

Веб-приложение генерации диаграмм классов на основе пользовательского Python-кода

Болтовский Гавриил Александрович

Приамурский государственный университет им. Шолом-Алейхема

Студент

Аннотация

В данной статье рассматривается реализация веб-приложения, которое позволит построить диаграмму классов согласно нотации UML на основе заданного программного кода, написанного на Python. Пользователь может загрузить программный код, включающий классы (в том числе абстрактные) и интерфейсы и на выходе получить SVG-изображение диаграммы классов. Данное приложение можно использовать в обучающих целях в рамках дисциплин, посвящённых изучению проектирования информационных систем.

Ключевые слова: UML, диаграммы классов, Python, FastAPI, генератор UML диаграммы

Web application for generating class diagrams based on user Python code

Boltovsky Gavriil Alexandrovich

Sholom-Aleichem Priamursky State University

Student

Abstract

This article discusses the implementation of a web application that allows you to build a class diagram according to UML notation based on a given Python program code. The user can upload program code that includes classes (including abstract ones) and interfaces and get an SVG image of the class diagram as output. This application can be used for educational purposes in disciplines related to the study of information system design.

Keywords: UML, class diagrams, Python, FastAPI, UML diagram generator

Анализ программного кода может представлять трудную задачу. Часто такие трудности проявляются на этапе анализа кода, включающего в себя переменные и функции, названия которых не определяют их назначение в программе, либо в коде, включающем в себя объявление классов и интерфейсов. Связи между классами могут быть не всегда очевидными на первый взгляд, что, в свою очередь, ведёт к неправильному пониманию архитектуры информационной системы.

При обучении объектно-ориентированному программированию (ООП) особое значение имеет наглядное представление архитектуры программного

кода, поскольку именно архитектурная организация обеспечивает формирование последовательной, логически связной и легко интерпретируемой бизнес-логики системы.

Наличие инструмента, способного анализировать программный код на наличие классов и интерфейсов, позволит значительно упростить задачу анализа классов путём построения специализированной диаграммы, которая в достаточной мере позволит описать структуру классов в программе.

Проблема обучения студентов объектно-ориентированному программированию является предметом исследования значительного числа исследователей. В работе Р. Р. Мухаметзянова [1] предложена методика обучения, основанная на принципе преемственности, согласно которому базовые концепции объектно-ориентированного программирования начинают изучаться еще на школьном этапе обучения. При этом подчеркивается, что переход к изучению ООП должен осуществляться только после формирования у обучающихся устойчивых алгоритмических навыков. Автор также обосновывает целесообразность использования языка программирования C в качестве основного инструмента обучения программированию в школе.

В исследовании Б. С. Садулаева и У. С. А. Мунаева [2] предлагается другой подход к обучению программированию. Они считают, что лучше начинать с объектно-ориентированного программирования, не изучая сначала структурное программирование. Это, по их мнению, облегчает переход между разными способами программирования.

Обучение должно проходить в два этапа. Сначала студенты учат основные понятия объектно-ориентированного программирования. Затем они узнают, как эти понятия применяются на практике. Подобная модульная структура, как отмечают авторы, обеспечивает соблюдение дидактического принципа последовательного перехода от простого к более сложному.

В этом исследовании используется UML-диаграммы, чтобы визуализировать структуру классов. UML, или унифицированный язык моделирования, является широко известным инструментом, который помогает описывать и анализировать архитектуру программных систем. Его можно использовать в различных областях, например, А. Д. Шатова [3] использовала UML-диаграммы, чтобы проанализировать и смоделировать бизнес-процессы. Е. А. Сидорович также показал, как UML можно использовать в гуманитарных исследованиях [4].

Целью исследования является разработка веб-приложения для генерации диаграмм классов согласно нотации UML, входными данными для которого являться программный код, написанный на языке программирования Python.

Для решения поставленной задачи используется язык программирования Python, фреймворк для разработки веб-приложений FastAPI и CSS-фреймворк UIKit.

В первую очередь требуется понимать, что такое диаграмма классов и как она строится в нотации UML.

Диаграмма классов определяет типы классов системы и различного рода статические связи, которые существуют между ними. На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между классами.

Класс (class) – это абстрактное описание множества однородных объектов, имеющих одинаковые атрибуты, операции и отношения с объектами других классов.

