# Введение

*СЛАЙД 1-2*

Сегодня на повестке структурные паттерны.

*СЛАЙД 3-4*

Определяют, как из классов и объектов образуются более крупные структуры.

*СЛАЙД 5*

Но для начала рассмотрим один пример, который нам пригодится в этой теме.

**Листинг 1. lesson.py**

|  |
| --- |
| *# Композиция vs Наследование  # Наследование* **class** Animal:   **def** say(self):  **pass** *# Кошка ЯВЛЯЕТСЯ Животным ? Да* **class** Cat(Animal):   **def** say(self):  **pass   class** Engine:   **def** move(self):  print(**'Move'**)   *# Машина не является двигателем* **class** Car(Engine):  **pass** car = Car() car.move()   *# Композиция # Двигатель это часть машины* **class** Car:   **def** \_\_init\_\_(self, engine):  self.engine = engine   **def** change\_engine(self, engine):  self.engine = engine   engine = Engine() car = Car(engine)  car.engine.move() |

У нас есть класс Животное и от него мы наследуем класс Кошка.

Очень часто мы реализуем наследование с одной целью – заимствовать некоторый функционал родителя.

Но это может принести нам много проблем в будущем и их придется решать.

Здесь нужно запомнить важное слово – является. Кошка является животным.

Если ответ – ДА, то наследование можно использовать. Если нет, то нельзя.

Но почему так?

Рассмотрим еще пример. У нас есть двигатель – Engine. И он умеет ехать. У нас есть класс Машина, она тоже может ехать. Унаследуем ее от движка.

Сначала это вроде работает, но потом превратится в ужасную картину. Почему?

Потому что машина не является двигателем. Значит наследование здесь – не комильфо.

Как быть, если машина не является двигателем?

Самое время поговорить о композиции!

Композиция означает, что один объект состоит из других. В нашем примере – двигатель – часть машины.

Логика такая: у нас есть класс и мы в него просто передаем другой объект.

**def** \_\_init\_\_(self, engine):

Получается: у машины есть движок, который может ехать.

car.engine.move()

Большинство структурных паттернов основаны именно на композиции. Композиция – лучше наследования.

*СЛАЙД 6-7*

*«Адаптер»*

«Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты»

Классом может быть отдельная программа, микросервис и т.д.

Например, у нас розетка 220 вольт, приезжаем в Индию, там 110 вольт.

Дырки в розетке – интерфейс.

В классе интерфейс – это набор атрибутов и методов.

Вилки в разные розетки мы втолкнуть не сможем.

Представим, что у нас есть ноутбук.

Если бы не было адаптеров, пришлось бы покупать новый ноут.

Адаптер – это переходник, поддерживающий другой интерфейс.

Как же это выглядит в программе?

Например, у нас есть класс с одним интерфейсом: методы a и b.

И есть класс с методами c и d.

Эти классы с разным интерфейсом. Как их использовать одинаково?

Мы можем сделать для класса адаптер с недостающим методом.

Мы обращаемся к адаптеру, а адаптер – к нашему классу.

Адаптеры бывают двух типов: **адаптер класса** и **адаптер объекта**.

**Листинг 2. adapter\_1.py**

|  |
| --- |
| **import** abc **import** math   *# нечто круглое, имеющее радиус* **class** Roundable(metaclass=abc.ABCMeta):  @abc.abstractmethod  **def** get\_radius(self):  **pass** *# окружность - имеет радиус* **class** Circle(Roundable):  **def** \_\_init\_\_(self, radius):  self.\_radius = radius   **def** get\_radius(self):  **return** self.\_radius   *# квадрат со стороной side* **class** Square:  **def** \_\_init\_\_(self, side):  self.\_side = side   **def** get\_side(self):  **return** self.\_side   *# круглый квадрат* **class** RoundableSquare(Square, Roundable):  **def** get\_radius(self):  **return** self.get\_side() \* math.sqrt(2) / 2   circle\_1 = Circle(5) roundable\_square\_1 = RoundableSquare(5)  print(circle\_1.get\_radius()) print(roundable\_square\_1.get\_radius())  print(issubclass(circle\_1.\_\_class\_\_, Roundable)) print(issubclass(roundable\_square\_1.\_\_class\_\_, Roundable)) |

В этом примере есть интерфейс Roundable c методом get\_radius. Есть конкретный класс – Circle и у него есть метод get\_radius. Есть класс Square с атрибутом side и методом get\_side. У этого класса другой интерфейс – у него нет метода get\_radius.

Представим, что у нас есть список фигур – кругов и квадратов и для каждой требуется вызывать метод get\_radius. Но не у всех классов в нашем примере есть такой интерфейс.

Мы берем и делаем класс «RoundableSquare» (круглый квадрат).

