C++ONLINE

KEVLIN HENNEY

KEYNOTE:

SIX IMPOSSIBLE THINGS

2025

6 Impossible Things

Kevlin Henney

@kevlin.bsky.social
@kevlin@mastodon.social
x.com/KevlinHenney
threads.net/@kevlin.henney
instagram.com/kevlin.henney
about.me/kevlin
linkedin.com/in/kevlin
kevlinhenney.medium.com
kevlin@curbralan.com

Kevlin Henney

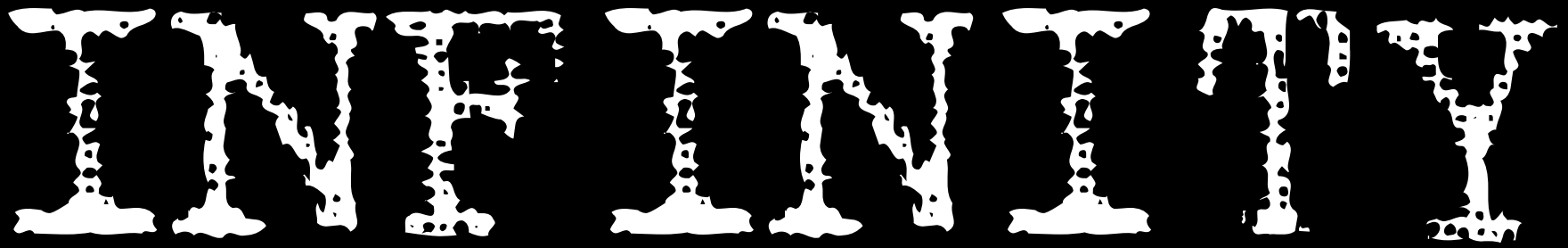
“Sometimes I’ve believed as many as six impossible things before breakfast.”

6 Representations can be infinite

Nothing in nature is truly infinite.

Michael Burnham


GLUTEN FREE
VEGAN
4.2% ABV
LASER JUICE
NEW-WAVE | SESSION IPA
SABRO | CRYO POP
BRISTOL BEER FACTORY
NORTH ST
CIDER
4.8% ALC VOL
MEDIUM DRY | 100% BRITISH
VEGAN | GLUTEN FREE
INFINITY
HELLES LAGER
GLUTEN FREE

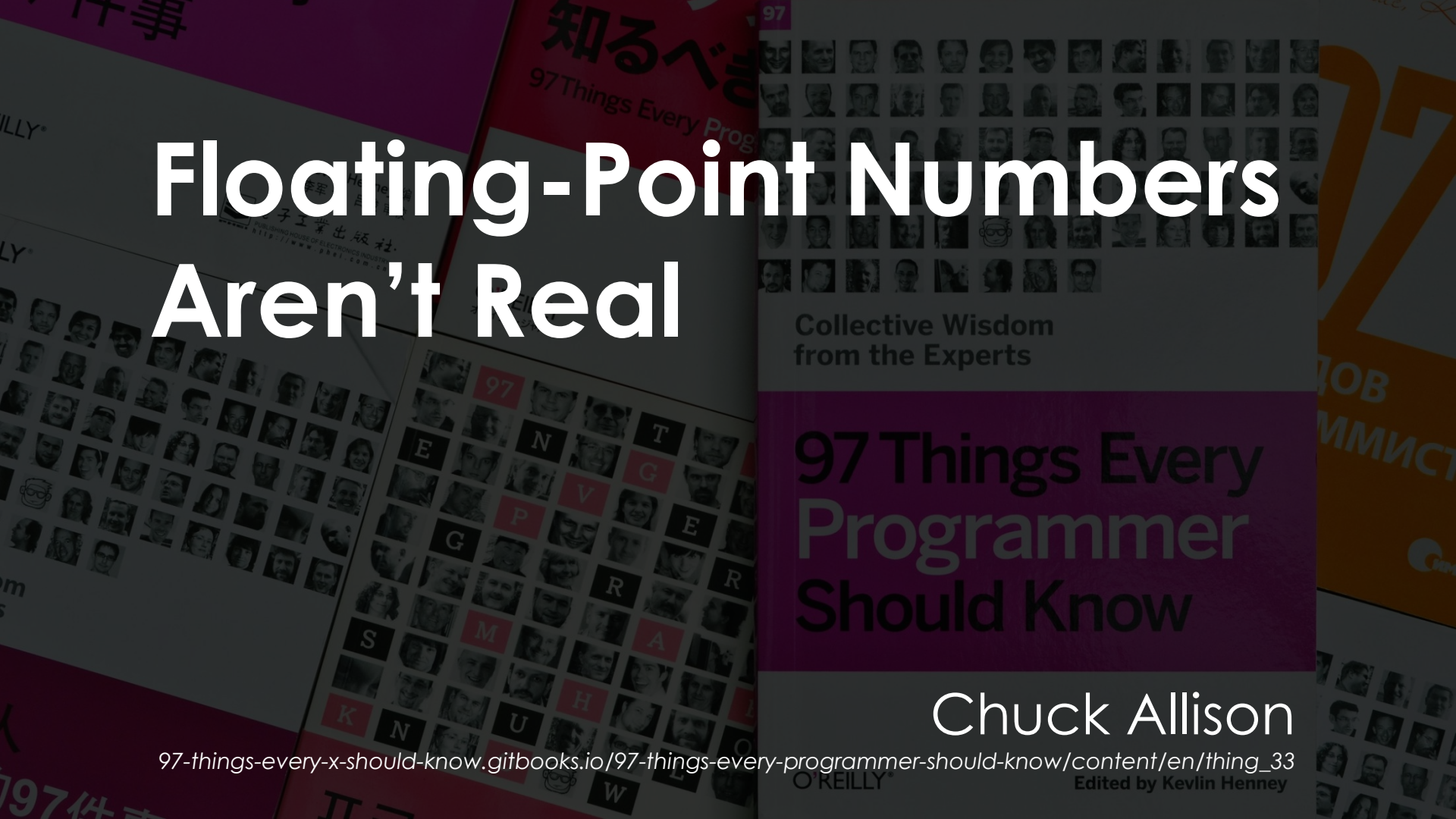

INFINITY

1.0/0.0

Division by zero is undefined for real numbers.

mathworld.wolfram.com/DivisionbyZero.html


Kevlin Henney 编
李军 译 吕骏 审校
電子工業出版社
O'REILLY
オライリー・ジャパン
97
Collective Wisdom
from the Experts
97 Things Every
Programmer
Should Know
O'REILLY
Edited by Kevlin Henney



Chuck Allison

97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_33

Real numbers have infinite precision and are therefore continuous and nonlossy; floating-point numbers have limited precision, so they are finite, and they resemble “badly behaved” integers.

Chuck Allison

97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_33

Richard Dalton
@richardadalton

FizzBuzz was invented to avoid the awkwardness of realising that nobody in the room can binary search an array.

10:29 AM · Apr 24, 2015

```
int BinSearch(int x, const int *a, int n)
{  int middle, left=0, right=n-1;
   if (x <= a[left]) return 0;
   if (x > a[right]) return n;
   while (right - left > 1)
   {  middle = (right + left)/2;
      (x <= a[middle] ? right : left) = middle;
   }
   return right;
}
```

```
int BinSearch(int x, const int *a, int n)
{
    int middle, left=0, right=n-1;
    if (x <= a[left])
        return 0;
    if (x > a[right])
        return n;
    while (right - left > 1)
    {
        middle = (right + left)/2;
        (x <= a[middle] ? right : left) = middle;
    }
    return right;
}
```

```
int BinSearch(int x, const int *a, int n)
{
    int middle, left = 0, right = n - 1;
    if (x <= a[left])
        return 0;
    if (x > a[right])
        return n;
    while (right - left > 1)
    {
        middle = (right + left) / 2;
        (x <= a[middle] ? right : left) = middle;
    }
    return right;
}
```

```
int BinSearch(int x, const int *a, int n)
{
    int middle, left = 0, right = n - 1;
    if (x <= a[left])
        return 0;
    if (x > a[right])
        return n;
    while (right - left > 1)
    {
        middle = (right + left) / 2;
        (x <= a[middle] ? right : left) = middle;
    }
    return right;
}
```

```
int BinSearch(int x, const int *a, int n)
{
    int left = 0, right = n - 1;
    if (x <= a[left])
        return 0;
    if (x > a[right])
        return n;
    while (right - left > 1)
    {
        int middle = (right + left) / 2;
        (x <= a[middle] ? right : left) = middle;
    }
    return right;
}
```

```
L:=1; U:=N
loop
    { MustBe(L,U) }
    if L>U then
        P:=0; break
    M := (L+U) div 2
    case
        X[M] < T:  L:=M+1
        X[M] = T:  P:=M; break
        X[M] > T:  U:=M-1
  endloop
```

Jon Bentley, “Writing Correct Programs”, Communications of the ACM, 26(12), December 1983

programming pearls

By Jon Bentley

WRITING CORRECT PROGRAMS

In the late 1960s people were talking about the promise of programs that verify the correctness of other programs. Unfortunately, it is now the middle of the 1980s, and, with precious few exceptions, there is still little more than talk about automated verification systems. Despite unrealized expectations, however, the research on program verification has given us something far more valuable than a black box that gobbles programs and flashes "good" or "bad"—we now have a fundamental understanding of computer programming.

The purpose of this column is to show how that fundamental understanding can help programmers write correct programs. But before we get to the subject itself, we must keep it in perspective. Coding skill is just one small part of writing correct programs. The majority of the task is the subject of the three previous columns: problem definition, algorithm design, and data structure selection. If you perform those tasks well, then writing correct code is usually easy.

The Challenge of Binary Search

I've given this problem as an in-class assignment in courses at Bell Labs and IBM. The professional programmers had one hour (sometimes more) to convert the above description into a program in the language of their choice; a high-level pseudocode was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take 30 minutes to examine their code, which the programmers did with test cases. In many different classes and with over a hundred programmers, the results varied little: 90 percent of the programmers found bugs in their code (and I wasn't always convinced of the correctness of the code in which no bugs were found).

I found this amazing: only about 10 percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult. In the history in Section 6.2.1 of his *Sorting and Searching*, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.

```
{ MustBe(1,N) }
L := 1; U := N
{ MustBe(L,U) }
loop
    { MustBe(L,U) }
    if L>U then
        { L>U and MustBe(L,U) }
        { T is nowhere in the array }
        P := 0; break
    { MustBe(L,U) and L<=U }
    M := (L+U) div 2
    { MustBe(L,U) and L<=M<=U }
    case
      X[M] < T:
        { MustBe(L,U) and CantBe(1,M) }
        { MustBe(M+1,U) }
        L := M+1
        { MustBe(L,U) }
      X[M] = T:
        { X[M] = T }
        P := M; break
      X[M] > T:
        { MustBe(L,U) and CantBe(M,N) }
        { MustBe(L,M-1) }
        U := M-1
        { MustBe(L, U)}
    { MustBe(L,U) }
  endloop
```

Jon Bentley, “Writing Correct Programs”, Communications of the ACM, 26(12), December 1983

One of the major benefits of program verification is that it gives programmers a language in which they can express that understanding.

These techniques are only a small part of writing correct programs; keeping the code simple is usually the key to correctness.

Jon Bentley, “Writing Correct Programs”, Communications of the ACM, 26(12), December 1983

On the other hand, several professional programmers familiar with these techniques have related to me an experience that is too common in my own programming: when they construct a program, the “hard” parts work the first time, while the bugs are in the “easy” parts.

Jon Bentley, “Writing Correct Programs”, Communications of the ACM, 26(12), December 1983

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```


public.nrao.edu/telescopes/vla

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = low + ((high - low) / 2);
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

```
L:=1; U:=N
loop
    { MustBe(L,U) }
    if L>U then
        P:=0; break
    M := (L+U) div 2
    case
        X[M] < T:  L:=M+1
        X[M] = T:  P:=M; break
        X[M] > T:  U:=M-1
  endloop
```

Jon Bentley, “Writing Correct Programs”, Communications of the ACM, 26(12), December 1983

programming pearls

By Jon Bentley

WRITING CORRECT PROGRAMS

In the late 1960s people were talking about the promise of programs that verify the correctness of other programs. Unfortunately, it is now the middle of the 1980s, and, with precious few exceptions, there is still little more than talk about automated verification systems. Despite unrealized expectations, however, the research on program verification has given us something far more valuable than a black box that gobbles programs and flashes "good" or "bad"—we now have a fundamental understanding of computer programming.

The purpose of this column is to show how that fundamental understanding can help programmers write correct programs. But before we get to the subject itself, we must keep it in perspective. Coding skill is just one small part of writing correct programs. The majority of the task is the subject of the three previous columns: problem definition, algorithm design, and data structure selection. If you perform those tasks well, then writing correct code is usually easy.

