

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б11-мм

# Анализ неустойчивости юнит-тестов, вызванной порядком запуска

***ШИШИН Кирилл Александрович***

Отчёт по производственной практике  
в форме «Производственное задание»

Научный руководитель:  
к. ф.-м. н., доц. Куликов Е. К.

Санкт-Петербург  
2025

# Оглавление

|                                                                                                            |           |
|------------------------------------------------------------------------------------------------------------|-----------|
| <b>1. Введение</b>                                                                                         | <b>3</b>  |
| <b>2. Постановка задачи</b>                                                                                | <b>6</b>  |
| <b>3. Обзор</b>                                                                                            | <b>7</b>  |
| 3.1. Существующие решения . . . . .                                                                        | 7         |
| 3.2. Технологии, используемые для разработки . . . . .                                                     | 9         |
| <b>4. Реализация</b>                                                                                       | <b>11</b> |
| 4.1. Анализатор ODFlaky тестов . . . . .                                                                   | 11        |
| 4.2. Модернизация Explyt Test Runtime модуля . . . . .                                                     | 12        |
| 4.3. Специализированный запрос к LLM . . . . .                                                             | 13        |
| 4.4. Проектирование интерфейса взаимодействия с анализатором и представления результатов анализа . . . . . | 14        |
| <b>5. Апробация</b>                                                                                        | <b>16</b> |
| 5.1. <i>Apache Dubbo</i> . . . . .                                                                         | 16        |
| 5.2. <i>WebFlux-R2DBC</i> . . . . .                                                                        | 19        |
| <b>Заключение</b>                                                                                          | <b>22</b> |
| <b>Список литературы</b>                                                                                   | <b>23</b> |

# 1. Введение

Тестирование программного обеспечения является важной и неотъемлемой частью процесса разработки. Однако, как и в любой компоненте программного обеспечения, в тестах могут допускаться ошибки: например, некорректная работа с многопоточностью, использование нестабильных внешних сервисов, отсутствие очистки общего для тестов состояния и т.п. Это может приводить к появлению так называемых моргающих тестов — тестов, результат выполнения которых недетерминирован при отсутствии изменений в тестируемом коде. Выявление и дальнейшее устранение таких тестов отнимает много сил и времени у программистов и тестировщиков. В связи с этим последние годы активно [1] разрабатываются решения для автоматического анализа моргающих тестов. Актуальность темы подтверждается в том числе и наличием постоянно пополняемого репозитория<sup>1</sup>, созданного специально для исследования моргающих тестов и содержащего уже более 7000 примеров таких тестов из реальных проектов (например, *Guava*<sup>2</sup>, *Gson*<sup>3</sup>, *Seata*<sup>4</sup>).

Несмотря на то количество исследований (см., например, обзорную работу [2]), которое уже было проведено, универсального инструмента, способного выявлять и устранять сразу все виды моргающих тестов, пока не существует. Большинство исследований направлено на изучение отдельных видов моргающих тестов, каждый из которых имеет свои уникальные причины возникновения и требует специального подхода к анализу [3], [4]. И среди всех этих видов, из-за чего им уделяется особое внимание, самыми распространенными в реальных проектах (согласно статистике [2], они составляют порядка 16% от всех моргающих тестов) являются поряdkозависимые моргающие (ODFlaky) тесты [5], [6], [7] — тесты, результаты выполнения которых определяются последовательностью запуска тестов.

Рассмотрим минималистичный пример тестового класса, содержаще-

---

<sup>1</sup>International Dataset of Flaky Tests. — URL: <https://github.com/TestingResearchIllinois/idoft>

<sup>2</sup>Guava: Google Core Libraries for Java. — URL: <https://github.com/google/guava>

<sup>3</sup>Gson: A Java serialization/deserialization library. — URL: <https://github.com/google/gson>

<sup>4</sup>Seata: Simple Extensible Autonomous Transaction Architecture. — URL: <https://github.com/apache/incubator-seata>

ro ODFlaky тест,

```
1 class CatTest {
2     private static final Cat cat = new Cat();
3
4     @Test
5     void initialNameTest() {
6         String initialCatName = cat.getCatName();
7
8         assertEquals("Cat", initialCatName);
9     }
10
11    @Test
12    void catRenameTest() {
13        cat.setCatName("Barsik");
14        String catName = cat.getCatName();
15
16        assertEquals("Barsik", catName);
17    }
18 }
```

где `Cat` — представленный ниже класс, имеющий поле `catName` и соответствующие ему методы `get` и `set`.

```
1 class Cat {
2     private String catName = "Cat";
3
4     public String getCatName() {
5         return catName;
6     }
7
8     public void setCatName(String catName) {
9         this.catName = catName;
10    }
11 }
```

Как можно видеть из результатов запусков некоторых последовательностей тестов из класса `CatTest`, представленных на рисунке 1, тест `initialNameTest` является ODFlaky, поскольку имеет разные результаты прохождения при запуске в одиночку и при последовательном запуске после `catRenameTest`. Проанализировав результаты запусков более подробно, можно дополнительно узнать, что причиной падения теста `initialNameTest` стало изменение значения поля `catName`, иницированное тестом `catRenameTest`.



