(self, arg):  **if** isinstance(self.\_subject, Sensor):  sensor = self.\_subject   t = sensor.t  delta\_low = t - self.low\_threshold   **if** delta\_low < 0:  t += self.step  print(**f'{**self.\_\_class\_\_.\_\_name\_\_**} heat impulse +{**self.step**}'**)  sensor.t = t   *# сенсором* sensor = Sensor()  *# подключаем наблюдателей за сенсором* sensor.attach(DisplayObserver()) sensor.attach(HeaterObserver(40, 20)) sensor.attach(SmsNotifier())  *# изменяем состояние сенсора* sensor.t = 20 |

Итак, у нас есть класс **Subject**. Это тот, чье состояние будет изменяться и за чем мы будем следить.

Например, заплатили за заказ и нужно уведомить менеджеров.

У этого класса есть набор наблюдателей.

self.\_observers = set()

И у класса есть некоторое состояние:

self.\_subject\_state = **None**

Добавление наблюдателя:

**def** attach(self, observer):  
 observer.\_subject = self  
 self.\_observers.add(observer)

Говорим, что наблюдатель подписан на класс **Subject**.

observer.\_subject = self

Удаление наблюдателя:

**def** detach(self, observer):  
 observer.\_subject = **None** self.\_observers.discard(observer)

Проходимся по подписчикам и у каждого вызываем update(), т.е. каждый наблюдатель реагирует на изменение состояния в **Subject**.

**def** \_notify(self):  
 **for** observer **in** self.\_observers:  
 observer.update(self.\_subject\_state)

Далее, есть класс-наблюдатель **Observer**.

С методом update().

Создаем конкретный класс, за которым мы наблюдаем:

**class** Sensor(Subject):

У него изменяется температура и мы вызываем notify().

self.\_subject\_state = t  
self.\_notify()

Далее идем к наблюдателям.

Простой наблюдатель – **DisplayObserver**.

Просто выводит температуру на экран. Arg – новое значение температуры.

Более сложный наблюдатель – **HeaterObserver**.

И т.д.

# Дополнительные

## Итератор

*«Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления»*

Реализован в Python.

Но можно сделать свой вариант итератора, перегрузив метод \_\_iter\_\_()

## Интерпретатор

*«Задаёт язык, определяет представление его грамматики, а также интерпретатор предложений этого языка»*

Очень редкий паттерн. Это по сути свой скриптовый язык.

Python предусматривает встроенный интерпретатор.

## Хранитель

*«Сохраняет и восстанавливает состояние объекта, не нарушая его инкапсуляции»*

Сейчас этот паттерн очень распространен и другое его название – сериализация.

Когда вы пишете API к сайту, вы используете этот паттерн.

Когда вы реализуете возможности json и pickle – это тоже паттерн Хранитель.

Позволяет сохранять состояние в независимый формат и потом его восстанавливать.

## Состояние

*«Позволяет объекту варьировать своё поведение в зависимости от внутреннего состояния»*

Поведение зависит от состояния (получил ошибку – вывел сообщение, успешный результат – сделал запись в файл).

## Стратегия

*«Позволяет использовать взаимозаменяемые алгоритмы»*

Похож на состояние. Здесь мы выбираем механизм действий, например, оплатить заказ картой или электронными деньгами.

