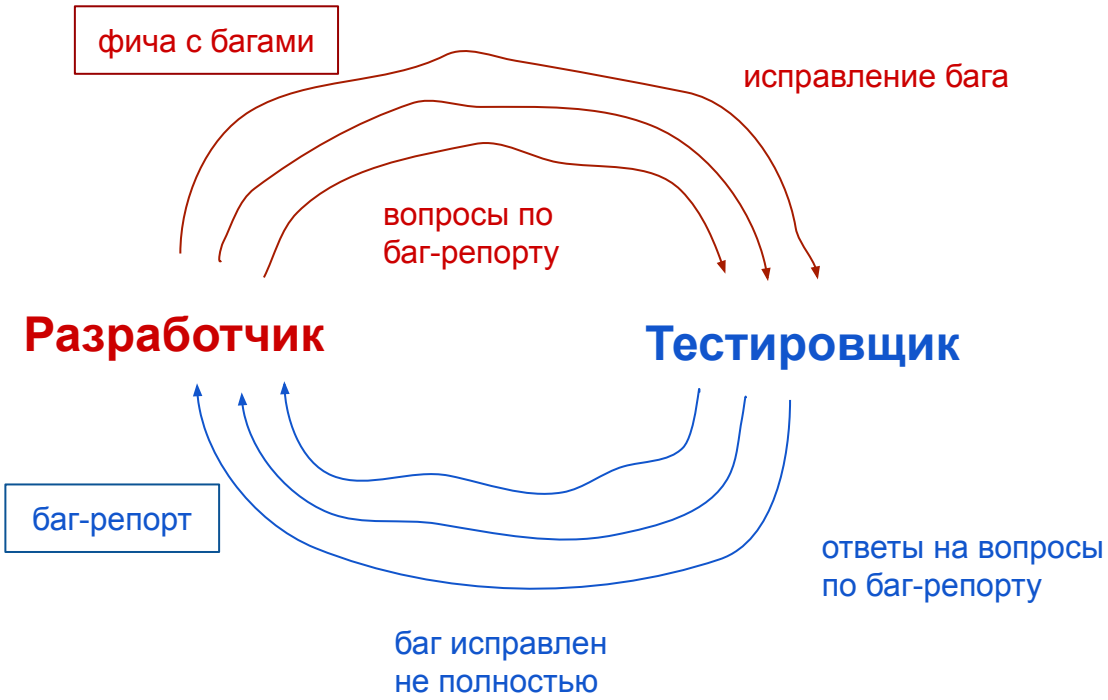
Как разработчику находить максимум багов за минимум времени?

Николай Москаленко

Проблема

Каждый баг, допущенный разработчиком, порождает дополнительные итерации взаимодействия между членами команды.

Каждый баг порождает дополнительные итерации взаимодействия между разработчиком и тестировщиком



Каждый баг, допущенный разработчиком, не зависимо от его критичности дополнительно утилизирует ресурсы команды, а также увеличивает время доставки продукта
до прода.

Решение

Разработчику следует тестировать свой код до того, как за дело возьмется тестировщик.

У разработчиков и тестировщиков разные цели тестирования

Тестировщик в тестировании

Цель - найти баги до того, как их найдут пользователи.

Важно качество багов!

Разработчик в тестировании

Цель - найти баги до того, как их найдут тестировщики.

Важно количество багов!
Но почему?

Разработчик:

“Я не могу тратить много времени на тестирование, к тому же я не умею тестировать.”

Сколько времени разработчик готов потратить на тестирование?

15 минут? Час? Два? Не вопрос!
Давайте отталкиваться от того количества времени, которым вы располагаете!

Научиться тестировать можно за 30 минут

Ликбез по тестированию для разработчиков

Как разработчику следует понимать, что такое тестирование?

Это эксплуатация программного обеспечения с целью нахождения дефектов.

В чем разница между мышлением тестировщика и разработчика?

Разработчик может посмотреть исходный код, ведь чаще всего он его и писал. “Зачем я буду это тестировать, если я вижу, что в коде все ОК?”

Тестировщик любое ПО воспринимает, как черный ящик. Он выполняет в приложении набор сценариев и сравнивает ожидаемый результат с фактическим. “По умолчанию все работает НЕКОРРЕКТНО, пока я не удостоверюсь в обратном”

Наша задача - найти баланс между этими двумя подходами.

