Паттерны ООП на Python для разработки приложения

Цель занятия

После освоения темы вы:

- познакомитесь с принципами проектирования;
- поймете суть SOLID-принципов проектирования;
- познакомитесь с основными паттернами ООП;
- узнаете, в каких ситуациях применять паттерны ООП;
- научитесь использовать паттерны, стандартизировать код;
- научитесь проектировать системы с использованием паттернов.

План занятия

Используемые термины

Паттерны — шаблоны проектирования.

Абстрактная фабрика (abstract factory) — это объект, который предоставляет интерфейс для создания семейства взаимосвязанных объектов.

Построитель (builder) — паттерн, который предназначен для построения из отдельных объектов сложной структуры.

Фабричный метод (factory method) — это метод, который позволяет разным объектам создавать единый интерфейс для создания связанного объекта.

Пул объектов (object pool) — паттерн, который предоставляет интерфейс для получения готовых к использованию объектов из заранее инициализированного набор.

Адаптер — паттерн, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Фасад — объект, обеспечивающий единую точку входа для взаимодействия с различными объектами некоторой подсистемы.

Мост — паттерн, который позволяет отделить абстрактный алгоритм от конкретной реализации отдельных операций.

Декоратор — паттерн, который используется для динамического подключения дополнительного поведения к объекту.

Приспособленец (flyweight — «легковесный элемент») — объект, минимизирующий потребление ресурсов за счет разделения некоторых данных с другими аналогичными объектами.

Итератор — это объект, реализующий метод `__next__`.

Команда — паттерн, который позволяет хранить информацию о выполняемых действиях в виде объектов.

Состояние — паттерн, который используется, когда поведение объекта должно зависеть от его состояния.

Наблюдатель (Observer) — паттерн, который позволяет объекту получать оповещения об изменении состояния других объектов.

Стратегия — паттерн, который позволяет реализовать набор взаимозаменяемых алгоритмов, из которых пользователь выбирает тот, который ему нужен.

Посетитель — паттерн, который используется, когда нужно применить одну и ту же логику к серии объектов.

Конспект занятия

1. Качество кода

Для чего нужно изучить темы этой недели:

- При создании крупных программ сложность задач быстро увеличивается. Для человека уже становится невозможным написать код «как-нибудь». Нужны определенные принципы и правила организации кода.
- Для эффективного решения усложняющихся задач необходимо следование определенным принципам.
- SOLID-принципы и паттерны проектирования — это систематизация многолетнего опыта программистов.

Что такое качественный код

Вспомним базовые принципы качественного кода:

- Если программа работает и выдает правильный результат, это еще не значит, что она написана хорошо. В некоторых случаях это можно считать вопросом исключительно эстетики. Но в более сложных ситуациях, когда программа большая, качество кода становится приоритетным.
- Качество кода важно для человека.
- Компьютер может выполнить любой код, если он корректен, как бы плохо он ни был написан. Читаемость кода чрезвычайно важна в длительных проектах, когда код приходится по многу раз доделывать и переделывать. Если код написан плохо, в некоторых случаях проще от него избавиться и написать заново, чем пытаться что-либо «доделать».

Базовые принципы написания качественного кода

Следование хорошему стилю (для Python – PEP8). Пример нехорошего стиля:

```
import numpy as q
O=q.array([(1,2,3),(4,5,6),(7,8,9)])
l=O+1 * 2
```

Ясность намерений. Например, в такой функции ничего не понятно:

```
def sd(a):
     b = a // 86400
     return b
```

Зачем ее написали? На самом деле здесь из количества секунд получают количество дней.

DRY (don't repeat yourself) — не повторяйся. Код должен быть «сухим». Если он содержит повторы, его нужно «высушить». Например, код рисования картинки:

```
#left ear
polygon(screen, 'grey',
    [(x[0]-r-3/32*r+r/16,x[1]-r/4-r/8-r/4+r/16-r/16),
    (x[0]-r-3/32*r-r/5+r/16,x[1]-r/4-r/8-r/4+r/16+r/10-r/16-r/32),
    (x[0]-r-3/32*r-r/5+r/5/10+r/16,x[1]-r/4-r/8-r/4+r/16+r/10-r/6-r/16)],
    0)
#right ear
polygon(screen, 'grey',
    [(x[0]-r-(-3/32*r+r/16),x[1]-r/4-r/8-r/4+r/16-r/16),
    (x[0]-r-(-3/32*r-r/5+r/16),x[1]-r/4-r/8-r/4+r/16+r/10-r/16-r/32),
    (x[0]-r-r/10-(-3/32*r-r/5+r/5/10+r/16),x[1]-r/4-r/8-r/4+r/16+r/
    10-r/6-r/16)],0)
```

Чисто алгебраически эти выражения можно упростить. Очевидно, что и первый, и второй кусок делают почти одно и то же. Можно обойтись без этого повтора и задать какую-то функцию.

