Санкт-Петербургский государственный университет

Кафедра системного программирования

Анализ нестабильности выделенного юнит-теста

Демченко Артем Евгеньевич, группа 22.Б11-мм

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

Санкт-Петербург
2025

Введение

- Хотя бы один тест упал — сборка считается неуспешной
- Нестабильные тесты — тесты, результат которых может меняться без изменения исходного кода
- 1,5% всех тестов Google — нестабильные[1]
- Наличие в проекте нестабильных тестов замедляет процесс разработки
- Возникает потребность в инструментах, анализирующих тесты на нестабильность
- Explyt Test — плагин для IntelliJ IDEA, динамично развивающийся продукт
  - Сейчас — генератор тестов для Java и Kotlin, AI-ассистент
  - В будущем — инструмент всестороннего ухода за тестами

[1] По данным из Google testing blog на 2016 год

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:

Задачи:

- Рассмотреть существующие способы анализа нестабильности тестов
- Реализовать анализатор юнит-теста на предмет нестабильности, извлекающий полезную для его исправления информацию
- Исследовать возможности больших языковых моделей в анализе причин нестабильности тестов
- Реализовать механизм представления результатов анализа пользователю
- Протестировать разработанный инструмент на наборе модельных примеров
- Внедрить разработанные компоненты в Explyt Test

Методы обнаружения нестабильных тестов

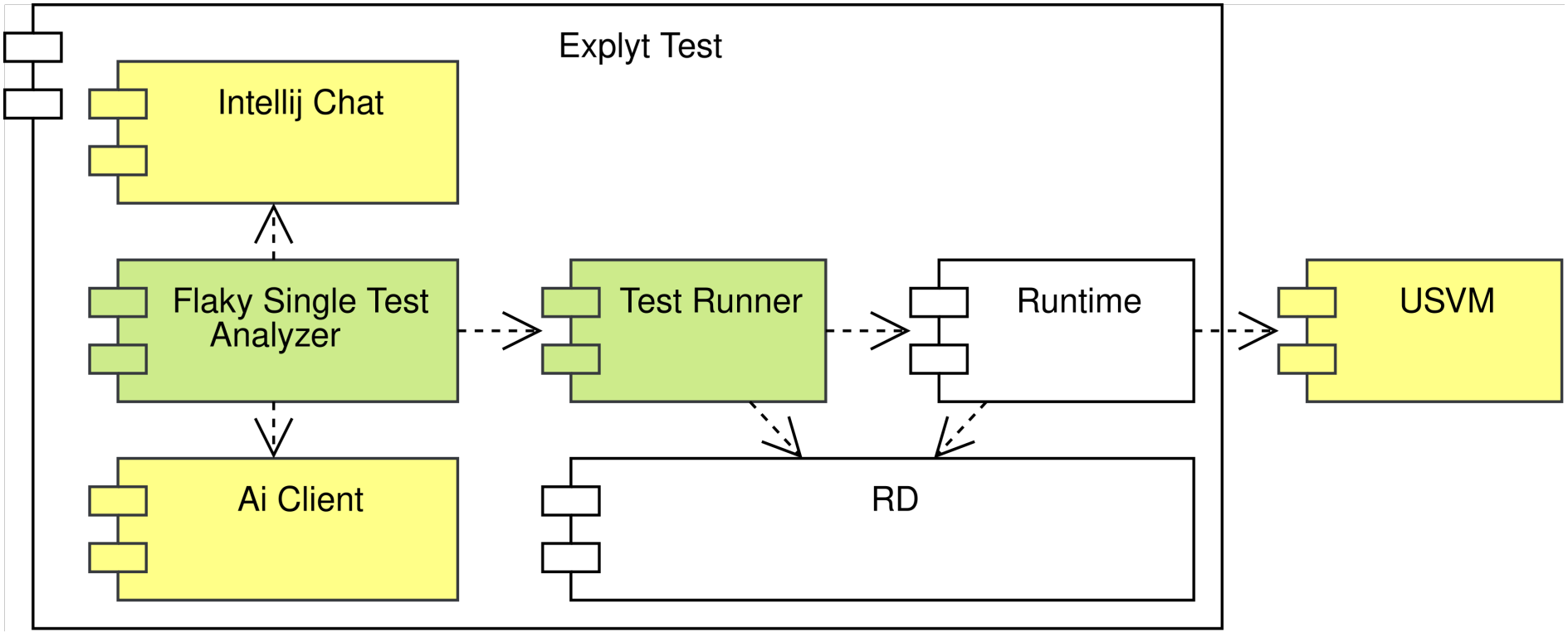
A Survey of Flaky Tests — Owain Parry, Gregory M. Kapfhammer, Michael Hilton, Phil McMinn, 2021, https://doi.org/10.1145/3476105

