*Урок 3. Принципы проектирования*

Оглавление

[Введение 2](#__RefHeading___1)

[Общие принципы проектирования 2](#__RefHeading___2)

[Методы проектирования 3](#__RefHeading___3)

[Принцип DRY 3](#__RefHeading___4)

[Принцип KISS 3](#__RefHeading___5)

[Принципы SOLID 4](#__RefHeading___6)

[ПРИНЦИП ЕДИНСТВЕННОЙ ОТВЕТСТВЕННОСТИ (The Single Responsibility Principle) - SRP 4](#__RefHeading___7)

[ПРИНЦИП ОТКРЫТОСТИ / ЗАКРЫТОСТИ (The Open Closed Principle) - OCP 5](#__RefHeading___8)

[ПРИНЦИП РАЗДЕЛЕНИЯ ИНТЕРФЕЙСА (The Interface Segregation Principle) - ISP 10](#__RefHeading___9)

[ПРИНЦИП ИНВЕРСИИ ЗАВИСИМОСТЕЙ (The Dependency Inversion Principle) - DIP 13](#__RefHeading___10)

[ПРИНЦИП ПОДСТАНОВКИ БАРБАРЫ ЛИСКОВ (The Liskov Substitution Principle) - LSP 14](#__RefHeading___11)

[Пара слов о GRASP 18](#__RefHeading___12)

# Введение

Сегодня мы рассмотрим принципы проектирования и больший упор сделаем на принципы SOLID, хотя ранее популярность имели принципы GRASP.

НО! Сперва, что такое **проектирование**? Говоря о проектировании, мы имеем в виду создание спецификации продукта, предназначенного для достижения заявленных целей, используя набор примитивных компонентов, имеющих заданные ограничения.

Имеется в виду уже более детальное проектирование программы, например, на базе классов или функций, или модулей и особенности их взаимодействия.

# Общие принципы проектирования

Принципы – это направления, куда нам предлагают двигаться.

* **Рассмотрение альтернативных подходов**. Часто мы решаем задачи по шаблону, как обычно привыкли, но в идеале нужно сравнить несколько вариантов. Делать функцию или класс, с какой структурой и т.д.
* **Разбиение на модули**. Каждая подсистема (модуль) должна быть законченным логическим объектом. Например, система «Магазин» делится на подсистемы: «Продажи» и «Аналитика». Далее деление продолжается до небольших фрагментов – функций. Функции также могут делиться. Каждая часть логически завершена и решает какую-то задачу.
* **Повторное использование блоков**. Система должна быть спроектирована так, чтобы ее компоненты можно было использовать повторно. Как в рамках текущей системы, так и в других системах. Это позволяет сделать систему гибкой и тестируемой.
* **Сокращение интеллектуальной дистанции**. Это, по сути, DDD.
* **Единообразие и целостность**. Имеется в виду, что мы выбрали некоторую архитектуру и «держим» это направление.
* **Возможность внесения изменений**. Заранее попытаться определить, какие будут изменения. Желательно быть готовыми к различного рода ситуациям и соответствующим изменениям. Например, мы проектируем систему «Магазин» и знаем, что у нас изменится система оплаты. Зная это, мы, например, вместо одной функции реализуем два класса. В результате система становится более гибкой.
* **Проектирование – это не кодирование**. Обычно, особенно у новичков, эти процессы сливаются в один общий. НО! Желательно их разделять.

# Методы проектирования

Методы – это уже конкретно, что делать.

* **Абстракция**. Реализовывать структурные блоки в том числе и для сущностей, не имеющих реального исполнения, например, классы «сокет», «сервер» и т.д.
* **Декомпозиция**. Нам предлагается разбивать систему на части и абстрагироваться от других частей. В примере «Магазин» абстракции – это, например, подсистемы «Продажи» и «Аналитика». А самая большая абстракция – сам магазин. В каждой подсистеме также можно сделать разделение и при реализации каждой части абстрагируемся от других частей. Ключевой смысл – забыть обо всем остальном.
* **Модульность**. Деление на модули для повторного использования.
* **Использование моделей**. Здесь имеется в виду DDD и проектирование на основе предметной области.
* **Тестирование**. Важный принцип – покрывать модули тестами.

