GeekBrains

Архитектура и шаблоны проектирования на Python

Принципы проектирования

© geekbrains.ru

На этом уроке

Принципы проектирования DRY, KISS, SOLID, GRASP.

Оглавление

Принципы проектирования

Проектирование программного обеспечения — процесс, посредством которого разработчик создаёт спецификацию предназначенного для достижения заявленных целей программного продукта, используя набор примитивных компонентов, имеющих заданные ограничения.

Общие принципы проектирования

- Разработчик должен рассмотреть альтернативные подходы, делая выводы о каждом из них на основе заявленных требований и доступных ресурсов.
- Проектирование должно соответствовать аналитической модели. В процессе декомпозиции сложная система делится на части, соответствие которых проектным требованиям легко проверить по нескольким критериям. В то же время необходимо следить за целостностью системы и её способностью удовлетворять заявленным требованиям.
- Не стоит изобретать велосипед. Система строится с использованием набора шаблонов проектирования — применять их выгоднее, чем создавать что-либо с нуля. Ограниченные время и ресурсы следует инвестировать в представление действительно новых идей на основе существующих шаблонов.
- Разработчик должен сокращать интеллектуальную дистанцию между программным продуктом и задачей как она существует в реальном мире. Иными словами, структура дизайна программного обеспечения должна по возможности имитировать структуру предметной области (эта тема рассмотрена на прошлом уроке).
- Дизайн должен демонстрировать единообразие и целостность. Для этого в процессе разработки команда должна следовать заранее оговоренным архитектурным принципам, лежащим в основе создаваемого приложения.
- Структура проектируемого продукта должна поддерживать внесение изменений.
- Следует различать проектирование и кодирование (реализацию проектных решений). Уровень абстракции проектных решений выше уровня реализации при кодировании. Проектирование — это стратегия, кодирование — тактика создания продукта.
- Проектные решения требуют независимой оценки для устранения концептуальных ошибок. Иногда разработчик сосредотачивается на мелочах и, как говорится, за деревьями леса не видит. Сначала проектная группа должна убедиться в корректности основных концептуальных элементов дизайна и уже потом переходить к проработке особенностей реализации.

Методы

- Абстракция.
- Декомпозиция.
- Модульность.
- Проектирование на основе предметной области (модель, основа коммуникаций с заинтересованными сторонами).
- Тестирование (избавляет от многих архитектурных проблем, т. к. требует создавать модульный слабо связанный код ещё на стадии разработки).

Принцип абстракции

«Каждая существенная область функциональности в программе должна быть реализована всего в одном месте программного кода. Если различные фрагменты кода реализуют аналогичную функциональность, то, как правило, имеет смысл слить их в один фрагмент, абстрагируя (abstracting out) различающиеся части».

Б. Пирс

«Абстракция выделяет существенные характеристики объекта, которые отличают его от всех других видов объектов и таким образом обеспечивают чётко определённые концептуальные границы относительно взгляда наблюдателя».

Г. Буч

«Абстракция в объектно-ориентированном программировании — это придание объекту характеристик, отличающих его от всех других объектов, чётко определяющих его концептуальные границы. Основная идея — отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов.

Рассмотрим пример. Предприятие оптовой торговли имеет дело с покупателями и продавцами, причём покупатель никогда не выступает в роли продавца, а продавец — в роли покупателя. Из этих двух типов следует выделить Абстракцию-контрагент и Абстракцию-вид контрагента — это сделает каждую новую полученную абстракцию более цельной и универсальной.

Любая проблема проектирования может быть решена введением дополнительного абстрактного слоя, за исключением проблемы слишком большого количества дополнительных абстрактных слоёв».

Неизвестный автор

Принцип DRY

Принцип DRY (Don’t Repeat Yourself) — «Не повторяйся». Его официальная формулировка в книге Эндрю Ханта и Дэвида Томаса «Программист-прагматик» звучит так: «Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы».

Если вы видите в коде схожие места, их можно выделить в отдельную функцию, класс, пакет и так далее. В дальнейшем при изменении внешних требований достаточно поменять код в одном месте, чтобы изменённый функционал стал доступен во всех использующих его частях программы.

