Модуль 3. Полиморфизм и абстракция (4 ак. ч.)

Подмена методов в производном классе.

Понятие абстрактного класса.

Практикум: Применение полиморфных классов.

[Подмена методов в производном классе. 1](#_Toc146844639)

[Абстрактные методы 4](#_Toc146844640)

[Множественное наследование 5](#_Toc146844641)

[Коллекция \_\_slots\_\_ 10](#_Toc146844642)

[Как работает \_\_slots\_\_ с property и при наследовании 12](#_Toc146844643)

[Поведение \_\_slots\_\_ при наследовании классов 14](#_Toc146844644)

[Объектно-ориентированный дизайн 15](#_Toc146844645)

[Принципы SOLID-проектирования (принципы дяди Боба) 15](#_Toc146844646)

[1. Принцип единственной ответственности 17](#_Toc146844647)

[2. Второй принцип SOLID Принцип открытости/закрытости 21](#_Toc146844648)

[Интерфейсы 25](#_Toc146844649)

[3. Принцип подстановки Барбары Лисков. Liskov's Substitution Principle 28](#_Toc146844650)

[4. The Interface Segregation Principle. Принцип (сегрегации) разделения интерфейса 33](#_Toc146844651)

[5. Принцип инверсии зависимостей Dependency inversion principle. 37](#_Toc146844652)

[Выводы 38](#_Toc146844653)

# Подмена методов в производном классе.

Хочу затронуть следующий важный вопрос ООП: что такое полиморфизм и как он реализуется на Python? Вначале вспомним, что

*Полиморфизм – это возможность работы с совершенно разными объектами (языка Python) единым образом.*

Кажется, пока не особо понятно? Поэтому давайте, как всегда, постигнем суть этого подхода на конкретном примере.

Вначале я продемонстрирую пример, где мы увидим один недостаток, который как раз исправляется с помощью полиморфизма. Предположим, у нас есть два класса Rectangle и Square:

class Rectangle:

    def \_\_init\_\_(self, w, h):

        self.w = w

        self.h = h

    def get\_rect\_pr(self):

        return 2\*(self.w+self.h)

class Square:

    def \_\_init\_\_(self, a):

        self.a = a

    def get\_sq\_pr(self):

        return 4\*self.a

И в них объявлены геттеры get\_rect\_pr() и get\_sq\_pr() для получения периметра соответствующих фигур: прямоугольника и квадрата. Далее, мы можем создать экземпляры этих классов и вывести в консоль значения периметров:

r1 = Rectangle(1, 2)

r2 = Rectangle(3, 4)

print(r1.get\_rect\_pr(), r2.get\_rect\_pr())

s1 = Square(10)

s2 = Square(20)

print(s1.get\_sq\_pr(), s2.get\_sq\_pr())

Все отлично, все работает. Но, теперь предположим, что все эти объекты помещаются в коллекцию:

geom = [r1, r2, s1, s2]

которую можно легко перебрать с помощью цикла for и где бы мы хотели получить значение периметра для каждой фигуры:

for g in geom:

    print(g.get\_rect\_pr())

Как вы понимаете, когда в цикле очередь дойдет до объекта s1, возникнет ошибка, т.к. в классе Square отсутствует метод get\_rect\_pr(). Конечно, зная, что в коллекции находятся объекты Rectangle и Square, можно было бы в цикле записать проверку:

for g in geom:

    if isinstance(g, Rectangle):

        print(g.get\_rect\_pr())

    else:

        print(g.get\_sq\_pr())

и все заработает. Но у такого кода мало гибкости и, например, при добавлении еще одного класса:

class Triangle:

    def \_\_init\_\_(self, a, b, c):

        self.a = a

        self.b = b

        self.c = c

    def get\_tr\_pr(self):

        return self.a + self.b + self.c

Получим снова ошибку:

t1 = Triangle(1,2,3)

t2 = Triangle(4,5,6)

geom = [r1, r2, s1, s2, t1, t2]

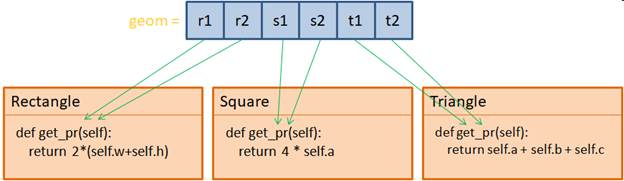
Конечно, в цикле for можно дополнительно проверить на соответствие классам Square и Triangle, но красоты и гибкости нашей программе это не придаст. Вот как раз здесь очень хорошо применим подход, который и называется полиморфизмом. Мы договоримся в каждом классе создавать методы с одинаковыми именами, например,

get\_pr()

Тогда в цикле будем просто обращаться к этому методу и получать периметры соответствующих фигур:

for g in geom:

    print( g. get\_pr() )



И это логично, так как каждая ссылка списка ведет на соответствующий объект класса и далее через нее происходит прямой вызов метода get\_pr(). Это и есть пример полиморфизма, когда к разным объектам мы обращаемся через индекс единого списка geom (единый интерфейс), а затем, вызываем геттер get\_pr() соответствующего объекта.

Мало того, мы можем сформировать этот список, сразу создавая в нем объекты соответствующих классов:

geom = [Rectangle(1, 2), Rectangle(3, 4),

        Square(10), Square(20),

        Triangle(1, 2, 3), Triangle(4, 5, 6)

        ]

Мне кажется, так программа выглядит несколько приятнее и читабельнее.

# Абстрактные методы

Но у нашей реализации есть один существенный недостаток. Что если мы забудем в каком-либо классе определить метод get\_pr(), например, в Triangle. Тогда, очевидно, программа приведет к ошибке. Как можно было бы этого избежать? Один из вариантов определить базовый класс для классов геометрических примитивов и в нем прописать реализацию геттера get\_pr(), используемую по умолчанию, например, так:

class Geom:

    def get\_pr(self):

        return -1

А все остальные классы унаследовать от него:

class Rectangle(Geom):

...

class Square(Geom):

...

class Triangle(Geom):

...

Теперь, после запуска программы, для треугольников будем получать значения -1.

Но и это не самое лучшее решение. Все же, нам бы хотелось, чтобы каждый дочерний класс имел бы обязательную реализацию метода get\_pr(). Для этого в геттере get\_pr() мы будем генерировать специальное исключение NotImplementedError, следующим образом:

class Geom:

    def get\_pr(self):

        raise NotImplementedError("В дочернем классе должен быть переопределен метод get\_pr()")

И если в каком-либо дочернем классе не будет определен метод get\_pr(), то вызовется метод базового класса и выдаст ошибку NotImplementedError, которая будет сигнализировать о том, что метод не переопределен.

Запустим программу и действительно видим это сообщение при попытке вызвать get\_pr() для объектов Triangle. Причем, видя ошибку NotImplementedError, мы понимаем, что она связана именно с необходимостью  переопределения get\_pr(), а не с чем-то другим. В этом плюс такого подхода.

В языках программирования методы, которые обязательно нужно переопределять в дочерних классах и которые не имеют своей собственной реализации называют абстрактными. Конечно, в языке Python нет чисто абстрактных методов. Здесь мы лишь выполнили имитацию их поведения, заставляя программиста определять геттер get\_pr() в дочерних классах, самостоятельно генерируя исключение NotImplementedError.

Итак, из этого занятия я, думаю, вы поняли, что из себя представляет полиморфизм и как он реализуется на Python. Также узнали, как можно определять методы, которые ведут себя подобно абстрактным с необходимостью их переопределения в дочерних классах.

## Множественное наследование

Мы продолжаем изучать тему наследования. В языке Python допускается множественное наследование, когда один дочерний класс образуется сразу от нескольких базовых, согласно синтаксису:

class A(base1, base2, …, baseN):  
…

Давайте посмотрим, как это работает и для чего вообще нужно. Признаюсь честно, я не сразу смог придумать учебный пример, где можно было бы показать целесообразность применения множественного наследования. То есть, он применяется не так часто, как обычное наследование от одного класса. Но, тем не менее это тоже важный механизм и некоторые подходы к программированию его активно используют. Например, идея миксинов (mixins) в Python реализуется через множественное наследование. И учебный пример я решил связать именно с ними, а заодно показать еще один прием в программировании.

Давайте представим, что мы делаем интернет-магазин по продаже товаров, например, ноутбуков, дисков, процессоров и т.п. Каждый товар будет определяться своим классом, а общим (базовым) для них всех будет класс Goods – товары. Также у каждого товара обязательно будут поля:

* name – наименование
* weight – вес
* price – цена

Для примера запишем в нашей программе базовый класс Goods:

class Goods:

    def \_\_init\_\_(self, name, weight, price):

        print("init MixinLog")

        self.name = name

        self.weight = weight

        self.price = price

    def print\_info(self):

        print(f"{self.name}, {self.weight}, {self.price}")

И один дочерний класс для ноутбуков:

class NoteBook(Goods):

    pass

Затем, мы можем создать объект для ноутбука:

n = NoteBook("Acer", 1.5, 30000)

и распечатать информацию о нем:

n.print\_info()

Пока ничего нового здесь нет. Но потом, к нам подходит тимлид и говорит:

- Дорогой сеньор, добавь, пожалуйста, возможность логирования товаров магазина.

