|  |  |  |
| --- | --- | --- |
| МИНОБРНАУКИ РОССИИ | МИНОБРНАУКИ РОССИИ | МИНОБРНАУКИ РОССИИ |
| *Федеральное государственное бюджетное образовательное учреждение*  *высшего образования*  ***«МИРЭА – Российский технологический университет»***  **РТУ МИРЭА** | *овательное учреждение*  *высшего образования*  ***«МИРЭА – Российский технологический университет»***  **РТУ МИРЭА** | *Федеральное государственное бюджетное образовательное учреждение*  *высшего образования*  ***«МИРЭА – Российский технологический университет»***  **РТУ МИРЭА** |



Институт Информационных технологий (ИТ)

Кафедра Математического обеспечения и стандартизации информационных технологий (МОСИТ)

ПРАКТИЧЕСКАЯ РАБОТА №5

по дисциплине

**«Тестирование и верификация программного обеспечения»**

|  |  |
| --- | --- |
| **Выполнил:**  Студент группы ИКБО-36-22 | Утенков Ю. Ю. |
| **Проверил:** | ассистент Петрова А. А. |

Москва 2024 г.

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc183962162)

[ЧАСТЬ 1. СТАТИЧЕСКИЕ АНАЛИЗАТОРЫ 4](#_Toc183962163)

[ЧАСТЬ 2. ДИНАМИЧЕСКИЕ АНАЛИЗАТОРЫ 11](#_Toc183962164)

[ЗАКЛЮЧЕНИЕ 15](#_Toc183962165)

[ПРИЛОЖЕНИЕ 16](#_Toc183962166)

# **ВВЕДЕНИЕ**

**Цель работы:** освоить принципы и методы использования статических и динамических анализаторов кода, научиться выявлять ошибки и потенциальные уязвимости на ранних этапах разработки, а также повысить качество и надёжность программного обеспечения с помощью анализа результатов автоматизированных инструментов.

В рамках работы нужно проверить разработанные ранее приложения на двух различных языках программирования двумя типами анализаторов:

**Статический анализатор** — это инструмент, который анализирует исходный код программы без её выполнения. Он проверяет синтаксис, стиль, типизацию и потенциальные ошибки (например, некорректные операции или уязвимости), выявляя проблемы на ранних этапах разработки.

**Динамический анализатор** — это инструмент, который анализирует программу во время её выполнения. Он помогает обнаружить ошибки времени выполнения, проблемы с производительностью, утечки памяти и другие проблемы, которые могут возникнуть только при реальном запуске программы с реальными данными.

# **ЧАСТЬ 1. СТАТИЧЕСКИЕ АНАЛИЗАТОРЫ**

Для анализа было выбрано два разработанных ранее приложения:

1. На языке программирования Python, генерирующий мелодию по заданным аккордам
2. На языке программирования Java – программа, которая реализует метод Эйлера для численного решения обыкновенных дифференциальных уравнений ОДУ.

Для начала проверим приложение на питоне (рис. 1-3). Pylint, mypy, Bandit – ищут типичные ошибки, например, несоответствие нужному синтаксису и ошибки стилизации кода. Pylint обнаружил ошибки потерянных модулей и т.д, что не обнаружили остальные два.