Принцип KISS

Принцип KISS (Keep It Simple Stupid) — «Сделай проще». Он запрещает использование более сложных средств, чем те, что необходимы для решения задачи. Принцип декларирует основной целью и / или ценностью простоту системы. Наиболее близок KISS широко известный принцип бритвы Оккама: «Не следует множить сущее без необходимости».

Принципы SOLID

Пять принципов объектно-ориентированного проектирования были предложены Робертом Мартином в начале 2000-х годов. SOLID — акроним от сокращенных названий принципов ООП:

Инициал	Принцип	Название, понятие
S	SRP	Принцип единственной ответственности (The Single Responsibility Principle): «Существует лишь одна причина, приводящая к появлению класса»

O	OCP	Принцип открытости/закрытости (The Open Closed Principle): «Программные сущности должны быть открыты для расширения, но закрыты для модификации»
L	LSP	Принцип подстановки Лисков (The Liskov Substitution Principle): «Объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы»
I	ISP	Принцип разделения интерфейса (The Interface Segregation Principle): «Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения»
D	DIP	Принцип инверсии зависимостей (The Dependency Inversion Principle): «Зависимость на абстракциях. Нет зависимости на что-то конкретное»

Рассмотрим их более подробно.

Принцип единственной ответственности (SRP)

Этот принцип гласит, что каждый модуль или класс должен нести ответственность только за одну часть функциональности, предоставляемой программным обеспечением, и эта ответственность должна быть полностью инкапсулирована классом.

Существует лишь одна причина, приводящая к изменению класса.

Ответственность класса — это стержень, вокруг которого выстраиваются его методы. Если класс несёт две или более ответственности, то и стержней становится более одного. При изменении одной из ответственностей такой класс уже может не соответствовать второй. Наглядная аналогия — слуга двух господ Фигаро.

Пример. Есть объект Order (заказ), включающий в себя следующие методы:

```
class Order:
    def get_items(self):
        pass

    def get_total(self):
        pass

    def validate(self):
        pass

    def save(self):
        pass

    def load(self):
        pass
```

Здесь смешиваются бизнес-логика и персистентность (сохранение состояния). При изменении методов, отвечающих за персистентность, очень легко «задеть» бизнес-логику, что нежелательно, поскольку клиентские классы (использующие класс Order) не ожидают изменений. Приложение становится хрупким — изменения в одной его части нарушают работу других концептуально независимых частей. Правильным решением будет отделить логику от сохранения:

```
class Order:
```

```
    def get_items(self):
        pass

    def get_total(self):
        pass

    def validate(self):
        pass
```

и

```
class OrderRepository:
    def save(self):
        pass

    def load(self):
        pass
```

Однако не следует чрезмерно усердствовать в разделении обязанностей класса, чтобы не впасть в другую крайность, когда логика класса размывается между несколькими мелкими объектами.

Принцип открытости / закрытости (OCP)

Широко известный благодаря Р. Мартину принцип открытости / закрытости имеет два варианта:

Программные сущности (классы, модули, функции и т. п.) должны быть открытыми для расширения, но закрытыми для модификации.

Первоначальная идея принадлежит Бертрану Мейеру:

Модули должны быть одновременно и открытыми, и закрытыми.

Модуль будет считаться открытым, если он по-прежнему доступен для расширения. Например, должно быть возможно добавить поля к содержащимся в нём структурам данных или новые элементы к набору функций, которые он выполняет.

Модуль будет считаться закрытым, если он доступен для использования другими модулями. Это означает, что модулю (его интерфейсу в смысле сокрытия информации) дано чёткое, окончательное описание. Модуль можно компилировать, сохранить в библиотеке и сделать доступным для использования другими модулями.

Рассмотрим этот принцип на классическом примере. Допустим, мы проектируем САПР, которая имеет дело с различными фигурами. Каждая фигура представлена своим классом, и требуется некоторая функция для отрисовки набора фигур в пользовательском интерфейсе.

```
// 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
```

В чём проблема? Чтобы при необходимости добавить, допустим, квадрат (Square), нам придется изменить поведение метода CAD.draw_all при выборе поведения и реализовать отрисовку квадрата в отдельном методе этого класса. Строго говоря, класс CAD не соответствует принципу закрытости / открытости, т. к. не закрыт от расширения типов наследников Figure.

