



МИНОБРНАУКИ РОССИИ  
Федеральное государственное бюджетное  
образовательное учреждение высшего образования  
«МИРЭА — Российский технологический университет»  
РТУ МИРЭА

---

Институт искусственного интеллекта (ИИИ)  
Кафедра промышленной информатики (ПИ)

**ОТЧЁТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №1**

**по дисциплине**  
«Методы верификации и валидации характеристик программного  
обеспечения»


Выполнил студент группы ИКМО-05-23

 Миронов Д.С.

Принял

 Петренко А. А.

Москва 2024

 12.11.2024

## Динамический и статический анализ программного продукта

На основе изучения материала лекций по дисциплине «Методы верификации и валидации характеристик программного обеспечения» требуется выполнить следующее.

1. Проверить ранее сделанный Вами любой учебный проект любым выбранным Вами статическим анализатором (желательно несколькими). Сделать вывод об адекватности найденных ошибок.
2. Внести разные типы ошибок и проверить работу анализатора (-ов).
3. Сделать вывод о целесообразности статического анализа.
4. Предложить варианты написания своего анализатора для решения проблем из своего опыта, которые возникали слишком часто или имели негативные последствия.
5. Проанализировать учебный код любым выбранным Вами динамическим анализатором.
6. Внести ошибки и проверить адекватность работы динамического анализатора.
7. Оценить возможность создания автоматной модели по коду примера

### Листинг 1

```
from abc import ABC, abstractmethod

class Strategy(ABC):
    @abstractmethod
    def execute(self, data):
        print('data ', data)
        pass

class ConcreteStrategyA(Strategy):
    def execute(self, data):
        return sorted(data)

class ConcreteStrategyB(Strategy):
    def execute(self, data):
        return list(reversed(sorted(data)))

class Context:
    def __init__(self, strategy: Strategy):
        self._strategy = strategy

    def set_strategy(self, strategy: Strategy):
```

```

        self._strategy = strategy

    def execute_strategy(self, data):
        return self._strategy.execute(data)

data = [1, 2, 3, 6, 4, 5]

context = Context(ConcreteStrategyA())
print(context.execute_strategy(data)) # Вывод: [1, 2, 3, 4, 5]

context.set_strategy(ConcreteStrategyB())
print(context.execute_strategy(data)) # Вывод: [5, 4, 3, 2, 1]

```

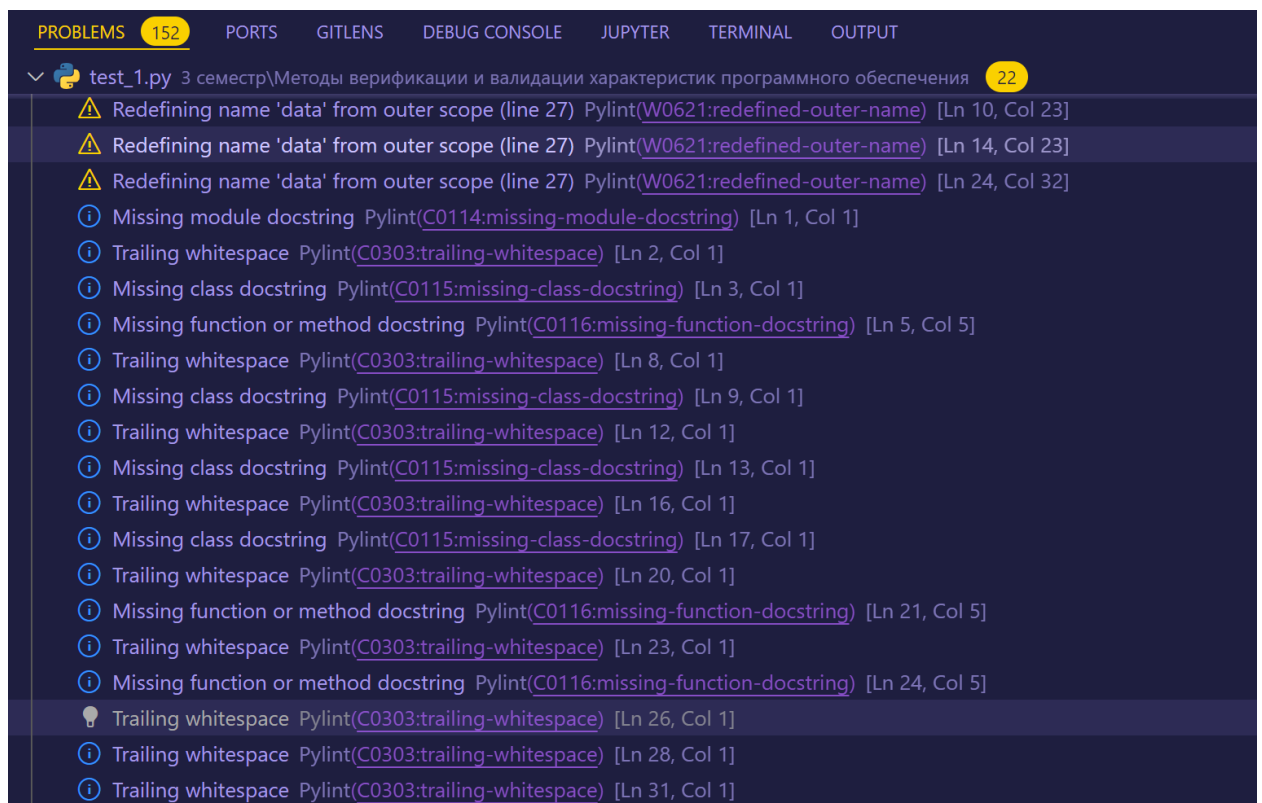


Рисунок 1 – найденные ошибки PyLint

Вывод найденных ошибок: выводимые ошибки говорят лишь об отсутствии комментариев и `generic` (указания типов данных) в коде.