- iDFlakies
  - Maven-плагин, работающий только с тестовым фреймворком JUnit
  - Используется стратегия ReRun — запуск тестов происходит несколько раз
- DeFlaker
  - Запуск тестов происходит один раз при внесении изменений в систему контроля версий
  - Считается покрытие кода тестами
  - Тест упал и измененный код не покрыт — тест помечается как нестабильный
  - Нацелен на обнаружение нестабильных тестов, которые уже существуют в проекте

Методы обнаружения нестабильных тестов

- NonDex
  - Поиск неверных предположений о детерминированности недетерминированных спецификаций в Java
  - Пример — порядок итерации по элементам HashMap
  - Тесты запускаются с инструментацией, подменяя реализацию недетерминированного API
  - Поддержка 41 спецификации
- Flakify
  - Подход на основе дообученной языковой модели CodeBERT
    - Модель имеет ограничение на количество входных токенов — 512
  - Инструмент не поддерживается более двух лет
  - Отсутствие документации по использованию

Зависимость модулей в Explyt Test



- Runtime — предоставляет java agent для инструментации
- RD — фреймворк для межпроцессного взаимодействия
- Ai Client — предоставляет API для взаимодействия с LLM
- Intellij Chat — чат для взаимодействия с пользователем
- USVM — добавляет возможность сбора трасс

Алгоритм работы анализатора

Определяем нестабильность теста:

1. Запускаем пользовательский юнит-тест N раз
2. Смотрим, есть ли разные результаты выполнения тестов

Получаем информацию о нестабильности:

1. Сравнивая успешную и провальную трассы, получаем инструкцию, где исполнение расходится
2. По совпадающему префиксу трасс строим стек вызовов
   - Call-инструкции кладем на стек
   - Встречаем return — убираем со стека последнюю добавленную
3. Сопоставляем каждой инструкции строчку в исходном файле

Составление запроса для LLM

Составляющие запроса:

- Текстовые инструкции
- Строчка кода, после которой расходится исполнение провального и успешного запусков
- Стек вызовов, приводящий к расхождению
- Дополнительный контекст - исходный код классов, достижимых из тестового метода

Тестирование

Были смоделированы примеры для следующих причин нестабильности:

- Проблемы, связанные с нестабильностью сети
- Некорректный многопоточный код, гонка данных
- Недетерминированность
  - Один тест такого типа был в кодовой базе Explyt Test

Считалось, что пример пройден анализатором, если он распознал нестабильный тест, и в ответе LLM была указана истинная причина нестабильности

Решенные задачи (осень)

В прошлом семестре были решены следующие задачи:

- Рассмотрены существующие способы анализа нестабильности тестов. Среди них для реализации был выбран тот, который лучше всего подходит для интеграции в Explyt Test
- 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

Решенные задачи (весна)

В ходе данной производственной практики были решены следующие задачи:

- Разработан специализированный промпт для взаимодействия с большой языковой моделью
- Реализован механизм представления результатов анализа пользователю
- Разработанный инструмент протестирован на наборе модельных примеров
- Компоненты внедрены в Explyt Test и включены в новый релиз[2]