Основными компонентами диаграммы классов являются сами классы, интерфейсы и связи между ними. Графически класс в нотации языка UML изображается в виде прямоугольника, который дополнительно может быть разделён горизонтальными линиями на разделы или секции. В этих секциях могут указываться имя класса, атрибуты и операции класса.

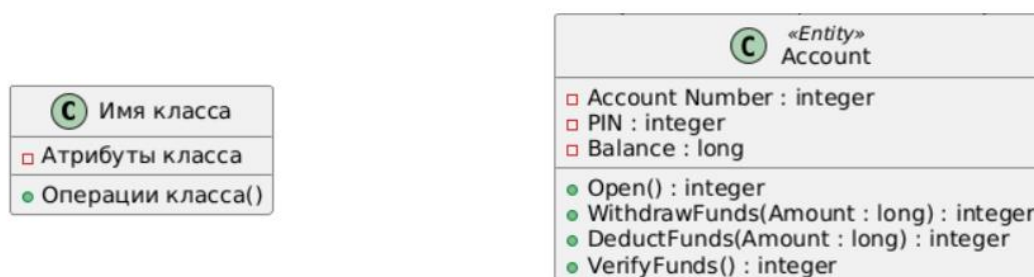


Рисунок 1. – Структура классов на диаграмме классов, а также пример реального класса приложения

Атрибут (Attribute) – это содержательная характеристика класса, описывающая множество значений, которые могут принимать отдельные объекты класса. Проще говоря, это характеристика (свойство) объекта, которая описывает его состояние и хранит данные.

Атрибут класса служит для представления отдельного свойства или признака, который является общим для всех объектов данного класса.

Операция класса (в программировании чаще называется методом) – это поведение, которое объект данного класса может выполнять.

Метод определяет действие, которое объект способен совершить, либо вычисление, которое он может произвести.

Например, следующий пример класса на языке программирования Python описывает структуру класса, в котором есть всего 1 атрибут, а также 2 метода:

```
class Service:
    repo: Repo

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

    def run(self):
        print(self.repo.get(1))
```

Класс `Service` содержит атрибут `геро`, который представляет собой объект другого класса `Реро`, а также 2 метода: конструктор класса (в Python обозначается как функция с названием `__init__`) и метод `run`, выводящий на экран первый элемент словаря из объекта `геро`. С помощью UML можно построить диаграмму данного класса (см. рисунок 2).

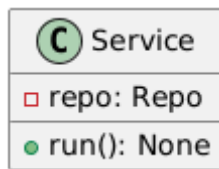


Рисунок 2. –Элемент класса `Service` на диаграмме классов

В UML для диаграмм классов предусмотрено четыре вида связей:

1. Ассоциации – это семантическая связь между классами. Её рисуют на диаграмме классов в виде обыкновенной линии.

2. Зависимости отражают связь между классами, но они всегда однонаправленны и показывают, что один класс зависит от определений, сделанных в другом. Зависимости изображают в виде стрелки, проведённой пунктирной линией.

3. Агрегация – это связь между целым и его частью (например, класс `Компьютер` и классы, описывающие его части). Агрегации визуализируют в виде линий с ромбиком у класса, являющегося целым.

4. Обобщения показывают связи наследования между двумя классами. Она позволяет одному классу наследовать все атрибуты, операции и связи другого.

В случае с атрибутом `геро` стоит рассматривать вариант связи зависимости, тем самым можно показать, что объект класса `Service` использует `Реро`, напрямую завися от его содержимого. Класс `Реро` определён следующим образом:

```
class Repo(Repository):
    def __init__(self):
        self.cache: dict[int, str] = {}

    def get(self, id: int) -> str:
        return self.cache.get(id, "")
```

Данный класс реализует интерфейс `Repository`, в котором реализован метод `get`.

Интерфейс (Interface) – это стереотип, задающий контракт в виде набора публичных операций без реализации, который обязуется выполнять любой класс, его реализующий.

То есть, это список действий, которые объект обещает уметь выполнять, без указания того, как именно он это делает. Классы могут реализовывать интерфейсы.

