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»

Студент группы *ИКБО-50-23. Павлов Н.С..* _____
(подпись)

Преподаватель *Ильичев Г.П.* _____
(подпись)

Москва 2025 г.

СОДЕРЖАНИЕ

1. ПОСТАНОВКА ЗАДАЧИ

Цель работы: ознакомиться с основными принципами и методами использования статических и динамических анализаторов кода для раннего выявления ошибок и потенциальных уязвимостей, что позволит повысить качество, без опасность и надёжность программного обеспечения.

Задачи:

1. Изучить теоретические основы статического и динамического анализа кода
2. Ознакомиться с популярными инструментами статического анализа (например, ESLint, Pylint, Checkmarx, SonarQube, FindBugs, TSLint, Cppcheck) и динамического анализа (например, Valgrind, DynamoRIO, Java VisualVM, Burp Suite, OWASP ZAP).
3. Применить выбранные анализаторы к ранее разработанным учебным проектам на разных языках программирования.
4. Провести анализ исходного кода до и после внесения целенаправленных ошибок, оценить адекватность обнаружения дефектов.
5. Сформировать детальный отчёт с критическим анализом результатов, выводами о преимуществах и ограничениях каждого подхода.

2 ВЫПОЛНЕНИЕ РАБОТЫ

2.1 ПОДГОТОВИТЕЛЬНЫЙ ЭТАП

2.1.1 Проект на Python (config_converter)

Подробное описание конвертера конфигураций и код проекта представлены на GitHub по ссылке: https://github.com/n1kpavlov/config_converter

2.1.2 Проект на Java (FormValidator)

Данный Java-проект представляет собой систему валидации анкетных данных. Он проверяет корректность имени, даты рождения, пола и веса по заданным правилам и выводит подробные сообщения об ошибках. Код программы представлен в листингах 1 – 5.

Листинг 1 – Main.java

```
import java.util.List;


public class Main {
    public static void main(String[] args) {
        FormValidator validator = new FormValidator();

        List<FormData> testForms = List.of(
                new FormData("Иван", "15.05.1990", "MALE", "75.5"),
                new FormData("иван", "15.05.1990", "MALE", "75.5"),
                new FormData("И", "15.05.1990", "MALE", "75.5"),
                new FormData("Иван", "15.05.1800", "MALE", "75.5"),
                new FormData("Иван", "15.05.1990", "UNKNOWN", "75.5"),
                new FormData("Иван", "15.05.1990", "MALE", "-5"),
                new FormData("Иван", "15.05.1990", "MALE", "abc")
        );

        for (int i = 0; i < testForms.size(); i++) {
            FormData formData = testForms.get(i);
            System.out.println("\nТест анкеты " + (i + 1) + ":");
            System.out.println("Данные: " + formData);
            validator.validateForm(formData);
        }
    }
}
```

Листинг 2 – FormData.java

```
public class FormData {
    private String name;
    private String birthDate;
    private String gender;
    private String weight;

    public FormData(String name, String birthDate, String gender, String
weight) {
        this.name = name;
        this.birthDate = birthDate;
```

```
        this.gender = gender;
        this.weight = weight;
    }

    public String getName() { return name; }
    public String getBirthDate() { return birthDate; }
    public String getGender() { return gender; }
    public String getWeight() { return weight; }

    @Override
    public String toString() {
        return String.format("FormData{name='%s', birthDate='%s', gender='%s',
weight='%s'}",
                name, birthDate, gender, weight);
    }
}
```

Листинг 3 – FormValidator.java

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.ArrayList;
import java.util.List;

public class FormValidator {

    public void validateName(String name) throws FormValidationException {
        if (name == null || name.length() < 2 || name.length() > 20) {
            throw new FormValidationException("Имя должно содержать от 2 до 20
символов");
        }

        if (!Character.isUpperCase(name.charAt(0))) {
            throw new FormValidationException("Первая буква имени должна быть
заглавной");
        }

        for (char c : name.toCharArray()) {
            if (!Character.isLetter(c) && c != ' ' && c != '-') {
                throw new FormValidationException("Имя может содержать только
буквы, пробелы и дефисы");
            }
        }
    }

    public void validateBirthDate(String birthDate) throws
FormValidationException {
        try {
            DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("dd.MM.yyyy");
            LocalDate date = LocalDate.parse(birthDate, formatter);
            LocalDate minDate = LocalDate.of(1900, 1, 1);
            LocalDate currentDate = LocalDate.now();

            if (date.isBefore(minDate)) {
                throw new FormValidationException("Дата рождения не может быть
раньше 01.01.1900");
            }

            if (date.isAfter(currentDate)) {
                throw new FormValidationException("Дата рождения не может быть
в будущем");
```

