ДИСЦИПЛИНА **Методы верификации и валидации характеристик программного обеспечения**

(полное наименование дисциплины без сокращений)

ИНСТИТУТ **информационных технологий**

КАФЕДРА **математического обеспечения и стандартизации информационных технологий**

(полное наименование кафедры)

ВИД УЧЕБНОГО МАТЕРИАЛА **Материалы для практических/семинарских занятий**

(в соответствии с пп.1-11)

ПРЕПОДАВАТЕЛЬ **Петренко Александр Анатольевич**

(фамилия, имя, отчество)

СЕМЕСТР **3, 2023-2024**

(указать семестр обучения, учебный год)

**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]
```

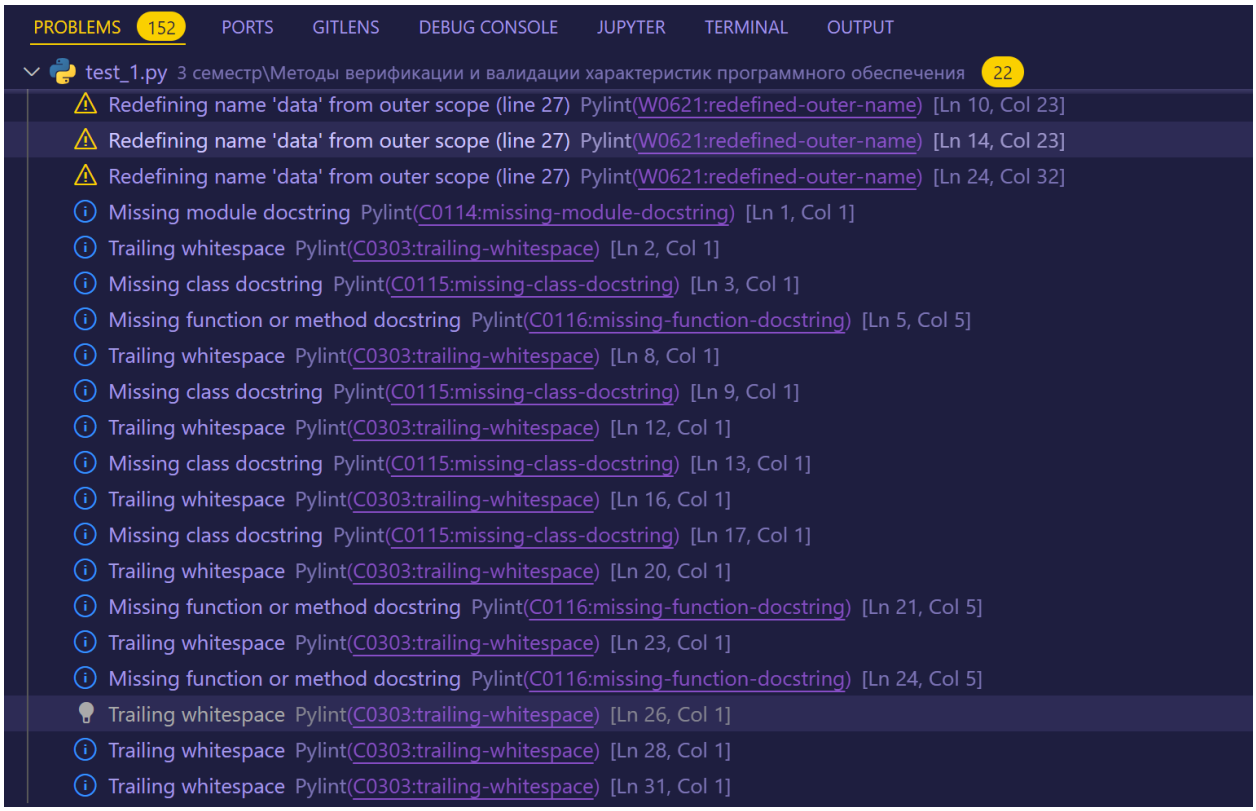


Рисунок 1 – найденные ошибки Pylint

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

Для примера, сделаем в коде синтаксическую ошибку.

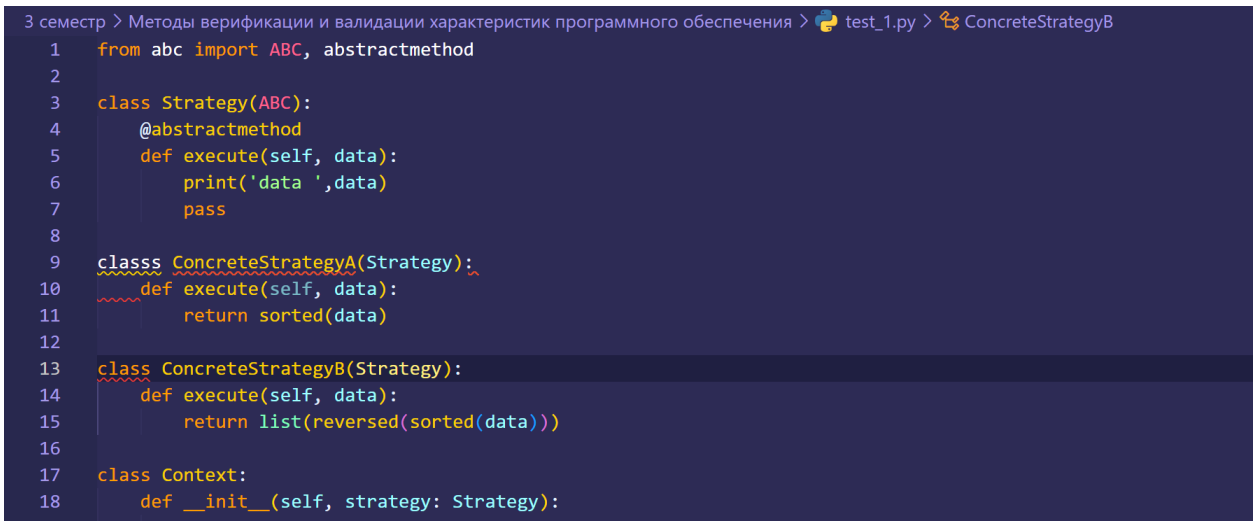


Рисунок 2 – синтаксическая ошибка в коде

Анализатор сразу же указывает на критическую ошибку

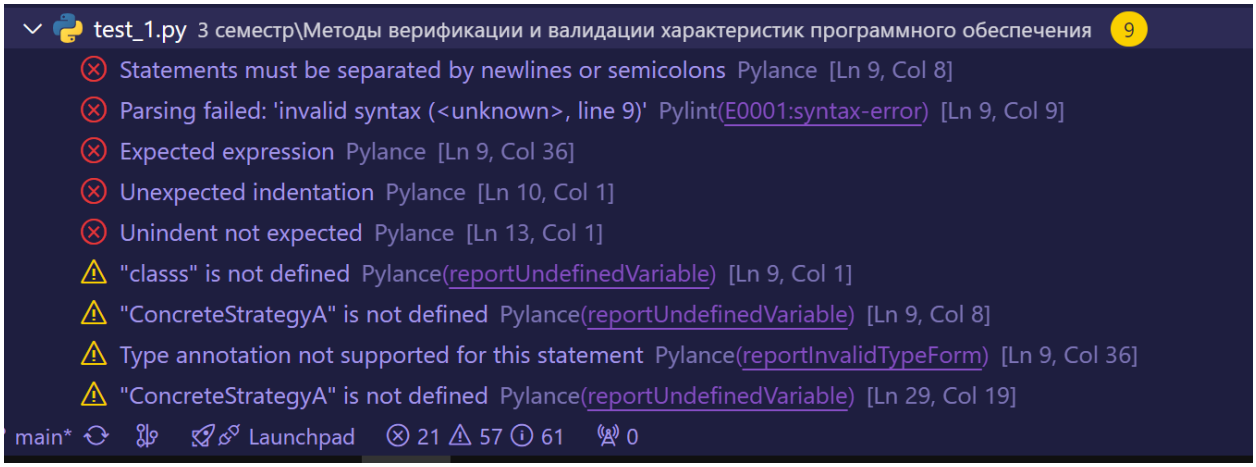


Рисунок 3 – работа анализатора

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.