The Challenge of Binary Search

I've given this problem as an in-class assignment in courses at Bell Labs and IBM. The professional programmers had one hour (sometimes more) to convert the above description into a program in the language of their choice; a high-level pseudocode was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take 30 minutes to examine their code, which the programmers did with test cases. In many different classes and with over a hundred programmers, the results varied little: 90 percent of the programmers found bugs in their code (and I wasn't always convinced of the correctness of the code in which no bugs were found).

I found this amazing: only about 10 percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult. In the history in Section 6.2.1 of his *Sorting and Searching*, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.

programming pearls

By Jon Bentley

WRITING CORRECT PROGRAMS

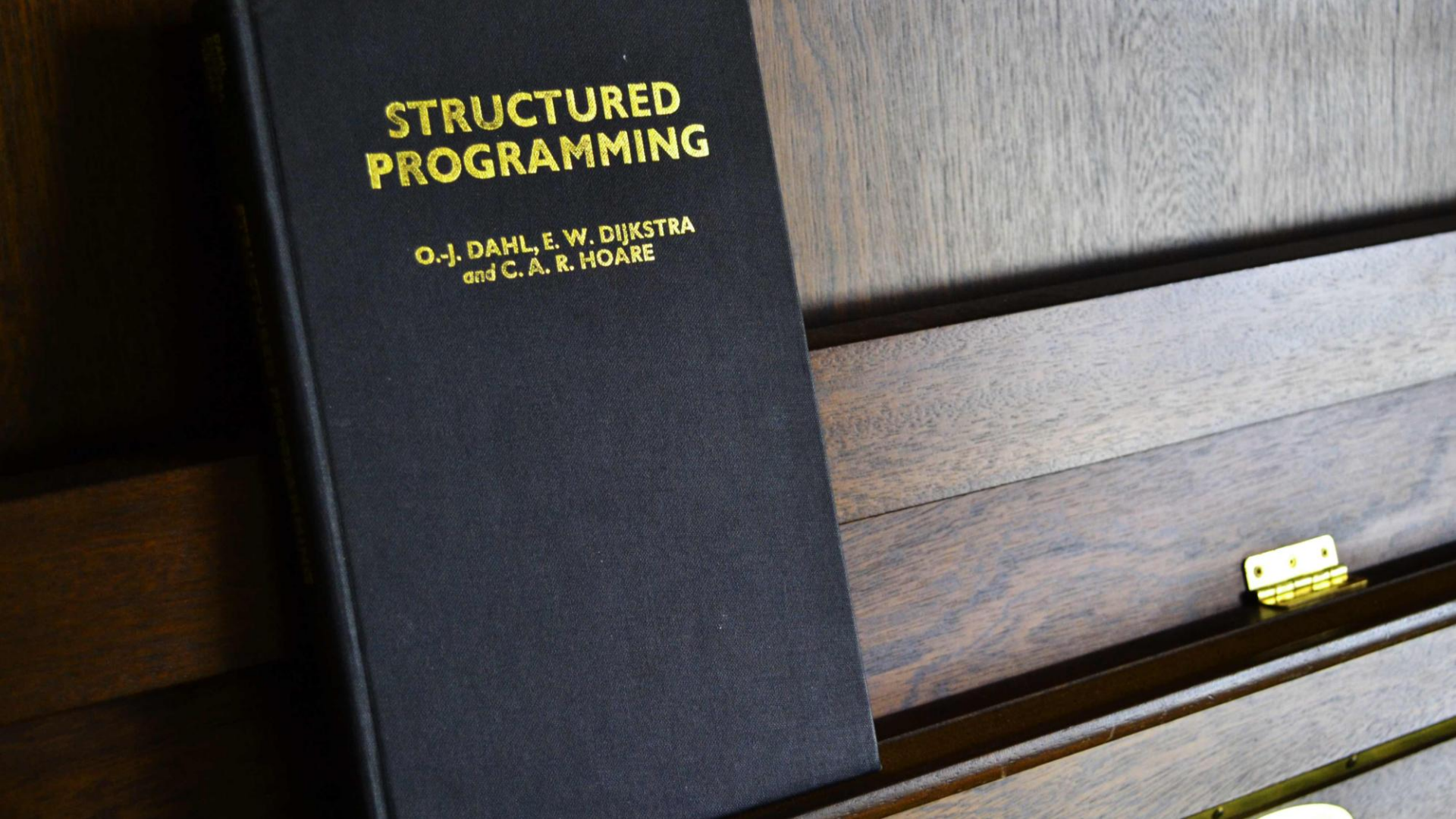
In the late 1960s people were talking about the promise of programs that verify the correctness of other programs. Unfortunately, it is now the middle of the 1980s, and, with precious few exceptions, there is still little more than talk about automated verification systems. Despite unrealized expectations, however, the research on program verification has given us something far more valuable than a black box that gobbles programs and flashes "good" or "bad"—we now have a fundamental understanding of computer programming.

The purpose of this column is to show how that fundamental understanding can help programmers write correct programs. But before we get to the subject itself, we must keep it in perspective. Coding skill is just one small part of writing correct programs. The majority of the task is the subject of the three previous columns: problem definition, algorithm design, and data structure selection. If you perform those tasks well, then writing correct code is usually easy.

The Challenge of Binary Search

I've given this problem as an in-class assignment in courses at Bell Labs and IBM. The professional programmers had one hour (sometimes more) to convert the above description into a program in the language of their choice; a high-level pseudocode was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take 30 minutes to examine their code, which the programmers did with test cases. In many different classes and with over a hundred programmers, the results varied little: 90 percent of the programmers found bugs in their code (and I wasn't always convinced of the correctness of the code in which no bugs were found).

I found this amazing: only about 10 percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult. In the history in Section 6.2.1 of his *Sorting and Searching*, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.


STRUCTURED
PROGRAMMING
O.-J. DAHL, E. W. DIJKSTRA
and C. A. R. HOARE

I have assumed “perfect arithmetic” and in my experience the validity of such proofs often gets questioned by people who argue that in practice one never has perfect arithmetic at ones disposal: admissible integer values usually have an absolute upper bound, real numbers are only represented to a finite accuracy etc.

Edsger W Dijkstra
“Notes on Structured Programming”

So what is the validity of such proofs?

If one proves the correctness of a program assuming an idealised, perfect world, one should not be amazed if something goes wrong when this ideal program gets executed by an "imperfect" implementation.

Edsger W Dijkstra
"Notes on Structured Programming"

```
int BinSearch(int x, const int *a, int n)
{  int middle, left=0, right=n-1;
   if (x <= a[left]) return 0;
   if (x > a[right]) return n;
   while (right - left > 1)
   {  middle = (right + left)/2;
      (x <= a[middle] ? right : left) = middle;
   }
   return right;
}
```

```
int BinSearch(int x, const int *a, int n)
{  int middle, left=0, right=n-1;
   if (x <= a[left]) return 0;
   if (x > a[right]) return n;
   while (right - left > 1)
   {  middle = (right + left)/2;
      (x <= a[middle] ? right : left) = middle;
   }
   return right;
}
```

```
int BinSearch(int x, const int *a, int n)
{  int middle, left=0, right=n-1;
   if (x <= a[left]) return 0;
   if (x > a[right]) return n;
   while (right - left > 1)
   {  middle = std::midpoint(left, right);
      (x <= a[middle] ? right : left) = middle;
   }
   return right;
}
```

```
int BinSearch(int x, const int *a, int n)
{
    return std::lower_bound(a, a + n, x) - a;
}
```

```
std::lower_bound(a, a + n, x)
```

5

Every question has an answer

Our Reply

31 December 1969

Your feedback will be used to improve Facebook. Thanks for taking the time to make a report.

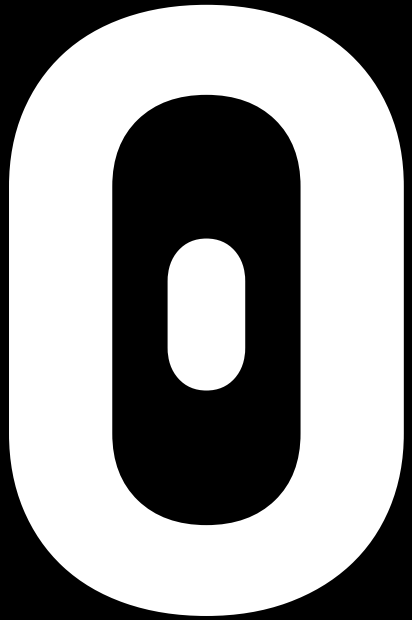
Our Reply

31 December 1969

Your feedback will be used to improve Facebook. Thanks for taking the time to make a report.

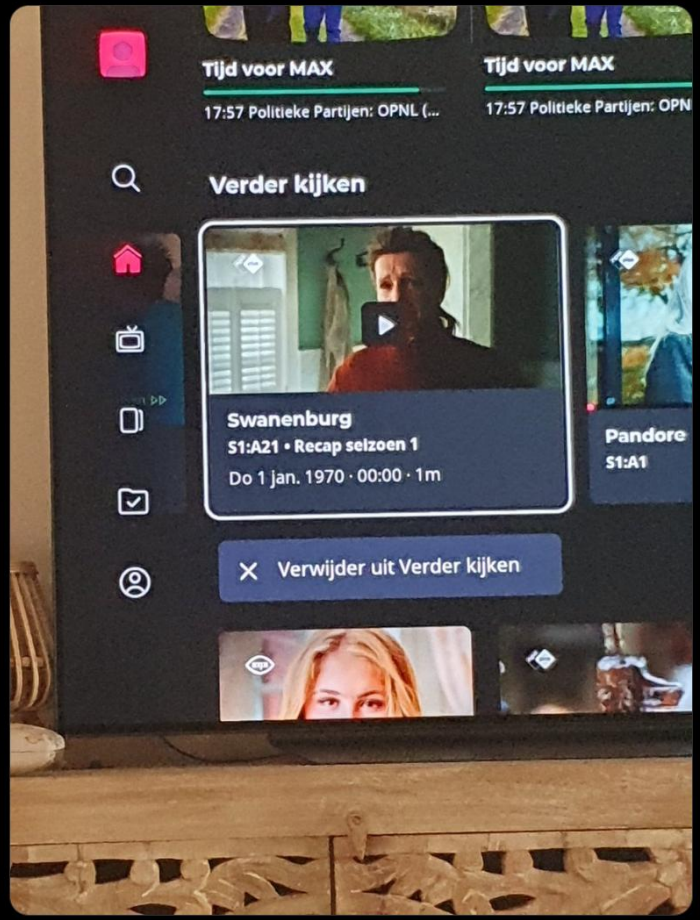
The `time` function shall return the value of time in seconds since the Epoch.

