

Сергей Марьин

sergei_mar@kontur.ru

**Как совершенно случайно
выстрелить себе в ногу
из стандартной библиотеки**

к^{cloud}нтур

System.Random

```
var random = new Random();  
int n = random.Next(0, 100);
```

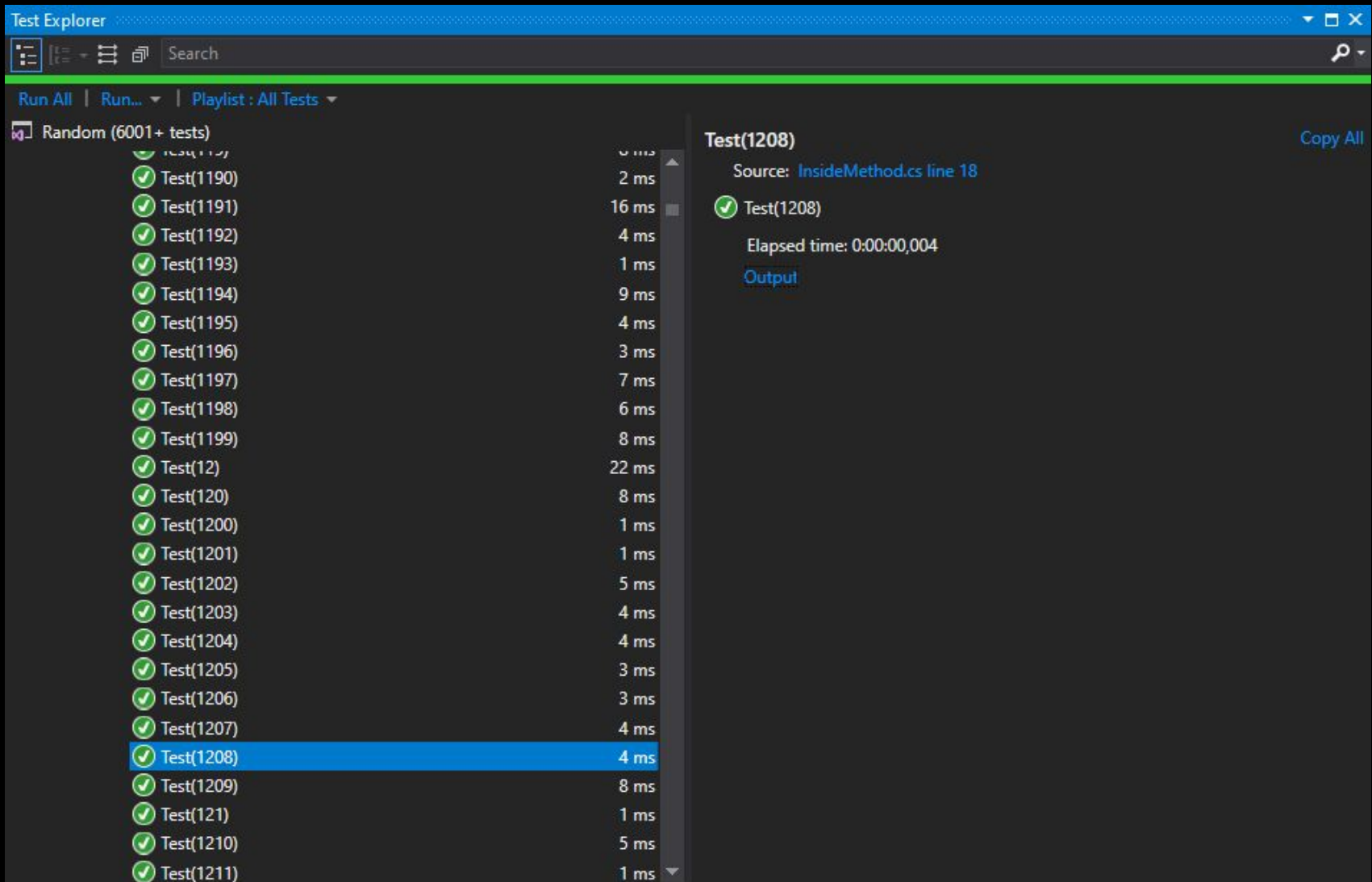
**Пример первый.
Рандомизированные тесты**

Рандомизированные тесты

```
[TestFixture]
public class RandomizedTests
{
    private Random rnd = new Random();

    [TestCaseSource(nameof(TestCases))]
    public void Test(int p)
    {
        for (int i = 0; i < 1000; i++)
        {
            int n = rnd.Next(0, 1_000_000);
            f(p, n).Should().BeGreaterThan(0);
        }
    }
}
```

Рандомизированные тесты



The screenshot shows the Test Explorer window in Visual Studio. The left pane displays a list of tests under the 'Random (6001+ tests)' category. Each test is marked with a green checkmark and its execution time. The test 'Test(1208)' is highlighted in blue. The right pane shows the details for 'Test(1208)', including its source location and elapsed time.