```python
from pyinstrument import Profiler
import time

def a():
    b()
    c()

def b():
    d()

def c():
    d()

def d():
    e()

def e():
    time.sleep(1)

profiler = Profiler()
profiler.start()

a()

profiler.stop()
print(profiler.output_text(unicode=True, color=True))
```

✓ 2.0s

```
  _     ._   __/__   _ _  _  _ _/_   Recorded: 14:16:41  Samples:  3
 /_//_/// /_\ / //_// / //_'/ //     Duration: 2.010     CPU time: 0.000
/   _/                      v5.0.0

Profile at C:\Users\Дмитрий\AppData\Local\Temp\ipykernel_5884\4006021108.py:21

2.011 ZMQInteractiveShell.run_ast_nodes  IPython\core\interactiveshell.py:3274
└─ 2.009 <module>  ..\..\..\Temp\ipykernel_5884\4006021108.py:1
   └─ 2.009 a  ..\..\..\Temp\ipykernel_5884\4006021108.py:4
      ├─ 1.005 b  ..\..\..\Temp\ipykernel_5884\4006021108.py:8
      │  └─ 1.005 d  ..\..\..\Temp\ipykernel_5884\4006021108.py:14
      │     └─ 1.005 e  ..\..\..\Temp\ipykernel_5884\4006021108.py:17
      │        └─ 1.005 sleep  <built-in>
      └─ 1.004 c  ..\..\..\Temp\ipykernel_5884\4006021108.py:11
         └─ 1.004 d  ..\..\..\Temp\ipykernel_5884\4006021108.py:14
            └─ 1.004 e  ..\..\..\Temp\ipykernel_5884\4006021108.py:17
               └─ 1.004 sleep  <built-in>
```

Рисунок 4 – работа динамического анализатора

Этот анализ покажет, сколько времени каждая функция тратит на выполнение, что поможет обнаружить узкие места в производительности.

**Внести ошибки и проверить адекватность работы динамического анализатора.**

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.

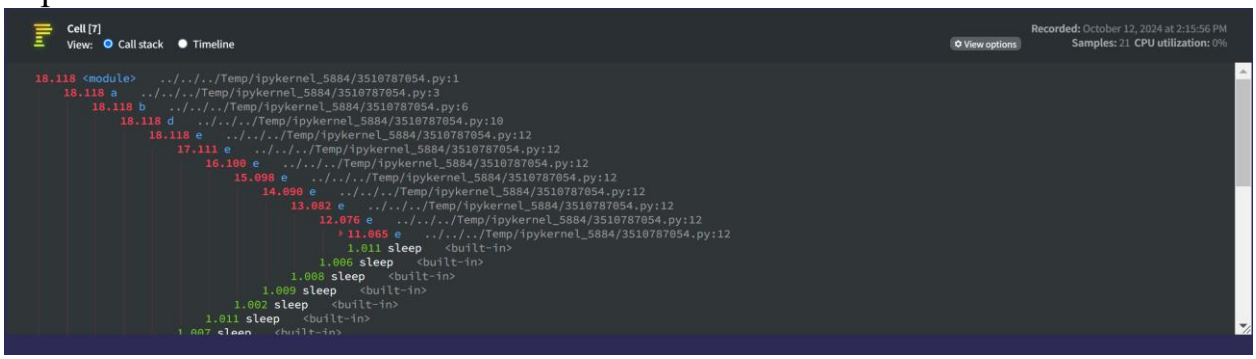


Рисунок 6 – переполнение стека

**Оценить возможность создания автоматной модели по коду примера**

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