Рис. 1: Результаты запусков тестов из класса `CatTest`

Информация о наборе тестов и, при возможности, о том, исполнение каких именно инструкций при выполнении этих тестов, привело к возникновению ODFlaky теста, стала бы ценным результатом для автоматического анализатора порядкозависимых тестов. А в условиях активного развития инструментов-ассистентов в тестировании, основанных на AI, эта информация также помогла бы повысить точность формирования запросов к большой языковой модели и увеличила бы вероятность получения качественных рекомендаций по устранению найденных ODFlaky тестов.

Одним из таких инструментов является Explyt Test<sup>5</sup>, разрабатываемый компанией Explyt в Санкт-Петербурге. Разработка инструмента началась в начале 2024 года, и уже к концу того же года состоялся его полноценный релиз, сопровождаемый презентациями на крупных конференциях Joker<sup>6</sup> [9] и HeisenBug<sup>7</sup> [10]. Ассистент совмещает в себе последние разработки в области AI с формальными методами, проводя анализ пользовательского кода перед формированием запросов в большую языковую модель. И поскольку Explyt Test позиционирует себя как инструмент всестороннего ухода за тестами, добавление модуля анализатора ODFlaky тестов стало бы отличным дополнением к уже существующим возможностям инструмента.

<sup>5</sup>Explyt Test: AI test generation. – URL: <https://explyt.com>

<sup>6</sup>Joker: Java conference for experienced developers. – URL: <https://jokerconf.com>

<sup>7</sup>HeisenBug: Conference on testing not just for testers – URL: <https://heisenbug.ru>

## 2. Постановка задачи

**Целью** данной работы является разработка основанного на AI подхода к анализу и устранению моргающих тестов, зависящих от порядка запуска, в инструменте Explyt Test.

Для достижения цели были выделены следующие задачи:

1. Исследовать предлагаемые подходы к анализу и устранению моргающих тестов, зависящих от порядка запуска, в научных публикациях и существующих инструментах
2. Разработать механизм выявления моргающего теста и приводящего его к морганию теста загрязнителя
3. Локализовать, если это возможно, инструкции в тесте загрязнителе, исполнение которых приводит к появлению моргающего теста
4. Сформировать запрос к LLM на основе полученной в процессе анализа тестов информации
5. Разработать способ представления данных, полученных в процессе анализа тестов, а также рекомендаций, предложенных LLM
6. Апробировать предложенное решение на наборе реальных проектов с моргающими тестами, зависящими от порядка запуска

## 3. Обзор

### 3.1. Существующие решения

На сегодняшний день проведен уже целый ряд исследований на тему выявления и устранения ODFlaky тестов. Обзорная статья [2], систематизирующая существующие подходы и достижения в этой области, стала основным источником поиска возможных методов решения поставленных задач.

Всю работу по выявлению и устранению ODFlaky тестов можно разделить на 3 части: поиск порядкозависимых тестов, анализ порядкозависимых тестов и их устранение. И если поиск таких тестов во всех исследованиях производится примерно по одной схеме: запуск тестов в разном порядке, то анализ и устранение имеют уже более разнообразные подходы.

#### 3.1.1. Анализ ODFlaky тестов

PolDet [8] — инструмент для обнаружения тестов, выполнение которых изменяет общее состояние системы и тем самым создает зависимости между тестами. Конечно, при использовании формальных методов для дальнейшего устранения ODFlaky тестов, информация обо всех изменяющихся во время выполнения теста состояниях ценна. Однако в контексте применения LLM этот подход может оказаться менее эффективным, при всей сложности его реализации, поскольку он не предоставляет конкретных подсказок для устранения ODFlaky тестов.

FlakyDoctor [5] для каждого ODFlaky теста находит все тесты, выполнение которых приводит к его морганию, а затем пытается локализовать причины возникновения моргания, анализируя stack trace запуска тестов для нахождения не прошедшего assert'а, собирая информацию обо всех статических полях тестового класса и методах для обновления состояний. Конечно, такая информация может быть полезной при формировании запроса к LLM, однако и она не предоставляет конкретных подсказок по устранению ODFlaky тестов.

### 3.1.2. Устранение ODFlaky тестов

Существует 3 известных инструмента для устранения ODFlaky тестов: iFixFlakies [6] и ODRRepair [7], построенные на применении формальных методов, а также FlakyDoctor, основанный на AI.

Согласно сравнению [5], проведенному разработчиками FlakyDoctor, их инструмент демонстрирует более высокую эффективность в устранении ODFlaky тестов, чем решения, основанные на формальных методах. На наборе из 332 ODFlaky тестов, FlakyDoctor смог исправить на 12% тестов больше, чем ODRRepair, и на 17% больше, чем iFixFlakies.

Однако, можно предположить, что эффективность FlakyDoctor может стать еще выше, при улучшении проводимого анализа ODFlaky тестов. В частности, инструмент не учитывает, что LLM было бы полезно знать о конкретных инструкциях, исполнение которых привело к появлению ODFlaky теста. Также FlakyDoctor, при формировании запроса к LLM, не учитывает важность добавления реализаций методов, вызываемых в тестах, но находящихся за пределами тестового класса.

## 3.2. Технологии, используемые для разработки

Выбор стека технологий был обусловлен интеграцией решения в продукт Explyt Test.

Языки реализации: Kotlin<sup>8</sup>, Java<sup>9</sup>.

В процессе разработки были использованы основные механизмы IntelliJ Platform<sup>10</sup>.