Что такое баг?

Это несоответствие фактического результата работы программного обеспечения ожидаемому.

Как тестировать быстро и находить много багов?

Писать юнит-тесты? Интеграционные? End-to-end? Тестировать вручную?...

Как тестировать быстро и находить много багов?

Писать юнит-тесты? Интеграционные? End-to-end? Тестировать вручную?...

Как тестировать быстро и находить много багов?

~~Писать юнит-тесты? Интеграционные? End-to-end? Тестировать вручную?~~

Это не важно! Вид тестирования - это всего лишь инструмент поиска багов. Ключ к быстрому нахождению багов - определять “правильные” тестовые сценарии.

Какие тестовые сценарии “правильные” для разработчика?

Это тест-кейсы, которые:

1. находят баги
2. их можно быстро написать и выполнить
3. поддаются автоматизации

Поговорим об этом более подробно на примере приложения.

Пример приложения

Разработчик:

“У меня нет времени проверять 100500 комбинаций.
Пусть это делает тестировщик”

Разработчик, используй подход

S.C.A.R.E.

и пиши только те тест-кейсы, которые вероятнее всего найдут баги за минимальное время.

S	Spots
C	Common scenarios
A	Alternative scenarios
R	Ranking
E	Equivalence

S

spots

Определяем места в приложении, где наибольшая вероятность появления багов с помощью экспертной оценки и разделения функционала на блоки.

Используем экспертную/статистическую оценку вероятности нахождения бага.

По моему опыту, ТОП 3 мест с наибольшим кол-вом багов - это:

1. обработка пользовательского ввода
2. взаимодействие с внешними интеграционными точками
3. бизнес-логика (условия, расчеты)

В нашем случае это будут: валидатор, взаимодействие с бэк-системой.

C

common scenarios

Выделяем основные сценарии, т.е. сценарии наиболее часто используемые со стороны пользователей.

Например, в большинстве случаев, пользователи будут вводить свой существующий логин. Чаще всего встречается логин из 8-12 латинских символов в нижнем регистре.

Составляем список тест-кейсов на основные сценарии:

- Ввод логина стандартной длины с использованием наиболее типичных комбинаций символов
- Обработка ответа от бэк-системы, в случае, если пользователь найден
- Работа UI в самом часто используемом браузере и разрешении экрана

A

alternative scenarios

Выделяем альтернативные сценарии, т.е. сценарии которые будет выполнять меньшая часть пользователей, а также сценарии, провоцирующие систему на ошибку.

Например, указание несуществующего логина, а также логина нестандартной длины или с редкой комбинацией символов. Также помним, что бэк-система может вернуть сообщение в некорректном формате.

Составляем список тест-кейсов на альтернативные сценарии:

- Ввод логина нестандартной длины с использованием редких комбинаций символов
- Обработка ответа от бэк-системы, в случае, если пользователь НЕ найден
- Обработка некорректного ответа от бэк-системы
- Работа UI в альтернативных браузерах и нестандартных разрешениях экрана

R

ranking

Ранжируем получившиеся тест-кейсы. Сортируем их по вероятности нахождения бага, функционалу, трудоемкости выполнения/написания.

В результате из всего списка отбираем ТОП N тест-кейсов, которые:

- с наибольшей вероятностью найдут баги
- затрагивают разные блоки функционала
- наименее трудоемкие для выполнения/написания
- наиболее приоритетные

E

equivalence

Это одно или несколько значений тестовых данных, **к обработки которых программное обеспечение применяет одинаковую логику.**

Разбиваем тесты на классы эквивалентности с сокращением их числа, но с сохранением покрытия требований.

За счет этого отсеивается огромное количество значений тестовых данных, использовать которые бессмысленно.

Пример комбинации нескольких классов эквивалентности в одном тесте: символы латинского алфавита в нижнем и верхнем регистре с цифрами, тире и нижним подчеркиванием длиной 32 символа.

