Лекция 11

- Код-ревью
- Нестабильные тесты

Зачем нужен кодревью?

Базовые вещи, которые обычно упоминают разработчики:

- ✓ улучшить качество кода (стиль, понятность, поддержка)
- ✓ найти ошибки
- ✓ передать знания
- ✓ повысить взаимную ответственность
- ✓ предложить лучшее решение
- ✓ проверить на соответствие гайдлайнам

Базовый контрольный список проверки кода

При просмотре кода задайте себе следующие основные вопросы:

- ✓ Могу ли я легко **понять** код?
- ✓ Соответствует ли код **стандартам/рекомендациям по кодированию** ?
- ✓ **Дублируется** ли один и тот же код более двух раз?
- ✓ Могу ли я легко **провести модульное тестирование/отладку** кода, чтобы найти основную причину?
- ✓ Является ли эта функция или класс **слишком большим** ? Если да, то имеет ли функция или класс слишком много обязанностей?

Подробный контрольный список проверки кода

https://www.evoketechnologies.com/blog/code-review-checklist-perform-effective-code-reviews/

Форматирование кода

- ✓ Используйте выравнивание (левое поле), правильное пустое пространство.
- ✓ Убедитесь, что соблюдены надлежащие соглашения об именовании (Pascal, CamelCase и т. д.).
- ✓ Код должен помещаться на стандартном 14-дюймовом экране ноутбука. Не должно быть необходимости прокручивать страницу по горизонтали, чтобы просмотреть код. На 21-дюймовом мониторе другие окна (панель инструментов, свойства и т. д.) могут быть открыты во время изменения кода, поэтому всегда пишите код, имея в виду 14-дюймовый монитор.
- ✓ Удалите закомментированный код, так как он всегда является блокировщиком при просмотре кода. Закомментированный код можно получить из системы управления исходным кодом (например, SVN), если это необходимо.

Лучшие практики кодирования

- ✓ Никакого жесткого кодирования, используйте константы/значения конфигурации.
- ✓ Группировка схожих значений под перечислением (enum).
- ✓ Комментарии – Не пишите комментарии о том, что вы делаете, вместо этого пишите комментарии о том, почему вы это делаете. Укажите о любых хаках, обходных путях и временных исправлениях. Кроме того, упоминайте в комментариях к вашим to-do ожидающие выполнения задачи, которые можно легко отследить.
- ✓ Избегайте использования нескольких блоков if/else.
- ✓ По возможности используйте возможности фреймворка вместо написания собственного кода.

Подробнее см. в хорошей статье https://www.evoketechnologies.com/blog/code-review-checklist-perform-effective-code-reviews/

Основной критерий: код должен быть вам понятным.