Мы указываем у него:

**class** RoundableSquare(Square, Roundable):

Здесь Square – класс, который нужно привести к новому интерфейсу.

Roundable – интерфейс, который мы хотим реализовать.

Хотя в Python Roundable – здесь это необязательно. Но желательно. Так будет более строго.

Теперь мы можем создать экземпляр круглого квадрата – RoundableSquare

Теперь рассмотрим адаптер на объектах. Он используется чаще.

**Листинг 3. adapter\_2.py**

|  |
| --- |
| **import** abc **import** math   *# нечто круглое, имеющее радиус* **class** Roundable(metaclass=abc.ABCMeta):  @abc.abstractmethod  **def** get\_radius(self):  **pass** *# окружность - имеет радиус* **class** Circle(Roundable):  **def** \_\_init\_\_(self, radius):  self.\_radius = radius   **def** get\_radius(self):  **return** self.\_radius   *# квадрат со стороной side* **class** Square:  **def** \_\_init\_\_(self, side):  self.\_side = side   **def** get\_side(self):  **return** self.\_side   *# адаптер квадрата к круглым фигурам* **class** RoundableSquareAdapter(Roundable):  **def** \_\_init\_\_(self, adaptee):  self.\_adaptee = adaptee   *# радиус квадрата - как радиус описанной окружности* **def** get\_radius(self):  **return** self.\_adaptee.get\_side() \* math.sqrt(2) / 2   *# создаем класс для сортировки объектов Roundable* **class** SortRoundable(RoundableSquareAdapter):  **def** compare\_order(self):  **if** issubclass(self.\_\_class\_\_, Roundable):  **return** self.get\_radius()  **else**:  **return** RoundableSquareAdapter(self).get\_radius()   *# список окружностей и квадратов* figures\_1 = [Circle(5), Square(5), Circle(2), Square(2)]   **def** compare\_order(item):  **if** issubclass(item.\_\_class\_\_, Roundable):  **return** item.get\_radius()  **else**:  **return** RoundableSquareAdapter(item).get\_radius()   *# отсортированный список окружностей и квадратов* ordered\_figures = sorted(figures\_1, key=SortRoundable.compare\_order)  *# выводим значения радиуса для фигур* **for** item **in** ordered\_figures:  **if** issubclass(item.\_\_class\_\_, Roundable):  print(item.get\_radius())  **else**:  print(RoundableSquareAdapter(item).get\_radius()) |

Тоже есть что-то круглое (абстрактный класс). А также конкретные классы – Круг и Квадрат.

И здесь мы в адаптер:

**class** RoundableSquareAdapter(Roundable):  
 **def** \_\_init\_\_(self, adaptee):  
 self.\_adaptee = adaptee

передаем ту штуку, которую требуется адаптировать.

В нашем случае, это квадрат.

Здесь мы уже используем композицию вместо наследования.

У нас есть список фигур:

figures\_1 = [Circle(5), Square(5), Circle(2), Square(2)]

Их нужно отсортировать по радиусу.

ordered\_figures = sorted(figures\_1, key=SortRoundable.compare\_order)

Если у нас что-то круглое, то мы просто берем у этого круглого радиус.

print(item.get\_radius())

Если у нас что-то не круглое, то мы берем наш адаптер

**return** RoundableSquareAdapter(item).get\_radius()

Такой необычный прием, т.к. пример взят из языка со статической типизацией.

**class** SortRoundable(RoundableSquareAdapter):

*СЛАЙД 9*

*«Декоратор»*

«Динамически добавляет объекту новые обязанности.»

Рассмотрим классический пример декоратора на объектах.

**Листинг 4. decorator\_example.py**

|  |
| --- |
| **import** abc   **class** Writer(metaclass=abc.ABCMeta):  @abc.abstractmethod  **def** write\_message(self):  **pass   class** ConcreteWriter(Writer):  **def** write\_message(self):  print(**'writing message'**)   **class** WriterDecorator(Writer, metaclass=abc.ABCMeta):  **def** \_\_init\_\_(self, component):  self.\_component = component   @abc.abstractmethod  **def** write\_message(self):  **pass   class** CheckLengthDecorator(WriterDecorator):  **def** write\_message(self):  print(**'checking message length'**)  self.\_component.write\_message()   **class** CompressDecorator(WriterDecorator):  **def** write\_message(self):  print(**'compressing message'**)  self.\_component.write\_message()  print(**'check compressed length'**)   concrete\_writer = ConcreteWriter() check\_length\_decorator = CheckLengthDecorator(concrete\_writer) compress\_decorator = CompressDecorator(check\_length\_decorator) compress\_decorator.write\_message()   *# На функциях* **def** old():  print(**'one'**)   **def** decorator(old\_f):  **def** inner(\*args, \*\*kwargs):  print(**'two'**)  **return** old\_f(\*args, \*\*kwargs)   **return** inner   old = decorator(old) old = decorator(old) old = decorator(old) old() |

У нас есть интерфейс Writer, который умеет писать сообщение.