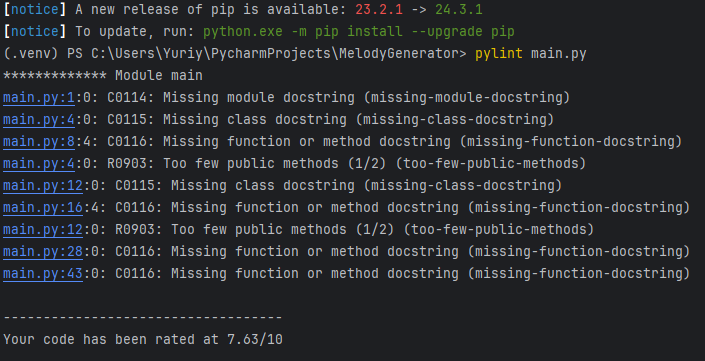


Рисунок 1 – Статический анализатор pylint

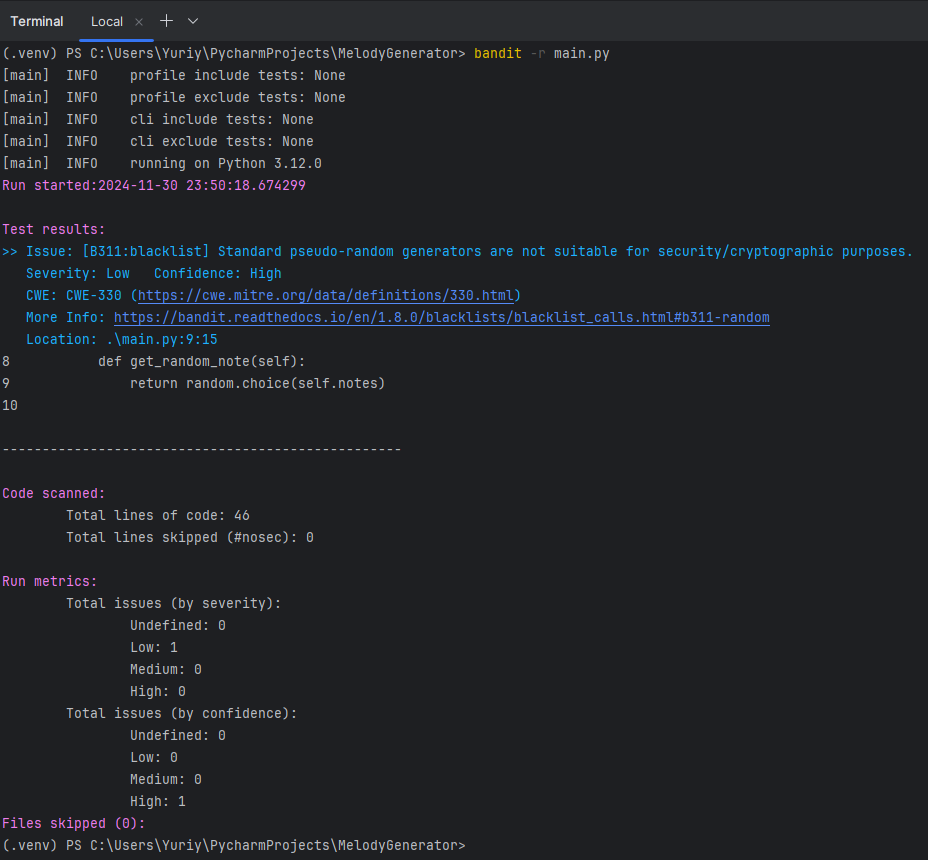


Рисунок 2 – Статический анализатор bandit

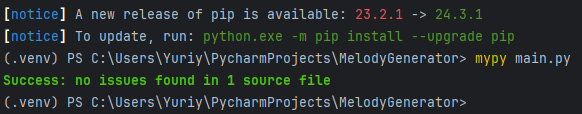
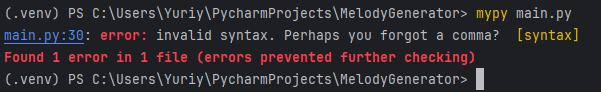
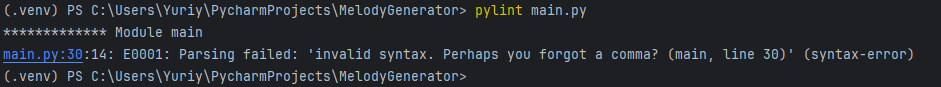


Рисунок 3 – Статический анализатор mypy

Далее добавим в код пару ошибок и проверим код анализаторами еще раз (рис. 4-6). Удалён ключ у словаря minor, лишние пробелы, нарушен синтаксис написания функции, удалена скобка в одном месте и нарушен стиль – использовали заглавную букву для переменной.

 Рисунок 4 – Статический анализатор mypy

 Рисунок 5 – Статический анализатор pylint

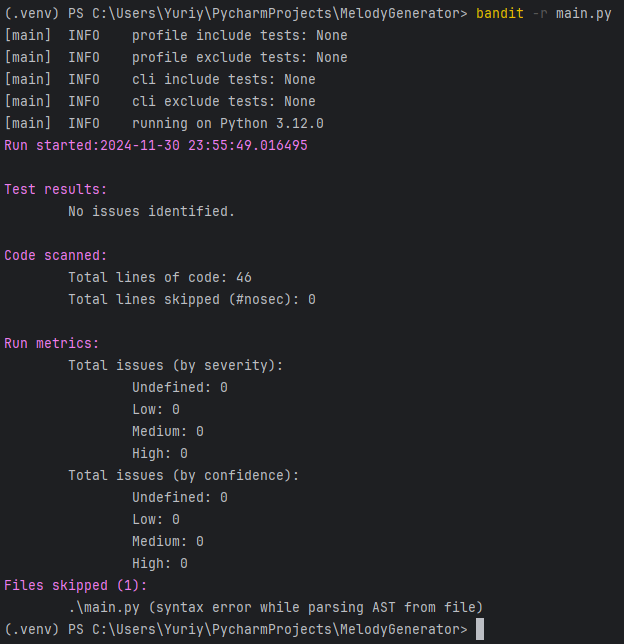


Рисунок 6 – Статический анализатор bandit

Как можно заметить, mypy и pylint прекратили анализ уже на не закрытой скобке, а bandit показал, что в файле просто присутствует синтаксическая ошибка.