# Принцип DRY

Дословно он гласит – «не повторяйся» или «не допускай дублирования».

Самый простой способ его отражения в своих проектах – делать рефакторинг кода, когда вы видите повторения. Заранее предугадать эти повторения мы вряд ли сможем. Поэтому лучше об этом не беспокоиться, а просто делать рефакторинг кода. Таким образом, наша система автоматически станет модульной.

# Принцип KISS

Дословно он гласит – «будь проще».

Не усложнять без необходимости. Идея в том, чтобы не гнаться за слишком гибкой системой, которая может все! А сосредоточиться на решении задачи и сделать это минимальными усилиями. Не такой точный принцип, как DRY.

Это базовые принципы, интуитивно понятные.

# Принципы SOLID

Сформулированы Робертом Мартином, повидавшем множество систем. Рассмотрим пять, хотя их больше. Но это и хорошо, что их мало, иначе начнут друг другу противоречить. Принципы SOLID весьма абстрактны.

## ПРИНЦИП ЕДИНСТВЕННОЙ ОТВЕТСТВЕННОСТИ (The **S**ingle **R**esponsibility **P**rinciple) - SRP

**«Класс должен иметь только одну причину для изменения»** – дословная формулировка.

**«Принцип ООП, означающий, что каждый объект должен иметь одну ответственность и эта ответственность должна быть полностью инкапсулирована в класс»** – адаптиваная формулировка.

Смысл – класс должен описывать один объект, а функция – делать одно дело.

**Листинг 1. Урок 3. Коды к уроку/srp.py**

|  |
| --- |
| *# bad* **class** Order:  **def** get\_items(self):  **pass   def** get\_total(self):  **pass   def** validate(self):  **pass   def** save(self):  **pass   def** load(self):  **pass** *# good* **class** Order:  **def** get\_items(self):  **pass   def** get\_total(self):  **pass   def** validate(self):  **pass   class** Repository:  **def** save(self):  **pass   def** load(self):  **pass** |

Есть класс **Order** с пятью методами: get\_items, get\_total, validate, save, load.

На первый взгляд все нормально, но почему написано bad??

Можно увидеть, что это класс занимается как работой с данными: get\_items, get\_total, так и валидацией, сохранением в базу, загрузкой из базы.

То есть нарушается принцип SRP, ведь класс делает слишком много. И делает то, что он делать не должен – save, load.

Как исправить ситуацию?

Предлагается разделить класс на два: **Order** и **OrderRepository**.

В **Order** мы оставим get\_items, get\_total, validate. Почему? Потому что эти методы непосредственно связаны именно с заказом.

А в класс **OrderRepository** вынести методы save, load. Эти методы могут быть не только у заказа. Теперь принцип SRP не нарушается.

Каждый класс «знает» о том, что ему делать и делает только свою работу. Например, ведь заказу не нужно знать, как там все сохраняется.

Плюс какой еще недостаток в прежнем подходе? Если мы оставим все как есть, то придется в каждом классе прописывать методы save, load. И получится уже нарушение принципа DRY.

## ПРИНЦИП ОТКРЫТОСТИ / ЗАКРЫТОСТИ (The **O**pen **C**losed **P**rinciple) - OCP

Весьма противоречивый принцип с не совсем понятной формулировкой.

«Программные сущности (классы, модули, функции) должны быть открыты для расширения, но закрыты для изменения».

Непонятно в каком контексте используются понятия: расширение и изменение.

Начнем издалека. Для начала поговорим об абстрактных классах.

**Листинг 2. Урок 3. Коды к уроку/ocp\_1.py**

|  |
| --- |
| **class** Figure:  **def** draw(self):  **pass   class** Circle(Figure):  **def** draw(self):  print(**'circle'**)   **class** Triangle(Figure):  **def** draw(self):  print(**'triangle'**) |

У нас есть три класса. Причем в каждом прописан метод **draw()**.

В базовом классе Figure этот метод определяется, в наследниках – переопределяется.

Пока все хорошо.

