

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б11-мм

Анализ неустойчивости выделенного ЮНИТ-теста

Демченко Артем Евгеньевич

Отчёт по производственной практике
в форме «Производственное задание»

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

Санкт-Петербург
2025

Оглавление

Введение	3
1. Постановка задачи	5
2. Обзор	6
2.1. Методы обнаружения нестабильных тестов	6
2.2. Используемые технологии	8
3. Реализация	10
3.1. Запуск пользовательских тестов	11
3.2. Определение нестабильного теста	11
3.3. Составление промпта и запрос в LLM	12
3.4. Взаимодействие с пользователем	13
4. Тестирование	15
4.1. Нестабильность сети	15
4.2. Гонка данных	16
Заключение	19
Список литературы	20

Введение

Как известно, тестирование — необходимый этап разработки ПО, помогающий обеспечить качество создаваемого продукта. Тестирование можно разделить на два вида — ручное и автоматизированное. При автоматизированном подходе тестовые сценарии проверяются без участия человека. Этот процесс инициируется довольно часто, например, при непрерывной интеграции или выпуске очередной версии продукта. В первом случае тесты запускаются каждый раз при попытке внедрения новых изменений в проект. От их результатов зависит, готовы ли эти изменения к интеграции. Зачастую хотя бы один провалившийся тест означает, что код нужно дорабатывать.

Нестабильные тесты — тесты, результат которых может меняться без изменения исходного кода. Причины могут быть разные — недетерминированность в коде, зависимость от внешних компонентов, некорректная инициализация и/или очистка тестового окружения. Выходит, что если какой-то тест в проекте не проходит, то программистам необходимо определять, менялся ли исходный код, относящийся к провальному тесту, и на основе этого выяснять, является ли этот тест нестабильным. В статье блога компании Google, посвященному тестированию, отмечается, что примерно 1,5% от всех их тестов являются нестабильными [10]. Таким образом, для проекта с 1000 тестами примерно 15 дают нестабильные результаты при каждом запуске. На практике зачастую такие тесты помещают в карантин, то есть исключают из набора запускаемых тестов. Это приводит к тому, что часть функциональности остается не протестированной. При исправлении же нестабильного теста возникают дополнительные трудности по сравнению с исправлением теста, который всегда не проходит. Во-первых, ошибка может быть как в тестируемом коде, так и в коде самого теста. Во-вторых, провальный результат нестабильного теста не всегда можно воспроизвести. Следовательно, процесс определения, является ли тест нестабильным, а также его исправление сказывается на времени разработки, замедляя ее [12]. Возникает задача по созданию инструментов,

помогающих разработчикам определять нестабильные тесты на ранних этапах и исправлять их.

Explyt Test [3] — динамично развивающийся продукт, представляющий собой плагин для IntelliJ IDEA. На данный момент он в основном нацелен на автоматическую генерацию тестов для Java и Kotlin, но планируется, что Explyt Test станет инструментом всестороннего ухода за тестами, в том числе помощником при работе с нестабильными тестами. Поэтому анализатор тестов на предмет нестабильности должен стать частью этого инструмента. Такая проверка будет полезна как для уже существующих в проекте тестов, так и для вновь сгенерированных с помощью Explyt Test.

1. Постановка задачи

Цель работы — разработать анализатор, определяющий, является ли выделенный юнит-тест нестабильным, и предоставляющий полезную для исправления информацию. Для достижения цели были поставлены задачи:

1. рассмотреть существующие способы анализа нестабильности тестов;
2. реализовать анализатор юнит-теста на предмет нестабильности, извлекающий полезную для его исправления информацию;
3. исследовать возможности больших языковых моделей в анализе причин нестабильности тестов;
4. реализовать механизм представления результатов анализа пользователю;
5. протестировать разработанный инструмент на наборе модельных примеров;
6. внедрить разработанные компоненты в Explyt Test.

