# Program with GUTs

@KevlinHenney

# Kevlin Henney

@KevlinHenney
about.me/kevlin
linkedin.com/in/kevlin
kevlin@curbralan.com

feature development

testing? what testing?
¯_(ツ)_/¯

feature development

test long after development

feature development

test just after development

feature development

plain ol' unit testing (POUT)

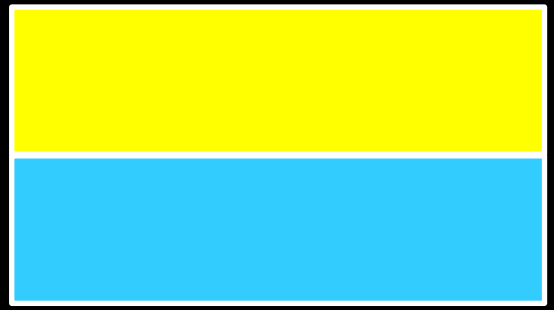

feature development
test alongside/continuously

Test Early.

Test Often.

Test Automatically.

Andrew Hunt & David Thomas
*The Pragmatic Programmer*




iterative test last (ITL)

test-driven development (TDD)

Very many people say “TDD” when they really mean, “I have good unit tests” (“I have GUTs”?)

Alistair Cockburn

*The modern programming professional has GUTs*

# GUTs

A bare minimum definition of what we mean by ‘good’ tests is that when the code is doing the right thing, the tests pass and when a test fails it means that the code is doing the wrong thing.

# algorithm

a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer

algorithm

Concise Oxford English Dictionary

Most of us know those simple sorting algorithms like bubble sort very well.

Or so we thought — have you ever found yourself needing to write down the pseudocode of bubble sort, only to realise that it is not as straightforward as you think and you couldn't get it right the first time?

"Is this the simplest (and most surprising) sorting algorithm ever?"
Stanley P Y Fung
*arxiv.org/abs/2110.01111*

It needs a bit of care to get the loop indices start and end at just the right values, not going out of bounds, or to handle some flag variables correctly. Wouldn't it be nice if there is a simple algorithm with no such hassle?

Here is such an algorithm, that sorts an array $A$ of $n$ elements in non-decreasing order.

"Is this the simplest (and most surprising) sorting algorithm ever?"
Stanley P Y Fung
*arxiv.org/abs/2110.01111*

**Algorithm 1** ICan'tBelieveItCanSort($A[1..n]$)

```
for i = 1 to n do
    for j = 1 to n do
        if A[i] < A[j] then
            swap A[i] and A[j]
```

That's it, really. Just loop over every pair of $(i, j)$ values in the standard double-for-loop way, and compare and swap. What can possibly be simpler?

"Is this the simplest (and most surprising) sorting algorithm ever?"
Stanley P Y Fung
*arxiv.org/abs/2110.01111*

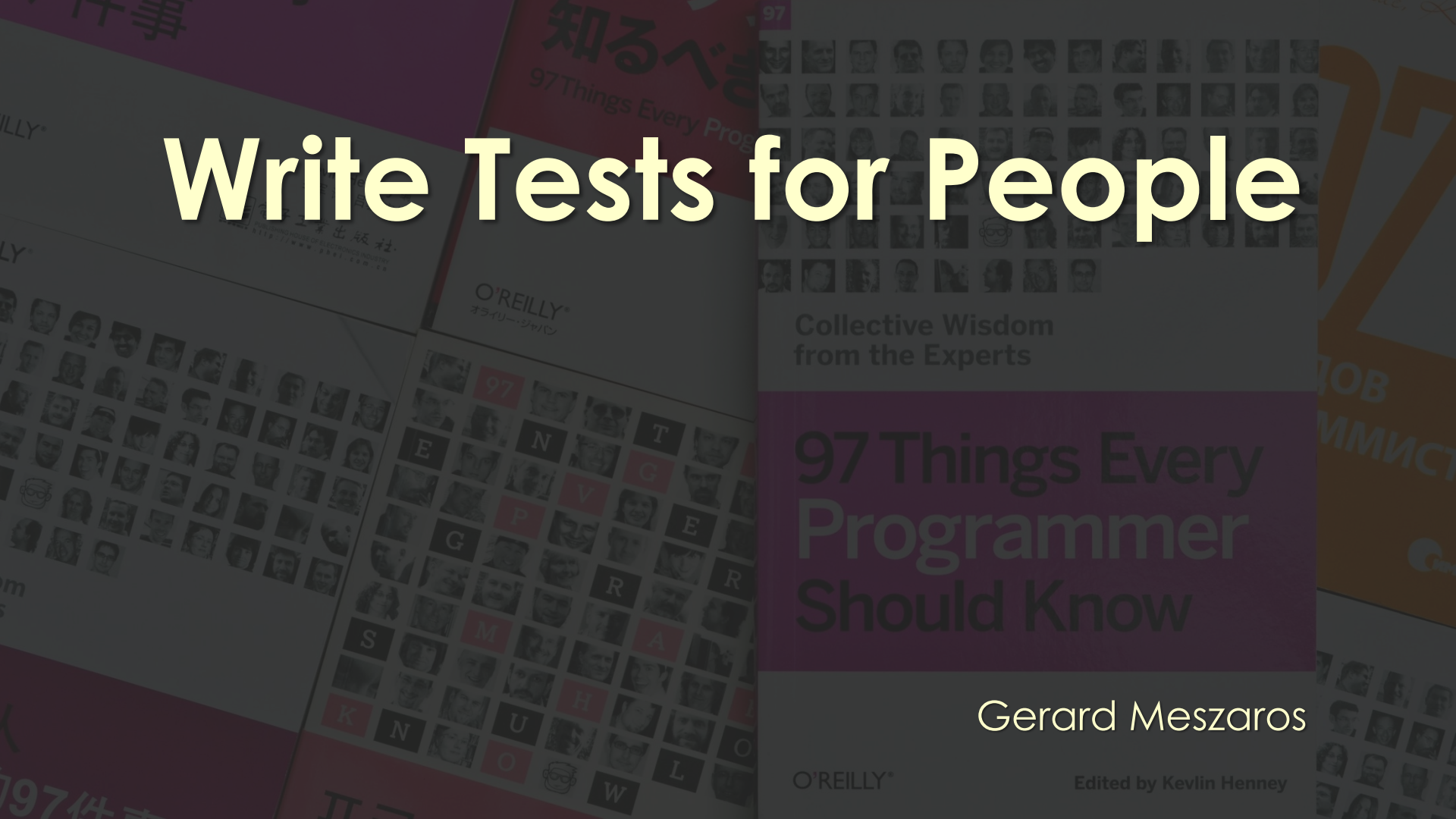
That's it, really. Just loop over every pair of (i, j) values in the standard double-for-loop way, and compare and swap. What can possibly be simpler?

The first reaction of someone seeing this algorithm might be "this cannot possibly be correct", or "you got the direction of the inequality wrong and also the loop indices wrong". But no, it does sort correctly in increasing order.

"Is this the simplest (and most surprising) sorting algorithm ever?"
Stanley P Y Fung
*arxiv.org/abs/2110.01111*

```
template<std::forward_iterator iterator>
void i_cant_believe_it_can_sort(iterator begin, iterator end)
{
    for (auto i = begin; i != end; ++i)
        for (auto j = begin; j != end; ++j)
            if (*i < *j)
                std::iter_swap(i, j);
}
```


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


Write Tests for People
Collective Wisdom
from the Experts
97 Things Every
Programmer
Should Know
Gerard Meszaros
O'REILLY
Edited by Kevlin Henney

So who should you be writing the tests for? For the person trying to understand your code.

Good tests act as documentation for the code they are testing.

Gerard Meszaros

```
TEST(Sorting, sorts) ...
```

```
TEST(Sorting, works) ...
```

```
TEST(Sorting, is_correct) ...
```

```
TEST(Sorting, gives_expected_result) ...
```

```
TEST(Sorting, gives_sorted_values) ...
```

```
TEST(Sorting, gives_sorted_values)
{
    auto many = random_values(20'000);
    i_cant_believe_it_can_sort(many.begin(), many.end());
    EXPECT_TRUE(std::is_sorted(many.begin(), many.end()));
}
```

```
TEST(Sorting_random_values, gives_sorted_values)
{
    auto many = random_values(20'000);
    i_cant_believe_it_can_sort(many.begin(), many.end());
    EXPECT_TRUE(std::is_sorted(many.begin(), many.end()));
}
```

```
auto random_values(std::size_t how_many)
{
    std::vector<unsigned> result;
    std::generate_n(std::back_inserter(result), how_many, std::rand);
    return result;
}
```

```
auto random_values(std::size_t how_many)
{
    const char * env_seed = std::getenv("SORTING_SEED");
    int seed = env_seed ? std::atoi(env_seed) : std::rand();
    std::cout << "SORTING_SEED = " << seed << "\n";
    std::srand(seed);

    std::vector<unsigned> result;
    std::generate_n(std::back_inserter(result), how_many, std::rand);
    return result;
}
```