Test Explorer

Run All | Run... | Playlist : All Tests

Random (6001+ tests)

Test Name	Elapsed Time
Test(1190)	2 ms
Test(1191)	16 ms
Test(1192)	4 ms
Test(1193)	1 ms
Test(1194)	9 ms
Test(1195)	4 ms
Test(1196)	3 ms
Test(1197)	7 ms
Test(1198)	6 ms
Test(1199)	8 ms
Test(12)	22 ms
Test(120)	8 ms
Test(1200)	1 ms
Test(1201)	1 ms
Test(1202)	5 ms
Test(1203)	4 ms
Test(1204)	4 ms
Test(1205)	3 ms
Test(1206)	3 ms
Test(1207)	4 ms
Test(1208)	4 ms
Test(1209)	8 ms
Test(121)	1 ms
Test(1210)	5 ms
Test(1211)	1 ms

Test(1208) [Copy All](#)

Source: [InsideMethod.cs line 18](#)

✓ Test(1208)

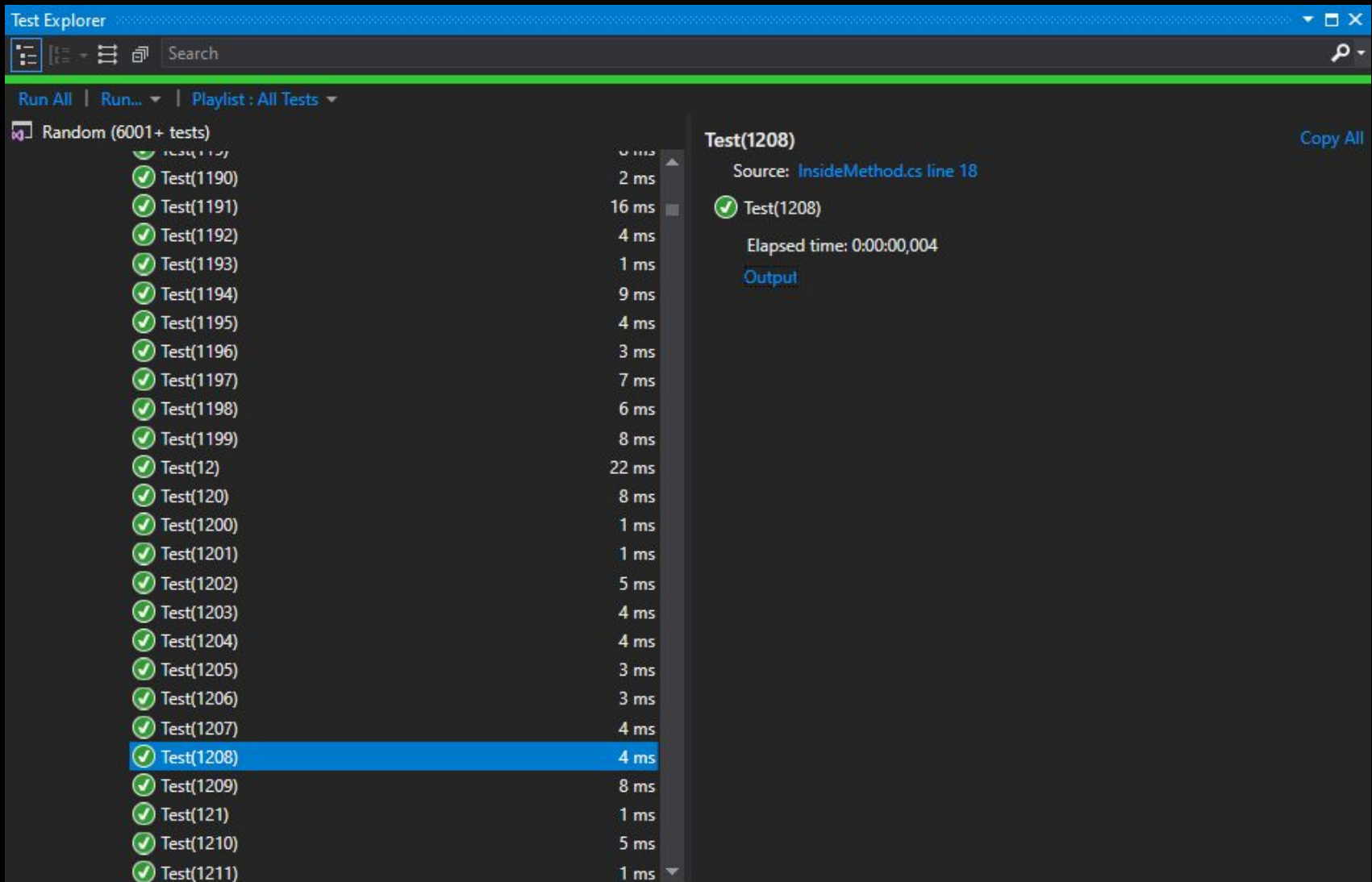
Elapsed time: 0:00:00,004

[Output](#)

Рандомизированные тесты

```
[TestFixture, Parallelizable(ParallelScope.All)] ←  
public class RandomizedTests  
{  
    private Random rnd = new Random();  
  
    [TestCaseSource(nameof(TestCases))]  
    public void Test(int p)  
    {  
        for (int i = 0; i < 1000; i++)  
        {  
            int n = rnd.Next(0, 1_000_000);  
            f(p, n).Should().BeGreaterThan(0);  
        }  
    }  
}
```

Рандомизированные тесты



The screenshot shows the Test Explorer window in Visual Studio. The left pane displays a list of tests under the 'Random (6001+ tests)' group. Each test is marked with a green checkmark and its duration. The right pane shows the details for the selected test, 'Test(1208)', including its source location and elapsed time.