2. Обзор

2.1. Методы обнаружения нестабильных тестов

Основным источником для поиска возможных подходов определения, является ли тест нестабильным, послужила большая обзорная статья [12] 2021 года, посвященная систематизации знаний из других 76 статей на тему нестабильных тестов. В статье также исследуются причины возникновения, последствия для проектов и способы исправления. Далее будут рассмотрены инструменты, нацеленные на обнаружение нестабильных тестов.

Инструмент iDFlakies [15] предназначен для обнаружения и классификации нестабильных тестов. Он представляет собой maven-плагин, работающий с тестовым фреймворком JUnit. В этом заключается ограничение данного инструмента. Для определения того, является ли тест нестабильным, авторы используют стратегию **ReRun** — запуск тестов несколько раз и сравнение результатов. Таким образом, проверка выполняется по определению нестабильного теста. Плюс данной стратегии — несложная реализация, минус — затраты ресурсов на повторный запуск тестов.

DeFlaker [2] — еще один инструмент, целью которого является автоматическое обнаружение нестабильных тестов. Его преимуществом перед стратегией **ReRun** является то, что запуск тестов происходит только один раз. Принцип работы таков: анализ тестов происходит при внесении каких-либо изменений в систему контроля версий. Осуществляется инструментированный запуск тестов и на основе их работы строится покрытие. Затем анализируются полученные данные. Если тест провалился и измененный код не покрыт, то такой тест считается нестабильным. Такой подход имеет недостаток, так как нацелен на обнаружение нестабильных тестов, которые уже существуют в проекте.

NonDex [11] — инструмент, предназначенный для поиска тестов, реализация которых опирается на неверные предположения о детерминированности недетерминированных спецификаций в Java. Примером это-

му может служить порядок получения элементов `HashMap`. При этом зачастую на одинаковых платформах и версиях Java реализация спецификаций будет одной и той же, поэтому определять нестабильность тестов в этом случае становится трудно. Авторы инструмента запускают тесты, при этом подменяя вызов недетерминированного API своими реализациями. Например, реализация `HashMap` заменяется такой, что при итерации по элементам их получение происходит в случайном порядке. Всего в инструменте поддерживается 41 недетерминированная спецификация Java [16]. Такой подход действительно находит неверные предположения о них, но в целом затрагивает довольно узкий спектр причин возникновения нестабильных тестов.

Также стоит отметить, что после 2021 года продолжают выходить статьи, посвященные анализу тестов на предмет нестабильности. В одной из них [4] описывается инструмент `Flakify` — анализатор, основанный на языковой модели `CodeBERT` [1], которая была дообучена авторами. `Flakify` определяет, является ли тест нестабильным, опираясь только на его исходный код. `CodeBERT` имеет ограничение на количество входных токенов (512). Авторы разработали эвристику уменьшения количества токенов, в которые конвертируется тестовый метод. Тем не менее полностью избавиться от этого ограничения нельзя. Код анализатора выгружен на GitHub [5], но он никак не поддерживается более двух лет. Хотя в README репозитория есть инструкция, как воспроизвести результаты, полученные авторами в статье, документации по использованию инструмента в других проектах нет. Исходный код также не документирован.

Таким образом, учитывая ограничения существующих инструментов, специфику `Explyt Test` и необходимость анализировать только что сгенерированные тесты, для анализатора была выбрана стратегия `ReRun`. Также такая стратегия позволяет получить информацию о трассах исполнения тестов, а затем анализировать расхождения успешного и провального запусков.