Есть конкретный писатель:

ConcreteWriter

У нас есть абстрактный класс WriterDecorator, который наследуется от класса Writer. Это довольно важно, поскольку у декоратора должен быть такой же интерфейс, как у старого объекта.

И вот начинается композиция. Мы в декоратор передаем объект, который хотим декорировать.

**def** \_\_init\_\_(self, component):  
 self.\_component = component

Далее идут CheckLengthDecorator и CompressDecorator

Это уже наши конкретные декораторы.

*СЛАЙД 11*

*«Заместитель»*

«Позволяет сослаться на объект более изощрённо, чем это возможно с простым указателем»

По английски, - прокси.

- Ленивая инициализация. Объект создается не сразу, а когда мы к нему обращаемся.

- Защита доступа. Защитный прокси.

- Кэширование. Мы обращаемся не к объекту, а к его кэшу.

- Преобразование протоколов. Удаленный прокси. Преобразование одного протокола в другой.

- Логгирование запросов.

ПЕРЕД ТЕМ, КАК РАЗБИРАТЬ ЗАМЕСТИТЕЛЯ, ПРОЙДЕМ ПО ВСЕМ паттернам этого урока.

1. Адаптер. Он брал объект и приводил его к другому интерфейсу.
2. Декоратор. Тоже брал объект, но интерфейс сохранял и добавлял новый функционал.

Прокси похож на декоратора, но он замещает, а не дополняет объект.

**Листинг 5. proxy.py**

|  |
| --- |
| **import** abc   **class** CurrencyRateService(metaclass=abc.ABCMeta):  @abc.abstractmethod  **def** get\_currency\_rate(self, currency):  **pass   class** CbrCurrencyRateService(CurrencyRateService):  **def** get\_currency\_rate(self, currency):  *# ... особенности реализации опущены* **return** 0.57   **class** ProxyCurrencyRateService(CurrencyRateService):  **def** \_\_init\_\_(self):  *# ссылка на реальный сервис* self.currencyRateService = CbrCurrencyRateService()   *# кэш курсов* self.rates = dict()   **def** get\_currency\_rate(self, currency):  **if** (currency **in** self.rates.keys()):  *# если курс уже имеется в кэше, выдать из кэша* print(**f'{**currency**}: from cache'**)  **return** self.rates[currency]  **else**:  *# если еще нет, то запросить реальный (медленный) сервис* print(**f'{**currency**}: from service'**)  rate = self.currencyRateService.get\_currency\_rate(currency)  self.rates.update({currency: rate})  **return** rate   *# создаем сервис* currency\_rate\_service = ProxyCurrencyRateService()  *# получаем курс из кэша или от цб - это уже решает прокси* yen\_rate\_request\_1 = currency\_rate\_service.get\_currency\_rate(**'yen'**) print(yen\_rate\_request\_1)  yen\_rate\_request\_2 = currency\_rate\_service.get\_currency\_rate(**'yen'**) print(yen\_rate\_request\_2) |

У нас есть абстрактный класс CurrencyRateService с методом get\_currency\_rate

Есть конкретная реализация этого класса – CbrCurrencyRateService

Наша задача – реализовать для него кэш.

Мы создаем класс ProxyCurrencyRateService

Наследуем его от класса CurrencyRateService

Поэтому интерфейс совпадает с интерфейсом родителя.

Если в декораторе мы передавали ссылку на компонент прямо в init, то здесь внутри инита создаем экземпляр класса CbrCurrencyRateService

**def** \_\_init\_\_(self):  
 *# ссылка на реальный сервис* self.currencyRateService = CbrCurrencyRateService()  
  
 *# кэш курсов* self.rates = dict()

Почему так? Потому что в декораторе есть две сущности – компонент и его декоратор. Мы можем отдельно использовать компонент, потом декорировать и снова использовать.

А в прокси есть только один объект. Мы либо работаем с самим объектом, либо с его прокси и нам не нужно с ними обоими работать. Класс Прокси замещает реальный класс.

Таким образом, клиент не знает, с каким классом он работает – то ли с исходным, то ли с его прокси.

А в декораторе мы точно знаем, что нам нужно декорировать компонент.

Декоратор – декорирование объекта

Прокси – полное замещение объекта.

Теперь по реализации:

Мы делаем некоторый кэш курсов:

self.rates = dict()

Метод

get\_currency\_rate

выглядит следующим образом: если курс в кэше, берем из кэша, если нет, то берем старый объект и вызываем у него: get\_currency\_rate

rate = self.currencyRateService.get\_currency\_rate(currency)

и потом записываем в кэш.

Клиент не заметит разницы по классам – исходный или прокси.