Далее выполним анализ Java-кода. Для этого будем использовать встроенный в IntellijIDEA Code Inspection, CheckStyle (Google Checks) и PMD (рис. 7-10). Первый анализатор не нашел изначальные ошибки кода, которые обнаружили следующие два анализатора.

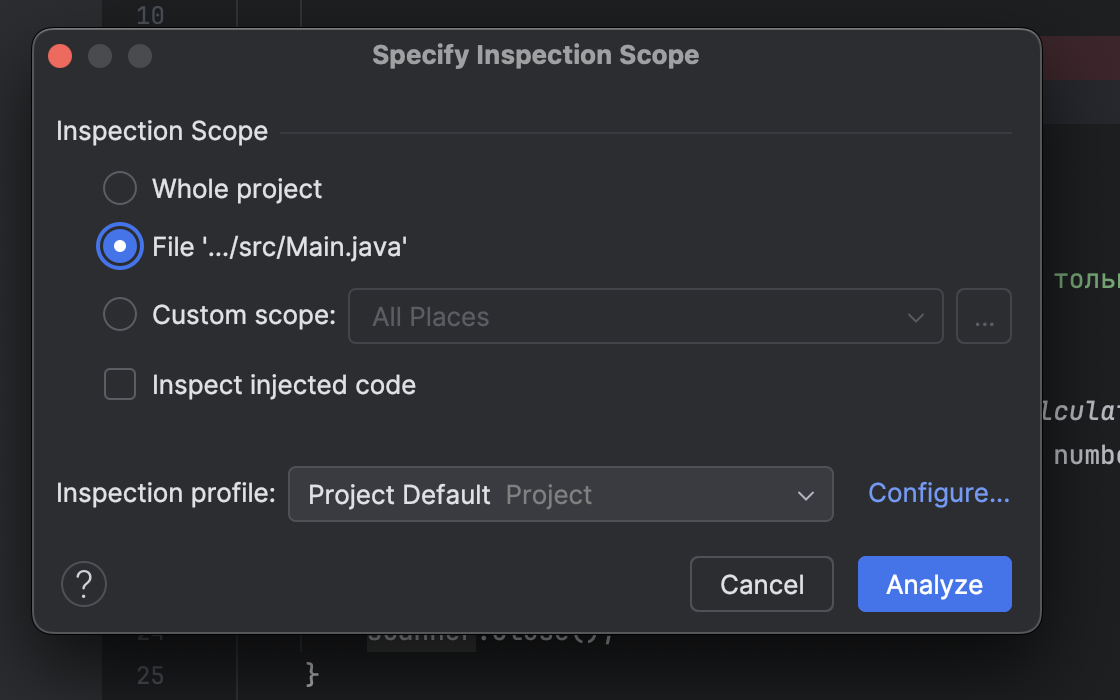


Рисунок 7 – Статический анализатор встроенный в IntellijIDEA

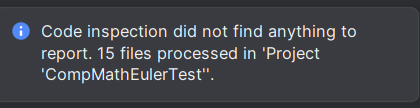


Рисунок 8 – Статический анализатор встроенный в IntellijIDEA не обнаружил ошибки

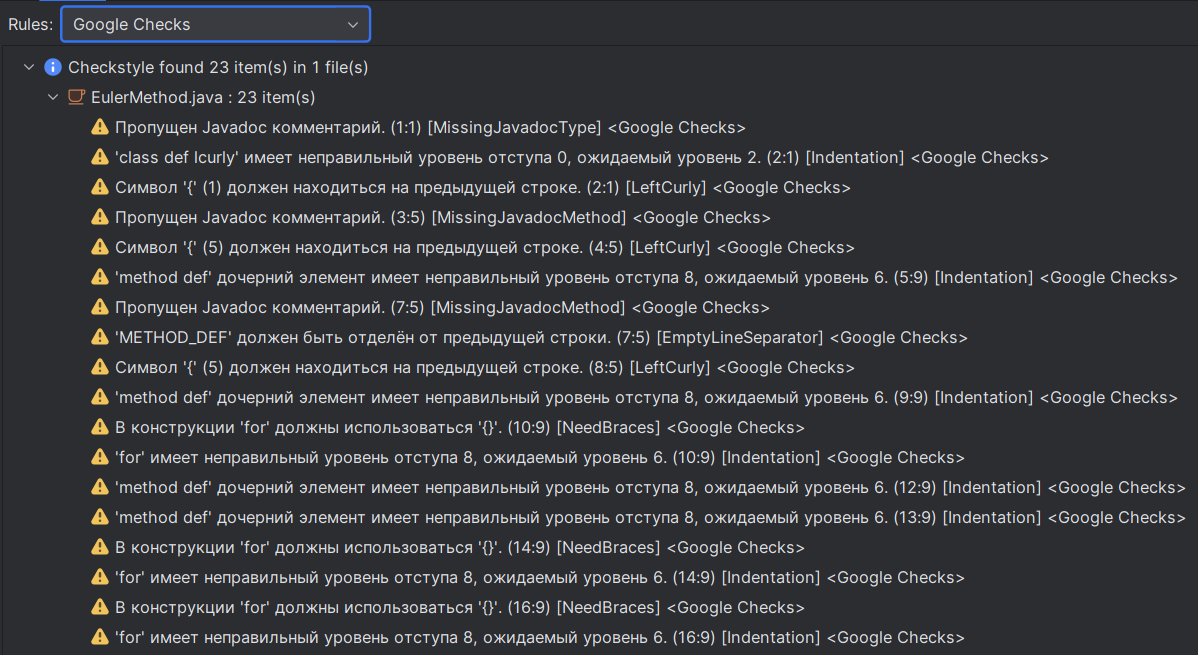


