Тест по прошлой лекции

https://forms.gle/sNMBVJqoJDoFwQzeA

Парадигма ООП вторая часть

Принципы SOLID

Направление «Искусственный интеллект и наука о данных», 23.Б16-мм, 23.Б18-мм

01.12.2023

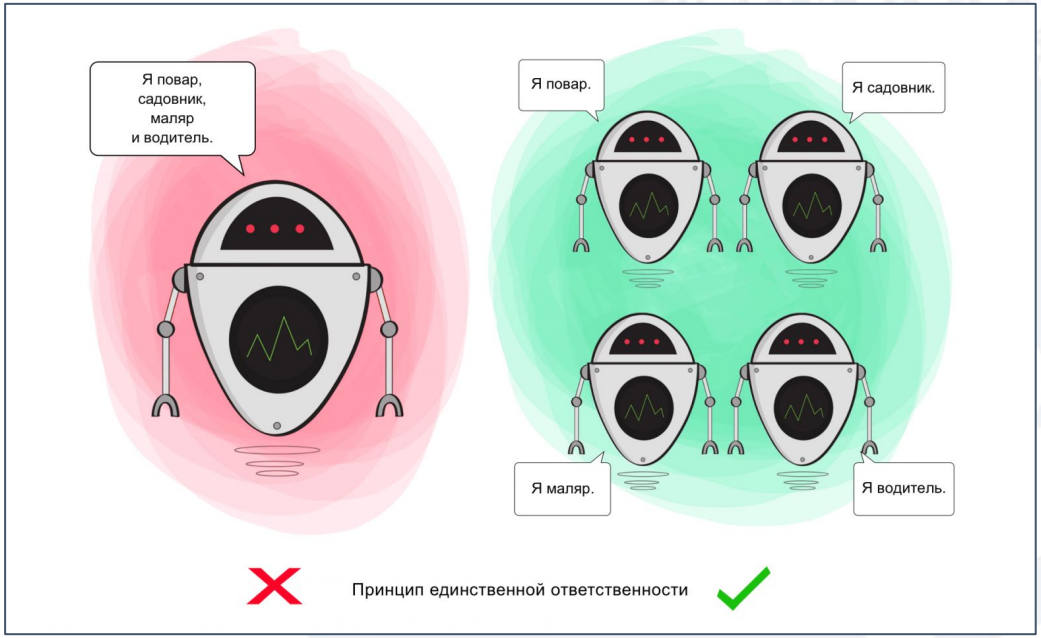
Абрамов Максим Викторович, Есин Максим Сергеевич

Что за принципы SOLID?

- **SOLID** – аббревиатура от названий пяти основных принципов ООП:
  - **Single responsibility** — принцип единственной ответственности
  - **Open-closed** — принцип открытости / закрытости
  - **Liskov substitution** — принцип подстановки Барбары Лисков
  - **Interface segregation** — принцип разделения интерфейса
  - **Dependency inversion** — принцип инверсии зависимостей

- Принципы SOLID говорят **не столько о процессе написания кода**, сколько о том, **как код должен выглядеть в итоге** независимо от процесса

1. Принцип единой ответственности



- Каждый класс должен быть **ответственен лишь за что-то одно**
- Если класс будет иметь **единственную зону ответственности**, то и **изменять** его нужно будет только **по одной причине**
- В случае, когда класс имеет несколько зон ответственности, при изменении класса можно затронуть другие зоны, не заметив этого
- Принцип применяется **не только к классам, но и к функциям и методам**

1. Принцип единой ответственности

- Класс *User* имеет н**есколько зон ответственности**: он сохраняет и удаляет пользователей из базы, отправляет письма, генерирует отчеты
- По смыслу названия этот класс **никак не должен отвечать за четыре операции сразу**
- Чтобы соответствовать принципу единой ответственности, нужно вынести методы в отдельные классы

