МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА − Российский технологический университет»

РТУ МИРЭА .

Институт информационных технологий (ИИТ)
Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №4

по дисциплине «Тестирование и верификация программного обеспечения»

Тема: Анализаторы кода

***Название команды*:** *ИКБО-52-23* «Тяпки»

Состав команды: Субботина Ю. О.
Солонин Е. Д.
Саватеев С. Л.

Дата выполнения: 10 ноября 2025 г.

Москва 2025 г.

2. ОПИСАНИЕ ПРОЕКТОВ И МЕТОДОВ АНАЛИЗА

2.1. Описание проектов

Python проект: Система управления задачами

Функциональность:

- Создание, обновление и удаление задач
- Управление приоритетами и дедлайнами
- Фильтрация задач по статусу и приоритету
- Поиск просроченных задач
- Сохранение и загрузка задач из JSON файла
- Экспорт задач в текстовый формат
- Ведение истории операций и статистики

Структура проекта:

- task.py - классы Task, Priority, TaskStatus
- task_manager.py - управление задачами
- storage.py - работа с файловым хранилищем
- main.py - главный модуль с консольным интерфейсом

Особенности:

Использование Enum для типизации, Работа с датами и временем, JSON сериализация/десериализация, Обработка исключений

Java проект: Система управления библиотекой

Функциональность: Добавление книг в библиотеку, Выдача книг читателям, Возврат книг в библиотеку, Поиск книг по автору, Ведение истории выдачи , Сохранение и загрузка данных из файла, Логирование всех операций

Структура проекта:

- Book.java - класс для представления книги
- Library.java - управление библиотекой
- FileManager.java - работа с файловым хранилищем

- Main.java - главный класс с консольным интерфейсом

Особенности:

- Использование коллекций (HashMap, ArrayList)

- Работа с файлами (FileReader, FileWriter)

- Обработка исключений

- Логирование операций

2.2. Используемые анализаторы

Статический анализ

Python проект:

https://github.com/NeoShaverma/tivpo4/tree/main/python-project

- **Pylint** - комплексный анализатор стиля и ошибок

- **Flake8** - комбинация PyFlakes, pycodestyle и McCabe

- **SonarQube** - платформа для непрерывного контроля качества кода

Java проект:

https://github.com/NeoShaverma/tivpo4/tree/main/java-project

- **FindBugs** - статический анализатор багов

- **PMD** - анализатор исходного кода

- **SonarQube** - платформа для непрерывного контроля качества кода

Динамический анализ

Python проект:

- **cProfile** - профилировщик производительности

- **memory_profiler** - анализатор использования памяти

Java проект:

- **Java VisualVM** - инструмент для мониторинга и профилирования

- **Java Mission Control (JMC)** - инструмент для анализа производительности с JFR

2.3. Применённые методы анализа

Статический анализ:

- Анализ исходного кода без выполнения
- Проверка соответствия стандартам кодирования
- Поиск потенциальных ошибок и уязвимостей
- Анализ структуры кода и архитектуры

Динамический анализ:

- Профилирование производительности во время выполнения
- Мониторинг использования памяти
- Анализ утечек памяти
- Отслеживание исключений и ошибок выполнения
- Анализ метрик JVM (для Java)

3. РЕЗУЛЬТАТЫ АНАЛИЗА: ТАБЛИЦЫ И ГРАФИКИ

3.1. Статический анализ - результаты до внесения ошибок

Python проект

Файл	Pylint	Flake8	SonarQube	Всего проблем
task.py	4	1	1	6
task_manager.py	3	0	2	5
storage.py	2	0	3	5
main.py	4	1	3	8
Итого	**13**	**2**	**9**	**24**

Распределение по критичности:

Критичность	Количество	Процент
Высокая	2	8.3%
Средняя	10	41.7%
Низкая	12	50.0%

Java проект

Файл	FindBugs	PMD	SonarQube	Всего проблем

Book.java	1	1	2	4
Library.java	2	0	4	6
FileManager.java	2	0	3	5
Main.java	2	1	4	7
Итого	**7**	**2**	**13**	**22**

Распределение по критичности:

Критичность	Количество	Процент
Высокая	3	13.6%
Средняя	12	54.5%
Низкая	7	31.9%

3.2. Статический анализ - результаты после внесения 5 ошибок

Python проект

Ошибка	Тип	Pylint	Flake8	SonarQube	Обнаружено
#1: Деление на ноль	Логическая	Нет	Нет	Да	1/3
#2: Неиспользуемая переменная	Стандарты	Да	Да	Да	3/3
#3: Отсутствие проверки на None	Null pointer	Нет	Нет	Частично	0.5/3
#4: Отсутствие валидации индекса	Индексация	Нет	Нет	Да	1/3
#5: Неправильная обработка	Исключения	Нет	Нет	Частично	0.5/3

исключений					

Статистика обнаружения: - Обнаружено Pylint: 1/5 (20%) - Обнаружено Flake8: 1/5 (20%) - Обнаружено SonarQube: 3.5/5 (70%) - Обнаружено всеми: 1/5 (20%)

Java проект

Ошибка	Тип	FindBugs	PMD	SonarQube	Обнаружено
#1: Утечка ресурсов	Ресурсы	Да	Нет	Да	2/3
#2: NumberFormatException	Исключения	Нет	Нет	Да	1/3
#3: NullPointerException	Null pointer	Да	Нет	Да	2/3
#4: Отсутствие валидации	Валидация	Да	Частично	Да	2.5/3
#5: InputMismatchException	Исключения	Нет	Нет	Частично	0.5/3

Статистика обнаружения: - Обнаружено FindBugs: 3/5 (60%) - Обнаружено PMD: 0.5/5 (10%) - Обнаружено SonarQube: 4/5 (80%) - Обнаружено всеми: 0/5 (0%)

3.3. Динамический анализ - результаты до внесения ошибок

Python проект

Метрика	Значение	Статус
Память (MB)	16.1	Норма
Время выполнения (мс)	1.0	Норма
Исключения	0	Нет проблем
Утечки памяти	Нет	Норма

Java проект

Метрика	Значение	Статус
Heap Memory (MB)	8.5	Норма
CPU Usage (%)	2-5	Норма
GC Pause Time (ms)	<10	Норма
Исключения	0	Нет проблем
Утечки памяти	Нет	Норма

3.4. Динамический анализ - результаты после внесения 3 ошибок

Python проект

Метрика	До ошибок	После ошибок	Изменение
Память (MB)	16.1	26.5	+65%
Время выполнения (мс)	1.0	1.15	+15%
Исключения	0	1	Появились
Утечки памяти	Нет	Да	Обнаружена

Обнаружение ошибок: - Утечка памяти: Обнаружена memory_profiler - Деление на ноль: Обнаружено при выполнении - Обработка исключений: Частично (требует тестирования)

Java проект

Метрика	До ошибок	После ошибок	Изменение
Heap Memory (MB)	8.5	45.2	+432%
CPU Usage (%)	2-5	8-12	+140%
GC Pause Time (ms)	<10	25-30	+200%
Исключения	0	2	Появились
Утечки памяти	Нет	Да	Обнаружена

Обнаружение ошибок: - Утечка памяти: Обнаружена VisualVM и JMC - NullPointerException: Обнаружена VisualVM и JMC - Проблема производительности: Обнаружена JMC

4. СРАВНИТЕЛЬНЫЙ АНАЛИЗ РЕЗУЛЬТАТОВ

4.1. Статический анализ: до и после внесения ошибок

Python проект

До внесения ошибок: - Всего проблем: 24 - Высоких: 2 (8.3%) - Средних: 10 (41.7%) - Низких: 12 (50.0%)

После внесения 5 ошибок: - Всего проблем: 29 (24 + 5) - Обнаружено инструментами: 6 (1 полностью, 5 частично) - Эффективность обнаружения: 20-70% (зависит от инструмента)

Java проект

До внесения ошибок: - Всего проблем: 22 - Высоких: 3 (13.6%) - Средних: 12 (54.5%) - Низких: 7 (31.9%)

После внесения 5 ошибок: - Всего проблем: 27 (22 + 5) - Обнаружено инструментами: 8 (все 5 ошибок обнаружены хотя бы одним инструментом) - Эффективность обнаружения: 10-80% (зависит от инструмента)

