# Введение

*СЛАЙД 1-3*

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

*СЛАЙД 4*

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

*СЛАЙД 5*

Принципы – это направления, куда нам предлагают двигаться.

1. Рассмотреть альтернативные подходы. 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.
2. Декомпозиция. 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.
3. 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.
4. Сокращать интеллектуальную дистанцию. Это, по сути, DDD.
5. Единообразие и целостность. Имеется в виду, что мы выбрали некоторую архитектуру и «держим» это направление.
6. Возможность внесения изменений. Заранее попытаться угадать, 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.
7. Проектирование != кодирование. Обычно, особенно у новичков, эти процессы сливаются в один общий. НО! Желательно их разделять.

МЕТОДЫ ПРОЕКТИРОВАНИЯ

*СЛАЙД 6*

Методы – это уже конкретно, что делать.

1. Абстракция.
2. 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.
3. Модульность. Деление на модули для повторного использования.
4. Модель. Здесь имеется в виду DDD и проектирование на основе предметной области.
5. Тестирование. Важный принцип – покрывать модули тестами.

**ВНИМАНИЕ!** В нашем курсе рассматриваются фундаментальные вещи, касаемо архитектуры и паттернов. Т.е. в целом курс может быть воспроизведен и применительно к языку C++, Java и т.д.

Сравним особенности языков, по отношению к Python, которые повлияют на курс.

*СЛАЙД 7*

*Принцип DRY*

*СЛАЙД 8*

Самый простой способ его применять – делать рефакторинг, 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*

*СЛАЙД 9*

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*

*СЛАЙД 10-11*

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.

1. Принцип единственной ответственности (The **S**ingle Responsibility Principle).

*СЛАЙД 12*

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

«Принцип ООП, обозначающий, 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. 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 OrderRepository:  def save(self):  pass   def load(self):  pass |

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

Вроде 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.

Предлагается разделить класс на два: Order, OrderRepository.

В Order мы оставим get\_items, get\_total, validate. Почему? Потому что заказ знает, что в нем лежит. И может делать с эти указанные операции.

А в класс OrderRepository вынести методы save, load. 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 что еще плохого? Если мы оставим все как есть, то придется в каждом классе прописывать методы save, load. И получится уже нарушение принципа DRY.

1. Принцип открытости / закрытости (The **O**pen Closed Principle).

*СЛАЙД 13*

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.

Рассмотрим пример, но для начала поговорим об абстрактных классах. Начнем издалека.

**Листинг 2. 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 вот такая иерархия. У нас есть три класса. Причем в каждом прописан метод draw(). Правда в дочерних классах он переопределяется. Пока все хорошо.

Но представьте, что мы хотим добавить класс Romb. Но мы забыли у него переопределить draw().

**Листинг 3. 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()) |

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

Вот 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.

Научим программу проверять наличие метода draw() у наследников.

**Листинг 4. 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()) |

Теперь запуск даст ошибку:

romb = Romb()

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

Хороший стиль – использование абстрактных классов.

Переходим непосредственно к разбору паттерна OCP:

**Листинг 5. ocp\_4.py**

|  |
| --- |
| *# Фигура* class Figure:  pass   *# Круг* class Circle(Figure):  pass   *# Треугольник* class Triangle(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 фигуры. Здесь сразу есть нарушение нескольких принципов.

Рассмотрим использование примера ocp\_4.py

**Листинг 6. 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 теперь это решить?

Положим метод draw() внутрь класса Figure. А также кладем 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().

**Листинг 7. 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   *# САПР* class CAD:  @classmethod  def draw\_all(cls, figures):  for figure in figures:  figure.draw() |

Воспользуемся этим примером.

**Листинг 8. 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.

Продолжим разбираться с OCP.

У нас в приложении появилось новое правило! Мы по каким-то причинам должны рисовать круги, а потом уже все остальное.

**Листинг 9. ocp\_6.py**

|  |
| --- |
| import abc   *# Абстрактная фигура* class Figure(abc.ABC):  @abc.abstractmethod  def draw(self):  pass   *# Треугольник* class Triangle(Figure):  def draw(self):  pass   *# решение упорядочиванием, уже лучше* class CAD:  @classmethod  def draw\_all(cls, figures):  for figure in figures:  if isinstance(figure, Circle):  figure.draw()   for figure in figures:  if isinstance(figure, Triangle):  figure.draw()   class Circle(Figure):  def draw(self):  pass   @staticmethod  def compare\_order(figure):  *# нарушение закрытости относительно других потомков Shape* return 0 if isinstance(figure, Circle) else 1 |

Идем по фигурам. Если это круг, то его рисуем, если треугольник, то его рисуем. Мы опять нарушили принцип – добавили в класс зависимость от фигуры. Если будем добавлять новый класс Romb, придется вносить изменения и в класс CAD.

Как предлагается сделать по-хорошему?