- **Actions**<sup>11</sup>. Механизм создания пользовательских команд в IntelliJ IDEA. Используется для добавления точки входа анализатора ODFlaky тестов.
- **Psi**<sup>12</sup>. Одна из ключевых частей архитектуры IntelliJ Platform, предоставляющая инструменты для работы с синтаксическим деревом проекта. Используется в реализации модуля анализатора ODFlaky тестов для работы с пользовательским кодом и взаимодействия с другими модулями Explyt Test.
- **User Interface Components**<sup>13</sup>. Механизм создания графического интерфейса в IntelliJ IDEA. Используется для представления результатов работы анализатора ODFlaky тестов.

При модернизации модуля инструментации пользовательских тестов — Explyt Test Runtime — использовались следующие технологии:

- **ASM**<sup>14</sup>. Фреймворк для манипуляций с байткодом. При реализации используется для конвертации массива байткода в абстракции ASM

---

<sup>8</sup>The Kotlin Programming Language. — URL: <https://kotlinlang.org>

<sup>9</sup>The Java Programming Language. — URL: <https://www.java.com>

<sup>10</sup>IntelliJ Platform. — URL: <https://plugins.jetbrains.com/docs/intellij/welcome.html>

<sup>11</sup>IntelliJ Platform, Actions. — URL: <https://plugins.jetbrains.com/docs/intellij/basic-action-system.html>

<sup>12</sup>IntelliJ Platform, Psi. — URL: <https://plugins.jetbrains.com/docs/intellij/psi.html>

<sup>13</sup>IntelliJ Platform, User Interface Components. — URL: <https://plugins.jetbrains.com/docs/intellij/user-interface-components.html>

<sup>14</sup>Java bytecode manipulation and analysis framework. — URL: <https://asm.ow2.io>

и обратно. Конвертация в абстракции ASM удобна для дальнейшей работы с библиотекой JacoDB.

- **JacoDB**<sup>15</sup>. Библиотека для работы с байткодом. При реализации используется для анализа и дальнейшей трансформации тестовых методов.

Для взаимодействия с LLM используется модуль Explyt AI-client. Формирование запроса происходит с использованием одной из реализаций интерфейса ExplytAiRequest.

---

<sup>15</sup>JacoDB. Fast and effective way to access and analyze java bytecode. – URL: <https://github.com/UnitTestBot/jacodb>

## 4. Реализация

В этом разделе приведена реализация, основанного на AI подхода к анализу и устранению ODFlaky тестов, в инструменте Explyt Test. В рамках работы был разработан сам анализатор порядкозависимых тестов, модернизирован модуль Explyt Test Runtime, использующийся для запуска и инструментации пользовательских тестов, сформирован специализированный запрос к LLM, а также разработан способ представления данных, получаемых во время работы анализатора.

### 4.1. Анализатор ODFlaky тестов

Весь анализ порядкозависимых тестов можно разделить на 3 основных этапа: поиск порядкозависимых тестов, нахождение тестов, приводящих к появлению порядкозависимых тестов, а также нахождение конкретной инструкции, исполнение которой приводит к появлению морганий. Схема работы анализатора приведена на рисунке 2.

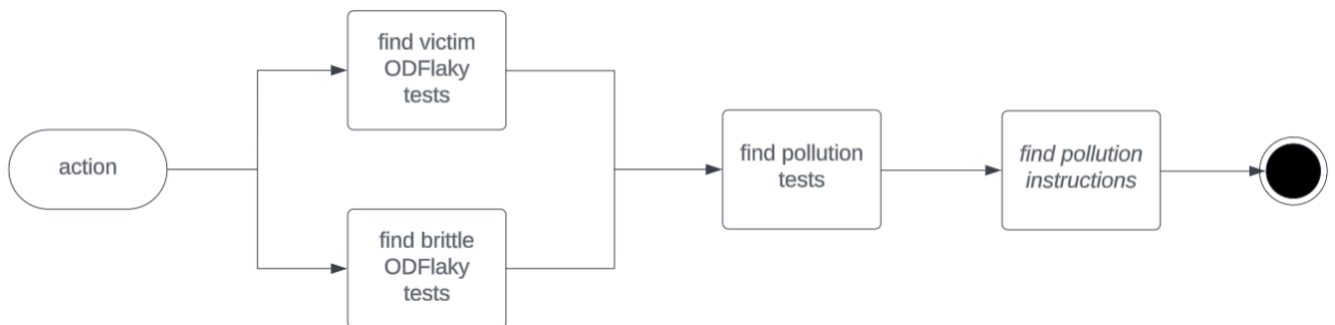


Рис. 2: Схема работы анализатора порядкозависимых тестов

#### 4.1.1. Поиск порядкозависимых тестов

На этапе поиска порядкозависимых тестов важно учитывать различные виды таких тестов:

1) Victim ODFlaky: успешно выполняется при запуске в одиночку, но падает при совместном запуске с каким-то другим тестом.

2) Brittle ODFlaky: падает при запуске в одиночку, однако успешно выполняется при совместном запуске с каким-то другим тестом.

Поскольку перебор и запуск всех возможных комбинаций тестов оказываются слишком долгими по времени при росте количества тестов в классе, было принято решение искать ODFlaky тесты, проявляющиеся при порядке, заданном тестовым фреймворком.

Таким образом, запуская сначала все тесты вместе, а затем каждый тест поодиночке, для каждого теста можно определить, является ли он ODFlaky и, если является, его вид.