Рисунок 9 – Статический анализатор CheckStyle

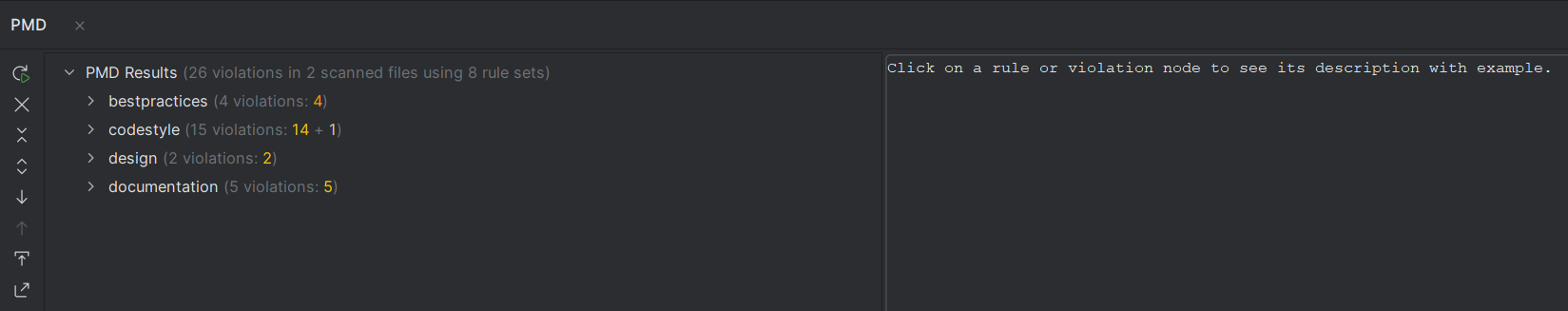


Рисунок 10 – Статический анализатор PMD

Добавим пару ошибок (неиспользуемая переменная, добавлен неиспользуемый метод, добавлен бесконечный цикл, который не позволит программе завершиться, необоснованное использование блока try исключений и 2 пробела между точкой и названием метода), которые должен обнаружить статический анализатор. Результаты анализа на рисунках 11-13. Как можно заметить, все ошибки были обнаружены всеми анализаторами.