И как бы вы поступили на месте этого сеньора? Плохой сеньор начнет прописывать логику логирования либо непосредственно в базовом классе Goods, либо уровнем выше (в иерархии наследования). А хороший воспользуется идеей миксинов. Для этого он создаст еще один класс, который можно назвать:

class MixinLog:

    ID = 0

    def \_\_init\_\_(self):

        print("init MixinLog")

        self.ID += 1

        self.id = self.ID

    def save\_sell\_log(self):

        print(f"{self.id}: товар продан в 00:00 часов")

Этот класс работает совершенно независимо от классов Goods и Notebook и лишь добавляет функционал по логированию товаров с использованием их id. Такие независимые базовые классы и получили название миксинов – примесей.

Добавим этот класс в цепочку наследования:

class NoteBook(Goods, MixinLog):

    pass

а ниже вызовем метод save\_sell\_log():

n.save\_sell\_log()

И видим ошибку. Очевидно, она связана с тем, что у второго класса MixinLog не был вызван инициализатор. Почему так произошло? Как мы уже знаем, при создании объектов инициализатор ищется сначала в дочернем классе, но так как его там нет, то в первом базовом Goods. Он там есть, выполняется и на этом инициализация нашего объекта NoteBook завершается. Однако, нам нужно также взывать инициализатор и второго базового класса MixinLog. В данном случае, сделать это можно с помощью объекта-посредника super(), которая и делегирует вызов метода \_\_init\_\_ класса MixinLog:

class Goods:

    def \_\_init\_\_(self, name, weight, price):

        super().\_\_init\_\_()

        print("init Goods")

        self.name = name

        self.weight = weight

        self.price = price

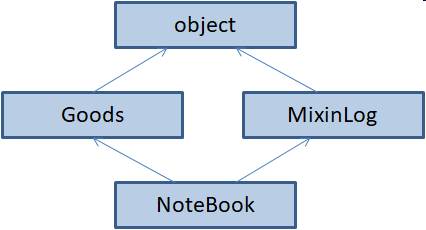
…

Теперь, после запуска программы, мы видим, что оба инициализатора сработали и ошибок никаких нет.

Но откуда функция super() «знает», что нужно обратиться ко второму базовому классу MixinLog, а, скажем, не к базовому классу object, от которого неявно наследуются все классы верхнего уровня? В Python существует специальный алгоритм обхода базовых классов при множественном наследовании. Сокращенно, он называется:

MRO – Method Resolution Order

И говорит, в каком порядке обходить базовые классы:



Мы можем увидеть эту цепочку обхода базовых классов, если распечатать специальную коллекцию \_\_mro\_\_ любого класса:

print(NoteBook.\_\_mro\_\_)

В консоли появится следующая последовательность:

(<class '\_\_main\_\_.NoteBook'>, <class '\_\_main\_\_.Goods'>, <class '\_\_main\_\_.MixinLog'>, <class 'object'>)

То есть, методы сначала ищутся в самом классе NoteBook, затем, в классе Goods, далее идет класс MixinLog и от него уже идет к классу object. Это цепочка обхода для нашего конкретного примера. При другой иерархии наследования эта цепочка может быть другой, но одно всегда неизменно –первый базовый класс, указанный при наследовании, выбирается первым (после дочернего, разумеется). И это важный момент. Вы всегда можете быть уверены, что инициализатор первого базового класса сработает в первую очередь. Почему это важно? Смотрите, при создании объекта NoteBook мы передаем ему три аргумента. Эти три аргумента, затем, передаются в инициализатор. И так как первым будет вызван инициализатор класса Goods, то мы уверены, что эти аргументы будут переданы именно в него, а не в какой-то другой инициализатор других базовых классов. И какая бы цепочка наследования у нас ни была, все равно первым будет вызываться метод \_\_init\_\_ класса Goods, потому что он записан первым. Это гарантирует работоспособность нашей программы при разных иерархиях множественного наследования.

Ради интереса, давайте поменяем местами базовые классы:

class NoteBook(MixinLog, Goods):

    pass

И мы сразу получаем ошибку, что в метод \_\_init\_\_ передаются четыре аргумента, а он принимает только один, так как здесь отрабатывает инициализатор уже класса MixinLog. Так что порядок следования базовых классов при множественном наследовании имеет важное значение. Первым должен идти «основной» класс и у него, как правило, инициализатор имеет несколько параметров. А далее, записываются классы, у которых, опять же, как правило, инициализаторы имеют только параметр self. Это второй важный момент. Когда мы собираемся использовать множественное наследование, то структуру классов следует продумывать так, чтобы инициализаторы вспомогательных классов имели только один параметр self, иначе будут сложности их использования. В чем они состоят?

Давайте для примера пропишем в инициализаторе класса MixinLog один параметр p1

class MixinLog:

    def \_\_init\_\_(self, p1):

        super().\_\_init\_\_(1, 2)

        …

И объявим еще один класс миксинов, где в инициализаторе два параметра:

class MixinLog2:

    def \_\_init\_\_(self, p1, p2):

        super().\_\_init\_\_()

        print("init MixinLog 2")

В каждом методе \_\_init\_\_ мы также делаем делегированный вызов инициализатора следующего базового класса. А цепочка наследования будет такой:

class NoteBook(Goods, MixinLog, MixinLog2):

    pass

Сейчас при запуске у нас не возникает никаких ошибок, так как последовательность MRO имеет вид:

(<class '\_\_main\_\_.NoteBook'>, <class '\_\_main\_\_.Goods'>, <class '\_\_main\_\_.MixinLog'>, <class '\_\_main\_\_.MixinLog2'>, <class 'object'>)

То есть, мы знаем, что функция super() в классе Goods вызовет метод \_\_init\_\_ класса MixinLog с одним дополнительным параметром, а затем, метод \_\_init\_\_ класса MixinLog2 с двумя дополнительными параметрами. И мы «жестко» это прописали. Но, как вы понимаете, если хотя бы немного изменится цепочка наследования, например, так:

class NoteBook(Goods, MixinLog2, MixinLog):

    pass

то все нарушится и получим ошибки. Чтобы в программах при множественном наследовании не возникало проблем с зависимостью последовательности наследования дополнительных базовых классов, их инициализаторы следует создавать с одним параметром self и в каждом из них прописывать делегированный вызов инициализатора следующего класса командой:

super().\_\_init\_\_()

Тогда точно никаких особых проблем при использовании множественного наследования не возникнет.

Последнее, что я хочу отметить на этом занятии, это вызов методов с одинаковыми именами из базовых классов. Давайте предположим, что в классе MixinLog имеется метод print\_info с тем же именем, что и в классе Goods:

    def print\_info(self):

        print("print\_info класса MixinLog")

Понятно, что если сейчас его вызвать через объект класса NoteBook:

n.print\_info()

то мы обратимся к методу класса Goods, так как он записан первым в цепочке наследования и в соответствии с алгоритмом обхода MRO он будет найден первым. Но что если мы хотим вызвать этот метод из второго базового класса MixinLog? Как поступить? Сделать это можно двумя способами. Либо напрямую вызвать этот метод через класс MixinLog:

MixinLog.print\_info(n)

Обратите внимание, что в этом случае нам обязательно нужно указать первым аргументом ссылку на объект класса NoteBook. Либо, определить какой-либо метод в классе NoteBook (пусть он называется также):

class NoteBook(Goods, MixinLog):

    def print\_info(self):

        MixinLog.print\_info(self)

И тогда будет вызываться метод именно второго базового класса MixinLog.

Обычно, если нужно делать такие подмены, то есть, из конкретного дочернего класса вызывать метод другого (не первого) базового класса, то создают метод в дочернем классе с тем же именем, а затем, явно указывают нужный базовый класс.

Вот так работает множественное наследование в языке Python.

## Коллекция \_\_slots\_\_

На этом занятии мы с вами познакомимся со специальной коллекцией \_\_slots\_\_, которую можно прописывать в любом классе языка Python. Для простоты восприятия информации возьмем простой класс для представления точки на плоскости:

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

Пока в этом классе нет коллекции \_\_slots\_\_, поэтому при создании его экземпляра можно свободно оперировать существующими свойствами:

pt = Point(1, 2)

pt.x

pt.y = 100

И создавать новые:

pt.z = 4

Как известно, список всех локальных свойств экземпляра класса хранится в его магическом списке \_\_dict\_\_:

pt.\_\_dict\_\_

Но что, если мы хотим объявить класс точки на плоскости, чтобы у его экземпляров были только два свойства x и y и никакие другие. Как это сделать? Для этого, как раз и применяется коллекция \_\_slots\_\_.