Test Explorer

Run All | Run... | Playlist : All Tests

Random (6001+ tests)

Test Name	Duration
Test(1190)	2 ms
Test(1191)	16 ms
Test(1192)	4 ms
Test(1193)	1 ms
Test(1194)	9 ms
Test(1195)	4 ms
Test(1196)	3 ms
Test(1197)	7 ms
Test(1198)	6 ms
Test(1199)	8 ms
Test(12)	22 ms
Test(120)	8 ms
Test(1200)	1 ms
Test(1201)	1 ms
Test(1202)	5 ms
Test(1203)	4 ms
Test(1204)	4 ms
Test(1205)	3 ms
Test(1206)	3 ms
Test(1207)	4 ms
Test(1208)	4 ms
Test(1209)	8 ms
Test(121)	1 ms
Test(1210)	5 ms
Test(1211)	1 ms

Test(1208) [Copy All](#)

Source: [InsideMethod.cs line 18](#)

✓ Test(1208)

Elapsed time: 0:00:00,004

[Output](#)

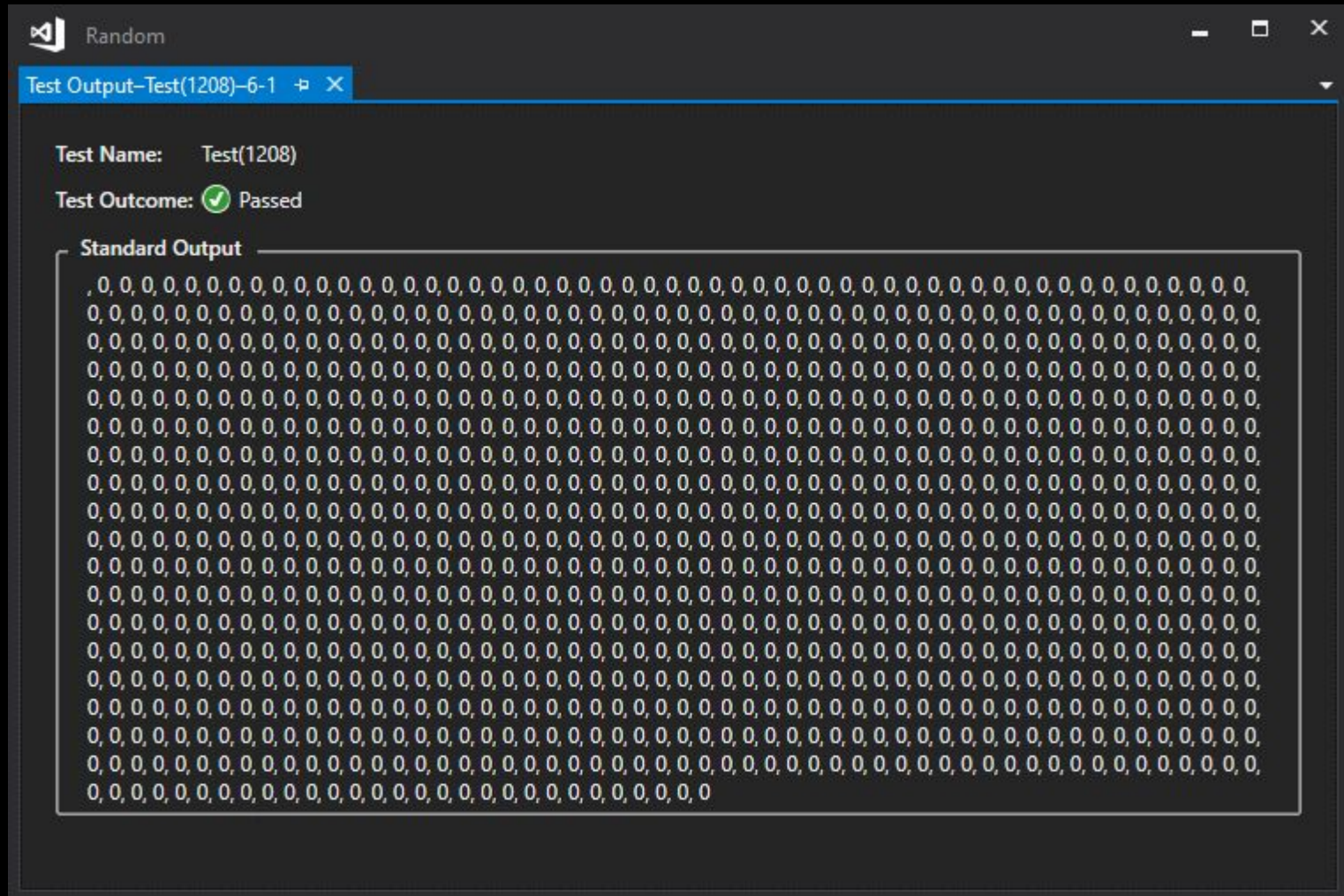
Рандомизированные тесты

```
[TestFixture, Parallelizable(ParallelScope.All)]
public class RandomizedTests
{
    private Random rnd = new Random();

    [TestCaseSource(nameof(TestCases))]
    public void Test(int p)
    {
        for (int i = 0; i < 1000; i++)
        {
            int n = rnd.Next(0, 1_000_000);
            f(p, n).Should().BeGreaterThan(0);

            Console.WriteLine($"", {n}"); ←
        }
    }
}
```


Рандомизированные тесты



Что произошло?