#### **4.1.2. Поиск загрязняющих тестов**

На следующем этапе анализатор приступает к поиску загрязняющих тестов — тестов, исполнение которых приводит к появлению ODFlaky.

Для каждого найденного на предыдущем этапе ODFlaky теста, мы знаем набор тестов, которые запускались перед ним. В этом случае перебор и запуск всех возможных наборов тестов, запускаемых перед ODFlaky, также будут слишком долгими по времени. Однако, согласно исследованиям, для появления ODFlaky зачастую хватает одного загрязняющего теста, поэтому для их поиска мы по отдельности запускаем каждый тест из набора с ODFlaky.

#### **4.1.3. Поиск загрязняющих инструкций**

Последним шагом работы анализатора является поиск конкретной инструкции, исполняемой во время выполнения загрязняющего теста и приводящей к появлению ODFlaky теста. Поиск такой инструкции было принято осуществлять ограничивая количество исполняемых в загрязняющем тесте инструкций при совместном запуске с ODFlaky. Таким образом с использованием бинарного поиска, просматривая результат выполнения ODFlaky на каждой итерации алгоритма, получается найти конкретную инструкцию.

### **4.2. Модернизация Explyt Test Runtime модуля**

Запуск пользовательских тестов в Explyt Test происходит с использованием IntelliJ Platform Execution API. Также в Explyt Test уже

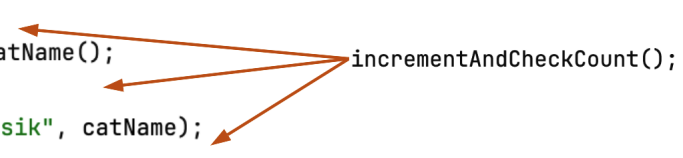
реализован модуль инструментации — Explyt Test Runtime, который помогает собирать trace исполнения пользовательских тестов.

В рамках работы над анализатором моргающих тестов, в Explyt Test Runtime был добавлен механизм позволяющий контролировать количество исполняемых во время выполнения теста инструкций.

#### 4.2.1. Ограничение количества исполняемых инструкций

Механизм ограничения количества инструкций, исполняемых во время выполнения пользовательского теста, основан на счетчике количества исполненных инструкций. В ходе инструментации байт-кода пользовательского приложения, после каждой инструкции добавляется дополнительная инструкция вызова метода, увеличивающего счетчик и, при достижении лимита, завершающего выполнение теста.

```
@Test
void catRenameTest() {
    cat.setCatName("Barsik");
    String catName = cat.getCatName();
    assertEquals(expected: "Barsik", catName);
}
```



### 4.3. Специализированный запрос к LLM

Формирование запросов к LLM в Explyt Test происходит с использованием Explyt AI-client. В рамках работы был сформирован специализированный запрос к LLM направленный на устранение ODFlaky тестов.

Сформированный запрос включает в себя следующие части:

1) Контекст приложения. Для ODFlaky тестов в запрос добавляется код тестируемого класса, тестового класса, а также всех методов, вызываемых во время выполнения ODFlaky и загрязняющего тестов.

2) Данные, найденные анализатором. В запрос попадает также информация, полученная в ходе анализа порядкозависимого теста: ODFlaky

тест, загрязняющий тест, загрязняющая инструкция, stack trace выполнения тестов.

3) Промпт, направленный на объяснение и устранение ODFlaky тестов, учитывающий контекст приложения и данные, полученные из анализатора.

#### 4.4. Проектирование интерфейса взаимодействия с анализатором и представления результатов анализа

Поскольку анализатор интегрируется в продуктовый инструмент Explyt Test, важно было разработать удобный для пользователя интерфейс взаимодействия с анализатором и наглядный способ представления результатов анализа.

Весь пайплайн работы анализатора был разделен на 2 этапа:

1) Анализ ODFlaky тестов и представление результатов анализа. Для более удобного представления информации принято решение показывать только первый найденный ODFlaky тест. Мотивация такого решения также заключается в том, что зачастую появление нескольких ODFlaky тестов в классе связано с одной ошибкой (напр., неправильно очищенный контекст приложения перед исполнением очередного теста). Разработанный способ представления результатов анализа приведен на рисунке 3.