Давайте я создам для сравнения еще один класс, в котором укажу эту коллекцию:

class Point2D:

    \_\_slots\_\_ = ('x', 'y')

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

Смотрите, мы в кортеже перечисляем атрибуты с именами x и y (указываются в виде строки) и только такие локальные свойства могут существовать в экземплярах этого класса. Проверим это. Создадим экземпляр:

pt2 = Point2D(10, 20)

Выведем свойства x и y, убедимся, что они присутствуют в экземпляре:

pt2.x

pt2.y

Но вот добавить новое уже не получится:

pt2.z = 30

Это произошло как раз по тому, что в классе прописана коллекция \_\_slots\_\_ и мы ее можем даже вывести:

pt2.\_\_slots\_\_

А вот привычная нам коллекция

pt2.\_\_dict\_\_

будет отсутствовать. То есть, класс и его экземпляры ведут себя немного по-другому. Мы совершенно спокойно можем изменять, удалять и добавлять локальные свойства x, y:

pt2.x = 50

del pt2.y

pt2.y = 100

Но только их и никакие другие. Причем, обратите внимание, речь идет только о локальных свойствах экземпляров. В сам класс мы по-прежнему можем добавлять любые атрибуты, например, MAX\_COORD:

class Point2D:

    \_\_slots\_\_ = ('x', 'y')

    MAX\_COORD = 100

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

Тогда в экземплярах этого класса появится это дополнительное свойство со значением 100.

Коллекция \_\_slots\_\_ помимо ограничения создаваемых локальных свойств, еще уменьшает объем памяти, занимаемый экземпляром класса. Смотрите, если создать  два экземпляра:

pt = Point(1, 2)

pt2 = Point2D(10, 20)

То первый будет содержать ссылку, как на объект класса, так и на коллекцию \_\_dict\_\_. А второй – только на пространство имен класса Point2D:

print(pt.\_\_sizeof\_\_(), pt.\_\_dict\_\_.\_\_sizeof\_\_())

print(pt2.\_\_sizeof\_\_())

И еще коллекция \_\_slots\_\_ ускоряет работу с локальными свойствами экземпляров класса. Например, если добавить в каждый класс метод:

    def calc(self):

        self.x += 1

        del self.y

        self.y = 0

И замерить скорость его работы:

t1 = timeit.timeit(pt.calc)

t2 = timeit.timeit(pt2.calc)

print(t1, t2)

То мы увидим, что t1 больше, чем t2, то есть, класс Point2D работает быстрее с локальными свойствами, чем класс Point.

Вот такие три особенности дает коллекция \_\_slots\_\_ экземплярам класса:

* ограничение создаваемых локальных свойств;
* уменьшение занимаемой памяти;
* ускорение работы с локальными свойствами.

На следующем занятии мы продолжим эту тему и посмотрим, как эта коллекция ведет себя при наследовании классов.

## Как работает \_\_slots\_\_ с property и при наследовании

Мы продолжаем изучение коллекции \_\_slots\_\_. Как мы говорили на предыдущем занятии, эта коллекция позволяет ограничивать список локальных свойств в экземплярах класса, в котором прописана. Например, для класса точки на плоскости мы разрешали только свойства с именами x и y:

class Point2D:

    \_\_slots\_\_ = ('x', 'y')

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

А теперь, сделаем следующее. Добавим в эту коллекцию еще одно имя:

\_\_slots\_\_ = ('x', 'y', 'length')

Тогда, очевидно, мы его тоже сможем использовать в нашем классе, например, в конструкторе:

class Point2D:

    \_\_slots\_\_ = ('x', 'y', 'length')

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

        self.length = (x\*x + y\*y) \*\* 0.5

И, далее, создавая экземпляр этого класса:

pt = Point2D(1, 2)

автоматически получаем свойство length с расстоянием до точки (относительно нуля):

pt.length

Но что если в этом классе прописать свойство (property) с тем же самым именем length? А в коллекции \_\_slots\_\_ задать приватное свойство:

\_\_slots\_\_ = ('x', 'y', '\_\_length')

Давайте добавим это свойство и посмотрим, что получится:

    @property

    def length(self):

        return self.\_\_length

    @length.setter

    def length(self, value):

        self.\_\_length = value

И, затем, после создания экземпляра класса, мы можем использовать такой синтаксис:

pt = Point2D(1, 2)

print(pt.length)

Смотрите, мы здесь обращаемся к свойству length, хотя в \_\_slots\_\_ нет такого разрешенного имени. И при этом нет никаких ошибок выполнения. В принципе, так и должно быть, потому что length – это не локальная переменная экземпляра класса, а атрибут класса Point2D. Коллекция \_\_slots\_\_ не накладывает ограничения на атрибуты класса, только на локальные атрибуты его экземпляров. Благодаря этому и появляется свойство length, которое работает как геттер и сеттер класса Point2D.

## Поведение \_\_slots\_\_ при наследовании классов

Следующий важный момент – это особенности работы коллекции \_\_slots\_\_ при наследовании классов. Я сразу приведу пример, чтобы показать первую особенность этой коллекции. Допустим, мы создаем еще один класс Point3D, который наследует от класса Point2D и он пока будет пустым:

class Point3D(Point2D):

    pass

Тогда, для экземпляров этого класса будет разрешено создание локальных свойств, несмотря на то, что в базовом классе присутствует коллекция \_\_slots\_\_:

pt3 = Point3D(10, 20)

pt3.z = 30

То есть, по умолчанию эта коллекция не наследуется дочерними классами и они ведут себя как обычные, без каких-либо ограничений, сохраняя все локальные свойства, используя коллекцию:

pt3.\_\_dict\_\_

Однако, если мы ее пропишем даже без указания каких-либо элементов:

class Point3D(Point2D):

    \_\_slots\_\_ = ()

то ограничения вступят в свои права и она будет как бы унаследована от базового класса Point2D. То есть, в классе Point3D сейчас доступны только два локальных свойства x и y.

Но нам нужно разрешить еще одно имя z. Для этого достаточно одно его и прописать в этой коллекции:

class Point3D(Point2D):

    \_\_slots\_\_ = 'z',

Тогда имена x, y будут наследоваться от класса Point2D, а z – добавляться классом Point3D. Также обратите внимание на висячую запятую. Она означает, что z – это элемент кортежа, а не просто строка. Предпочтительно записывать в таком виде. Хотя, и без запятой тоже все будет работать.

Очевидно, следующим шагом надо добавить конструктор в дочерний класс. Запишем его в следующем виде:

class Point3D(Point2D):

    \_\_slots\_\_ = 'z',

    def \_\_init\_\_(self, x, y, z):

        super().\_\_init\_\_(x, y)

        self.z = z

И экземпляр будем создавать уже с тремя параметрами:

pt3 = Point3D(10, 20, 30)

Теперь, наш дочерний класс содержит уникальный код, расширяющий функционал базового класса без каких-либо повторений. При этом, в обоих классах используется ограничение, но с разным набором локальных свойств.

Надеюсь, вы стали лучше понимать работу этой коллекции и при необходимости применять ее в своих проектах.

# Объектно-ориентированный дизайн

Принципы ООП = Алфавит-слова-предложения

ОО Дизайн = Как написать книгу, разделы, абзацы, смысловая составляющая

В нашей жизни бизнес-процессы постоянно изменяются и эволюционируют.

Вышла новая версия браузера, половина расширений перестала работать. Так было не раз с firefox, opera и т.п. люди уходят на другие приложения.

Приложение не только должно хорошо работать, оно должно быть легко доступно для изменений и поддержки.

2 подхода:

Поддержка без изменения существующего кода вообще

Либо изменения должны быть минимальными.

Сообщество программистов ООП напридумывали кучу принципов, которых нужно придерживаться. Принципы хорошего поведения.

Но, иногда правила созданы, для того, чтобы их нарушать. Кто не переходил дорогу на красный цвет? Разные ситуации бывают.

## Принципы SOLID-проектирования (принципы дяди Боба)

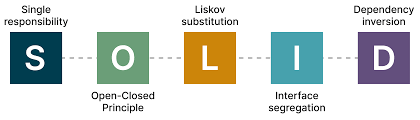
Самыми известными и важными считаются принципы проектирования, предложенные Робертом К. Мартином (также известным как Дядя Боб).

Дядя Боб представил много разных принципов проектирования, однако самых популярных всего 5, сокращенно их называют SOLID-принципами. В основном они сфокусированы вокруг объектно-ориентированной парадигмы проектирования ПО.

Если учитывать эти рекомендации при разработке объектно-ориентированного ПО, код станет не таким сложным, снизится риск поломок, улучшится взаимодействие между различными объектами и код станет более гибким, читаемым и управляемым.







SOLID-принципы Дяди Боба расшифровываются следующим образом:

• S – Принцип единственной ответственности (Single Responsibility Principle),

• O – Принцип открытости/закрытости (Open‐Closed Principle),

• L – Принцип подстановки Барбары Лисков (Liskov Substitution Principle),

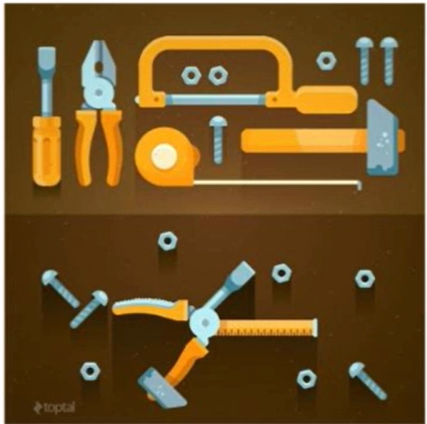
• I – Принцип разделения интерфейсов (Interface Segregation Principle),

• D – Принцип инверсии зависимостей (Dependency Inversion Principle).