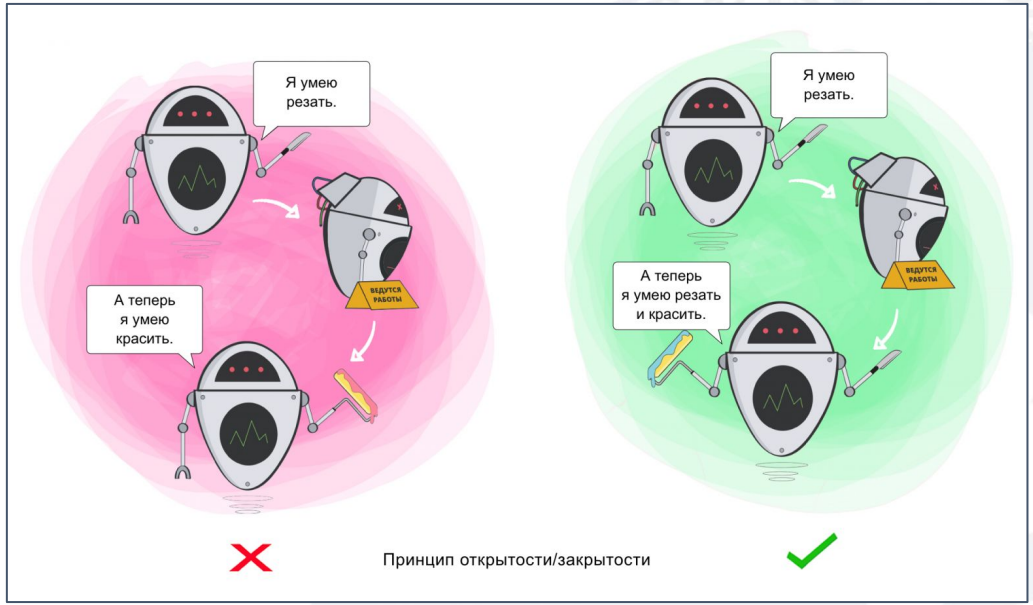
```python
class User:
    def __init__(self, username, email):
        self.username = username
        self.email = email

    def save(self):
        # Логика сохранения пользователя в базе данных
        pass

    def delete(self):
        # Логика удаления пользователя из базы данных
        pass

    def send_email(self, message):
        # Логика отправки письма пользователю
        pass

    def generate_report(self):
        # Логика генерации отчета пользователя
        pass
```

1. Принцип единой ответственности


```python
class User:
    def __init__(self, username, email):
        self.username = username
        self.email = email

    def save(self):
        # Логика сохранения пользователя в базе данных
        pass

    def delete(self):
        # Логика удаления пользователя из базы данных
        pass

class EmailSender:
    def send_email(self, user, message):
        # Логика отправки электронной почты пользователю
        pass

class ReportGenerator:
    def generate_report(self, user):
        # Логика генерации отчета пользователя
        pass
```

- Теперь все три класса удовлетворяют принципу единой ответственности, потому что **каждый отвечает за что-то одно**
- Следование принципу помогает разделить функциональность на более мелкие и специализированные классы, упрощает понимание кода и делает его более гибким для изменений

2. Принцип открытости/закрытости



- Программные сущности, такие как классы, модули и функции, должны быть **открыты для расширения,** но **закрыты для модификации**
- При этом, расширение класса не должно требовать модификации существующего кода
- Предполагается, что для расширения нужно использовать **комбинацию наследования и полиморфизма**

2. Принцип открытости/закрытости

- Классы *Rectangle* и *Circle* наследуются от абстрактного *Shape*, определяя метод *calculate_area*
- Класс *AreaCalculator* **использует экземпляры наследников** *Shape*, чтобы вычислять их суммарную площадь
- Это еще один наглядный пример того, как работает **полиморфизм**

```python
class AreaCalculator:
    def calculate_total_area(self, shapes):
        total_area = 0
        for shape in shapes:
            total_area += shape.calculate_area()
        return total_area
```

```python
class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return pi * self.radius ** 2
```