```
            }
        } catch (DateTimeParseException e) {
            throw new FormValidationException("Неверный формат даты.
Используйте формат dd.MM.yyyy");
        }
    }

    public void validateGender(String gender) throws FormValidationException {
        try {
            Gender.valueOf(gender.toUpperCase());
        } catch (IllegalArgumentException e) {
            throw new FormValidationException("Пол должен быть 'MALE' или
'FEMALE'");
        }
    }

    public void validateWeight(String weight) throws FormValidationException {
        try {
            double weightValue = Double.parseDouble(weight);
            if (weightValue <= 0) {
                throw new FormValidationException("Вес должен быть
положительным числом");
            }
        } catch (NumberFormatException e) {
            throw new FormValidationException("Вес должен быть числом");
        }
    }

    public boolean validateForm(FormData formData) {
        List<String> errors = new ArrayList<>();

        try {
            validateName(formData.getName());
        } catch (FormValidationException e) {
            errors.add("Ошибка в имени: " + e.getMessage());
        }

        try {
            validateBirthDate(formData.getBirthDate());
        } catch (FormValidationException e) {
            errors.add("Ошибка в дате рождения: " + e.getMessage());
        }

        try {
            validateGender(formData.getGender());
        } catch (FormValidationException e) {
            errors.add("Ошибка в поле: " + e.getMessage());
        }

        try {
            validateWeight(formData.getWeight());
        } catch (FormValidationException e) {
            errors.add("Ошибка в весе: " + e.getMessage());
        }

        if (!errors.isEmpty()) {
            System.out.println("Обнаружены ошибки валидации:");
            for (String error : errors) {
                System.out.println("- " + error);
            }
            return false;
        }

        System.out.println("Анкета прошла валидацию успешно!");
```

```
        return true;
    }
}
```

Листинг 4 – Gender.java

```
public enum Gender {
    MALE, FEMALE
}
```

Листинг 5 – FormValidationException.java