Примеры кода, которые я привожу, будут весьма минималистичны по своей природе, поскольку они написаны с единственной целью – объяснить соответствующий принцип. Они могут быть неполными или не соответствовать какому-либо другому принципу или передовой практике. Прошу моих учеников принять это во внимание при изучении кода каждого принципа.

## 1. Принцип единственной ответственности

Рассказывая про данный принцип, обычно показывают картинку со швейцарским ножом, который можно делать всё и сразу. Миллион разных штучек в нем.

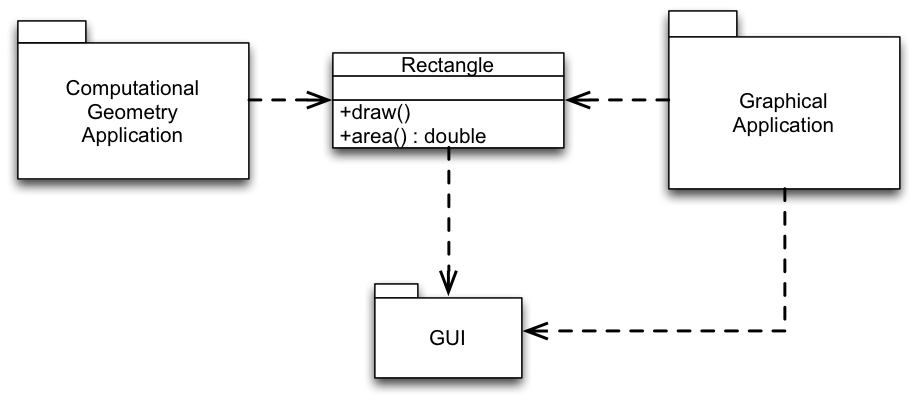


Принцип единой ответственности гласит, что у каждого класса должна быть только одна «ответственность» и он не должен брать на себя другие обязанности. Роберт К. Мартин объяснял его так: «У класса должна быть лишь одна причина для изменения».

Нам не нужен швейцарский нож, нам не нужен такой инструмент.

При изменении кода, у нас появляется код, который отвечает за что-то иное.

Хрестоматийно, обычно рассматривают пример дяди Боба

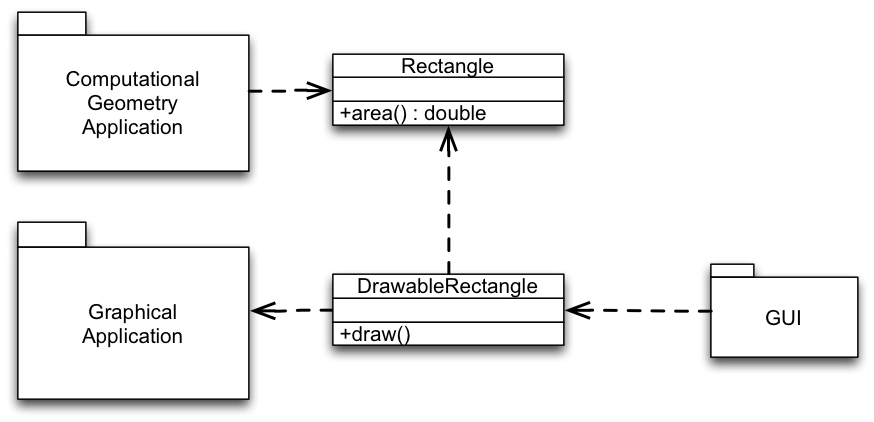


У нас есть класс Rectangle, которые вычисляет площадь прямоугольника и отрисовывает его в каком-то GuI.

Этот класс используют для различных приложения. Условно говоря – вычислитель и второй графическое изображение.

И класс через API GUI рисует наш прямоугольник. Этот пример демонстрирует явное нарушение принципа единственной ответственности.

Почему? Класс выполняет две различные операции, а потом рисует gui. Один считает, другой рисует. Счетовод и художник не должны сидеть в одной комнате.



Правильные вариант – класс счетовод Rectangle обсчитывает периметр, площадь и т.п.

И при необходимости рисовать мы пишем класс рисователь, которые наследует (берет) от счетовода необходимые данные для рисования.

Каждый метод должен заниматься своим делом. Хорошая функция, это такая функция, в которую уже нечего напихать, а та из которой нечего выбросить.

В функции должен остаться минимум, ради чего она затевалась.

Т.о. методы в классы объединяются по однотипности выполняемых действий.

Класс list, string, tuple.

Послать на принтер, отрисовать что-то, отправить данные на сервер – это все разные классы.

Однако не стоит слепо подчиняться этому правилу. Если слепо следовать – вообще ничего не напишите, или появится куча избыточности.

Для разминки … ООП не всё явно и однозначно. Задачка. Устное программирование. У вас есть три объекта. Бумага, ручка, текст.

Необходимо рисовать метод Рисователь Write. У какого объекта будем его писать.

Варианты:

1. Текст – написать на бумаге этой ручкой
2. Бумага. Написать ручкой на этой бумаге
3. Ручка. Написать на бумаге этот текст

Как правильно? Никак! Точнее и так и так и так.

Абсолютной однозначности нет.

Если я бы выбирал – я бы засадил бы этот метод в ручку.

В качестве второго примера рассмотрим приложение для телефонного справочника. Мы будем делать телефонный справочник, в котором будет класс TelephoneDirectory.

Он будет «нести ответственность» за ведение записей справочника, то есть телефонных номеров и названий организаций, которым принадлежат номера. Ожидается, что класс будет выполнять следующие операции: добавлять новую запись (Name и TelephoneNumber), удалять существующую запись, изменять номер телефона, присвоенный сущности Name, и предоставлять поиск, который будет возвращать номер, присвоенный сущности Name.

Класс TelephoneDirectory может выглядеть следующим образом:

class TelephoneDirectory:  
 def \_\_init\_\_(self):  
 self.telephonedirectory = {}  
  
 def add\_entry(self, name, number):  
 self.telephonedirectory[name] = number  
  
 def delete\_entry(self, name):  
 self.telephonedirectory.pop(name)  
  
 def update\_entry(self, name, number):  
 self.telephonedirectory[name] = number  
  
 def lookup\_number(self, name):  
 return self.telephonedirectory[name]  
  
 def \_\_str\_\_(self):  
 ret\_dct = ""  
 for key, value in self.telephonedirectory.items():  
 ret\_dct += f'{key} : {value}\n'  
 return ret\_dct  
  
  
myTelephoneDirectory = TelephoneDirectory()  
myTelephoneDirectory.add\_entry("Bob", 123456)  
myTelephoneDirectory.add\_entry("Tom", 678452)  
print(myTelephoneDirectory)  
  
myTelephoneDirectory.delete\_entry("Bob")  
myTelephoneDirectory.add\_entry("Bob", 123456)  
myTelephoneDirectory.update\_entry("Tom", 776589)  
print(myTelephoneDirectory.lookup\_number("Tom"))  
print(myTelephoneDirectory)

Результат:

Bob : 123456

Tom : 678452

776589

Tom : 776589

Bob : 123456

Сейчас наш класс TelephoneDirectoryвыглядит хорошо, в нем точно реализованы ожидаемые функции:

А теперь скажем, что в проекте есть еще два требования – Сохранить содержимое справочника в базе данных и перенести содержимое справочника в файл.

Теперь добавим еще два метода в класс TelephoneDirectory*,*как показано ниже:

def save\_to\_file(self, file\_name, location):  
 *# code to save the contents of telephonedirectory dictionary to the file* pass  
  
def persist\_to\_database(self, database\_details):  
 *# code to persist the contents of telephonedirectory dictionary to database* pass

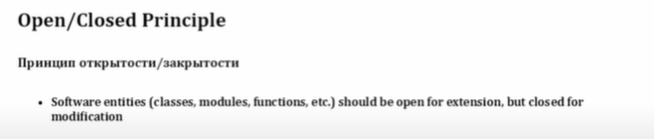
Так вот, именно сейчас мы нарушили принцип единственной ответственности. Добавив функции сохранения в базу данных и сохранения в файл, мы дали классу дополнительные обязанности, которые не входят в его основную зону ответственности. Теперь в классе есть дополнительные функции, которые могут привести к его изменению. В будущем, если появятся какие-то требования, связанные с сохранением данных, это может привести к изменениям в классе TelephoneDirectory. Получается, что класс TelephoneDirectory подвержен изменениям по причинам, которые не являются его основной ответственностью.

Принцип единственной ответственности требует от нас не добавлять дополнительные обязанности к классу, чтобы нам не приходилось менять класс, когда нам нужно изменить функционал сохранения справочника в базу данных или в файл. Мы можем передать экземпляр класса TelephoneDirectory экземплярам этих классов и записать любые дополнительные функции в них.