**Листинг 3. Урок 3. Коды к уроку/ocp\_2.py**

|  |
| --- |
| **class** Figure:  **def** draw(self):  **pass   class** Circle(Figure):  **def** draw(self):  print(**'circle'**)   **class** Triangle(Figure):  **def** draw(self):  print(**'triangle'**)   **class** Romb(Figure):  **pass** romb = Romb() print(romb.draw()) |

Представьте, что мы добавили новый класс **Romb**. Но забыли у него переопределить метод **draw()**.

Как бы сделать так, чтобы программа указала нам, что мы забыли переопределить метод **draw()**?

**Листинг 4. Урок 3. Коды к уроку/ocp\_3.py**

|  |
| --- |
| **import** abc   **class** Figure(abc.ABC):  @abc.abstractmethod  **def** draw(self):  **pass   class** Circle(Figure):  **def** draw(self):  print(**'circle'**)   **class** Triangle(Figure):  **def** draw(self):  print(**'triangle'**)   **class** Romb(Figure):  **pass** romb = Romb() print(romb.draw()) |

Этот класс **Figure** называется абстрактным классом! А в нем есть абстрактный метод. Почему он называется абстрактным? Потому что мы не знаем реализацию метода. Есть фигура, ее можно рисовать! А как? Мы не знаем. Абстрактный класс еще называют интерфейсом. Это класс с набором методов, но без реализации.

Зачем он нужен? Чтобы определить какие методы должны быть у наследников и что наследники обязательно должны эти методы реализовать.

Хороший стиль – использование абстрактных классов, хотя они не являются обязательными в применении.

Теперь запуск даст ошибку:

romb = Romb()

TypeError: Can't instantiate abstract class Romb with abstract methods draw

Проверка наличия метода **draw()** у наследников работает.

**Листинг 5. Урок 3. Коды к уроку/ocp\_4.py**

|  |
| --- |
| *# Фигура* **class** Figure:  **pass** *# Круг* **class** Circle(Figure):  **pass** *# Треугольник* **class** Triangle(Figure):  **pass   class** Romb(Figure):  **pass   class** Line(Figure):  **pass** *# САПР* **class** CAD:  @classmethod  **def** draw\_all(cls, figures):  **for** figure **in** figures:  *# выбор поведения в зависимости от типа входного объекта* **if** isinstance(figure, Circle):  cls.draw\_circle(figure)  **elif** isinstance(figure, Triangle):  cls.draw\_triangle(figure)   *# рисуем круг* @staticmethod  **def** draw\_circle(circle):  **pass** *# рисуем треугольник* @staticmethod  **def** draw\_triangle(triangle):  **pass** |

У нас есть класс САПР (CAD). Он рисует две фигуры. В метод **draw\_all()** мы передаем список объектов фигур, которые нужно отрисовать.

Рассмотрим использование примера ocp\_4.py.

**Листинг 6. Урок 3. Коды к уроку/use\_ocp\_4.py**

|  |
| --- |
| **from** ocp\_4 **import** CAD, Figure   **class** Romb(Figure):  **pass** figures = [Romb()] CAD.draw\_all(figures) |

Запускаем и ничего нет. Почему?

Потому что в классе **CAD** нет метода для отрисовки ромба.

Чтобы теперь работать с ромбом, нам придется изменить класс **CAD**.

В этом и нарушение принципа OCP. Мы не можем добавить отрисовку ромба, не изменив класс **CAD**. А нужно, чтобы мы могли расширять поведение системы без изменения ее частей, т.е. в примере это без изменения класса **CAD**!

Как решить эту проблему?

**Листинг 7. Урок 3. Коды к уроку/ocp\_5.py**

|  |
| --- |
| **import** abc   *# Абстрактная фигура* **class** Figure(abc.ABC):  @abc.abstractmethod  **def** draw(self):  **pass** *# Круг* **class** Circle(Figure):  **def** draw(self):  **pass** *# Треугольник* **class** Triangle(Figure):  **def** draw(self):  **pass   class** Romb(Figure):  **def** draw(self):  **pass   class** Line(Figure):  **def** draw(self):  **pass   class** Line(Figure):  **def** draw(self):  **pass   class** Square(Figure):  **def** draw(self):  **pass   class** Romb(Figure):  **def** draw(self):  **pass** *# САПР* **class** CAD:  @classmethod  **def** draw\_all(cls, figures):  **for** figure **in** figures:  figure.draw() |

Разместим метод **draw()** внутрь класса **Figure**. А также разместим этот метод внутрь классов отдельных фигур. Т.е. теперь рисованием фигуры занимается каждая фигура сама. Теперь класс **CAD** пробегает по фигурам и просто вызывает у каждой метод **draw()**.