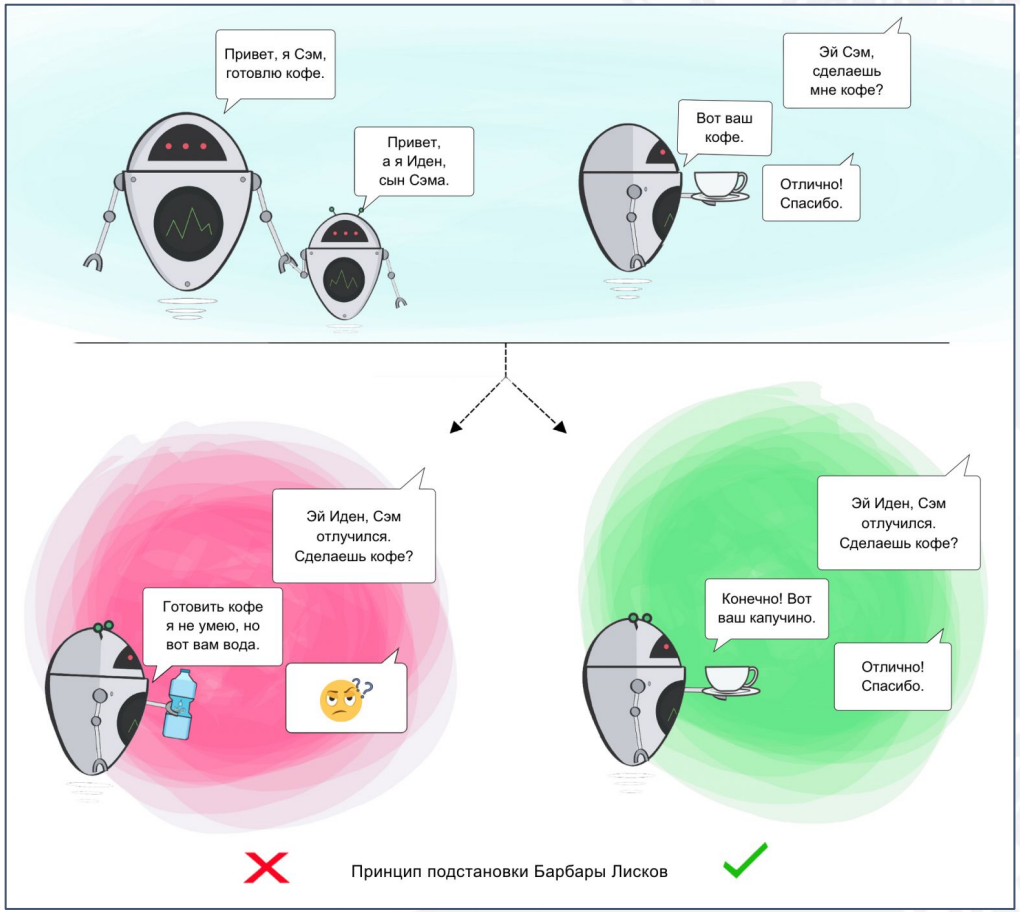
2. Принцип открытости/закрытости


```python
class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def calculate_area(self):
        return 0.5 * self.base * self.height


# Использование
shapes = [Rectangle(4, 5), Circle(3), Triangle(6, 2)]
calculator = AreaCalculator()
total_area = calculator.calculate_total_area(shapes)
print(total_area)
```

- Чтобы расширить функциональность класса *AreaCalculator*, создаем еще одного наследника класса *Shape* — класс *Triangle*
- Таким образом, код класса *AreaCalculator* при расширении функциональности никак не изменился

```python
class AreaCalculator:
    def calculate_total_area(self, shapes):
        total_area = 0
        for shape in shapes:
            total_area += shape.calculate_area()
        return total_area
```