IEEE Std 1003.1-2017

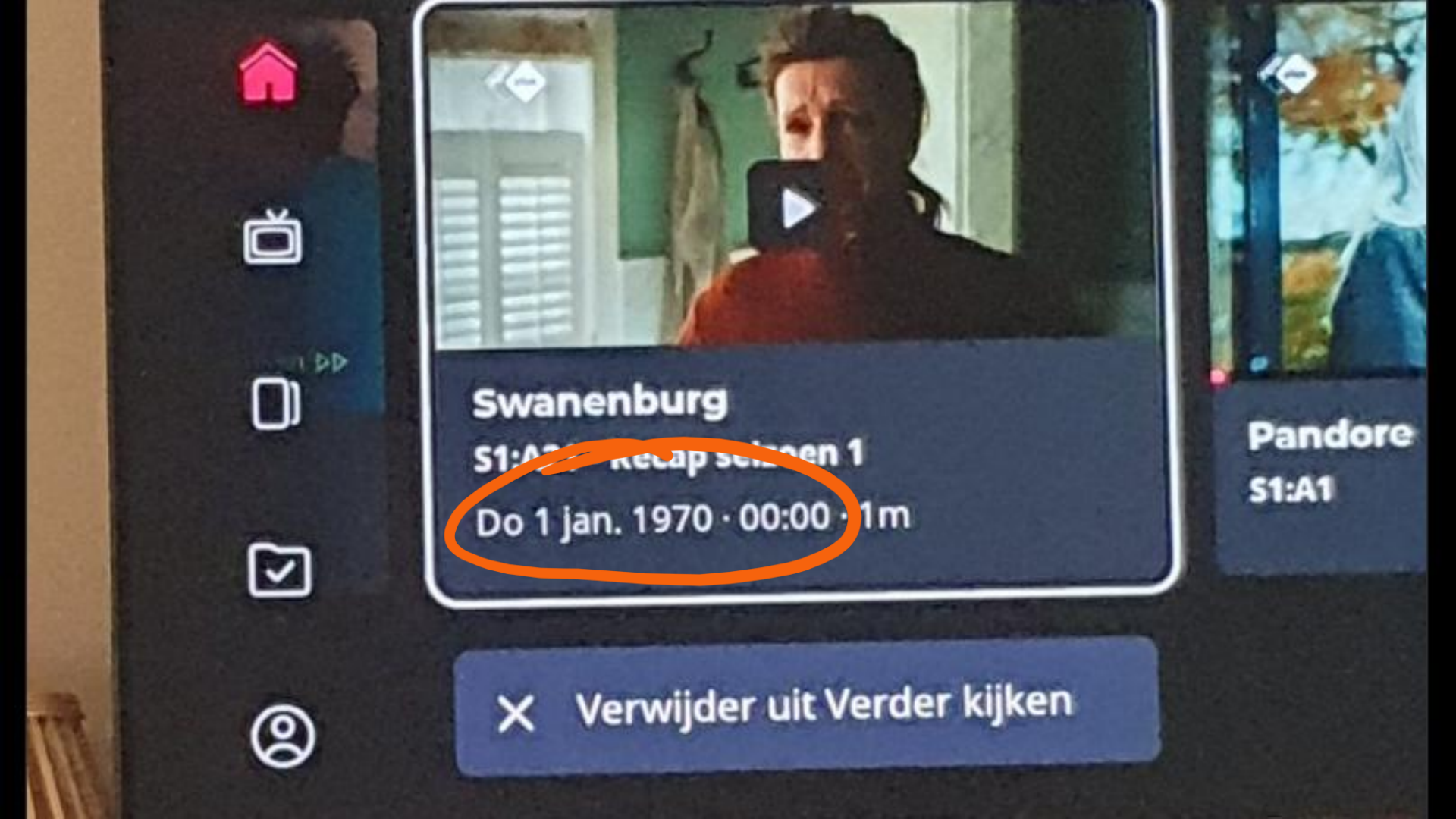


Paul Rijke
@prijke

Quite an old episode @KevlinHenney



4:58 PM · Apr 25, 2024 · **611** Views


Swanenburg
Do 1 jan. 1970 · 00:00 · 1m
Pandore
S1:A1
Verwijder uit Verder kijken

Anil Dash
@anildash

The natural enemy of the programmer is the timezone.

5:32 AM · Dec 16, 2019

1.4K 333 Share this Tweet

x.com/anildash/status/1206447093872812033

Our Reply

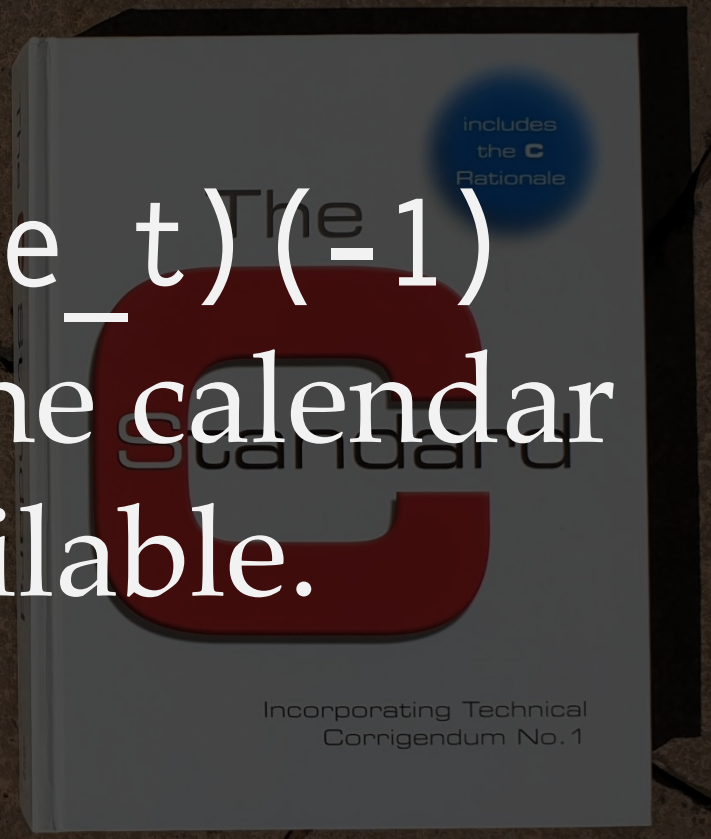
31 December 1969

Your feedback will be used to improve Facebook. Thanks for taking the time to make a report.

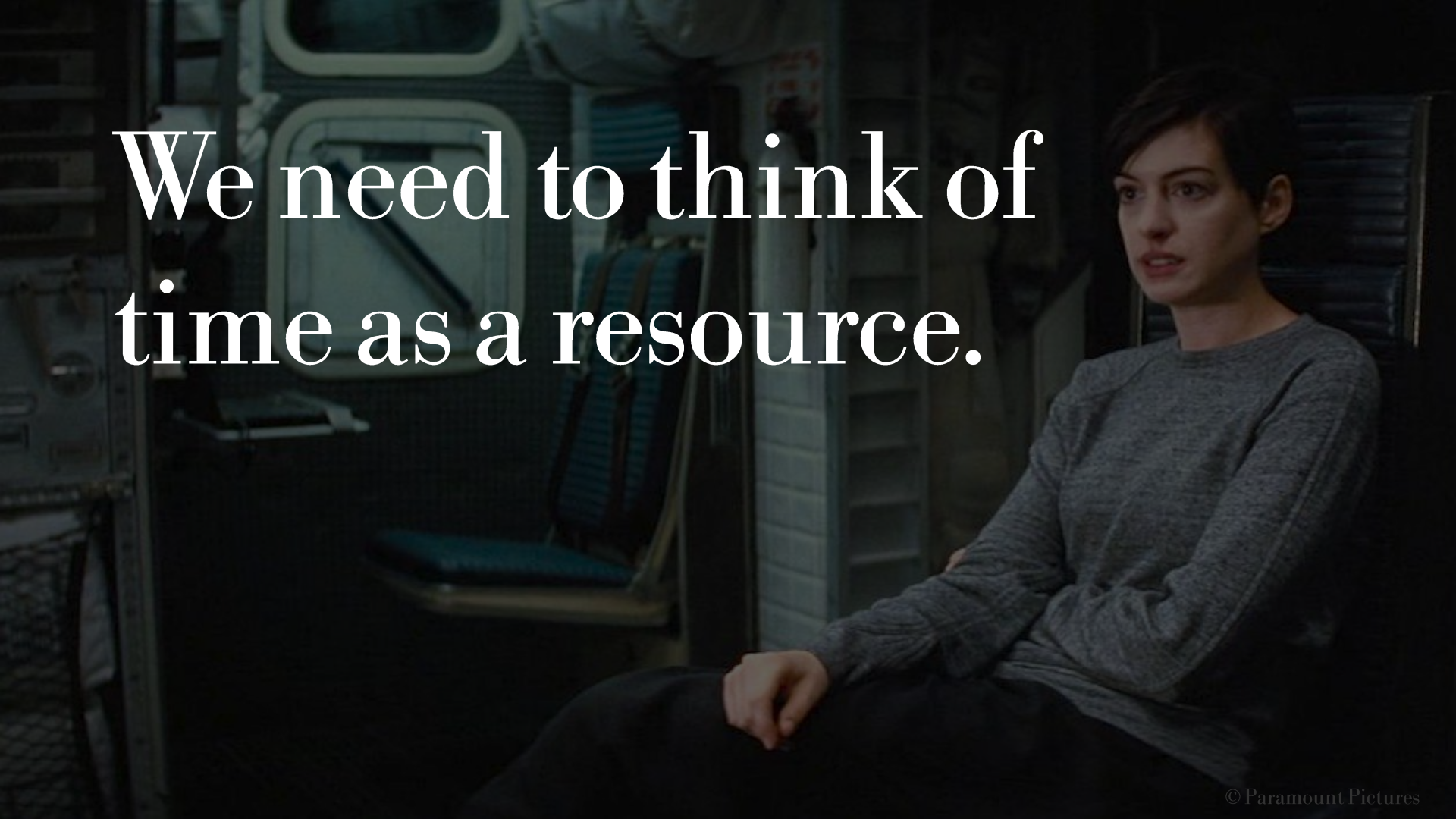
-1


includes
the C
Rationale
The
C
Standard
Incorporating Technical
Corrigendum No. 1

The value `(time_t)(-1)` is returned if the calendar time is not available.




© Paramount Pictures


We need to think of time as a resource.
© Paramount Pictures

```
double arithmetic_mean(auto begin, auto end);
```

```
double arithmetic_mean(auto begin, auto end)
{
    return std::accumulate(begin, end, 0.0) /
           std::distance(begin, end);
}
```

Given...

```
const auto values = {1, 2, 3, 4, 5, 6};
const auto begin  = values.begin();
const auto end    = values.end();
```

When...

```
arithmetic_mean(begin, end)
```

Then...

```
3.5
```

Given...

```
const auto values = {1, 2, 3, 4, 5, 6};
const auto begin  = values.begin();
const auto end    = values.begin();
```

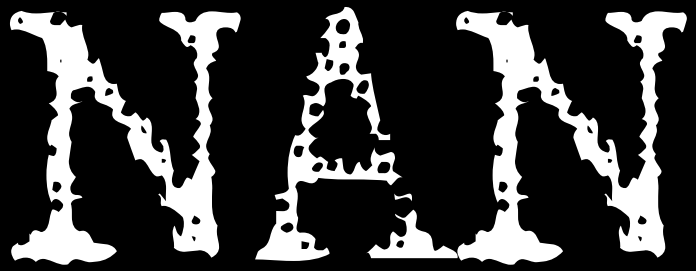
When...

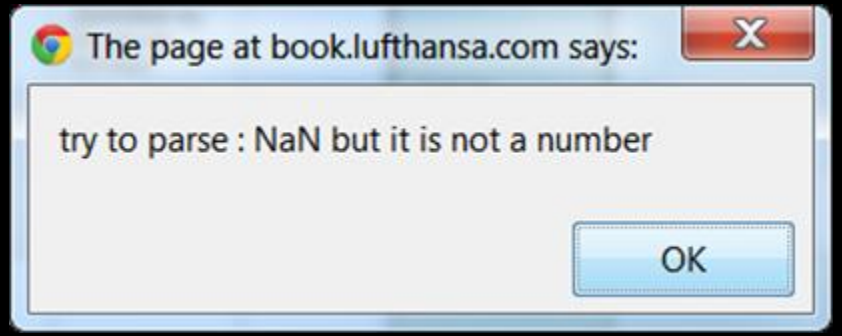
```
arithmetic_mean(begin, end)
```

Then...

```
NaN
```

0.0/0.0


NAN


The page at book.lufthansa.com says:
try to parse : NaN but it is not a number
OK

Given...

```
std::set<double> values;
values.insert(0);
values.insert(42);
values.insert(-273.15);
```

When...

```
values.size()
```

Then...

```
3
```

Given...

```
std::set<double> values;
values.insert(0);
values.insert(42);
values.insert(-273.15);
values.insert(NAN);
```

When...

```
values.size()
```

Then...

```
3
```

Given...

```
std::set<double> values;
values.insert(NAN);
values.insert(0);
values.insert(42);
values.insert(-273.15);
```

When...

```
values.size()
```

Then...

```
1
```

Given...

```
std::set<double> values;
values.insert(NAN);
values.insert(0);
values.insert(42);
values.insert(-273.15);
```

When...

```
values.count(NAN)
```

Then...

```
1
```

Given...

```
std::set<double> values;
values.insert(0);
values.insert(42);
values.insert(-273.15);
values.insert(NAN);
```

When...

```
values.count(NAN)
```

Then...

```
1
```

Given...

```
std::set<double> values;
values.insert(0);
values.insert(42);
values.insert(-273.15);
```

When...

```
values.count(NAN)
```

Then...

```
1
```

Given...