Способы «высушивания» кода: использование списков, написание функций, классов, библиотек, новых языков программирования (например, создать язык, на котором эта задача будет легко решаться — domain specific language) и т. д.

Документация:

- говорящие имена объектов;
- аннотации типов;
- документ-строки;
- информативные сообщения коммитов;
- комментарии в коде (редко).

Важно! Документация должна отвечать на вопрос «Зачем?», а не «Что?» и «Как?».

Что делает та или иная функция, и как она это делает, понятно из самого кода. Важна не только техническая документация, но и сам код должен быть самодокументирующимся.

Базовых принципов недостаточно
Далее появляются вопросы архитектуры — устройства нашей программы.

Когда задачи становятся сложнее, нам нужно думать:

- о разбиении большой задачи на более мелкие компоненты;
- разбиении логики на отдельные функции;
- взаимодействии этих компонентов.

Желательно при этом сделать компоненты относительно независимыми друг от друга, чтобы один и тот же компонент можно было использовать в разных ситуациях. И, наоборот, чтобы при необходимости можно было заменить какой-либо компонент другой версией.

2. SOLID-принципы

В статье Robert Martin. Design Principles and Design Patterns (2000) вводятся несколько принципов разработки ПО. Пять из них стали известны под аббревиатурой SOLID благодаря Майклу Физерсу.

Эти принципы подвергаются критике, отчасти справедливой:

- Недостаточная систематичность — почему именно эти 5 признаков.
- Некоторые из них сформулированы недостаточно конкретно и ясно, и часто обнаруживается их неправильное понимание и использование.

Но в целом принципы достаточно полезны, рассмотрим их подробнее.

4 признака «гниющего» кода

1. **Жесткость** (rigidity) — внесение даже небольших изменений в проект требует большого объема работы.

2. **Хрупкость** (fragility) — изменения в одной части проекта порождают проблемы в других частях, никак концептуально не связанных с первой.
3. **Неподвижность** (immobility) — невозможно использовать одни и те же компоненты в разных проектах.
4. **Вязкость** (viscosity) — при внесении изменений трудно сохранять изначальные принципы проектирования.

SOLID-принципы предлагаются в качестве рецепта для избежания «гниющего» кода:

- **S**ingle-responsibility principle.
- **O**pen–closed principle.
- **L**iskov substitution principle.
- **I**nterface segregation principle.
- **D**ependency inversion principle.

Разберем все принципы по очереди.

Single-responsibility principle

Принцип единственной ответственности. Один из самых неясных принципов.

«Модуль должен иметь одну и только одну причину для изменения. Пользователи и заинтересованные лица и есть та самая причина для изменений» (Чистая архитектура, глава 7).

Самый частый пример. Функция, которая что-то загружает из базы данных, потом что-то рассчитывает и записывает результат в какой-то файл. Если поменяется база данных, нам нужно поменять что-то в этой функции. Если поменяется алгоритм расчета, нужно опять лезть в эту функцию. Если хотим сохранять в каком-то другом формате, то тоже придется менять функцию. Но должна быть одна причина для изменения. В данном же случае мы легко видим три компонента функции, и они мысленно легко разделяются.

Пример нарушения принципа SRP:

```
def settings_menu(beginning_flag):
    """Обновляет состояния настроек и музыки"""
```

```
    if settings.on:
        fighting.stop()
        if beginning_flag == True:
            start.play(-1)
                beginning_flag = False
        menu.settings = False
        settings.draw()
        for event in events:
            if event.type == pygame.MOUSEBUTTONDOWN:
                pos = pygame.mouse.get_pos()
                if settings.volume.button_rect.collidepoint(pos):
                    settings.volume.hit = True
            if event.type == pygame.MOUSEBUTTONUP:
                settings.volume.hit = False
        start.set_volume(settings.volume.start_val/100)
        fighting.set_volume(settings.volume.start_val/100)
        walk.set_volume(settings.volume.start_val/100)
        settings.check_events()

    if not settings.on:
        menu.on = True
    return beginning_flag
```

Непонятно, что делает эта функция. Уже из документации функции явно видно, что что-то не так. Функция обновляет состояние настроек и музыки. Какая связь между настройками и музыкой?