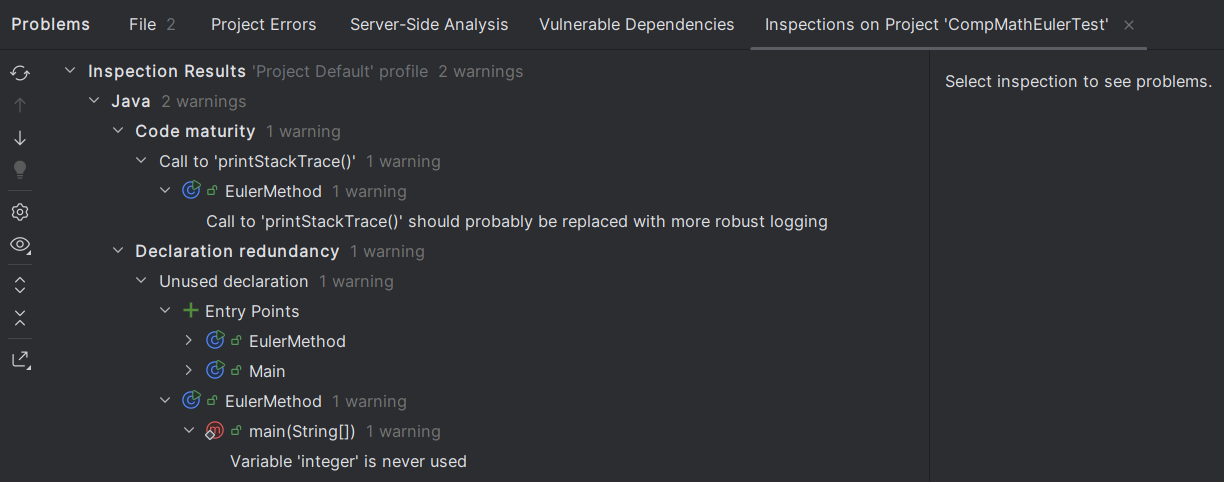


Рисунок 11 – Статический анализатор встроенный в IntellijIDEA

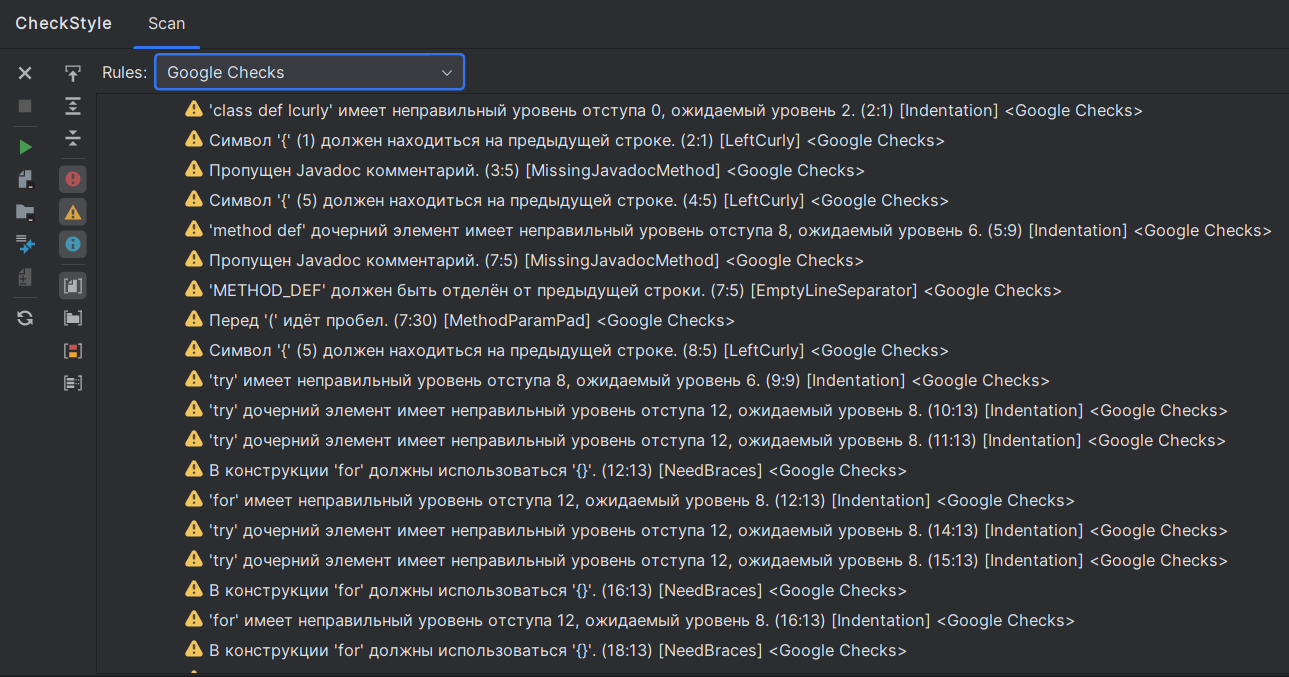


Рисунок 12 – Статический анализатор CheckStyle

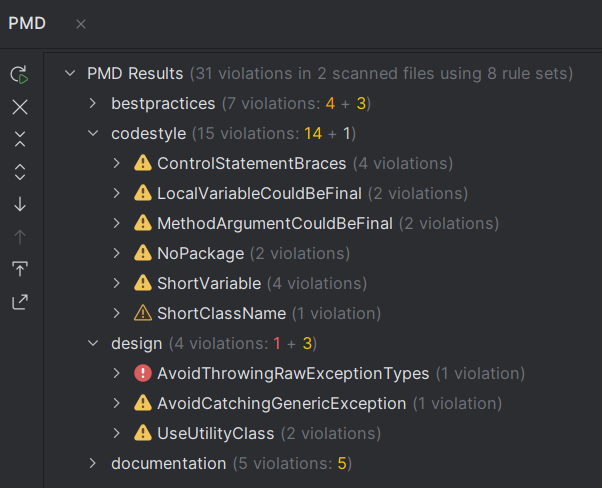


Рисунок 13 – Статический анализатор PMD

Статический анализ — это мощный инструмент, который позволяет улучшить качество, читаемость и безопасность кода на ранних этапах разработки. Однако он не заменяет динамический анализ (тесты, профайлинг), так как не выявляет ошибок, связанных с реальными данными или поведением программы во время выполнения.

**Минусы использования статического анализа** высока в следующих случаях:

* Большие проекты или работа в команде, где важно поддерживать единые стандарты кода.
* Программы, работающие с чувствительными данными, где ошибки безопасности критичны.
* Сложные проекты с потенциально большим количеством логических ошибок.