3. Принцип подстановки Барбары Лисков



- Принцип предполагает, что объекты подклассов должны быть полностью **взаимозаменяемы** с объектами своих родительских классов
- Любой дочерний класс должен **вести себя точно так же, как родительский класс**, то есть обрабатывать те же запросы, что и родитель, и выдавать тот же результат

3. Принцип подстановки Барбары Лисков

- Класс *Logger* является родительским для класса *CustomLogger*
- Очень важно заметить, что *CustomLogger* **не переопределяет метод** *log(level, message)*, **а добавляет свой**, с другим порядком параметров и аргументом по умолчанию
- Тогда **при вызове метода** *log(level, message)* у экземпляра класса CustomLogger **будет вызван метод из наследника**, а не из родителя

```
class Logger:
    def log(self, level, message):
        print(f'[{level}]: {message}')

class CustomLogger(Logger):
    def log(self, message, level='debug'):
        super().log(level, message)
```

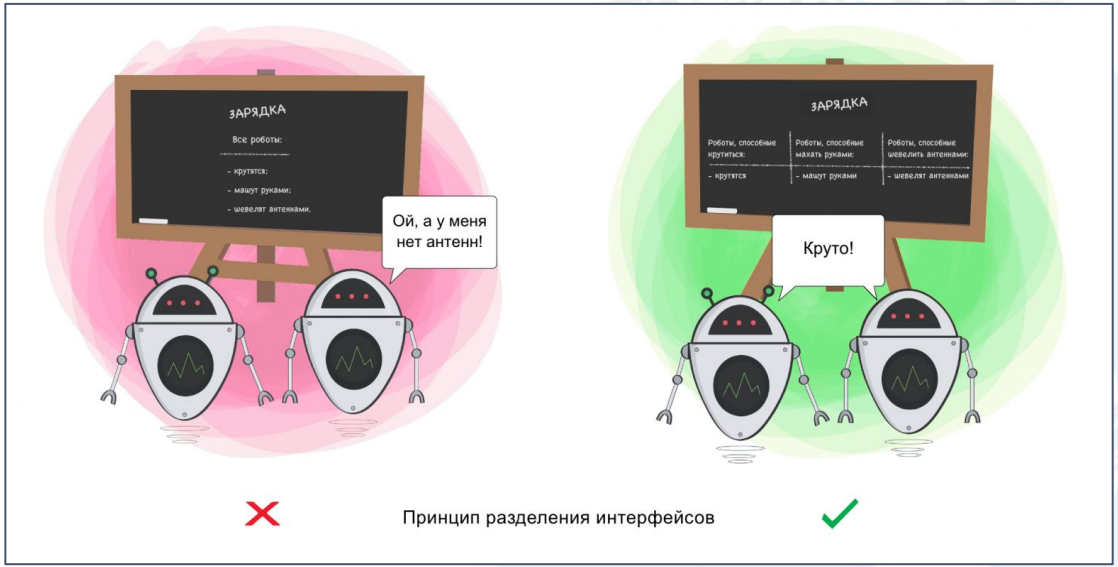
```
logger = Logger()
logger.log('debug', 'Doing work')
logger.log('info', 'Useful for debugging')

customLogger = CustomLogger()
customLogger.log('Doing work') # По умолчанию debug
customLogger.log('Useful for debugging', 'info')
```

3. Принцип подстановки Барбары Лисков

- Level обычно имеет одно из 8 значений, например, *debug, info, error, fatal*
- Когда в level передается что-то другое, например, строка *‘boom!’*, нарушается принцип подстановки: **наследник работает не так,** как его **родитель**
- Принято говорить, что **иерархия классов** CustomLogger и Logger **не удовлетворяет принципу** подстановки Барбары Лисков

```
logger = CustomLogger()
database.setLogger(logger);

database.doSomething()
# Внутри вызывается логгер
# logger.log('info', 'boom!')
```

4. Принцип разделения интерфейсов



- Класс должен производить только те **операции,** которые необходимы **для осуществления его функций**
- Все другие действия следует либо удалить совсем, либо переместить, если есть вероятность, что они понадобятся другому классу в будущем