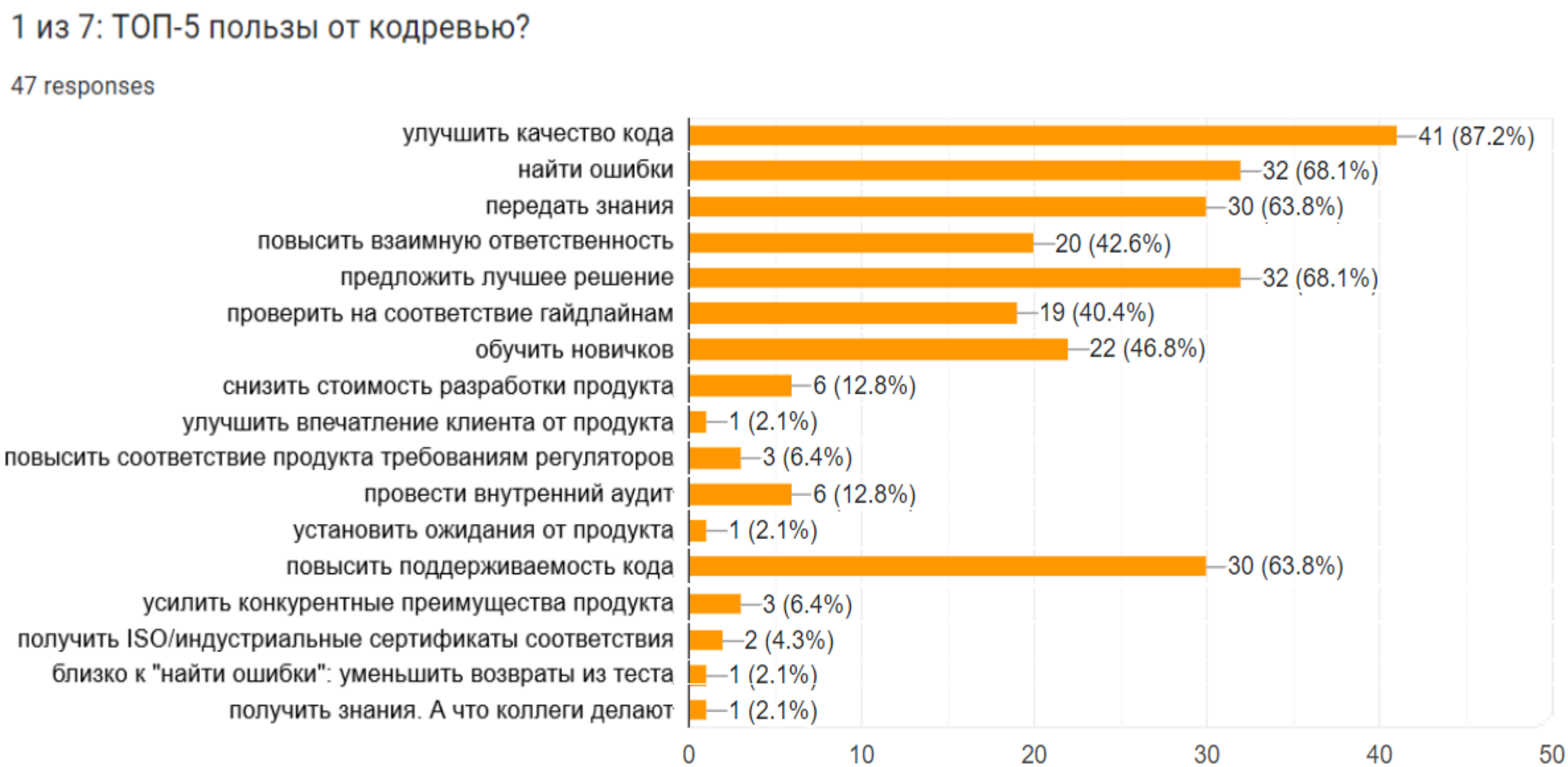
«Пишите код так, как будто поддерживать его будет склонный к насилию психопат, который знает, где вы живёте».

Типовые проблемы и решения

1. Код с душком
2. Дизайн с душком
3. Рефакторинг
4. Антипаттерны
5. Паттерны (шаблоны) проектирования
6. SOLID, DRY, KISS, YAGNI, BDUF, APO, Бритва Оккама
7. Уязвимости в коде
8. Гниение кода
9. Тёмные паттерны
10. Статический анализ кода