Что можно сделать? Надо абстрагировать отрисовку фигур. Сделать класс Figure абстрактным, объявив абстрактный метод draw():

```
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()
```

Новое требование к системе: круги нужно рисовать первыми, треугольники после них. Решение «в лоб» снова приведет нас к зависимости класса CAD от наследников Figure.

```
// 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()
```

Дополнительный уровень абстракции, абстрагирующий порядок отрисовки, помогает, но помещать его в иерархию Figure опрометчиво: мы нарушим закрытость наследников Figure относительно друг друга и принцип SRP относительно ответственности потомков Figure, заставляя их отвечать за порядок сортировки.

```
class Circle(Figure):
    ...
    @staticmethod
    def compare_order(figure):
        // нарушение закрытости относительно других потомков Shape
        return 0 if isinstance(figure, Circle) else 1
```

Идея правильная, однако место для абстрагирования выбрано неудачно. Вынесем абстракцию порядка за рамки иерархии Figure — объявим класс, отвечающий только за порядок сортировки.

```
class CAD:
    ...
  // упорядоченная отрисовка
```

```
    @classmethod
    def draw_all_ordered(cls, figures):
        clone = sorted(figures, key=FigureOrderComparator.compare_order)
        cls.draw_all(clone)
```

```
// компаратор фигур относительно порядка отрисовки
class FigureOrderComparator:
    @staticmethod
    def compare_order(figure):
        return 0 if isinstance(figure, Circle) else 1
```

Соблюдение принципа открытости / закрытости достигается применением дополнительного уровня абстракции и размещением его в открытой для расширения части модуля. Используя абстракции, невозможно создать такую систему, которая будет удовлетворять всем точкам зрения. Мастерство разработчика заключается в способности предугадать наиболее возможные из них и заложить эти абстракции в основу расширения системы.

Принцип подстановки Лисков (LSP)

Принцип предложен Барбарой Лисков в 1987 году:

«Пусть q(x) является свойством, верным относительно объектов x некоторого типа T.
Тогда q(y) также должно быть верным для объектов y типа S, где S является подтипом типа T».

Р. Мартин формулирует его так:

«Должна быть возможность вместо базового типа подставить любой его подтип».

Рассмотрим классический *пример*. В системе САПР есть класс прямоугольников:

```
// 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
```

Через какое-то время потребовался класс, представляющий квадраты. Математически квадрат — это частный случай прямоугольника, поэтому, наследуя прямоугольник, мы получаем квадрат. Так как у квадрата все стороны равны, мы перегружаем методы присвоения длин сторон.

```
// 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)
```

Теперь проверим реализацию Square на соответствие принципу LSP:

```
class SquareTest(unittest.TestCase):
    def test_area(self):
        rectangle = Square()
        // нарушение принципа LSP - не ожидаем, что высота ТОЖЕ изменится
        rectangle.set_width(5)
        rectangle.set_height(4)
        // тест провален, актуальное значение 16
        self.assertEqual(rectangle.area, 20)
```

Тест провален: реализация Square нарушает принцип LSP в методах установки длин сторон, так как совершенно неожиданно для прямоугольника квадрат модифицирует соседнее значение размера фигуры.

Варианты решения данной проблемы:

1. Проектирование по контракту, которое включает в себя предусловия и постусловия:
    a. Предусловия (то, что должно быть выполнено вызывающей стороной перед вызовом метода) не могут быть усилены в классе-наследнике.
    b. Постусловия (то, что гарантируется вызываемым методом) не могут быть ослаблены в классе-наследнике.

    В примере квадрат нарушает постусловия для прямоугольника, модифицируя одновременно два поля базового класса.

    К сожалению, Python не поддерживает программирование по контракту из коробки, но вы можете воспользоваться сторонней библиотекой, например, zope.interface.

2. Использование Immutable (неизменяемых) классов:

```
// 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, неизменяемый
class SquareImmutable(RectangleImmutable):
    def __init__(self, size):
        super().__init__(size, size)
```

Попробуем проверить реализацию Square на соответствие принципу LSP:

```
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
```

В таком случае принцип LSP будет соблюдён: отсутствует предусловие по зависимости / независимости изменения длин прямоугольника и квадрата, так как они назначаются один раз при инициализации.