4. Принцип разделения интерфейсов

- Вспомним **пример с геометрическими фигурами**: есть абстрактный класс *Shape* с одним абстрактным методом *calculate_area()*
- У *Shape* есть **два наследника**: *Rectangle* и *Parallelepiped*
- Параллелепипед – это вообще-то объемная фигура, у нее **можно вычислить объем**
- Как получить возможность вычислять объем? В данной иерархии проще всего **добавить абстрактный метод** *calculate_volume* в *Shape*

```
class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        # вычисление площади
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

class Parallelepiped(Shape):
    def __init__(self, width, height, length):
        self.width = width
        self.height = height
        self.length = length

    def calculate_area(self):
        return 2 * (
            self.width * self.height +
            self.height * self.length +
            self.length * self.width)
```

4. Принцип разделения интерфейсов

- Но теперь и **классу прямоугольника нужно реализовать этот метод**. Как и зачем?

```python
class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        # вычисление площади
        pass

    @abstractmethod
    def calculate_volume(self):
        # вычисление объема
        pass
```

```python
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

    def calcultate_volume(self):
        # зачем прямоугольнику объем? как его вычислять?
        pass
```

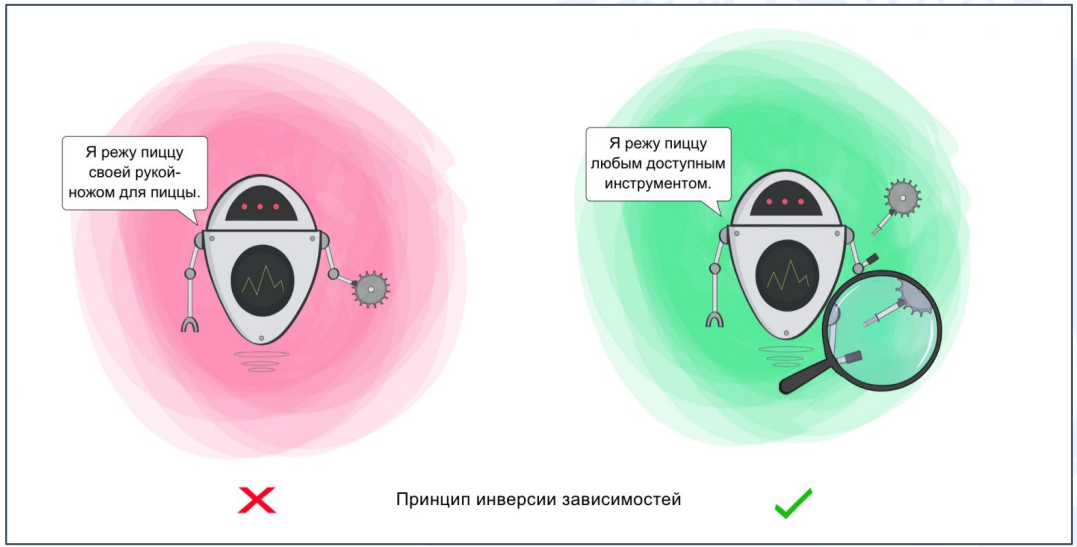
4. Принцип разделения интерфейсов

- Разделим методы для объема и площади как в иерархии *Shape* и *VolumeShape*
- Тогда класс параллелепипеда будет наследником *VolumeShape*, а класс прямоугольника — наследником *Shape*

```
class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        # вычисление площади
        pass

class VolumeShape(Shape):
    @abstractmethod
    def calculate_volume(self):
        # вычисление объема
        pass
```

```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

class Parallelepiped(VolumeShape):
    def __init__(self, width, height, length):
        self.width = width
        self.height = height
        self.length = length

    def calculate_area(self):
        return 2 * (
            self.width * self.height +
            self.height * self.length +
            self.length * self.width)

    def calculate_volume(self):
        return self.width * self.height * self.length
```