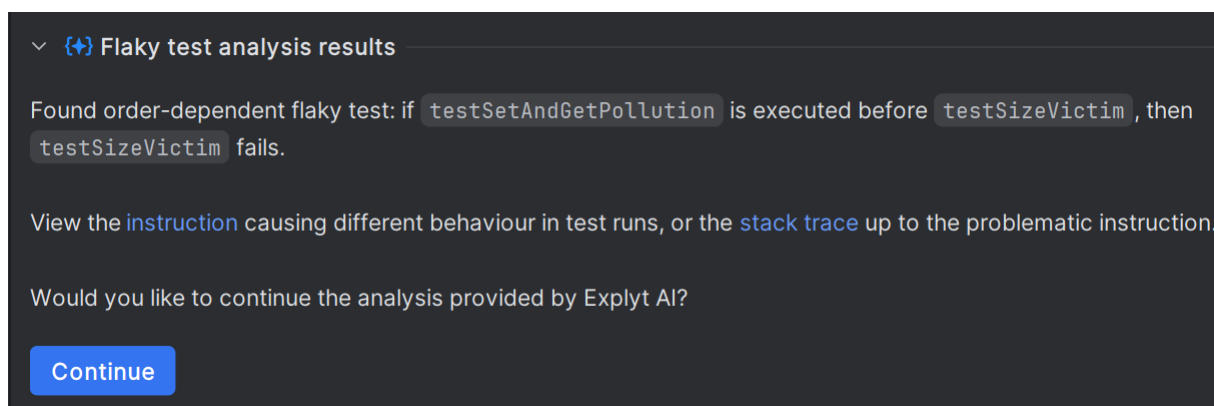
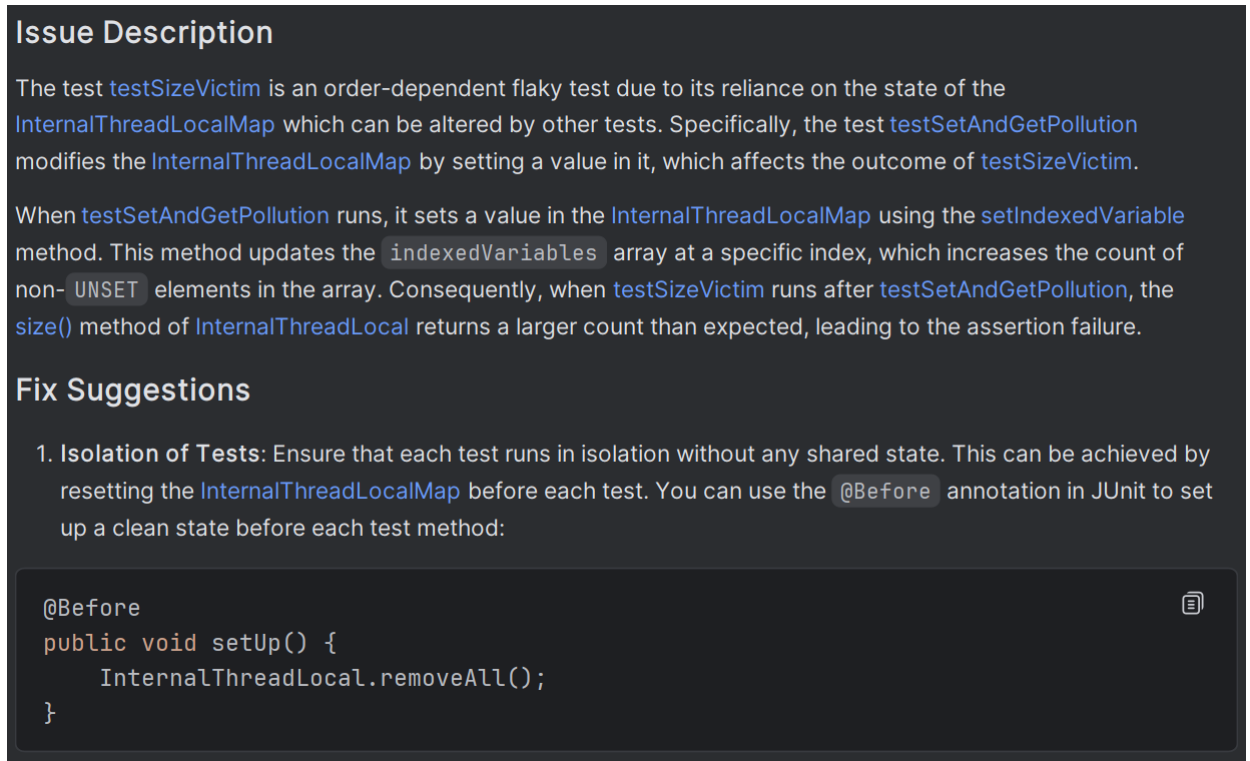


Рис. 3: Способ представления результатов работы анализатора

2) Запрос к LLM. После получения результатов анализа пользователь может ознакомиться с ними и, при желании, сделать специализированный запрос в LLM. В ответ пользователь получает подробное описание причины возникновения моргания, а также варианты его устранения. Пример ответа от LLM приведен на рисунке 4.



The screenshot displays an LLM response with two main sections: 'Issue Description' and 'Fix Suggestions'. The 'Issue Description' explains that the `testSizeVictim` test is flaky due to its reliance on the state of `InternalThreadLocalMap`, which is modified by `testSetAndGetPollution`. The 'Fix Suggestions' section provides a single recommendation: 'Isolation of Tests', suggesting the use of the `@Before` annotation to reset the `InternalThreadLocalMap` before each test.

### Issue Description

The test `testSizeVictim` is an order-dependent flaky test due to its reliance on the state of the `InternalThreadLocalMap` which can be altered by other tests. Specifically, the test `testSetAndGetPollution` modifies the `InternalThreadLocalMap` by setting a value in it, which affects the outcome of `testSizeVictim`.

When `testSetAndGetPollution` runs, it sets a value in the `InternalThreadLocalMap` using the `setIndexedVariable` method. This method updates the `indexedVariables` array at a specific index, which increases the count of non- `UNSET` elements in the array. Consequently, when `testSizeVictim` runs after `testSetAndGetPollution`, the `size()` method of `InternalThreadLocal` returns a larger count than expected, leading to the assertion failure.