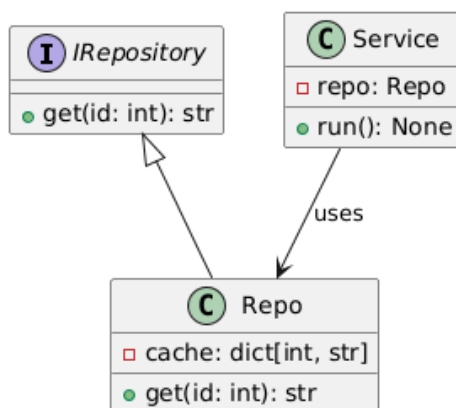


Рисунок 3. – Диаграмма классов по описанному выше примеру

Имея общее представление о принципах построения диаграмм классов можно перейти к генерации таких диаграмм на основе программного кода на языке программирования Python.

В первую очередь была определена структура приложения. Само приложение разработано с применением Python, FastAPI, HTML, JavaScript и CSS-фреймворка UIKit. Основное назначение заключается в парсинге программного пользовательского кода с поиском классов и определения связи между ними. На основе полученной информации приложение строит диаграмму классов и позволяет сохранить её в SVG-формате.

Приложение включает в себя несколько компонентов: основного скрипта, запускающего веб-сервер и само приложение, парсера программного кода, генератора SVG-изображений, а также из необходимого для работы JS-скрипта и шаблона HTML-страницы. Также в проект включены 2 тестового примера, на основе которых будет продемонстрирована работа приложения:

```

| requirements.txt
| sample_code1.py
| sample_code2.py
|--- app
|   | main.py
|   | parser.py
|   | svg.py
|   | __init__.py
|--- static
|   | app.js
|--- templates
|   | index.html
  
```

Была подготовлена HTML-страница, на которой есть поле для ввода программного кода, а также специальная область для предпросмотра диаграммы (см. рисунок 4). Саму диаграмму можно скачать после её генерации:

Генератор UML-диаграмм классов из Python-кода

SVG

Python код

Предпросмотр

ПОКАЗАТЬ ЦЕЛИКОМ

Вставьте код с class ...

> СГЕНЕРИРОВАТЬ

• Вставьте Python-код.

Рисунок 4. – Главная страница веб-приложения

Стоит рассмотреть принцип работы парсера в приложении: парсер берёт весь текст, строит AST-дерево с помощью функции `ast.parse(code)` и смотрит только верхний уровень модуля (`tree.body`). То есть он учитывает только классы, объявленные прямо в файле, вида:

```
class A: ...  
class B(A): ...
```

Вложенные классы (класс внутри функции/внутри другого класса), а также классы, созданные динамически, не учитываются – потому что обход идёт по верхнему уровню модуля, а не по всему дереву рекурсивно (иначе диаграмма может быть сильно перегружена и включать в себя много лишних деталей).

На первом проходе создаётся «каркас» для каждого найденного класса: имя и список базовых классов (основанных на принципе наследования в ООП). Базы берутся из `node.bases` и превращаются в строку через функцию `ast.unparse`.

Ниже представлен пример программного кода формирования массива баз:

```
for node in tree.body:  
    if isinstance(node, ast.ClassDef):  
        name = node.name  
        bases = [(_expr_to_str(b) or "") for b in node.bases]  
        classes[name] = UMLClass(name=name, bases=[b for b in  
bases if b])
```

Класс помечается как `is_interface`, если выполняется хотя бы одно из условий:

1. Среди баз есть `ABC`, `abc.ABC`, `ABCMeta`, `abc.ABCMeta`.
2. Среди баз есть `Protocol`, `typing.Protocol`, `typing_extensions.Protocol`.
3. Внутри класса найден хотя бы один метод с декоратором, заканчивающимся на `abstractmethod` (или ровно `abstractmethod`).

То есть сам стереотип интерфейса выводится эвристикой по базовым классам и наличию `@abstractmethod`. У интерфейса на диаграмме в секции с названием класса добавляется соответствующий стереотип `<<interface>>`. Связи наследования рисуются пунктиром, если родитель – интерфейс, а ребёнок – не интерфейс.

В приложении было решено строить только 2 вида связей:

1. Наследование. Берётся из секции кода `class Child(Base): ...` только если `Base` тоже найден среди классов этого же кода. Тогда рисуется стрелка наследования.

2. Ассоциации. Ассоциация появляется, если в типах атрибутов, аргументов методов или возвращаемого типа встречается имя другого класса.

Для извлечения имён из типа парсер выполняет функцию `ast.parse()` и собирает все названия (например, из `typing.List[Foo]` функция извлечёт `List` и `Foo`, из `mod.Foo` – только `Foo`). Потом алгоритм строит пересечение результата со множеством имён найденных классов.