[2] https://explyt.ai/docs/explyt-test/features2503RU

Пример ответа LLM, если тест flaky (без трасс)

Проект: fastjson2[3]

```java
package com.alibaba.fastjson2.issues_2600;

import com.alibaba.fastjson2.JSON;
import com.alibaba.fastjson2.JSONWriter;
import com.alibaba.fastjson2.filter.PropertyPreFilter;
import org.junit.jupiter.api.Test;

import java.util.HashMap;
import java.util.Map;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class Issue2672 {
    @Test
    public void test() {
        Map<String, Object> map = new HashMap<>();
        map.put("k", "v");
        map.put("selfRef", map);
        assertEquals( expected: "{\"selfRef\":{\"$ref\":\"..\"},\"k\":\"v\"}",
                JSON.toJSONString(map, JSONWriter.Feature.ReferenceDetection));

        PropertyPreFilter propertyJsonFilter = ( JSONWriter serializer, Object source, String proper
        assertEquals( expected: "{\"selfRef\":{\"$ref\":\"$\"},\"k\":\"v\"}",
                JSON.toJSONString(map, propertyJsonFilter, JSONWriter.Feature.ReferenceDetection))
    }
}
```

Flaky tests analysis

The failure here is not in Fast-json2 itself but in the test. You're using a plain `HashMap`, whose iteration order is not guaranteed. Because you assert against a *specific* ordering of keys (`"selfRef"` first, `"k"` second), the test will randomly pass or fail depending on the hash-bucket layout at runtime.

Why it's flaky

1. **HashMap is unordered**
   A `HashMap` does *not* preserve insertion order (that's `LinkedHashMap`) nor guarantee any particular order of iteration.
2. **Test asserts exact key order**
   You do

```java
assertEquals(
  "{\"selfRef\":{\"$ref\":\"..\"},\"k\":\"v\"}",
  JSON.toJSONString(map, JSONWriter.Feature.ReferenceDetection)
);
```

but if map iterates as `("k","v")` then `("selfRef",map)` you'll see `{"k":"v","selfRef":{"$ref":".."}}` and the test will fail.

How to fix

Choose one of the following approaches:

1. **Use a LinkedHashMap**
   Preserve the insertion order you expect:

[3] https://github.com/alibaba/fastjson2

Пример ответа LLM, если тест НЕ flaky (без трасс)

Проект: spring-petclinic[4]

```
@Test
void testProcessCreationFormWithBlankName() throws Exception {
    mockMvc
        .perform(post( uriTemplate: "/owners/{ownerId}/pets/new", TEST_OWNER_ID)
            .param( name: "name",  ...values: "\t \n")
            .param( name: "birthDate",  ...values: "2015-02-12"))
        .andExpect(model().attributeHasNoErrors( ...names: "owner"))
        .andExpect(model().attributeHasErrors( ...names: "pet"))
        .andExpect(model().attributeHasFieldErrors("pet", "name"))
        .andExpect(model().attributeHasFieldErrorCode("pet", "name", "required"))
        .andExpect(status().isOk())
        .andExpect(view().name( expectedViewName: "pets/createOrUpdatePetForm"));
}
```

Flaky tests analysis

After going through **all** of the PetControllerTests, there is **no** evidence of flakiness in testProcessCreationFormWithBlankName (or in any of the other tests). Each one

1. Is purely in-memory (no network or file I/O),
2. Has no randomness or threading/race conditions,
3. Uses a fresh Mockito stub for OwnerRepository in a `@BeforeEach`, and
4. Exercises Spring MVC's binder/validator deterministically.

Why testProcessCreationFormWithBlankName Is Not Flaky

- It always POSTS a blank name (`"\t \n"`) and expects a validation error on the `pet.name` field.
- The test setup consistently provides an Owner with two pets, so the "duplicate name" logic never interferes here.
- There is no time-based or external dependency; each run starts from the same state.

[4] https://github.com/spring-projects/spring-petclinic