Подробнее про каждый пункт со ссылками на источники

https://habr.com/ru/articles/741186/

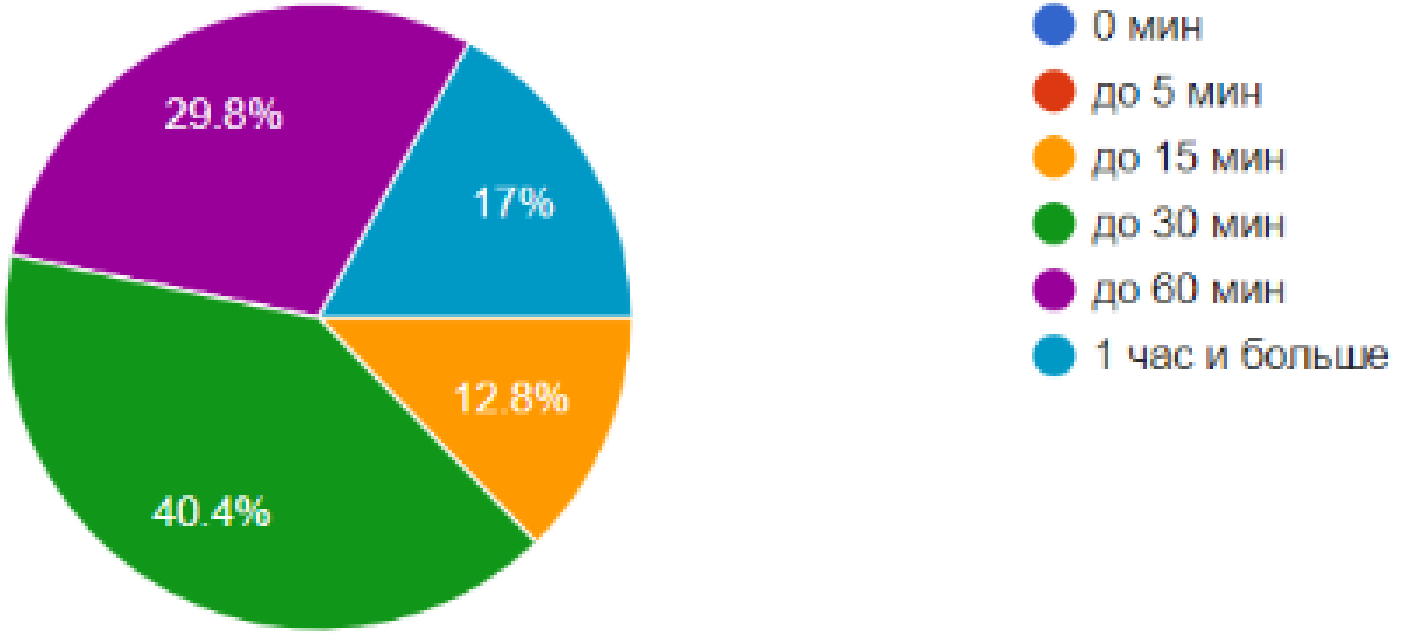


https://habr.com/ru/articles/741186/

https://habr.com/ru/articles/741186/

4 из 7: Сколько времени тратите на кодревью в-среднем за день?

47 responses



https://habr.com/ru/articles/741186/

Чтобы сделать процесс проверки кода более интересным и увлекательным, напомните разработчикам соответствующие цитаты/пословицы:

- ✓ «*Любой глупец может написать программу, понятную компьютеру, но только хорошие программисты пишут код, понятный людям*» — Мартин Фаулер
- ✓ «*Измерять прогресс программирования по строкам кода — это все равно, что измерять прогресс в строительстве самолета по весу*». — Билл Гейтс
- ✓ «*При отладке новички вставляют корректирующий код; эксперты удаляют дефектный код*». – Ричард Паттис

Проблемы при проведении код-ревью:

- ✓ Сильная эмоциональна привязанность к своему коду
- ✓ Ментальная модель «Мы против Них»

Хорошая практика:

- ✓ Перед началом установить стандарты оформления кода и определиться с термином «Готово»
- ✓ Использовать чек-листы проверки кода
- ✓ Придерживаться методологии «Гуманных код-ревью».

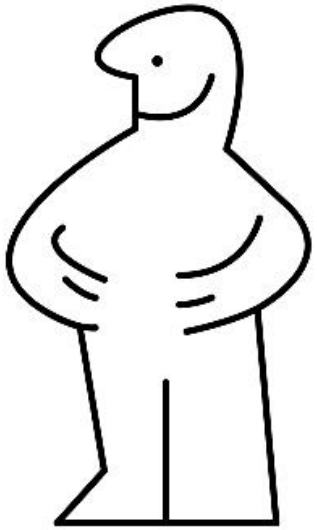
Методология «Гуманных код-ревью» ключевые принципы:

- ✓ Любой код, в том числе ваш, может быть лучше. Если кто-то хочет изменить что-то в вашем коде, то это **не нужно воспринимать как личное нападение**.
- ✓ **Речь идёт о коде, а не его авторе**.
- ✓ Никаких личных чувств. Избегайте мыслей вроде «...потому что мы всегда делали это так», «Я потратил на это уйму времени», «Это не я дурак, а ты», «Мой код лучше» и так далее. **Улыбнитесь, простите себя, взбодритесь и двигайтесь вперёд.** **Неудача – мать успеха** (кит. пословица)
- ✓ **Пусть все ваши комментарии будут позитивными и направлены на улучшение кода.** Будьте добры к его автору, но с ним не церемоньтесь.

Вместо того чтобы говорить: «**Ты пишешь код, как школьник**», попробуйте так: «**Мне сложно понять, что тут происходит**».
Чувствуете разницу? Первый вариант – личное оскорбление, а второй – конструктивная обратная связь.