# **ЧАСТЬ 2. ДИНАМИЧЕСКИЕ АНАЛИЗАТОРЫ**

Для проведения анализа на генераторе мелодий динамическим анализатором был выбран cProfile, который мониторит время выполнения программы.

На рисунках 1-3 представлен код проверки этой программы.

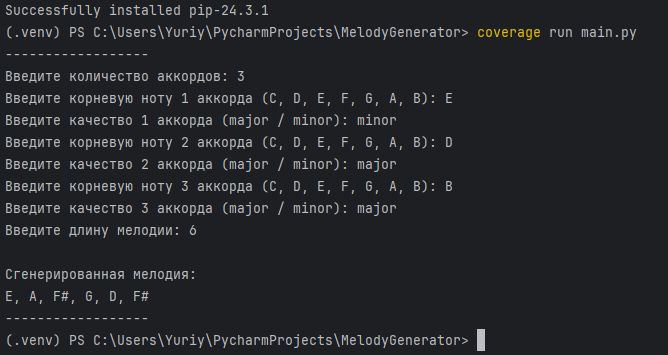
****

Рисунок 1 – Использование динамического анализатора coverage

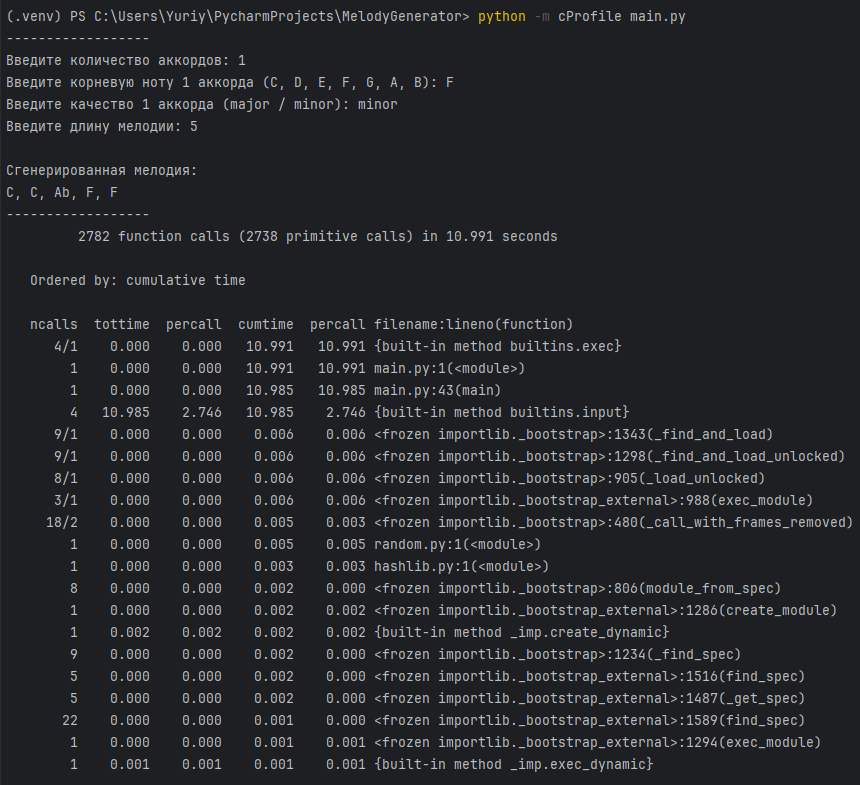
****

Рисунок 2 – Динамический анализатор cProfile

Добавим ошибки (логическая ошибка third, fifth = notes[root][quality] – присвоение одного и того же значения из словаря двум разным переменным), некорректный ключ у словаря нот: notes.

return 'Неизвестный диагноз')), которые могут быть обнаружены этими анализаторами и запустим проверку еще раз (рис. 3-4). Первый анализатор их обнаружил и прервал дальнейшую проверку: KeyError, после первого обнаружения, второй показал место, на котором остановилось выполнение программы и показал за какое время это произошло.