Data race (состояние гонки) в классе Random.

Начиная с какого-то момента — только нули.

Итог: проверяется одно состояние ($n = 0$).

Почему?

Почему?

Сам виноват

MSDN

numeric sequences, we recommend that you create one [Random](#) object to generate many random numbers over time, instead of creating new [Random](#) objects to generate one random number.


However, the [Random](#) class isn't thread safe. If you call [Random](#) methods from multiple threads, follow the guidelines discussed in the next section.

The System.Random class and thread safety

Instead of instantiating individual [Random](#) objects, we recommend that you create a single [Random](#) instance to generate all the random numbers needed by your app. However, [Random](#) objects are not thread safe. If your app calls [Random](#) methods from multiple threads, you must use a synchronization object to ensure that only one thread can access the random number generator at a time. If you don't ensure that the [Random](#) object is accessed in a thread-safe way, calls to methods that return random numbers return 0.

The following example uses the C# [lock Statement](#) and the Visual Basic [SyncLock statement](#) to ensure that a single random number generator is accessed by 11 threads in a thread-safe manner. Each thread generates 2 million random numbers, counts the number of random numbers generated and calculates their sum, and then updates the totals for all threads when it finishes executing.

C#

 Copy

```
using System;
using System.Threading;

public class Example
{
```

Почему?

Сам виноват

Почему?

~~Сам виноват~~

Проблема в библиотеке?

Кто виноват?