### Fix Suggestions

- Isolation of Tests:** Ensure that each test runs in isolation without any shared state. This can be achieved by resetting the `InternalThreadLocalMap` before each test. You can use the `@Before` annotation in JUnit to set up a clean state before each test method:

```
@Before
public void setUp() {
    InternalThreadLocal.removeAll();
}
```

Рис. 4: Способ представления ответа LLM

## 5. Апробация

Разработанный анализатор способен находить, правильно анализировать и предлагать грамотный способ устранения ODFlaky для тестового класса `CatTest` из модельного примера, приведенного во введении 1. В этом разделе я также привожу примеры работы анализатора на реальных проектах с открытым исходным кодом: *Apache Dubbo*<sup>16</sup> и *WebFlux-R2DBC*<sup>17</sup>.

### 5.1. *Apache Dubbo*

*Apache Dubbo* — один из проектов, представленных в датасете репозитория<sup>18</sup> с известными flaky тестами из реальных проектов. В нем было обнаружено целое множество классов, содержащих порякозависимые моргающие тесты.

Рассмотрим один из них: `InternalThreadLocalTest`. ODFlaky тестом в данном тестовом классе является метод `testSize`, а загрязняющим — метод `testSetAndGet`.

```
1 class InternalThreadLocalTest {
2     @AfterEach
3     public void setup() {
4         InternalThreadLocalMap.remove();
5     }
6
7     ...
8
9     @Test
10    void testSize() {
11        final InternalThreadLocal<Integer> internalThreadLocal = new
12            InternalThreadLocal<Integer>();
13        internalThreadLocal.set(1);
14        Assertions.assertEquals(1, InternalThreadLocal.size(), "size method
15            is wrong!");
```

<sup>16</sup>The java implementation of Apache Dubbo. – URL: <https://github.com/apache/dubbo>

<sup>17</sup>WebFlux-R2DBC. – URL: <https://github.com/razvn/webflux-r2dbc-kotlin>

<sup>18</sup>International Dataset of Flaky Tests. – URL: <https://github.com/TestingResearchIllinois/idoft>

```

15     final InternalThreadLocal<String> internalThreadLocalString = new
        InternalThreadLocal<String>();
16     internalThreadLocalString.set("value");
17     Assertions.assertEquals(2, InternalThreadLocal.size(), "size method
        is wrong!");
18     InternalThreadLocal.removeAll();
19 }

20
21 @Test
22 void testSetAndGet() {
23     final Integer testVal = 10;
24     final InternalThreadLocal<Integer> internalThreadLocal = new
        InternalThreadLocal<Integer>();
25     internalThreadLocal.set(testVal);
26     Assertions.assertEquals(testVal, internalThreadLocal.get(), "set is
        not equals get");
27 }
28
29 ...
30 }

```

Анализатор справляется найти и проанализировать этот ODFlaky тест:

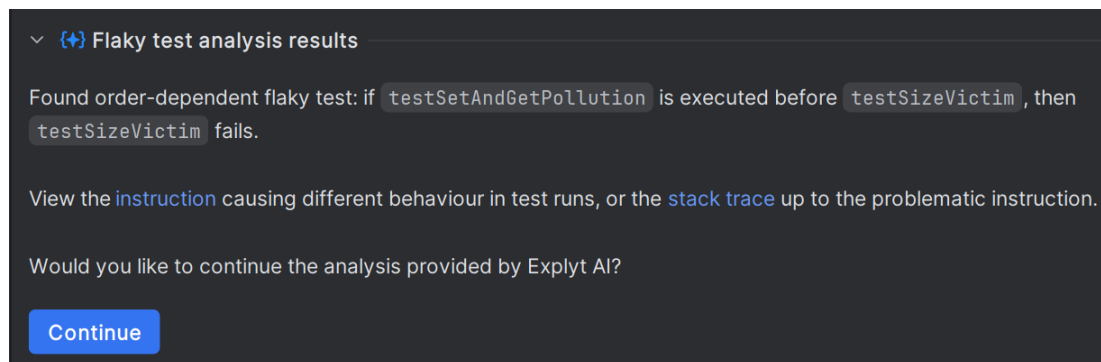


Рис. 5: Результаты работы анализатора для тестового класса `InternalThreadLocalTest`

Анализатор также справляется найти инструкцию, исполнение которой приводит к появлению моргания теста `testSize` (см., рисунок 6).

```

/**
 * @return {@code true} if and only if a new thread-local variable has been created
 */
public boolean setIndexedVariable(int index, Object value) { 3 usages
    Object[] lookup = indexedVariables;
    if (index < lookup.length) {
        Object oldValue = lookup[index];
        lookup[index] = value;
        return oldValue == UNSET;
    } else {
        expandIndexedVariableTableAndSet(index, value);
        return true;
    }
}

```

Рис. 6: Найденная анализатором инструкция, исполнение которой приводит к морганию теста `testSize`

На основе полученной из анализатора информации, LLM справляется подробно объяснить причину появления моргания, а также предложить корректный способ его устранения.

### Issue Description

The test `testSizeVictim` is an order-dependent flaky test due to its reliance on the state of the `InternalThreadLocalMap` which can be altered by other tests. Specifically, the test `testSetAndGetPollution` modifies the `InternalThreadLocalMap` by setting a value in it, which affects the outcome of `testSizeVictim`.

When `testSetAndGetPollution` runs, it sets a value in the `InternalThreadLocalMap` using the `setIndexedVariable` method. This method updates the `indexedVariables` array at a specific index, which increases the count of non- `UNSET` elements in the array. Consequently, when `testSizeVictim` runs after `testSetAndGetPollution`, the `size()` method of `InternalThreadLocal` returns a larger count than expected, leading to the assertion failure.