3. Наследовать оба класса от какой-то более общей абстракции, развивая их основное отличие: все стороны квадрата равны, что неверно для прямоугольника.

Принцип разделения интерфейса (ISP)

Формулировка:

Клиенты не должны вынужденно зависеть от методов, которыми не пользуются.

В языке Python реализовано множественное наследование, поэтому механизм интерфейсов как таковой не имеет смысла. Вместо них будем создавать абстрактные классы.

Пример. Предположим, теперь нам надо выводить фигуры на печать. Естественно, мы добавим в абстрактный класс фигуры метод plot(), который конкретные фигуры будут реализовывать.

```
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')
```

Рано или поздно появятся фигуры, которые не выводятся на печать, но присутствуют на экране: оси, направляющие и так далее. Естественно, их реализация будет пустой:

```
// направляющая, непечатная
class GuideLine(Figure):
    def draw(self):
        print('draw Circle')

    def plot(self):
        pass // пустой метод, вынужденная реализация
```

В соответствии с принципом ISP из интерфейса фигуры методы, относящиеся к печати, должны быть выделены в отдельный интерфейс, который будут реализовывать только те типы фигур, которым это необходимо:

```
// 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')
```

Глядя только на класс, мы можем судить лишь о том, соблюдает ли он принцип SRP. Относительно принципа ISP мы такого сказать не можем – необходимо знать контекст использования класса, как его используют классы-клиенты. Если они интересуются разными аспектами анализируемого класса, стоит подумать о более тонкой сегрегации его интерфейсов.

Принцип инверсии зависимостей (DIP)

Представим себе достаточно крупное приложение с множеством классов. При естественном порядке построения сперва мы опишем базовые низкоуровневые сущности, затем перейдём к сущностям более высокого порядка, которые будут определены в терминах низкоуровневых сущностей. Этот путь кажется логичным, но приводит к «жёсткому» дизайну: одни компоненты приложения «впадают» в сильную зависимость от других.

Формулировка:

Модули верхних уровней не должны зависеть от модулей нижних уровней.
Оба типа модулей должны зависеть от абстракций.

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

Пример. Торговое приложение B2B. Оформляем заказы.

```
// 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()
```

В какой-то момент бизнес-правила меняются, и теперь сущность товара представляет другой класс.

```
// превосходный  товар
class PerfectItem:
    def get_price(self):
        pass
```

В статически типизируемом языке программирования может возникнуть проблема: мы не можем добавлять экземпляры PerfectItem в имеющийся класс Order, так как последний сильно зависит от старого класса SimpleItem. Мы нарушили принцип инверсии зависимостей — заказ зависит от товара. Для языка Python мы сознательно сымитировали такую ситуацию. Перепроектируем решение в соответствии с принципом. Выделим абстракцию в интерфейс, и уже от неё будут зависеть классы обоих уровней.

```
// 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):
        self.total += item.get_price()
```

Принцип инверсии зависимостей — это фундаментальный низкоуровневый механизм, лежащий в основе многих преимуществ, которые обещают объектно-ориентированные технологии. Его надлежащее применение необходимо для создания повторно используемых каркасов.

Принципы / шаблоны GRASP

Принципы / шаблоны GRASP были предложены Крейгом Ларманом в 1998 году. Двойственность названия сам автор отсылает к «отцам-основателям»:

«Шаблон одного разработчика — это простейший строительный блок другого».

Принципы / шаблоны GRASP представляют отличающийся от SOLID взгляд на проблемы проектирования программного обеспечения, но их автор приходит к выводам, аналогичным SOLID-принципам. Рассмотрим их более подробно.

Creator // Создатель

Сущности создаются, меняются, уничтожаются. С последними двумя действиями всё понятно — есть то, что требуется модифицировать или удалить. Но кто или что должно создавать сущность? В управляемых языках мы можем воспользоваться **new**, а дальше? Не хотелось бы, например, чтобы одна строка заказа оказалась в двух разных заказах. Итак, мы пришли к необходимости назначения ответственности за создание сущностей класса некоторому конкретному классу.