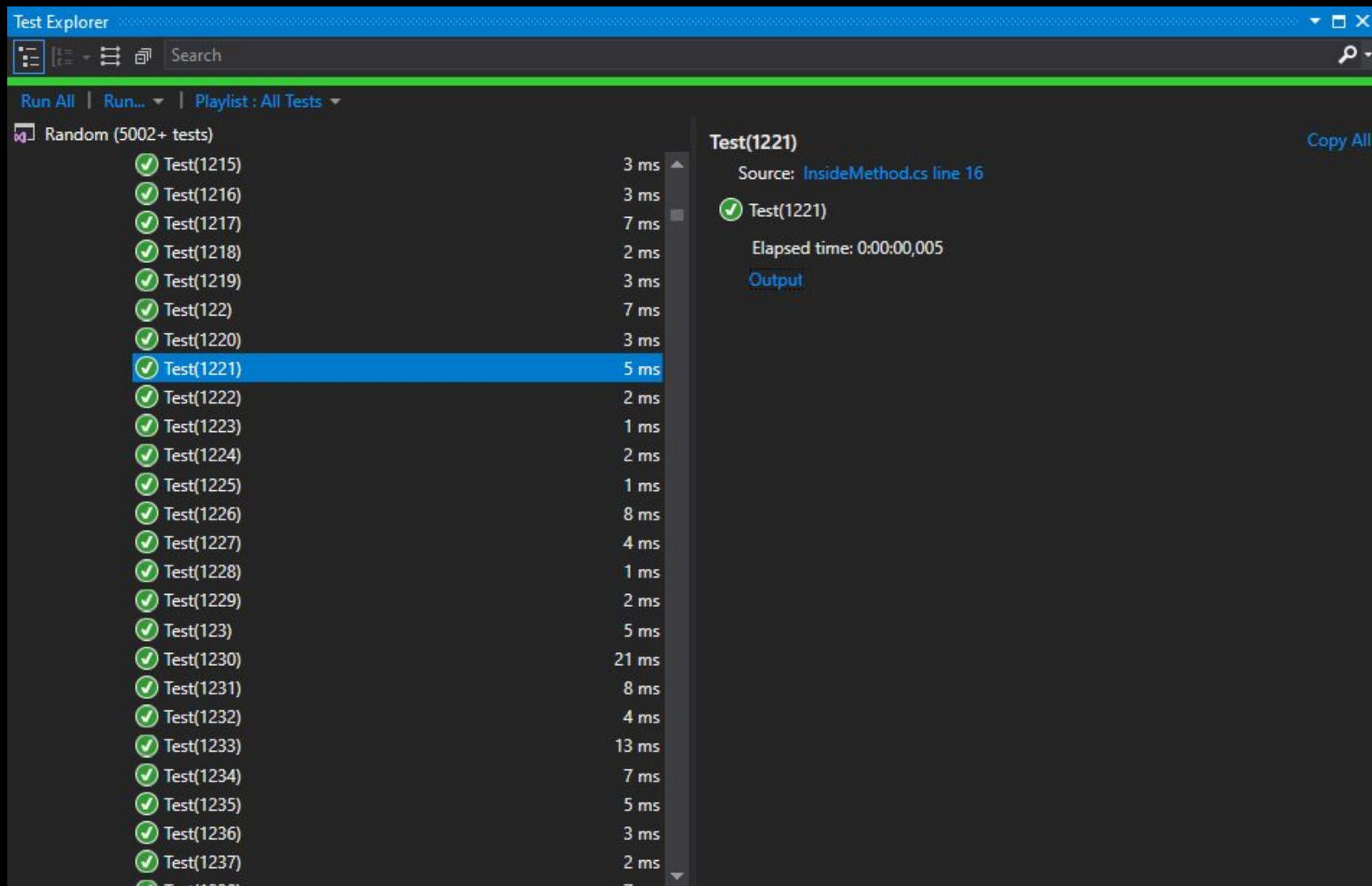
**Пример второй.
Исправленные тесты**

Пытаемся исправить

```
[TestFixture, Parallelizable(ParallelScope.All)]
public class RandomizedTests
{
    [TestCaseSource(nameof(TestCases))]
    public void Test(int p)
    {
        Random rnd = new Random(); ←
        for (int i = 0; i < 1000; i++)
        {
            int n = rnd.Next(0, 1_000_000);
            f(p, n).Should().BeGreaterThan(0);

            Console.WriteLine($" {n}");
        }
    }
}
```

Пытаемся исправить



The screenshot shows the Test Explorer window in Visual Studio. The window has a blue title bar and a toolbar with icons for running tests and a search bar. Below the toolbar, there are buttons for 'Run All', 'Run...', and a dropdown menu for 'Playlist: All Tests'. The main area displays a list of tests under the heading 'Random (5002+ tests)'. Each test entry consists of a green checkmark icon, the test name, and its execution time. The test 'Test(1221)' is highlighted with a blue background. To the right of the list, a detailed view for 'Test(1221)' is shown, including its source location and elapsed time.

Test Name	Elapsed Time
Test(1215)	3 ms
Test(1216)	3 ms
Test(1217)	7 ms
Test(1218)	2 ms
Test(1219)	3 ms
Test(122)	7 ms
Test(1220)	3 ms
Test(1221)	5 ms
Test(1222)	2 ms
Test(1223)	1 ms
Test(1224)	2 ms
Test(1225)	1 ms
Test(1226)	8 ms
Test(1227)	4 ms
Test(1228)	1 ms
Test(1229)	2 ms
Test(123)	5 ms
Test(1230)	21 ms
Test(1231)	8 ms
Test(1232)	4 ms
Test(1233)	13 ms
Test(1234)	7 ms
Test(1235)	5 ms
Test(1236)	3 ms
Test(1237)	2 ms

Test(1221) [Copy All](#)

Source: [InsideMethod.cs line 16](#)

Test(1221)

Elapsed time: 0:00:00,005

[Output](#)

Пытаемся исправить

The image shows two overlapping windows from the Visual Studio Test Explorer. The top window is titled 'Random' and shows the test 'Test(1220)'. The bottom window is titled 'Random - Test Output-Test(1221)-7-1' and shows the test 'Test(1221)'. Both tests have a 'Passed' status, indicated by a green checkmark. The 'Standard Output' pane for each test displays a long list of numbers, which appear to be a sequence of integers. The numbers in the top window's output end with '96129', while the numbers in the bottom window's output end with '415422'.

Test(1220)
Test Outcome: Passed

Standard Output

, 545535, 476821, 121442, 486450, 668274, 757576, 385897, 170205, 736647, 100712, 528412, 222841, 771837, 384459, 729941, 717813, 742336, 378926, 24152, 142746, 949746, 144608, 910524, 576056, 849850, 154657, 510456, 308578, 885096, 476112, 883261, 940952, 196129, 34206, 614907, 25225, 888007, 641403, 795105, 397230, 865197, 404777, 638416, 513244, 603529, 74472, 166547, 450943, 568779, 431298, 771107, 662913, 286360, 408002, 488158, 400927, 566296, 545386, 636600, 513617, 247119, 77318, 285108, 260534, 217450, 587459, 26711, 737630, 769552, 704716, 829806, 100933, 583820, 626921, 277549, 544969, 506191, 397280, 972526, 775377, 988110, 59513, 739799, 453798, 705005, 220347, 654591, 788127, 546048, 213980, 458928, 342620, 4803, 281488, 150110, 787878, 119668, 377881, 295793, 16069, 47761, 428916, 681391, 864062, 601492, 670174, 79093, 659438, 130452, 943188, 894735, 169015, 572859, 861222, 525506, 187606, 337519, 831310, 555529, 997102, 932868, 238583, 191582, 555572, 245787, 487185, 96129, 302331, 476810, 489670, 425301, 128310, 101487, 956457, 727616, 559193, 378122, 875736, 852305, 34830, 141254, 995153, 657674, 602859, 319244, 289912, 769761, 143580, 755982, 962504, 450358, 288357, 822351, 298690, 83201, 809177, 237333, 125818, 618275, 114306, 574044, 776761, 182627, 640782, 517887, 766425, 67528, 616402, 133605, 966312, 809484, 461783, 979004, 520698, 855847, 937714, 580908, 588722, 236327, 955875, 717424, 952549, 546349, 514306, 39311, 136943, 305958, 802796, 873255, 918438, 321860, 252303, 257460, 737999, 460786, 364493, 812525, 16892, 84971, 552819, 222383, 153358, 9974, 789669, 153019, 988575, 309352, 301652, 442842, 145486, 228268, 648611, 889491, 662400, 396882, 621495, 230411, 668320, 601470, 380944, 460466, 264732, 743146, 215167, 644452, 557180, 204322, 241005, 59912, 491354, 125189, 564016, 503751, 683507, 733491, 564065, 942574, 756680, 361286, 50736, 827590, 4305, 359953, 727768, 690169, 673248, 362811, 595060, 341116, 839291, 134082, 144204, 415422