**Листинг 10. ocp\_7.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 CAD:   *# упорядоченная отрисовка* @classmethod  def draw\_all\_ordered(cls, figures):  clone = sorted(figures, key=FigureOrderComparator.compare\_order)  cls.draw\_all(clone)   class FigureOrderComparator:  @staticmethod  def compare\_order(figure):  return 0 if isinstance(figure, Circle) else 1 |

В CAD мы опять их рисуем, но рисуем отсортированными. И появляется еще класс FigureOrderComparator, который занимается их сортировкой.

И тогда мы сортируем фигуры:

clone = sorted(figures, key=FigureOrderComparator.compare\_order)

Рисуем:

cls.draw\_all(clone)

А класс FigureOrderComparator знает как сортировать. У него есть метод compare\_order(), который возвращает 0, если это круг. Иначе 1.

У нас здесь теперь и SRP соблюдается и нет никаких зависимостей. Могут приходить любые объекты с методом draw.

1. Принцип разделения интерфейса (The **I**nterface Segregation Principle).

«Клиенты не должны зависеть от тех методов, которые не используют»

*СЛАЙД 14*

**Листинг 11. 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)  # class Special(Order): # pass # # # class Special(models.Model): # # order = models.ForeignKey(order=True)* |

У нас есть класс Figure с двумя методами! Мы ее будем рисовать и печатать.

Есть класс Круг, мы его рисуем и печатаем.

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

def plot(self):  
 pass *# пустой метод, вынужденная реализация*Вот оно нарушение принципа разделения интерфейса.

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?

**Листинг 12. 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'**) |

Разделить класс Фигура 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)*

1. Принцип инверсии зависимостей (The **D**ependency Inversion Principle).

*СЛАЙД 15*

«Все части кода должны зависеть от абстракции и не зависеть от конкретики»

**Листинг 13. 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 |

Тут используется аннотация, что item должен быть SimpleItem.

Теперь делаем класс PerfectItem и все, передать его в Order мы уже не можем. Имеем зависимость от конкретики, что, конечно, не есть хорошо.

Как решить проблему?

**Листинг 13. 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 Order:  ...   *# добавить абстрактный товар, детали реализации товара не интересуют* def add(self, item: ItemInterface):  self.total += item.get\_price() |

Теперь мы ожидаем что-то, у чего есть метод get\_price(). Зависимости от конкретики больше нет.

Теперь мы знаем, видим в классах ссылки на конкретику (различные if’s) – это плохо и нужно выполнять рефакторинг.

1. Принцип подстановки Барбары Лисков (The **L**iskov Substitution Principle).

*СЛАЙД 16*

«Должна быть возможность заменить тип подтипом»

На пальцах! У нас есть абстрактный класс и от него что-то наследуется. Например, наследуется класс B. Мы создаем для него экземпляр.

b = B()

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

Это сложно проверять. Наследник должен расширять поведение родителя, а не изменять.

**Листинг 14. test\_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 = Square()  *# нарушение принципа LSP, мы не ожидаем тут, что и высота ТОЖЕ изменится* rectangle.set\_width(5)  rectangle.set\_height(4)  *# 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.

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.

Запуская пример, мы ожидаем, что получим 20.

И да, вот так

rectangle = Rectangle()

все работает

Но если мы сделаем вот так:

Но на деле тест оказывается проваленным, ведь получаем мы не 20, а 16.

Так получается, что когда мы на шаге

rectangle.set\_height(4)

меняем высоту, то меняется и ширина.

Как решить проблему?

**Листинг 15. test\_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)   class SquareTest(unittest.TestCase):  def test\_area(self):  rectangle = SquareImmutable(4)  *# тест пройден* self.assertEqual(rectangle.area, 16)   *# пытаемся изменить атрибуты* with self.assertRaises(AttributeError):  rectangle.width = 5   with self.assertRaises(AttributeError):  rectangle.width = 4 |

Прямоугольник и квадрат мы делаем неизменяемыми. В прямоугольнике две стороны, в квадрате – одна. Теперь значение мы передаем в init и на ходу свойство заменить не можем.

Пару слов про GRASP

*СЛАЙДЫ 17-18*

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

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

SOLID более популярные.

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.

*СЛАЙДЫ 20-24*

Теперь по ДЗ. Настраиваем шаблонизатор, переходим на базовые шаблоны.

Чтобы по шаблонам у нас все работало, нам нужны вот эти методы:

env.loader = FileSystemLoader(folder)  
*# находим шаблон в окружении*template = env.get\_template(template\_name)

def render(template\_name, folder=**'templates'**, \*\*kwargs):  
 env = Environment()  
 *# указываем папку для поиска шаблонов* env.loader = FileSystemLoader(folder)  
 *# находим шаблон в окружении* template = env.get\_template(template\_name)  
 return template.render(\*\*kwargs)

*ЗАДАНИЕ НЕБОЛЬШОЕ, ПОЭТОМУ ВЫ МОЖЕТЕ УСПЕТЬ ДОДЕЛАТЬ ФРЕЙМВОРК!!*