*Урок 5. Структурные паттерны*

Оглавление

[Введение 2](#_Toc90394485)

[Адаптер 3](#_Toc90394486)

[Адаптер класса 4](#_Toc90394487)

[Адаптер объекта 5](#_Toc90394488)

[Декоратор 6](#_Toc90394489)

[Заместитель (прокси) 8](#_Toc90394490)

[Фасад 11](#_Toc90394491)

[Компоновщик 12](#_Toc90394492)

[Выводы 14](#_Toc90394493)

# Введение

Сегодня на повестке структурные паттерны.

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

Перед тем как приступить к разбору самих паттернов, нам нужно коснуться двух важных терминов: наследование и композиция. Чем они отличаются и где используются. Потому как текущая тема непосредственно связана с этими терминами, прежде всего с композицией.

**Листинг 1. Урок 5. Коды к уроку/inheritance\_vs\_composition.py**

|  |
| --- |
| *# Наследование vs Композиция   # Наследование* **class** Animal:  **def** say(self):  **pass   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**.

Здесь нужно запомнить важное слово – «является». Кошка является животным?

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

Рассмотрим еще пример. У нас есть класс **Двигатель** и класс **Машина**. Унаследуем машину от двигателя. Но это будет некорректно.

Потому что машина не является двигателем.

И что делать?

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 **Машина** и в конструктор мы передаем ссылку на другой объект, например, двигатель. Ведь машина в том числе состоит и из двигателя.

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

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

# Адаптер

Переходим к каталогам паттернов.

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

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

Пример из жизни: мы приезжаем отдыхать в Индию и хотим выполнить зарядку телефона, но там напряжение в сети составляет 110 В. И зарядку мы выполнить никак не можем. Там придется перепаивать наше зарядное устройство под нужное напряжение, но мы этого не можем сделать, т.е. не можем просто так взять и переделать интерфейс нашего устройств. Что делать? Покупать новые телефон?

А решение очень простое – всего на всего нужно воспользоваться адаптером.

Адаптер – это переходник, поддерживающий другой интерфейс.

Как же это выглядит в программе?

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

И есть класс **E** с методами **c** и **d**.

Эти классы с разным интерфейсом. Как, например, через объект класса **C** вызвать метод **d**?

Вмешаться явно в логику класса **C** мы не можем.

Мы можем сделать для класса адаптер с недостающим методом.

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

Мы обращаемся к адаптеру, а адаптер – к классу, чей интерфейс нужно получить.

Адаптер класса

**Листинг 2. Урок 5. Коды к уроку/class\_adapter.py**

|  |
| --- |
| **from** abc **import** ABCMeta, abstractmethod **from** math **import** sqrt   *# нечто круглое, имеющее радиус* **class** Roundable(metaclass=ABCMeta):  @abstractmethod  **def** get\_radius(self):  **pass** *# конкретный класс - 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 (вписанная окружность)* **class** RoundableSquare(Square, Roundable):  **def** get\_radius(self):  **return** self.get\_side() \* 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 абстрактный класс **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 создаем класс **RoundableSquare**.

Мы указываем у него:

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

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

**Roundable** – интерфейс, к которому мы хотим привести наш класс **Square**.

Теперь мы можем создать экземпляр класса-адаптера – **RoundableSquare**.

В данном случае мы опираемся на наследование.

Адаптер объекта

Теперь рассмотрим адаптер на объектах. Он используется чаще.

**Листинг 3. Урок 5. Коды к уроку/object\_adapter.py**

|  |
| --- |
| **from** abc **import** ABCMeta, abstractmethod **from** math **import** sqrt   *# нечто круглое, имеющее радиус* **class** Roundable(metaclass=ABCMeta):  @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** RoundableAdapter(Roundable):  **def** \_\_init\_\_(self, adaptee):  self.\_adaptee = adaptee  print(self.\_adaptee)   *# радиус квадрата - как радиус описанной окружности* **def** get\_radius(self):  **return** self.\_adaptee.get\_side() \* sqrt(2) / 2   *# список окружностей и квадратов* figures\_1 = [Circle(5), Square(5), Circle(2), Square(2)] |

Тоже есть что-то круглое (абстрактный класс **Roundable**). А также конкретные классы – **Circle** и **Square**.

И здесь мы в адаптер:

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

Передаем тот компонент, который требуется адаптировать.

В нашем случае, это квадрат.

Здесь мы уже используем композицию вместо наследования.

# Декоратор

*«Динамически добавляет объекту новые обязанности»*

После курса «Клиент-серверные приложения» мы знаем, как функционируют декораторы в Python.

**Листинг 4. Урок 5. Коды к уроку/decorator\_python.py**

|  |
| --- |
| *# На функциях* **def** decorator(old\_f):  **def** inner(\*args, \*\*kwargs):  print(**'two'**)  **return** old\_f(\*args, \*\*kwargs)   **return** inner   @decorator **def** old():  print(**'one'**)   old = decorator(old) old() |

Рассмотрим классический пример декоратора на объектах.