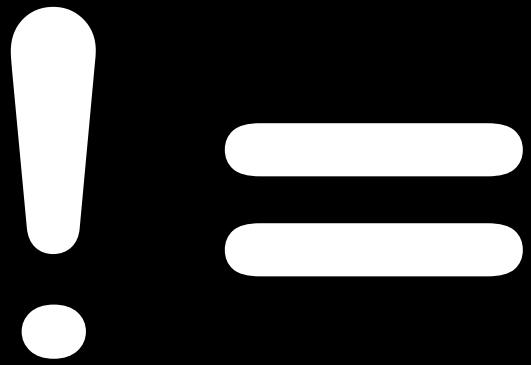
```
std::multiset<double> values;
values.insert(0);
values.insert(42);
values.insert(-273.15);
```

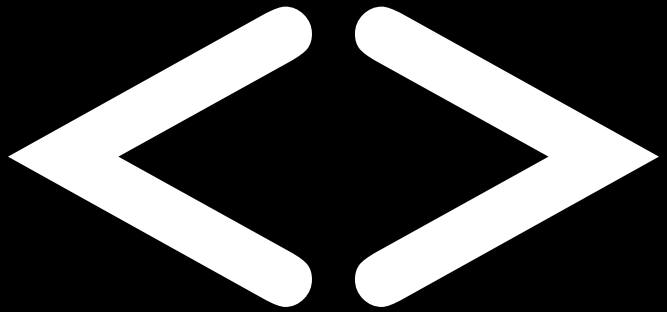
When...

```
values.count(NAN)
```

Then...

```
3
```





Driverless racecar drives straight into a wall

So during this initialization lap something happened which apparently caused the steering control signal to go to NaN and subsequently the steering locked to the maximum value to the right.

reddit.com/r/formula1/comments/jk9jrg/ot_roborace_driverless_racecar_drives_straight

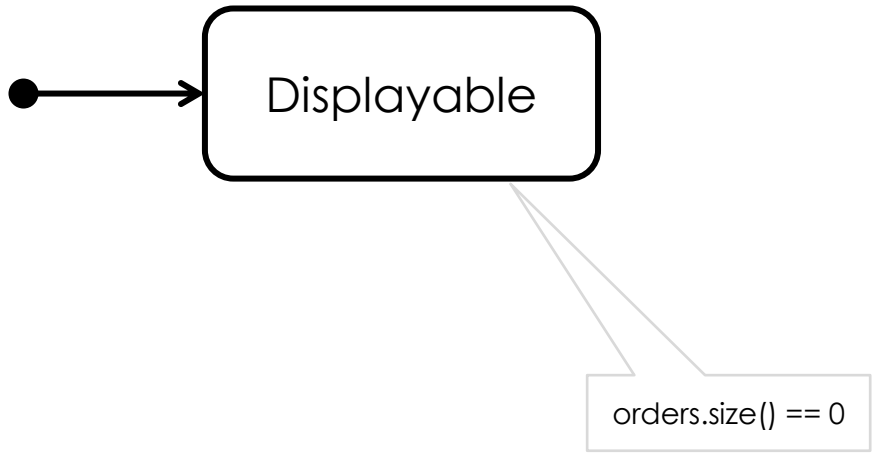

11:11
AACHEN
Delayed
Calling at:
AACHEN only.
Page 1 of 1
10:28 Platform 14
Greenford
Calling at:
Acton Main Line
Ealing Broadway
West Ealing
Drayton Green
Castle Bar Park
South Greenford
& Greenford
NOT A TRAIN
First Great Western

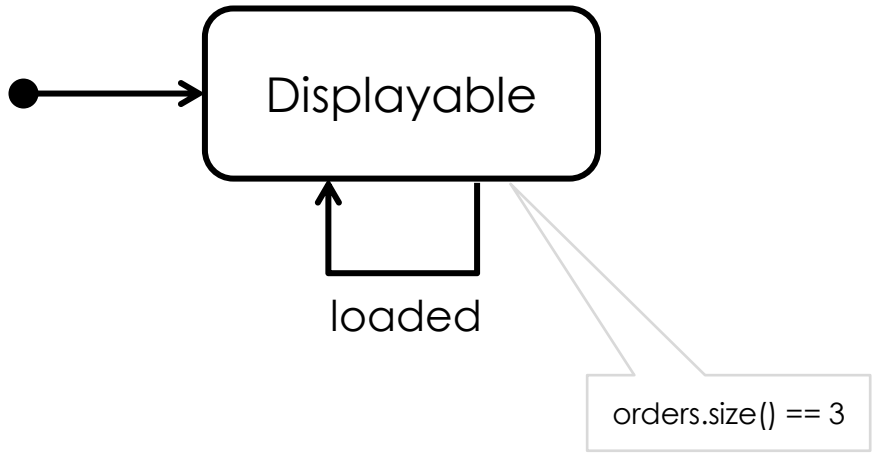
You have 0 orders

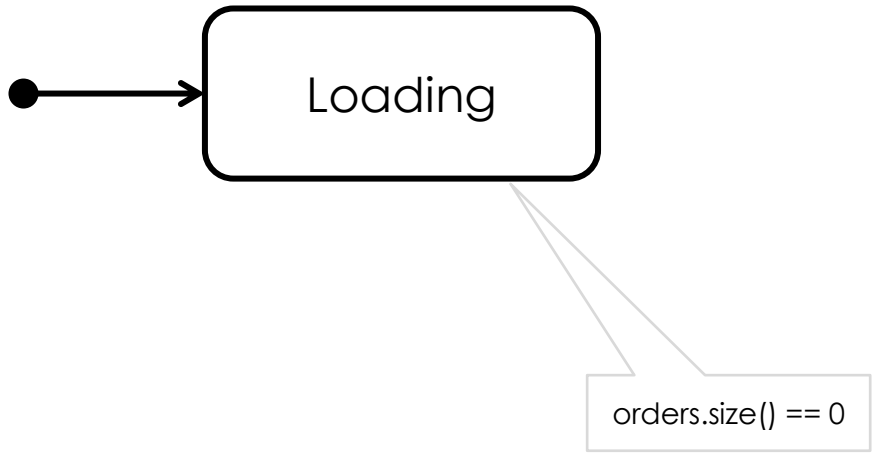
You have 3 orders

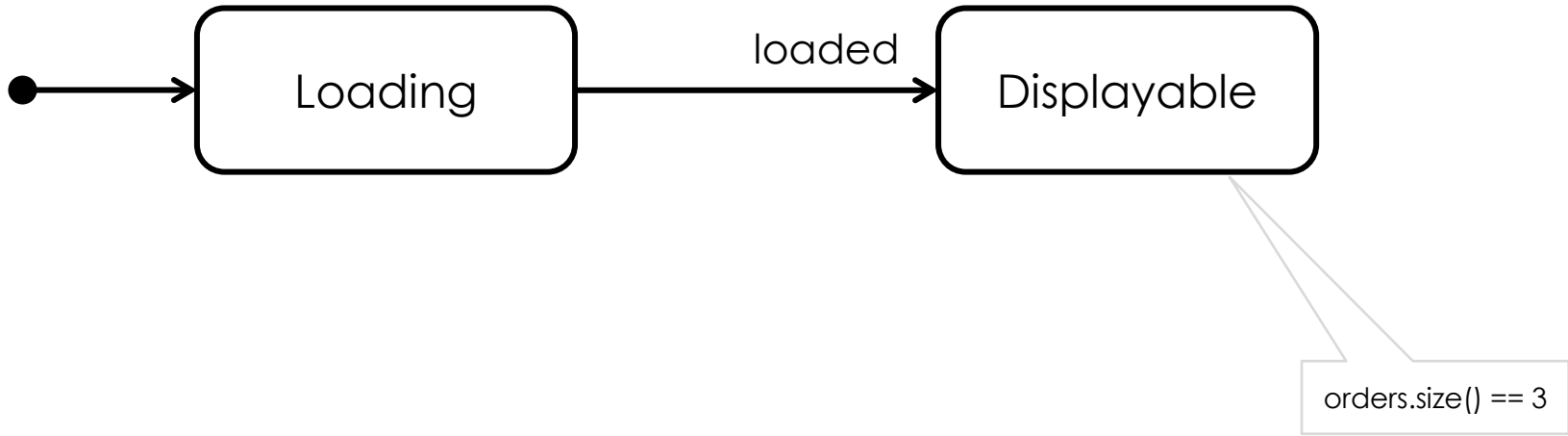
A large fraction of the flaws in software development are due to programmers not fully understanding all the possible states their code may execute in.

John Carmack


Displayable
orders.size() == 0


Displayable
loaded
orders.size() == 3


Loading
orders.size() == 0


Loading
loaded
Displayable
orders.size() == 3

Please wait...

You have 3 orders

There are only two hard things in Computer Science: cache invalidation and naming things.

Phil Karlton

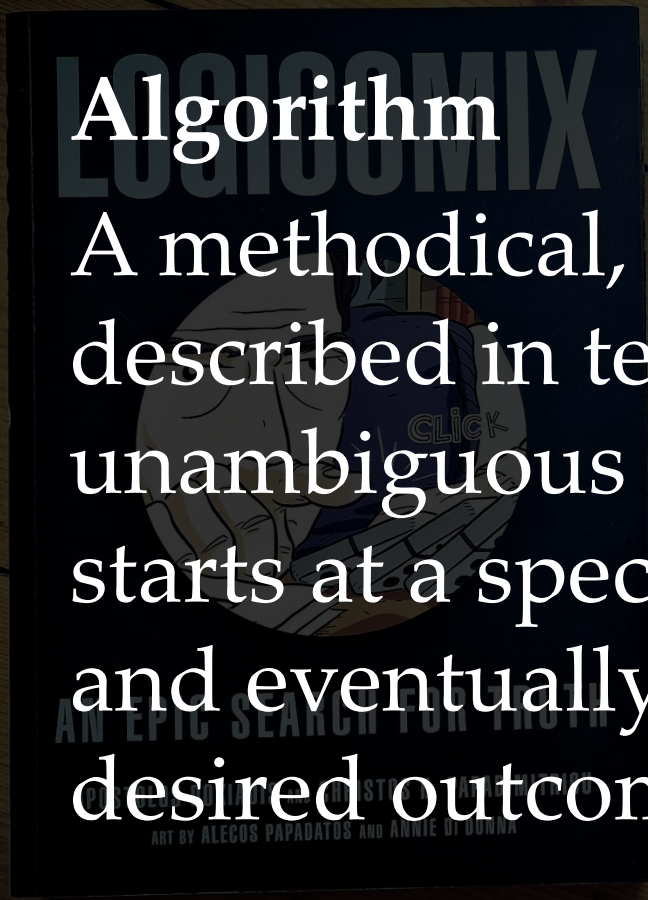
<algorithm>

LOGICOMIX

AN EPIC SEARCH FOR TRUTH

APOSTOLOS DOXIADIS AND CHRISTOS H. PAPADIMITRIOU

ART BY ALECOS PAPADATOS AND ANNIE DI DONNA



Algorithm

A methodical, step-by-step procedure described in terms of totally unambiguous instructions, which starts at a specified initial condition and eventually terminates with the desired outcome.

std::find

std::linear_search

`std::binary_search`

std::lower_range

`std::sort`

std::introsort

Permutation sort takes us to $O(n!)$ — that's right, factorial time.

$O(MG)!$

In essence, it is an unoptimised search through the permutations of the input values until it finds the one arrangement that is sorted.

Kevlin Henney
"A Sort of Permutation"
kevlinhenney.medium.com/a-sort-of-permutation-768c1a7e029b

```
void permutation_sort(auto begin, auto end)
{
    while (std::next_permutation(begin, end))
        ;
}
```

Surely, there can be nothing worse than permutation sort in terms of performance?

Kevlin Henney
"The Most Bogus Sort"
kevlinhenney.medium.com/the-most-bogus-sort-3879e2e98e67


Please
hold...
newcrafts
instanceof American Pale Ale

Surely, there can be nothing worse than permutation sort in terms of performance?

Don’t be so sure.

The essence of bogosort is to shuffle the values randomly until they are sorted.

Kevlin Henney
“The Most Bogus Sort”
kevlinhenney.medium.com/the-most-bogus-sort-3879e2e98e67

```
void bogosort(auto begin, auto end)
{
    while (!std::is_sorted(begin, end))
        std::random_shuffle(begin, end);
}
```

```
void bogosort(auto begin, auto end)
{
    do
        std::random_shuffle(begin, end);
    while (!std::is_sorted(begin, end));
}
```

```
void bogosort(auto begin, auto end)
{
    std::mt19937 randomness;
    do
        std::shuffle(begin, end, randomness);
    while (!std::is_sorted(begin, end));
}
```

Any one who considers arithmetical methods of producing random numbers is, of course, in a state of sin.