Код-ревью авто-тестов

```
// Ждем обновления сообщения об ошибке
await bro.pause(1000);
```



Find in Files 45 matches in 29 files

bro.pause(

In Project Module Directory Scope /Users/dmitrytuchs/IdeaProj

```
await bro.pause(1000);
await bro.pause(1000);
await bro.pause(1000);
await bro.pause(250);
await bro.pause(2000);
await bro.pause(2000);
```

 **Bright_Translate** 29 мар в 16:00

Делаем код-ревью правильно

https://habr.com/ru/companies/ruvds/articles/803127/

В начале своей карьеры мы работали над платформой сентимент-анализа для социальных сетей. Наша команда состояла из семи человек. Мы были молоды и полны энтузиазма. Наш девиз можно было описать как: «Мы гибкие, быстрые и всё ломаем!». Да, мы действительно гордились своей скоростью. **Код-ревью? Я вас умоляю. Мы считали эту практику бюрократическим пережитком корпоративного мира.**

И что вы думаете? **Через несколько месяцев** наша база кода стала подобна минному полю. Проблема заключалась в том, что **никто не мог понять код, написанный другими**. У нас во многих местах дублировалась логика, и в модулях использовались разные стили кода.

Тогда до нас дошло! Нужно взять всё под контроль. **Код-ревью реально помогают сохранять код читаемым, обслуживаемым и масштабируемым.**

Итак, в двух словах: ***если вы не проводите код-ревью****, или делаете их «для галочки», то обрекаете себя на боль, пусть не сразу, но в конечном итоге однозначно*. **Это можно сравнить с возведением дома на фундаменте из песка. Какое-то время он, может, и простоит, но явно недолго.** А в мире стартапов второго шанса у вас может уже не быть.