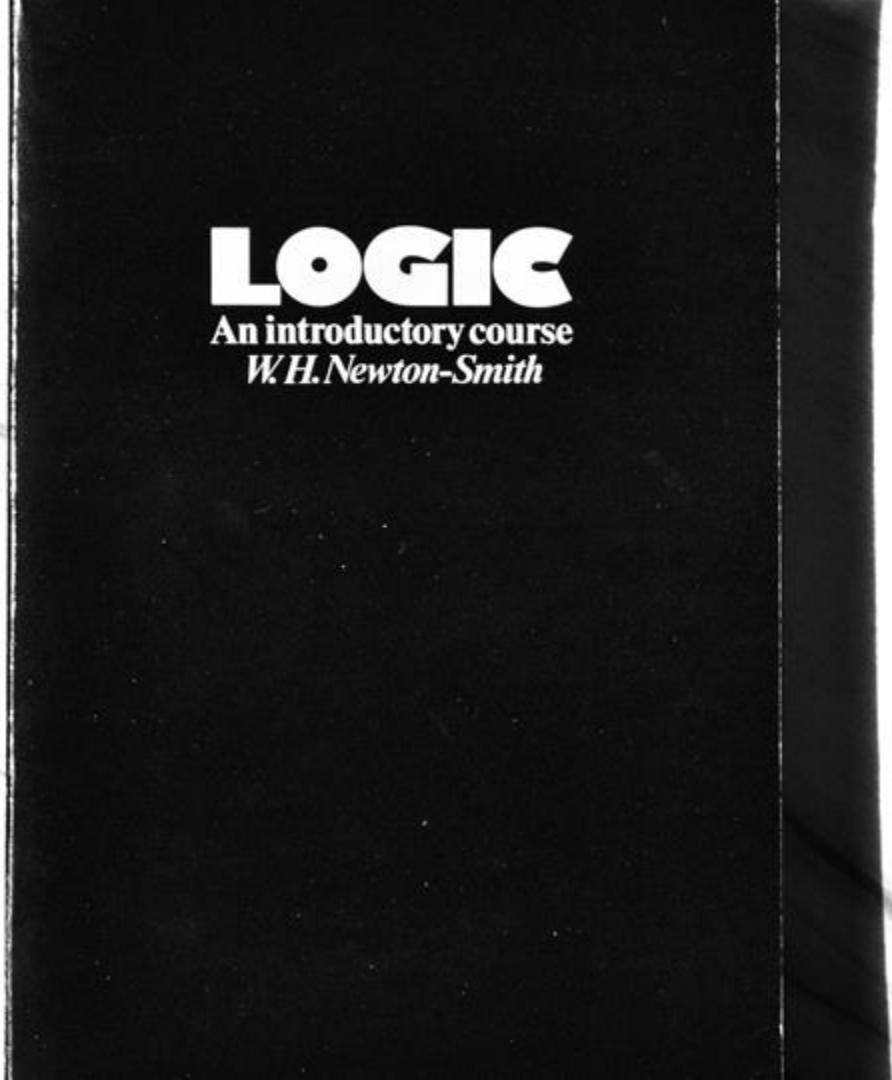
```
TEST(Sorting_random_values, gives_sorted_values)
{
    auto many = random_values(20'000);
    i_cant_believe_it_can_sort(many.begin(), many.end());
    EXPECT_TRUE(std::is_sorted(many.begin(), many.end()));
}
```

```
TEST(Sorting_random_values, gives_sorted_permutation_of_original_values)
{
    auto many = random_values(20'000);
    const auto original = many;

    i_cant_believe_it_can_sort(many.begin(), many.end());

    EXPECT_TRUE(std::is_sorted(many.begin(), many.end()));
    EXPECT_TRUE(std::is_permutation(many.begin(), many.end(), original.begin()));
}
```

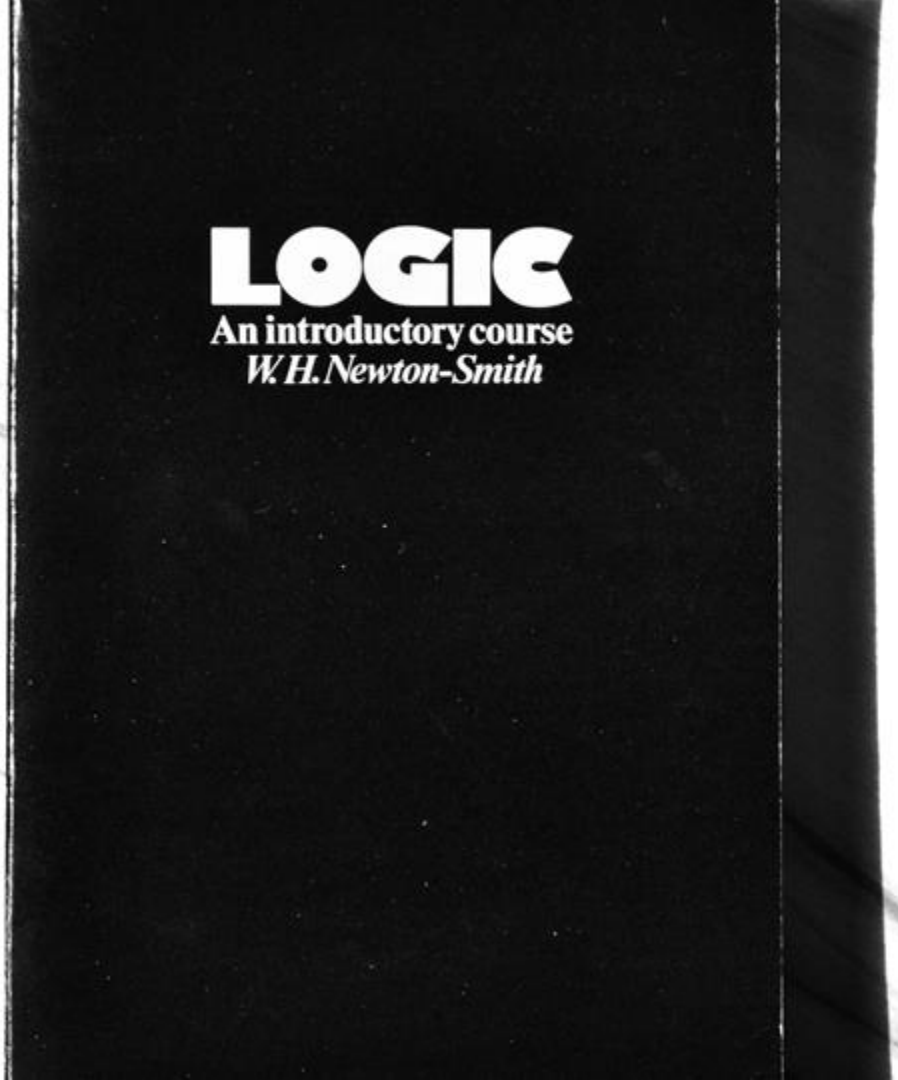
# LOGIC

An introductory course

*W. H. Newton-Smith*




**Propositions are vehicles for stating how things are or might be.**




**Thus only indicative sentences which it makes sense to think of as being true or as being false are capable of expressing propositions.**

```
TEST(Sorting_random_values, gives_sorted_permutation_of_original_values)
{
    auto many = random_values(20'000);
    const auto original = many;

    i_cant_believe_it_can_sort(many.begin(), many.end());

    EXPECT_TRUE(std::is_sorted(many.begin(), many.end()));
    EXPECT_TRUE(std::is_permutation(many.begin(), many.end(), original.begin()));
}
```

```
TEST(Sorting_random_values, gives_same_result_as_std_sort)
{
    auto many = random_values(20'000);
    auto expected = many;
    std::sort(expected.begin(), expected.end());

    i_cant_believe_it_can_sort(many.begin(), many.end());

    ASSERT_EQ(many, expected);
}
```

For tests to drive development they must do more than just test that code performs its required functionality: they must clearly express that required functionality to the reader.

Nat Pryce & Steve Freeman

*Are your tests really driving your development?*

That is, they must be clear specifications of the required functionality.

Nat Pryce & Steve Freeman

*Are your tests really driving your development?*

```
TEST(Sorting_an_empty_range, has_no_effect) ...
TEST(Sorting_a_single_value, leaves_value_unchanged) ...
TEST(Sorting_an_equal_pair, leaves_values_unchanged) ...
TEST(Sorting_an_ascending_pair, leaves_values_unchanged) ...
TEST(Sorting_a_descending_pair, exchanges_values) ...
TEST(Sorting_equal_values, leaves_values_unchanged) ...
TEST(Sorting_ascending_values, leaves_values_unchanged) ...
TEST(Sorting_descending_values, reverses_values) ...
TEST(Sorting_mixed_values, puts_them_into_non_descending_order) ...
```

```
TEST(Sorting_an_empty_range, has_no_effect) ...
TEST(Sorting_a_single_value, leaves_value_unchanged) ...
TEST(Sorting_an_equal_pair, leaves_values_unchanged) ...
TEST(Sorting_an_ascending_pair, leaves_values_unchanged) ...
TEST(Sorting_a_descending_pair, exchanges_values) ...
TEST(Sorting_equal_values, leaves_values_unchanged) ...
TEST(Sorting_ascending_values, leaves_values_unchanged) ...
TEST(Sorting_descending_values, reverses_values) ...
TEST(Sorting_mixed_values, puts_them_into_non_descending_order) ...
TEST(Sorting_random_values, gives_sorted_permutation_of_original_values) ...
TEST(Sorting_random_values, gives_same_result_as_std_sort) ...
```

```
TEST(Sorting_an_empty_range, has_no_effect)
{
    std::vector<std::string> empty;
    ASSERT_NO_THROW(i_cant_believe_it_can_sort(empty.begin(), empty.end()));
}
TEST(Sorting_a_single_value, leaves_value_unchanged) ...
TEST(Sorting_an_equal_pair, leaves_values_unchanged) ...
TEST(Sorting_an_ascending_pair, leaves_values_unchanged) ...
TEST(Sorting_a_descending_pair, exchanges_values) ...
TEST(Sorting_equal_values, leaves_values_unchanged) ...
TEST(Sorting_ascending_values, leaves_values_unchanged) ...
TEST(Sorting_descending_values, reverses_values) ...
TEST(Sorting_mixed_values, puts_them_into_non_descending_order) ...
TEST(Sorting_random_values, gives_sorted_permutation_of_original_values) ...
TEST(Sorting_random_values, gives_same_result_as_std_sort) ...
```

```
TEST(Sorting_an_empty_range, has_no_effect)
{
    std::vector empty {"Z"s, "A"s};
    const auto unchanged = empty;
    i_cant_believe_it_can_sort(empty.begin() + 1, empty.end() - 1);
    ASSERT_EQ(empty, unchanged);
}
TEST(Sorting_a_single_value, leaves_value_unchanged) ...
TEST(Sorting_an_equal_pair, leaves_values_unchanged) ...
TEST(Sorting_an_ascending_pair, leaves_values_unchanged) ...
TEST(Sorting_a_descending_pair, exchanges_values) ...
TEST(Sorting_equal_values, leaves_values_unchanged) ...
TEST(Sorting_ascending_values, leaves_values_unchanged) ...
TEST(Sorting_descending_values, reverses_values) ...
TEST(Sorting_mixed_values, puts_them_into_non_descending_order) ...
TEST(Sorting_random_values, gives_sorted_permutation_of_original_values) ...
TEST(Sorting_random_values, gives_same_result_as_std_sort) ...
```

```
TEST(Sorting_an_empty_range, has_no_effect) ...
TEST(Sorting_a_single_value, leaves_value_unchanged) ...
{
    using limits = std::numeric_limits<int>;
    std::vector single {limits::max(), 42, limits::min()};
    const auto unchanged = single;
    i_cant_believe_it_can_sort(single.begin() + 1, single.end() - 1);
    ASSERT_EQ(single, unchanged);
}
TEST(Sorting_an_equal_pair, leaves_values_unchanged) ...
TEST(Sorting_an_ascending_pair, leaves_values_unchanged) ...
TEST(Sorting_a_descending_pair, exchanges_values) ...
TEST(Sorting_equal_values, leaves_values_unchanged) ...
TEST(Sorting_ascending_values, leaves_values_unchanged) ...
TEST(Sorting_descending_values, reverses_values) ...
TEST(Sorting_mixed_values, puts_them_into_non_descending_order) ...
TEST(Sorting_random_values, gives_sorted_permutation_of_original_values) ...
TEST(Sorting_random_values, gives_same_result_as_std_sort) ...
```