John von Neumann

Various Techniques Used in Connection with Random Digits

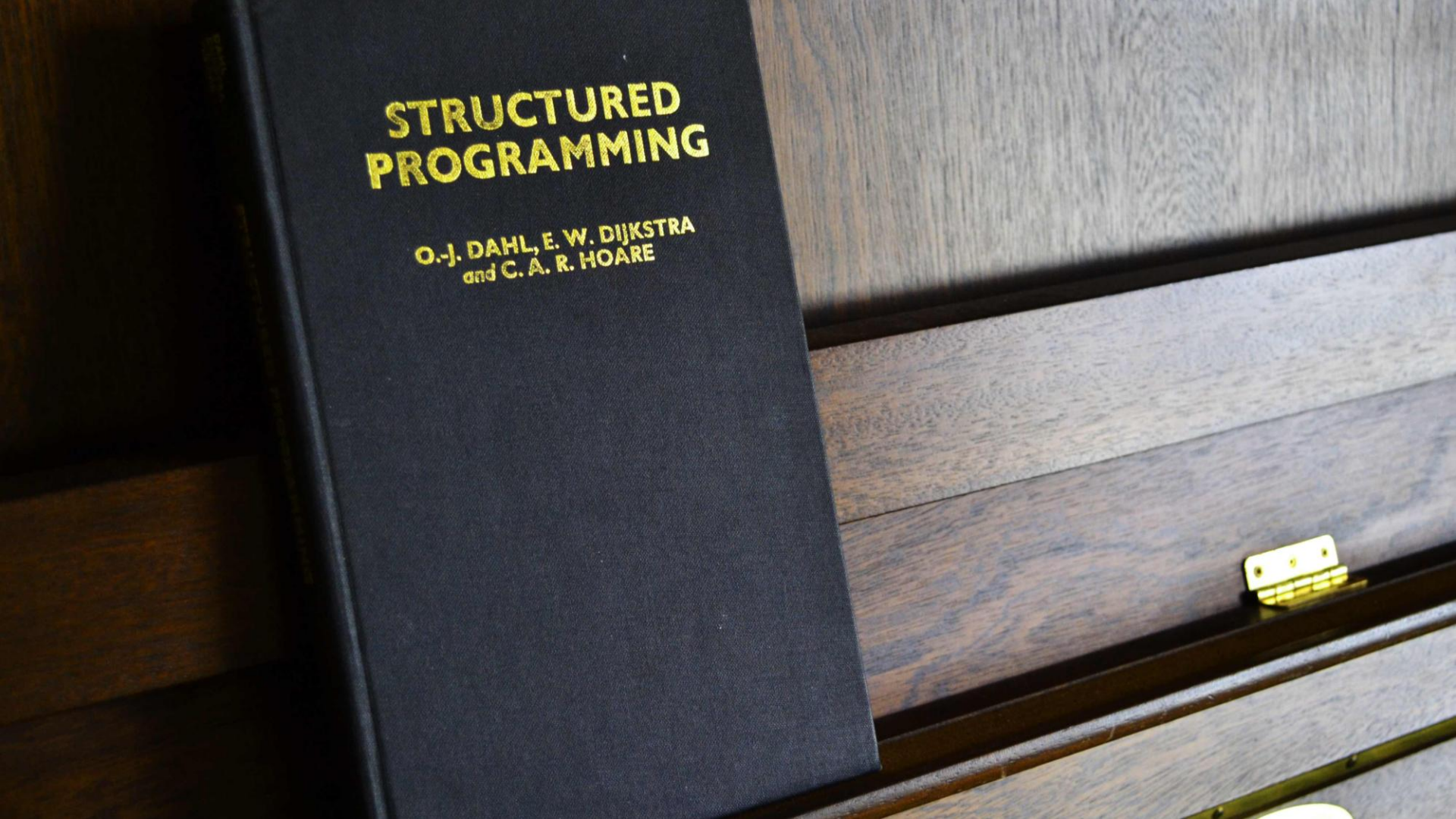
```
void bogosort(auto begin, auto end)
{
    std::random_device randomness;
    do
        std::shuffle(begin, end, randomness);
    while (!std::is_sorted(begin, end));
}
```

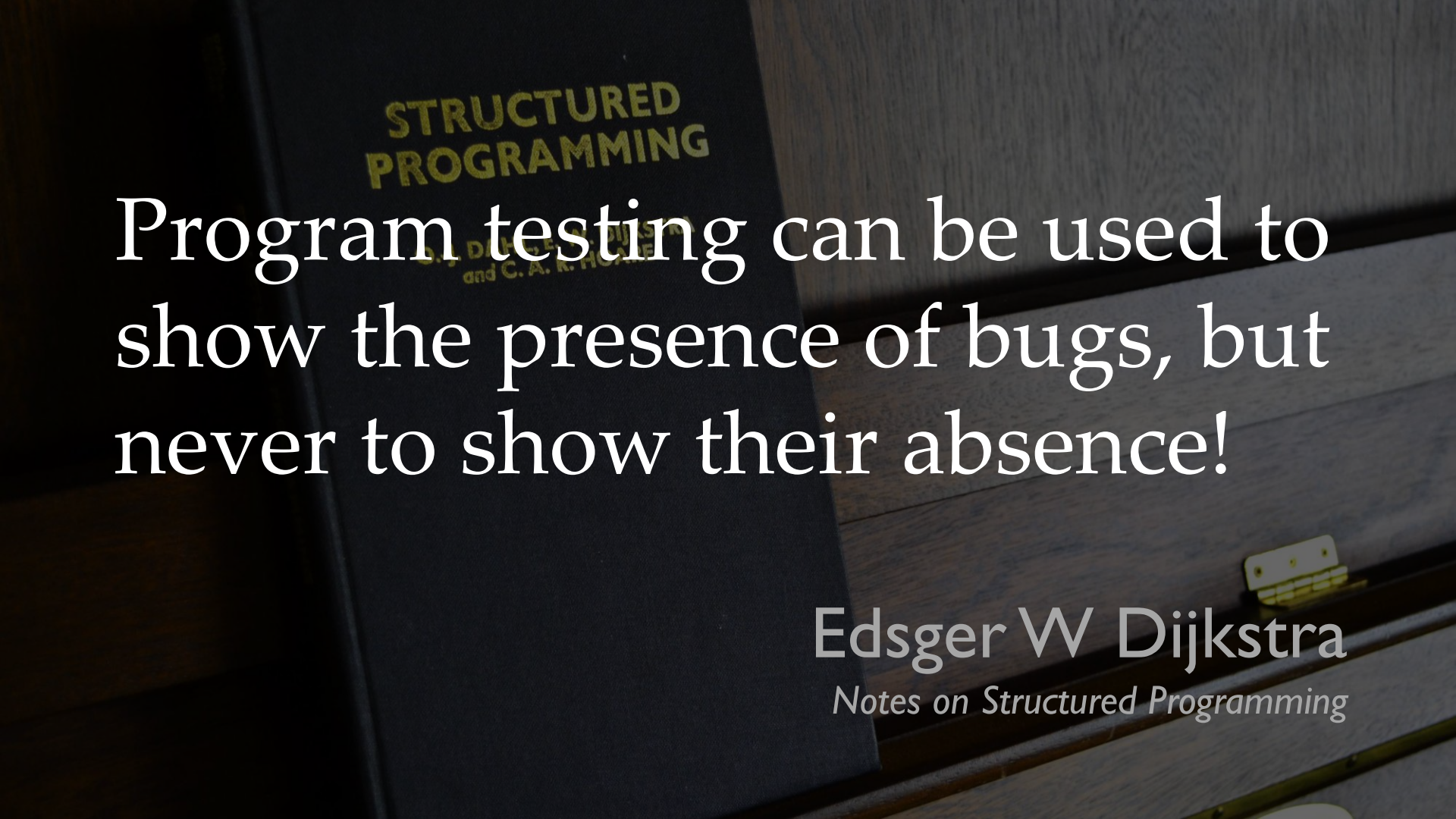
```
void bogosort(auto begin, auto end)
{
    std::random_device seed;
    std::mt19936 randomness(seed());
    do
        std::shuffle(begin, end, randomness);
    while (!std::is_sorted(begin, end));
}
```

algorithm?

A procedure which always terminates is called an *algorithm*.

John E Hopcroft & Jeffrey D Ullman
Formal Languages and their Relation to Automata


STRUCTURED
PROGRAMMING
O.-J. DAHL, E. W. DIJKSTRA
and C. A. R. HOARE



Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger W Dijkstra

Notes on Structured Programming

```
std::vector actual {3, 1, 4, 1, 5, 9};

bogosort(actual.begin(), actual.end());

assert(std::is_sorted(actual.begin(), actual.end());
```

```
std::vector actual {3, 1, 4, 1, 5, 9};
const auto copy = actual;
bogosort(actual.begin(), actual.end());

assert(std::is_sorted(actual.begin(), actual.end());
assert(
    std::is_permutation(
        actual.begin(), actual.end(), copy.begin()));
```

```
std::vector actual {3, 1, 4, 1, 5, 9};
const std::vector expected {1, 1, 3, 4, 5, 9};
bogosort(actual.begin(), actual.end());

assert(actual == expected);
```

```
std::signal(SIGALRM, [](int) {assert(false);});
alarm(???);
std::vector actual {3, 1, 4, 1, 5, 9};
const std::vector expected {1, 1, 3, 4, 5, 9};
bogosort(actual.begin(), actual.end());

assert(actual == expected);
```

```
std::signal(SIGALRM, [](int) {assert(false);});
alarm(INFINITY);
std::vector actual {3, 1, 4, 1, 5, 9};
const std::vector expected {1, 1, 3, 4, 5, 9};
bogosort(actual.begin(), actual.end());

assert(actual == expected);
```

```
std::signal(SIGALRM, [](int) {assert(false);});
alarm(INFINITY);
std::vector actual {3, 1, 4, 1, 5, 9};
const std::vector expected {1, 1, 3, 4, 5, 9};
bogosort(actual.begin(), actual.end());
alarm(0);
assert(actual == expected);
```

```
std::signal(SIGALRM, [](int) {assert(false);});
alarm(static_cast<unsigned>(INFINITY));
std::vector actual {3, 1, 4, 1, 5, 9};
const std::vector expected {1, 1, 3, 4, 5, 9};
bogosort(actual.begin(), actual.end());
alarm(0);
assert(actual == expected);
```

```
std::signal(SIGALRM, [](int) {assert(false);});
alarm(1);
std::vector actual {3, 1, 4, 1, 5, 9};
const std::vector expected {1, 1, 3, 4, 5, 9};
bogosort(actual.begin(), actual.end());
alarm(0);
assert(actual == expected);
```


Halting
Problem

4 Every truth can be established where it applies

On Formally Undecidable Propositions Of Principia Mathematica And Related Systems

KURT GÖDEL

Translated by
B. MELTZER

Introduction by
R B. BRAITHWAITE

In 1911 Russell & Whitehead published Principia Mathematica, with the goal of providing a solid foundation for all of mathematics.

In 1931 Gödel's Incompleteness Theorem shattered the dream, showing that for any consistent axiomatic system there will always be theorems that cannot be proven within the system.

Adrian Colyer

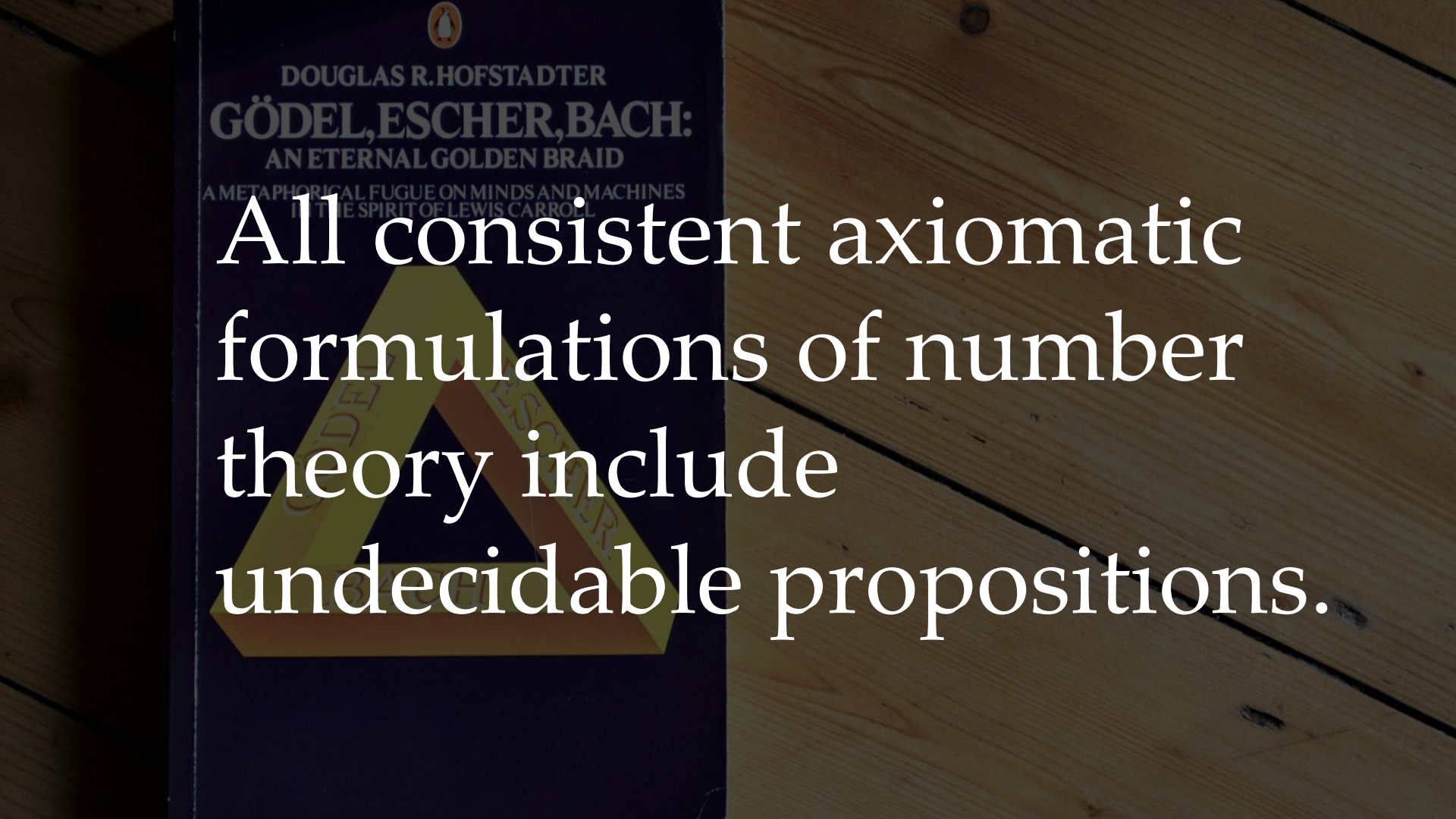
blog.acolyer.org/2020/02/03/measure-mismeasure-fairness