4.2. Динамический анализ: до и после внесения ошибок

Python проект

До внесения ошибок: - Память: 16.1 MB (стабильно) - Время выполнения: 1.0 мс - Исключения: 0 - Утечки памяти: Нет

После внесения 3 ошибок: - Память: 26.5 MB (+65%) - Время выполнения: 1.15 мс (+15%) - Исключения: 1 (ZeroDivisionError) - Утечки памяти: Да (обнаружена memory_profiler)

Java проект

До внесения ошибок: - Heap Memory: 8.5 MB (стабильно) - CPU Usage: 2-5% - GC Pause Time: <10 ms - Исключения: 0 - Утечки памяти: Нет

После внесения 3 ошибок: - Heap Memory: 45.2 MB (+432%) - CPU Usage: 8-12% (+140%) - GC Pause Time: 25-30 ms (+200%) - Исключения: 2 (NullPointerException) - Утечки памяти: Да (обнаружена обоими инструментами)

4.3. Сравнительная таблица эффективности

Аспект	Статический анализ	Динамический анализ
Python: обнаружение ошибок	20-70%	67% (100% с тестированием)
Java: обнаружение ошибок	10-80%	100%

Утечки памяти	Частично (Java)	Отлично
Исключения	Частично	Отлично
Производительность	Нет	Отлично
Логические ошибки	Частично	Частично
Время анализа	Секунды-минуты	Минуты-часы
Нагрузка на систему	Нет	5-20% overhead

5. СРАВНЕНИЕ СТАТИЧЕСКОГО И ДИНАМИЧЕСКОГО АНАЛИЗА

5.1. Ложные срабатывания

Статический анализ

Python: - Pylint: 4 ложных срабатывания (23% от общего числа) - Требование docstring для **str** - Слишком мало публичных методов - Слишком много аргументов - Имя функции print_menu - Flake8: 1 ложное срабатывание (слишком длинные строки) - SonarQube: 2 ложных срабатывания (проверка на None, конфликт имён)

Java: - FindBugs: 1 ложное срабатывание (unread field) - PMD: 2 ложных срабатывания (unused assignment, hardcoded IP) - SonarQube: 2 ложных срабатывания (thread safety, magic numbers)

Общий процент ложных срабатываний: ~21% (12 из 56 обнаружений)

Динамический анализ

Python: - Ложных срабатываний не обнаружено - Все обнаруженные проблемы подтвердились

Java: - Ложных срабатываний не обнаружено - Все обнаруженные проблемы подтвердились

Вывод: Динамический анализ не имеет ложных срабатываний, так как анализирует реальное выполнение кода.

5.2. Пропущенные ошибки

Статический анализ

Python: - Деление на ноль: обнаружено только SonarQube (20% инструментов) - Проблемы с None: частично обнаружено SonarQube - Обработка исключений: частично обнаружено SonarQube

Java: - NumberFormatException: обнаружено только SonarQube - InputMismatchException: частично обнаружено SonarQube - Некоторые проблемы производительности: не обнаружены

Причины пропуска: - Логические ошибки требуют анализа условий выполнения - Проблемы с пользовательским вводом сложно анализировать статически - Некоторые ошибки проявляются только при специфических данных

Динамический анализ

Python: - Обработка исключений: обнаружена только при тестировании с повреждённым файлом - Требует покрытия всех сценариев выполнения

Java: - Все внесённые ошибки обнаружены - Пропусков не было

Причины пропуска (для Python): - Требует выполнения всех путей кода - Некоторые ошибки проявляются только при специфических условиях

5.3. Возможные причины различий в результатах

Статический анализ

1. **Разные алгоритмы анализа:**
    - Pylint фокусируется на стиле и конвенциях
    - Flake8 проверяет базовые правила
    - SonarQube использует сложный анализ потока данных
2. **Ограничения статического анализа:**
    - Не может проанализировать все пути выполнения
    - Сложность анализа логических условий
    - Зависит от качества аннотаций и типов
3. **Конфигурация инструментов:**
    - Разные правила и пороги
    - Настройка под проект влияет на результаты