2.2. Используемые технологии

Анализатор должен быть интегрирован в Explyt Test — плагин для IntelliJ IDEA, построенный на IntelliJ Platform [6]. Поэтому разрабатываемый анализатор также будет использовать данное API. Написание плагинов поддерживается на языках Java и Kotlin. Для анализатора был выбран Kotlin, так как на нем написан Explyt Test. В Explyt Test основной способ взаимодействия с пользователями — использование Action-ов [7], предоставляемых IntelliJ Platform. Например, точка входа для генерации тестов использует этот механизм. Поэтому точка входа для анализа нестабильности юнит-теста должна быть реализована в виде Action-а, обеспечивая единый стиль взаимодействия с пользователем. При этом IntelliJ Platform также предоставляет Execution API [8], благодаря которому становится возможным запуск тестов в проекте пользователя и их анализ. Ключевые его интерфейсы:

- `RunConfiguration`. Описывает процесс, который может быть запущен. Позволяет указывать рабочую директорию, переменные среды, а также аргументы командной строки.
- `ProgramRunner`. Ответственен за процесс исполнения. Должен имплементировать точку расширения `com.intellij.programRunner`.
- `RunConfigurationExtension`. Позволяет модифицировать `RunConfiguration`, добавляя, например, аргумент командной строки.

Для межпроцессного взаимодействия в Explyt Test используется фреймворк JetBrains RD (Reactive Distributed) [9]. RD предоставляет возможность создания модели данных, которая автоматически синхронизируется на обеих сторонах (клиентской и серверной). Также поддерживается автоматическая генерация исходного кода для классов модели и методов сериализации и десериализации этой модели на основе декларативного описания, что уменьшает количество кода и упрощает разработку.

Также в Explyt Test уже реализован компонент runtime, который предоставляет java agent, позволяющий совершать инструментированные запуски тестов и собирать трассы исполнения. Предполагается, что результаты работы тестов и собранные трассы исполнения передаются по протоколу RD.

Трассы исполнения, получаемые благодаря компоненту runtime, представляют собой список инструкций. Каждая является имплементацией интерфейса JcInst — части библиотеки JacoDB [14]. Это библиотека, предоставляющая способ получить некоторую информацию о байт-коде вне процесса JVM. Js-инструкции будут полезны для предоставления пользователю информации о расхождении успешного и провального запусков нестабильного теста. Для этого нужно будет разработать способ сопоставлять конкретной Js-инструкции строчку в исходном файле.

Для отправки запросов в LLM и дальнейшего анализа результатов будет использоваться Ai Client — модуль в Explyt Test, отвечающий за взаимодействие с большими языковыми моделями.

3. Реализация

На рисунке 1 представлена архитектура инструмента с зависимыми модулями. Для осуществления стратегии ReRun в проекте был реализован модуль Test Runner, осуществляющий запуск тестов пользователя. Для сбора трасс исполнения тестов использовался существующий модуль Runtime, предоставляющий java agent, который позволяет выполнять инструментированные запуски и собирать трассы. Изначально в проекте трассы собирались в виде покрытия кода. Поскольку для анализатора необходима полная последовательность выполненных инструкций, были внесены изменения в USVM [13] — библиотеку с открытым исходным кодом, которая используется для сбора трасс. Модули Runtime и Test Runner взаимодействуют по протоколу RD. В основе взаимодействия лежит общая модель данных. На стороне Test Runner регистрируется обработчик изменений этой модели, а на стороне Runtime в нее записываются результаты запуска тестов и трассы.

В итоге модуль с анализатором использует API, предоставляемое модулем Test Runner для повторного запуска тестов. При определении, что пользовательский тест является нестабильным, отправляется запрос в LLM с помощью модуля Ai Client.