В коде функции мы видим проверку события нажатия на клавиши мыши. Это тоже никак не связано с обновлением настроек и музыки.

Как разделить эту функцию на отдельные компоненты? Это невозможно, так как изначально все перемешано. Придется просто переписывать программу заново. Действия в этой функции присутствуют и в других функциях.

Важно! Любая причина изменения должна затрагивать один объект или несколько тесно связанных друг с другом. Когда мы думаем, как выделить компоненты в программе, нужно задать вопрос, а кого стоит «уволить», если в программе что-то пойдет не так.

Open–closed principle

Принцип открытости-закрытости. Это главный принцип, из которого следует все остальное.

«Следует писать модули так, чтобы их функциональность могла быть расширена без необходимости внесения изменений в их код».

Пример 1 — композиция объектов. У нас есть какая-то игра, у которой есть графический движок:

```
class Game:
    def __init__(self):
        self.graphic_engine = GraphicEngine()
        ...
```

Лучше сделать так, чтобы объект уже получал на вход графический движок и просто его сохранял. Тогда мы легко можем взять другой графический движок без необходимости изменения этого кода. Мы просто передаем другой параметр:

```
class Game:
    def __init__(self, graphic_engine):
        self.graphic_engine = graphic_engine
        ...
```

Возникает вопрос, что в игре есть не только графический движок, но и другие объекты. Когда конструктор принимает на вход 15 параметров, это не очень хорошее решение, так как делает систему достаточно хрупкой. Обсудим это чуть позже.

Пример 2. Функция, которая считывает целое число с клавиатуры и производит некоторую валидацию:

```
def read_int(prompt='', validator: Callable[[int], None] =
None):
    """
    validator - функция, выполняющая проверку корректности
    введенного значения, должна возбудить ошибку ValidationError
    с понятным для  пользователя сообщением или вернуть None.
    """
    while True:
        try:
            value = int(input(prompt))
            if validator is not None:
                validator(value)
```

```
        except ValueError:
            print('Некорректное число’)
        except ValidationError as e:
            print(e)
        else:
            return value
```

Это тоже пример выполнения функцией принципа открытости-закрытости. Эту функцию легко использовать в новом контексте, расширить ее функциональность за счет написания нового валидатора. При этом саму функцию трогать мы не должны. Можем просто спокойно импортировать ее в готовом виде из модуля и даже не думать о том, как она устроена.

Пример 3. Есть консольная программа, которая получает от пользователя сообщения и что-то с ними делает:

```
while True:
    try:
        msg = input('Введите сообщение: ').lower()
    except EOFError:
        print('До свидания’)
        break
```

Довольно простой цикл. Что с ним можно сделать? Допустим, мы хотим перевести эту программу в другой язык — чтобы она общалась и на русском, и на английском языке. Придется много поправить в коде.

Лучше сделать так:

```
PROMPT = 'Введите сообщение: ‘
GOOD_BYE_MSG = 'До свидания’

while True:
    try:
        msg = input(PROMPT).lower()
    except EOFError:
        print(GOOD_BYE_MSG)
        break
```

Настраиваемые данные вынести в отдельную часть или загрузить их из текстового файла, чтобы вообще не трогать программу. В настройках указать другой язык, и тогда функция загрузит из другого места.

Разделение данных и кода, с которым данные работают, достаточно полезный прием. Он позволяет удобнее и проще настраивать в дальнейшем программу.

Liskov substitution principle

Принцип подстановки Лисков.

«Функции, использующие базовый класс, должны быть в состоянии работать с объектами подклассов, даже не зная об этом».

Рассмотрим пример, как этот принцип может нарушаться и как сделать правильно.