5. Принцип инверсии зависимостей



- **Классы должны зависеть от абстракций**, а не от конкретных реализаций
- Высокоуровневые модули не должны зависеть от низкоуровневых модулей
- При этом оба типа модулей должны зависеть от своих абстракций

5. Принцип инверсии зависимостей


```python
class EmailSender:
    def send_notification(self, message):
        # Логика отправки уведомления по электронной почте
        pass

class User:
    def __init__(self, username, email):
        self.username = username
        self.email = email
        self.notification_service = EmailSender()

    def send_notification(self, message):
        self.notification_service.send_notification(message)
```

- Рассмотрим класс *User*, который использует класс *EmailSender* как **вспомогательный**: он отправляет с помощью него письма
- Сейчас классы *User* и *EmailSender* **сильно связаны**: у User **нельзя заменить вспомогательный класс** для отправки оповещений без изменений в коде

5. Принцип инверсии зависимостей


```python
class Notification(ABC):
    @abstractmethod
    def send_notification(self, message):
        pass


class EmailSender(Notification):
    def send_notification(self, message):
        # Логика отправки уведомления по электронной почте
        pass


class SMSNotification(Notification):
    def send_notification(self, message):
        # Логика отправки уведомления по SMS
        pass
```

```python
class User:
    def __init__(self, username, email, notification_service):
        self.username = username
        self.email = email
        self.notification_service = notification_service

    def send_notification(self, message):
        self.notification_service.send_notification(message)
```

```python
email_sender = EmailSender()
user = User("John", "john@example.com", email_sender)
user.send_notification("Hello!")

sms_notification = SMSNotification()
user = User("Jane", "jane@example.com", sms_notification)
user.send_notification("Hi there!")
```

- Создадим иерархию для наследников Notification
- Теперь в конструктор User можно передавать любые объекты наследников Notification, User **будет работать без привязки к объекту класса инструмента**
- Это очередной пример применения полиморфизма

Итоги раздела

- **Принцип единственной ответственности**. Каждый класс должен иметь только одну зону ответственности
- **Принцип открытости-закрытости**. Классы должны быть открыты для расширения, но закрыты для изменения
- **Принцип подстановки Барбары Лисков**. Должна быть возможность вместо родительского класса подставить любой его класс-наследник, при этом работа программы не должна измениться
- **Принцип разделения интерфейсов**. Данный принцип означает, что не нужно заставлять класс реализовывать интерфейс, который не имеет к нему отношения
- **Принцип инверсии зависимостей**. Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций

Парадигма ООП вторая часть

Принципы SOLID

Направление «Искусственный интеллект и наука о данных», 23.Б16-мм, 23.Б18-мм

01.12.2023

Абрамов Максим Викторович, Есин Максим Сергеевич

- **Ссылка на презентации с лекций**:
- https://drive.google.com/drive/folders/1rTUCWtOYSdtJirf-LVicGEGeBLbDzmd3?usp=sharing

- **Ссылка на билеты (вкладка Билеты)**:
- https://drive.google.com/drive/folders/1rTUCWtOYSdtJirf-LVicGEGeBLbDzmd3?usp=sharing

Формат зачета для тех, у кого не хватает баллов

Санкт-Петербургский государственный университет

- 5 вопросов, 20-40 минут подготовки. За время подготовки нужно подготовить краткий ответ: главные определения, заготовки для примеров, какие-то важные пометки
- Если баллов нужно немного, то можно отвечать без подготовки: тяните 5 вопросов, выбираете понравившиеся и сразу рассказываете, не обязательно полно
- Для получения 10 баллов из 10 за вопрос нужно будет ответить на все дополнительные вопросы по разделу
- Если у вас 35+ баллов и вы хотите отвечать на вопросы без подготовки, есть возможность сдать онлайн в заранее оговоренное время. Для этого нужно записаться в таблице в один из временных слотов и подключиться с камерой и микрофоном к созвону