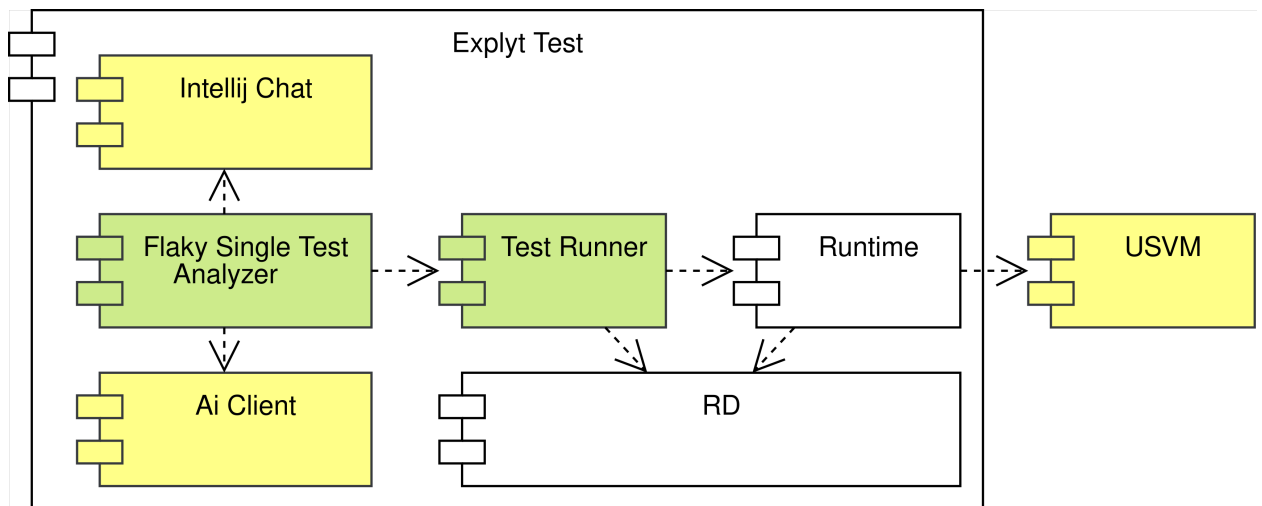


Рис. 1: Зависимость модулей в Explyt Test. Зеленым отмечены те, что были реализованы в данной работе. Желтым — те, в которые были внесены изменения

Представление результатов осуществляется благодаря модулю IntelliJ Chat. Он предоставляет гибкий интерфейс для взаимодействия с пользователем через чат, открывающийся в отдельном окне в среде разработки.

3.1. Запуск пользовательских тестов

На рисунке 2 представлена архитектура модуля Test Runner. Этот модуль реализован с целью предоставления публичного API для запуска пользовательских тестов и сбора трасс исполнения. Непосредственно запуск осуществляется механизмами IntelliJ Platform. Для возможности инструментированного запуска потребовалось реализовать выделенные компоненты. Например, в методе `updateJavaParameters(...)` реализована логика добавления опции `-javaagent` к команде процесса запуска тестов.

