

# Поговорим про performance-тестирование

Андрей Акиншин, JetBrains

DotNext 2017 Moscow

# О чём будем разговаривать

**Мы хотим:**

## Мы хотим:

- 1 Не допустить performance-деградаций

## Мы хотим:

- 1 Не допустить performance-деградаций
- 2 Если допустили, то вовремя об этом узнать

## Мы хотим:

- 1 Не допустить performance-деградаций
- 2 Если допустили, то вовремя об этом узнать
- 3 Снизить количество ошибок первого рода  
(Проблемы нет, а мы думаем, что есть)

## Мы хотим:

- 1 Не допустить performance-деградаций
- 2 Если допустили, то вовремя об этом узнать
- 3 Снизить количество ошибок первого рода  
(Проблемы нет, а мы думаем, что есть)
- 4 Снизить количество ошибок второго рода  
(Проблема есть, а мы думаем, что нет)

## Мы хотим:

- 1 Не допустить performance-деградаций
- 2 Если допустили, то вовремя об этом узнать
- 3 Снизить количество ошибок первого рода  
(Проблемы нет, а мы думаем, что есть)
- 4 Снизить количество ошибок второго рода  
(Проблема есть, а мы думаем, что нет)
- 5 Автоматизировать всё, что можно

**RD**  
—  
**Rider**  
New .NET IDE

Cross-platform .NET IDE  
inspired by IntelliJ IDEA  
and ReSharper

The lower portion of the slide features a detailed graphic of a blue printed circuit board (PCB) with intricate orange and yellow traces. Two prominent black integrated circuit (IC) chips are positioned in the center. The chip on the left is labeled 'R#' in white, and the chip on the right is labeled 'IJ' in white. The 'R#' chip is connected to a pink and purple geometric shape, while the 'IJ' chip is connected to a red and orange geometric shape. The overall aesthetic is modern and technical, emphasizing the software's foundation in hardware and its lineage from IntelliJ IDEA and ReSharper.

## Rider

- Файлов: 200'000+
- Строчек кода: 20'000'000+
- Тестов: 100'000+
- Зависимости: IntelliJ IDEA + ReSharper
- Рантаймы внутри: JVM, .NET Framework/Mono
- Целевые ОС: Windows, Linux, macOS

## Rider

- Файлов: 200'000+
- Строчек кода: 20'000'000+
- Тестов: 100'000+
- Зависимости: IntelliJ IDEA + ReSharper
- Рантаймы внутри: JVM, .NET Framework/Mono
- Целевые ОС: Windows, Linux, macOS

*Хочется, чтобы Rider работал очень быстро...*

# Заметка 1

## Источники performance-данных

# Источники performance-данных: крупно

## «Специальные» performance-тесты

# Источники performance-данных: крупно

## «Специальные» performance-тесты

- Микробенчмарки 

# Источники performance-данных: крупно

## «Специальные» performance-тесты

- Микробенчмарки 
- Холодный и горячий старт

# Источники performance-данных: крупно

## «Специальные» performance-тесты

- Микробенчмарки 
- Холодный и горячий старт
- Performance-тесты на GUI

# Источники performance-данных: крупно

## «Специальные» performance-тесты

- Микробенчмарки 
- Холодный и горячий старт
- Performance-тесты на GUI
- Нагрузочное тестирование (Яндекс.Танк, JMeter и т.д.)

# Источники performance-данных: крупно

## «Специальные» performance-тесты

- Микробенчмарки 
- Холодный и горячий старт
- Performance-тесты на GUI
- Нагрузочное тестирование (Яндекс.Танк, JMeter и т.д.)
- Автоматизированный поиск точек отказа (capacity planning)

## «Специальные» performance-тесты

- Микробенчмарки 
- Холодный и горячий старт
- Performance-тесты на GUI
- Нагрузочное тестирование (Яндекс.Танк, JMeter и т.д.)
- Автоматизированный поиск точек отказа (capacity planning)
- Тестирование асимптотики

# Источники performance-данных: крупно

## «Специальные» performance-тесты

- Микробенчмарки 
- Холодный и горячий старт
- Performance-тесты на GUI
- Нагрузочное тестирование (Яндекс.Танк, JMeter и т.д.)
- Автоматизированный поиск точек отказа (capacity planning)
- Тестирование асимптотики
- ...

# Источники performance-данных: крупно

## «Специальные» performance-тесты

- Микробенчмарки 
- Холодный и горячий старт
- Performance-тесты на GUI
- Нагрузочное тестирование (Яндекс.Танк, JMeter и т.д.)
- Автоматизированный поиск точек отказа (capacity planning)
- Тестирование асимптотики
- ...

## «Обычные» тесты

- Все тесты, которые у вас уже есть ( $>10\text{ms}$ )

# Обычные тесты vs. Специальные тесты

Обычные тесты

Специальные тесты

# Обычные тесты vs. Специальные тесты

Обычные тесты

Случайные машины

Специальные тесты

Выделенные машины

# Обычные тесты vs. Специальные тесты

Обычные тесты

Случайные машины

Виртуальное окружение

Специальные тесты

Выделенные машины

Реальное железо

# Обычные тесты vs. Специальные тесты

Обычные тесты

Случайные машины

Виртуальное окружение

Легко писать

Специальные тесты

Выделенные машины

Реальное железо

Сложно писать

# Обычные тесты vs. Специальные тесты

Обычные тесты

Случайные машины

Виртуальное окружение

Легко писать

Много и бесплатно

Специальные тесты

Выделенные машины

Реальное железо

Сложно писать

Мало и дорого

## Много данных не бывает!

Можно собирать и анализировать:

- Время прохождения теста
- Время прохождения отдельных частей теста
- Железные данные: CPU, Memory, Disk, Network, ...
- Характеристики рантайма (например, GC.CollectionCount)
- По каждой метрике смотрим на:  
Min/Max/Mean/Median/Q1/Q3/Percentiles/Distribution

## Много данных не бывает!

Можно собирать и анализировать:

- Время прохождения теста
- Время прохождения отдельных частей теста
- Железные данные: CPU, Memory, Disk, Network, ...
- Характеристики рантайма (например, GC.CollectionCount)
- По каждой метрике смотрим на:  
Min/Max/Mean/Median/Q1/Q3/Percentiles/Distribution

*Не забываем про проблему множественных сравнений ...*

# Заметка 2

## История

Однажды, после перехода с Mono 4.9 на Mono 5.2 наши perf-тесты стали падать ...

Однажды, после перехода с Mono 4.9 на Mono 5.2 наши perf-тесты стали падать ...

```
private readonly IList<object> array = new string[0];
```

```
[Benchmark]
```

```
public int CountProblem() => array.Count;
```

Однажды, после перехода с Mono 4.9 на Mono 5.2 наши perf-тесты стали падать ...

```
private readonly IList<object> array = new string[0];
```

```
[Benchmark]
```

```
public int CountProblem() => array.Count;
```

	Linux	macOS
Mono 4.9	≈5ns	≈5ns
Mono 5.2	≈1400ns	≈2600ns

# Заметка 3

## Оборона против performance-деградаций



- **Merge robot**

Проверяем производительность на каждый мердж

- **Merge robot**

Проверяем производительность на каждый мердж

- **Daily tests**

Запускаем тесты каждый день

- **Merge robot**  
Проверяем производительность на каждый мердж
- **Daily tests**  
Запускаем тесты каждый день
- **Retrospective**  
Анализируем исторические данные

- **Merge robot**  
Проверяем производительность на каждый мердж
- **Daily tests**  
Запускаем тесты каждый день
- **Retrospective**  
Анализируем исторические данные
- **Check points**  
Проверяем опасные изменения

- **Merge robot**  
Проверяем производительность на каждый мердж
- **Daily tests**  
Запускаем тесты каждый день
- **Retrospective**  
Анализируем исторические данные
- **Check points**  
Проверяем опасные изменения
- **Pre-release**  
Проверяем регрессию перед самым релизом