DOUGLAS R. HOFSTADTER

GÖDEL, ESCHER, BACH:

AN ETERNAL GOLDEN BRAID

A METAPHORICAL FUGUE ON MINDS AND MACHINES
IN THE SPIRIT OF LEWIS CARROLL



All consistent axiomatic formulations of number theory include undecidable propositions.

undecidable propositions

How long is a piece of string?

```
size_t strlen(const char * s);
```

```
size_t strlen(const char * s)
{
    size_t n = 0;
    while (s[n] != '\0')
        ++n;
    return n;
}
```

```
size_t strlen(const char * s)
{
    assert(s != NULL);

    size_t n = 0;
    while (s[n] != '\0')
        ++n;
    return n;
}
```

```
size_t strlen(const char * s)
{
    assert(s != NULL);
    assert(∃n (s[n] == '\0') &&
           ∀i∈0..n (s[i] is defined));
    size_t n = 0;
    while (s[n] != '\0')
        ++n;
    return n;
}
```

```
void well_defined(void)
{
    char s[] = "Be excellent to each other";
    printf("%s\n%zu\n", s, strlen(s));
}
```

Be excellent to each other
26

```
void insufficient_space(void)
{
    char s[5] = {'B', 'o', 'g', 'u', 's'};
    printf("%s\n%zu\n", s, strlen(s));
}
```

```
void insufficient_space(void)
{
    char s[5] = "Bogus";
    printf("%s\n%zu\n", s, strlen(s));
}
```

Bogus
5

```
void uninitialised_pointer_value(void)
{
    char * s;
    printf("%s\n%zu\n", s, strlen(s));
}
```

Wovon man nicht sprechen kann, über muss man schweigen.

Ludwig Wittgenstein

Logisch-philosophische Abhandlung

Whereof one cannot speak, thereof one must be silent.

Ludwig Wittgenstein

Tractatus Logico-Philosophicus

One premise of many models of fairness in machine learning is that you can measure ('prove') fairness of a machine learning model from within the system – i.e. from properties of the model itself and perhaps the data it is trained on.

To show that a machine learning model is fair, you need information from outside of the system.

Adrian Colyer

blog.acolyer.org/2020/02/03/measure-mismeasure-fairness


99 second hand smartphones
are transported in a handcart
to generate virtual traffic jam
in Google Maps.
Simon Weckert
simonweckert.com/googlemapshacks.html

engagement

the state of being engaged is

engagement

engagement

is emotional involvement or commitment

“engagement”

clicks & shares

We must be careful not to confuse data with the abstractions we use to analyse them.

William James

3 The future is knowable before it happens

To me programming is more than an important practical art. It is also a gigantic undertaking in the foundations of knowledge.

Grace Hopper

0. lack of ignorance
1. lack of knowledge
2. lack of awareness
3. lack of process
4. meta-ignorance

Phillip G Armour

Five Orders of Ignorance

0. lack of ignorance
1. lack of knowledge
2. lack of awareness
3. lack of process

Phillip G Armour
Five Orders of Ignorance

known knowns
known unknowns
unknown unknowns
unknowable unknowns

known knowns
known unknowns
unknown unknowns
unknowable unknowns

Halting Problem

Seth Rosen
@sethrosen

When naming things like tables, always future proof

If you sell wine, calling a table “wines” will be confusing when expanding to beer

“Beverages” will be confusing when you sell ice

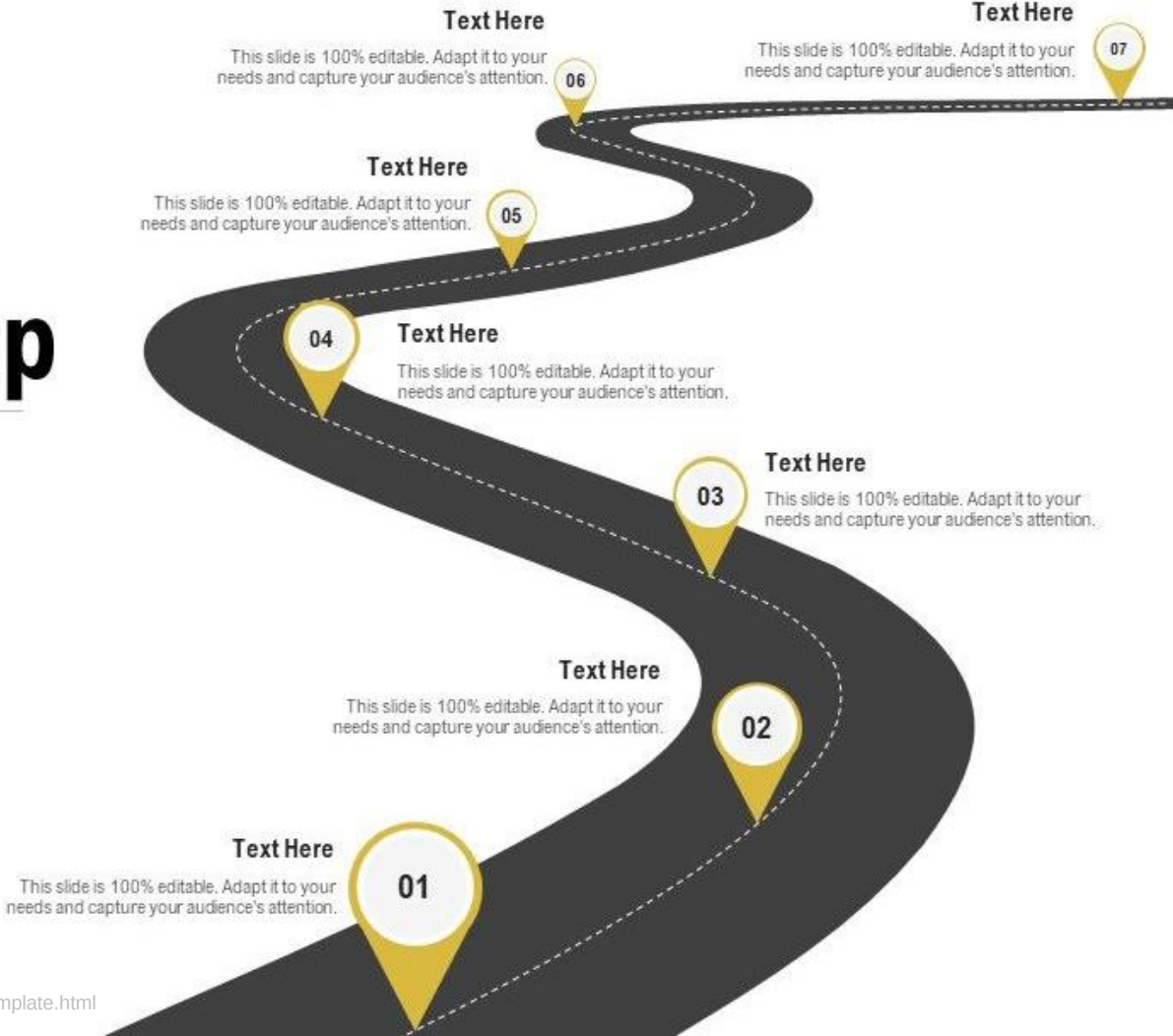
“Products” confusing when expand to services

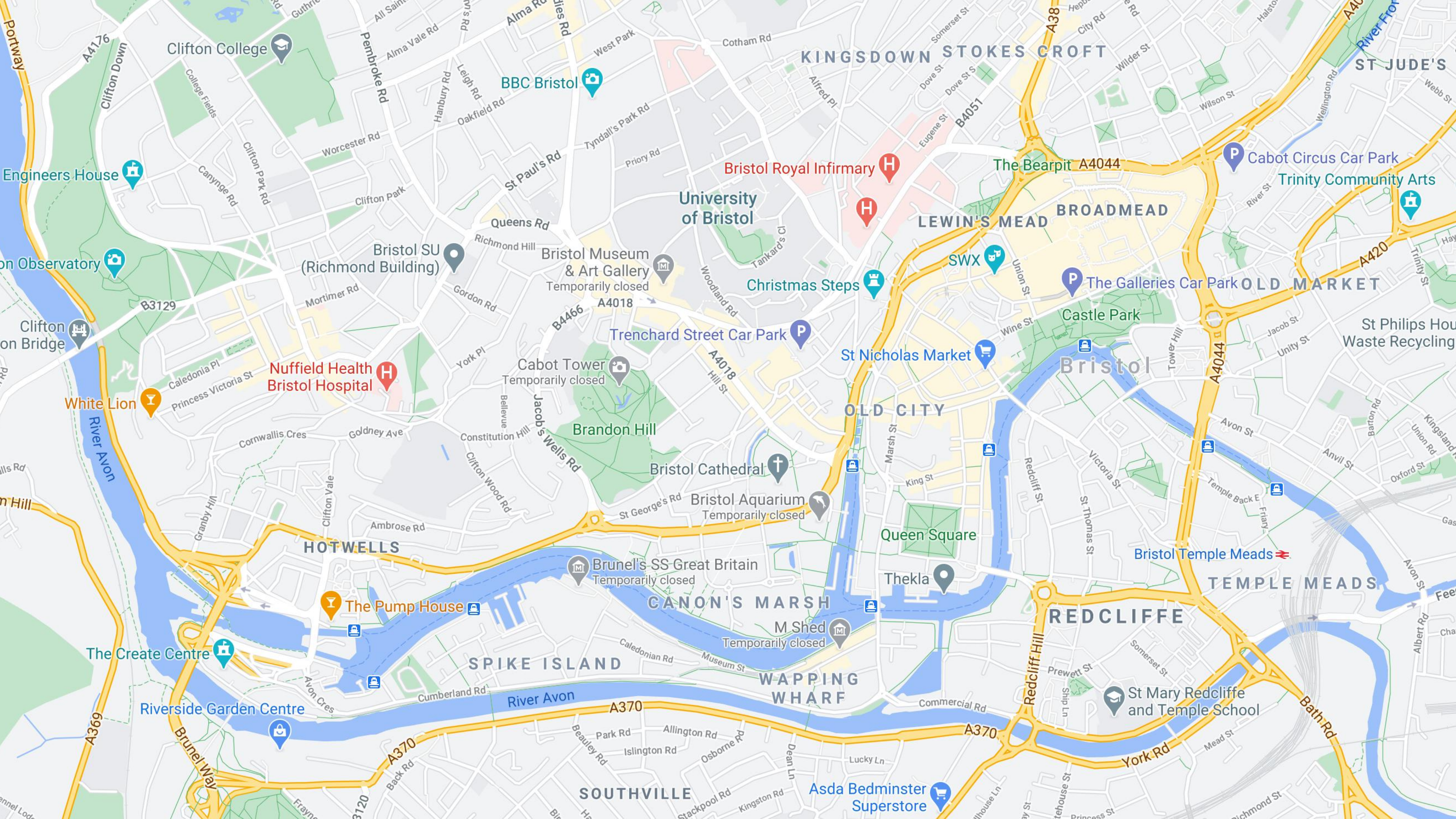
recommend tables be called “stuff” or “table1”