Пример нарушения. Нам нужно создать список с четко ограниченным размером:

```
class LimitedList(list):
     def __init__(self, size, *args):
         super().__init__(*args)
         self.size = size

     def append(self, value):
         if len(self) >= self.size:
           raise OverflowError('size limit exceeded’)
       super().append(value)
...
def rotate(a: list):
     a.append(a[0])
     a.pop(0)
```

Это плохая идея — использовать в Python наследование от других классов. При переопределении поведения базового класса он может не измениться. Встроенные классы в Python работают не совсем так, как пользовательские классы. Некоторые методы могут не вызываться в некоторых классах, когда один метод заданного объекта опирается на другой метод этого объекта. Например, метод `get` в словаре в Python работает по-старому, его нужно переопределить. Метод `append` не позволяет добавить в список элементов больше, чем мы задали ограничение. Если ограничение не выполняется, то генерируется ошибка.

Функция в примере LSP делает циклический сдвиг — берет первый элемент списка и переставляет его в конец. И функция «не знает», что список имеет измененную логику, что он имеет максимальный размер. И тогда `append` вызовет ошибку. Хотя по существу никакой ошибки не должно быть. Мы не увеличиваем размер списка, просто переставляем элементы с места на место.

Как сделать правильно:

```
class LimitedList:
     def __init__(self, size, *args):
         self._container = list(*args)
         self.size = size
      def append(self, value):
          if len(self) >= self.size:
              raise OverflowError('size limit exceeded’)
          self._container.append(value)
```

Не использовать наследование, а использовать композицию. Создать обычный список в некотором поле и реализовать собственный метод `append`. Вместо вызова функции `super` и вызова метода базового класса вызываем метод `container`. Теперь проблема решена.

Наследование стоит использовать там, где оно изначально задумано.

Interface segregation principle

Принцип разделения интерфейса.

«Клиенты не должны зависеть от интерфейсов, которые они не используют».

Пример работы принципа ISP. Есть абстрактный класс базы данных, который умеет подключаться к базе данных, авторизоваться, получать объекты и добавлять объекты в базу данных:

```
from abc import ABC, abstractmethod

class DataBase(ABC):
     @abstractmethod
     def connect(self, address):
         ...
```

```
    @abstractmethod
    def authorize(self, login, password):
        ...

    @abstractmethod
    def get(self, id):
        ...

    @abstractmethod
    def add(self, obj):
        ...
```

Теперь объявляем конкретный класс, реализующий этот интерфейс. Этот класс представляет локальное хранилище. Нам к нему не надо ни подключаться, ни авторизовываться:

```
class LocalStorage(Database):
    def connect(self, address):
        pass # We don't need this

    def authorize(self, login, password):
        pass # We don't need this

    def get(self, id):
        # реализация

    def add(self, obj):
        # реализация
```

Мы вынуждены писать бессмысленную «реализацию» методов, которые нам не нужны. Метод, в теле которого просто `pass`, вызывает вопрос, зачем он написан.

Требуется комментарий, зачем мы это сделали.

Это требуется интерфейсом. Абстрактный класс не даст создать «наследника», который не переопределил эти абстрактные методы.

Ситуация, когда мы вынуждены писать какие-то «заглушки» в местах, которыми не пользуемся, это явное нарушение принципа разделения интерфейса.

Как сделать правильно:

```
class DataBase(ABC):
    @abstractmethod
    def get(self, id): ...

    @abstractmethod
    def add(self, obj): ...

class RemoteDataBase(DataBase):
    @abstractmethod
    def connect(self, address): ...

    @abstractmethod
    def authorize(self, login, password): ...
```

Нужно сделать абстрактный класс базы данных, который умеет только получать и добавлять объекты. И сделать подкласс удаленной базы данных, к которой нужно подключиться и авторизоваться. Тогда проблема будет решена, интерфейс будет четко разделен на два компонента. Мы можем использовать только один из них, не переживая за другой.

Как понять, что интерфейс нужно разделить на компоненты? Из практики. Если возникает необходимость использовать только часть интерфейса, а другую часть заменить «заглушкой», значит, нужно разделение.

Dependency inversion principle

Принцип инверсии зависимостей.

«Зависеть можно от абстракции, а не от конкретной реализации».

Пример. Есть функция, которая по id пользователя возвращает его e-mail.

Плохой вариант:

```
def get_user_email(user_id):
    db = RedisDataBase('redis://127.0.0.1:6379/0’)
    user = db.get(user_id)
```

```
    return user.email
```

Функция подключается к базе данных, получает пользователя, возвращает его e-mail. Чем этот вариант плох? Эта функция зависит от конкретной реализации базы данных. Нужно ли ей знать, что база данных – это именно база данных Redis на заданном адресе? Ей достаточно, чтоб была какая-то база данных, из которой можно получить пользователя.