Так мы гарантируем, что у класса TelephoneDirectoryесть лишь одна причина для изменения – это изменения в его основной «ответственности».

class persist\_to\_database:  
 *# functionality of the class* def \_\_init\_\_(self, object\_to\_persist):  
 pass  
  
class save\_to\_file:  
 *# functionality of the class* def \_\_init\_\_(self, object\_to\_save):  
 pass

## 2. Второй принцип SOLID Принцип открытости/закрытости



Принцип открытости/закрытости впервые был сформулирован Бернардом Мейером в 1988 году. Роберт К. Мартин говорил о нем так «Наиболее важный принцип открытости/закрытости гласит «Сущности программы (классы, модули, функции и т.п.) должны быть открыты для расширения, но закрыты для изменений».

Т.е. мы должны предусмотреть. Потому что однажды разработанная реализация класса в дальнейшем требует только исправления ошибок. А если нам нужно что-то новое или что-то изменить – это требует создания нового класса.

К примеру, два класса:

[Клиент] ---[Сервер]

К примеру веб-сервер Apache и наш клиент привязывается со всеми его ньюансами. Какая проблему нас ожидает. Завтра мы ставим другой сервер Nginx или какой-нить не дай бог виндовс-сервер, нам напрочь придется менять архитектуру всей клиентской части.

Т.к. у нас счас в клиенте реализованы тонкости конкретного сервера Apache.

В таком случае нам поможет абстракция.

[Клиент] ---[Абстрактный сервер] < ---[Сервер]

Один из китов ООП. Четвертый принцип ООП. Абстракция!

Не конкретный сервер. Не апач, не nginx, а некий общий абстрактный сервер.

Напоминаю мы программируем не на языка, а с помощью азыка.

Сначала придумываем алгоритм работы программы или другими словами архитектуру, или дизайн приложения. А потом уже пишем реализацию.

Те кто жестко привязывается к языку, получает больше проблем. Так произошло при переходе со 2-ей версии Пайтона на 3-ю.

Абстрактные классы начисто закрыты для изменений. А конкретный сервер открыт для расширений.

Для этого в программировании используется соответствующий абстрактный класс. Который является основой для расширения.

Если мы говорили о классе, как о чертеже объекта, то в данном случае абстрактный класс это эскиз (набросок) чертежа, калька, через которую просвечивает будущий сервер , обводилка.

Или просто чистый лист, на котором написано здесь будет сервер, здесь будут его порты

class AbsractServer:  
 def \_\_init\_\_(self, version):  
 self.version = version  
  
 def get\_version(self):  
 return self.version  
  
class ServerApache(AbsractServer):  
 pass  
  
class ServerIIS(AbsractServer):  
 pass  
  
class ServerNginx(AbsractServer):  
 pass

Помимо абстрактных классов, абстрактные классы могут иметь абстрактные методы.

Вопрос? Что делает абстрактный метод? А что делает абстрактный класс.

Ничего. Абстрактный метод – это метод без реализации.

def connect(self):  
 pass

Этот метод должен быть реализован в конкретных классах.

Тот кто будет писать клиенты, тот и будет писать метод connect.

Важно! Программисты обязательно должны описать этот метод. Это особенность абстрактных методов.

В других языках нельзя создать класс от конкретного абстрактого класса, не описав его методов в классах наследниках.

В пайтоне меньше авторитарности. Больше свободы, либерализма, соглашений, конвенций и т.п.

Но здесь серьёзная ситуация, как принудить более жестко борцов за свободу соблюдать общепринятые правила.

Для этого есть модуль abc.

Догадайтесь, как расшифровывается данная аббревиатура – abc.

Пример

import abc

class АбстрактныйКласс(abc.ABC):  
  
 @abc.abstractmethod  
 def абстрактный\_метод(self):  
 pass  
  
  
class КонкретныйКласс(АбстрактныйКласс):  
 pass  
  
объект = КонкретныйКласс() *# Ошибка!*

Отнаследуем АбстрактныйКласс от модуля abc. После этого у нас появится очень хороший декоратор, который предписывает описать абстрактные метод в наследниках. И не следование этому приведет к ошибке:

TypeError: Can't instantiate abstract class КонкретныйКласс with abstract method

Нужно переописать абстрактный класс

class КонкретныйКласс(АбстрактныйКласс):  
 def абстрактный\_метод(self):  
 pass

Применительно к нашим веб-серверам, перепишем наш базовый класс

import abc  
  
class AbsractServer(abc.ABC):  
 def \_\_init\_\_(self, version):  
 self.version = version  
  
 def get\_version(self):  
 return self.version  
 @abc.abstractmethod  
 def connect(self):  
 pass

Теперь мы не сможем создавать экземпляры классов ServerApache, ServerNginx

без описание методов connect

class ServerApache(AbsractServer):  
 def connect(self):  
 print('Connection from Apache ...')

s2 = ServerApache(2)  
s2.connect()

Тот, кто будет писать клиента, будет ориентироваться не на этих ребят ServerApache, ServerNginx, а на абстрактный class следующим образом

class Client:  
 def connect\_to(self,server):  
 if isinstance(server,AbsractServer):  
 server.connect()

кто будет на месте AbsractServer разработчику клиентской части знать не нужно, это его дело на десятом месте.

Т.е. если нам потребуется добавить какой-то другой сервер, с которым будет соединяться наш клиент, или внести изменения в какой-то сервер, то нам клиента менять не надо будет вообще.

Старайтесь абстрагироваться! Начиная прямо с этого момента. Мы постоянно этим занимаемся.

Не обращаем внимание на детали, мы потом до них дойдем. До обработки ошибок, до обработки исключений. Вначале абстракции, потом реализация, потом детали.

Когда мы учимся водить автомобиль, вас не должно волновать какого он цвета, из какого металла сделан руль, что внутри коробки передач. Какая разница, как работает стеклоподьемник и т.п., Задача научиться управлять с ошибками. Пусть она глохнет через каждые 100 м, но она едет.

Важна абстракция, абстракция – путь к успеху.

## Интерфейсы

С абстракцией тесно связаны интерфейсы.

Суть интерфейса и абстрактного класса – одна и та же.

Есть ЯП в которых реализовано и понятие абстрактного класса и интерфейса. Есть ЯП в которых реализован только интерфейс.

Как правило под интерфейсом понимается АК, в котором только абстрактные методы.

Который просто задает поведение объекта. АК подразумевает не только наличие абстрактных методов, но и нормальных методов, реализованных. Это уже не интерфейс.

А интерфейс – это перечисление только того, что д.б.

Вот классический интерфейс сервера. Хрестоматийный пример. Соедини, разъедини, прими, получи

class IServer(abc.ABC):  
 @abc.abstractmethod  
 def connect(self):  
 pass  
  
 @abc.abstractmethod  
 def disconnect(self):  
 pass  
  
 @abc.abstractmethod  
 def request(self):  
 pass  
  
 @abc.abstractmethod  
 def response(self):  
 pass

нет никакой конкретики, есть метод connect – значит соединяй. А как меня не волнует.

На примере автомобиля – ты должен уметь завести, включить сигналы поворота, наличие руля, нажимать клаксон. И все это д.б описано в базовом абстрактном классе.

А реализовано в классах наследниках. Электромобиле, автобусе, троллейбусе, мусорной машине. Кстати бывают треугольные рули. Но руль есть всегда.

Абстракция позволяет реально расширять. Т.е. сначала пишем абстрактный класс СуперКар, а потом уже все остальные производные легковый (бмв, мерс), грузовой (камаз). Животные – млекопитающие – парнокопытные.

Но очень легко наделать ошибок. На быть сам-себе режиссером.

Прежде чем мы перейдем к принципу Барбары, посмотрим на еще один пример

Допустим, у нас есть приложение для магазина одежды. Среди функций системы есть функция применения специальных скидок в зависимости от типа одежды.

Пример ниже показывает один из способов реализации этого требования.

В примере у нас есть класс DiscountCalculator, который умеет хранить тип одежды. В нем есть функция, которая рассчитывает скидку в зависимости от типа одежды и возвращает новую стоимость за вычетом суммы скидки.

from enum import Enum  
  
  
class Products(Enum):  
 SHIRT = 1  
 TSHIRT = 2  
 PANT = 3  
  
  
class DiscountCalculator():  
 def \_\_init\_\_(self, product\_type, cost):  
 self.product\_type = product\_type  
 self.cost = cost  
  
 def get\_discounted\_price(self):  
 if self.product\_type == Products.SHIRT:  
 return self.cost - (self.cost \* 0.10)  
 elif self.product\_type == Products.TSHIRT:  
 return self.cost - (self.cost \* 0.15)  
 elif self.product\_type == Products.PANT:  
 return self.cost - (self.cost \* 0.25)  
  
  
dc\_Shirt = DiscountCalculator(Products.SHIRT, 100)  
print(dc\_Shirt.get\_discounted\_price())  
  
dc\_TShirt = DiscountCalculator(Products.TSHIRT, 100)  
print(dc\_TShirt.get\_discounted\_price())  
  
dc\_Pant = DiscountCalculator(Products.PANT, 100)  
print(dc\_Pant.get\_discounted\_price())

Вспомним, о каком принципе мы сейчас говорим!?