Test(1221)
Test Outcome: Passed

Standard Output

, 545535, 476821, 121442, 486450, 668274, 757576, 385897, 170205, 736647, 100712, 528412, 222841, 771837, 384459, 729941, 717813, 742336, 378926, 24152, 142746, 949746, 144608, 910524, 576056, 849850, 154657, 510456, 308578, 885096, 476112, 883261, 940952, 196129, 34206, 614907, 25225, 888007, 641403, 795105, 397230, 865197, 404777, 638416, 513244, 603529, 74472, 166547, 450943, 568779, 431298, 771107, 662913, 286360, 408002, 488158, 400927, 566296, 545386, 636600, 513617, 247119, 77318, 285108, 260534, 217450, 587459, 26711, 737630, 769552, 704716, 829806, 100933, 583820, 626921, 277549, 544969, 506191, 397280, 972526, 775377, 988110, 59513, 739799, 453798, 705005, 220347, 654591, 788127, 546048, 213980, 458928, 342620, 4803, 281488, 150110, 787878, 119668, 377881, 295793, 16069, 47761, 428916, 681391, 864062, 601492, 670174, 79093, 659438, 130452, 943188, 894735, 169015, 572859, 861222, 525506, 187606, 337519, 831310, 555529, 997102, 932868, 238583, 191582, 555572, 245787, 487185, 96129, 302331, 476810, 489670, 425301, 128310, 101487, 956457, 727616, 559193, 378122, 875736, 852305, 34830, 141254, 995153, 657674, 602859, 319244, 289912, 769761, 143580, 755982, 962504, 450358, 288357, 822351, 298690, 83201, 809177, 237333, 125818, 618275, 114306, 574044, 776761, 182627, 640782, 517887, 766425, 67528, 616402, 133605, 966312, 809484, 461783, 979004, 520698, 855847, 937714, 580908, 588722, 236327, 955875, 717424, 952549, 546349, 514306, 39311, 136943, 305958, 802796, 873255, 918438, 321860, 252303, 257460, 737999, 460786, 364493, 812525, 16892, 84971, 552819, 222383, 153358, 9974, 789669, 153019, 988575, 309352, 301652, 442842, 145486, 228268, 648611, 889491, 662400, 396882, 621495, 230411, 668320, 601470, 380944, 460466, 264732, 743146, 215167, 644452, 557180, 204322, 241005, 59912, 491354, 125189, 564016, 503751, 683507, 733491, 564065, 942574, 756680, 361286, 50736, 827590, 4305, 359953, 727768, 690169, 673248, 362811, 595060, 341116, 839291, 134082, 144204, 415422

Что произошло?

Инициализация текущим временем.

Дискретность системного таймера.

Итог: повторение многих тест-кейсов.

Remarks

The value of this property is derived from the system timer and is stored as a 32-bit signed integer. Note that, because it is derived from the system timer, the resolution of the [TickCount](#) property is limited to the resolution of the system timer, which is typically in the range of 10 to 16 milliseconds.

Почему?

Почему?

Сам виноват

Почему?

~~Сам виноват~~

Проблема в библиотеке?

Почему?

~~Сам виноват~~

Проблема в библиотеке (“защита от дурака”)

Решение: .NET Core

```
119
120  /*=====GenerateSeed=====
121  **Returns: An integer that can be used as seed values for consecutively
122             creating lots of instances on the same thread within a short period of time.
123  =====*/
124  private static int GenerateSeed()
125  {
126      Random? rnd = t_threadRandom;
127      if (rnd == null)
128      {
129          int seed;
130          lock (s_globalRandom)
131          {
132              seed = s_globalRandom.Next();
133          }
134          rnd = new Random(seed);
135          t_threadRandom = rnd;
136      }
137      return rnd.Next();
138  }
139
```

**Пример третий.
Доступ по ссылке**

Доступ по ссылке

Пользователь загружает файлы (картинки).

Шарит по ссылке.

Не хотим давать возможность перебирать по айдишнику.

Доступ по ссылке

Плохо:

example.com/images/10001

example.com/images/10002

example.com/images/10003

Хорошо:

example.com/images/a8fdc205a9f19cc1c750

example.com/images/07a60c4f01b13d11d7fd

example.com/images/da39a3ee5e6b4b0d325

Генератор псевдослучайных чисел

```
x[i+1] = (a*x[i] + b) % m
```

```
a, b, m = const
```

```
x[0] = seed
```

Подбор злоумышленником

1. Следующие/предыдущие значения по формуле
2. Подбор стартового значения (seed) по времени инициализации

<https://habr.com/ru/company/skbkontur/blog/347758/>

<https://github.com/hyprwired/untwister>

Решение: RNGCryptoServiceProvider

Since RNGRandomNumberGenerator only returns byte arrays, you have to do it like this:

27

```
static string RandomString(int length)
{
    const string valid = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";
    StringBuilder res = new StringBuilder();
    using (RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider())
    {
        byte[] uintBuffer = new byte[sizeof(uint)];

        while (length-- > 0)
        {
            rng.GetBytes(uintBuffer);
            uint num = BitConverter.ToUInt32(uintBuffer, 0);
            res.Append(valid[(int)(num % (uint)valid.Length)]);
        }
    }

    return res.ToString();
}
```

Note however that this has a flaw, 62 valid characters is equal to 5,9541963103868752088061235991756 bits ($\log(62) / \log(2)$), so it won't divide evenly on a 32 bit number (uint).