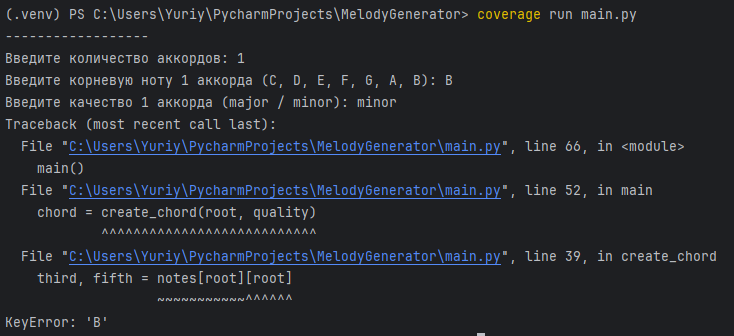
****

Рисунок 3 – Динамический анализатор Unittest c покрытием coverage

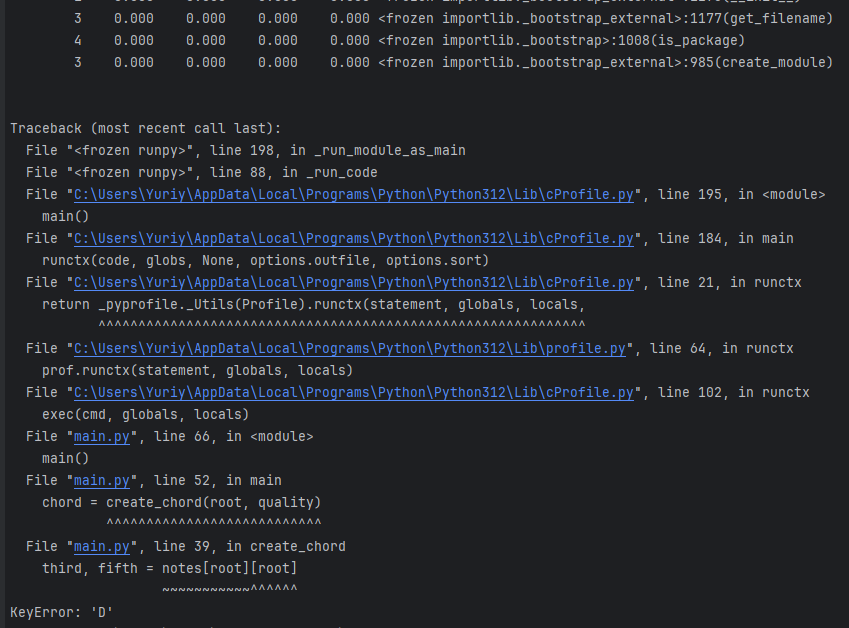
****

Рисунок 4 – Динамический анализатор cProfile

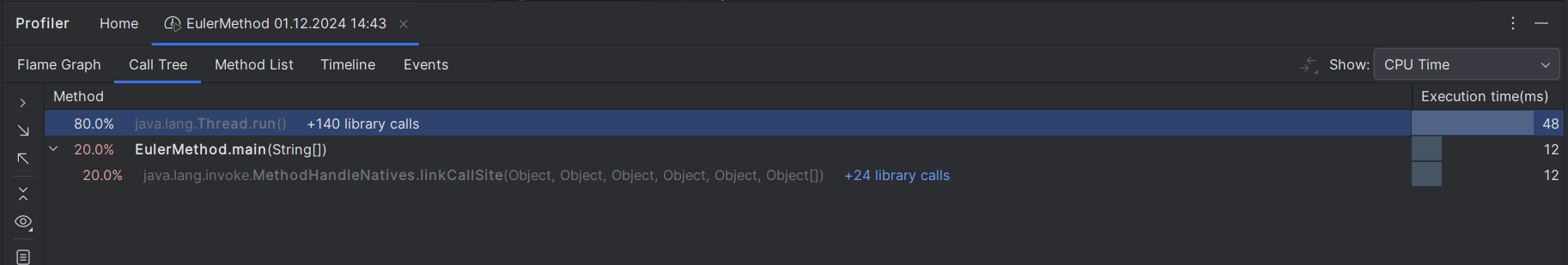
Теперь проверим Java-код, используя встроенный в IntellijIDEA Profiler (рисунок. 5). Был добавлен метод, намеренно создающий утечку и метод c бесконечным циклом и метод, который не используется в программе. На рисунках можно увидеть разницу во времени выполнения и в расходе памяти во время мониторинга. 

Рисунок 5 – Динамический анализатор встроенный в IntellijIDEA Profiler

Теперь воспользуемся VisualVM (рисунок 6). В интерфейсе программы увидим, что за ~одно и то же время выполнения программы с добавленными ошибками было потрачено куда больше памяти, чем при нормальной реализации программы.