### Fix Suggestions

1. **Isolation of Tests:** Ensure that each test runs in isolation without any shared state. This can be achieved by resetting the `InternalThreadLocalMap` before each test. You can use the `@Before` annotation in JUnit to set up a clean state before each test method:

```

@Before
public void setUp() {
    InternalThreadLocal.removeAll();
}

```

Рис. 7: Объяснение причины возникновения ODFlaky `testSize` и способ устранения моргания

## 5.2. *WebFlux-R2DBC*

Проект *WebFlux-R2DBC* был найден случайно в ходе апробации генерации тестов инструментом Explyt Test. При генерации новых тестов для класса `UserServiceIT` анализатор обнаружил появление моргающих тестов, связанное с неаккуратной реализацией уже имеющихся в классе тестов.

Рассмотрим интересующую нас часть тестового класса `UserServiceIT`.

```
1 @ExperimentalCoroutinesApi
2 @TestInstance(TestInstance.Lifecycle.PER_CLASS)
3 @TestMethodOrder(MethodOrderer.OrderAnnotation::class)
4 @DataR2dbcTest
5 @ContextConfiguration(initializers = [TestContainerInitializer::class])
6 internal class UserServiceIT(
7     @Autowired private val connectionFactory: ConnectionFactory,
8     @Autowired private val repo: UserRepository
9 ) {
10     private lateinit var service: UserService
11
12     init {
13         val resourceLoader: ResourceLoader = DefaultResourceLoader()
14         val scripts = arrayOf(
15             resourceLoader.getResource("classpath:schema.sql"),
16             resourceLoader.getResource("classpath:data.sql")
17         )
18         ResourceDatabasePopulator(*scripts).execute(connectionFactory).block()
19     }
20
21     @BeforeAll
22     fun beforeAll() {
23         initDatabase()
24         service = UserService(repo)
25     }
26
27     ...
28
29     @Test
30     fun `addOne adds a user`() {
```

```

31     runBlocking {
32         val newUser = UserDTO("New user", "newuser", "newuser@users.com")
33         val resp = service.addOne(newUser)
34         assertThat(resp).isNotNull()
35         assertThat(resp?.id).isNotNull()
36         val dbUser = service.findByEmail(newUser.email).first()
37         assertThat(dbUser).isNotNull()
38     }
39 }
40
41 /**
42  * - Given multiple users exist in the database
43  * - When a request is made to retrieve all users
44  * - Then the system should return a list containing all users
45  * - And the count of users in the list should match the number of users
46     in the database
47  */
48 @Test
49 fun `findAll returns all users when multiple users exist`() {
50     runBlocking {
51         val resp = service.findAll().toList()
52         resp.forEach {
53             println(it)
54         }
55         assertThat(resp).hasSize(3)
56         assertThat(resp.map { it.name
57             }).containsExactlyInAnyOrder("User1", "User2", "User3")
58     }
59 }

```

Здесь `addOne` adds a user — тест добавленный разработчиками проекта, а `findAll returns all users when multiple users exist` — тест, сгенерированный Explyt Test. Анализатор ODFlaky тестов выдает следующее:

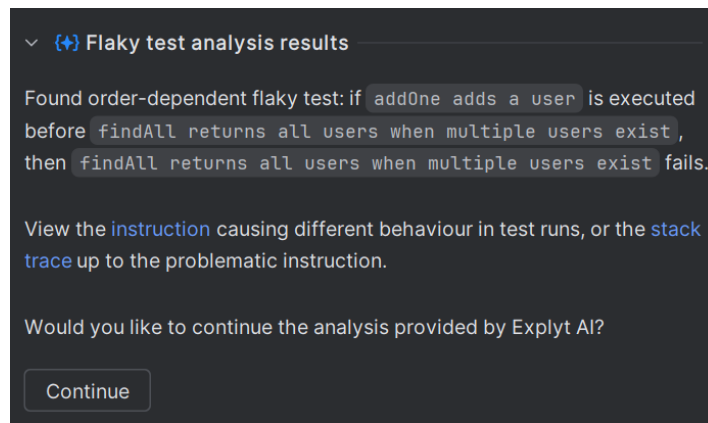


Рис. 8: Результаты работы анализатора для тестового класса UserServiceIT

LLM справляется подробно объяснить причину возникновения моргания и предложить способ устранения ошибки.

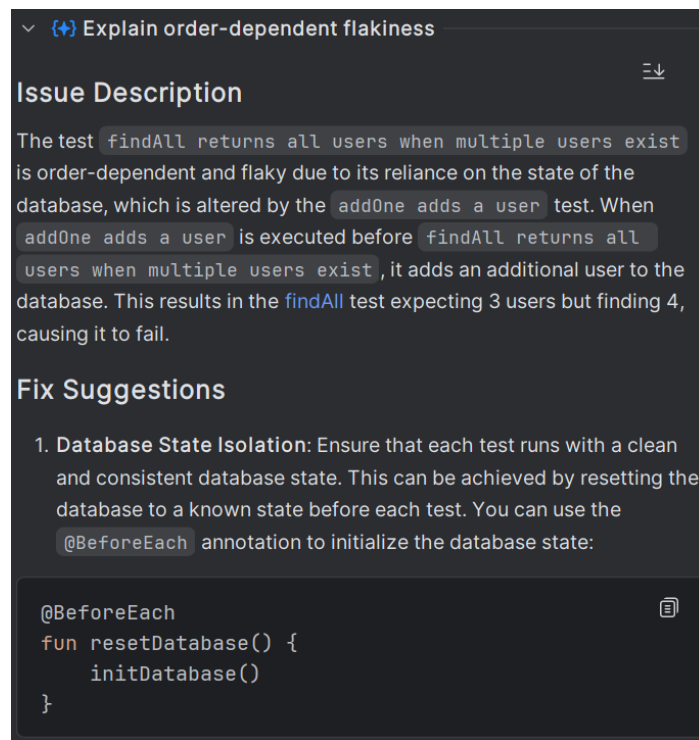


Рис. 9: Объяснение причины возникновения ODFlaky addOne adds a user и способ устранения моргания