For each usage scenario, the test(s):

- Describe the context, starting point, or preconditions that must be satisfied
- Illustrate how the software is invoked
- Describe the expected results or postconditions to be verified

Gerard Meszaros

```
TEST(Sorting_a_single_value, leaves_value_unchanged)
{
    using limits = std::numeric_limits<int>;
    std::vector single {limits::max(), 42, limits::min()};
    const auto unchanged = single;

    i_cant_believe_it_can_sort(single.begin() + 1, single.end() - 1);

    ASSERT_EQ(single, unchanged);
}
```

```
TEST(Sorting_a_single_value, leaves_value_unchanged)
{
    // Arrange:
    using limits = std::numeric_limits<int>;
    std::vector single {limits::max(), 42, limits::min()};
    const auto unchanged = single;

    // Act:
    i_cant_believe_it_can_sort(single.begin() + 1, single.end() - 1);

    // Assert:
    ASSERT_EQ(single, unchanged);
}
```

```
TEST(Sorting_a_single_value, leaves_value_unchanged)
{
    // Arrange:
    using limits = std::numeric_limits<int>;
    std::vector single {limits::max(), 42, limits::min()};
    const auto unchanged = single;

    // Act:
    i_cant_believe_it_can_sort(single.begin() + 1, single.end() - 1);

    // Assert:
    ASSERT_EQ(single, unchanged);
}
```

```
TEST(Sorting_a_single_value, leaves_value_unchanged)
{
    Arrange:
    using limits = std::numeric_limits<int>;
    std::vector single {limits::max(), 42, limits::min()};
    const auto unchanged = single;

    Act:
    i_cant_believe_it_can_sort(single.begin() + 1, single.end() - 1);

    Assert:
    ASSERT_EQ(single, unchanged);
}
```

```
TEST(Sorting_a_single_value, leaves_value_unchanged)
{
    // Arrange:
    using limits = std::numeric_limits<int>;
    std::vector single {limits::max(), 42, limits::min()};
    const auto unchanged = single;

    // Act:
    i_cant_believe_it_can_sort(single.begin() + 1, single.end() - 1);

    // Assert:
    ASSERT_EQ(single, unchanged);
}
```

```
TEST(Sorting_a_single_value, leaves_value_unchanged)
{
    // Given:
    using limits = std::numeric_limits<int>;
    std::vector single {limits::max(), 42, limits::min()};
    const auto unchanged = single;

    // When:
    i_cant_believe_it_can_sort(single.begin() + 1, single.end() - 1);

    // Then:
    ASSERT_EQ(single, unchanged);
}
```

From time to time I hear people asking what value of test coverage they should aim for, or stating their coverage levels with pride.
Such statements miss the point.

Martin Fowler

*martinfowler.com/bliki/TestCoverage.html*

I expect a high level of coverage.
Sometimes managers require one.
There's a subtle difference.

Brian Marick

*martinfowler.com/bliki/TestCoverage.html*


Adam Jacot de Boinod
I NEVER KNEW THERE WAS A WORD FOR IT
'Very funny'
STEPHEN FRY
WORDSWORTH REFERENCE
The Wordsworth Book of Intriguing Words
The insomniac's dictionary of the outrageous, odd and unusual
Paul Hellweg
REFERENCE
CONCISE OXFORD DICTIONARY OF
English
mology
T. F. HOAD
word origins
the language report
Joie de vivre n.
COMPILED BY
A COMPREHENSIVE WORD-FINDING DICTIONARY
Chambers
BILL
BRYSON
TROUBLESOME WORDS
'Combines the virtues of a first-class work of reference with the pleasure of a good read' The Times
FULLY REVISED AND UPDATED
Oxford
Samuel Johnson's
LOCK
COLLINS
REFERENCE DICTIONARY
MATHEMATICS
E.J. BOROWSKI AND J.M. BORWEIN
JEFFREY KACIRK
AUTHOR OF Forgotten English
WORDS

/ WordFriday

# **Goodhart's law**, *noun*

- Once a metric becomes a target, it loses its meaning as a measure.
- Named after Charles Goodhart, professor of economics at the LSE and former advisor to the Bank of England, who in 1975 observed that "Any observed statistical regularity will tend to collapse once pressure is placed upon it for control purposes."

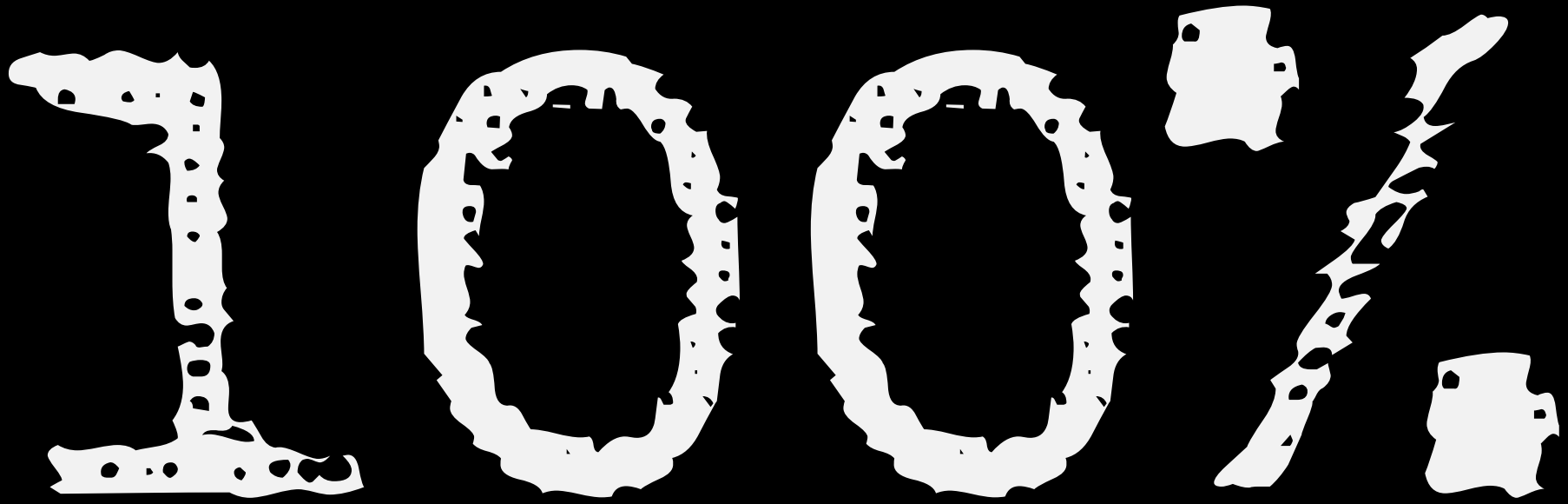
*facebook.com/WordFriday/posts/320731068014857*

# coverage

# statement coverage

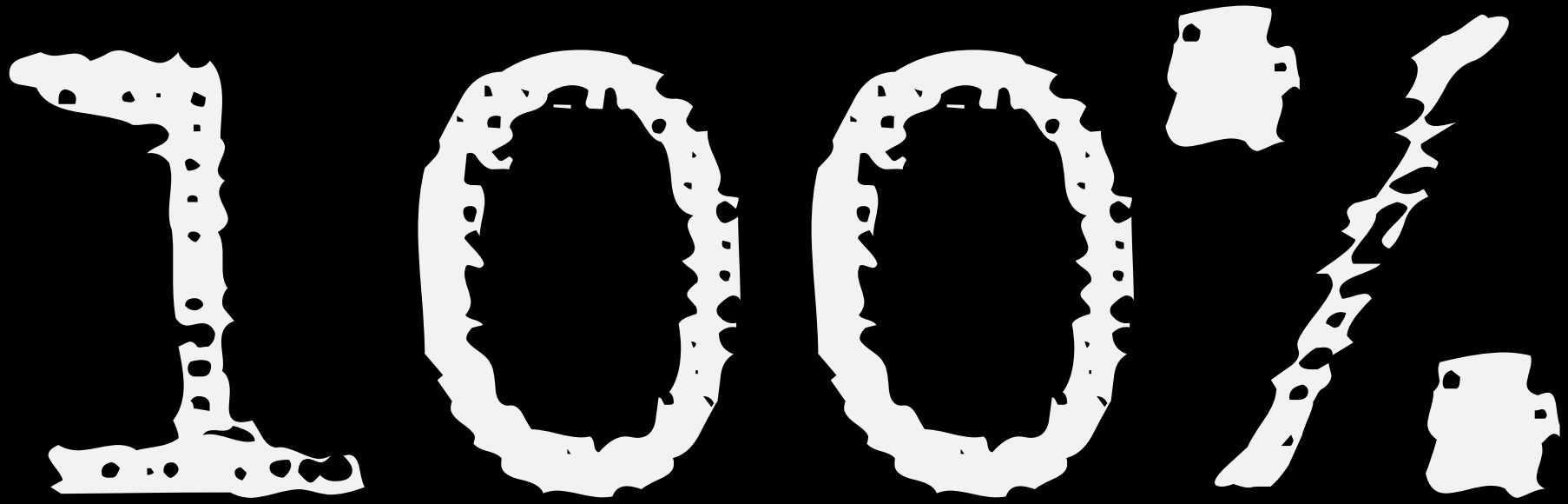
```
TEST(Sorting_an_empty_range, has_no_effect) ...
TEST(Sorting_a_single_value, leaves_value_unchanged) ...
TEST(Sorting_an_equal_pair, leaves_values_unchanged) ...
TEST(Sorting_an_ascending_pair, leaves_values_unchanged) ...
TEST(Sorting_a_descending_pair, exchanges_values) ...
TEST(Sorting_equal_values, leaves_values_unchanged) ...
TEST(Sorting_ascending_values, leaves_values_unchanged) ...
TEST(Sorting_descending_values, reverses_values) ...
TEST(Sorting_mixed_values, puts_them_into_non_descending_order)
{
    std::forward_list mixed {3, 1, 4, 1, 5, 9};
    const std::forward_list expected {1, 1, 3, 4, 5, 9};

    i_cant_believe_it_can_sort(mixed.begin(), mixed.end());

    ASSERT_EQ(mixed, expected);
}
TEST(Sorting_random_values, gives_sorted_permutation_of_original_values) ...
TEST(Sorting_random_values, gives_same_result_as_std_sort) ...
```