Вариант еще хуже:

```
def get_user_email(user_id, db_type):
    if db_type == 'redis':
        db = RedisDataBase('redis://127.0.0.1:6379/0')
    elif db_type == ...:
        db = ...
        ...
    user = db.get(user_id)
    return user.email
```

Во-первых, плохое решение в качестве типа объекта указывать просто строку. В тексте легко сделать опечатку, и никакая система об этом не сообщит. При названии объекта с опечаткой будет явная ошибка, мы это увидим.

Во-вторых, в данном варианте приходится писать довольно много кода. Хотя эта функция всего лишь должна найти пользователя и сообщить его e-mail, она зачем-то должна знать много подробностей о разных типах баз данных, об их адресах, подключениях.

Правильно передать готовый объект базы данных:

```
def get_user_email(user_id, db):
    user = db.get(user_id)
    return user.email
```

Здесь возникает необходимость принципа инверсии зависимости. Здесь недостаточно ясно, что такое `db`. Нужна абстракция, которая это прояснит.

Лучше всего такие типы указывать как протоколы:

```
from typing import Protocol

from user import User

class UserRepository(Protocol):
    def get(self, id: int) -> User:
     ...
def get_user_email(user_id, db: UserRepository):
    user = db.get(user_id)
    return user.email
```

Это инверсия зависимостей, потому что изначально наша функция зависела от конкретной реализации базы данных. Теперь она зависит только от абстракции. Но от этой же абстракции зависит и конкретная реализация. Раньше зависимость шла в одну сторону — от функции к реализации базы данных. Теперь появились две зависимости, которые «идут» навстречу друг другу, к одной и той же абстракции.

3. Порождающие паттерны проектирования

В книге Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования» (1994) обобщен некоторый опыт объектно-ориентированного проектирования в виде набора стандартных шаблонов, которые часто применяются. Эту работу также часто критикуют из-за недостатка систематичности. Почему именно такие наборы паттернов, достаточно ли их. Действительно, их недостаточно, есть и другие. Тем не менее не бесполезно с этим опытом познакомиться.

Книга Кристофера Александера «Язык шаблонов» (1977) не о программировании, а об архитектуре. Описывает все возможные архитектурные решения, как комбинацию стандартных шаблонных решений.

Многие названия паттернов напоминают архитектурные термины — абстрактная фабрика, фабричный метод, строитель, мост, фасад, декоратор.

Виды паттернов проектирования

- **Порождающие паттерны** описывают создание объектов.
- **Структурные паттерны** описывают компоновку объектов.

- **Поведенческие паттерны** описывают взаимодействие объектов.

Абстрактная фабрика (abstract factory)

Это объект, который предоставляет интерфейс для создания семейства взаимосвязанных объектов.

Зачастую такие фабрики используются при построении графических интерфейсов:

```
def generate_ui(ui_factory):
  main = ui_factory.get_main_widget()
  label = ui_factory.get_label(text="Hello world")
  button = ui_factory.get_button(text="Click me!")
  main.add(label)
  main.add(button)
  return main
```

При необходимости изменения интерфейса мы просто меняем эту абстрактную фабрику или создаем новую, но код остается прежним.

Построитель (builder)

Похож на абстрактную фабрику, но предназначен не для получения отдельных объектов, а для построения из них сложной структуры:

```
def generate_ui(ui_builder):
   label = ui_builder.add_label(text="Hello world")
   button = ui_builder.add_button(text="Click me!")
   ...
```

Если есть построитель, мы просто добавляем в него текстовое поле.

Эти паттерны могут использоваться не только для построения графических интерфейсов. Иногда возникает необходимость передать в качестве параметров в какой-то конструктор очень много объектов. Создавать их заранее, потом передавать 15 аргументов не самое лучшее решение. Более подходящее решение — передать в конструктор фабрику или построитель, которые уже можно отдельно настроить.

Фабричный метод (factory method)

Это метод, который позволяет разным объектам (подклассам одного и того же суперкласса) создавать связанные с ними объекты разных типов. Предоставляет единый интерфейс для создания связанного объекта.