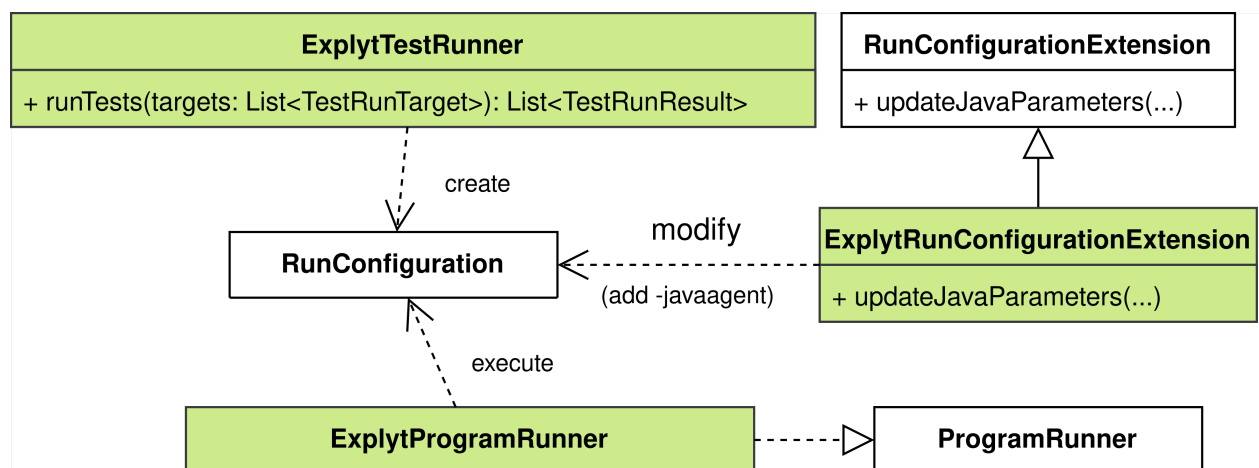


Рис. 2: Архитектура модуля Test Runner

3.2. Определение нестабильного теста

Ниже приведен алгоритм, реализующий стратегию ReRun и собирающий информацию о запусках, если тест нестабильный.

1. Тест запускается N раз.

2. Проверяется, есть ли различные результаты среди запусков. Если есть — тест распознается как нестабильный и совершается переход к следующему шагу, иначе алгоритм завершается.
3. Рассматриваются две трассы исполнения: успешная и провальная. Находится наибольший общий префикс. Таким образом выделяется последняя общая для двух трасс инструкция.
4. По префиксу трассы строится стек вызовов. Выполняется линейный проход по списку инструкций. Если встречается инструкция «call», то она добавляется на стек. Встречается «return» — со стека удаляется последняя инструкция. Остальные инструкции игнорируются.
5. Каждой инструкции сопоставляется строчка в исходном файле.
6. Получены последняя общая строчка кода и стек вызовов, алгоритм завершается.

3.3. Составление промпта и запрос в LLM

Цель запроса к LLM — правильно распознать причины нестабильности теста. Поэтому в промпт включается информация о запусках, полученная на этапе выявления нестабильности. Промпт содержит:

1. Текстовые инструкции. Здесь детально описывается поставленная задача, а также формат данных, которые будут переданы.
2. Данные о расхождении трасс, а именно:
 - первая строчка исходного кода, после которой расходится поведение успешного и провального запусков;
 - стек вызовов, приведший к расхождению.
3. Контекст тестового метода. Здесь учитывается исходный код тестового класса и классов, методы которых вызывались во время исполнения теста.

3.4. Взаимодействие с пользователем

Большинство принятых решений обусловлено тем, что инструмент должен интегрироваться в Explyt Test. Анализ начинается с того, что пользователь нажимает иконку около тестового метода и выбирает вариант, соответствующий анализу теста на нестабильность. После чего начинается первая часть алгоритма - определение нестабильности и получение информации о запуске. Далее все взаимодействие с пользователем происходит через интерактивный чат, который открывается в правой части среды разработки. Такой формат выбран потому, что такой механизм уже применяется в иных функциях плагина — например, при генерации тестов и анализе покрытия.

После того, как этап с запуском тестов закончен, пользователю показывается сообщение с результатами.