# Сравнение видов обороны

Обнаружение	Точность	Процесс
-------------	----------	---------

# Сравнение видов обороны

	Обнаружение	Точность	Процесс
Merge robot	Вовремя	Средняя	Автоматический

# Сравнение видов обороны

	Обнаружение	Точность	Процесс
Merge robot	Вовремя	Средняя	Автоматический
Daily tests	Поздно	Хорошая	Полуручной
Retrospective	Слишком поздно	Отличная	Полуручной

# Сравнение видов обороны

	Обнаружение	Точность	Процесс
Merge robot	Вовремя	Средняя	Автоматический
Daily tests	Поздно	Хорошая	Полуручной
Retrospective	Слишком поздно	Отличная	Полуручной
Check points	Вовремя	Отличная	Ручной
Pre-release	Зависит	Отличная	Ручной

# Сравнение видов обороны

	Обнаружение	Точность	Процесс
Merge robot	Вовремя	Средняя	Автоматический
Daily tests	Поздно	Хорошая	Полуручной
Retrospective	Слишком поздно	Отличная	Полуручной
Check points	Вовремя	Отличная	Ручной
Pre-release	Зависит	Отличная	Ручной

И ещё несколько важных факторов:

- Время прогона
- Количество билд-агентов и их стоимость
- Спектр проблем, которые можно обнаружить

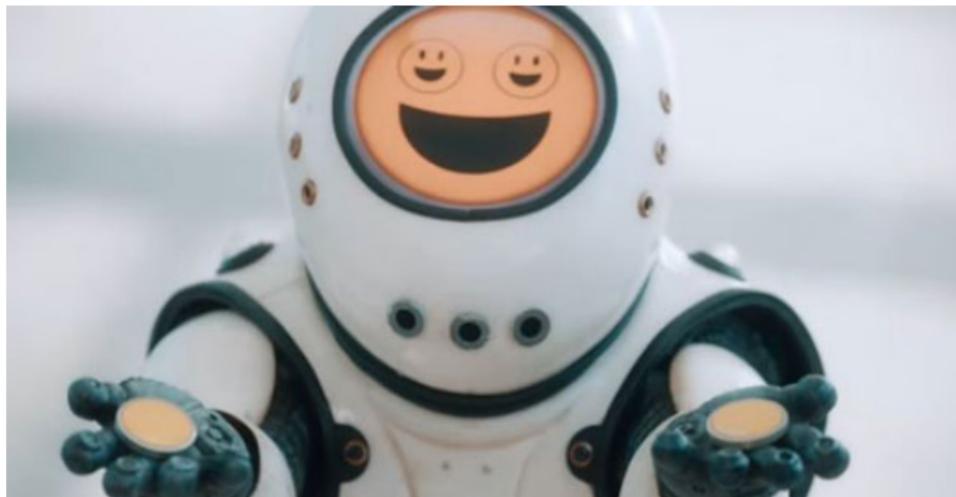
# Заметка 4

## Когда бить тревогу?



## Ручной

- Специальный программист, который мониторит тесты 24/7



## Автоматизированный

- Абсолютный timeout
- Относительный timeout
- Исторические данные и статистика



## Полуавтоматизированный

- Система, которая формирует список performance-проблем
- Специальный программист, который разбирает этот список

# Заметка 5

## Performance-аномалии

# Охота за performance-аномалиями

# Охота за performance-аномалиями

- **Performance-пространство** — множество всех performance-метрик тестов на всевозможных конфигурациях.

# Охота за performance-аномалиями

- **Performance-пространство** — множество всех performance-метрик тестов на всевозможных конфигурациях.
- **Performance-аномалия** — ситуация, при которой performance-пространство выглядит странно.

# Охота за performance-аномалиями

- **Performance-пространство** — множество всех performance-метрик тестов на всевозможных конфигурациях.
- **Performance-аномалия** — ситуация, при которой performance-пространство выглядит странно.
- **Performance-чистота** — отсутствие performance-аномалий

# Наведение performance-чистоты позволяет



# Наведение performance-чистоты позволяет



- Находить проблемы, на которые не стоят assert'ы

# Наведение performance-чистоты позволяет



- Находить проблемы, на которые не стоят assert'ы
- Ускорять билд

# Наведение performance-чистоты позволяет



- Находить проблемы, на которые не стоят assert'ы
- Ускорять билд
- Находить flaku-тесты

# Наведение performance-чистоты позволяет



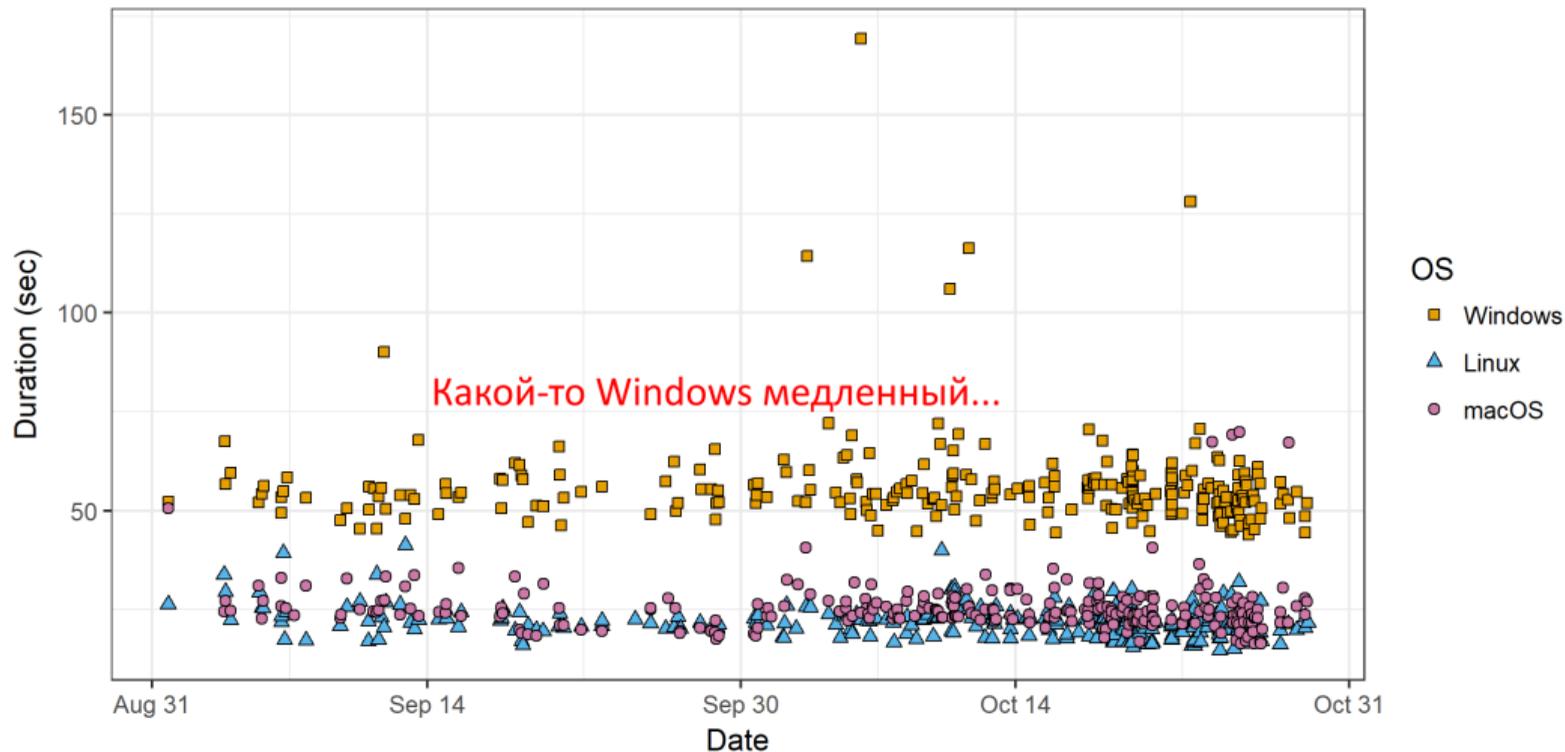
- Находить проблемы, на которые не стоят assert'ы
- Ускорять билд
- Находить flaky-тесты
- Уменьшать количество тестов, которые в будущем помешают найти действительно критичные проблемы

