Каждый аргумент метода должен быть аннотирован, должно быть известно какой тип данных возвращает метод.

Задание 1:

Предположим, что у нас есть класс User, который представляет пользователя нашего приложения. Класс имеет атрибуты name, email, password и role, которые хранят имя, электронную почту, пароль и роль пользователя соответственно. Класс также имеет методы register, login, logout, change_password, send_email и generate_report, которые позволяют регистрировать, входить, выходить, менять пароль, отправлять электронные письма и генерировать отчеты для пользователя.

```
class User:

    def __init__(self, name, email, password, role):
        self.name = name
        self.email = email
        self.password = password
        self.role = role

    def register(self):
        print(f"{self.name} is registered with email {self.email} and password {self.password}")

    def login(self):
        print(f"{self.name} is logged in with email {self.email} and password {self.password}")

    def logout(self):
        print(f"{self.name} is logged out")

    def change_password(self, new_password):
        print(f"{self.name} is changed password {self.password} to {new_password}")

    def send_email(self, subject, message, recipients):
        print(f"{self.name} is sent message {message} with {subject} to {recipients}")

    def generate_report(self, data):
        print(f"{self.name} is generated report with data {data}")
```

Этот класс нарушает принцип единственной ответственности, так как имеет несколько причин для изменения. Например, если мы захотим изменить способ регистрации или входа, мы должны изменить методы register и login. Если мы захотим изменить способ отправки писем или формат отчетов, мы должны изменить методы send_email и generate_report. Это может привести к ошибкам, несовместимостям или сложности поддержки кода.

Чтобы исправить эту проблему, мы разбейте класс User на несколько маленьких классов, каждый из которых будет иметь одну ответственность. Например, мы можем создать класс AuthService, который будет отвечать за регистрацию, вход и выход пользователя. Мы можем создать класс EmailService, который будет отвечать за отправку писем от имени пользователя. Мы можем создать класс ReportService, который будет отвечать за генерацию отчетов для пользователя. Класс User будет иметь только атрибуты, связанные с данными пользователя, и метод change_password, который будет менять пароль пользователя. Класс User будет

использовать объекты классов AuthService, EmailService и ReportService для выполнения своих задач.

Задание 2:

Предположим, у нас есть класс Employee, который хранит информацию о сотруднике и его зарплате. Класс имеет метод calculate_salary(), который вычисляет зарплату сотрудника в зависимости от его типа: менеджер, разработчик, тестировщик и т.д. Метод выглядит примерно так:

class Employee:

```
def __init__(self, name, type, base_salary):
    self.name = name
    self.type = type
    self.base_salary = base_salary

def calculate_salary(self):
    bonus = 0
    if self.type == "manager":
        bonus = self.base_salary * 0.2
    elif self.type == "developer":
        bonus = self.base_salary * 0.1
    elif self.type == "tester":
        bonus = self.base_salary * 0.3
    return self.base_salary + bonus
```

Этот класс нарушает принцип OCP, потому что если мы захотим добавить новый тип сотрудника или изменить правила расчета бонусов, нам придется изменить код метода calculateSalary(). Это может привести к ошибкам, сложности тестирования и нарушению работы существующего кода, который использует этот класс.

Чтобы исправить эту проблему необходимо сделать следующее:

• Создать абстрактный класс Employee, который будет содержать общие поля и методы для всех сотрудников, а также абстрактный метод calculate_salary(), который будет переопределяться в подклассах.

• Создать подклассы Manager, Developer и Tester, которые будут наследоваться от класса Employee и реализовывать метод calculate_salary() в соответствии с их логикой расчета бонусов.

Задание 3:

Предположим, у нас есть класс Bird, который хранит информацию о птице и имеет метод fly(), который позволяет птице летать. Класс выглядит примерно так:

class Bird:

```
def __init__(self, name, wingspan):

    self.name = name
```

```
        self.wingspan = wingspan

    def fly(self):
        print(f"{self.name} is flying with wingspan {self.wingspan}")
```

Этот класс нарушает принцип LSP, потому что если мы создадим подкласс Penguin, который наследуется от класса Bird, то мы не сможем использовать метод fly(), так как пингвины не умеют летать. Если мы попытаемся вызвать метод fly() для объекта типа Penguin, то мы получим нелогичный или ошибочный результат. Например:

```
class Penguin(Bird):
    def __init__(self, name, wingspan):
        super().__init__(name, wingspan)

p = Penguin("Pingu", 0.5)
p.fly() # Pingu is flying with wingspan 0.5
```

Чтобы исправить эту проблему необходимо сделать следующее:

- Создать абстрактный класс Animal, который будет содержать общие поля и методы для всех животных, но не будет иметь метод fly().
- Создать подкласс FlyingAnimal, который будет наследоваться от класса Animal и добавлять метод fly(), который будет реализовывать логику полета.
- Создать подклассы Bird и Bat, которые будут наследоваться от класса FlyingAnimal и переопределять метод fly() в соответствии с их особенностями.
- Создать подкласс Penguin, который будет наследоваться от класса Animal, но не будет иметь метод fly().

