# Введение

*СЛАЙД 1-4*

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

Рассмотрим четыре основных и семь дополнительных.

*СЛАЙД 5*

**«Цепочка ответственности»**

*СЛАЙД 6*

**@Дать шанс обработать запрос нескольким участникам, связанным последовательно@**

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

Встречается довольно редко. По этому паттерну нам нужно что-то сделать, потом передать дальше.

Напоминает связный список, когда один элемент в своей структуре имеет ссылку на другой и т.д. Еще существуют списки двусвязанные и кольцевые.

**Листинг 1. chain\_of\_responsibility.py**

|  |
| --- |
| **import** abc **import** random   **class** Handler(metaclass=abc.ABCMeta):  @abc.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** random.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.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(3):  handler.handle(Request()) |

Здесь Handler – абстрактный класс для элемента цепочки. У него есть метод link – это привязка к следующему элементу. И еще метод handle(). Он выполняет некоторую обработку и передает ее дальше. В этом моменте и будут отличаться классы. Они будут делать разное.

У нас есть список data с данными и есть функция get\_data(), которая случайным образом извлекает элемент из этого списка.

Класс Operator имеет атрибут probability. Это некоторая вероятность занятости. В зависимости от этой вероятности он либо отвечает, либо перекидывает запрос дальше по цепочке.

В этом классе мы переопределяем метод handle().

Если оператор занят, то перекидываем следующему оператору:

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

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

Поэтому добавляется класс BusyHandler, который отвечает за то, факт регистрации прохождения круга.

Он хранит в себе request и проверяет, если пришел тот же самый request, то print(**'Все операторы заняты, пожалуйста подождите'**)

И начинает снова пускать запрос по кругу:

super().handle(request)

А в первый раз он запоминает запрос:

self.request = request

Можно менять вероятность занятости и получать разные результаты.

**«Команда»**

*СЛАЙД 7*

**@Действие как объект. Позволит:**

* **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 это уже реализовано. Функция – является объектом.

Но этого мало, чтобы передавать команды. Нам нужно еще дополнительные условия, например, логировать действие, т.е. при выполнении функции пишем что-то в лог.

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. command.py**

|  |
| --- |
| **import** abc   **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=abc.ABCMeta):  **def** \_\_init\_\_(self, receiver):  self.\_receiver = receiver   @abc.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'**)   commands\_receiver = CommandsReceiver()  action\_command = ActionCommand(commands\_receiver) pause\_command = PauseCommand(commands\_receiver)  commands\_invoker = CommandsInvoker()  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».

Мы в нем создаем список, где будут храниться наши команды:

self.\_commands\_list = []

То есть наши действия.

Делаем метод для добавления действия в список:

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

Делаем метод для выполнения действия:

**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.

Здесь есть параметр receiver – это тот, кто выполняет команду. Это абстрактный класс. От него мы наследуемся и создаем конкретные действия.

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

Третья часть – класс Receiver.

В нем команды просто записаны, как процедуры.

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

По сути здесь мы реализуем отсроченное выполнение команд.

Теперь рассмотрим и Питоновскую реализацию.

**Листинг 3. 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(ParamClass(**'my param'**)) command\_invoker.store\_command(**lambda**: param\_command(**'my param'**))  command\_invoker.execute\_commands() |

Здесь логика немного упростится. Но нам понадобится класс CommandsInvoker.

Мы знаем, что функция в Python является объектом, поэтому в класс CommandsInvoker мы вместо command.execute() можем просто поставить command()

То есть берем любой callable-объект, сохраняем и вызываем.

Берем наши команды просто как они есть. В Python функции – это сразу объекты.

**def** action():  
 print(**'action in receiver'**)  
  
  
**def** pause():  
 print(**'pause in receiver'**)

Также пропишем команды с параметрами:

**def** param\_command(param):  
 print(**f'console {**param**}'**)  
  
  
**def** param\_close():  
 param\_command(**'my param'**)

И тогда возникнет небольшая проблема:

У нас будет вызываться команда, но вызывается то она без параметра:

**def** execute\_commands(self):  
 **for** command **in** self.\_commands\_list:  
 print(**'execute'**, dir(command))  
 command()

И как быть?

Вариант первый: сделать замыкание.

**def** param\_command(param):  
 print(**f'console {**param**}'**)  
  
  
**def** param\_close():  
 param\_command(**'my param'**)

Вариант второй – более продвинутый. Это сделать класс с методом call.

**class** ParamClass:  
  
 **def** \_\_init\_\_(self, param):  
 self.param = param  
  
 **def** \_\_call\_\_(self, \*args, \*\*kwargs):  
 print(**f'console {**self.param**}'**)

Вариант третий – замыкание через лямбда-функцию:

command\_invoker.store\_command(**lambda**: param\_command(**'my param'**))

**«Посредник»**

*СЛАЙД 8*

**@Определяет и координирует взаимодействие объектов@**

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

Пример: риелтор.