При этом связь класса с самим собой удаляется, а также имена базовых классов (наследование) из ассоциаций игнорируются, чтобы не дублировать наследование как «ассоциацию».

Основной метод парсера возвращает словарь с информацией о классах. Например, ниже представлен результат работы парсера на основе ранее рассмотренного примера с классами `Service` и `Repo`:

```
{
  'IRepository': UMLClass(
    name='IRepository',
    bases=['ABC'],
    is_interface=True,
    attributes={},
    methods={
      'get': UMLMethod(
        name='get',
        args=[('id', 'int')],
        returns='str',
        visibility='+',
        is_abstract=True
      )
    },
    associations=set()
  ),
  'Repo': UMLClass(
    name='Repo',
    bases=['IRepository'],
```

```
        is_interface=False,
        attributes={
            'cache': UMLAttribute(
                name='cache',
                type_str='dict[int, str]',
                visibility='+'
            )
        },
        methods={
            '__init__': UMLMethod(
                name='__init__',
                args=[],
                returns=None,
                visibility='+',
                is_abstract=False
            ),
            'get': UMLMethod(
                name='get',
                args=[('id', 'int')],
                returns='str',
                visibility='+',
                is_abstract=False
            )
        },
        associations=set()
    ),
    'Service': UMLClass(
        name='Service',
        bases=[],
        is_interface=False,
        attributes={
            'repo': UMLAttribute(
                name='repo',
                type_str='Repo',
                visibility='+'
            )
        },
        methods={
            '__init__': UMLMethod(
                name='__init__',
                args=[('repo', 'Repo')],
                returns=None,
                visibility='+',
                is_abstract=False
            ),
            'run': UMLMethod(
                name='run',
                args=[],
                returns='None',
                visibility='+',
                is_abstract=False
            )
        },
    ),
```

```

        associations={'Repo'}
    )
}

```

На основе данного словаря генератор `svg.py` строит соответствующую диаграмму.

Структуру словаря можно описать следующим образом: ключами словаря являются сами классы программного кода. Этим ключам соответствуют значения, представленные в виде объектов специального класса `UMLClass`, описывающий класс для диаграммы. Класс включает в себя: название, базы, атрибуты и методы, а также множество ассоциаций и является ли он интерфейсом.

Стоит отметить, что также учитывается видимость свойств и методов. В UML для полей или методов ставится:

1. + если имя не начинается с `_`.
2. # если начинается с `_` (protected).
3. - если начинается с `__` (private), но не учитываются магические методы (например, конструкторы и деструкторы и др.).

Генератор SVG-диаграммы работает как визуализатор ранее разобранный UML-модели: на вход получает словарь классов (который был описан выше), а на выходе возвращает цельную строку SVG с прямоугольниками-классами и соединяющими линиями.

Для каждого класса формируются две текстовые секции:

- список атрибутов (поля);
- список методов (с аргументами, типами и признаком абстрактности).

Поля/методы сортируются по имени, а если секция пустая – подставляется строка-заглушка.

Чтобы диаграмма выглядела структурно сначала оценивается «глубина» класса в дереве наследования (сколько уровней до корня), затем классы группируются по этим уровням, после чего каждый уровень рисуется отдельной горизонтальной «строкой».

Итоговые размеры SVG (`width/height`) подбираются так, чтобы вместить всю раскладку (с минимальными границами по ширине/высоте).

Генератор UML-диаграмм классов из Python-кода

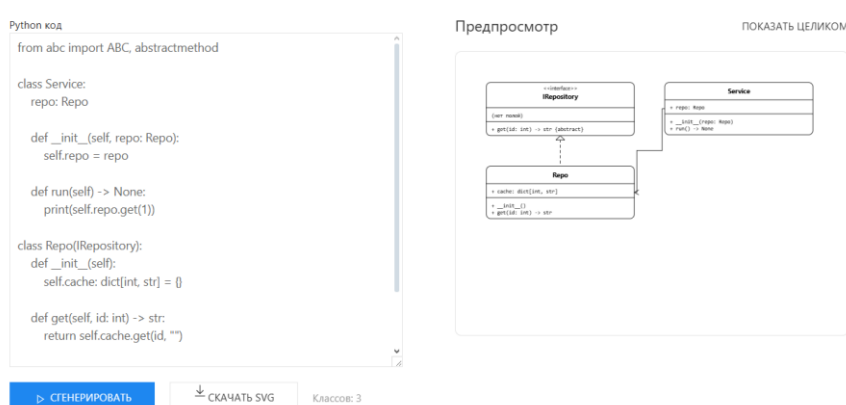


Рисунок 5. – Результат работы генератора SVG-изображений

Было проведено тестирование приложения на нескольких примерах:

1. Один класс во входных данных.