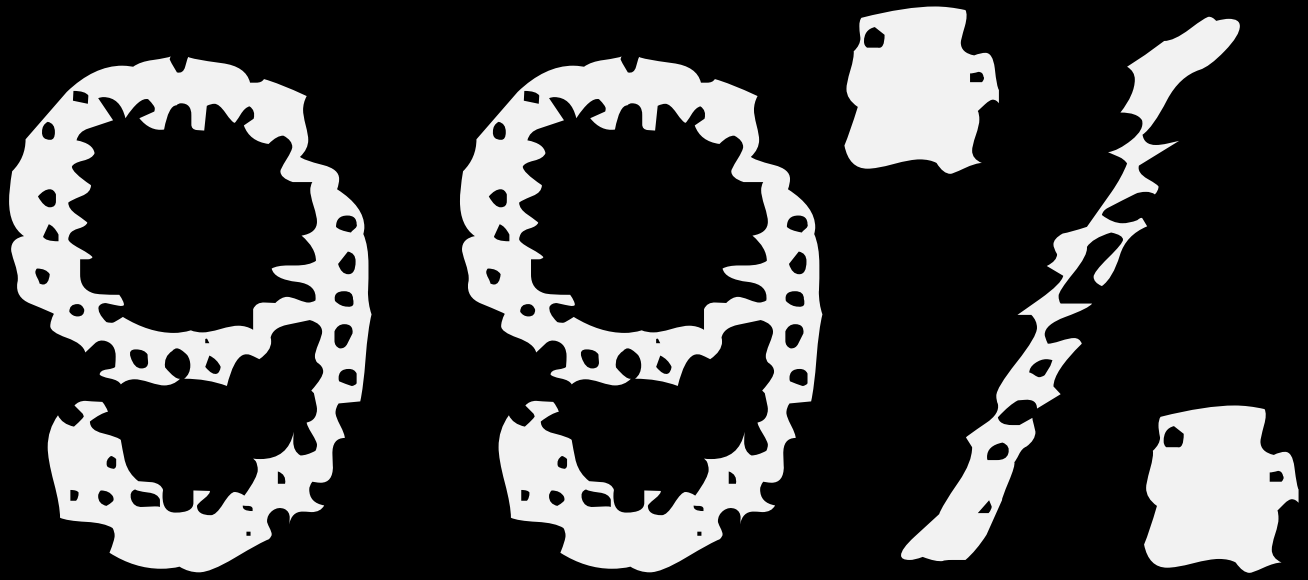
```
TEST(Sorting_mixed_values, puts_them_into_non_descending_order)
{
    std::forward_list mixed {3, 1, 4, 1, 5, 9};
    const std::forward_list expected {1, 1, 3, 4, 5, 9};

    i_cant_believe_it_can_sort(mixed.begin(), mixed.end());

    ASSERT_EQ(mixed, expected);
}
```


100%.


DOJO

function coverage
statement coverage
branch coverage
loop coverage
condition coverage
multiple condition cover
path coverage
parameter value covera


100%



*What suggestions do you have for dealing with dead code?*

Find it. Delete it.

Deleting dead code is not a technical problem; it is a problem of mindset and culture.

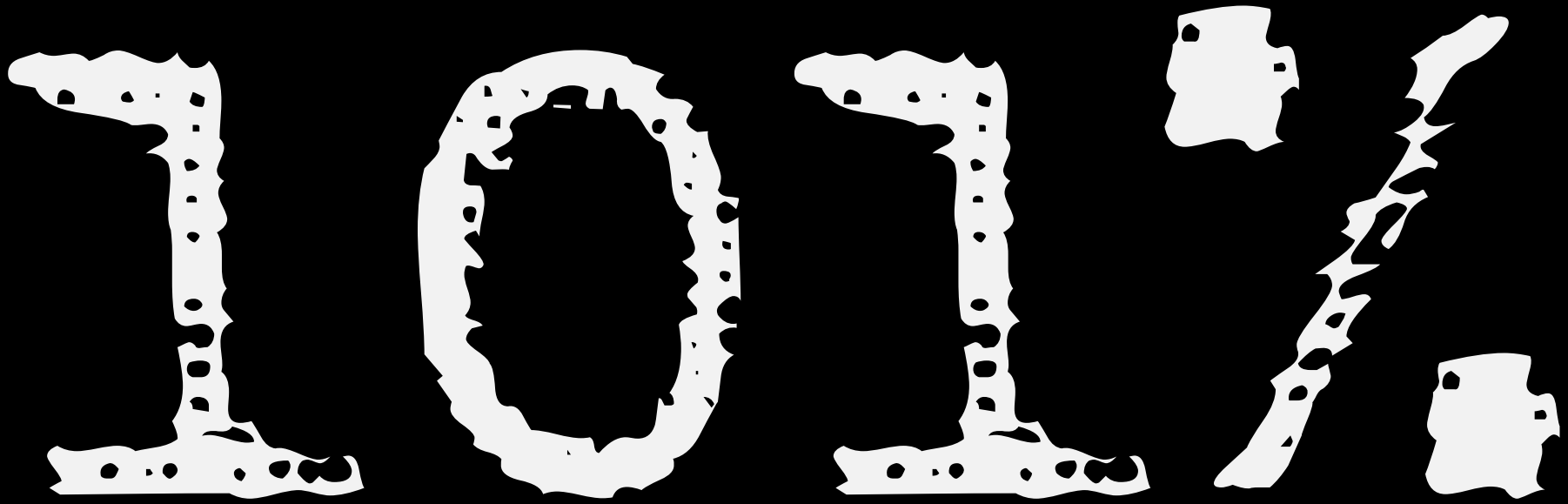
Kevlin Henney

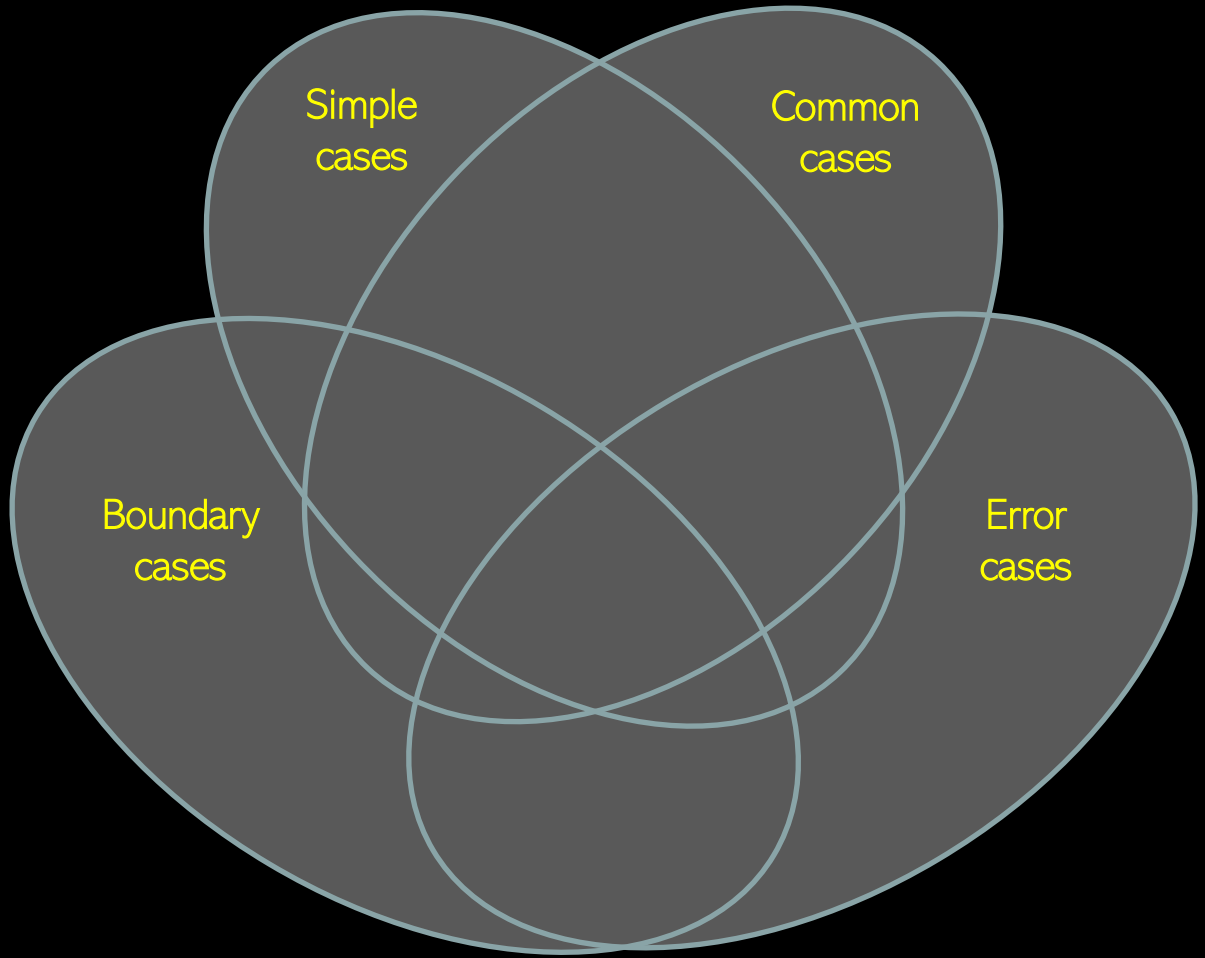
*infoq.com/news/2017/02/dead-code*

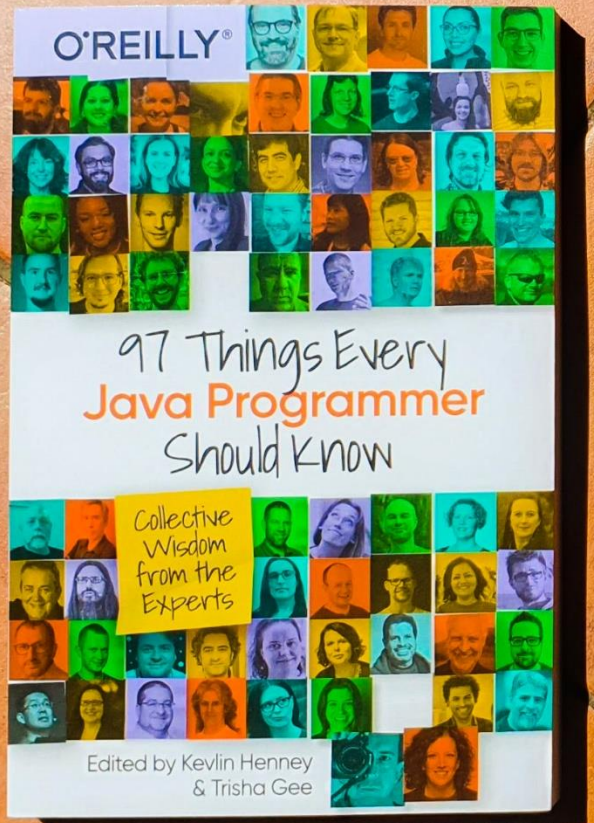
We instituted a rigorous regression test for all of the features of AWK. Any of the three of us who put in a new feature into the language ... first had to write a test for the new feature.