# Заметка 6

## Кластеризация

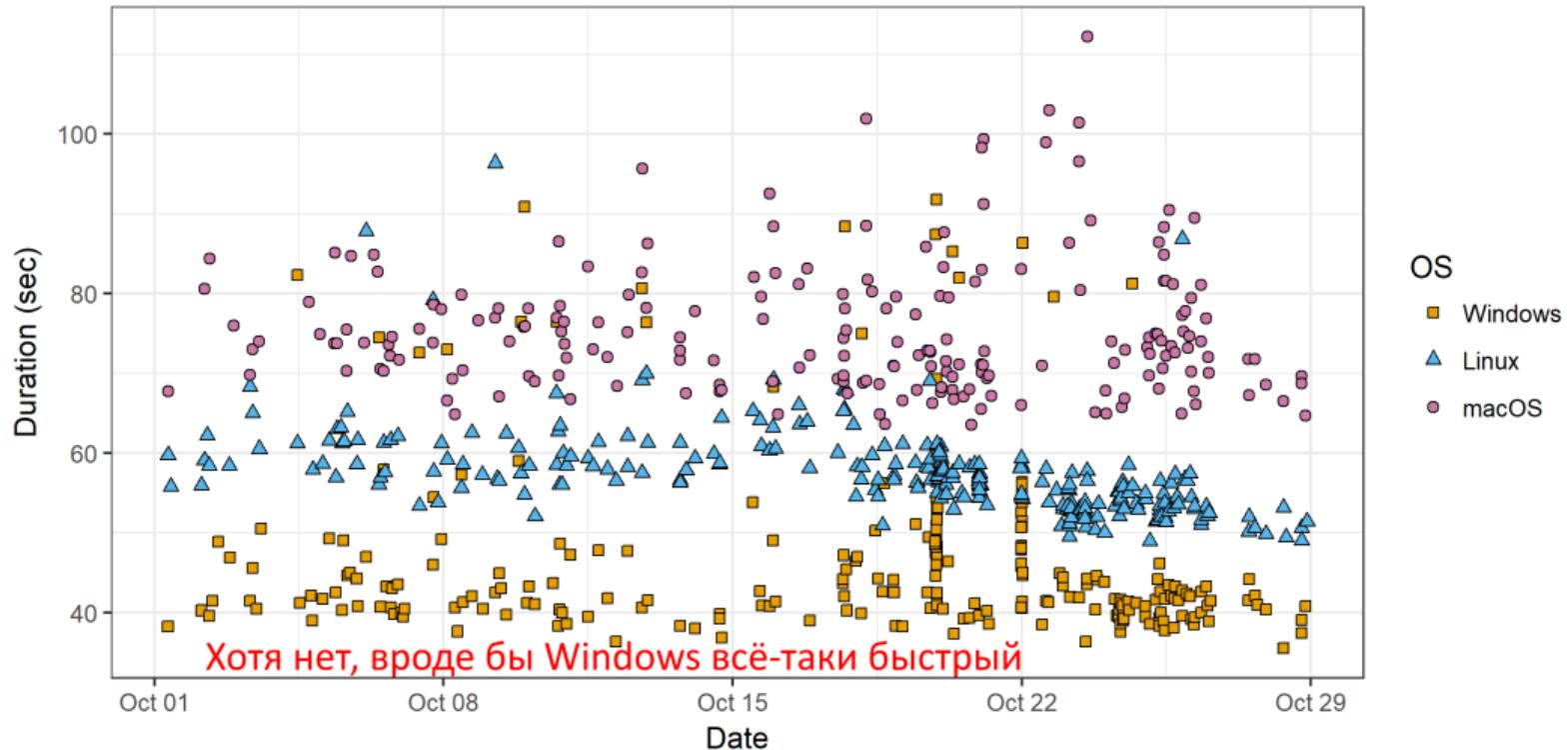
# Какая OS самая быстрая?