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

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

Примерно так отрабатывает концепция MVT.

**«Наблюдатель»**

*СЛАЙД 9*

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

Достаточно сложный паттерн.

ИТАК:

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

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

*СЛАЙД 10*

Разберем паттерн на пальцах. У нас есть то, за чем мы следим – subject.

И на изменения субъекта могут подписываться наблюдатели (observer-ы).

Что-то меняется в субъекте, и он может уведомить об этом всех наблюдателей.

Как это реализовать?

Наблюдатели должны иметь одинаковый интерфейс. Который позволяет вызвать метод notify.

Субъект должен иметь возможность добавить себе observer-а.

Субъект сможет в любой момент сделать вызов метода notify и уведомить всех об изменениях.

**Листинг 4. observer.py**

|  |
| --- |
| **import** abc **import** time **import** random   **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=abc.ABCMeta):  **def** \_\_init\_\_(self):  self.\_subject = **None** self.\_observer\_state = **None** @abc.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   *# демо* sensor = Sensor()  *# подключаем наблюдателей за сенсором* sensor.attach(DisplayObserver()) sensor.attach(HeaterObserver(40, 20)) sensor.attach(SmsNotifier())  *# начальное значение* sensor.t = 20  *# цикл энтропии – естественное охлаждение сенсора* **for** \_ **in** range(5):  random\_t = random.random() \* 10  sensor.t = sensor.t - random\_t  time.sleep(0.5) |

Итак, у нас есть класс Subject. Это тот, кто будет изменяться.

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 этот 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

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

Далее, есть абстрактный класс Observer

С методом update (что нужно уведомить)

Создаем конкретный субъект:

**class** Sensor(Subject):

У него изменяется температура.

Меняем температуру и 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

Он будет нагревать нашу систему при понижении температуры.

У него есть нижняя граница - low\_threshold

И шаг, на который нагревать - step

Меняется температура -> нагреватель начинает нагревать и снова меняется температура.

ПЕРЕХОДИМ К ДРГИМ СЕМИ ПОВЕДЕНЧЕСКИМ ПАТТЕРНАМ

*СЛАЙД 11-12*

**«Итератор»**

*СЛАЙД 13*

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

Реализован в Python.

Но можно сделать свой вариант, переопределив метод \_\_iter\_\_

Можно написать свой итератор.

**«Интерпретатор»**

*СЛАЙД 14*

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

Очень редкий паттерн. Это по сути свой скриптовый язык.

У нас есть система, в которой есть правила и которые проще описать своим скриптовым языком.

Пример – игровые приложения. Скрипты на C#, а сам код на С++.

**«Хранитель»**

*СЛАЙД 15*

**@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.

Можем сохранять состояние в независимый формат и потом его восстанавливать.

**«Состояние»**

*СЛАЙД 16*

**@Позволяет объекту варьировать своё поведение в зависимости от внутреннего состояния@**

Поведение зависит от состояния (пришел голодный и потребовал от жены обед, пришел сытый – и ушел спать)

**«Стратегия»**

*СЛАЙД 17*

**@Позволяет использовать взаимозаменяемые алгоритмы @**

Похож на состояние. Здесь мы выбираем механизм действий, например, потерпеть.

Это по сути композиция – уже знакомый паттерн.

Только раньше мы говорили о композиции по структуре, а теперь – по поведению.

**Листинг 5. strategy.py**

|  |
| --- |
| **import** abc   **class** PaymentStrategy(metaclass=abc.ABCMeta):  @abc.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 это через if-ы.

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.

В паттерне стратегия все примерно также, только мы говорим не о стратегии, а о состоянии.

**«Шаблонный метод»**

*СЛАЙД 18*

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

**Листинг 6. template\_method.py**

|  |
| --- |
| **import** abc   **class** Notifier(metaclass=abc.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* @abc.abstractmethod  **def** \_login(self):  **pass** *# отправка сообщения* @abc.abstractmethod  **def** \_send(self, address, subject, message):  **pass** *# выход* @abc.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\_\_)   *# from django.views.generic import ListView # # # class MyListView(ListView): # # def get\_queryset(self): # return ... # # def get\_context\_data(self, \*, object\_list=None, \*\*kwargs): # pass* |

Есть класс Notifier с набором методов.

А также классы 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 прописаны шаги того, что нам нужно сделать.

И сам этот метод notify мы переопределять не будем. Он четко задан.

А вот уже приведенные в нем конкретные шаги он позволяет переопределить.

Получается пользователю мы задали изначально структуру, и она не должна меняться.

Пользователю не нужно знать, как происходит notify, он лишь переопределяет конкретные шаги в этом методе notify.

Мы фиксируем как должно быть и даем переопределить каждый конкретный кусок.