Рассмотрим использование примера ocp\_5.py.

**Листинг 8. Урок 3. Коды к уроку/use\_ocp\_5.py**

|  |
| --- |
| **from** ocp\_5 **import** CAD, Figure   **class** Romb(Figure):  **def** draw(self):  print(**'romb'**)   figures = [Romb()] CAD.draw\_all(figures) |

Все работает. Теперь мы можем расширить поведение системы – добавить новые фигуры, без изменения самого класса **CAD**. Его мы больше вообще не трогаем.

## ПРИНЦИП РАЗДЕЛЕНИЯ ИНТЕРФЕЙСА (The **I**nterface **S**egregation **P**rinciple) - ISP

**«Клиенты не должны зависеть от тех методов, которые не используют»** – дословная формулировка.

**«Принцип означающий, что в классе должны быть только методы, которые мы используем и которые нам реально нужны»** – адаптиваная формулировка.

**Листинг 9. Урок 3. Коды к уроку/isp\_1.py**

|  |
| --- |
| **import** abc   **class** Figure(abc.ABC):  @abc.abstractmethod  **def** draw(self):  *'''draws a figure'''* @abc.abstractmethod  **def** plot(self):  *'''plots a figure'''* **class** Circle(Figure):  **def** draw(self):  print(**'draw Circle'**)   **def** plot(self):  print(**'plot Circle'**)   *# направляющая, непечатная* **class** GuideLine(Figure):  **def** draw(self):  print(**'draw Circle'**)   **def** plot(self):  **pass** *# пустой метод, вынужденная реализация   # from django.db import models   # class Order(models.Model): # ... # is\_active = models.BooleanField(blank=True) # is\_group = models.BooleanField(blank=True) # payment = models.ForeignKey(null=True)* |

У нас есть класс **Figure** с двумя методами! Т.е. существует некоторая абстрактная фигура, которую мы рисуем и печатаем.

Создадим конкретную фигура – класс **Circle**, мы его рисуем и печатаем.

Но появляется класс **GuideLine**. Это специальный объект, который печатать не нужно.

def plot(self):

pass *# пустой метод, вынужденная реализация*

Вот оно нарушение принципа разделения интерфейса. Нам приходится в классе круга прописывать метод, который там не нужен, но без него мы получим ошибку.

Вспомним про Джанго!

Есть атрибут, который может быть только у нескольких товаров, а у остальных его можно не ставить – blank = True. Получается одна модель хранит и нужные всем заказам атрибуты и нужные только части заказов.

Как решить проблему? Разделить класс на классы и определить в них нужный интерфейс.

**Листинг 10. Урок 3. Коды к уроку/isp\_2.py**

|  |
| --- |
| **import** abc   *# фигура* **class** Figure(abc.ABC):  @abc.abstractmethod  **def** draw(self):  *'''draws a figure'''   # нечто печатаемое* **class** Plottable(abc.ABC):  @abc.abstractmethod  **def** plot(self):  *'''plots a figure'''   # круг, печатаемый - наследуемся от двух классов* **class** Circle(Figure, Plottable):  **def** draw(self):  print(**'draw Circle'**)   **def** plot(self):  print(**'plot Circle'**)   *# направляющая, просто фигура, непечатная* **class** GuideLine(Figure):  **def** draw(self):  print(**'draw Circle'**)  *# from django.db import models   # class Order(models.Model): # ... # is\_active = models.BooleanField(blank=True) # is\_group = models.BooleanField(blank=True) # payment = models.ForeignKey(null=True)  # class Special(Order): # pass # # # class Special(models.Model): # # order = models.ForeignKey(order=True)* |

Нужно разделить класс **Figure** на два класса – **Figure**, **Plottable**. А в конкретных реализациях фигур используем множественное наследование.