build\_BuildSolutionTest\_testSolutions\_SharpZipLib

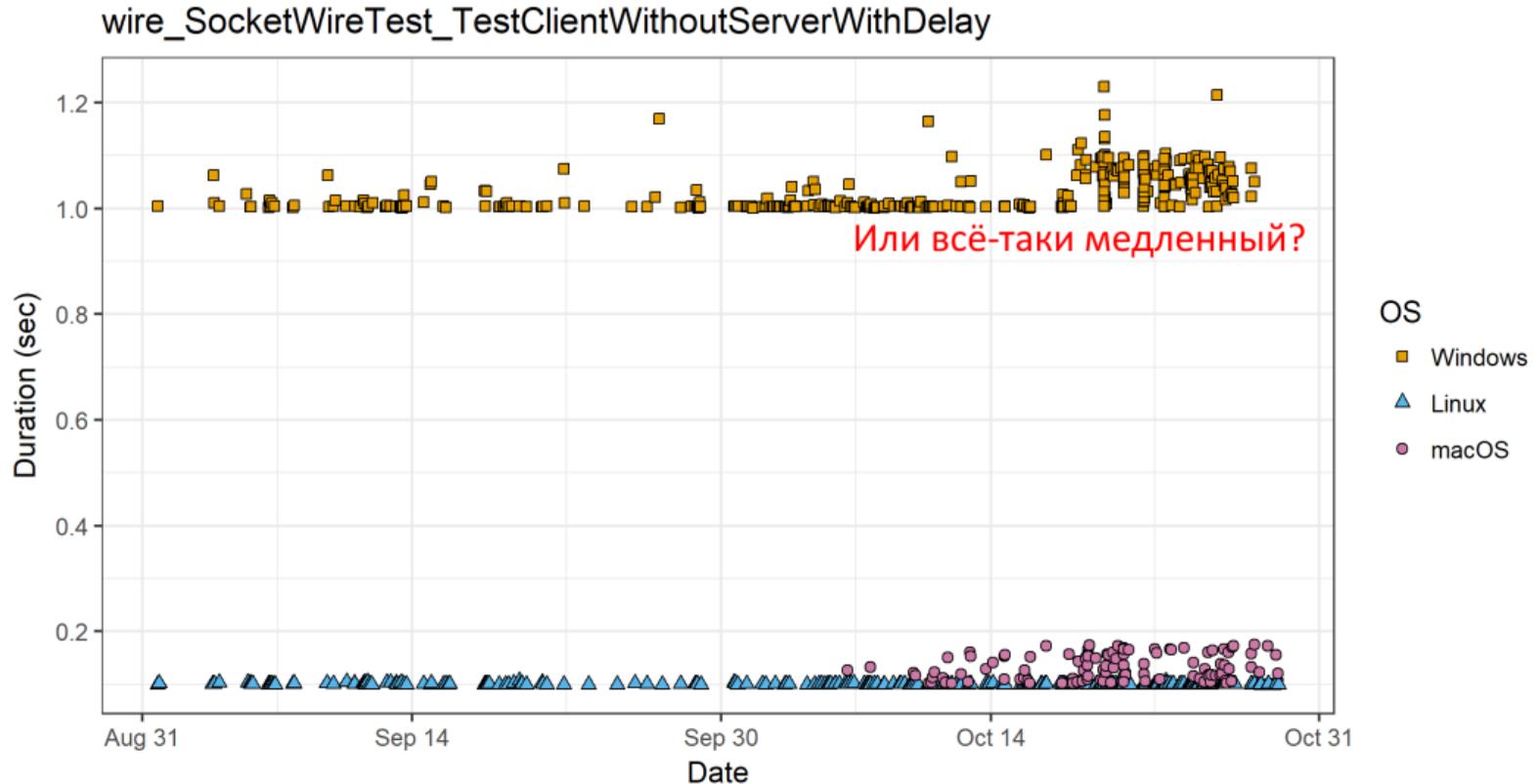


# Какая OS самая быстрая?

templates\_RiderTemplatesTestCore1\_mvcCoreTemplate



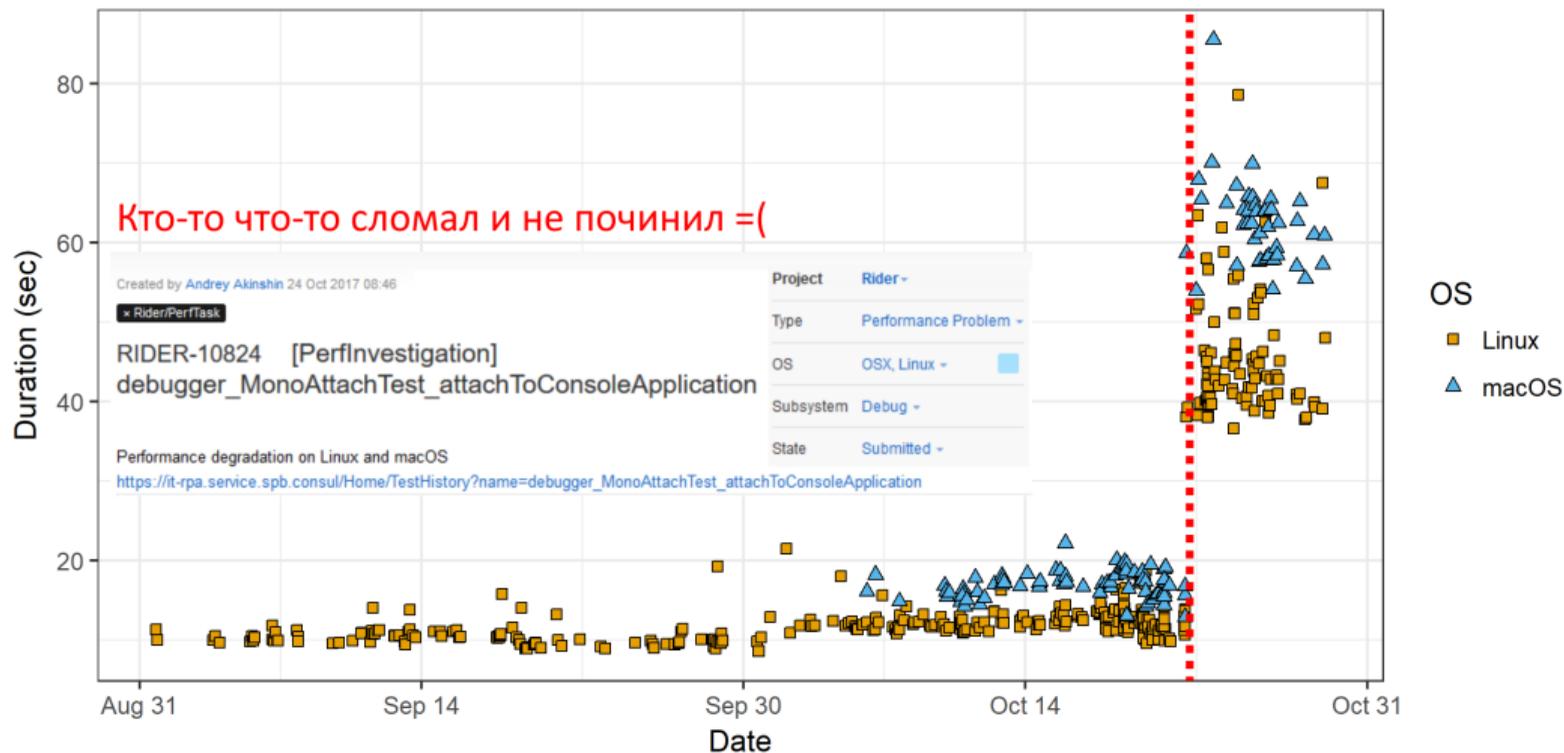
# Какая OS самая быстрая?