Получается, что сама процедура жестко задана, а шаги – гибко.

Тут еще добавляется фабричный метод.

НА самом деле с этим 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»**

*СЛАЙД 19*

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

Разберем его на пальцах.

Например, я – питон разработчик, а также умею делать ремонт, чинить машину, готовить, крестиком вышивать.

И вдруг у меня сломалась стиральная машина. Ее я чинить не умею. Что делать?

Вот эти мои умения – аналог интерфейса класса.

Чтобы починить машину мне нужно расширить свой интерфейс – научиться чинить стиральную машину.

Т.е. есть некоторая система, которая уже много чего умеет, например, класс. И по каким-то причинам мы не можем расширять ее интерфейс.

Мы берем и зовем ремонтников и даже паттерн называется «Визитор».

Т.е. мы не расширяем интерфейс своего класса. А создаем новый класс, который решит нашу проблему.

А в старом классе сделаем вызов Визитора.

Получается гибкая система, ремонтники могут быть разные.

**Листинг 7. visitor.py**

|  |
| --- |
| **import** abc   **class** ConstructionElement(metaclass=abc.ABCMeta):  @abc.abstractmethod  **def** accept(self, visitor):  **pass   class** LeverElement(ConstructionElement):  **def** accept(self, visitor):  visitor.visit\_lever\_element(self)   **class** GearElement(ConstructionElement):  **def** accept(self, visitor):  visitor.visit\_gear\_element(self)   **class** ConstructionElementVisitor(metaclass=abc.ABCMeta):  @abc.abstractmethod  **def** visit\_lever\_element(self, lever\_element):  **pass** @abc.abstractmethod  **def** visit\_gear\_element(self, gear\_element):  **pass   class** StrengthCalculatorVisitor(ConstructionElementVisitor):  **def** visit\_lever\_element(self, lever\_element):  print(**f'do Strength Calculation for {**lever\_element**}'**)   **def** visit\_gear\_element(self, gear\_element):  print(**f'do Strength Calculation for {**gear\_element**}'**)   **class** SpecificationCalculatorVisitor(ConstructionElementVisitor):  **def** visit\_lever\_element(self, lever\_element):  print(**f'do Specification Calculation for {**lever\_element**}'**)   **def** visit\_gear\_element(self, gear\_element):  print(**f'do Specification Calculation for {**gear\_element**}'**)   strength\_calculator\_visitor = StrengthCalculatorVisitor() specification\_calculator\_visitor = SpecificationCalculatorVisitor()  lever\_1 = LeverElement() gear\_1 = GearElement()   lever\_1.accept(strength\_calculator\_visitor) lever\_1.accept(specification\_calculator\_visitor)  gear\_1.accept(strength\_calculator\_visitor) gear\_1.accept(specification\_calculator\_visitor) |

У нас есть абстрактный класс, в котором реализуем метод accept

И есть два конкретных класса:

**class** LeverElement(ConstructionElement):  
 **def** accept(self, visitor):  
 visitor.visit\_lever\_element(self)  
  
  
**class** GearElement(ConstructionElement):  
 **def** accept(self, visitor):  
 visitor.visit\_gear\_element(self)

Но мы принимаем визитора и у него вызываем нужный метод.

Есть сам Визитор с конкретными методами. Это уже будут процедуры:

**class** ConstructionElementVisitor(metaclass=abc.ABCMeta):  
 @abc.abstractmethod  
 **def** visit\_lever\_element(self, lever\_element):  
 **pass** @abc.abstractmethod  
 **def** visit\_gear\_element(self, gear\_element):  
 **pass**

После этого делаем конкретные Визиторы и они реализуют конкретные методы.

Получается очень гибкая система. Один элемент, например, lever\_1

Может посетить два разных Визитора.

Применяется, когда мы больше не можем расширять интерфейс класса. Например, есть 10 абстрактных методов уже. И новых добавить будет проблематично.

Пример с ремонтниками:

**Листинг 8. visitor\_example.py**

|  |
| --- |
| **import** abc   **class** Human(metaclass=abc.ABCMeta):  @abc.abstractmethod  **def** accept(self, visitor):  **pass   class** Proger(Human):   **def** accept(self, visitor):  visitor.repair\_car(self)   **class** Yurist(Human):  **def** accept(self, visitor):  visitor.repair(self)   **class** ConstructionElementVisitor(metaclass=abc.ABCMeta):   @abc.abstractmethod  **def** repait(self):  **pass** @abc.abstractmethod  **def** repait\_car(self):  **pass   class** Cool(ConstructionElementVisitor):   @abc.abstractmethod  **def** repait(self):  print(**'Дорого'**, **'Круто'**)   **class** NotCool(ConstructionElementVisitor):   @abc.abstractmethod  **def** repait(self):  print(**'Дешево'**, **'Не Круто'**)   *# class Category: # pass # # # for item in Category():* |