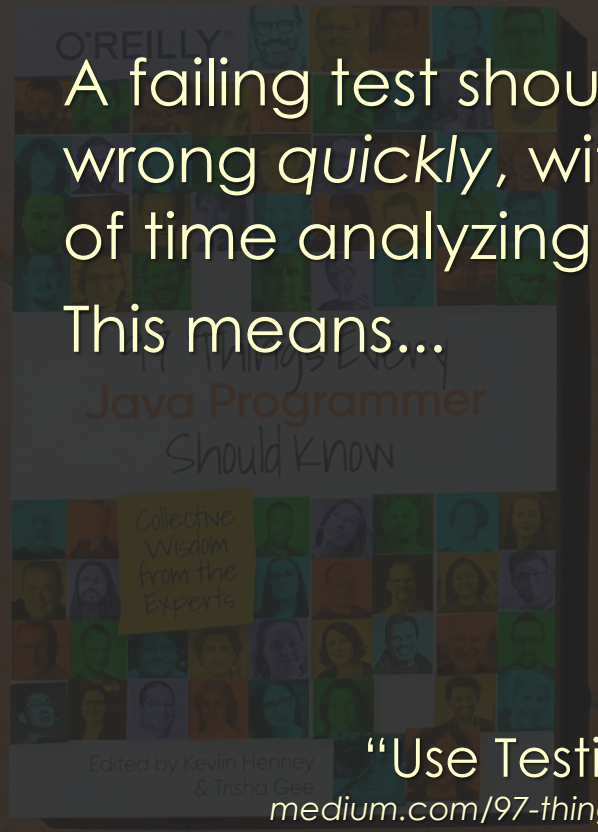
Alfred Aho

*computerworld.com.au/article/216844/a-z_programming_languages_awk*


101%


Simple cases
Common cases
Boundary cases
Error cases


O'REILLY®
97 Things Every
Java Programmer
Should Know
Collective Wisdom from the Experts
Edited by Kevlin Henney
& Trisha Gee




A failing test should tell you *exactly* what is wrong *quickly*, without you having to spend a lot of time analyzing the failure.

This means...

Marit van Dijk

"Use Testing to Develop Better Software Faster"

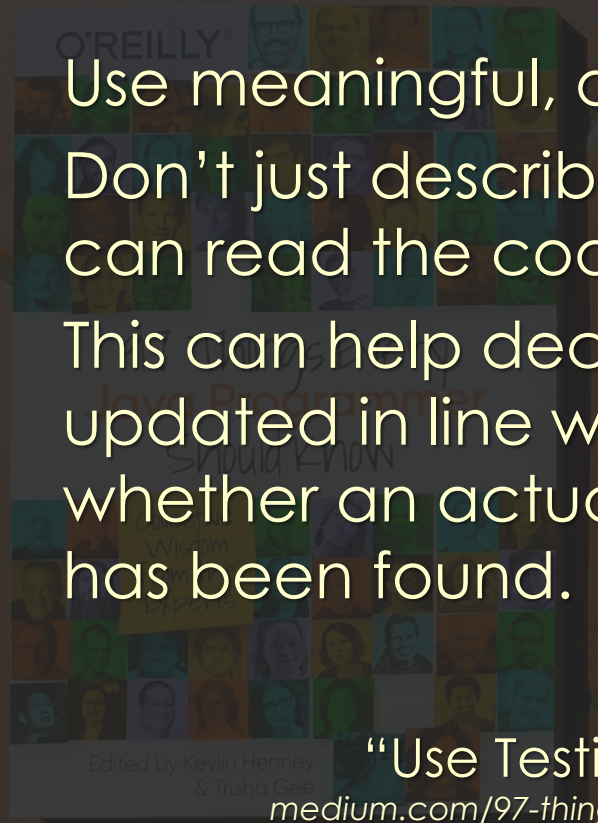
*medium.com/97-things/use-testing-to-develop-better-software-faster-9dd2616543d3*

# Each test should test one thing.


Marit van Dijk

"Use Testing to Develop Better Software Faster"

*medium.com/97-things/use-testing-to-develop-better-software-faster-9dd2616543d3*




Use meaningful, descriptive names.

Don't just describe what the test does either (we can read the code), tell us why it does this.

This can help decide whether a test should be updated in line with changed functionality or whether an actual failure that should be fixed has been found.

Marit van Dijk

"Use Testing to Develop Better Software Faster"

*medium.com/97-things/use-testing-to-develop-better-software-faster-9dd2616543d3*

# Never trust a test you haven't seen fail.


Marit van Dijk

"Use Testing to Develop Better Software Faster"

*medium.com/97-things/use-testing-to-develop-better-software-faster-9dd2616543d3*

```
TEST(Sorting_an_empty_range, has_no_effect) ...
TEST(Sorting_a_single_value, leaves_value_unchanged) ...
TEST(Sorting_an_equal_pair, leaves_values_unchanged) ...
TEST(Sorting_an_ascending_pair, leaves_values_unchanged) ...
TEST(Sorting_a_descending_pair, exchanges_values) ...
TEST(Sorting_equal_values, leaves_values_unchanged) ...
TEST(Sorting_ascending_values, leaves_values_unchanged) ...
TEST(Sorting_descending_values, reverses_values) ...
TEST(Sorting_mixed_values, puts_them_into_non_descending_order) ...
TEST(Sorting_random_values, gives_sorted_permutation_of_original_values) ...
TEST(Sorting_random_values, gives_same_result_as_std_sort) ...
```

```
template<std::forward_iterator iterator>
void i_cant_believe_it_can_sort(iterator begin, iterator end)
{
    for (auto i = begin; i != end; ++i)
        for (auto j = begin; j != end; ++j)
            if (*i < *j)
                std::iter_swap(i, j);
}
```

```
template<std::forward_iterator iterator>
void i_cant_believe_it_can_sort(iterator begin, iterator end)
{
    for (auto i = begin; i != end; ++i)
        for (auto j = begin; j != end; ++j)
            if (*i > *j)
                std::iter_swap(i, j);
}
```

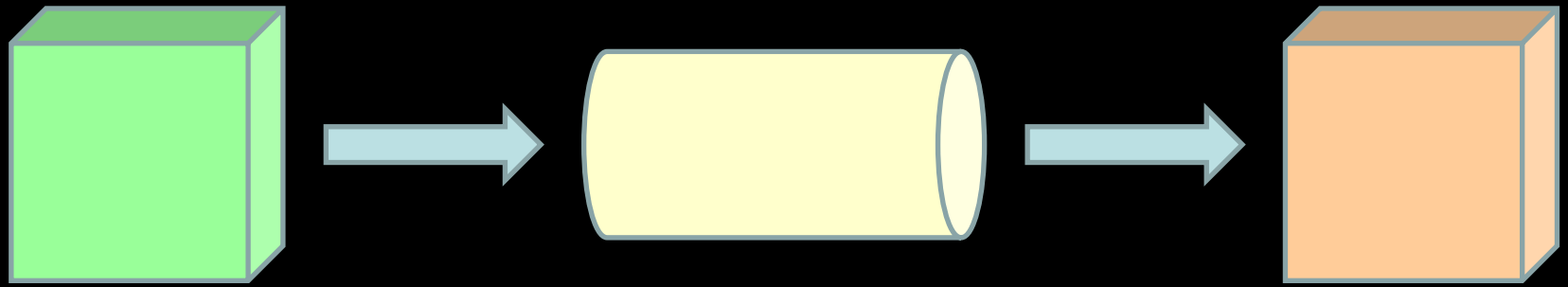
```
TEST(Sorting_an_empty_range, has_no_effect) ...
TEST(Sorting_a_single_value, leaves_value_unchanged) ...
TEST(Sorting_an_equal_pair, leaves_values_unchanged) ...
TEST(Sorting_an_ascending_pair, leaves_values_unchanged) ...
TEST(Sorting_a_descending_pair, exchanges_values) ...
TEST(Sorting_equal_values, leaves_values_unchanged) ...
TEST(Sorting_ascending_values, leaves_values_unchanged) ...
TEST(Sorting_descending_values, reverses_values) ...
TEST(Sorting_mixed_values, puts_them_into_non_descending_order) ...
TEST(Sorting_random_values, gives_sorted_permutation_of_original_values) ...
TEST(Sorting_random_values, gives_same_result_as_std_sort) ...
```

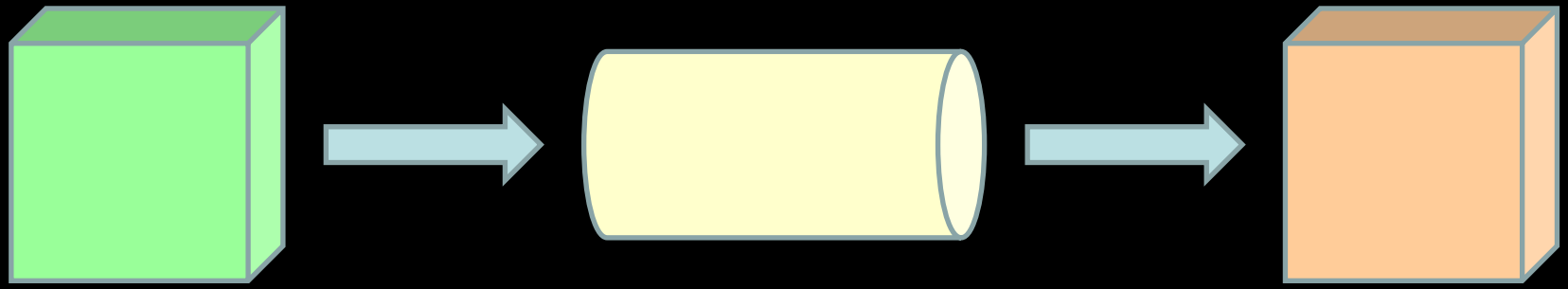
An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects.