# Заметка 7

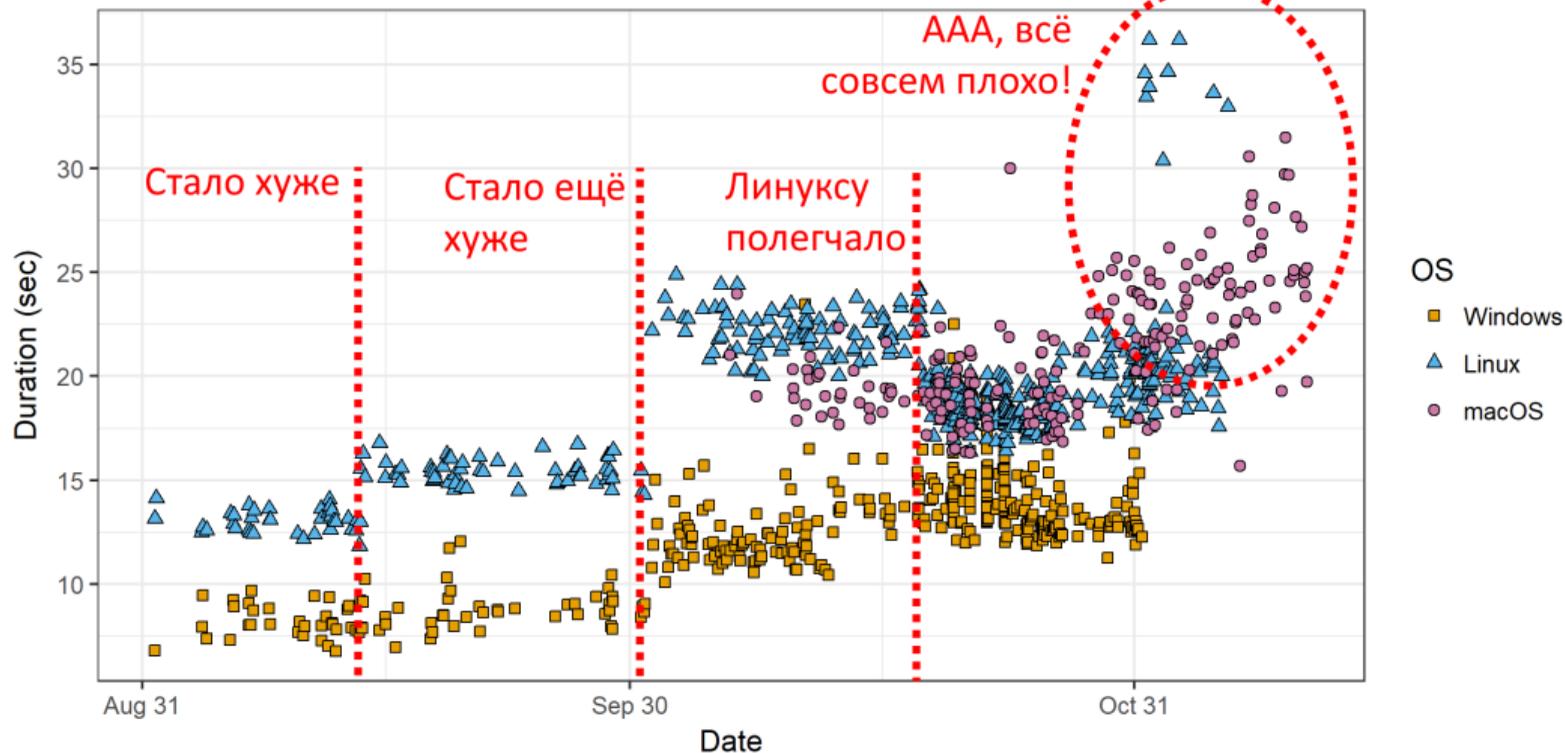
## Деградации: большие и маленькие

## debugger\_MonoAttachTest\_attachToConsoleApplication

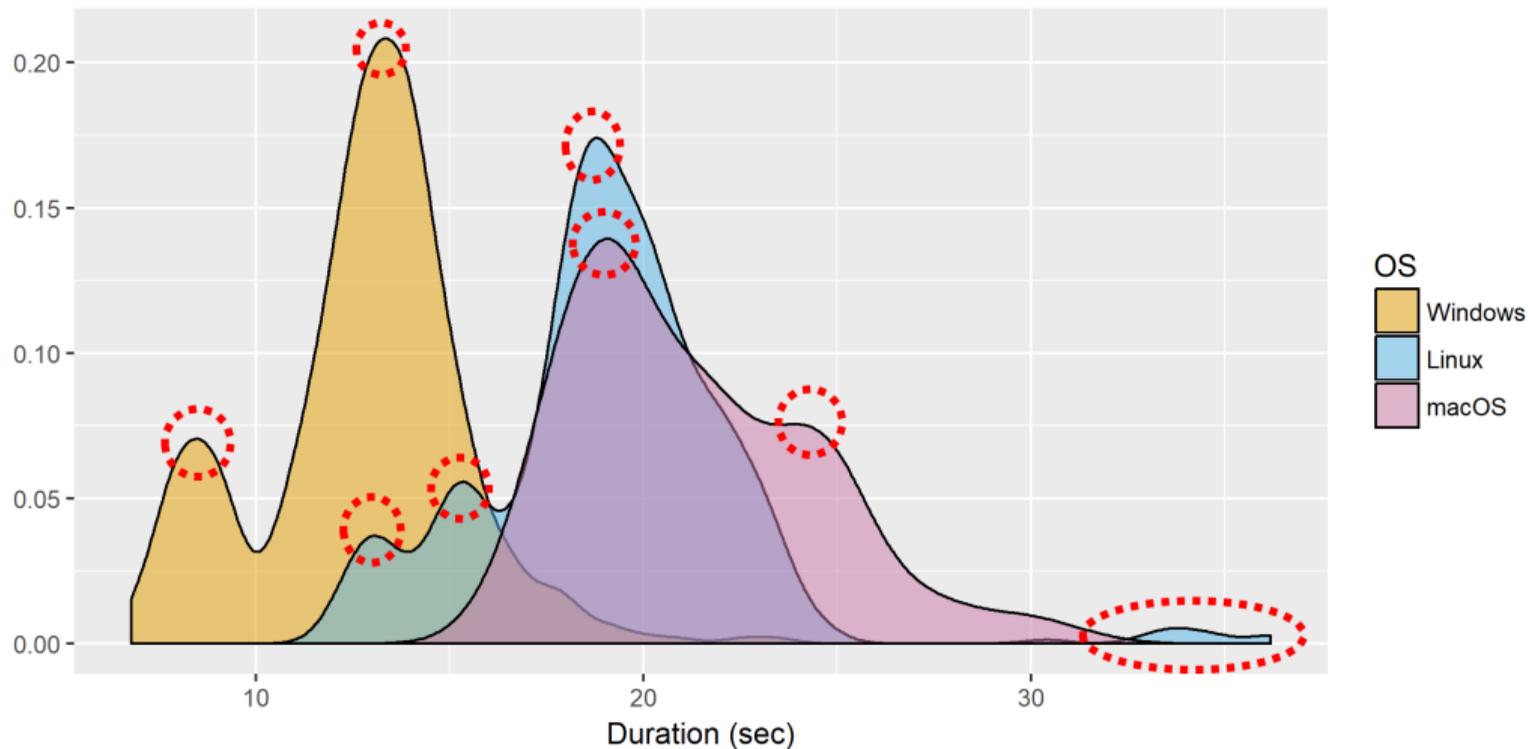


# Маленькие деградации

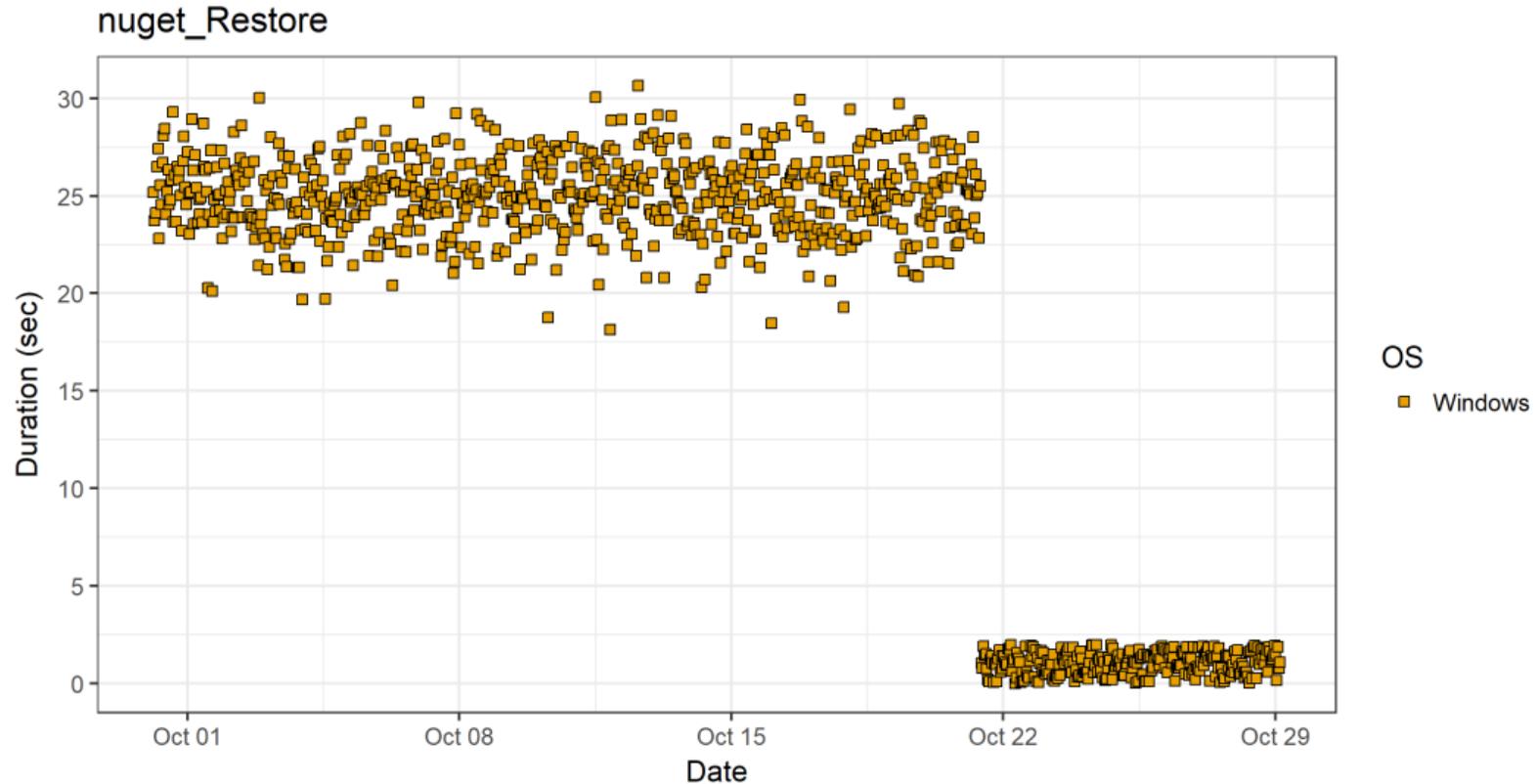
run\_RunConfigurationGeneratorTest\_testChangeOutputTypeInCoreCsProj