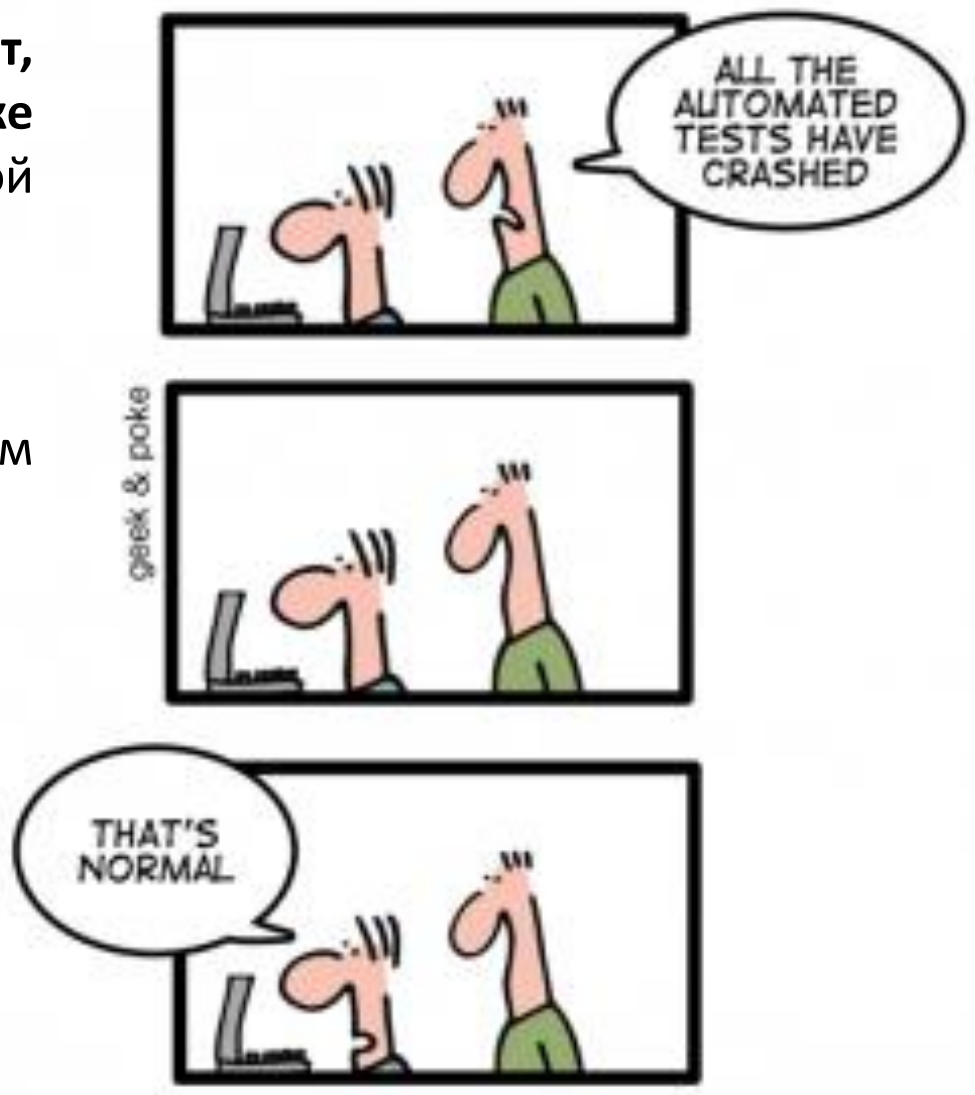
Нестабильные тесты

Проблемы авто-тестов

Нестабильные тесты

Flaky-тест в ИТ-тестирования **означает нестабильный тест, который иногда “pass”, иногда “fail” учитывая одну и ту же конфигурацию теста**, и трудно понять, по какой закономерности.

- Такие тесты сложно обнаружить.
- На такие тесты тратится много времени и ресурсов.
- Возникает задержка, пока команда не разберется, в чем дело.



НО такие тесты могут помочь найти ошибки, которые не могли бы быть обнаружены без такой нестабильности. Можно раскрыть некоторые плохие инфраструктуры или конструкции тестовой среды.

Парадокс: подавление всех нестабильных тестов не является полностью лучшим решением (дорого) и в целом недостижимо. В действительности **существует баланс, порог толерантности.**

Классификация нестабильных тестов это первостепенный шаг, позволяющий лучше использовать ресурсы (людей, время, CI и стоимость).

Цель данной классификации - иметь возможность разделить ненадежные тесты на группы в соответствии с их происхождением, чтобы расставить их по приоритетам.

Можно заметить две категории, которые кажутся более очевидными, чем другие, в программных проектах:

- ✓ **Независимые нестабильные тесты**: Тест, который дает сбой независимо, вне или внутри тестового набора. Благодаря легкости воспроизводимости их легче заметить, отладить и решить.
- ✓ **Системные нестабильные тесты**: тесты, которые не срабатывают из-за проблем с окружением, общим состоянием или даже из-за их порядка в тестовом наборе. Их гораздо сложнее обнаружить и отладить, поскольку их поведение может меняться вместе с эволюцией системы или рабочего процесса.

Наиболее часто встречающиеся причины нестабильности тестов:

- ✓ **Недостаточная изоляция:** тесты, не использующие копии ресурсов, могут привести к условиям гонки или конкуренции ресурсов при параллельном запуске. Кроме того, тесты, которые изменяют состояние системы или взаимодействуют с базами данных, всегда должны очищаться после использования.
- ✓ **Параллелизм**: несколько параллельных потоков взаимодействуют нежелательным образом
- ✓ **Зависимость от порядка тестирования**: тесты, которые могут завершиться неудачей или успехом в зависимости от порядка их выполнения в тестовом наборе.
- ✓ **Сеть:** Тесты, полагающиеся на сетевое подключение, которое не является параметром, который можно полностью контролировать.
- ✓ **Время:** Тесты, основанные на системном времени, могут быть недетерминированными и их трудно воспроизвести в случае сбоя.
- ✓ **Асинхронное ожидание, вызовы остались несинхронизированными или плохо синхронизированными:** тесты, которые выполняют асинхронные вызовы, но не ждут должным образом результата. Тесты должны избегать любого фиксированного периода сна. Время ожидания может различаться в зависимости от среды.
- ✓ **Зависимость от среды:** результаты теста могут различаться в зависимости от среды, в которой он выполняется.

Наиболее часто встречающиеся причины нестабильности тестов

(продолжение):

- ✓ **Операции ввода/вывода:** Тест может работать нестабильно, если он неправильно собирает мусор и не закрывает ресурсы, к которым он получил доступ.
- ✓ **Доступ к системам или сервисам, которые не являются абсолютно стабильными:** лучше использовать фиктивные сервисы как можно полнее, чтобы избежать зависимости от внешних, неконтролируемых факторов.
- ✓ **Использование генерации случайных чисел:** При использовании генерации случайных чисел или других объектов полезно регистрировать сгенерированное значение, чтобы избежать ненужного сложного воспроизведения сбоя теста.
- ✓ **Неупорядоченные коллекции:** не делайте предположений о порядке элементов в неупорядоченном объекте.
- ✓ **Жестко заданные значения:** тест, использующий постоянные значения, в которых элементы или механика могут изменяться со временем.
- ✓ **Слишком узкий диапазон тестирования:** при использовании выходного диапазона для утверждения может оказаться, что не все результаты были рассмотрены, и в случае их возникновения тест будет провален.