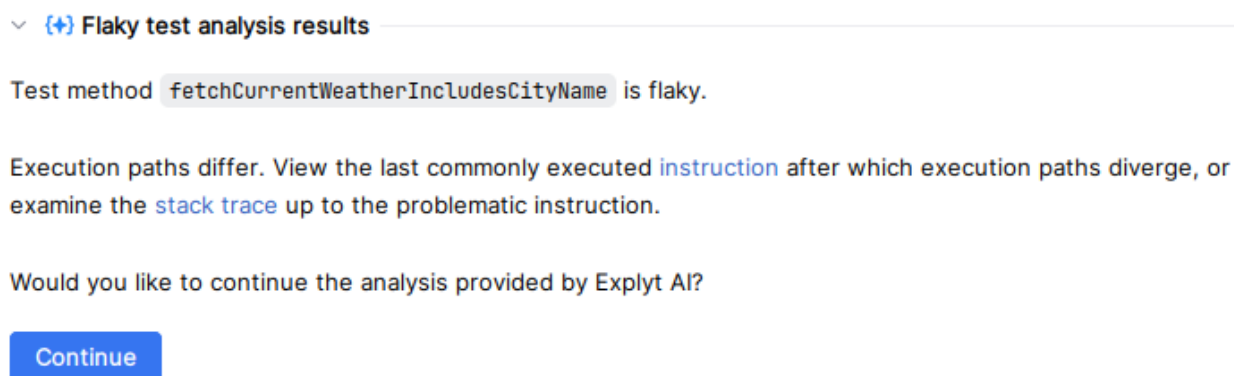


Рис. 3: Сообщение, показывающее пользователю результат работы анализатора, где тест был распознан как нестабильный

При нажатии на гиперссылку, соответствующую инструкции, в редакторе открывается файл, где курсор размещается на строке, где содержится инструкция. При нажатии на стек вызовов в редакторе открывается файл в режиме чтения, который содержит цепочку вызовов, как видно на рисунке 4.

При нажатии кнопки Continue на рисунке 3 в LLM отправляется запрос с промптом, составление которого описано в разделе 3.3. В конечном итоге пользователь видит сообщение с объяснением причины нестабильности (рисунок 5).



Рис. 4: Файл в редакторе, содержащий стек вызовов, приведший к расхождению в трассах

⚡ Explain single test flakiness

Issue Description

The test

```
@Test
void fetchCurrentWeatherIncludesCityName() throws IOException,
InterruptedException {
    WeatherReport report = service.getCurrentWeather("London");
    assertEquals("London", report.city());
}
```

is *flaky* because it depends on a real HTTP call to `https://wttr.in/London?format=3` with a 1 s timeout. In practice:

- **Network variability** (latency, connectivity) can trigger timeouts (`HttpTimeoutException`) or slow responses.
- **External service changes**: `wttr.in` may return different formats, HTTP errors or redirects.
- **Regional or rate-limiting issues**: You might get blocked or receive different outputs (e. g. localized names, ads, multi-line responses), causing `WeatherParser` to mis-parse the city or default to `"unknown"`.

All of the above lead to nondeterministic pass/fail behavior.

Рис. 5: Сообщение, показывающее пользователю объяснение причины нестабильности теста

4. Тестирование

Анализатор тестировался на созданных примерах, где смоделированы разные причины возникновения нестабильности, и на самом проекте Explyt Test. Считалось, что пример пройден анализатором, если он распознал нестабильный тест, и в ответе LLM была указана истинная причина нестабильности. Используемая LLM — o4-mini.

4.1. Нестабильность сети

В этом примере нестабильность теста вызвана тем, что в исходном коде класса `WeatherClient` при получении погоды с внешнего сервиса стоит ограничение в одну секунду на получение ответа.

Листинг 1: Исходный код классов из примера с нестабильностью сети

```
1 public final class WeatherClient {
2     private static final Duration REQUEST_TIMEOUT = Duration.ofSeconds(1);
3     ...
4     public String fetchCurrentWeather(String city) throws IOException,
5         InterruptedException {
6         HttpRequest request = HttpRequest.newBuilder()
7             .uri(URI.create("https://wttr.in/" + city + "?format=3"))
8             .timeout(REQUEST_TIMEOUT)
9             .GET()
10            .build();
11        return http.send(request, HttpResponse.BodyHandlers.ofString()).body();
12    }
13 }
14 public final class WeatherService {
15     ...
16     public WeatherReport getCurrentWeather(String city) throws IOException,
17         InterruptedException {
18        String raw = client.fetchCurrentWeather(city);
19        return parser.parse(raw);
20    }
21 }
```

Листинг 2: Нестабильный тест из примера с нестабильностью сети

```
1 class WeatherServiceTimeoutTest {
2     private final WeatherService service = new WeatherService(new
        WeatherClient(), new WeatherParser());
3     @Test
4     void fetchCurrentWeatherIncludesCityName() throws IOException,
        InterruptedException {
5         WeatherReport report = service.getCurrentWeather("London");
6         assertEquals("London", report.city());
7     }
8 }
```

Результаты работы анализатора на этом примере можно видеть на рисунках 3, 4 и 5.