Согласно шаблону Creator, этот класс C должен удовлетворять одному из критериев:

- Класс C содержит или агрегирует объекты A.
- Класс C записывает экземпляры объектов A.
- Класс C активно использует объекты A.
- Класс C обладает данными инициализации, которые будут передаваться объектам A при их создании.

Например, согласно первому критерию, заказ (Order) — хороший претендент на роль Создателя объектов типа строки заказа (OrderItem).

Information Expert // Информационный эксперт

Как распределить ответственность между классами системы?

Ответственность должна быть назначена тому, кто владеет максимумом необходимой информации для исполнения, — информационному эксперту.

Например, заказ (Order) состоит из строк заказа (OrderItem), содержащих ссылку на товар (Item) и количество (поле qty). Согласно этому принципу, обязанность по вычислению общей стоимости заказа следует возложить на класс заказа (Order), так как он обладает полнотой информации о составе заказа — строки OrderItem. При вычислении суммы по всем строкам заказа объект класса Order вынужден обратиться к информационному эксперту стоимости строки заказа — к объектам класса OrderItem, а те, в свою очередь, — к объектам товаров Item, которые будут информационными экспертами относительно цены товара.

Low Coupling // Низкая связанность

Степень связанности — это мера, определяющая, насколько сильно один элемент связан с другими, либо каким количеством данных о них он обладает. Использование класса с высокой связанностью приводит к таким проблемам:

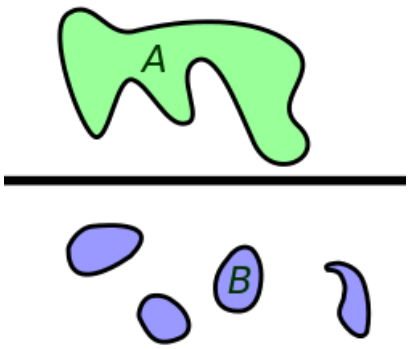
- Нестабильность связанных классов приводит к нестабильности в данном классе.
- Усложняется повторное использование класса с высокой связанностью.

Другая крайность — нулевая связанность классов, что также нежелательно, т. к. система распадается на множество независимых классов со сложной реализацией. Система, возможно, будет устойчива к изменениям, но сложность каждого из её компонентов повысится.

Таким образом, необходимо найти баланс степени связанности, позволяющий классам совместно выполнять функции системы.

High Cohesion // Высокая связность (Высокое зацепление)

Связность — мера силы взаимосвязанности обязанностей класса. Обязанности класса с высокой степенью связности связаны между собой, и наоборот: класс со множеством разнообразных не связанных между собой обязанностей имеет слабую связность.



Множество A связно, множество B — несвязно.
Источник: «Википедия», статья «Связное пространство».

Применение классов с низкой связностью чревато следующими последствиями:

- Трудности понимания.
- Сложности при повторном использовании.
- Сложности поддержки.
- Низкая надёжность.

Controller // Контроллер

Контроллер — это объект, отвечающий на запросы пользователей системы, но не являющийся интерфейсным. Контроллер имеет представление о системе в целом и делегирует обязанности по выполнению запросов конкретным исполнителям.

Типичный пример — шаблон MVC, в котором контроллер, получая запросы пользователя, делегирует их тем или иным методам манипулирования моделью.

Polymorphism // Полиморфизм

При реализации похожих, но различных по особенностям реализации действий, следует применять полиморфные свойства наследования классов, а не условную логику проверки типа классов. Выделение общего интерфейса позволяет однотипно обрабатывать объекты с различной реализацией.

Protected variations // Устойчивость к изменениям

Шаблон Protected variations позволяет защитить одни объекты от изменений в других, связанных с ними. Это достигается путём обёртки различных реализаций в один неизменный интерфейс. Таким образом мы получаем стабильность интерфейса и свободу в его реализации.

Indirection // Посредник

Для уменьшения связанности между элементами системы используется промежуточный объект — *Посредник*. Пример — шаблон Controller, рассмотренный выше. Controller «развязывает» объекты пользовательского интерфейса и модель. Другой пример использования шаблона *Посредник* — шаблон *Адаптер*. Объект *Адаптер* защищает класс *Клиент* от изменений во внешней системе.