Итак, что делать, чтобы нестабильных тестов было меньше:

- тесты должны быть написаны в правильном слое "той самой пирамиды": **чем ближе слой к модульным (юнит) тестам (а лучше именно в них), тем меньше шансов на моргания, потому что зависимостей меньше.**
- **основные причины нестабильсти это асинхронные операции** (async wait), многопоточность (concurrency), порядок тестов, утечка ресурсов, проблемы с зависимостями (сеть, время). Поэтому, чем меньше этого в тестах, тем они стабильнее.
- основной объем бизнес-логики проверяем максимально близко к месту логики и **с максимальным количеством замокированных зависимостей** (но не переусердствуйте, а то будут другие проблемы)

Что делать, если они появились?

- если сейчас нет возможности разобраться с ошибкой, **переместите этот тест в "карантин", чтобы позже с ним разобраться**. Не надо держать в наборе запускаемых тестов тот, доверия к результатам которого нет.
- **активно используйте трейсинг (логирование)** в тестах и продакшен-коде для того, чтобы воспользоваться ими при расследовании. Совет: здорово, если у вас есть возможность "объединить" логи тестируемой системы с логами тестов. Мы активно использовали запись меток о начале/завершении теста в продакшен логах приложения. Очень помогало.
- для UI-тестов имейте возможность **включить запись видео или скриншоты в момент проверки**
- попробуйте переместить моргающую проверку на другой слой пирамидки
- **если тест не поддается, подумайте, может стоит его удалить?** Все равно смысла от него немного, особенно если думать про него не "случайно упавший", а "случайно успешный". Ну и в целом "Flaky tests are worse than _no_ tests".
- **иногда советуют перезапускать упавшие тесты в надежде на удачу.** В целом рабочий способ, но не надо им злоупотреблять. Он хорошо помогает с подтверждением проблемы и поиском test war. Но обнаруженные проблемы, например, с медленной инфраструктурой/сетью, особенностями фреймворков важно всегда фиксировать и планировать время на исправление.

Допустимый процент нестабильных тестов зависит от контекста проекта, но есть общепринятые практические рекомендации:

1. Жесткие стандарты (критичные системы)

0-1% нестабильных тестов

Примеры:

- Авиация, медицина, финансы (где цена ошибки крайне высока)
- Ядро ОС, криптография

  Действия:

- Флаки исправляются в приоритетном порядке
- Падение = блокировка релиза

2. Стандартные коммерческие проекты

До 5% нестабильных тестов

Примеры:

- Веб-приложения, мобильные приложения
- Enterprise-софт

Действия:

- Допускаются маркированные `@flaky` тесты
- Автоматический перезапуск (1-3 раза)

- **Цель:** стремиться к **0%**, но реально **до 5%** — разумный компромисс.
- **Важно:** даже 1% флаков в **критичном модуле** — повод для срочного исправления.

Критерии приемлемости

1. **Не маскируют реальные баги**
    - Если флак мешает обнаруживать регрессии — это красная зона.
2. **Не замедляют процесс**
    - Многократные перезапуски не должны увеличивать время CI/CD.
3. **Известны и контролируются**
    - Есть список ожидаемо нестабильных тестов с пояснениями.

Когда можно игнорировать?

- Если тест падает **редко** и баг **некритичный**.
- Если **стоимость исправления** выше, чем потенциальный урон.

Экскурс в «святая святых» ОК: как мы пишем и ревьюим код автотестов

14.08.2024 00:00

https://software-testing.ru/library/testing/testing-automation/4246-ok

Материал подготовлен по мотивам доклада руководителя команды автоматизации тестирования ОК Эмилии Куцаревой и младшего инженера по автоматизации тестирования соцсети Евгения Буровникова на ИТ-конференции «Стачка».

ОК — одна из самых популярных социальных сетей в рунете, которая представлена на всех возможных платформах (web, mobile web, API, android, iOS).
Наш продукт высоконагружен, имеет сложный бэкенд и сотни сервисов.
Например, у него «под капотом»: 50 тысяч Docker-контейнеров, 1 эксабайт данных и обработка данных в 7 дата-центрах.

Так, сейчас у нас более 10 тысяч автотестов:

- web — около 3150;
- API — около 2500 (+1500 автосгенерированных);
- Android — около 1400;
- mobile web — около 1200;
- iOS — около 950.