1:15 PM · Aug 20, 2021

x.com/sethrosen/status/1428692052968185863

Roadmap





Clifton College
BBC Bristol
KINGSDOWN
STOKES CROFT
ST JUDE'S
Engineers House
Bristol Royal Infirmary
University of Bristol
The Bearpit
A4044
Cabot Circus Car Park
Trinity Community Arts
LEWIN'S MEAD
BROADMEAD
Bristol SU (Richmond Building)
Bristol Museum & Art Gallery
Temporarily closed
SWX
Christmas Steps
The Galleries Car Park
OLD MARKET
Castle Park
Clifton Suspension Bridge
Trenchard Street Car Park
St Nicholas Market
St Philips Household Waste Recycling
Bristol
Nuffield Health Bristol Hospital
Cabot Tower
Temporarily closed
White Lion
Brandon Hill
OLD CITY
Bristol Cathedral
Bristol Aquarium
Temporarily closed
Queen Square
Bristol Temple Meads
HOTWELLS
Brunel's SS Great Britain
Temporarily closed
Thekla
TEMPLE MEADS
The Pump House
CANON'S MARSH
REDCLIFFE
M Shed
Temporarily closed
The Create Centre
SPIKE ISLAND
WAPPING WHARF
St Mary Redcliffe and Temple School
Riverside Garden Centre
River Avon
SOUTHVILLE
Asda Bedminster Superstore
Portway
A4176
Clifton Down
Pembroke Rd
Alma Vale Rd
Cotham Rd
Worcester Rd
Oakfield Rd
Tyndall's Park Rd
St Paul's Rd
Clifton Park
Queens Rd
Richmond Hill
B3129
Mortimer Rd
Gordon Rd
A4018
B4466
York Pl
Jacob's Wells Rd
Constitution Hill
Goldney Ave
Cornwallis Cres
Clifton Wood Rd
St George's Rd
Ambrose Rd
B4051
A38
A420
Union St
Wine St
Marsh St
King St
Redcliff St
Victoria St
Temple Back E
Avon St
Redcliff Hill
Prewett St
Commercial Rd
York Rd
Bath Rd
A370
Cumberland Rd
Caledonian Rd
Museum St
Park Rd
Allington Rd
Islington Rd
Beauley Rd
Back Rd
Brunel Way
A369
Lucky Ln
Dean Ln
Kingston Rd
Stackpool Rd

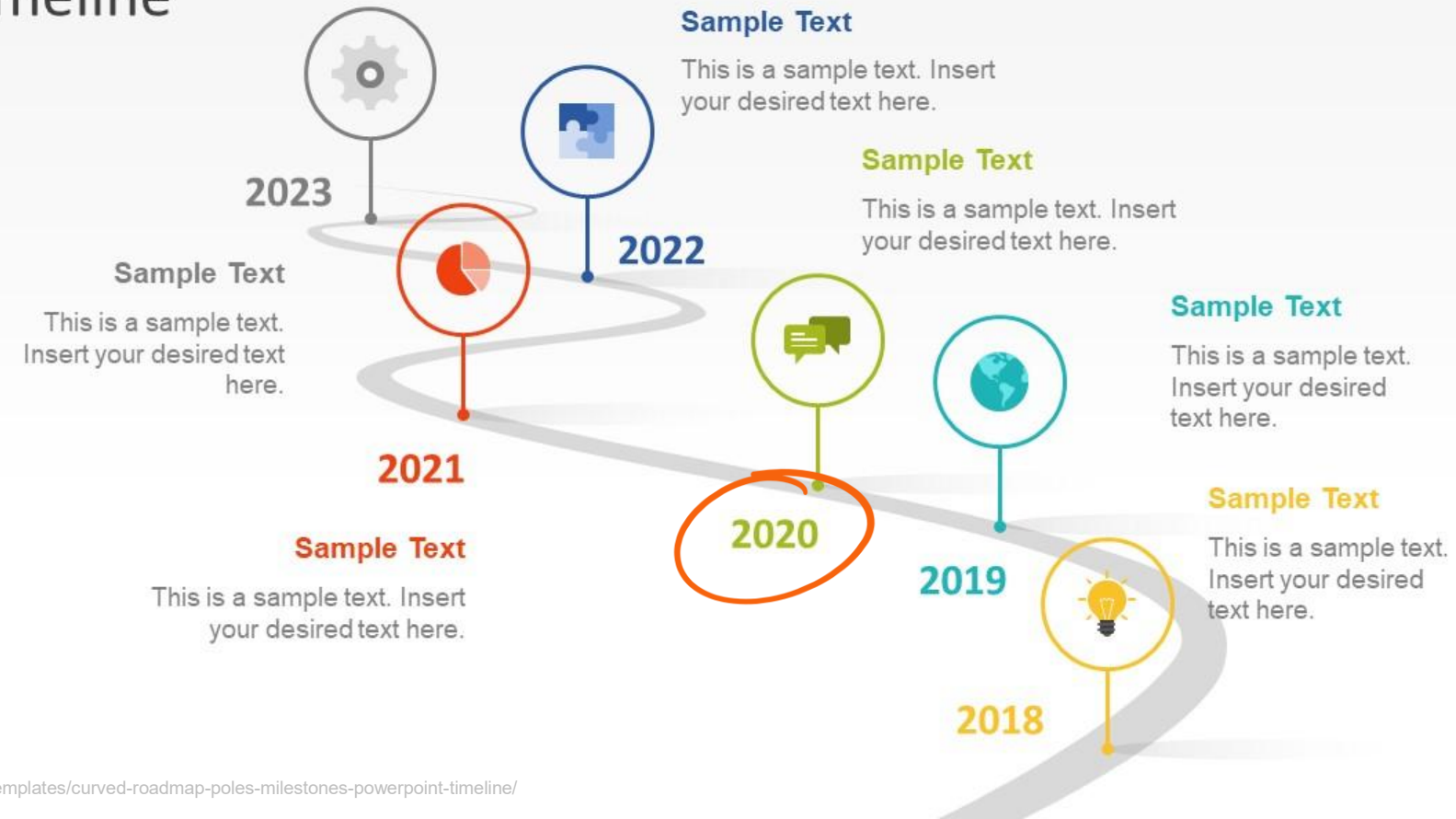
The whole point of a roadmap is that it shows you a network of roads, not just one. A real roadmap is filled with possibilities and choices.

That's the point of roadmaps, whether on paper or online. Your decisions are not set in stone; you can respond to change.

Kevlin Henney

devm.io/programming/software-development-roadmap

Curved Roadmap with Poles Milestones PowerPoint Timeline




prioritise by
business value

prioritise by business value estimate

Impossible to see the future is.

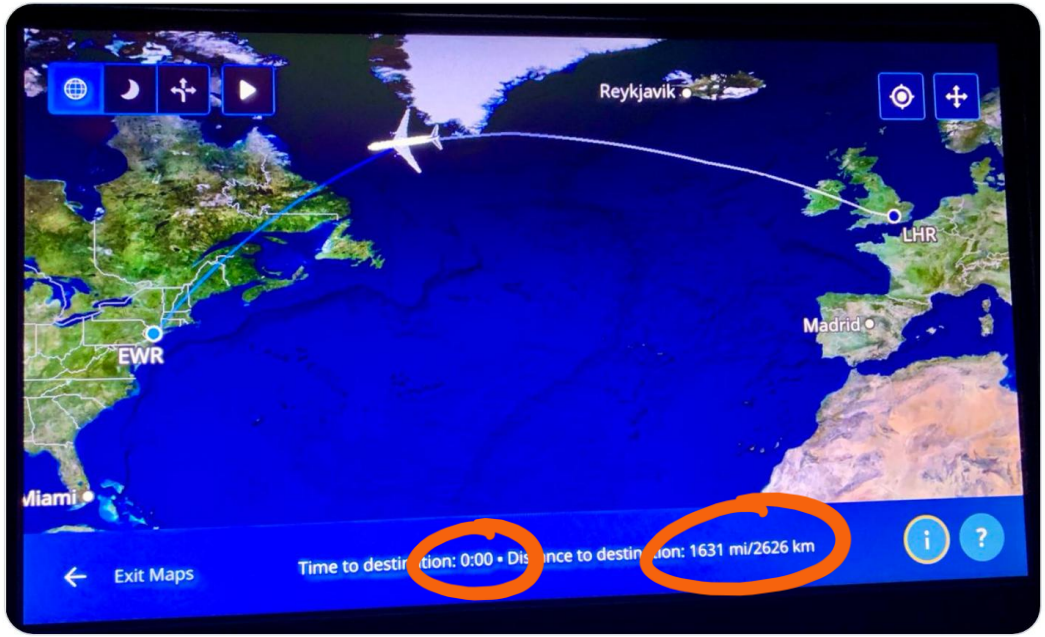
Yoda

2

A distributed system is knowable

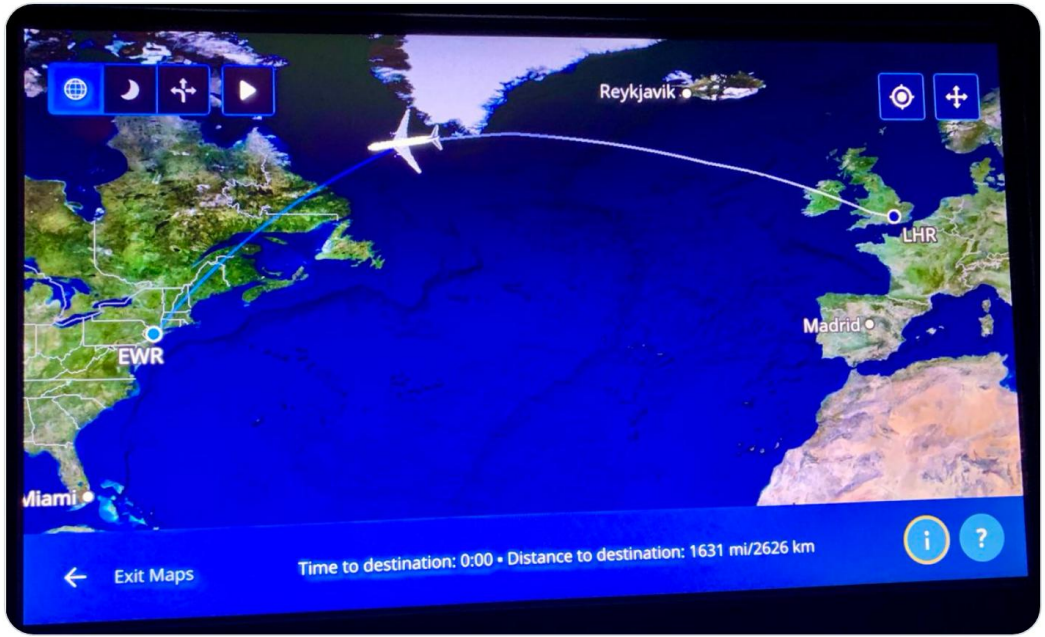
Martin Fowler
@martinfowler

@KevlinHenney you ought to know that @united may have discovered a kink in the time-space continuum over the Atlantic. But sadly they weren't able to take advantage of it and get me home earlier.

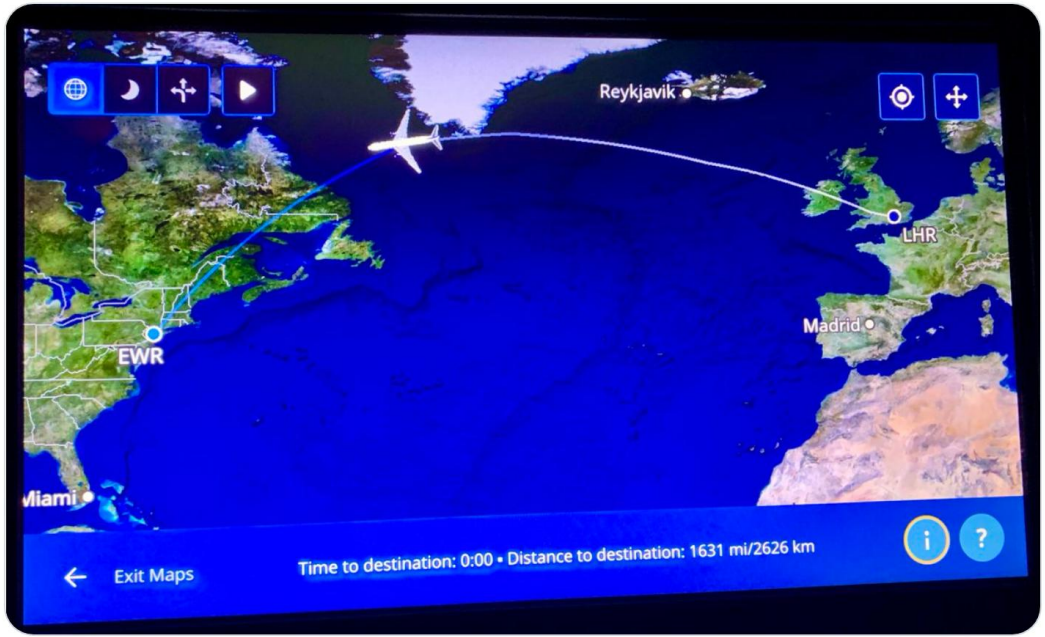


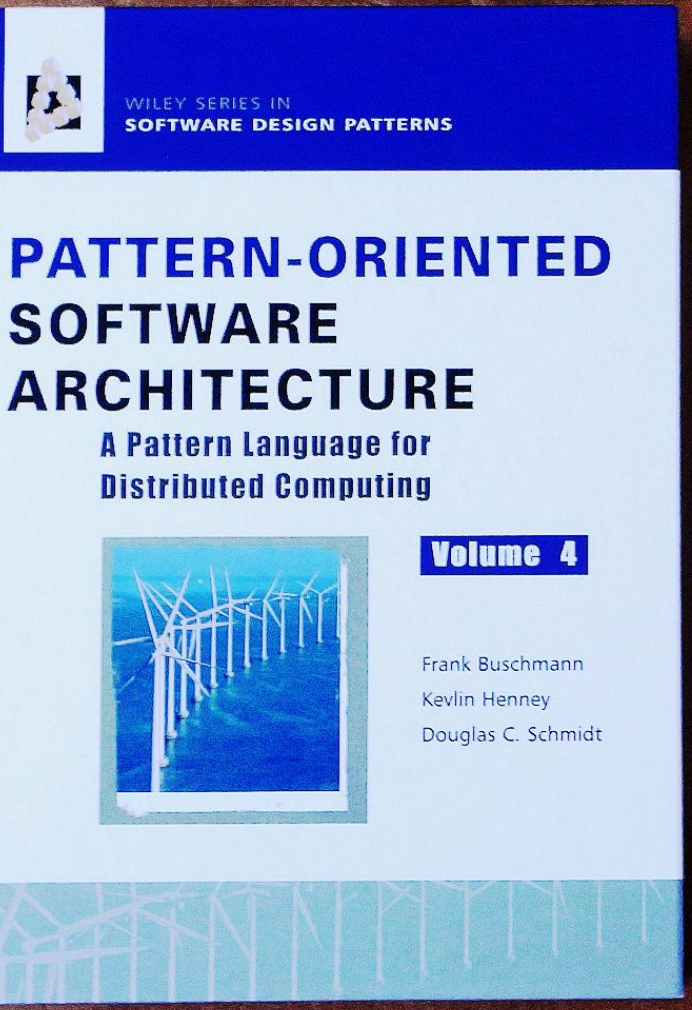
12:44 AM · Oct 7, 2022 twitter.com/martinfowler/status/1578154212974022657

