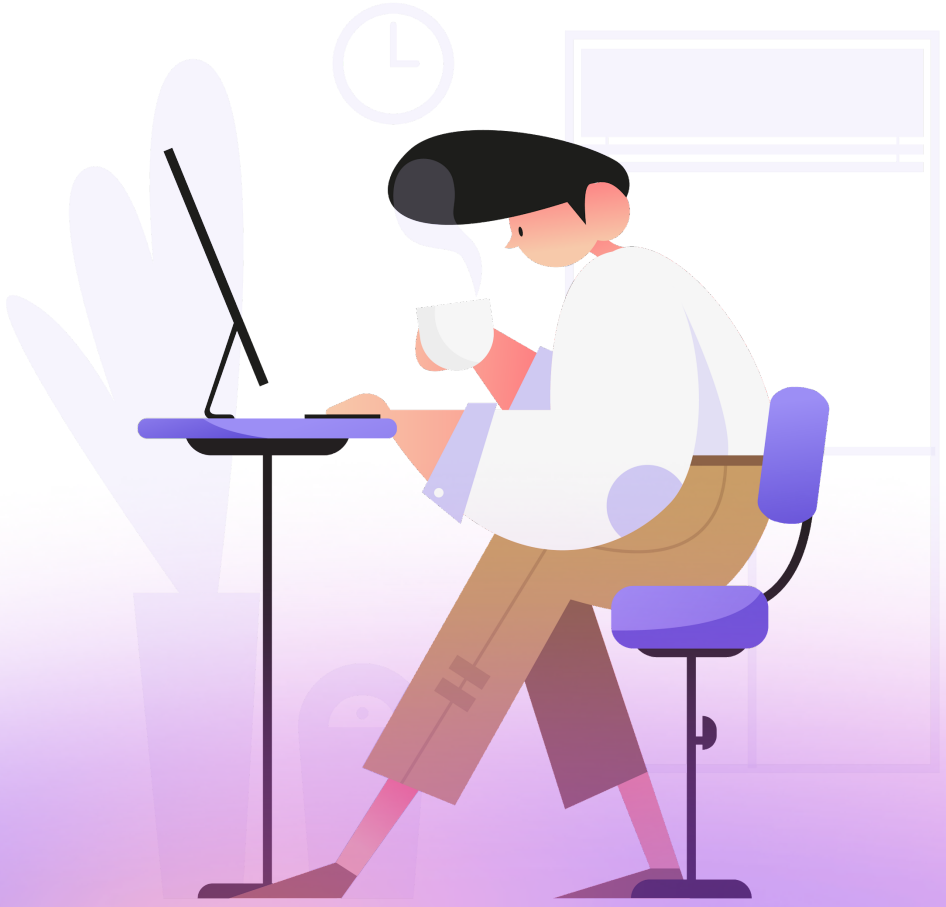
GeekBrains

Объектно-ориентированные паттерны, группа структурные: Adapter / Wrapper



Введение

На этом уроке мы начнем обсуждение блока структурных паттернов, в частности, паттерна «Адаптер».

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.

Композиция и наследование

Рассмотрим пример.

Листинг 1. Урок 12. 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 **Animal** и от него мы наследуем класс **Cat**.

Здесь ключевое слово – «является». Кошка является животным?

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

Рассмотрим еще пример. У нас есть класс **Engine** и класс **Car**. Унаследуем «машину от двигателя». Но это будет некорректно.

Потому что машина не является двигателем.

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

```
def __init__(self, engine):
    self.engine = engine
```

Большинство структурных паттернов основаны именно на композиции.

Адаптер

Переходим к разбору нашего первого структурного паттерна.

«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. Нам придется разбирать зарядное устройство и переделывать его под нужное напряжение, но мы этого не можем сделать, т.е. не можем просто так взять и переделать наше зарядное устройство. Что делать? Покупать новый телефон с зарядным устройством под 110В?

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

Адаптер – это переходник, поддерживающий другой интерфейс.

Как 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 12. 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 данном листинге есть абстрактный класс **Roundable** с методом **get_radius()**. Есть конкретный класс – **Circle** с переопределенным методом **get_radius()**. Есть класс **Square** с атрибутом **side** и методом **get_side()**. У этого класса другой интерфейс – у него отсутствует метод **get_radius()** и значит мы его не можем вызвать для объекта квадрата. Но если хотим это сделать?

Мы создаем класс-адаптер **RoundableSquare**.

И при описании класса указываем класс, чей интерфейс адаптируем (**Square**) и класс, к интерфейсу которого нужно выполнить адаптирование.

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

В данном случае мы опираемся на наследование.

Напишем и запустим простой клиентский код:

```
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))
```

Все получилось. Теперь мы можем вызывать метод **get_radius()** и для квадрата, но вызываем через адаптер.

Казалось бы, что все получилось и работает как надо, но есть один недостаток. Что если таких классов, как квадрат, у нас будет несколько десятков, например, Romb, Line и т.д. – придется для каждого из них создавать свой адаптер. Оптимально ли это? Конечно нет. Нам нужен адаптер “объект”. Мы избежим необходимости для каждой фигуры создавать специальный класс-адаптер.

Адаптер объекта

Теперь рассмотрим адаптер на объектах. Он используется чаще.

Листинг 3. Урок 12. 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)]
```

Идея схожая. Есть 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 объект квадрата.

Здесь мы уже используем композицию вместо наследования.

Но вместо объекта квадрата может быть и объект ромба или любой другой фигуры.

Благодаря адаптеру объект, сам класс-адаптер один и тот же. Нет необходимости создавать его под каждую фигуру.

Сделаем итоговый вывод. Идея адаптера в том, чтобы привести объект к другому интерфейсу.