Был создан класс User для проверки работы приложения:

```
class User:
    name: str
    age: int

    def is_adult(self) -> bool:
        return self.age >= 18
```

Для атрибутов и метода is_adult также дополнительно указаны типы данных. Результатом является диаграмма, состоящая из одного класса (см. рисунок 6).

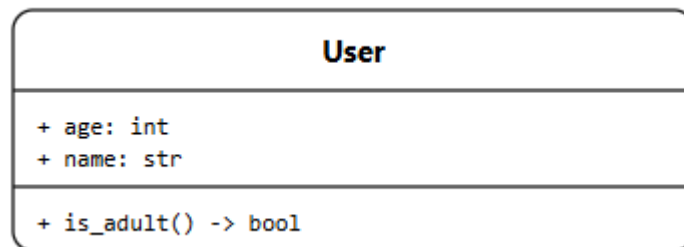


Рисунок 6. – Результат проведения первого теста работы приложения

2. Два класса, ассоциация через поле.

Были созданы 2 простых класса, один из которых содержит свойство, тип данных которого является объектом другого класса:

```
class Engine:
    power: int

class Car:
    engine: Engine
    model: str
```

Результатом является диаграмма классов с демонстрацией связи между ними (см. рисунок 7).

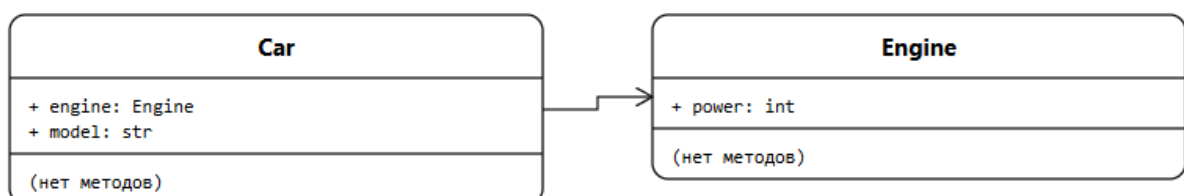


Рисунок 7. – Результат проведения второго теста работы приложения

3. Демонстрация наследования.

Были созданы 2 класса, один из которых наследует свойства и методы другого:

```
class Animal:
    name: str

class Dog(Animal):
    def bark(self) -> None:
        pass
```

На выходе получилась диаграмма с демонстрацией связи наследования, при этом стоит отметить, что в самом классе нигде не уточняется информация о наследовании, так как такая информация не является компонентом диаграммы классов (см. рисунок 8).

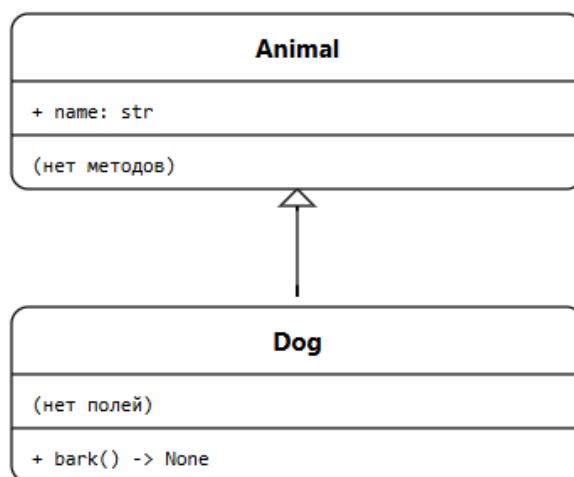


Рисунок 8. – Результат проведения третьего теста работы приложения

4. Реализация интерфейса.

Были разработаны интерфейс `Repository` и класс `FileRepository`, реализующий этот интерфейс:

```
from abc import ABC, abstractmethod

class Repository(ABC):
    @abstractmethod
    def save(self, data: str) -> None:
        pass

class FileRepository(Repository):
    def save(self, data: str) -> None:
        pass
```

В результате была построена диаграмма с визуализацией связи реализации между классом и интерфейсом (см. рисунок 9). Стоит отметить

направленность стрелки связи реализации: она всегда строится от класса к интерфейсу, что тоже важно было учесть.