Barbara Liskov

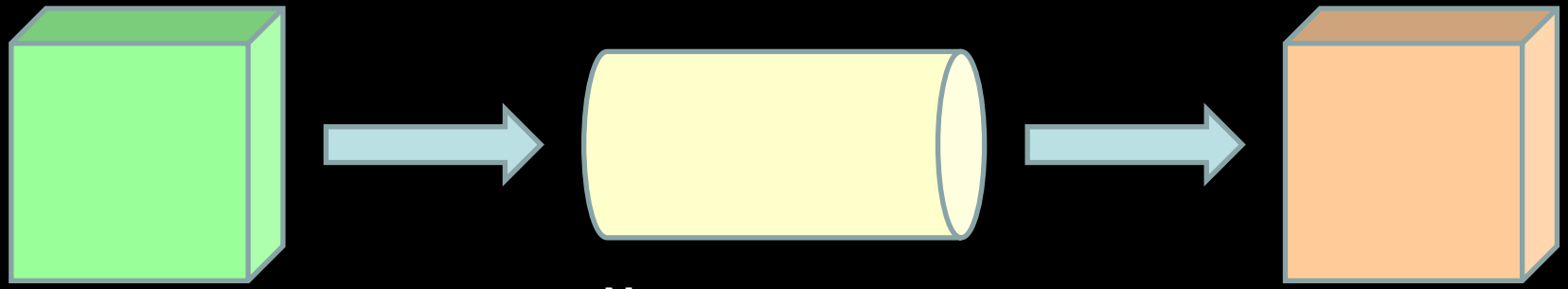
*Programming with Abstract Data Types*

```
template<typename value_type>
class queue
{
public:
    ...
    std::size_t length() const;
    std::size_t capacity() const;
    bool enqueue(const value_type &);
    std::optional<value_type> dequeue();
private:
    ...
};
```

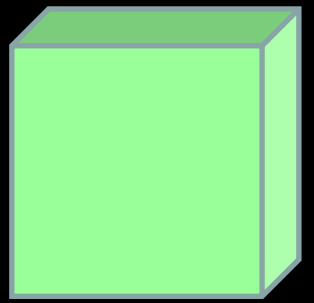

producer
queue
consumer

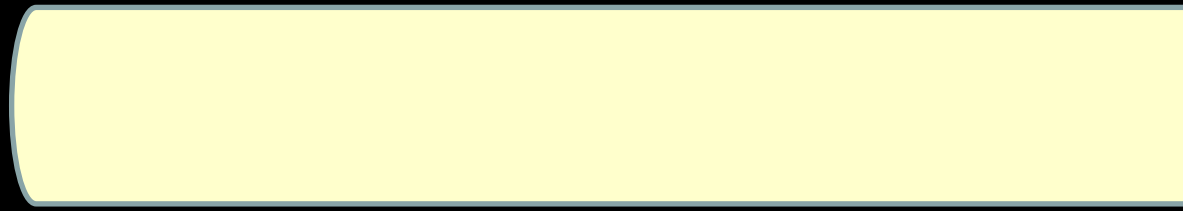

producer
spacetime decoupling
consumer

```
template<typename value_type>
class queue
{
public:
    ...
    ~queue() = default;
    queue(const queue &) = delete;
    queue & operator=(const queue &) = delete;

    std::size_t length() const;
    std::size_t capacity() const;

    bool enqueue(const value_type &);
    std::optional<value_type> dequeue();
private:
    ...
};
```

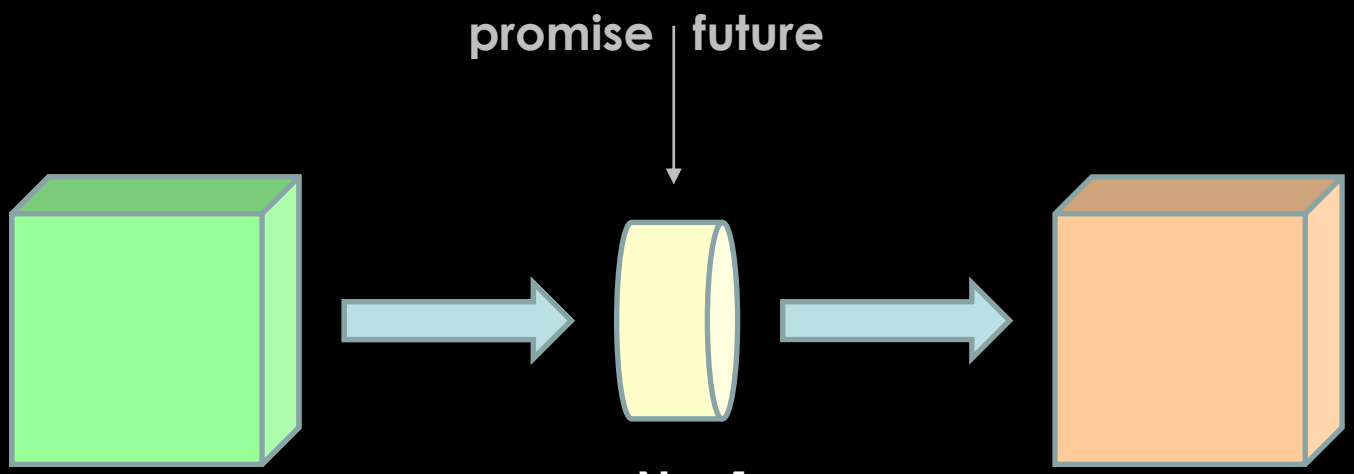


**N**
buffered
bounded
asynchronous

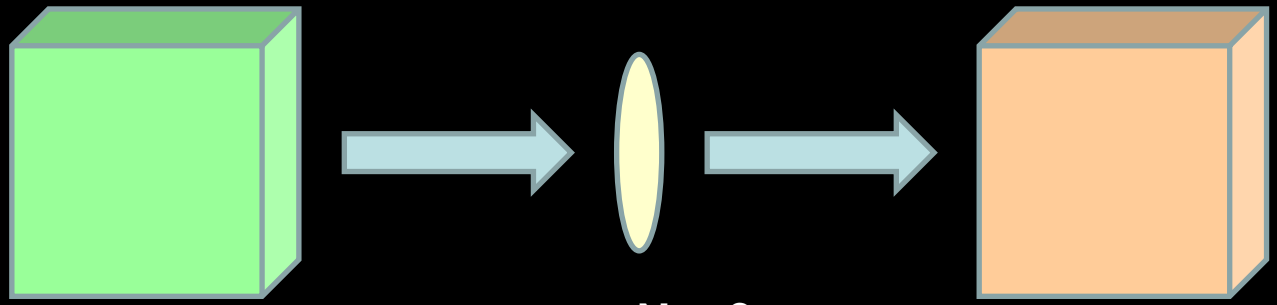




**N = ∞**
buffered
unbounded
asynchronous




**N = 1**
buffered
bounded
asynchronous



**N = 0**
unbuffered
bounded
synchronous

```
template<typename value_type>
class queue
{
public:
    explicit queue(const std::size_t capacity);
    ~queue() = default;
    queue(const queue &) = delete;
    queue & operator=(const queue &) = delete;

    std::size_t length() const;
    std::size_t capacity() const;

    bool enqueue(const value_type &);
    std::optional<value_type> dequeue();
private:
    ...
};
```

A programmer ... is concerned only with the behavior which that object exhibits but not with any details of how that behavior is achieved by means of an implementation.

Barbara Liskov

*Programming with Abstract Data Types*

```
template<typename value_type>
class queue
{
public:
    explicit queue(const std::size_t capacity)
    : max_length(capacity)
    {
        if (capacity == 0)
            throw std::invalid_argument("queue cannot have zero capacity");
    }

    ~queue() = default;
    queue(const queue &) = delete;
    queue & operator=(const queue &) = delete;

    std::size_t length() const
    {
        return items.size();
    }
    std::size_t capacity() const
    {
        return max_length;
    }
    bool enqueue(const value_type & to_enqueue)
    {
        bool enqueuing = length() < capacity();

        if (enqueuing)
            items.push_back(to_enqueue);

        return enqueuing;
    }
    std::optional<value_type> dequeue()
    {
        std::optional dequeued {length() == 0 ? std::nullopt : std::optional {items.front()}};

        if (dequeued)
            items.pop_front();

        return dequeued;
    }
private:
    std::deque<value_type> items;
    const std::size_t max_length;
};
```

```
template<typename value_type>
class queue
{
public:
    explicit queue(const std::size_t capacity);
    ~queue() = default;
    queue(const queue &) = delete;
    queue & operator=(const queue &) = delete;

    std::size_t length() const;
    std::size_t capacity() const;

    bool enqueue(const value_type &);
    std::optional<value_type> dequeue();
private:
    std::deque<value_type> items;
    const std::size_t max_length;
};
```

```
template<typename value_type>
class queue
{
public:
    explicit queue(const std::size_t capacity);
    ~queue() = default;
    queue(const queue &) = delete;
    queue & operator=(const queue &) = delete;

    std::size_t length() const;
    std::size_t capacity() const;

    bool enqueue(const value_type &);
    std::optional<value_type> dequeue();
private:
    std::list<value_type> items;
    const std::size_t max_length;
};
```