Задание 4:

Предположим, у нас есть класс Phone, который хранит информацию о телефоне и имеет методы для разных функций: звонить, отправлять сообщения, снимать фото, играть в игры и т.д. Класс выглядит примерно так:

```
class Phone:
    def __init__(self, model, number):
        self.model = model
        self.number = number
```

```
    def call(self, other_number):
        print(f"Calling {other_number} from {self.number}")

    def send_message(self, other_number, text):
        print(f"Sending '{text}' to {other_number} from {self.number}")

    def take_photo(self):
        print(f"Taking a photo with {self.model}")

    def play_game(self, game):
        print(f"Playing {game} on {self.model}")
```

Этот класс нарушает принцип ISP, потому что если мы создадим объект типа Phone, то мы будем зависеть от всех методов, даже если нам нужна только часть из них. Например, если мы хотим использовать телефон только для звонков и сообщений, то нам не нужны методы take_photo() и play_game(). Если мы хотим изменить логику этих методов, то мы можем повлиять на работу других клиентов, которые используют этот класс.

Чтобы исправить эту проблему, необходимо сделать следующее:

• Создать интерфейс Phone, который будет содержать только общие поля и методы для всех телефонов, например, model и number.

• Создать интерфейсы CallPhone, MessagePhone, PhotoPhone и GamePhone, которые будут наследоваться от интерфейса Phone и добавлять методы для конкретных функций, например, call(), send_message(), take_photo() и play_game().

• Создать классы BasicPhone, CameraPhone и SmartPhone, которые будут реализовывать разные комбинации интерфейсов в зависимости от их возможностей. Например, BasicPhone будет реализовывать только CallPhone и MessagePhone, CameraPhone будет реализовывать CallPhone, MessagePhone и PhotoPhone, а SmartPhone будет реализовывать все интерфейсы.

Задание 5:

Предположим, у нас есть класс Order, который хранит информацию о заказе и имеет метод process(), который обрабатывает заказ и сохраняет его в базу данных. Класс выглядит примерно так:

```
class Order:
    def __init__(self, items, total):
        self.items = items
        self.total = total
```

```
def process(self):
    db = Database() # создаем объект класса Database
    db.connect() # подключаемся к базе данных
    db.save(self) # сохраняем заказ в базе данных
```

Этот класс нарушает принцип DIP, потому что он зависит от конкретного класса Database, который реализует работу с базой данных. Если мы захотим изменить способ хранения заказов, например, использовать файлы или облачные сервисы, то нам придется изменить код класса Order. Это может привести к ошибкам, сложности тестирования и нарушению работы существующего кода, который использует этот класс.

Чтобы исправить эту проблему, необходимо сделать следующее:

• Создать интерфейс Storage, который будет содержать метод save(), который принимает объект заказа и сохраняет его в каком-то хранилище.

• Создать класс Database, который будет реализовывать интерфейс Storage и иметь методы connect() и save(), которые будут работать с базой данных.

• Создать класс File, который будет реализовывать интерфейс Storage и иметь метод save(), который будет сохранять заказ в файл.

• Изменить класс Order, чтобы он не создавал объект класса Database, а принимал объект типа Storage в качестве параметра в методе process(). Таким образом, класс Order не будет знать о деталях реализации хранилища, а будет использовать общий интерфейс.

Задание 6:

Создать data transfer object (DTO) для хранения и передачи информации о студенте. DTO должен иметь следующие атрибуты: имя, фамилия, возраст, курс, средний балл. Реализовать DTO с использованием четырех разных способов: dataclass, namedtuple, typeddict и pydantic. Добавить валидацию полей с помощью аннотаций типов, проверок и исключений. Проверки:

- Имя и фамилия должны состоять только из букв английского алфавита и начинаться с заглавной буквы
- Возраст должен быть не младше 18 и не старше 30
- Курс должен быть не меньше 1 и не больше 6
- Средний балл должен быть не больше 100 и не меньше 1