Для примера, сделаем в коде синтаксическую ошибку.

```

3 семестр > Методы верификации и валидации характеристик программного обеспечения > test_1.py > ConcreteStrategyB
1  from abc import ABC, abstractmethod
2
3  class Strategy(ABC):
4      @abstractmethod
5      def execute(self, data):
6          print('data ',data)
7          pass
8
9  classs ConcreteStrategyA(Strategy):
10     def execute(self, data):
11         return sorted(data)
12
13     class ConcreteStrategyB(Strategy):
14         def execute(self, data):
15             return list(reversed(sorted(data)))
16
17     class Context:
18         def __init__(self, strategy: Strategy):

```

Рисунок 2 – синтаксическая ошибка в коде

Анализатор сразу же указывает на критическую ошибку

```

test_1.py 3 семестр\Методы верификации и валидации характеристик программного обеспечения 9
✗ Statements must be separated by newlines or semicolons Pylance [Ln 9, Col 8]
✗ Parsing failed: 'invalid syntax (<unknown>, line 9)' Pylint(E0001:syntax-error) [Ln 9, Col 9]
✗ Expected expression Pylance [Ln 9, Col 36]
✗ Unexpected indentation Pylance [Ln 10, Col 1]
✗ Unindent not expected Pylance [Ln 13, Col 1]
⚠ "classs" is not defined Pylance(reportUndefinedVariable) [Ln 9, Col 1]
⚠ "ConcreteStrategyA" is not defined Pylance(reportUndefinedVariable) [Ln 9, Col 8]
⚠ Type annotation not supported for this statement Pylance(reportInvalidTypeForm) [Ln 9, Col 36]
⚠ "ConcreteStrategyA" is not defined Pylance(reportUndefinedVariable) [Ln 29, Col 19]
main* ↺ ⚙️ ⚙️ Launchpad ⊗ 21 ⚠ 57 ⓘ 61 🐞 0

```

Рисунок 3 – работа анализатора

Статический анализ имеет ряд преимуществ:

- **Раннее обнаружение ошибок:** ошибки можно обнаружить еще до этапа тестирования, что экономит время и ресурсы.
- **Повышение качества кода:** анализаторы помогают поддерживать код в соответствии с установленными стандартами.
- **Повышение безопасности:** многие инструменты проверяют код на уязвимости.

Недостатки:

- **Ложные срабатывания:** иногда анализаторы выдают предупреждения на нормально работающий код.
- **Ограниченность:** статический анализ не всегда может выявить ошибки, которые проявляются только во время выполнения программы.

## **Варианты написания собственного анализатора**

Исходя из опыта, можно предложить разработку собственного анализатора, если у вас часто возникали определенные проблемы, которые существующие анализаторы не выявляли.

Примеры:

- **Анализатор логических ошибок:** если в вашем проекте часто возникают логические ошибки, можно создать инструмент, который проверяет корректность арифметических операций, работы с условными операторами и циклами.
- **Анализатор нарушений бизнес-логики:** если проект реализует сложные правила бизнеса, можно создать анализатор, который будет проверять соответствие кода этим правилам.

**Проанализировать учебный код любым выбранным Вами динамическим анализатором.**

Для анализа кода можно использовать такой инструмент, как Pyinstrument (для Python), или даже отладчики, такие как gdb. Этот анализ помогает выявить утечки памяти, неинициализированные переменные и другие ошибки выполнения.



Добавим ошибки, например, рекурсивный вызов функции `e()`, чтобы вызвать переполнение стека:

```
def e():  
    time.sleep(1)  
    e()
```

Рисунок 5 – работа динамического анализатора

Теперь инструмент должен выявить, что программа падает из-за переполнения стека.

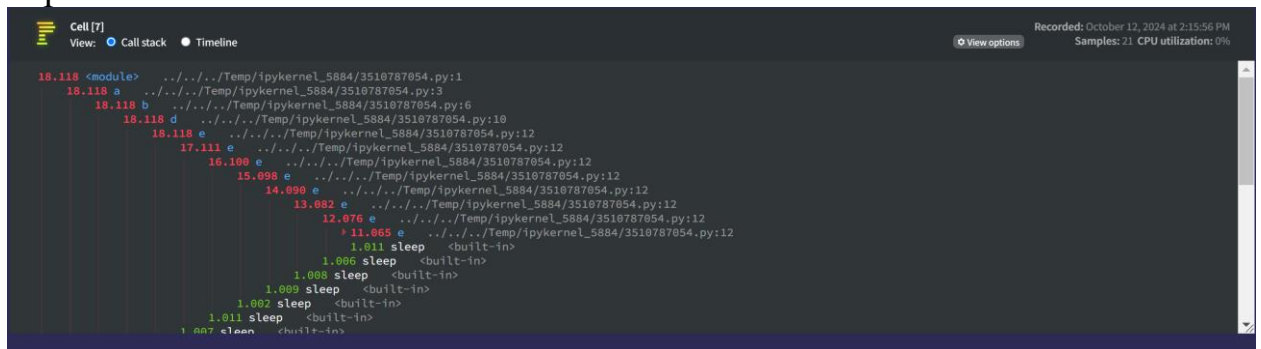


Рисунок 6 – переполнение стека

## Оценить возможность создания автоматной модели по коду примера

Для автоматной модели функции можно представить, как состояния, а вызовы функций — как переходы между ними.