Pure Fabrication // Чистая синтетика (Чистая выдумка)

Классы в модели предметной области отражают сущности реального мира, но зачастую построение системы, удовлетворяющей принципам Low Coupling / High Cohesion только на основе сущностей реального мира, невозможно. В таком случае проектировщик использует объекты *Чистая выдумка*, не имеющие прямого отображения в реальном мире, но позволяющие добиться соблюдения принципов Low Coupling / High Cohesion при реализации системы. Например, мы имеем объект заказ (Order), который надо каким-то образом сохранить, например, в БД. Включение логики сохранения заказа в БД в объект заказа неминуемо снизит его уровень связности, т. к. объект будет выполнять уже две мало связанные между собой обязанности. В то же время сохранение в БД потребуется для множества различных типов объектов, это достаточно рутинная операция в целом. Таким образом мы подошли к созданию некоторого объекта, отвечающего за сохранение объектов сущностей предметной области в БД. Он и будет «чистой выдумкой». Одна из широко известных реализаций этой задачи — ORM.

Принцип — не догма

Принципы, рассмотренные выше, называются принципами, потому что представляют некоторые убеждения, как стоит или не стоит проектировать программное обеспечение.

Вы можете им следовать, получая преимущества, о которых заботились их создатели, но, возможно, в конкретных условиях слепое следование какому-то принципу (или нескольким) не будет оправданно.

Как разработчик, вы можете (и это будет полезно) рассмотреть при решении вопросов проектирования несколько вариантов, каждый из которых имеет плюсы и минусы. Какое именно решение возобладает, во многом зависит от конкретных условий и целей.

Создание WSGI-фреймворка. Использование базовых и включённых шаблонов

Выше мы рассмотрели принцип DRY, в котором нам предлагается не повторяться. Дублирование касается любого кода не только на Python, но и на JavaScript и HTML. В фреймворке Django у нас была возможность убирать дублирование HTML-кода с помощью механизма наследования шаблонов и механизма включённых шаблонов.

Так как мы используем шаблонизатор Jinja2, в нём тоже есть такие возможности. Основные трудности могут возникнуть при поиске данных шаблонов в нашем проекте. Рассмотрим пример настройки шаблонизатора для поиска шаблонов в файловой системе:

```
from jinja2 import FileSystemLoader
from jinja2.environment import Environment


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)
```

Практическое задание

В этой самостоятельной работе тренируем умения:

1. Работать с шаблонизатором.
2. Использовать базовые и включённые шаблоны.

Зачем:

Чтобы избегать дублирования в шаблонах.

Последовательность действий:

1. Внести в WSGI-фреймворк изменения, которые позволят использовать механизм наследования и включения шаблонов.
2. Создать базовый шаблон для всех страниц сайта.
3. Если нужно, создать один или несколько включённых шаблонов.
4. Добавить на сайт меню, которое будет отображаться на всех страницах.
5. Улучшить существующие страницы с использованием базовых и включённых шаблонов.
6. Проверить, что фреймворк готов для дальнейшего использования (при желании добавить какой-либо полезный функционал).

Дополнительные материалы

1. SOLID is OOP for Dummies.
2. Принципы и паттерны проектирования.

Используемая литература

1. Пирс Б. Типы в языках программирования. М.: Лямбда пресс, Добросвет, 2011.
2. Буч Г. и др. Объектно-ориентированный анализ и проектирование с примерами приложений. М.: Вильямс, 2008.
3. The Principles of OOD.
4. Мартин Р. и др. Быстрая разработка программ. Принципы, Примеры, Практика. М.: Вильямс, 2004.
5. Мартин Р. Принципы, паттерны и методики гибкой разработки на языке C#. СПб.: Символ-Плюс, 2011.
6. Мейер Б. Объектно-ориентированное конструирование программных систем. М.: Русская Редакция, 2005.
7. Ларман К. Применение UML 2.0 и шаблонов проектирования. М.: Вильямс, 2013.
8. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns. MA.: Addison-Wesley, 1995.
9. Class diagram.
10. Sequence diagram.
11. Eiffel — Википедия.
12. Programming stuff. Блог Сергея Теплякова.
13. Иерархия принципов проектирования, или самые важные слова для инженеров.
14. Программа для работы с UML SoftwareIdeas.
15. Программа для работы с UML Bouml.