4.2. Гонка данных

В этом примере нестабильность теста вызвана тем, что в исходном коде класса `InventoryService` хранилище `stock` не является потокобезопасным, в то время как в тесте два потока пытаются зарезервировать товар.

Листинг 3: Исходный код классов из примера с гонкой данных

```
1 public final class InventoryService {
2     private final Map<String, Integer> stock = new HashMap<>();
3     ...
4     public boolean reserve(String sku, int qty) {
5         Integer available = stock.get(sku);
6         if (available == null || available < qty) {
7             return false;
8         }
9
10        stock.put(sku, available - qty);
11        return true;
12    }
13 }
```

Листинг 4: Исходный код нестабильного теста из примера с гонкой данных

```
1 class InventoryServiceFlakyTest {
2     @Test
3     void shouldNotOversellSingleUnit() throws Exception {
4         InventoryService service = new InventoryService();
5         String sku = "SKU-123";
6         service.addInbound(sku, 1);
7
8         ExecutorService pool = Executors.newFixedThreadPool(2);
9         CountDownLatch ready = new CountDownLatch(2);
10        CountDownLatch start = new CountDownLatch(1);
11        AtomicBoolean reversed1 = new AtomicBoolean(false);
12        AtomicBoolean reversed2 = new AtomicBoolean(false);
13
14        pool.submit(() -> {
15            ready.countDown();
16            start.await();
17            reversed1.set(service.reserve(sku, 1));
18            return null;
19        });
20        pool.submit(() -> {
21            ready.countDown();
22            start.await();
23            reversed2.set(service.reserve(sku, 1));
24            return null;
25        });
26
27        ready.await();
28        start.countDown();
29        pool.shutdown();
30        pool.awaitTermination(2, TimeUnit.SECONDS);
31
32        int remaining = service.available(sku);
33        assertTrue((reversed1.get() ^ reversed2.get()) && remaining == 0);
34    }
35 }
```

Результаты работы анализатора на этом примере можно видеть на рисунках 6, 7.

✚ Flaky test analysis results

Test method `shouldNotOverSellSingleUnit` is flaky.

Execution paths differ. View the last commonly executed [instruction](#) after which execution paths diverge, or examine the [stack trace](#) up to the problematic instruction.

Would you like to continue the analysis provided by Explyt AI?

Continue

Рис. 6: Сообщение, показывающее пользователю, что тест из примера с гонкой данных нестабильный

✚ Explain single test flakiness

Issue Description

The flakiness here is **not** in JUnit but in the fact that `InventoryService.reserve(...)` is *not* thread-safe. Internally it does:

```
Integer available = stock.get(sku);
if (available == null || available < qty) return false;
stock.put(sku, available - qty);
return true;
```

When two threads call `reserve(...)` at the very same time:

1. Both read `available = 1`.
2. Both pass the `available < qty` check.
3. Each writes back `0`.
4. Both return `true`.

You end up with **two successful reservations** of a single unit—i.e. an oversell—so your assertion

```
assertTrue((reversed1 ^ reversed2) && remaining == 0);
```

sometimes fails. Whether it passes or fails is purely a race condition, hence the “flaky” behavior.

Рис. 7: Сообщение, объясняющее причины нестабильности теста из примера с гонкой данных

Заключение

В ходе данной производственной практики в осеннем семестре были решены следующие задачи:

1. Рассмотрены существующие способы анализа нестабильности тестов. Среди них для реализации был выбран тот, который лучше всего подходит для интеграции в Explyt Test.
2. Реализован анализатор, определяющий, является ли выделенный юнит-тест нестабильным, и предоставляющий данные для его исправления. Они включают первую строчку кода, после которой расходится исполнение провального и успешного запуска, и стек вызовов, приводящий к расхождению.