run\_RunConfigurationGeneratorTest\_testChangeOutputTypeInCoreCsProj



# Внезапные улучшения



# Заметка 8

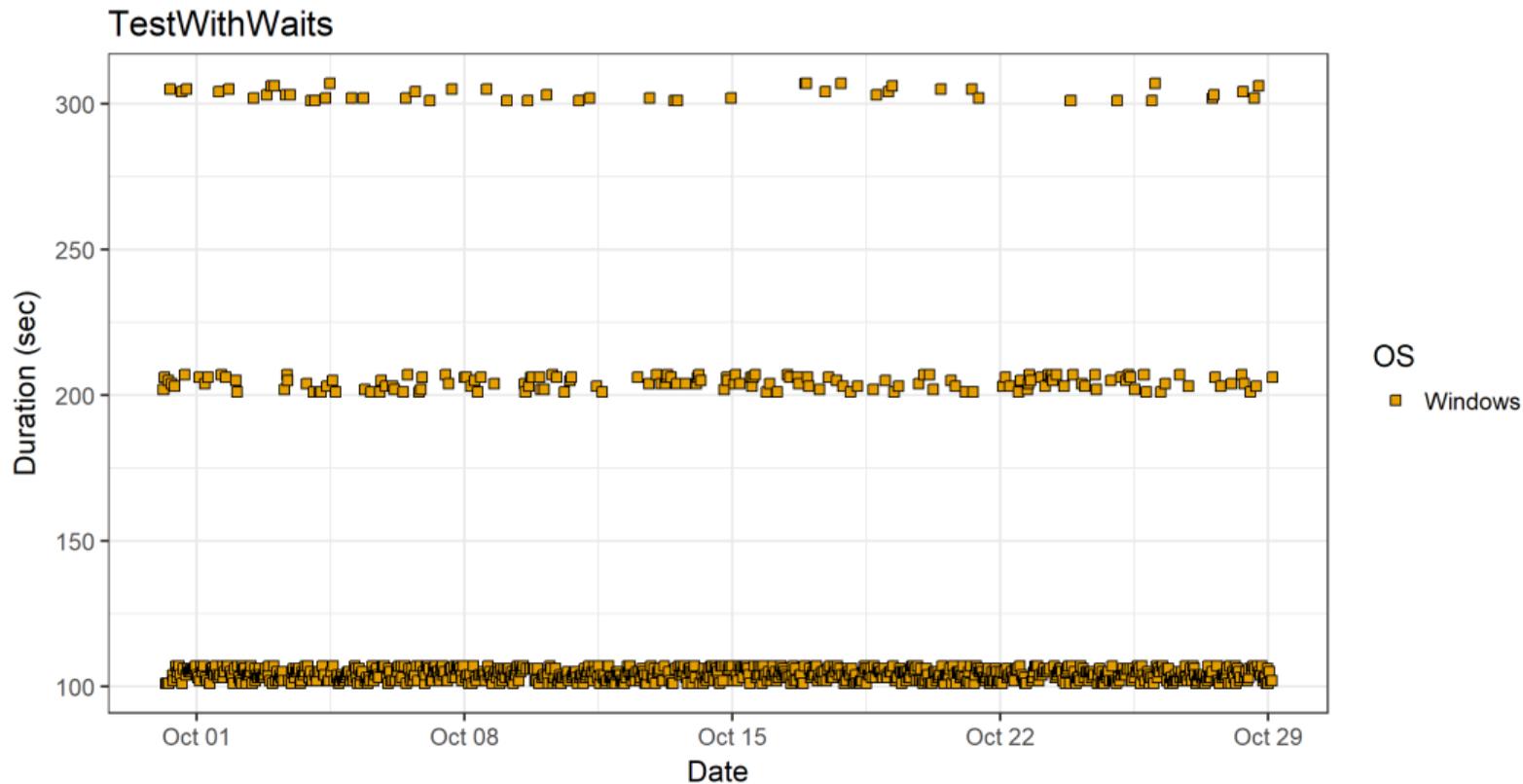
## Распределения странной формы

```
while (!isReady) {  
    Thread.Sleep(100);  
    // Update state  
}
```

```
while (!isReady) {  
    Thread.Sleep(100);  
    // Update state  
}
```