Таким образом можно сделать вывод, что генератор строит связи реализации корректно, как и другие связи, рассмотренные ранее.

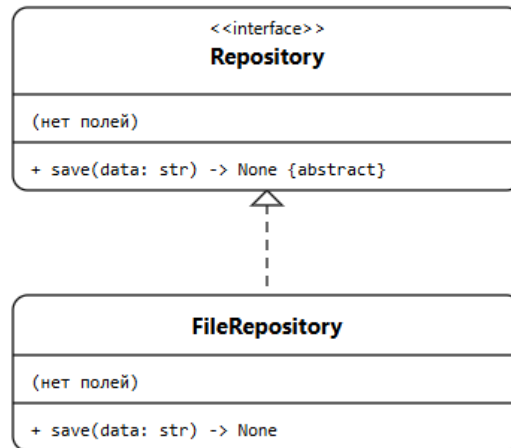


Рисунок 9. – Результат проведения четвёртого теста работы приложения

5. Демонстрация всех связей на диаграмме.

Были реализованы 4 небольших класса для демонстрации реализации интерфейсов, ассоциации и наследования классов:

```

from abc import ABC, abstractmethod

class Transport(ABC):
    @abstractmethod
    def move(self) -> None:
        pass

class Engine:
    power: int

class Car(Transport):
    engine: Engine

    def move(self) -> None:
        pass

class Driver:
    car: Car
  
```

В результате получилась диаграмма, на которой присутствовали неточности в визуализации стрелок, но сами связи были определены корректно (см. рисунок 10).

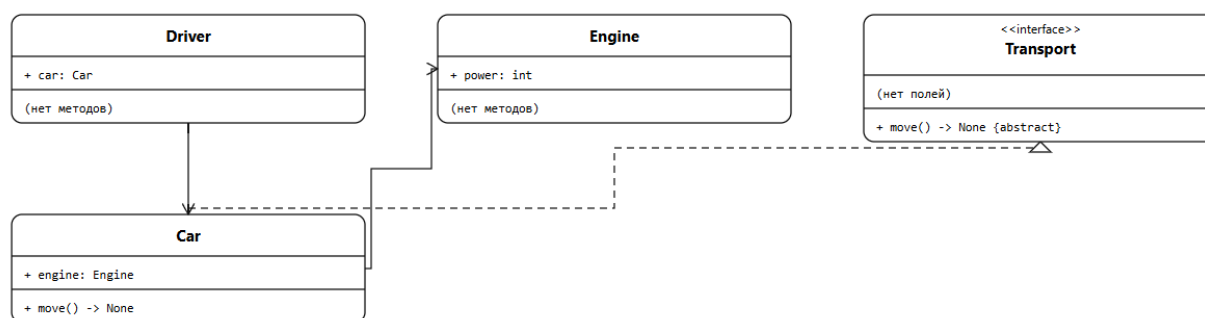


Рисунок 10. – Результат проведения пятого теста работы приложения

Таким образом, было разработано веб-приложение для построения диаграмм классов на основе программного кода, включающий в себя описание классов на языке программирования Python. Тестирование показало, что большинство функций приложение выполняет корректно и требуется доработка только в отрисовке связей между классами.

Приложение можно расширить, добавив возможность построения остальных видов связей, а также предоставить возможность работы с различными стереотипами классов (на данный момент поддерживаются только сущности и интерфейсы).

Разработанное веб-приложение можно использовать для обучения основам ООП и для демонстрации структуры классов приложения.

Библиографический список

1. Мухаметзянов Р. Р. Обучение объектно-ориентированному программированию // Информатика и образование. 2017. №. 7. С. 35-39.
2. Садулаева Б. С., Мунаев У. С. А. Обучение объектно-ориентированному программированию будущих бакалавров информатики // Теория и практика общественного развития. 2015. №. 20. С. 304-306.
3. Шатова А. Д. Использование UML-диаграмм для оптимизации бизнес-процессов // Научный аспект. 2024. Т.49. №6. С. 6282 -6286.
4. Сидорович Е. А. Применение объектно-ориентированного программирования в исследовании положения мусульман в социальном пространстве Королевства Кастилия и Леон (XIII-XV вв.) // Историческая информатика. 2025. №. 1 (51). С. 39-48.