What consequences does this have? As a result, the random output won't be uniform. Characters which are lower in value will occur more likely (just by a small fraction, but still it happens).

Почему?

Сам виноват?

Проблема в библиотеке?

Почему?

~~Сам виноват?~~

~~Проблема в библиотеке?~~

Ошибки проектирования (не так просто исправить)



Эрик Липперт о рандоме в JS и .NET



442



+100

I was one of the implementers of JScript and on the ECMA committee in the mid to late 1990s, so I can provide some historical perspective here.

The JavaScript `Math.random()` function is designed to return a floating point value between 0 and 1. It is widely known (or at least should be) that the output is not cryptographically secure

First off: the design of many RNG APIs is **horrible**. The fact that the .NET Random class can trivially be misused in multiple ways to produce long sequences of the same number is awful. An API where the natural way to use it is also the wrong way is a "pit of failure" API; we want our APIs to be pits of success, where the natural way and the right way are the same.

I think it is fair to say that if we knew then what we know now, the JS random API would be different. Even simple things like changing the name to "pseudorandom" would help, because as you note, in some cases the implementation details matter. At an architectural level, there are good reasons why you want `random()` to be a factory that returns an object representing a random or pseudo-random sequence, rather than simply returning numbers. And so on. Lessons learned.

<https://security.stackexchange.com/a/181623>

**Пример четвёртый.
RNGCryptoServiceProvider всюду**

RNGCryptoServiceProvider всюду

Разработчики библиотеки:

Имплементация `System.Random` через `RNGCryptoServiceProvider` внутри.

Пользователи библиотеки:

Просто везде использовать `RNGCryptoServiceProvider`.

Криптогенераторы внутри

1. Сложнообратимая функция получения следующего элемента (типа хеш-функции).

Проблема: скорость.

2. Стартовое значения (seed) из собранной энтропии.

Проблема: нехватка энтропии.

Решение

Не использовать RNGCryptoServiceProvider (и аналоги) везде подряд :(

Почему?

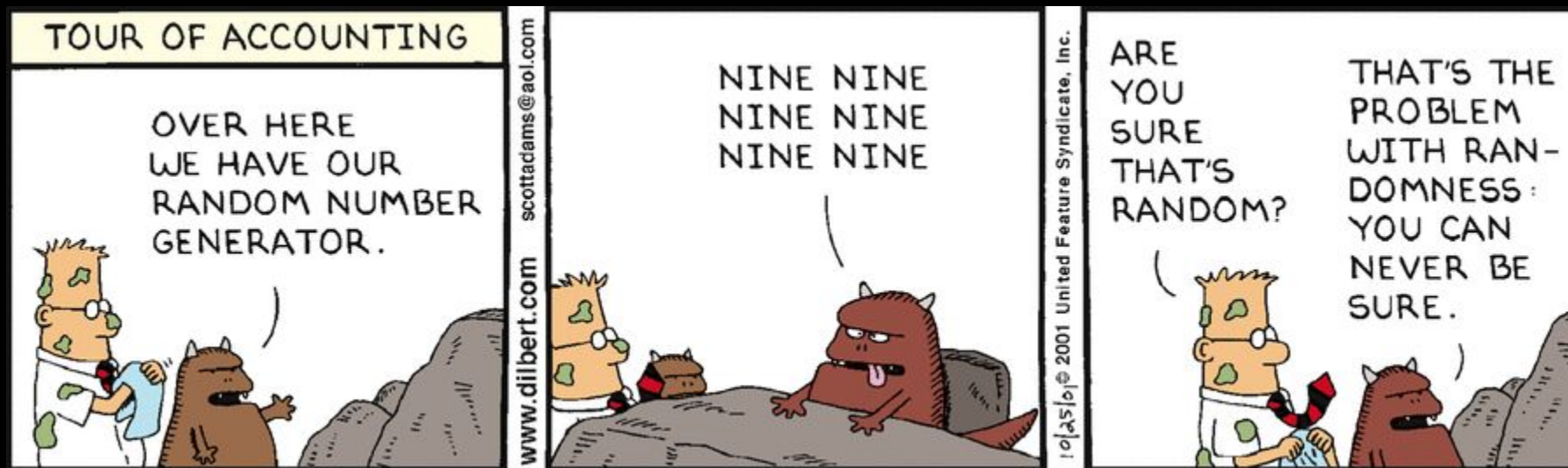
Сам виноват?

Проблема в библиотеке?

Ошибки проектирования?

Почему?

Сложная предметная область



Почему?

Сам виноват

Проблема в библиотеке

Ошибки проектирования

Сложная предметная область

Почему?



Сам виноват

Проблема в библиотеке

Ошибки проектирования

Сложная предметная область

**Пример пятый.
Не только случайность**

Не только случайность

```
[Test]
public void TestConcurrentDictionary()
{
    var dict = new ConcurrentDictionary<string, int>();
    dict["a"] = 0;

    Parallel.For(0, 1000, i =>
    {
        dict["a"] += 1;
    });

    dict["a"].Should().Be(1000);
}
```

Не только случайность

TestConcurrentDictionary

Copy All

Source: [NUnitTests.cs line 30](#)

✖ TestConcurrentDictionary

Message: Expected dict["a"] to be 1000, but found 465.