# Заключение

В ходе выполнения работы в осеннем семестре были решены следующие задачи:

1. Исследованы предлагаемые подходы к анализу и устранению моргающих тестов, зависящих от порядка запуска, в научных публикациях и существующих инструментах
2. Разработан механизм выявления моргающего теста и приводящего его к морганию теста загрязнителя
3. Частично разработан механизм локализации инструкций в тесте загрязнителя, исполнение которых приводит к появлению моргающего теста
4. Реализован прототип полнофункционального решения
5. Проведена апробация анализатора на модельных примерах

В весеннем семестре были решены следующие задачи:

1. Доработан механизм локализации инструкций в тесте загрязнителя, исполнение которых приводит к появлению моргающего теста
2. Сформирован запрос к LLM на основе полученной в процессе анализа тестов информации
3. Разработан способ представления данных, полученных в процессе анализа тестов, а также рекомендаций, предложенных LLM
4. Предложенное решение было апробировано на наборе реальных проектов с моргающими тестами, зависящими от порядка запуска

Разработанный анализатор порядкозависимых тестов был включен в новый релиз<sup>19</sup> инструмента Explyt Test.

Код анализатора не может быть предоставлен, поскольку находится под договором о неразглашении.

---

<sup>19</sup>Explyt Test 3.0 features — URL: <https://explyt.ai/docs/explyt-test/features2503RU>

## Список литературы

- [1] A. Tahir, S. Rasheed, J. Dietrich, N. Hashemi, and Lu Zhang, “*Test flakiness’ causes, detection, impact and responses: A multivocal review*,” *Journal of Systems and Software*, Volume 206, 2023, doi: <https://doi.org/10.1016/j.jss.2023.111837> (дата обращения: 18 июня 2025 г.)
- [2] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMin, “*A Survey of Flaky Tests*,” *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 17, 2022, 74 pages, doi: <https://doi.org/10.1145/3476105> (дата обращения: 18 июня 2025 г.)
- [3] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi, “*Domain-Specific Fixes for Flaky Tests with Wrong Assumptions on Underdetermined Specifications*,” In *Proceedings of the 43rd International Conference on Software Engineering (ICSE ’21)*, IEEE Press, pp. 50–61, doi: <https://doi.org/10.1109/ICSE43902.2021.00018> (дата обращения: 18 июня 2025 г.)
- [4] Y. Pei, J. Sohn, S. Habchi, and M. Papadakis, “*Non-Flaky and Nearly-Optimal Time-based Treatment of Asynchronous Wait Web Tests*,” *ACM Trans. Softw. Eng. Methodol.* Just Accepted, 2024, doi: <https://doi.org/10.1145/3695989> (дата обращения: 18 июня 2025 г.)
- [5] Y. Chen and R. Jabbarvand, “*2024. Neurosymbolic Repair of Test Flakiness*,” In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*, Association for Computing Machinery, New York, NY, USA, pp. 1402–1414, doi: <https://doi.org/10.1145/3650212.3680369> (дата обращения: 18 июня 2025 г.)
- [6] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, “*IFixFlakies: a framework for automatically fixing order-dependent flaky tests*,” In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software*

*Engineering* (ESEC/FSE 2019), Association for Computing Machinery, New York, NY, USA, pp. 545–555, doi: <https://doi.org/10.1145/3338906.3338925> (дата обращения: 18 июня 2025 г.)

- [7] C. Li, C. Zhu, W. Wang, and A. Shi, “*Repairing order-dependent flaky tests via test generation*,” In *Proceedings of the 44th International Conference on Software Engineering (ICSE ’22)*, Association for Computing Machinery, New York, NY, USA, pp. 1881–1892, doi: <https://doi.org/10.1145/3510003.3510173> (дата обращения: 18 июня 2025 г.)
- [8] A. Gyori, A. Shi, F. Hariri, and D. Marinov, “*Reliable testing: detecting state-polluting tests to prevent test dependency*,” In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*, Association for Computing Machinery, New York, NY, USA, pp. 23–233, doi: <https://doi.org/10.1145/2771783.2771793> (дата обращения: 18 июня 2025 г.)
- [9] E. Kulikov and I. Muravev, “*Automatic Test Generation is Like Fighting Dragons*,” url: <https://jokerconf.com/en/talks/5b4dd75f9453450393121e835fc09eb8/> (дата обращения: 18 июня 2025 г.)
- [10] E. Kulikov and I. Muravev, “*Невыносимая легкость*” автоматической генерации тестов,” url: <https://heisenbug.ru/archive/2024%20Autumn/talks/aa0de44a3294492b9e9a17eb1cdf0fa4/> (дата обращения: 18 июня 2025 г.)