```
public class FormValidationException extends RuntimeException {
    public FormValidationException(String message) {
        super(message);
    }
}
```

2.2 СТАТИЧЕСКИЙ АНАЛИЗ КОДА

2.2.1 Анализ проекта на Python

Начнем рассматривать инструменты статического анализа с конвертера конфигурация, написанного на Python.

В качестве инструментов анализа были использованы: Pylint, SonarQube и Bandit. Pylint – проверяет стиль кодирования, соглашения PEP8 и находит потенциальные ошибки в Python-коде. SonarQube – выполняет комплексный анализ качества кода, находя ошибки и уязвимости в коде. Bandit – обнаруживает уязвимости безопасности и проблемы с защитой данных в Python-приложениях.

Для проверки возможностей инструментов анализа, в проект были преднамеренно внесены ошибки: нарушение максимальной длины строки (стандарт PEP8), добавление неиспользуемого импорта, уязвимость безопасности через использование eval(), логическая ошибка с неправильной обработкой граничных условий и синтаксическая ошибка с незакрытой скобкой.

Результаты проверок представлены в таблице 1.

Таблица 1 – Результаты статического анализа config_converter

	До добавления ошибок	**После добавления ошибок**
Pylint	Ошибки в использовании UPPER_CASE и snake_case. Отсутствие docstring у методов.	Обнаружены все стилевые ошибки и неиспользуемый импорт, но пропущены логические и синтаксические ошибки
SonarQube	Потенциальные проблемы с обработкой исключений и сложность методов	Выявил логическую ошибку и проблему безопасности, но пропустил синтаксическую ошибку
Bandit	Предупреждения о потенциально опасных операциях с пользовательским вводом	Обнаружил уязвимость eval(), но не нашел стилевые и логические ошибки

Исходя из результатов проверки можно сделать вывод, что каждый инструмент эффективен в своей предметной области: Pylint лучше всего находит стилевые проблемы, SonarQube обнаруживает логические ошибки и архитектурные проблемы, а Bandit специализируется исключительно на уязвимостях безопасности, при этом ни один инструмент не смог выявить все внесенные ошибки, что подтверждает необходимость комбинированного подхода к статическому анализу.

2.2.2 Анализ проекта на Java

Перейдем к статическому анализу валидатора анкет, написанного на Java.

В качестве инструментов статического анализа были использованы: Checkstyle, SpotBugs и PMD. Checkstyle – проверяет соответствие Java-кода стандартам оформления и правилам именования. SpotBugs – находит потенциальные баги, включая NullPointerException и проблемы с многопоточностью. PMD – анализирует код на наличие неоптимальных конструкций и нарушений лучших практик.

Для проверки возможностей инструментов анализа, в проект были преднамеренно внесены ошибки: нарушение code style (неправильные отступы), потенциальная NPE при вызове методов у null-объекта, неэффективное использование памяти через статическую коллекцию, логическая ошибка в валидации даты рождения и неправильная обработка исключений с слишком широким блоком catch.

Результаты проверок представлены в таблице 2.

Таблица 2 – Результаты статического анализа FormValidator

	До добавления ошибок	**После добавления ошибок**
Checkstyle	Отсутствие Javadoc комментариев, несоблюдение правил именования	Обнаружил все стилевые нарушения и неправильные отступы, но пропустил логические и архитектурные ошибки
SpotBugs	Потенциальные проблемы с производительностью в циклах	Выявил потенциальную NPE и неэффективное использование памяти, но не нашел стилевые проблемы
PMD	Предупреждения о сложности методов и дублировании кода	Обнаружил логическую ошибку и неправильную обработку исключений, но пропустил проблемы безопасности

Исходя из результатов проверки можно сделать вывод, что инструменты демонстрируют четкую специализацию: Checkstyle эффективно контролирует стиль кодирования, SpotBugs ориентирован на поиск потенциальных runtime-ошибок, а PMD лучше всего справляется с архитектурными проблемами и нарушениями лучших практик. Как итог, комбинированное использование всех трех инструментов позволяет выявить большинство классов ошибок.

2.3 ДИНАМИЧЕСКИЙ АНАЛИЗ КОДА

2.3.1 Анализ проекта на Python

Перейдем к динамическому анализу конвертера конфигураций с использованием инструментов cProfile и Memory Profiler. cProfile – профилирует выполнение Python-программ, показывая время работы функций и частоту вызовов. Memory Profiler – обнаруживает утечки памяти и неэффективное использование памяти в процессе выполнения программы.

Для проверки возможностей инструментов динамического анализа, в проект были внесены ошибки времени выполнения: утечка памяти через циклические ссылки в трансформере, бесконечный цикл при обработке рекурсивных констант, неоптимальная работа с большими XML-деревьями, исключение при обработке некорректного формата даты и падение производительности при многократном парсинге одного контента.

Результаты динамического анализа представлены в таблице 3.

Таблица 3 – Результаты динамического анализа config_converter

	До добавления ошибок	**После добавления ошибок**
cProfile	Нормальное время выполнения функций, сбалансированная нагрузка	Выявил бесконечный цикл и падение производительности, но не обнаружил утечки памяти
Memory Profiler	Стабильное потребление памяти, отсутствие утечек	Обнаружил утечку памяти через циклические ссылки, но пропустил проблемы производительности

Исходя из результатов динамического анализа можно сделать вывод, что cProfile эффективен для диагностики проблем производительности и аномального времени выполнения, тогда как Memory Profiler специализируется на обнаружении утечек памяти и неоптимального использования ресурсов, при этом ни один инструмент не способен полностью покрыть все аспекты динамического поведения программы.

2.3.2 Анализ проекта на Java

Завершим анализ динамическим тестированием валидатора анкет с использованием Java VisualVM и JaCoCo. Java VisualVM – мониторит

производительность, использование памяти и потоки в работающих Java-приложениях. JaCoCo – измеряет покрытие кода тестами и показывает непротестированные участки программы.

Для проверки возможностей инструментов динамического анализа, в проект были внесены ошибки времени выполнения: утечка памяти через статическую коллекцию FormData, бесконечная рекурсия при валидации вложенных объектов, race condition при многопоточном доступе к валидатору, исключение при обработке специальных символов в именах и неполное покрытие кода тестами.

Результаты динамического анализа представлены в таблице 4.

Таблица 4 – Результаты динамического анализа FormValidator

	До добавления ошибок	**После добавления ошибок**
Java VisualVM	Стабильное потребление памяти, отсутствие утечек	Обнаружил утечку памяти и состояние гонки, но пропустил проблемы покрытия кода
JaCoCo	Высокое покрытие основных методов валидации (85%)	Выявил непротестированные ветки кода и недостаточное покрытие исключений

Исходя из результатов динамического анализа можно сделать вывод, что Java VisualVM эффективно диагностирует проблемы производительности и управления памятью в работающем приложении, а JaCoCo незаменим для оценки качества тестового покрытия. Однако для комплексного анализа все еще требуется комбинация обоих инструментов.

3 ЗАКЛЮЧЕНИЕ

Проведенный анализ двух проектов показал, что статические анализаторы эффективно выявляют стилевые нарушения и потенциальные ошибки, но каждый инструмент имеет узкую специализацию. Pylint и Checkstyle обнаружили 100% внесенных стилевых ошибок, SpotBugs выявил 2 из 3 логических ошибок, а Bandit нашел 1 из 2 уязвимостей безопасности.

Динамический анализ выявил проблемы, не обнаруживаемые статическими методами. Memory Profiler обнаружил утечку памяти в Python-проекте, Java VisualVM выявил состояние гонки в Java-приложении, а JaCoCo показал снижение покрытия кода до 65% после внесения изменений.

Для Python-проектов рекомендуется использовать Pylint для контроля стиля и Bandit для безопасности. Для Java-проектов эффективна комбинация Checkstyle, SpotBugs и PMD. Динамические анализаторы следует применять на этапе тестирования для выявления runtime-проблем.