А в Джанго это решилось бы так:

*# class Special(Order):  
# pass  
#  
#  
# class Special(models.Model):  
#  
# order = models.ForeignKey(order=True)*

## ПРИНЦИП ИНВЕРСИИ ЗАВИСИМОСТЕЙ (The **D**ependency **I**nversion **P**rinciple) - DIP

**«Все части кода должны зависеть от абстракции и не зависеть от конкретики»** – дословная формулировка.

**Листинг 11. Урок 3. Коды к уроку/dip\_1.py**

|  |
| --- |
| *# просто товар* **class** SimpleItem:  **def** get\_price(self):  **pass** *# заказ* **class** Order:  total = 0 *# итого* **def** add(self, item: SimpleItem): *# добавить товар в заказ* self.total += item.get\_price()   *# превосходный товар* **class** PerfectItem:  **def** get\_price(self):  **pass   class** SuperItem:  **def** get\_price(self):  **pass** |

Здесь используется аннотация, что **item** должен принадлежать типу **SimpleItem**.

Теперь сделаем класс **PerfectItem**, но передать его в **Order** мы уже не можем. Налицо зависимость от конкретики, что, конечно, не есть хорошо.

Хотя аннотации носят рекомендательный характер, все-таки желательно их придерживаться.

Как решить проблему?

**Листинг 12. Урок 3. Коды к уроку/dip\_2.py**

|  |
| --- |
| **import** abc   *# абстракция товара* **class** ItemInterface(abc.ABC):  @abc.abstractmethod  **def** get\_price(self):  *'''returns the price'''   # превосходный товар наследуем от абстрактного товара* **class** PerfectItem(ItemInterface):  **def** get\_price(self):  **pass   class** SuperItem(ItemInterface):  **def** get\_price(self):  **pass   class** Order:  ...   *# добавить абстрактный товар, детали реализации товара не интересуют* **def** add(self, item: ItemInterface):  self.total += item.get\_price() |

Теперь мы ожидаем не конкретный класс, а любой класс, у которого определен метод **get\_price()**. Зависимости от конкретики больше нет.

## ПРИНЦИП ПОДСТАНОВКИ БАРБАРЫ ЛИСКОВ (The **L**iskov **S**ubstitution **P**rinciple) - LSP

**«Должна быть возможность заменить тип подтипом»** – дословная формулировка.

Например, у нас есть абстрактный класс и от него что-то наследуется. Скажем, наследуется класс B. У класса **B** есть наследник – класс **D**. Мы создаем для класса **B** экземпляр.

**b = B()**

И смысл в том, что, если мы вместо **B** поставим его наследника, т.е. **D**, код должен работать без ошибок.

**Т.е. наследник должен расширять поведение родителя, а не изменять.**

**Листинг 13. Урок 3. Коды к уроку/lsp\_1.py**

|  |
| --- |
| **import** unittest   *# Прямоугольник* **class** Rectangle:  @property  **def** width(self):  **return** self.\_\_width   **def** set\_width(self, width):  self.\_\_width = width   @property  **def** height(self):  **return** self.\_\_height   **def** set\_height(self, height):  self.\_\_height = height   @property  **def** area(self):  **return** self.\_\_width \* self.\_\_height   *# Квадрат - ?* **class** Square(Rectangle):  **def** set\_width(self, width):  super().set\_width(width)  super().set\_height(width)   **def** set\_height(self, height):  super().set\_height(height)  super().set\_width(height)   **class** SquareTest(unittest.TestCase):  **def** test\_area(self):  *#* rectangle = Rectangle()  *# нарушение принципа LSP, мы не ожидаем тут, что и высота ТОЖЕ изменится  # Простыми словами — классы-наследники не должны противоречить базовому классу.  # если наследник не расширил, а ИЗМЕНИЛ поведение родителя?* rectangle.set\_width(5)  rectangle.set\_height(4)  *#rectangle.set\_height(40)  #rectangle.set\_height(40)  # тест провален, актуальное значение 16* self.assertEqual(rectangle.area, 20) |

У нас есть прямоугольник – класс **Rectangle**. У него есть методы и свойства.