*СЛАЙД 10*

*«Фасад»*

«Определяет интерфейс более высокого уровня, который упрощает использование подсистемы»

Очень простой, удобный и полезный паттерн.

Он делает обертку над какой-то частью системы, даже над несколькими частями и предоставляет пользователю более удобный интерфейс.

У нас во фреймворке, например, может быть 1000 классов, но сам класс Application – он фасад и пользователю не нужно знать о всех внутренних реализованных классах.

**Листинг 6. facade.py**

|  |
| --- |
| **class** Site1Checker:  **def** russian\_auto(self):  print(**'prices of russian cars on site 1'**)   **def** foreign\_auto(self):  print(**'prices of foreign cars on site 1'**)   **class** Site2Checker:  **def** russian\_auto(self):  print(**'prices of russian cars on site 2'**)   **def** foreign\_auto(self):  print(**'prices of foreign cars on site 2'**)   **class** FacadeSiteChecker:  **def** \_\_init\_\_(self):  self.\_subsys\_1 = Site1Checker()  self.\_subsys\_2 = Site2Checker()   **def** russian\_auto(self):  self.\_subsys\_1.russian\_auto()  self.\_subsys\_2.russian\_auto()   **def** foreign\_auto(self):  self.\_subsys\_1.foreign\_auto()  self.\_subsys\_2.foreign\_auto()   facade\_site\_checker = FacadeSiteChecker() facade\_site\_checker.russian\_auto() facade\_site\_checker.foreign\_auto() |

Есть класс Site1Checker, на котором есть машинки, и мы оттуда получаем русские авто и зарубежные. И второй сайт такой же.

Но если клиенту без разницы, с какого сайта эти авто получать, ему главное получить эти машинки.

То мы можем объединить эти классы в один FacadeSiteChecker.

Этот сайт более высокого уровня. Первый, например, берет с авито, второй с другого сайта. А третий – большой, просто ищет все машины.

И клиенты будут пользоваться только методами:

**def** russian\_auto(self):  
 self.\_subsys\_1.russian\_auto()  
 self.\_subsys\_2.russian\_auto()  
  
 **def** foreign\_auto(self):  
 self.\_subsys\_1.foreign\_auto()  
 self.\_subsys\_2.foreign\_auto()

И им без разницы, с какого сайта взято.

*СЛАЙД 8*

*«Компоновщик»*

«**Компонует объекты в древовидные иерархические структуры для представления иерархий ‘часть — целое‘**»

Стоит немного особняком от других. Применяется к рекурсивным объектам, т.е. к древовидным объектам.

**Листинг 8. composite.py**

|  |
| --- |
| **import** abc   **class** Component(metaclass=abc.ABCMeta):  @abc.abstractmethod  **def** operation(self):  **pass   class** MachineOperation(Component):  **def** \_\_init\_\_(self, name):  self.name = name   **def** operation(self):  print(self.name)   **class** CompositeOperation(Component):  **def** \_\_init\_\_(self):  self.\_child = set()   **def** operation(self):  **for** child **in** self.\_child:  child.operation()   **def** append(self, component):  self.\_child.add(component)   **def** remove(self, component):  self.\_child.discard(component)   *# инициализация операций* operation\_1 = MachineOperation(**'drill 5 mm'**) operation\_2 = MachineOperation(**'drill 15 mm'**) composite\_1 = CompositeOperation() composite\_1.append(operation\_1) composite\_1.append(operation\_2)  operation\_3 = MachineOperation(**'assemble'**) operation\_4 = MachineOperation(**'paint'**) composite\_2 = CompositeOperation() composite\_2.append(composite\_1) composite\_2.append(operation\_3) composite\_2.append(operation\_4)  *# использование разных по структуре операций идентично* composite\_2.operation() *# operation\_1.operation()* |

Класс MachineOperation просто что-то делает, а класс CompositeOperation – делает, плюс еще добавляет компоненты (это объекты и MachineOperation, и CompositeOperation)

CompositeOperation – аналогия папки

Component – аналогия файла

Важные моменты:

1. Оба класса реализуют один и тот же интерфейс. У обоих значит есть метод operation.
2. У конкретной операции берем и реализуем конкретные действия. А у к CompositeOperation перебираем все компоненты, мы не знаем файл или папка и вызываем метод operation()

Результат:

folder

paint

assemble

folder

drill 5 mm

drill 15 mm

ИТАК:

Мы разобрали:

1. Адаптер. Берет объект, для того чтобы привести его к другому интерфейсу.
2. Мост.
3. Компоновщик. Работает с иерархическими древовидными структурами и делает одинаковый интерфейс у узла и листа.
4. Декоратор. Берет объект и добавляет в него что-то новое.
5. Фасад. Группирует объекты в более абстрактные структуры.
6. Замещает какой-то объект, добавляя новое поведение, либо изменяя его.