Динамический анализ

1. **Зависимость от выполнения:**
    - Анализируются только выполненные пути
    - Требует тестовых сценариев
    - Зависит от входных данных
2. **Различия в инструментах:**
    - cProfile фокусируется на производительности
    - memory_profiler специализируется на памяти
    - VisualVM и JMC имеют разные возможности

3. **Окружение выполнения:**
    - Результаты зависят от JVM/Python версии
    - Влияние системных ресурсов
    - Различия в конфигурации

5.4. Взаимодополняемость методов

Статический анализ обнаруживает: - Проблемы стиля и конвенций (100%) - Неиспользуемые переменные (100%) - Потенциальные NullPointerException (частично) - Утечки ресурсов (частично, для Java)

Динамический анализ обнаруживает: - Утечки памяти (100%) - Исключения во время выполнения (100% для Java) - Проблемы производительности (100% для Java) - Реальные runtime ошибки

Вывод: Методы дополняют друг друга: - Статический анализ - для раннего обнаружения (до выполнения) - Динамический анализ - для runtime проблем (во время выполнения)

6. ВЫВОДЫ И РЕКОМЕНДАЦИИ

6.1. Целесообразность применения статического анализа

Преимущества: - Раннее обнаружение ошибок (до выполнения кода) - Быстрота анализа (секунды-минуты) - Интеграция в CI/CD - Экономическая эффективность (снижение стоимости исправления в 10-100 раз) - Образовательные преимущества (обучение лучшим практикам)

Ограничения: - Не обнаруживает все типы ошибок (особенно логические) - Ложные срабатывания (~21%) - Требует настройки под проект - Не заменяет тестирование

Рекомендации по применению: - Использовать на этапе разработки (pre-commit hooks) - Интегрировать в CI/CD для автоматической проверки - Комбинировать несколько инструментов для максимального покрытия - Настроить правила под специфику проекта - Приоритизировать ошибки, найденные несколькими инструментами

6.2. Целесообразность применения динамического анализа

Преимущества: - Обнаружение runtime ошибок (утечки памяти, исключения) - Анализ в реальных условиях выполнения - Точные метрики производительности - Детальная диагностика проблем

Ограничения: - Высокая нагрузка на систему - Необходимость настройки тестовой среды - Покрытие только выполненных путей кода - Требует времени на выполнение

Рекомендации по применению: - Использовать при каждом релизе - Мониторинг метрик в продакшене - Комбинировать с тестированием для полного покрытия - Настроить автоматические алерты при превышении порогов

6.4. Интеграция анализа в процессы разработки

Этап разработки

1. **Pre-commit hooks:** быстрый статический анализ (Pylint/Flake8 для Python, SpotBugs для Java), блокировка коммита при критических ошибках
2. **Локальная разработка:** интеграция анализаторов в IDE, мгновенная обратная связь

Этап тестирования

1. **CI/CD pipeline:** полный статический анализ при каждом коммите, динамический анализ при сборке релиза, автоматические отчёты
2. **Code Review:** анализ отчётов статического анализа, проверка исправления найденных проблем

Этап продакшена

1. **Мониторинг:** динамический анализ в продакшене (с ограниченным overhead), алерты при превышении порогов, регулярный анализ метрик
2. **Оптимизация:** анализ производительности на основе реальных данных, выявление узких мест

6.5. Выводы

1. **Статический анализ эффективен для:** раннего обнаружения ошибок (90% типичных дефектов), проверки стиля и конвенций, интеграции в процесс разработки, снижения стоимости исправления ошибок
2. **Динамический анализ эффективен для:** обнаружения runtime ошибок (67-100% в зависимости от языка), анализа производительности, обнаружения утечек памяти, диагностики проблем в реальных условиях
3. **Комбинированный подход:** статический анализ - для раннего обнаружения, динамический анализ - для runtime проблем, тестирование - для функциональности, максимальное покрытие дефектов достигается при использовании всех методов
4. **Рекомендации:** использовать оба типа анализа в процессе разработки, настроить инструменты под специфику проекта, регулярно анализировать и улучшать процесс