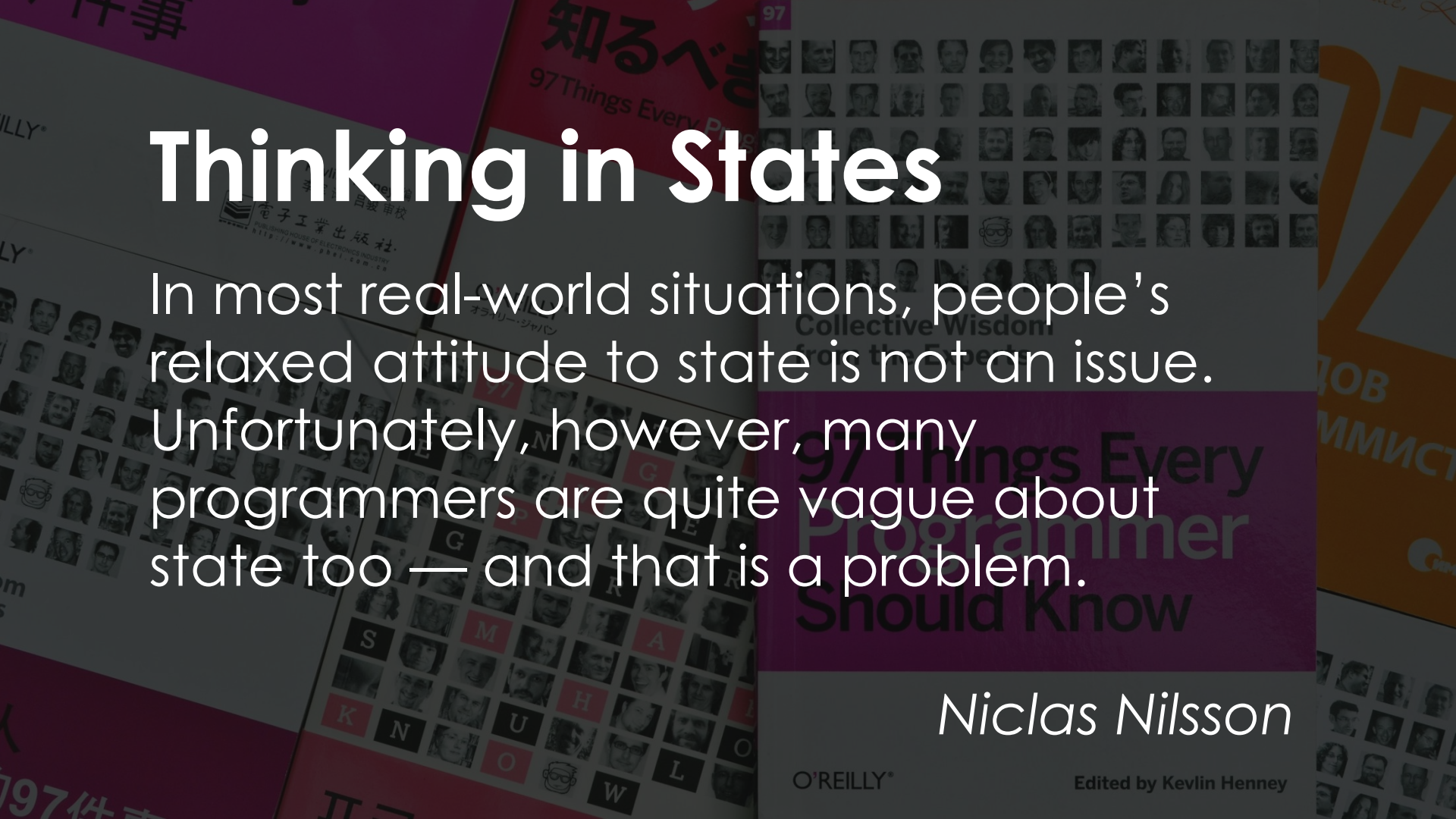

No public
access

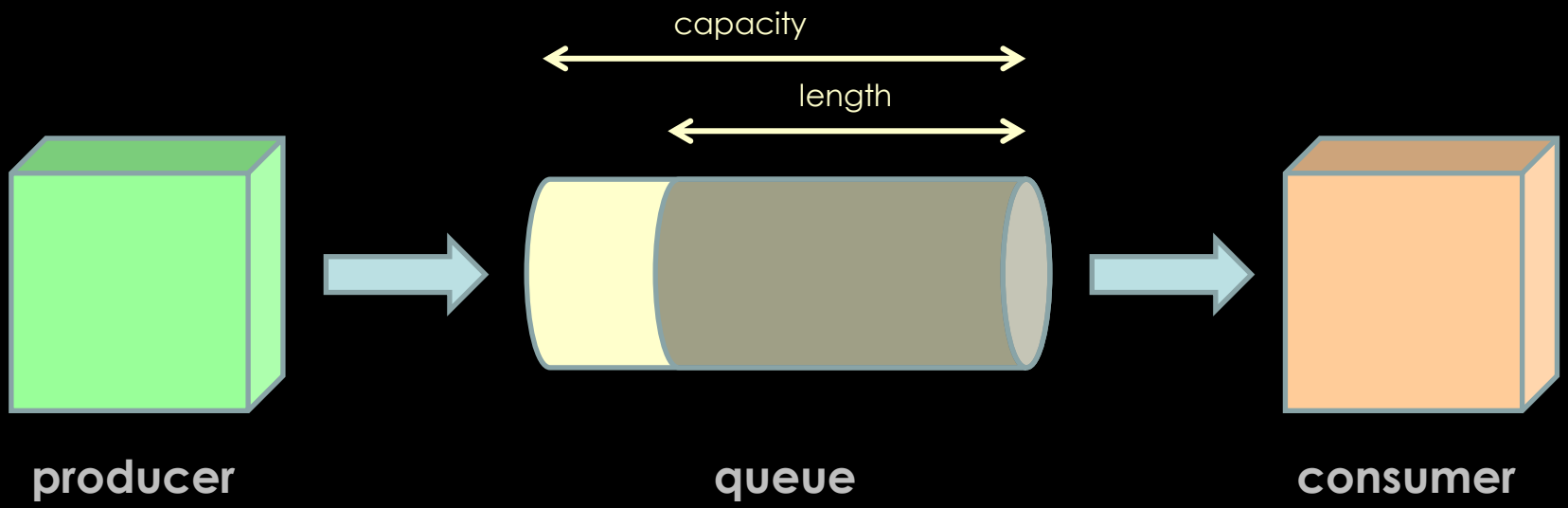
```
SCENARIO("Queue specification")
{
    SECTION("constructor") ...
    SECTION("length") ...
    SECTION("capacity") ...
    SECTION("enqueue") ...
    SECTION("dequeue") ...
}
```

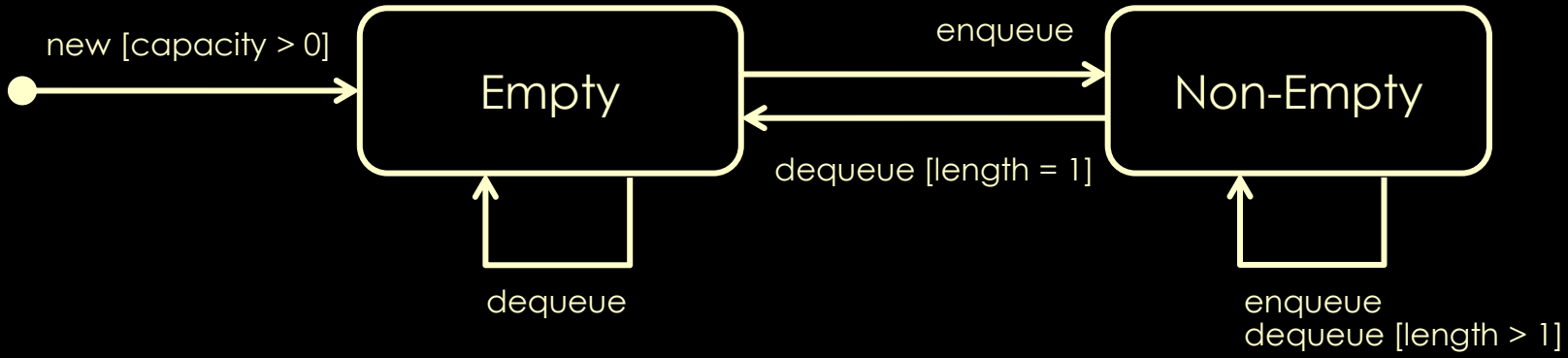
# Thinking in States

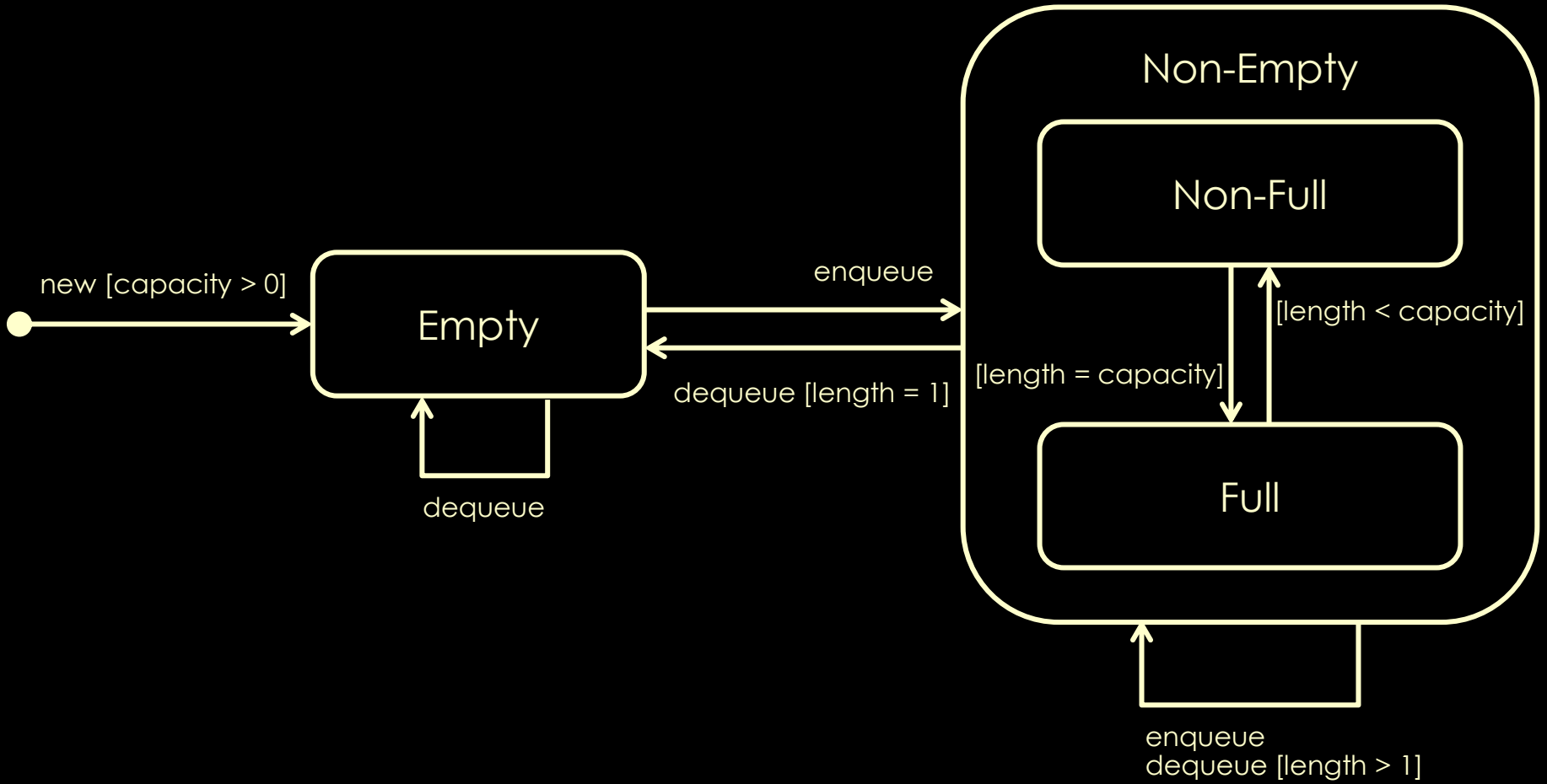
In most real-world situations, people's relaxed attitude to state is not an issue. Unfortunately, however, many programmers are quite vague about state too — and that is a problem.