# Принцип открытия-закрытия. Программные объекты (классы, модули, функции и т. д.) должны быть открыты для расширения, но закрыты для модификации.

'''

Эта конструкция нарушает принцип Open Closed, поскольку этот класс потребует модификации, если

а). Должен быть включен новый тип одежды и

б). Если сумма скидки на любую одежду изменится

class DiscountCalculator(abc.ABC):  
  
 @abc.abstractmethod  
 def get\_discounted\_price(self):  
 pass  
  
  
class DiscountCalculatorShirt(DiscountCalculator):  
 def \_\_init\_\_(self, cost):  
 self.cost = cost  
  
 def get\_discounted\_price(self):  
 return self.cost - (self.cost \* 0.10)  
  
  
class DiscountCalculatorTshirt(DiscountCalculator):  
 def \_\_init\_\_(self, cost):  
 self.cost = cost  
  
 def get\_discounted\_price(self):  
 return self.cost - (self.cost \* 0.15)  
  
  
class DiscountCalculatorPant(DiscountCalculator):  
 def \_\_init\_\_(self, cost):  
 self.cost = cost  
  
 def get\_discounted\_price(self):  
 return self.cost - (self.cost \* 0.25)  
  
  
dc\_Shirt = DiscountCalculatorShirt(100)  
print(dc\_Shirt.get\_discounted\_price())  
  
dc\_TShirt = DiscountCalculatorTshirt(100)  
print(dc\_TShirt.get\_discounted\_price())  
  
dc\_Pant = DiscountCalculatorPant(100)  
print(dc\_Pant.get\_discounted\_price())

Как видно из примера выше, теперь у нас есть очень простой базовый класс DiscountCalculator с одним абстрактным методом get\_discounted\_price.

Мы создали новые классы для одежды, которые расширяют базовый класс DiscountCalculator.

Следовательно, теперь каждый подкласс будет реализовывать функционал скидок самостоятельно.

Сделав так, мы устранили предыдущие ограничения, которые требовали внесения изменений в базовый класс. Теперь, не изменяя базовый класс, мы можем добавлять больше одежды, а также изменять размер скидки на отдельный вид одежды по мере необходимости.

## 3. Принцип подстановки Барбары Лисков. Liskov's Substitution Principle

* ##### Subtypes must be substitutable for their base types
* ##### Functions that use references to base classes must be able to use objects of derived classes without knowing it

1. Подклассы не могут замещать поведение базовых классов
2. Функции, которые используют базовый тип должны иметь возможность его использовать (т.е. экземпляры базового типа) не зная об этом.

Принцип подстановки Лисков был одним из самых сложных принципов для меня, и чтобы понять его правильно, мне пришлось посмотреть различные примеры в Интернете. Я считаю, что после осознания, этот принцип станет одним из самых простых, среди принципов, которых следует придерживаться при разработке объектно-ориентированных приложений.

Принцип подстановки Лисков гласит: «Объекты в программе должны быть заменяемы экземплярами их подтипов без ущерба корректности работы программы».

Когда мы говорим о наследовании. Когда какой-то класс наследуется другой класс. И этом можно проверить с помощью глагола is.

наследование == 'is a'

Администратор - это СуперПользователь?

class Администратор(СуперПользователь)

если администратор является Суперпользователем, тогда мы его наследуем от Суперпользователя. Менеджер является Суперпользователем? Тогда мы его не наследуем.

Что нам это дает?

Рассмотрим на примере.

У птицы есть метод лети

class Птица:  
 def лети(self):  
 print("Полетели")

далее пишем функцию полетаем

def полетаем(птица):  
 if isinstance(птица, Птица):  
 птица.лети()

создаем сокола и прокидываем его в функцию полетаем

class Сокол(Птица): pass  
  
  
сокол = Сокол()  
полетаем(сокол)

Полетели?

Создаем еще один класс Страус

class Страус(Птица): pass  
  
  
страус = Страус()  
полетаем(страус)

полетели? Ага. Но это же бага.

Согласно принципа Лисков – функции которые используют базовый тип, должны иметь возможность использовать подтипы сокола, страуса, не зная об этом. И это должно быть корректным. Если это птица – то она должна летать, а мы промахнулись – у нас некорректное наследование.

Что делать? Как это нужно реализовать?

Мы должны описать нечто общее для птиц. К примеру, все птицы имею крылья, она умеет ходить. Точно? Или нет?

Оказывается стриж, ласточка и колибри не умею ходит, умеют только цепляться. А воробей только прыгает.

Что мы делаем. Убираем из базового класса метод лети.

Опасности:

* #### Предусловия могут быть усилены в подклассе
* #### Постусловия могут быть ослаблены в подклассе

class Птица:  
 def лети(self):  
 print("Полетели")  
 return 'сколько-то километров'

Функция ориентируется только на базовый класс. Баг!

И снова функция полетаем запускает птицу

def полетаем(птица):  
 if isinstance(птица, Птица):  
 расстояние = птица.лети()  
 print(расстояние)

что может произойти

class Сокол(Птица):  
 def лети(self, x): *# !!! появился x ! Наследование же!* print("Полетели")  
 *# !!!*

за один момент сделали две плохие вещи.

Мы усилили некорректное предусловие. И ослабили постусловие. Мы его убрали.

Т.е если даже птицы

сокол = Сокол()  
полетаем(сокол)

Получаем ошибку, потому-что у нас потерялся один аргумент

TypeError: Сокол.лети() missing 1 required positional argument: 'x'.

Т.е мы передали, а там его нет. Как исправить багу? Удалить x.

class Сокол(Птица):  
 def лети(self): *# !!!* print("Полетели")  
 return 100

Полетели

100

Однако если программист напишет так:

class Сокол(Птица):  
 def лети(self): *# !!!* print("Полетели")  
 *#return 100*

то в результате получим None

Полетели

None

Это значит мы ослабили пост-условие. Ничего не вернули, а д.были вернуть в эту строчку

print(расстояние)

Засада. Этот принцип нужно постоянно проверять. Выключаем сам себе-режиссера. Вас трое, один программирует Птицу, второй страуса, третий сокола.

Но следуем не слепо, а вдумчиво, Какая перед нами стоит задача.

Еще один пример:

class Car():  
 def \_\_init\_\_(self, type):  
 self.type = type  
  
class PetrolCar(Car):  
 def \_\_init\_\_(self, type):  
 self.type = type  
  
car = Car("SUV")  
car.properties = {"Color": "Red", "Gear": "Auto", "Capacity": 6}  
  
petrol\_car = PetrolCar("Sedan")  
petrol\_car.properties = ("Blue", "Manual", 4)

что не так?

cars = [car, petrol\_car]  
  
def find\_red\_cars(cars):  
 red\_cars = 0  
 for car in cars:  
 if car.properties['Color'] == "Red":  
 red\_cars += 1  
 print(f'Number of Red Cars = {red\_cars}')  
  
find\_red\_cars(cars)

Как видно из кода, мы пытаемся просмотреть список объектов Car. Именно здесь мы нарушаем принцип подстановки Лисков, поскольку мы не можем заменить объекты супертипа Car объектами подтипа PetrolCar внутри функции поиска красных автомобилей.

Лучшим варианты было бы реализовать методы setter и getterв суперклассе Car. С их помощью мы можем устанавливать и получать свойства автомобиля, не оставляя эту реализацию последующим разработчикам. Таким образом, мы просто получаем свойства с помощью метода setter, и его реализация остается инкапсулированной в суперклассе.

Так мы сможем соблюсти принцип подстановки Лисков, как показано ниже:

class Car():  
 def \_\_init\_\_(self, type):  
 self.type = type  
 self.car\_properties = {}  
  
 def set\_properties(self, color, gear, capacity):  
 self.car\_properties = {"Color": color, "Gear": gear, "Capacity": capacity}  
  
 def get\_properties(self):  
 return self.car\_properties  
  
  
class PetrolCar(Car):  
 def \_\_init\_\_(self, type):  
 self.type = type  
 self.car\_properties = {}  
  
  
car = Car("SUV")  
car.set\_properties("Red", "Auto", 6)  
  
petrol\_car = PetrolCar("Sedan")  
petrol\_car.set\_properties("Blue", "Manual", 4)  
  
cars = [car, petrol\_car]  
  
def find\_red\_cars(cars):  
 red\_cars = 0  
 for car in cars:  
 if car.get\_properties()['Color'] == "Red":  
 red\_cars += 1  
 print(f'Number of Red Cars = {red\_cars}')  
  
find\_red\_cars(cars)

Принцип подстановки Лисков гласит: «Объекты в программе должны быть заменяемы экземплярами их подтипов без ущерба корректности работы программы». Дядя Боб включил этот принцип в число 5 лучших принципов проектирования SOLID.