Рассмотрим абстрактную операцию, которая относится к какой-то категории:

```
class Operation(ABC):
   @abstractmethod
   def get_category(self): ...

class Expense(Operation):
   def get_category(self):
      return ExpenseCategory(self.category)

class Income(Operation):
   def get_category(self):
      return IncomeCategory(self.category)
```

Одна и та же функция может работать с разными объектами, пользуясь этим методом одинаково.

Пул объектов (object pool)

Предоставляет интерфейс для получения готовых к использованию объектов из заранее инициализированного набора. Это бывает полезно, если создание новых объектов ресурсоёмко.

Например, мы можем поддерживать пул соединений для того, чтобы заново их использовать, не открывая каждый раз новое:

```
class ConnectionsPool:
   async def get_connection(self):
      await self.wait_for_free_connection()
      con = self._free_connections.pop()
      con.free = False
      return con
class Connection:
   def __init__(self, pool):
      self._pool = pool
   def close(self):
```

```
        self.free = True
        self._pool.add(self)
```

Этот паттерн часто используется при программировании игр.

Пул объектов требует написания не самого простого кода, чтобы иметь гарантию, что все будет работать правильно. Что при получении объекта мы обязательно его возвращаем в пул, когда он нам больше не нужен, и что он возвращается в правильном состоянии, не храня в себе какие-то следы его использования. Поэтому нужно предусмотреть взаимодействие самого пула объектов с самими объектами либо в самом пуле, либо в объекте.

Это непростой паттерн, и нужно оценивать, насколько оправдано его написание.

Общая идея порождающих паттернов в том, что при создании объектов мы создаем какую-то дополнительную логику, которая позволяет нам лучше контролировать и настраивать процесс. Эти паттерны полезны при работе со сложными структурами, где много разных объектов.

4. Структурные паттерны

Начнем с одного из самых простых и очевидных структурных паттернов.

Адаптер

Позволяет объектам с несовместимыми интерфейсами работать вместе. Это «переходник» между одним интерфейсом и другим.

Предположим, что у нас есть класс с каким-то репозиторием, который умеет читать данные. Есть функция, которая пользуется репозиторием, чтобы прочитать из него данные, но у них несовместимый интерфейс:

```
class SomeRepository:
    def read(self):
        ...
def get_from_repository(repo):
    repo.get()
        ...
```

Возможно, что и класс, и функция написаны не нами, и уже используются во многих местах проекта, то есть переписать их мы не можем. Нужно создать адаптер:

```
class SomeRepository:
    def read(self):
        ...
class SomeRepoAdapter:
    def get(self):
        return self.repo.read()
def get_from_repository(repo):
    repo.get()
        ...
```

Адаптер обратится к исходному классу, вызовет у него нужный метод и сделает это под другим названием.

Довольно простая структура, часто применяется на практике.

Фасад
Это объект, обеспечивающий единую точку входа для взаимодействия с различными объектами некоторой подсистемы.

Зачастую у нас есть набор объектов, каждый из которых реализует свой компонент логики, но для выполнения какой-то конкретной практически полезной операции нам нужно взаимодействовать с несколькими объектами, причем определенным упорядоченным образом.

Например, есть система ведения бухгалтерии, нам нужно выдать зарплату сотруднику. Нужно посчитать зарплату, добавить расходную операцию и пересчитать бюджет. И все эти операции нужно сделать вместе:

```
class Bookkeeper:
    salary_counter: SalaryCounter
    expenses: ExpensesTable
    budget: Budget

    def pay_salary(self, worker):
        salary = self.salary_counter.calculate(worker)
        self.expenses.add(Expense(salary,
```

```
                    f"Salary for  {worker.name}"))
        self.budget.decrease(salary)
```

Если все операции будем делать вручную, это может привести к повторам одного и того же кода и логики, и может привести к ошибкам. Если мы из этих действий забудем что-то сделать или сделаем не в правильном порядке, потом будет сложно эту ошибку найти.

Мост

Позволяет отделить абстрактный алгоритм (последовательности действий) от конкретной реализации отдельных операций.

Например, есть класс, который отрисовывает столбчатую диаграмму, но при этом сам процесс рисования вынесен в отдельный класс, и их может быть несколько:

```
class BarCharterer:
    renderer: BarRenderer

    def render(self, caption, pairs):
        max_value = max(value for _, value in pairs)
        self.renderer.initialize(len(pairs), max_value)
        self.renderer.draw_caption(caption)
        for name, value in pairs:
            self.renderer.draw_bar(name, value)
        self.renderer.finalize()
```

Возможно, мы эту диаграмму будем рисовать в текстовом виде в терминале, на html странице, в svg формате и др. Но сама логика построения диаграммы из исходных данных остается постоянной.