*Niclas Nilsson*




capacity
length
producer
queue
consumer


new [capacity > 0]
Empty
enqueue
dequeue [length = 1]
Non-Empty
dequeue
enqueue
dequeue [length > 1]


Non-Empty
Non-Full
Empty
Full
new [capacity > 0]
enqueue
dequeue [length = 1]
[length = capacity]
[length < capacity]
dequeue
enqueue
dequeue [length > 1]

```
SCENARIO("Queue specification") ...
    SECTION("A new queue") ...
        SECTION("is empty") ...
        SECTION("preserves positive bounding capacity") ...
        SECTION("cannot be created with zero bounding capacity") ...
    SECTION("An empty queue") ...
        SECTION("dequeues an empty value") ...
        SECTION("becomes non-empty when value enqueued") ...
    SECTION("A non-empty queue") ...
        SECTION("that is not full") ...
            SECTION("becomes longer when value enqueued") ...
            SECTION("becomes full when enqueued up to capacity") ...
        SECTION("that is full") ...
            SECTION("ignores further enqueued values") ...
            SECTION("becomes non-full when dequeued") ...
        SECTION("dequeues values in order enqueued") ...
```

```
SCENARIO("Queue specification") ...
    SECTION("A new queue") ...
        SECTION("is empty") ...
        SECTION("preserves positive bounding capacity") ...
        SECTION("cannot be created with zero bounding capacity") ...
    SECTION("An empty queue") ...
        SECTION("dequeues an empty value") ...
        SECTION("becomes non-empty when value enqueued") ...
    SECTION("A non-empty queue") ...
        SECTION("that is not full") ...
            SECTION("becomes longer when value enqueued") ...
            SECTION("becomes full when enqueued up to capacity") ...
        SECTION("that is full") ...
            SECTION("ignores further enqueued values") ...
            SECTION("becomes non-full when dequeued") ...
        SECTION("dequeues values in order enqueued") ...
```

Given

When

Then

# Given

# When

# Then

**Given** can be used to group tests for operations with respect to common initial state

Given

When

Then

**When** can be used to group tests by operation, regardless of initial state or outcome

# Given

# When

# Then

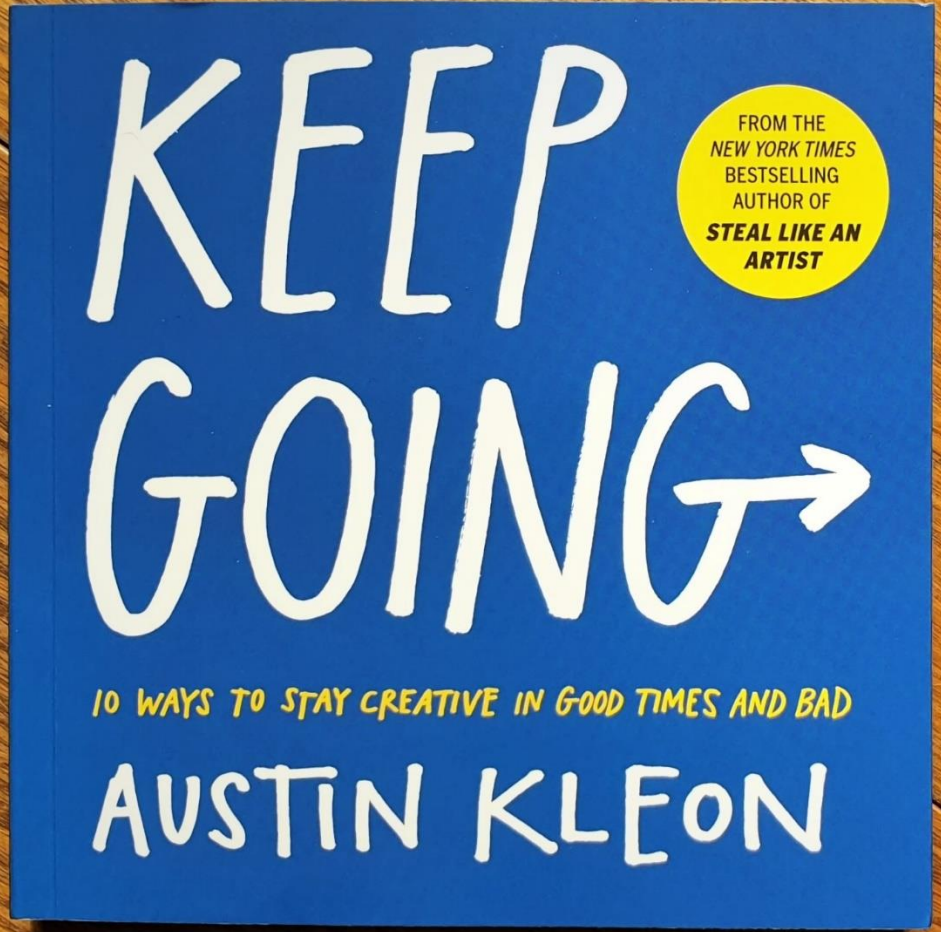
**Then** can be used to group tests by common outcome, regardless of operation or initial state

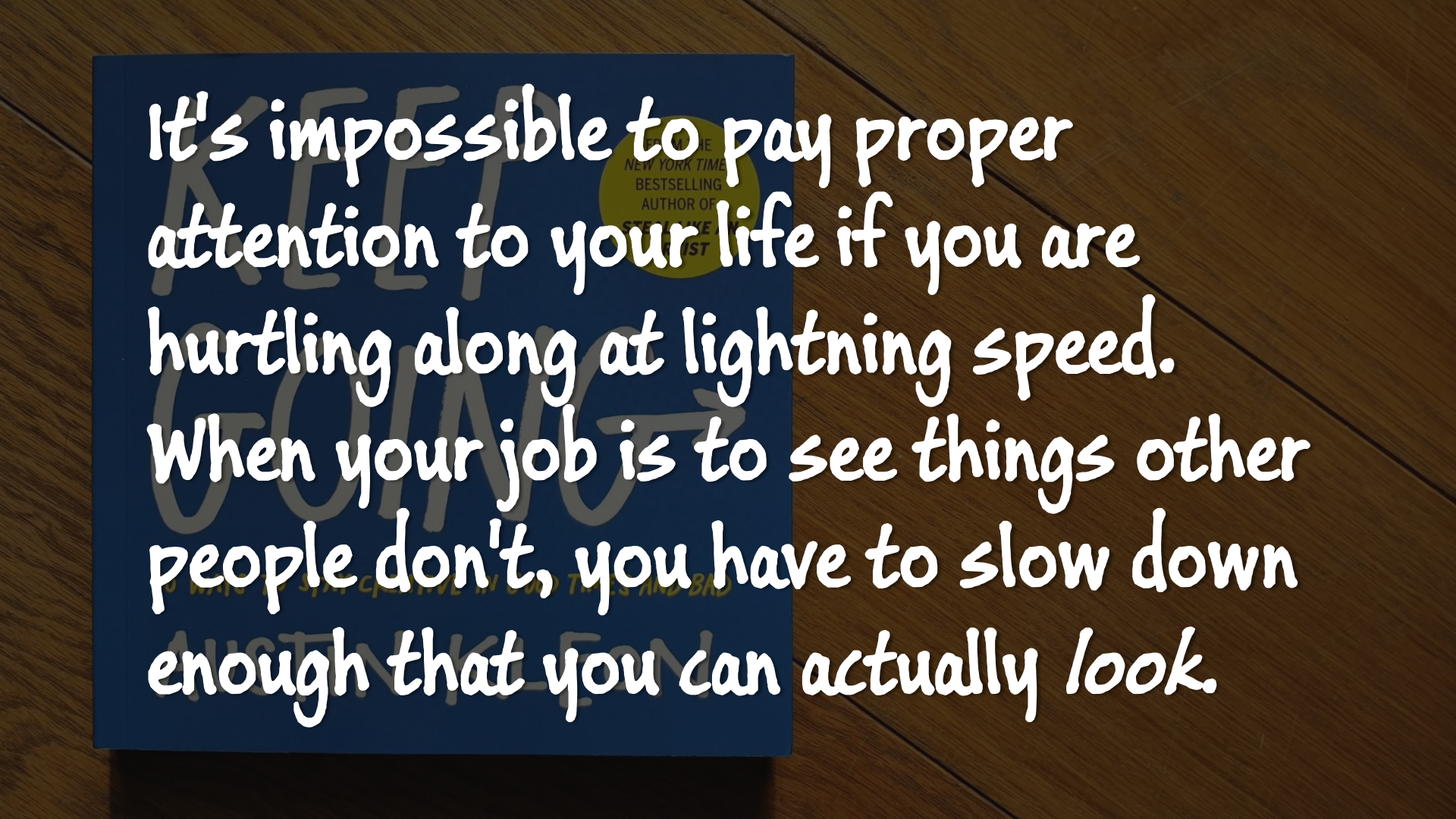
```
SCENARIO("Queue specification")
{
    ...
    SECTION("An empty queue")
    {
        queue<int> a_queue(3);
        ...
        SECTION("becomes non-empty when value enqueued")
        {
            bool enqueued = a_queue.enqueue(42);
            REQUIRE(enqueued);
            REQUIRE(a_queue.length() == 1);
        }
    }
    ...
}
```

```
SCENARIO("Queue specification")
{
    ...
    GIVEN("An empty queue")
    {
        queue<int> a_queue(3);
        ...
        WHEN("a value is enqueued")
        {
            bool enqueued = a_queue.enqueue(42);
            THEN("it becomes non-empty")
            {
                REQUIRE(enqueued);
                REQUIRE(a_queue.length() == 1);
            }
        }
    }
    ...
}
```

```
template<typename value_type>
class queue
{
public:
    struct options
    {
        std::function<void()> on_empty = []{};
        std::function<void(const value_type &) on_full = [](auto){};
        std::size_t capacity = std::numeric_limits<std::size_t>::max();
    };
    explicit queue(const options &);
    ...
private:
    std::deque<value_type> items;
    const options limits;
};
```

```
SCENARIO("Queue specification")
{
    ...
    SECTION("An empty queue")
    {
        int notified = 0;
        queue<int> a_queue(
        {
            .on_empty = [&]
            {
                ++notified;
            }
        });
        ...
        SECTION("notifies by callback on dequeuing")
        {
            a_queue.dequeue();
            REQUIRE(notified == 1);
        }
    }
    ...
}
```


KEEP GOING
FROM THE NEW YORK TIMES BESTSELLING AUTHOR OF STEAL LIKE AN ARTIST
10 WAYS TO STAY CREATIVE IN GOOD TIMES AND BAD
AUSTIN KLEON


It's impossible to pay proper attention to your life if you are hurtling along at lightning speed. When your job is to see things other people don't, you have to slow down enough that you can actually look.