

# SOLID

## ВВЕДЕНИЕ

### Задачи урока

- 01 Понять принципы SOLID
- 02 Разобраться как и на что они влияют
- 03 Понять как можно облегчить разработку и поддержку кода

### SOLID

это аббревиатура, состоящая из первых букв пяти принципов объектно-ориентированного программирования, помогающих создавать гибкие, расширяемые и поддерживаемые программные системы. Давайте рассмотрим каждый принцип более подробно и приведем примеры

### 01

#### Принцип единственной ответственности Single Responsibility Principle - SRP

Каждый класс должен отвечать только за одну функцию или задачу. Это гарантирует легкость поддержки и позволяет вносить изменения без воздействия на другие части программы

#### Пример

```
...
class UserManager:
    def __init__(self):
        self.users = []
    def add_user(self, user):
        # логика добавления пользователя

    def remove_user(self, user):
        # логика удаления пользователя

    def update_user(self, user):
        # логика обновления информации о пользователе
```

```
class EmailSender:  
    def send_email(self, user, message):  
        # логика отправки электронной почты  
  
    def validate_email(self, email):  
        # логика проверки правильности электронной почты
```

В этом примере класс **UserManager** отвечает только за управление пользователями, в то время как класс **EmailSender** отвечает только за отправку электронной почты. Каждый класс имеет только одну ответственность

## 02

### Принцип открытости/закрытости Open/Closed Principle - OCP

Программные сущности должны быть открыты для расширения и закрыты для модификации. Это означает, что вместо изменения существующего кода следует добавлять новый код для внесения изменений

#### Пример

```
...  
  
class Shape:  
    def area(self):  
        pass  
  
class Rectangle(Shape):  
    def __init__(self, width, height):  
        self.width = width  
        self.height = height  
  
    def area(self):  
        return self.width * self.height  
  
class Circle(Shape):  
    def __init__(self, radius):  
        self.radius = radius  
  
    def area(self):  
        return 3.14 * self.radius ** 2
```

Здесь классы **Rectangle** и **Circle** наследуются от базового класса **Shape** и переопределяют его метод **area()**. Если в будущем мы захотим добавить новую фигуру, например, **Triangle**, мы сможем сделать это без изменения кода классов **Rectangle** и **Circle**

## 03

### Принцип подстановки Барбары Лисков Liskov Substitution Principle - LSP

Объекты должны быть заменяемыми своими типами наследников без изменения корректности программы. Это означает, что классы-наследники должны соответствовать интерфейсу своих базовых классов и не нарушать их инварианты

#### Пример

```
...
class Bird:
    def fly(self):
        pass

class Duck(Bird):
    def fly(self):
        # логика полета утки

class Penguin(Bird):
    def fly(self):
        raise NotImplementedError("Пингвины не умеют летать!")
```

Здесь классы `Duck` и `Penguin` наследуются от класса `Bird` и реализуют его метод `fly()`. Однако, `Penguin` вызывает исключение, так как пингвины не умеют летать. Это отражает правило **LSP**, поскольку мы можем использовать объекты `Duck` и `Penguin` вместо объекта `Bird`, не нарушая корректность программы

## 04

### Принцип разделения интерфейса Interface Segregation Principle - ISP

Клиенты не должны зависеть от интерфейсов, которые они не используют. Вместо одного общего интерфейса следует создавать несколько маленьких, специфичных для каждого клиента

## Пример

```
...
class Printer:
    def print_document(self, document):
        pass

class Scanner:
    def scan_document(self):
        pass

class Fax:
    def send_fax(self, document):
        pass

class OfficeMachine(Printer, Scanner, Fax):
    pass
```

В этом примере класс **OfficeMachine** наследуется от различных интерфейсов, которые предоставляют функциональность для печати, сканирования и отправки факсов. Однако, клиенты могут использовать только те методы, которые им нужны, независимо от всего функционала **OfficeMachine**

## 05

### Принцип инверсии зависимостей Dependency Inversion Principle - DIP

Модули высокого уровня не должны зависеть от модулей низкого уровня. Оба типа модулей должны зависеть от абстракций. Кроме того, принцип утверждает, что абстракции не должны зависеть от деталей, а детали должны зависеть от абстракций

## Пример

```
...
from abc import ABC, abstractmethod

class PaymentProcessor(ABC):

    @abstractmethod
    def process_payment(self, amount):
        pass

class CreditCardPayment(PaymentProcessor):
    def process_payment(self, amount):
```

```
# Логика обработки платежа с использованием кредитной карты

class PayPalPayment(PaymentProcessor):
    def process_payment(self, amount):
        # Логика обработки платежа с использованием PayPal

class Order:

    def __init__(self, payment_processor):
        self.payment_processor = payment_processor

    def process_order(self, amount):
        # Логика обработки заказа
        self.payment_processor.process_payment(amount)
```

В приведенном примере у нас есть абстрактный класс **PaymentProcessor**, который определяет метод **process\_payment()**. Затем есть два класса **CreditCardPayment** и **PayPalPayment**, которые наследуются от **PaymentProcessor** и реализуют свою собственную логику обработки платежей

Затем у нас есть класс **Order**, который зависит от абстракции **PaymentProcessor**. При создании экземпляра **Order** мы передаем конкретный объект **PaymentProcessor** (например, **CreditCardPayment** или **PayPalPayment**). Это позволяет нам гибко заменять или добавлять новые способы обработки платежей без изменений в коде **Order**. Мы можем легко добавить еще один класс, например, **BitcoinPayment**, реализующий метод **process\_payment()**, и использовать его без изменения кода **Order**

## ИТОГИ

- ✓ SOLID помогают облегчить поддержку и разработку кода. Например если наш класс должен использовать несколько зависимостей то нам стоит использовать DIP который помогает избавиться от загрузок всех (даже ненужных) зависимостей
- ✓ Перед разработкой кода всегда стоит продумать архитектуру, понять какие классы для чего будут использоваться после чего реализовать классовую структуру таким образом чтобы она соответствовала принципам SOLID. Это сэкономит вам время на поддержке и доработки кода в будущем