**Листинг 5. Урок 5. Коды к уроку/decorator\_base.py**

|  |
| --- |
| **from** abc **import** ABCMeta, abstractmethod   **class** Writer(metaclass=ABCMeta):  @abstractmethod  **def** write\_message(self):  **pass   class** ConcreteWriter(Writer):  **def** write\_message(self):  print(**'writing message'**)   **class** WriterDecorator(Writer, metaclass=ABCMeta):  **def** \_\_init\_\_(self, component):  self.\_component = component   @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()  **""" print('compressing message') print('checking message length') print('writing message') print('check compressed length') """** |

У нас есть класс **Writer**, который умеет писать сообщение.

Есть конкретный класс-писатель:

**class** ConcreteWriter(Writer):  
 **def** write\_message(self):  
 print(**'writing message'**)

У нас есть абстрактный класс **WriterDecorator**, который наследуется от класса **Writer**.

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

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

Далее идут классы **CheckLengthDecorator** и **CompressDecorator**.

Это уже наши конкретные декораторы.

# Заместитель (прокси)

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

Яркий пример – кэширование. Мы обращаемся не к объекту, а к его кэшу.

ПЕРЕД ТЕМ, КАК РАЗБИРАТЬ ЗАМЕСТИТЕЛЯ, ПРОЙДЕМ ПО УЖЕ ИЗУЧЕННЫМ 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 6. Урок 5. Коды к уроку/proxy.py**

|  |
| --- |
| **from** abc **import** ABCMeta, abstractmethod   **class** CurrencyRateService(metaclass=ABCMeta):  @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 непосредственно в \_\_init\_\_(), то здесь внутри \_\_init\_\_() создаем экземпляр класса нашего сервиса. **CbrCurrencyRateService**, т.е. прокси содержит ссылку на того, кого он замещает.

**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 создаем некоторый кэш курсов:

self.rates = dict()

Метод get\_currency\_rate() выглядит следующим образом: если курс в кэше, берем из кэша:

**return** self.rates[currency]

Если курс не в кэше, то берем наш реальный класс (а не прокси) и вызываем у него:

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

И потом записываем в кэш:

self.rates.update({currency: rate})

Клиент не заметит разницы по классам – исходный или прокси.

# Фасад

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

Яркий пример – кэширование. Мы обращаемся не к объекту, а к его кэшу.

Очень простой, удобный и полезный паттерн.

Он делает реализует 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. Аналогичная ситуация с классом **Engine**.

**Листинг 7. Урок 5. Коды к уроку/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** Site3Checker:  **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()  self.\_subsys\_3 = Site3Checker()   **def** russian\_auto(self):  self.\_subsys\_1.russian\_auto()  self.\_subsys\_2.russian\_auto()  self.\_subsys\_3.russian\_auto()   **def** foreign\_auto(self):  self.\_subsys\_1.foreign\_auto()  self.\_subsys\_2.foreign\_auto()  self.\_subsys\_3.russian\_auto()   facade\_site\_checker = FacadeSiteChecker() facade\_site\_checker.russian\_auto() facade\_site\_checker.foreign\_auto() |

Существует класс **Site1Checker**, представляющий собой сайт-агрегатор с объявлениями о продаже авто – отечественными и зарубежными. Второй сайт аналогичный.

Но т.к. клиенту без разницы, с какого сайта получать эти объявления (ему главное получить эти объявления).

То мы можем объединить эти классы в один **FacadeSiteChecker**.

Этот сайт более высокого уровня (фасад).

Первый сайт, например, берет объявления с Авито, второй с Авто. А третий – фасад, просто собирает все объявления.

И клиенты будут пользоваться только методами класса-фасада:

**def** russian\_auto(self):  
 self.\_subsys\_1.russian\_auto()  
 self.\_subsys\_2.russian\_auto()  
 self.\_subsys\_3.russian\_auto()  
  
**def** foreign\_auto(self):  
 self.\_subsys\_1.foreign\_auto()  
 self.\_subsys\_2.foreign\_auto()  
 self.\_subsys\_3.russian\_auto()

И клиентам без разницы, с какого сайта извлекаются объявления.

# 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. Урок 5. Коды к уроку/composite.py**

|  |
| --- |
| **from** abc **import** ABCMeta, abstractmethod   **class** Component(metaclass=ABCMeta):  @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):  print(**'folder'**)  **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) print(composite\_2.\_child)  *# использование разных по структуре операций идентично* composite\_2.operation() *# operation\_1.operation()* |

Класс **MachineOperation** выполняет какую-то задачу (это аналог файла), а класс **CompositeOperation** (аналог папки) включает как файлы, так и другие папки.

Важные моменты:

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

# Выводы

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