Elapsed time: 0:00:00,23

▲ StackTrace:

```
LateBoundTestFramework.Throw(String message)
TestFrameworkProvider.Throw(String message)
DefaultAssertionStrategy.HandleFailure(String message)
AssertionScope.FailWith(Func`1 failReasonFunc)
AssertionScope.FailWith(Func`1 failReasonFunc)
AssertionScope.FailWith(String message, Object[] args)
NumericAssertions`1.Be(T expected, String because, Object[] becauseArgs)
NUnitTests.TestConcurrentDictionary()
```

Не только случайность

```
[Test]
public void TestConcurrentDictionary()
{
    var dict = new ConcurrentDictionary<string, int>();
    dict["a"] = 0;

    Parallel.For(0, 1000, i =>
    {
        dict["a"] += 1;
    });

    dict["a"].Should().Be(1000);
}
```

Пример шестой.
left-pad

left-pad

the **npm** blog

Blog about npm things.



kik, left-pad, and npm

Earlier this week, many npm users suffered a disruption when a package that many projects depend on — directly or indirectly — was unpublished by its author, as part of a dispute over a package name. The event generated a lot of attention and raised many concerns, because of the scale of disruption, the circumstances that led to this dispute, and the actions npm, Inc. took in response.

Here's an explanation of what happened.

Timeline

In recent weeks, **Azer Koçulu** and **Kik** exchanged **correspondence** over the use of the module name kik. They weren't able to come to an agreement. Last week, a representative of Kik contacted us to ask for help resolving the disagreement.

left-pad

```
1 module.exports = leftpad;
2 function leftpad (str, len, ch) {
3   str = String(str);
4   var i = -1;
5   if (!ch && ch !== 0) ch = ' ';
6   len = len - str.length;
7   while (++i < len) {
8     str = ch + str;
9   }
10  return str;
11 }
```

left-pad

left-pad / left-pad Archived

Watch

24

★ Star

1.1k

Fork

129

<> Code

🔔 Issues 3

🔗 Pull requests 7

📁 Projects 0

🛡 Security

📊 Insights

$O(n^2)$ time #15

🔒 Closed polkovnikov-ph opened this issue on Mar 24, 2016 · 6 comments



polkovnikov-ph commented on Mar 24, 2016

...

You've opened eyes on the policies of `npm` regarding to the people who are "going to be banging on your door and taking down your accounts and stuff like that". To show my gratitude I'd like to ask you to fix an issue with `O(n^2)` execution time, e.g.

```
module.exports = function(str, len, ch) {  
  return new Array(Math.max(0, len - str.length) + 1).join(ch || ' ') + str;  
};
```



2



1

Assignees

No one assigned

Labels

None yet

Projects

None yet

Milestone

No milestone

7 participants



Giacom commented on Mar 24, 2016

...

Isn't the current implementation the fastest due to VM optimizations?

Что делать?

Универсальный рецепт

Универсальный рецепт

Нету :(

Проблемы

Сам виноват

Проблема в библиотеке

Ошибки проектирования

Сложная предметная область

Сложная предметная область + Сам виноват

Решение:

1. Расширять кругозор
2. Читать исходники библиотек
- ~~3. Читать документацию~~

Проблема в библиотеке

Решение:

1. Регулярно обновляться
2. Читать исходники библиотек

Ошибки проектирования

Проектировать свои библиотеки с оглядкой на
сценарии использования

(“воронка успеха” vs воронка неудач”)

Что делать?

0. Не паниковать
1. Регулярно обновляться
2. Расширять кругозор
3. Читать исходники библиотек
4. Проектировать свои библиотеки с оглядкой на сценарии использования

Что делать?

0. Не паниковать
1. ~~Регулярно обновляться~~ Регулярные физические нагрузки
2. ~~Расширять кругозор~~ Есть побольше овощей
3. ~~Читать исходники библиотек~~ Ложиться спать до полуночи
4. ~~Проектировать свои библиотеки с оглядкой на сценарии использования~~ Звонить маме почаще

**Что делать?
(конкретно)**

Обновления

1. Обновить самый “ненужный” микросервис
2. Новые сервисы/либы на .NET Core/Standart
3. Хобби-проект на .NET Core
4. Запустить на Linux

Кругозор

1. Записаться на следующий митап
2. Почитать ссылки из презентации
3. Открыть книгу по алгоритмам
4. Записаться на [Russian AI Cup](#) или [CodinGame](#)

Russian AI Cup 2019



Читать исходники библиотек

Каких?

1. [System.Random в .NET Core](#)
2. Самый медленный внешний метод в проекте (профилируйте)
3. Куда охота контрибьютить

THIS IS
THE STORE
YOU'VE BEEN
LOOKING FOR



Сценарии использования API

Вспомнить проблемы:

1. Бэклог
2. Слак/телеграм/устно

Что делать?

0. Не паниковать
1. Регулярно обновляться
2. Расширять кругозор
3. Читать исходники библиотек
4. Проектировать свои библиотеки с оглядкой на сценарии использования



sergei_mar@kontur.ru
@sergei_mar