**Листинг 5. Урок 6. Коды к уроку/strategy.py**

|  |
| --- |
| **from** abc **import** ABCMeta, abstractmethod   **class** PaymentStrategy(metaclass=ABCMeta):  @abstractmethod  **def** pay(self, amount):  **pass   class** PayPalPaymentStrategy(PaymentStrategy):  *# требуем учетку от paypal* **def** \_\_init\_\_(self, email, token):  self.email = email  self.token = token   **def** pay(self, amount):  print(**f'processing {**amount**} via PayPal account {**self.email**}'**)   **class** CreditCard:  **def** \_\_init\_\_(self, number):  self.\_number = number   **def** get\_number(self):  **return** self.\_number   **class** CreditCardPaymentStrategy(PaymentStrategy):  *# требуется кредитка* **def** \_\_init\_\_(self, card):  self.card = card   **def** pay(self, amount):  print(**f'processing {**amount**} via credit card {**self.card.get\_number()**}'**)   **class** Item:  **def** \_\_init\_\_(self, name, price):  self.name = name  self.price = price   **class** Order:  **def** \_\_init\_\_(self):  self.\_items = []   **def** pay(self, strategy):  total = self.get\_total()  strategy.pay(total)   **def** get\_total(self):  total = 0  **for** \_item **in** self.\_items:  total += \_item.price   **return** total   **def** add\_item(self, item):  self.\_items.append(item)   *# товары* item1 = Item(**"Book"**, 515) item2 = Item(**"Magazine"**, 298)  *# создаём и наполняем заказ* order = Order() order.add\_item(item1) order.add\_item(item2)  *# выбор конкретной стратегии и оплата заказа* paypal\_payment\_strategy = PayPalPaymentStrategy(**"patterns@geekbrains.com"**, **"token"**) order.pay(paypal\_payment\_strategy)  *# выбор конкретной стратегии и оплата заказа* credit\_card = CreditCard(**"1234 5678 9101 2131 4156"**) credit\_card\_payment\_strategy = CreditCardPaymentStrategy(credit\_card) order.pay(credit\_card\_payment\_strategy) |

У нас есть заказ, за который нужно заплатить.

order = Order()

Желательно разбить модель на две – сам заказ и стратегия оплаты.

В обычном варианте мы бы сделали реализацию выбора способа оплаты через многочисленные if-else.

Но это вариант негибкий, придется все время дорабатывать метод оплаты заказа.

Пусть стратегия отвечает за то, как мы платим, а **Order** от этого не зависит.

У нас есть класс-заказ **Order** со свойствами и методами.

При реализации метода pay() мы передаем в него стратегию оплаты, т.е. логика оплаты находится в стратегии оплаты.

У нас две стратегии-класса: **PayPalPaymentStrategy** и **CreditCardPaymentStrategy**.

Теперь мы можем заменять и стратегии, и заказы. И заказ не зависит от стратегии.

## Шаблонный метод

*«Определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом»*

**Листинг 6. Урок 6. Коды к уроку/template\_method.py**

|  |
| --- |
| **from** abc **import** ABCMeta, abstractmethod   **class** Notifier(metaclass=ABCMeta):  **def** \_\_init\_\_(self):  self.\_log\_list = []   **def** notify(self, address, subject, message):  self.\_login()  self.\_send(address, subject, message)  self.\_logout()  self.\_log(address, subject, message)   *# войти в ситему доставки сообщений* @abstractmethod  **def** \_login(self):  **pass** *# отправка сообщения* @abstractmethod  **def** \_send(self, address, subject, message):  **pass** *# выход* @abstractmethod  **def** \_logout(self):  **pass** *# внутреннее логирование, задаем поведение по умолчанию* **def** \_log(self, address, subject, message):  self.\_log\_list.append([address, subject, message])   **class** EmailNotifier(Notifier):  **def** \_\_init\_\_(self):  super().\_\_init\_\_()  self.mail\_from = **''   def** \_login(self):  *# no need to login* **pass   def** \_send(self, mail\_to, subject, message):  *# send\_mail(self.mail\_from, mail\_to, subject, message)* print(**f'send\_mail: {**mail\_to**}, {**subject**}, {**message**}'**)   **def** \_logout(self):  *# no need to logout* **pass   class** FacebookNotifier(Notifier):  **def** \_login(self):  *# login to facebook* print(**'login to facebook'**)   **def** \_send(self, address, subject, message):  *# send facebook message* print(**f'send facebook message: {**address**}, {**subject**}, {**message**}'**)   **def** \_logout(self):  *# logout from facebook* print(**'logout from facebook'**)   *# переопределяем поведение шага внутреннего логирования* **def** \_log(self, address, subject, message):  *# не будем внутренне логировать нотификацию по FB, ибо это избыточно.* **pass   class** NotifierFabric:  @staticmethod  **def** get\_notifier(communication\_type):  **if** communication\_type == **'EMAIL'**:  **return** EmailNotifier()  **elif** communication\_type == **'FACEBOOK'**:  **return** FacebookNotifier()  *# ...   # инстанциируем объект конкретного нотификатора, используя паттерн Фабричный метод* notifier\_1 = NotifierFabric.get\_notifier(**'EMAIL'**) notifier\_1.notify(**'patterns@geekbrains.ru'**, **'notify\_1'**, **'hello world'**) print(notifier\_1.\_\_dict\_\_)  *# инстанциируем объект конкретного нотификатора* notifier\_2 = NotifierFabric.get\_notifier(**'FACEBOOK'**) notifier\_2.notify(**'patterns\_facebook'**, **'notify\_2'**, **'hi'**) print(notifier\_2.\_\_dict\_\_) |