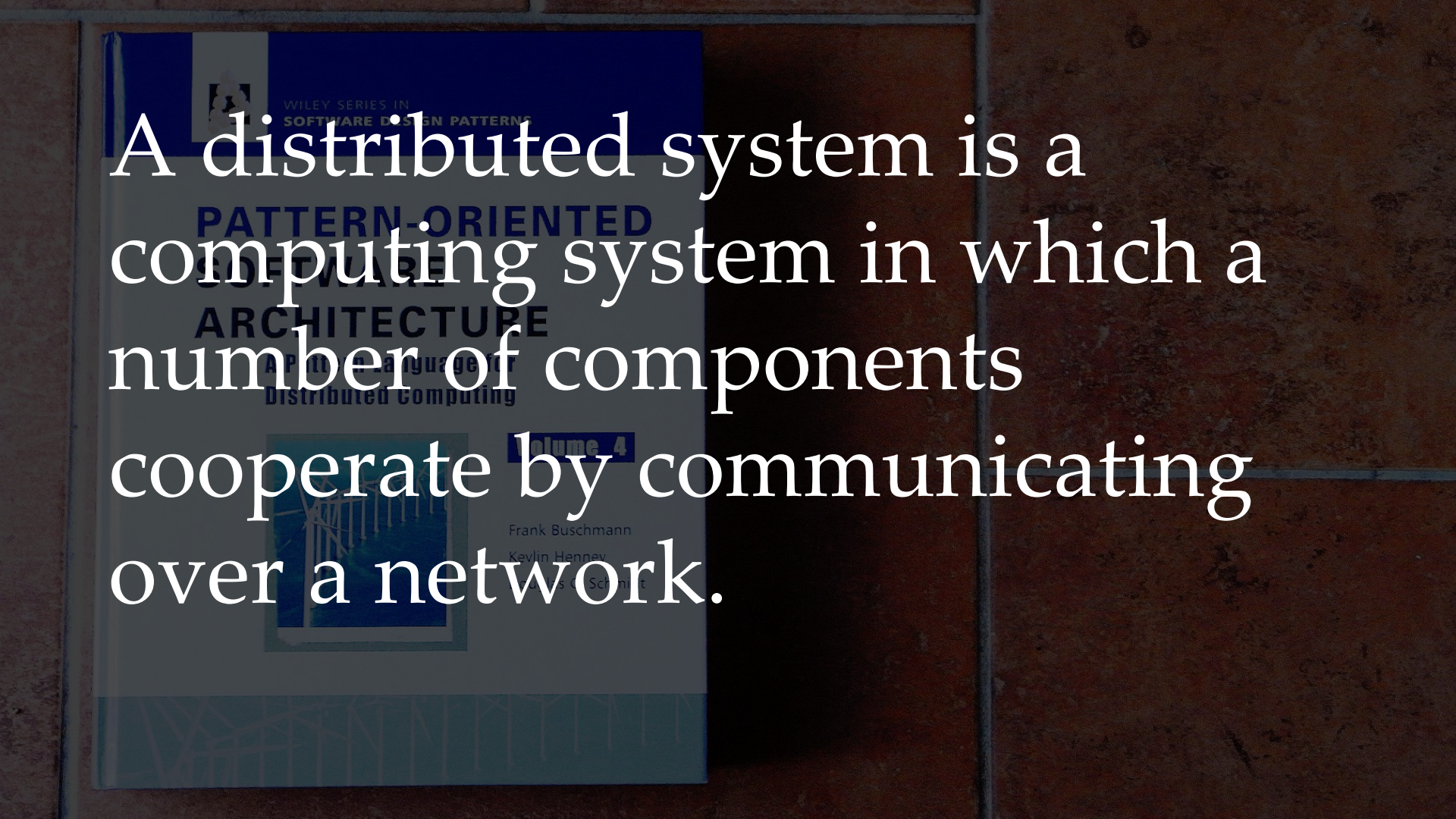
5577 km


Reykjavik
LHR
Madrid
EWR
Miami
Exit Maps
Time to destination: 0:00 • Distance to destination: 1631 mi/2626 km

19 c ms


Reykjavik
LHR
Madrid
EWR
Miami
Exit Maps
Time to destination: 0:00 • Distance to destination: 1631 mi/2626 km


WILEY SERIES IN
SOFTWARE DESIGN PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
A Pattern Language for Distributed Computing
Volume 4
Frank Buschmann
Kevlin Henney
Douglas C. Schmidt



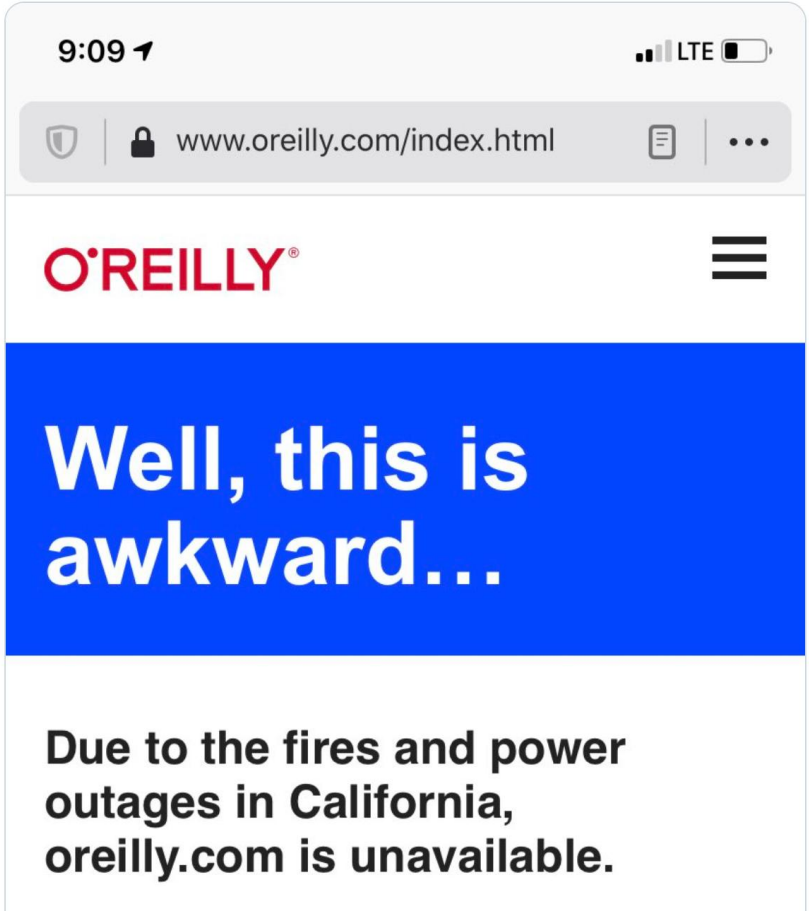
A distributed system is a computing system in which a number of components cooperate by communicating over a network.

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Leslie Lamport

Charlie Morris
@cdmo

Fire in California, can't read your ebook in Pennsylvania



twitter.com/cdmo/status/1188806227079389184

Brewer’s theorem

CAP theorem

C

A

P

Consistency
Availability
Partition tolerance

Consistency

Availability

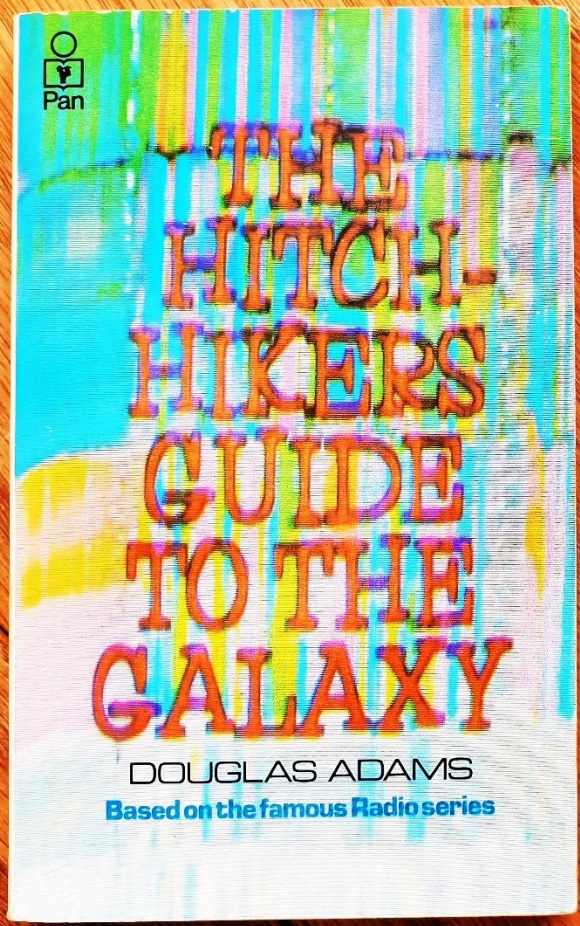
Partition tolerance

~~Consistency~~

Availability

Partition tolerance

Consistency
Availability
Partition tolerance


Pan
THE HITCH-HIKERS GUIDE TO THE GALAXY
DOUGLAS ADAMS
Based on the famous Radio series

We demand rigidly defined areas of doubt and uncertainty!

$$\Delta x \, \Delta p \geq \frac{\hbar}{2}$$

There are only two hard things in Computer Science: cache invalidation and naming things.

Phil Karlton

You have 2 orders

You have 3 orders

It is a feature of a distributed system that it may not be in a consistent state, but it is a bug for a client to contradict itself.

*twitter.com/KevlinHenney/status/135195694287755264*6

Technical debt is quantifiable as financial debt


As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.
Meir M Lehman
“Programs, Life Cycles, and Laws of Software Evolution”


maintenance


technical
debt


technical
neglect

Technical debt is a wonderful metaphor developed by Ward Cunningham to help us think about this problem.

Martin Fowler

martinfowler.com/bliki/TechnicalDebt.html

Like a financial debt, the technical debt incurs interest payments, which come in the form of the extra effort that we have to do in future development because of the quick and dirty design choice.

Martin Fowler

martinfowler.com/bliki/TechnicalDebt.html

Technical debt is a wonderful metaphor developed by Ward Cunningham to help us think about this problem.

Martin Fowler

martinfowler.com/bliki/TechnicalDebt.html

metaphor

metaphor

Found myself again cautioning against the category error of treating the technical debt metaphor literally and numerically: converting code quality into a currency value on a dashboard.

Kevlin Henney

twitter.com/KevlinHenney/status/1265766381692846080

technical debt = cost of repaying the debt

technical debt ≠ cost of repaying the debt

technical debt =
cost of owning
the debt

That is the message of the technical debt metaphor: it is not simply a measure of the specific work needed to repay the debt; it is the additional time and effort added to all past, present, and future work that comes from having the debt in the first place.

Kevlin Henney

"On Exactitude in Technical Debt"

oreilly.com/radar/on-exactitude-in-technical-debt/

0 Finishing on time

Reality cannot be ignored except at a price.

Aldous Huxley