Thread.Sleep — источник *постоянных* проблем

# Мультимодальное распределение



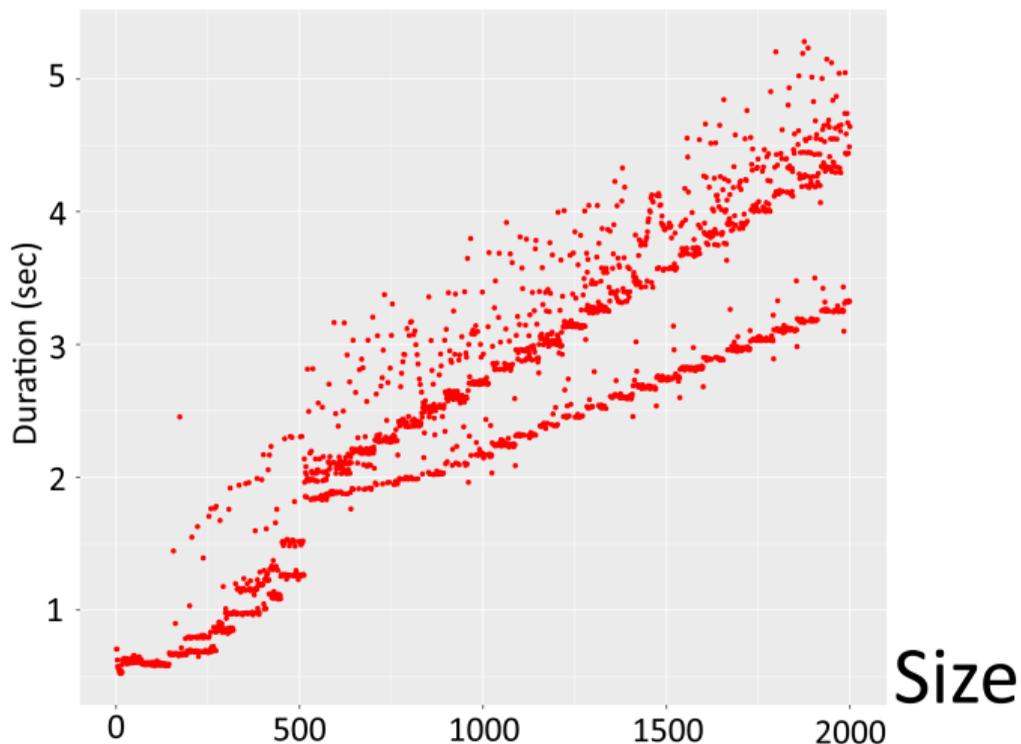
# Бимодальное распределение

OutputLineSplitterTest\_testFlushing



```
// OutputLineSplitterTest_testFlushing
while (!isFinished.get()) {
    mySplitter.process(StringUtil.repeat("A", 100), each);
    i++;
    if (i % 10 == 0) {
        written.release();
        try {
            if (!read.tryAcquire(10, TimeUnit.SECONDS))
                throw new TimeoutException();
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

# Просто странное распределение



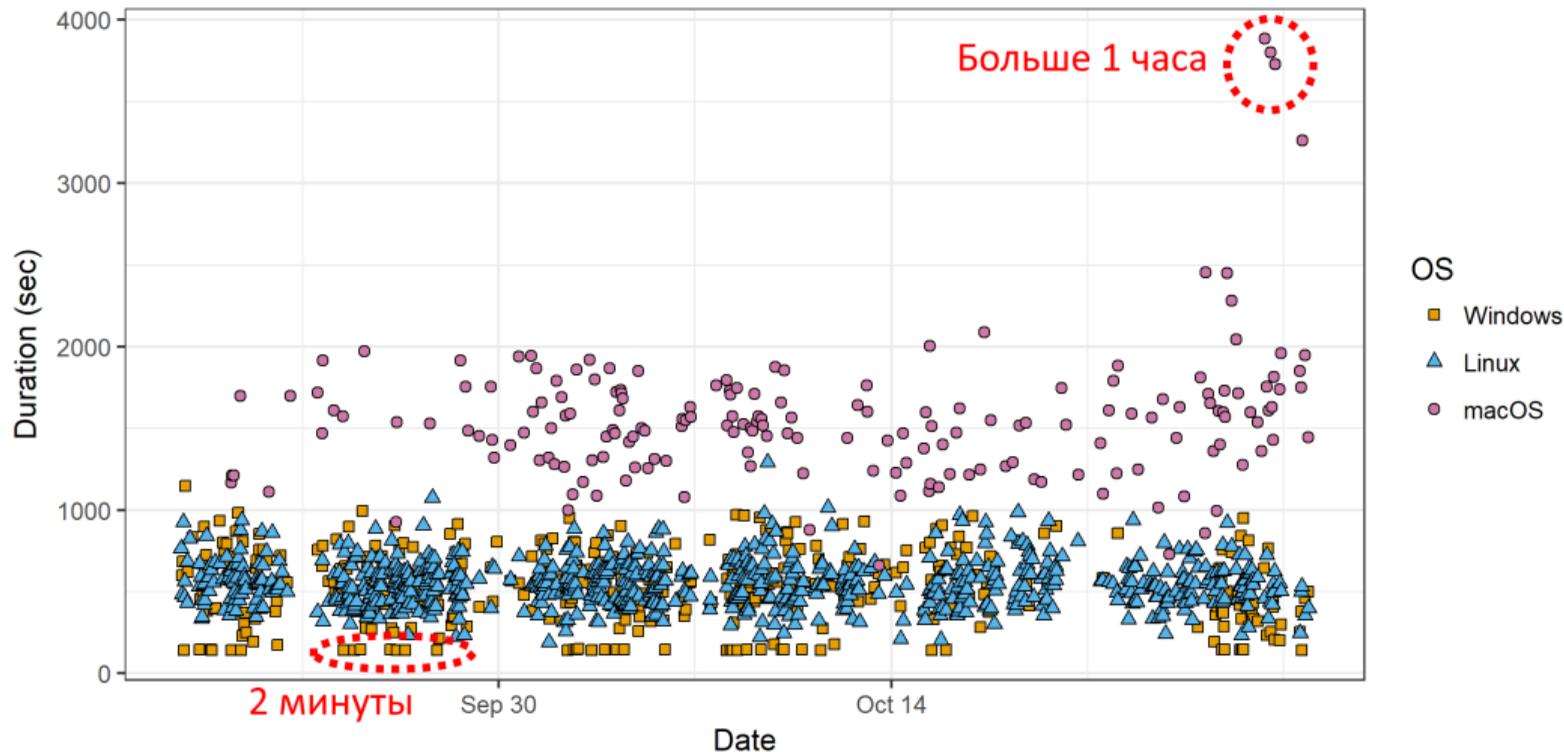
*По мотивам экспериментов с .NET Core*

# Заметка 9

## Большая дисперсия

# Очень большая дисперсия

concurrency\_JobUtilTest\_testProcessorReturningFalseDoesNotCrashTheOtherThread





```
[Test]
public void TestFullTrigrams()
{
    const int start = 256;
    const int limit = 512;
    for (var a = start; a <= limit; ++a)
        for (var b = start; b <= limit; ++b)
            for (var c = start; c <= limit; ++c)
                TestTrigramToken((char)a, (char)b, (char)c);
}
```

```
private static void TestTrigramToken(char a, char b, char c)
{
    var testedTrigram = new TrigramToken(a, b, c);
    var testedTrigramHash = (int)testedTrigram;
    var expectedIsShort = ((a | b | c) >> 8 == 0);

    Assert.AreEqual(expectedIsShort, testedTrigram.IsShort());

    Assert.AreEqual(c,
        TrigramToken.ExtractLastCharacter(testedTrigramHash));
}
```

Unit tests - 17-Oct-17, 6:42:05 PM – JetBrains dotTrace Performance Viewer

File Edit View Help



All Calls Overview



- 88.16 % ThreadStart • 3,349,815 ms • 15 calls • System.Threading.ThreadHelper.ThreadStart
  - 58.63 % ThreadPoolProc • 2,227,683 ms • 10 calls • JetBrains.Application.Threading.Tasks.Scheduler.JetSchedulerThread.ThreadPoolProc
  - 5.89 % ManagerThreadProc • 223,748 ms • 1 call • JetBrains.Application.Threading.Tasks.Scheduler.JetScheduler.ManagerThreadProc
  - 5.88 % <CreateDispatcherThread>b\_0 • 223,596 ms • 1 call • JetBrains.Threading.JetDispatcher+<>c\_\_DisplayClass19\_1.<CreateDisp
  - 5.45 % TestFullTrigrams • 206,962 ms • 1 call • JetBrains.ReSharper.Feature.Services.Tests.FeatureServices.Text.TrigramTokenTest.TestFu
  - 5.43 % TestTrigramToken • 206,471 ms • 16,974,593 calls • JetBrains.ReSharper.Feature.Services.Tests.FeatureServices.Text.TrigramTo
    - 4.75 % That • 180,392 ms • 67,898,372 calls • NUnit.Framework.Assert.That(Object, IResolveConstraint, String, params Object[])
    - 0.19 % AreEqual • 7,071 ms • 33,949,186 calls • NUnit.Framework.Assert.AreEqual(Int32, Int32, String, params Object[])
    - 0.14 % EqualConstraint..ctor • 5,472 ms • 33,949,186 calls • NUnit.Framework.Constraints.EqualConstraint..ctor(Object)
    - 0.04 % get\_Empty • 1,684 ms • 33,949,186 calls • NUnit.Framework.Constraints.Tolerance.get\_Empty
    - 0.03 % NUnitEqualityComparer..ctor • 988 ms • 33,949,186 calls • NUnit.Framework.Constraints.NUnitEqualityComparer..ctor
    - 0.02 % Constraint..ctor • 673 ms • 33,949,186 calls • NUnit.Framework.Constraints.Constraint..ctor(Object)
    - 0.18 % AreEqual • 6,658 ms • 33,949,186 calls • NUnit.Framework.Assert.AreEqual(Object, Object, String, params Object[])

```
// NUnit 2.6.4
public static void AreEqual(int expected, int actual)
{
    Assert.That(actual, Is.EqualTo(expected), null, null);
}

public static void AreEqual(object expected, object actual)
{
    Assert.That(actual, Is.EqualTo(expected), null, null);
}