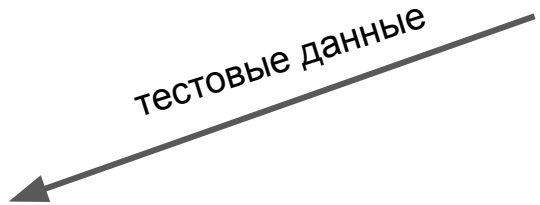
Разработчик:

“А есть ли способ написать 50 тестов, но при этом потратить времени, как если бы я написал 10 тестов?”

BDD подход с использованием библиотеки jest-cucumber

Feature-файл с таблицей параметров

```
1  Feature: Sum Pairs
2    It sums pairs of numbers
3
4    Scenario Outline: adds x + y to equal sum
5      Given x is <x>
6      When add <y>
7      Then the sum is <sum>
8
9    Examples:
10     | x | y | sum |
11     | 1 | 2 | 3 |
12     | 0 | 1 | 1 |
13     | -1 | 1 | 0 |
14     | 1 | 0 | 1 |
```



Реализация теста с помощью jest-cucumber

```
import { defineFeature, loadFeature } from 'jest-cucumber';
import sum from '../../utils/sum';

const feature = loadFeature('./src/features/sum_pairs.feature');

defineFeature(feature, test => {
  test('adds x + y to equal sum', ({ given, when, then }) => {
    let x: number;
    let z: number;

    given(/^x is (.*)$/, (givenXStr: string) => {
      const givenX = parseInt(givenXStr, 10);
      x = givenX;
    });

    when(/^add (.*)$/, (givenYStr: string) => {
      const givenY = parseInt(givenYStr, 10);
      z = sum(x, givenY);
    });

    then(/^the sum is (.*)$/, (givenSumStr: string) => {
      const givenSum = parseInt(givenSumStr, 10);
      expect(z).toBe(givenSum);
    });
  });
});
```

Feature-файл, где есть взаимодействие с UI

```
Feature: Counter
  It displays an incrementing / decrementing counter starting at 0

  Scenario: showing 0 initially
    Given mount counter
    When initially
    Then showing 0

  Scenario: clicking - decrements
    Given mount counter
    When clicking -
    Then showing -1

  Scenario: clicking + increments
    Given mount counter
    When clicking +
    Then showing 1
```

Реализация теста с помощью jest-cucumber

```
1  import { defineFeature, loadFeature } from 'jest-cucumber';
2  import React from 'react';
3  import TestRenderer from 'react-test-renderer';
4  import Counter from '../../components/Counter';
5
6  const feature = loadFeature('./src/features/counter.feature');
7
8  defineFeature(feature, test => {
9    test('showing 0 initially', ({ given, when, then }) => {
10     let testInstance: TestRenderer.ReactTestInstance;
11
12     given('mount counter', () => {
13       const testRenderer = TestRenderer.create(<Counter />);
14       testInstance = testRenderer.root;
15     });
16
17     when('initially', () => {
18       // INITIALLY
19     });
20
21     then('showing 0', () => {
22       const divInstance = testInstance.findByProps({ id: 'rootCounter' });
23       expect(divInstance.props.children).toBe('0');
24     });
25   });
26
```

Демо

Разработчик:

“Есть ли способ выполнить тест, если бэк-система не возвращает необходимые данные/недоступна”

Используйте заглушки!

“Про это многие знают, но мало кто применяет на практике”

Заглушки в юнит-тестах с помощью Jest

Заглушить внешнюю точку интеграции - это просто (пример будет изменен под приложение)

biblia/__mocks__/axios.js

```
const axios = {
  get: jest.fn(() => Promise.resolve({ data: {} }))
};

module.exports = axios;
```

biblia/__tests__/search.test.js

```
const search = require("../js/search");
const mockAxios = require("axios")

test("fetches results from google books api", () => {
  mockAxios.get.mockImplementationOnce(() =>
    Promise.resolve(dummy_response_data_here)
  );

return search.fetchBooks().then(response => {
    expect(response).toEqual();
  });
});
```

Демо

Итог

- Всегда тестируйте свой код перед тем, как отдать его тестировщику.
- Для выявления тестовых сценариев, находящих баги за короткое время используйте метод S.C.A.R.E.
- Не забывайте использовать заглушки в интеграционных и юнит-тестах - это не сложно.
- Попробуйте писать юнит тесты с помощью Gherkin

Ссылка на репозиторий

https://github.com/AToutLeMonde/js-testing-sample