«Отрисовщик» renderer может быть разным. Это взаимозаменяемые компоненты. Данный паттерн очень похож на поведенческий паттерн «шаблонный метод». Это метод, который представляет некий алгоритм, а шаги должен реализовать кто-то другой. Но шаблонный метод обычно используется при наследовании, когда у нас в классе родителя реализован сам шаблонный метод, а классы наследники должны реализовать конкретные шаги выполнения алгоритма.

В нашем случае это не наследование, а отдельный объект, которому мы все передаем. И дело может не ограничиваться только одним методом, в этом классе

может быть набор методов, соответственно, это не шаблонный метод, а шаблонный класс. Но логика та же.

Декоратор

В Python имеет особое значение. Используется для динамического подключения дополнительного поведения к объекту. В исходной версии декоратор – это объект, который берет какой-то другой объект, реализует такой же или чуть более расширенный интерфейс, при этом обращается к интерфейсу исходного объекта и что-то к нему добавляет:

```
class Logging:
  def __init__(self, obj):
     self._obj = obj

   def do_something(self, *args):
     logging.info(
        f'{self._obj} is doing something with {args}')
     return self._obj.do_something(*args)
```

Декораторы настолько полезны, что в Python они внедрены непосредственно в синтаксис языка:

```
def logging(func):
     def wrapper(*args):
        logging.info(
            f'{func.__name__} running with {args}')
        return func(*args)

     return wrapper
```

В Python декораторы используются для самых разных целей, значительно шире, чем исходный паттерн:

- для модификации поведения функции;
- для модификации класса (часто можно использовать вместо наследования);
- для удобной передачи функции в качестве параметра.

Например, регистрация обработчика какой-то команды:

```
@register_handler(command)
def handle_command():
```

```
    ...
# можно было бы сделать и так:
register_handler(command, handle_command)
```

У разных вариантов есть свои плюсы и свои минусы. Обычно фреймворки позволяют делать и так, и так. У синтаксиса декоратора лучше читаемость, так как логика описана здесь же. Но если функция написана не нами, мы можем передать ее в виде параметра.

Приспособленец (flyweight — «легковесный элемент»)
Объект, минимизирующий потребление ресурсов за счет разделения некоторых данных с другими аналогичными объектами.

Предположим, у нас в игре есть космический корабль, который при инициализации должен прочитать какие-то спрайты из файлов изображения:

```
class Spaceship(Sprite):
    def __init__(self, ...):
        ...
        self.images = [
            load_image(f'spaceship{i}.png')
            for i in range(16)
        ]
```

Неужели при создании каждого экземпляра нужно заново загружать все картинки и хранить в каждом экземпляре собственную копию этого списка? Этот вопрос решается в Python очень просто за счет добавления атрибута класса вместо атрибута каждого экземпляра:

```
class Spaceship(Sprite):
    images = [
        load_image(f'spaceship{i}.png')
        for i in range(16)
    ]
    def __init__(self, ...):
        ...
```

Мы пишем один атрибут класса, причем код будет выполнен во время объявления класса, во время считывания кода из файла еще до создания каких-либо объектов.

5. Поведенческие паттерны

Итератор

Паттерн итератор — часть языка Python. Итератор — это объект, реализующий метод `__next__`.

В других языках программирования таких встроенных структур нет, поэтому приходится что-то придумывать самостоятельно.

Многие паттерны — это попытка преодолеть ограниченность языка, на котором пишем. И наоборот, некоторые языки заранее реализовали паттерны, чтобы ими было удобно пользоваться.

Опишем некоторые паттерны, которых в Python исходно нет.

Команда

Позволяет хранить информацию о выполняемых действиях в виде объектов. Часто бывает полезно для сохранения истории действий.

Два частых случая использования:

1. Создание последовательности команд для выполнения позже.
2. Создание истории команд с возможностью отмены.

Реализация паттерна:

```
class Command(ABC):
    @abstractmethod
    def do(self):
        # сначала сохранить информацию для
        # возможной отмены,
        # потом выполнить действие
        ...
    @abstractmethod
    def undo(self):
        ...
```

Состояние
Используется, когда поведение объекта должно зависеть от его состояния. Часто используется при программировании интерактивных интерфейсов, если требуется диалог с пользователем, и мы должны знать, в каком состоянии сейчас этот диалог находится.