## 4. The Interface Segregation Principle. Принцип (сегрегации) разделения интерфейса

#### Clients should not be forced to depend upon interfaces that they do not use

Принцип разделения интерфейсов был предложен Робертом К. Мартином, когда он консультировал компанию Xerox.

Принцип разделения интерфейсов гласит, что «Ни один клиент не должен зависеть от методов, которые он не использует».

По-русски: много специализированных интерфейсов – лучше, чем один специальный.

Вспомним нашу птицу. Её интерфейс. И летай она у нас тоже

import abc  
  
  
class ИнтерфейсПтицы(abc.ABC):  
  
 @abc.abstractmethod  
 def ешь(self):  
 pass  
  
 @abc.abstractmethod  
 def иди(self):  
 pass  
  
 @abc.abstractmethod  
 def чирикай(self):  
 pass  
  
 @abc.abstractmethod  
 def лети(self):  
 pass

Как правильно?! Вот так! Наша абстрактная птица может только есть.

class ИнтерфейсПтицы(abc.ABC):  
  
 @abc.abstractmethod  
 def ешь(self):  
 pass  
 *# Другие абстрактные методы*class ИнтерфейсЛетающейПтицы(ИнтерфейсПтицы):  
 @abc.abstractmethod  
 def лети(self):  
 pass  
  
  
def полетаем(птица):  
 if isinstance(птица, ИнтерфейсЛетающейПтицы):  
 расстояние = птица.лети()  
 print(расстояние)  
  
  
class Сокол(ИнтерфейсЛетающейПтицы):  
 def ешь(self):  
 print("Поел")  
 *# Реализация остальных методов* def лети(self):  
 print("Полетели")  
  
  
class Страус(ИнтерфейсПтицы):  
 def ешь(self):  
 print("Поел")  
 *# Реализация остальных методов*

Интерфейс летающей птицы цепляет интерфейс абстрактной птицы.

Т.о. мы избегаем использования методов, которые как бы не нужны, которые в предыдущем случае подвисали или заставляли страуса летать.

Функция полетаем проверяем наличие у птицы летательного аппарата (крыльев). И хрестоматийно реализует его интерфейс.

Сокол – интерфейс летающие птицы, Страус – простой птицы.

Напоминаем, что счас мы разбираем букву I у SOLID

Придумаем еще один практический пример для этого принципа

Принцип разделения интерфейсов предполагает создание небольших интерфейсов, известных как «ролевые интерфейсы», вместо большого интерфейса, состоящего из нескольких методов. Разделяя методы по ролям на более мелкие интерфейсы, клиенты будут зависеть только от методов, которые имеют к ним отношение.

Допустим, мы разрабатываем приложение для различных коммуникационных устройств. Мы говорим, что устройство связи – это устройство, которое будет иметь одну или несколько из следующих функций: совершать звонки, отправлять SMS или искать в Интернете. Итак, мы создаем интерфейс с именем CommunicationDevice и добавляем соответствующие абстрактные методы для каждой из этих функций, чтобы любой создаваемый класс смог реализовать эти методы.

import abc  
  
  
class CommunicationDevice(abc.ABC):  
 @abc.abstractmethod  
 def make\_calls(self):  
 pass  
  
 @abc.abstractmethod  
 def send\_sms(self):  
 pass  
  
 @abc.abstractmethod  
 def browse\_internet(self):  
 pass  
  
  
class SmartPhone(CommunicationDevice):  
 def make\_calls(self):  
 *# implementation* pass  
  
 def send\_sms(self):  
 *# implementation* pass  
  
 def browse\_internet(self):  
 *# implementation* pass  
  
class LandlinePhone(CommunicationDevice):  
 def make\_calls(self):  
 *# implementation* pass  
  
 def send\_sms(self):  
 *# just pass or raise exception as this feature is not supported* pass  
  
 def browse\_internet(self):  
 *# just pass or raise exception as this feature is not supported* pass

Затем мы создаем класс SmartPhone с помощью интерфейса CommunicationDevice и реализуем функционал абстрактных методов. До сих пор все было в порядке.

Теперь предположим, что нам нужно создать стационарный телефон. Он тоже является устройством связи, поэтому мы создаем новый класс LandlinePhone через тот же интерфейс CommunicationDevice. Именно здесь мы сталкиваемся с проблемой из-за объемного интерфейса CommunicationDevice.

В классе LandlinePhone мы реализовываем метод make\_calls(), но поскольку мы также наследуем абстрактные методы send\_sms() и browse\_internet(), мы должны предоставить реализацию и этих двух абстрактных методов в классе LandlinePhone, даже если они в принципе неприменимы к этому виду телефонов.

Мы можем либо создать исключение, либо оставить pass вместо реализации, но нам все равно нужно ее предоставить.

Все можно исправить, следуя принципу разделения интерфейсов, как в примере ниже. Вместо создания большого интерфейса мы создаем более маленькие ролевые интерфейсы для каждого метода. Соответствующие классы будут использовать только связанные интерфейсы.

Правильная реализация

import abc  
  
class CallingDevice():  
 @abc.abstractmethod  
 def make\_calls(self):  
 pass  
  
class MessagingDevice():  
 @abc.abstractmethod  
 def send\_sms(self):  
 pass  
  
class InternetbrowsingDevice():  
 @abc.abstractmethod  
 def browse\_internet(self):  
 pass  
  
class SmartPhone(CallingDevice, MessagingDevice, InternetbrowsingDevice):  
 def make\_calls(self):  
 *#implementation* pass  
  
 def send\_sms(self):  
 *#implementation* pass  
  
 def browse\_internet(self):  
 *#implementation* pass  
  
class LandlinePhone(CallingDevice):  
 def make\_calls(self):  
 *#implementation* pass

Т.о., повторяем, много специализированных интерфейсов – лучше, чем один специальный.

Последний D-принцип

## 5. Принцип инверсии зависимостей Dependency inversion principle.

* ##### High-level modules should not depend on low-level modules
  + ##### Both should depend on abstractions
* ##### Abstractions should not depend on details
  + ##### Details should depend on abstractions

Принцип инверсии зависимостей гласит:

1. Модуль высокого уровня не должен зависеть от модулей низкого уровня. И то, и другое должно зависеть от абстракций.

2. Абстракции не должны зависеть от деталей реализации. Детали реализации должны зависеть от абстракций.

Если ваш код уже реализует принципы открытости/закрытости и подстановки Лисков, он уже будет неявно согласован с принципом инверсии зависимостей.

# Выводы

Вспомним самое начало занятий по ООП:

Фундаментальные основы

Программа: чтение с клавиатуры; вывод в терминал.

def process():

msg = input('Введите что-нибудь: ')

print(msg)

process()

Добавляем: чтение из файла

def file\_read():

return "Зачитали файл"

def process(from\_file=False):

if from\_file:

msg = file\_read()

else:

msg = input('Введите что-нибудь: ')

print(msg)

process()

process(True)

Добавляем: отправить данные по сети

def file\_read():

return "Зачитали файл"

def from\_terminal():

return input('Введите что-нибудь: ')

def to\_terminal(msg):

print(f'Послали в терминал: {msg}')

def send\_message(msg):

print(f"Послали по сети: {msg}")

def process(from\_file=False, send\_to=False):

if from\_file:

msg = file\_read()

else:

msg = from\_terminal()

if send\_to:

send\_message(msg)

else:

to\_terminal(msg)

process()

process(True)

process(True, True)

process(False, True)

Грязно

Много условных выражений

Сложно поддерживать

Сложно понять

Нас спасёт ООП!

Как бы мы теперь, знаю ООП спроектировали дизайн такого приложения?

class FileRead:

def \_\_init\_\_(self, name):

self.\_name = name

def read(self):

print(f'Читаем из файла {self.\_name}')

return 'Прочитали из файла'

class KeyboardRead:

def readln(self):

print('Читаем с клавиатуры')

return 'Прочитали ввод с клавиатуры'

class SendMessage:

def send(self, msg):

print(f'Послали сообщение: {msg}')

и всё объединяем классом Processing.

class Processing:

def \_\_init\_\_(self, read, send):

if not isinstance(read, FileRead):

raise TypeError('Ошибка при чтении из файла')

if not isinstance(send, SendMessage):

raise TypeError('Ошибка при отправке сообщения')

self.\_reader = read

self.\_sender = send

def process(self):

msg = self.\_reader.read()

self.\_sender.send(msg)

f = FileRead("file.txt")

s = SendMessage()

p = Processing(f, s)

p.process()

Возвращаясь к задачке с бумагой, ручкой и текст.

Правильный ответ написать четвертый класс – Писатель (Writer).

Которые брал ручку, бумагу и на пиши текст.

С одной стороны всё написано правильно и структурировано. Однако…

Класс Processing нарушает принцип инверсии зависимостей. Классы (модули) верхних уровней не должны зависеть от модулей нижних уровней.

Но оба типа модулей должны зависеть от абстракций.

Абстракции не должны зависеть от деталей.

А детали должны зависеть от абстракций.

Что означает модули верхних, нижних уровней.

Наш класс Processing является высокоуровневым. Он главный, он объединяет, он выше.

Все остальные модули – низкоуровневые. А в чем проблема?