И есть квадрат – класс **Square**.

Квадрат – частный случай прямоугольника. Это в математике. Но так ли в программе?

Мы думаем, что также и поэтому наследуем квадрат от прямоугольника.

Т.е. квадрат является подтипом для прямоугольника, а прямоугольник – типом для квадрата. Мы должны иметь возможность в клиентском коде заменить тип подтипом и в программе ничего не сломается.

И мы придумали, что когда задаем ширину – set\_width(), то меняется и ширина, и длина. И наоборот, но по такому же принципу работаем с высотой. Потому что стороны одинаковые.

Вот наш клиентский код:

**class** SquareTest(unittest.TestCase):  
 **def** test\_area(self):  
 rectangle = Rectangle()rectangle.set\_width(5)  
 rectangle.set\_height(4)self.assertEqual(rectangle.area, 20)

Запуская пример, мы ожидаем, что получим 20. Все работает!

Заменим теперь тип подтипом.

**class** SquareTest(unittest.TestCase):  
 **def** test\_area(self):  
 rectangle = Square()rectangle.set\_width(5)  
 rectangle.set\_height(4)  
self.assertEqual(rectangle.area, 20)

Тест провален:

AssertionError: 16 != 20

Почему так получилось?

Дело в том, что когда мы на шаге

rectangle.set\_height(4)

меняем высоту, то меняется и ширина.

Поэтому актуальное значение для площади будет не 20, а 16.

Как решить проблему?

**Листинг 14. Урок 3. Коды к уроку/lsp\_2.py**

|  |
| --- |
| **import** unittest   *# Прямоугольник, неизменяемый* **class** RectangleImmutable:  \_\_slots\_\_ = (**'\_width'**, **'\_height'**)   **def** \_\_init\_\_(self, width, height):  super().\_\_setattr\_\_(**'\_width'**, width)  super().\_\_setattr\_\_(**'\_height'**, height)   @property  **def** width(self):  **return** self.\_width   @property  **def** height(self):  **return** self.\_height   **def** \_\_setattr\_\_(self, key, value):  **raise** AttributeError(**'attributes are immutable'**)   @property  **def** area(self):  **return** self.\_width \* self.\_height   *# Квадрат, неизменяемай* **class** SquareImmutable(RectangleImmutable):  **def** \_\_init\_\_(self, size):  super().\_\_init\_\_(size, size) |

Прямоугольник и квадрат мы делаем неизменяемыми. В прямоугольнике две стороны, в квадрате – одна. Эти значения мы передаем в \_\_init\_\_() и на ходу, в клиентском коде значения свойств изменить не сможем.

Проверим:

**class** SquareTest(unittest.TestCase):  
 **def** test\_area(self):  
 rectangle = RectangleImmutable(4, 4)self.assertEqual(rectangle.area, 16)

Работает.

Теперь заменим тип подтипом:

**class** SquareTest(unittest.TestCase):  
 **def** test\_area(self):  
 rectangle = SquareImmutable(4)self.assertEqual(rectangle.area, 16)

Работоспособность клиентского кода сохранилась.

Если же мы захотим на ходу поменять значения ширины или высоты:

**class** SquareTest(unittest.TestCase):  
 **def** test\_area(self):  
 rectangle = SquareImmutable(4)  
rectangle.width = 5  
 self.assertEqual(rectangle.area, 16)

То, получим ошибку:

raise AttributeError('attributes are immutable')

AttributeError: attributes are immutable

# Пара слов о GRASP

Они расписаны в методичке. Почему мы не изучаем их подробно?

Это другой набор принципов, но говорят они почти о том же, что и SOLID.

SOLID более популярные, а GRASP были популярными в прошлом.

Например, Информационный эксперт, по сути, аналог SRP. «Объект, владеющий информацией о чем-то, должен давать соответствующие методы».

Но вот интересный, которого нет, называется Чистая выдумка. – «Если у нас какой-то объект должен за что-то быть ответственным, а такого объекта в жизни нет, то мы его просто придумываем». Например, объект Socket.