В некоторых случаях это реализуется просто, без специальных паттернов, но, в частности, при разработке, например, телеграм-бота это невозможно. Мы не можем дать сообщение и ждать ответа от функции. Мы должны дать сообщение и вернуть управление в главный цикл, который получает обновления с серверов. Когда придет ответ, мы должны понимать, что с ним делать.

Пример из телеграм-бота, который выдает студентам задачки и проверяет ответы:

```
student_router = Router(name='student_router’)

class StudentState(StatesGroup):
    new = State()
    idle = State()
    new_task = State()
    task_given = State()

@student_router.message(commands=["start"])
async def start(message: Message, state: FSMContext):
    if await state.get_state() is None:
       await state.set_state(StudentState.new)

@student_router.message(StudentState.new)
async def get_code(message: Message, state: FSMContext):
    code = message.text
    ...

@student_router.message(StudentState.task_given,
    F.text.casefold().in_(['пропустить', 'не знаю']))
async def give_up(message: Message, state: FSMContext):
    ...

@student_router.message(StudentState.task_given)
async def get_answer(message: Message, state: FSMContext):
    ...
```

```
@student_router.message(StudentState.idle)
async def default_answer(message: Message, state: FSMContext):
    await message.answer('Заданий пока нет.')
```

Мы видим, что этот паттерн реализуется без написания каких-либо классов. Есть класс student.router, но он не нами написан. Он занимается обработкой всех состояний. Вся система в целом называется **конечным автоматом** или **машиной состояний**.

Наблюдатель (Observer)

Позволяет объекту получать оповещения об изменении состояния других объектов. Для этого нужно описать специальный класс «Наблюдаемое» (Observable). Этот объект должен «знать», кто за ним наблюдает, должен иметь список наблюдателей, может их добавлять / удалять и оповещать:

```
class Observable:
    def __init__(self):
        self._observers = []

    def add_observer(self, observer):
        self._observers.append(observer)
        observer.update(self)

    def remove_observer(self, observer):
        self._observers.remove(observer)

    def notify_observers(self):
        for observer in self._observers:
            observer.update(self)
class History:
    def __init__(self):
        self._data = []

    def update(self, model):
        self._data.append((model.value, time.time()))

class Model(Observable):
    ...
    @property
```

```
    def value(self):
        return self._value

    @value.setter
    def value(self, new_value):
        if new_value != self._value:
            self._value = new_value
            self.notify_observers()
```

Мы можем использовать это также для сохранения истории действий.
Паттерн позволяет разделить два уровня логики:

- с одной стороны, объект, который выполняет какие-то действия;
- с другой стороны, другой объект, который эти действия обрабатывает другой логикой.

Канал событий

Паттерн «Наблюдатель» может быть расширен за счет использования центрального канала событий. В нем разные объекты могут генерировать события, а другие объекты на них подписываться, даже не зная о существовании друг друга. Например, пересчет бюджета в отдельном месте.

Широко используется в программировании графических интерфейсов.

Стратегия

Позволяет реализовать набор взаимозаменяемых алгоритмов, из которых пользователь выбирает тот, который ему нужен.

Данный валидатор реализует паттерн стратегия:

```
def read_int(prompt='', validator: Callable[[int], None] =
None):
    """
    validator - функция, выполняющая проверку корректности
    введенного значения, должна возбудить ошибку ValidationError
    с понятным для пользователя сообщением или вернуть None.
    """
    while True:
        try:
            value = int(input(prompt))
```

```
            if validator is not None:
                validator(value)
        except ValueError:
            print('Некорректное число’)
        except ValidationError as e:
            print(e)
        else:
            return value
```

Вообще, многие паттерны в Python или уже реализованы, или реализуются гораздо проще.

Посетитель
Используется, когда нужно применить одну и ту же логику к серии объектов.
Реализуется в Python функцией map:

```
for x in map(func, values): ...
```

Или генераторным выражением:

```
[func(x) for x in values]
```

Дополнительные материалы для самостоятельного изучения

1. Robert Martin. Design Principles and Design Patterns (2000)
2. Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования» (1994)
3. Кристофер Александер «Язык шаблонов» (1977)