Есть базовый класс **Notifier** с набором методов.

А также классы-наследники **EmailNotifier** и **FacebookNotifier**.

В них есть переопределение некоторых базовых методов.

Очень похоже на обычное наследование, но это не совсем так! Почему?

Ключевая часть паттерна вот она:

**def** notify(self, address, subject, message):  
 self.\_login()  
 self.\_send(address, subject, message)  
 self.\_logout()  
 self.\_log(address, subject, message)

Четко прописана последовательность шагов, которые нам нужно выполнить.

И сам этот метод notify() мы переопределять не будем. Он четко задан и определяет алгоритм действий.

А вот уже приведенные в нем конкретные шаги мы будем переопределять. Т.е. последовательность шагов не меняется, а действия в рамках каждого шага мы можем менять

Тут еще добавляется фабричный метод.

**class** NotifierFabric:  
 @staticmethod  
 **def** get\_notifier(communication\_type):  
 **if** communication\_type == **'EMAIL'**:  
 **return** EmailNotifier()  
 **elif** communication\_type == **'FACEBOOK'**:  
 **return** FacebookNotifier()  
 *# ...*

На самом деле с этим паттерном вы уже встречались ранее в Django:

*# from django.views.generic import ListView  
#  
#  
# class MyListView(ListView):  
#  
# def get\_queryset(self):  
# return ...  
#  
# def get\_context\_data(self, \*, object\_list=None, \*\*kwargs):  
# pass*

## Посетитель

*«Позволяет определить новую операцию для иерархии классов, не изменяя сами классы»*

**Листинг 7. Урок 6. Коды к уроку/visitor.py**

|  |
| --- |
| **from** abc **import** ABCMeta, abstractmethod   **class** Human(metaclass=ABCMeta):  @abstractmethod  **def** accept(self, visitor):  **pass   class** Programmer(Human):   **def** accept(self, visitor):  visitor.repair(self)   **class** Lawyer(Human):  **def** accept(self, visitor):  visitor.repair(self)   **class** ConstructionElementVisitor(metaclass=ABCMeta):   @abstractmethod  **def** repair(self):  **pass   class** Cool(ConstructionElementVisitor):   @abstractmethod  **def** repair(self):  print(**'Дорого'**, **'Круто'**)   **class** NotCool(ConstructionElementVisitor):   @abstractmethod  **def** repair(self):  print(**'Дешево'**, **'Не Круто'**) |

Например, у нас есть программист и юрист – специалисты в своих областях.

Например, у каждого из них сломалась стиральная машина. Что делать?

Можно научиться ремонту, но это очень долго и сложно. И по сути означает расширение интерфейса классов **Programmer** и **Lawyer**.

Но мы не можем этого сделать и значит будем обращаться к специалисту по ремонту.

Специалистов может быть много и идея паттерна в том, что мы можем принять услуги любого ремонтника, независимо от того, какой у него возраст, где он живет и т.д.

Главное – его умение ремонтировать.

Получается гибкая система, ремонтники могут быть разные.