public static EqualConstraint EqualTo(object expected)
{
    return new EqualConstraint(expected);
}
```

## Заметка 10

Когда аномалия есть, а проблемы нету

# Ложные друзья перформанс-инженера



# Ложные друзья перформанс-инженера



- Изменения в тесте или порядке тестов

# Ложные друзья перформанс-инженера



- Изменения в тесте или порядке тестов
- Изменения в агенте или окружении

# Ложные друзья performance-инженера



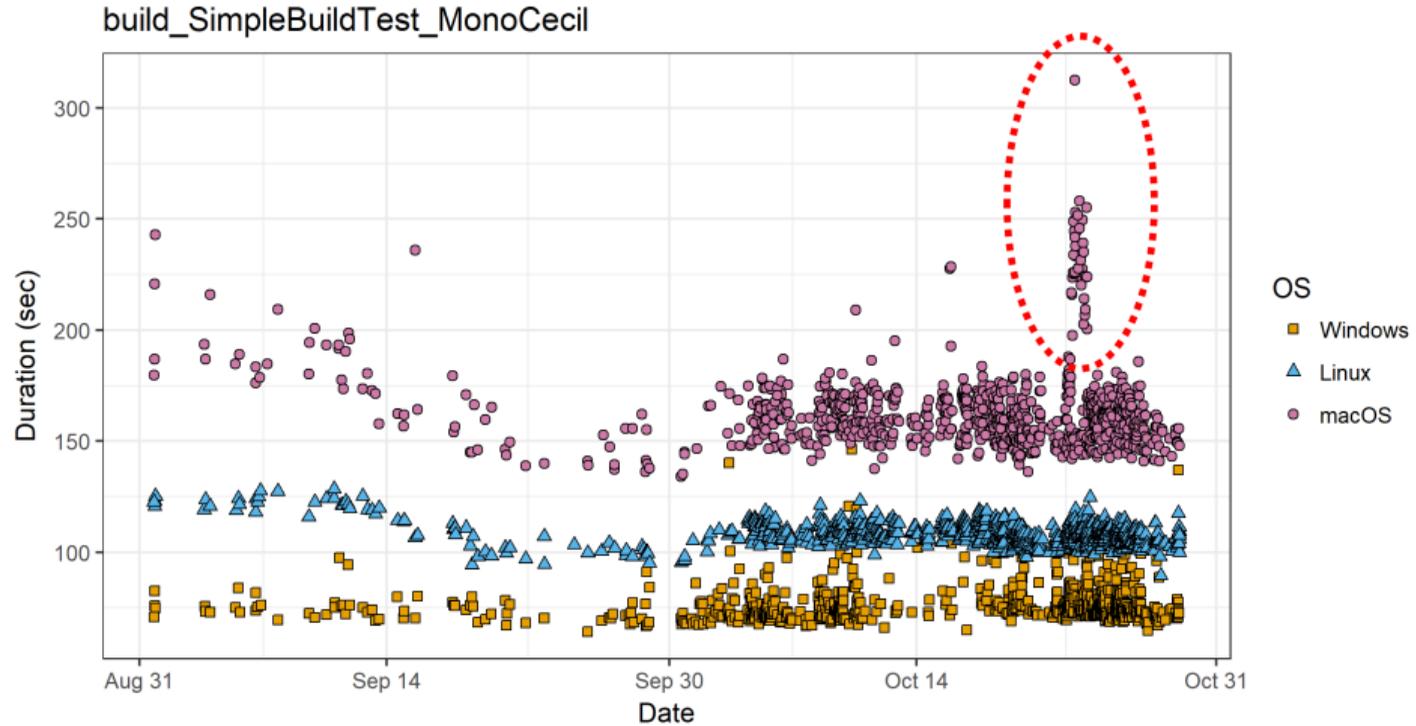
- Изменения в тесте или порядке тестов
- Изменения в агенте или окружении
- Изменения в tradeoff-ах

# Ложные друзья performance-инженера

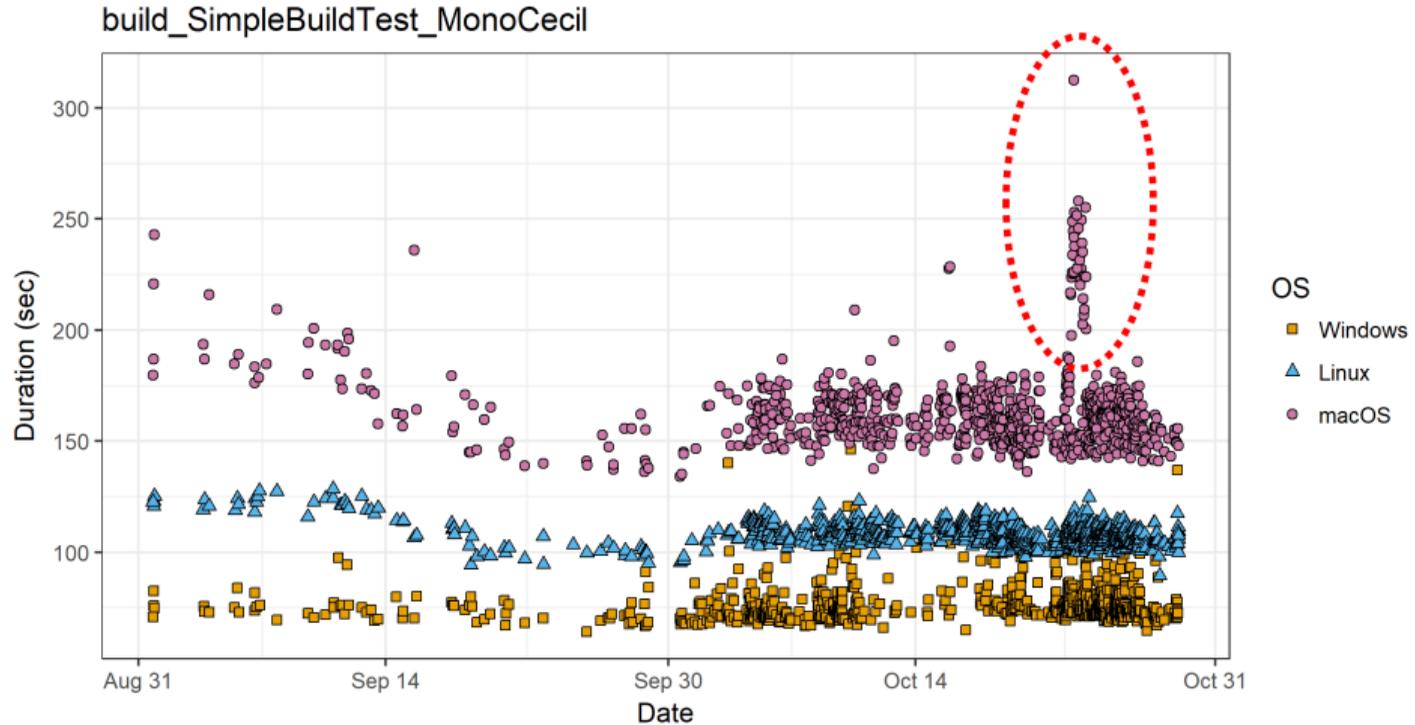


- Изменения в тесте или порядке тестов
- Изменения в агенте или окружении
- Изменения в tradeoff-ах
- Да и вообще, любые изменения

# Найди деградацию!



# Найди деградацию!



*А её нету, просто macOS-агентам было плохо...*

# Заметка 11

## Performance Driven Development (PDD)

# Performance Driven Development (PDD)

# Performance Driven Development (PDD)

- 1 Пишем тест

# Performance Driven Development (PDD)

- 1 Пишем тест
- 2 Собираем performance-метрики и задаём performance-требования

# Performance Driven Development (PDD)

- 1 Пишем тест
- 2 Собираем performance-метрики и задаём performance-требования
- 3 Проверяем, что тест **красный**

# Performance Driven Development (PDD)

- 1 Пишем тест
- 2 Собираем performance-метрики и задаём performance-требования
- 3 Проверяем, что тест **красный**
- 4 Делаем рефакторинг и оптимизации

# Performance Driven Development (PDD)

- 1 Пишем тест
- 2 Собираем performance-метрики и задаём performance-требования
- 3 Проверяем, что тест **красный**
- 4 Делаем рефакторинг и оптимизации
- 5 Проверяем, что тест **зелёный**  
(И все остальные тесты тоже)

# Performance Driven Development (PDD)

- 1 Пишем тест
- 2 Собираем performance-метрики и задаём performance-требования
- 3 Проверяем, что тест **красный**
- 4 Делаем рефакторинг и оптимизации
- 5 Проверяем, что тест **зелёный**  
(И все остальные тесты тоже)
- 6 Возвращаемся к шагу 1

# Заметка 12

## Performance Culture



# Заметка 13

## Заключительные мысли



- Performance-тестирование — увлекательно и полезно, но сложно.

- Performance-тестирование — увлекательно и полезно, но сложно.
- Тестировать performance можно и нужно.

- Performance-тестирование — увлекательно и полезно, но сложно.
- Тестировать performance можно и нужно.
- Соблюдайте performance-чистоту!

- Performance-тестирование — увлекательно и полезно, но сложно.
- Тестировать performance можно и нужно.
- Соблюдайте performance-чистоту!
- Будьте performance-культурными!

Андрей Акиншин

<http://aakinshin.net>

<https://github.com/AndreyAkinshin>

[https://twitter.com/andrey\\_akinshin](https://twitter.com/andrey_akinshin)

[andrey.akinshin@gmail.com](mailto:andrey.akinshin@gmail.com)