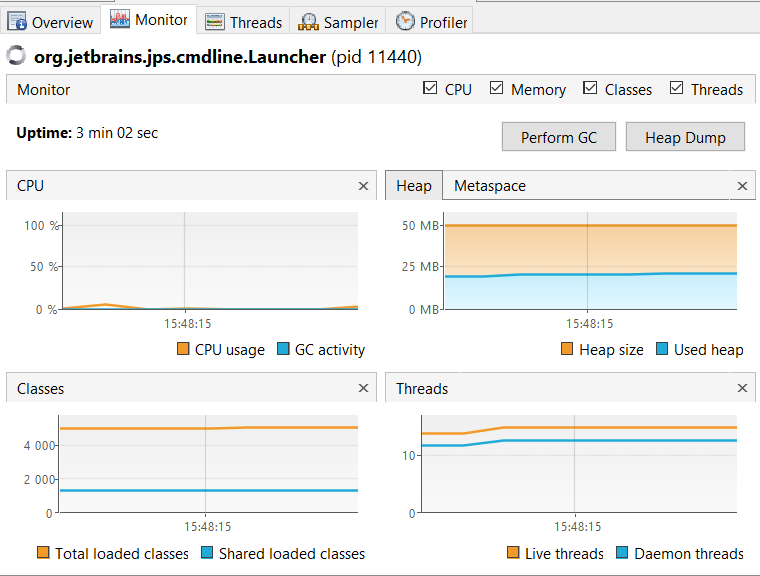
****

Рисунок 6 – Динамический анализатор VisualVM

Динамический анализ — это ключевой этап разработки программного обеспечения, который позволяет проверить, как программа работает в реальных условиях, выявить ошибки времени выполнения и улучшить производительность. Его целесообразность особенно высока в следующих случаях:

* Когда нужно выявить ошибки, которые невозможно обнаружить при статическом анализе, например, ошибки доступа к данным, деление на ноль, утечки памяти.
* Для проверки производительности программы, особенно в случае с большими объемами данных или сложными вычислениями.
* При тестировании и оптимизации кода для улучшения взаимодействия программы с реальной средой (файлы, база данных, сеть).
* Для обеспечения высокого качества и надежности программ через функциональные и интеграционные тесты.

# **ЗАКЛЮЧЕНИЕ**

Статический анализ кода помогает выявить ошибки и улучшить качество программного обеспечения на ранних этапах разработки. Он проверяет синтаксис, стиль и безопасность кода без фактического выполнения программы.

Динамический анализ проводится во время выполнения программы. Он позволяет обнаруживать ошибки времени выполнения, анализировать производительность и взаимодействие с системой.

Комбинируя оба типа анализа, можно достичь более высокого качества и надежности ПО. Статический анализ помогает на ранних этапах, а динамический - выявляет проблемы при фактическом выполнении программы.

# **ПРИЛОЖЕНИЕ**

Код к приложению «Генератор мелодий»:

|  |
| --- |
| import random class Chord:  def \_\_init\_\_(self, root, major\_third, perfect\_fifth):  self.notes = [root, major\_third, perfect\_fifth]  def get\_random\_note(self):  return random.choice(self.notes) class MelodyGenerator:  def \_\_init\_\_(self, chords):  self.chords = chords  def generate\_melody(self, length):  melody = []  current\_chord\_index = 0  for \_ in range(length):  note = self.chords[current\_chord\_index].get\_random\_note()  melody.append(note)  current\_chord\_index = (current\_chord\_index + 1) % len(self.chords)  return melody def create\_chord(root, quality):  notes = {  'C': {'major': ['E', 'G'], 'minor': ['Eb', 'G']},  'D': {'major': ['F#', 'A'], 'minor': ['F', 'A']},  'E': {'major': ['G#', 'B'], 'minor': ['G', 'B']},  'F': {'major': ['A', 'C'], 'minor': ['Ab', 'C']},  'G': {'major': ['B', 'D'], 'minor': ['Bb', 'D']},  'A': {'major': ['C#', 'E'], 'minor': ['C', 'E']},  'B': {'major': ['D#', 'F#'], 'minor': ['D', 'F#']}  }  third, fifth = notes[root][quality]  return Chord(root, third, fifth) def main():  print("------------------")  num\_chords = int(input("Введите количество аккордов: "))  chords = []  for i in range(num\_chords):  root = input(f"Введите корневую ноту {i + 1} аккорда (C, D, E, F, G, A, B): ").upper()  quality = input(f"Введите качество {i + 1} аккорда (major / minor): ").lower()  chord = create\_chord(root, quality)  chords.append(chord)  melody\_length = int(input("Введите длину мелодии: "))  generator = MelodyGenerator(chords)  melody = generator.generate\_melody(melody\_length)  print("\nСгенерированная мелодия:")  print(", ".join(melody))  print("------------------") if \_\_name\_\_ == "\_\_main\_\_":  main() |

Код программы на java для численного решения ОДУ:

|  |
| --- |
| package Various.CompMath;  public class Euler\_method {  public static double func(double x, double y)  {  return x \* x + 0.1 \* y;  }  public static void main(String[] args)  {  double[] x = new double[5];  for (int i = 0; i < x.length; i++)  x[i] = (double) i / 2;  double[] y = new double[5];  y[0] = 0.8;  for (int i = 1; i < y.length; i++)  y[i] = y[i - 1] + 0.5 \* *func*(x[i - 1], y[i - 1]);  for (int i = 0; i < y.length; i++)  System.*out*.println("Значение f(x[" + i + "], y[" + i + "]) = " + *func*(x[i], y[i]));  System.*out*.println();  for (int i = 0; i < y.length; i++)  System.*out*.println("Значение x[" + i + "] " + x[i] + "\t\tЗначение y[" + i + "] " + y[i]);  } } |