Наш класс Processing очень четко зависит от низкоуровневых модулей. Проверка

Isinstance на FileRead и SendMessage.

А если у нас появился класс KeyboardRead мне снова нужно залезать в него, а вдруг у нас появится голосовой ввод или рукописный ввод с тачпада

А предыдущие принципы о чем говорят? Класс д.б. закрыт для изменений. Только ошибки должны фикситься. Нужно от чего то наследоваться. Но так, чтобы не запутаться в наследованиях.

Processing для чтения из файлы, с клавиатуры, с микрофона и т.п. Это будет бардак.

Кто виноват и что делать?

Почему в классе FileRead функция называется read, а в KeyboardRead – readln?

А в функции

def process(self):  
 msg = self.\_reader.read()  
 self.\_sender.send(msg)

я дергаю read. А как быть, если придет readln.

Придется вносить изменения. А это нарушения принципа

Внесение изменений потребует изменения. Какой буквы? Правильно O!

Класс Processing не должен беспокоится, как читаются данные.

И снова нам на помощь приходит абстракция!

Опять пишем интерфейс. Для чтения и записи. Один только читает, второй еще и принимает месседж.

class IReader(abc.ABC):  
 @abc.abstractmethod  
 def read(self):  
 pass  
  
class IWriter(abc.ABC):  
 @abc.abstractmethod  
 def write(self, msg):  
 pass  
  
Все читательные и писательные класс должны реализовать эти интерфейсы (абстракции).

Класс InputKeyboard читает с клавиатуры и описывает read.

class InputKeyboard(IReader):  
 def read(self):  
 return 'Прочитали ввод с клавиатуры'

Класс OutputNetwork наследует IWriter и реализует write

class OutputNetwork(IWriter):  
 def write(self, msg):  
 print(f"Послали сообщение {msg} по сети")

А теперь самый высокоуровневый Processing. Его реализация. Ему приходит ввод-вывод (откуда-куда). Но проверка не на конкретный класс, а на интерфейс.

Ему абсолютно без разницы, что сюда придет. Читалка из файла или с клавиатуры. Самое главное – что у него есть метод read.

А у второго, соответсвенно, метод write.

class Processing:  
 def \_\_init\_\_(self, input, output):  
 if not isinstance(input, IReader):  
 raise TypeError('Ошибка при чтении данных')  
 if not isinstance(output, IWriter):  
 raise TypeError('Ошибка при записи данных')  
 self.\_input = input  
 self.\_output = output  
  
 def process(self):  
 msg = self.\_input.read()  
 self.\_output.write(msg)

Теперь наш высокоуровневый Processing не зависит от низкоуровневых уровней. Но все они зависят от абстракций.

Абстракция не зависит не от какой конкретики, а конкретика зависит от абстракции. ВСЁ.

Классика жанра!

Создали соответствующие экземпляры

kbd = InputKeyboard()  
net = OutputNetwork()

и отправили их в Processing

proc = Processing(kbd, net)

proc.process()

Добавляем класс чтения из файла:

class InputFile(IReader):  
 def read(self):  
 return 'Прочитали из файла'

class Processing стоит как вкопанный. Идеальная ситуация.

И нам не важно, что они возвращают (строку или число), главное, чтобы они работали вместе.

Следуя принципу открытости/закрытости, вы создаете интерфейсы, которые можно использовать для предоставления различных высокоуровневых реализаций.

Следуя принципу подстановки Лисков, вы гарантируете, что сможете заменить экземпляры класса низкого уровня объектами класса высокого уровня без какого-либо негативного воздействия на приложение.

Таким образом, следуя этим двум принципам, вы гарантируете, что ваши классы высокого уровня и классы низкого уровня зависят от интерфейсов. Следовательно, вы неявно следуете принципу инверсии зависимостей.

Еще один пример:

есть класс Student, который мы используем для создания экземпляров Student и класса TeamMemberships, который содержатся сведения о принадлежности учеников к разным командам.

Теперь мы определим высокоуровневый класс Analysis, где нам нужно отсеять всех учеников, принадлежащих красной команде.

Как видно из реализации, мы напрямую используем team\_student\_memberships.team\_memberships в высокоуровневом классе Analysis, и мы используем реализацию этого списка непосредственно в классе высокого уровня. На данный момент все нормально, но представьте ситуацию, в которой нам нужно изменить эту реализацию со списка на что-то другое. В этом случае наш класс высокого уровня Analysis сломается, поскольку он зависит от деталей реализации TeamMemberships низкого уровня.

from enum import Enum  
  
class Teams(Enum):  
 BLUE\_TEAM = 1  
 RED\_TEAM = 2  
 GREEN\_TEAM = 3  
  
class Student:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
class TeamMemberships():  
 def \_\_init\_\_(self):  
 self.team\_memberships = []  
  
 def add\_team\_memberships(self, student, team):  
 self.team\_memberships.append((student, team))  
  
class Analysis():  
 def \_\_init\_\_(self, team\_student\_memberships):  
 memberships = team\_student\_memberships.team\_memberships  
 for members in memberships:  
 if members[1] == Teams.RED\_TEAM:  
 print(f'{members[0].name} is in RED team')  
  
student1 = Student('Bob')  
student2 = Student('Tom')  
student3 = Student('Mike')  
  
team\_memberships = TeamMemberships()  
team\_memberships.add\_team\_memberships(student1, Teams.BLUE\_TEAM)  
team\_memberships.add\_team\_memberships(student2, Teams.RED\_TEAM)  
team\_memberships.add\_team\_memberships(student3, Teams.GREEN\_TEAM)

Теперь взгляните на пример ниже, в котором мы меняем эту реализацию и приводим ее в соответствие с принципом инверсии зависимостей.

from enum import Enum  
from abc import abstractmethod  
  
class Teams(Enum):  
 BLUE\_TEAM = 1  
 RED\_TEAM = 2  
 GREEN\_TEAM = 3  
  
class TeamMemberShipLookup():  
 @abstractmethod  
 def find\_all\_students\_of\_team(self, team):  
 pass  
  
class Student:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
class TeamMemberships(TeamMembershipLookup):  
 def \_\_init\_\_(self):  
 self.team\_memberships = []  
  
 def add\_team\_memberships(self, student, team):  
 self.team\_memberships.append((student, team))  
  
 def find\_all\_students\_of\_team(self, team):  
 for members in self.team\_memberships:  
 if members[1] == team:  
 yield members[0].name  
  
class Analysis():  
 def \_\_init\_\_(self, team\_membership\_lookup):  
 for student in team\_membership\_lookup.find\_all\_students\_of\_team(Teams.RED\_TEAM):  
 print(f'{student} is in RED team.')  
  
student1 = Student('Tom')  
student2 = Student('Bob')  
student3 = Student('Mike')  
  
team\_memberships = TeamMemberships()  
team\_memberships.add\_team\_memberships(student1, Teams.BLUE\_TEAM)  
team\_memberships.add\_team\_memberships(student2, Teams.RED\_TEAM)  
team\_memberships.add\_team\_memberships(student3, Teams.GREEN\_TEAM)  
  
Analysis(team\_memberships)

Чтобы следовать принципу инверсии зависимостей, нам необходимо убедиться, что класс высокого уровня Analysis не зависит от конкретной реализации класса низкого уровня TeamMembership. Вместо этого он должен зависеть от некоторой абстракции.

Итак, мы создаем интерфейс TeamMembershipLookup, который содержит абстрактный метод find\_all\_students\_of\_team, передающийся любому классу, наследующему этот интерфейс. Мы наследуем наш класс TeamMembership от этого интерфейса, следовательно, теперь класс TeamMembership должен предоставлять реализацию функции find\_all\_students\_of\_team. Затем эта функция передает результаты любому другому вызывающему ее объекту. Мы перенесли обработку, которая делалась в классе высокого уровня Analysis в TeamMemberships через интерфейс TeamMembershipLookup.

Сделав все это, мы убрали зависимость класса Analysis от класса TeamMemberships и перенесли ее в интерфейс TeamMembershipLookup. Теперь класс высокого уровня не зависит от деталей реализации класса низкого уровня. Любые изменения в деталях реализации класса низкого уровня не влияют на класс высокого уровня.

Итог:

|  |  |
| --- | --- |
| Принцип | Смысл |
| Принцип единственной ответственности | У класса должна быть всего одна причина для изменения. |
| Принцип открытости/закрытости | Сущности программы (классы, модули, функции и т.п.) должны быть открыты для расширения, но закрыты для изменений. |
| Принцип подстановки Барбары Лисков | Объекты в программе должны быть заменяемы экземплярами их подтипов без ущерба корректности работы программы. |
| Принцип разделения интерфейсов | Ни один клиент не должен зависеть от методов, которые он не использует. |
| Принцип инверсии зависимостей | Модуль высокого уровня не должен зависеть от модулей низкого уровня. И то, и другое должно зависеть от абстракций.  Абстракции не должны зависеть от деталей реализации. Детали реализации должны зависеть от абстракций. |

Сложновато.

Для этих целей есть такое понятие как шаблоны проектирования.