В весеннем семестре были решены следующие задачи:

1. Разработан специализированный промпт для взаимодействия с большой языковой моделью.
2. Реализован механизм представления результатов анализа пользователю.
3. Разработанный инструмент протестирован на наборе модельных примеров.
4. Компоненты внедрены в Explyt Test и включены в новый релиз¹ инструмента.

Код изменений в UnitTestBot usvm доступен по ссылке:
<https://github.com/UnitTestBot/usvm/pull/229>

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

¹<https://explyt.ai/docs/explyt-test/features2503RU>

Список литературы

- [1] [CodeBERT: A Pre-Trained Model for Programming and Natural Languages](#) / Zhangyin Feng, Daya Guo, Duyu Tang et al. — 2020. — 01. — P. 1536–1547.
- [2] [DeFlaker: automatically detecting flaky tests](#) / Jonathan Bell, Owolabi Legunsen, Michael Hilton et al. // Proceedings of the 40th International Conference on Software Engineering. — ICSE '18. — New York, NY, USA : Association for Computing Machinery, 2018. — P. 433–444. — URL: <https://doi.org/10.1145/3180155.3180164>.
- [3] Explyt Test. — URL: <https://explyt.com/test/> (дата обращения: 23 июня 2025 г.).
- [4] Fatima Sakina, Ghaleb Taher A., Briand Lionel. Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests // [IEEE Transactions on Software Engineering](#). — 2023. — Vol. 49, no. 4. — P. 1912–1927.
- [5] Flakify Test Smell Detector. — URL: <https://github.com/uOttawa-Nanda-Lab/Flakify> (дата обращения: 23 июня 2025 г.).
- [6] IntelliJ Platform. — URL: <https://plugins.jetbrains.com/docs/intellij/welcome.html> (дата обращения: 23 июня 2025 г.).
- [7] IntelliJ Platform Actions. — URL: <https://plugins.jetbrains.com/docs/intellij/basic-action-system.html> (дата обращения: 23 июня 2025 г.).
- [8] IntelliJ Platform Execution API. — URL: <https://plugins.jetbrains.com/docs/intellij/execution.html> (дата обращения: 23 июня 2025 г.).
- [9] JetBrains RD. — URL: <https://github.com/JetBrains/rd> (дата обращения: 23 июня 2025 г.).

- [10] Micco John. Flaky Tests at Google and How We Mitigate Them. — 2016. — URL: <https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html> (дата обращения: 23 июня 2025 г.).
- [11] NonDex: a tool for detecting and debugging wrong assumptions on Java API specifications / Alex Gyori, Ben Lambeth, August Shi et al. // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. — FSE 2016. — New York, NY, USA : Association for Computing Machinery, 2016. — P. 993–997. — URL: <https://doi.org/10.1145/2950290.2983932>.
- [12] A Survey of Flaky Tests / Owain Parry, Gregory M. Kapfhammer, Michael Hilton, Phil McMinn // ACM Trans. Softw. Eng. Methodol. — 2021. — . — Vol. 31, no. 1. — 74 p. — URL: <https://doi.org/10.1145/3476105>.
- [13] Unit Test Bot USVM. — URL: <https://github.com/UnitTestBot/usvm> (дата обращения: 23 июня 2025 г.).
- [14] UnitTestBot JacoDB. — URL: <https://github.com/UnitTestBot/jacodb> (дата обращения: 23 июня 2025 г.).
- [15] iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests / Wing Lam, Reed Oei, August Shi et al. // 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). — 2019. — P. 312–322.
- [16] Исходный код инструмента NonDex. — URL: <https://github.com/TestingResearchIllinois/NonDex> (дата обращения: 23 июня 2025 г.).