Structure and Interpretation of Test Cases

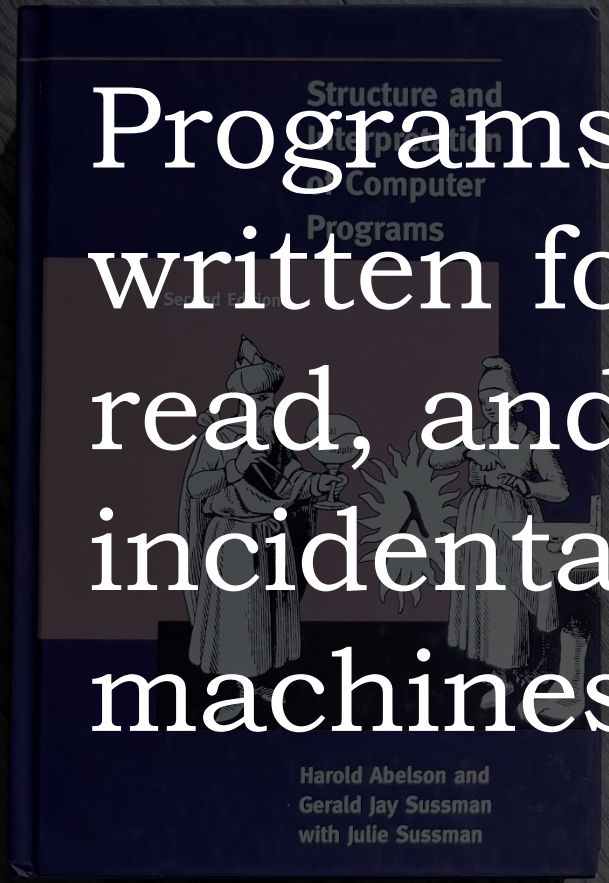
@KevlinHenney

Structure and Interpretation of Computer Programs

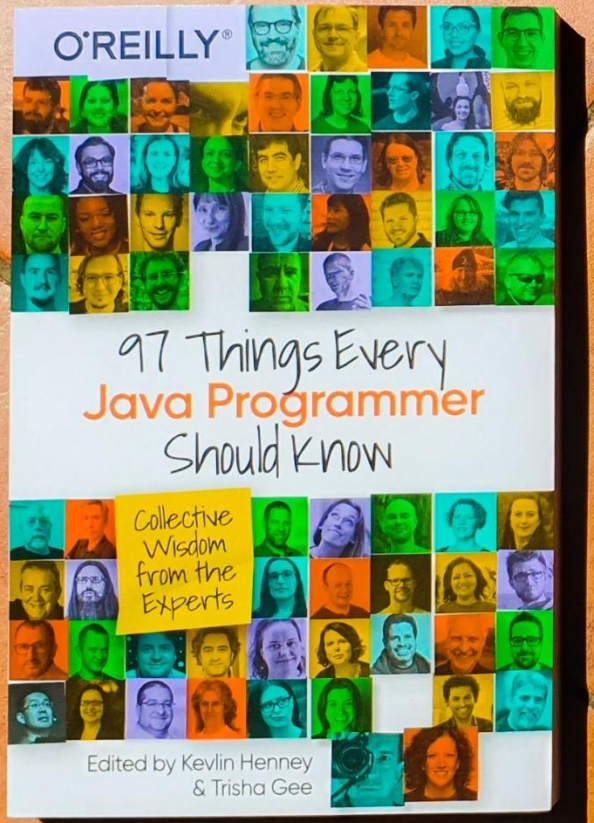
Second Edition

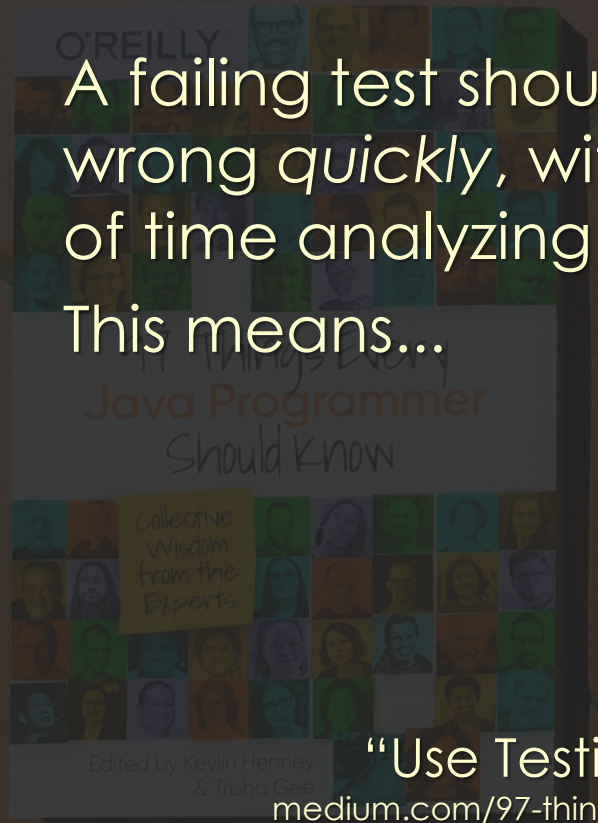


Harold Abelson and
Gerald Jay Sussman
with Julie Sussman



Programs must be written for people to read, and only incidentally for machines to execute.


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

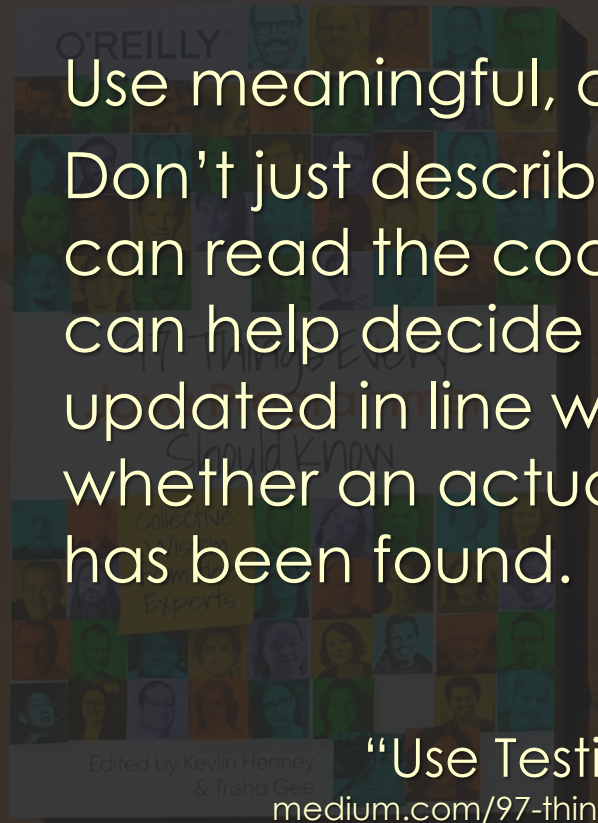
Each test should test one thing.



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

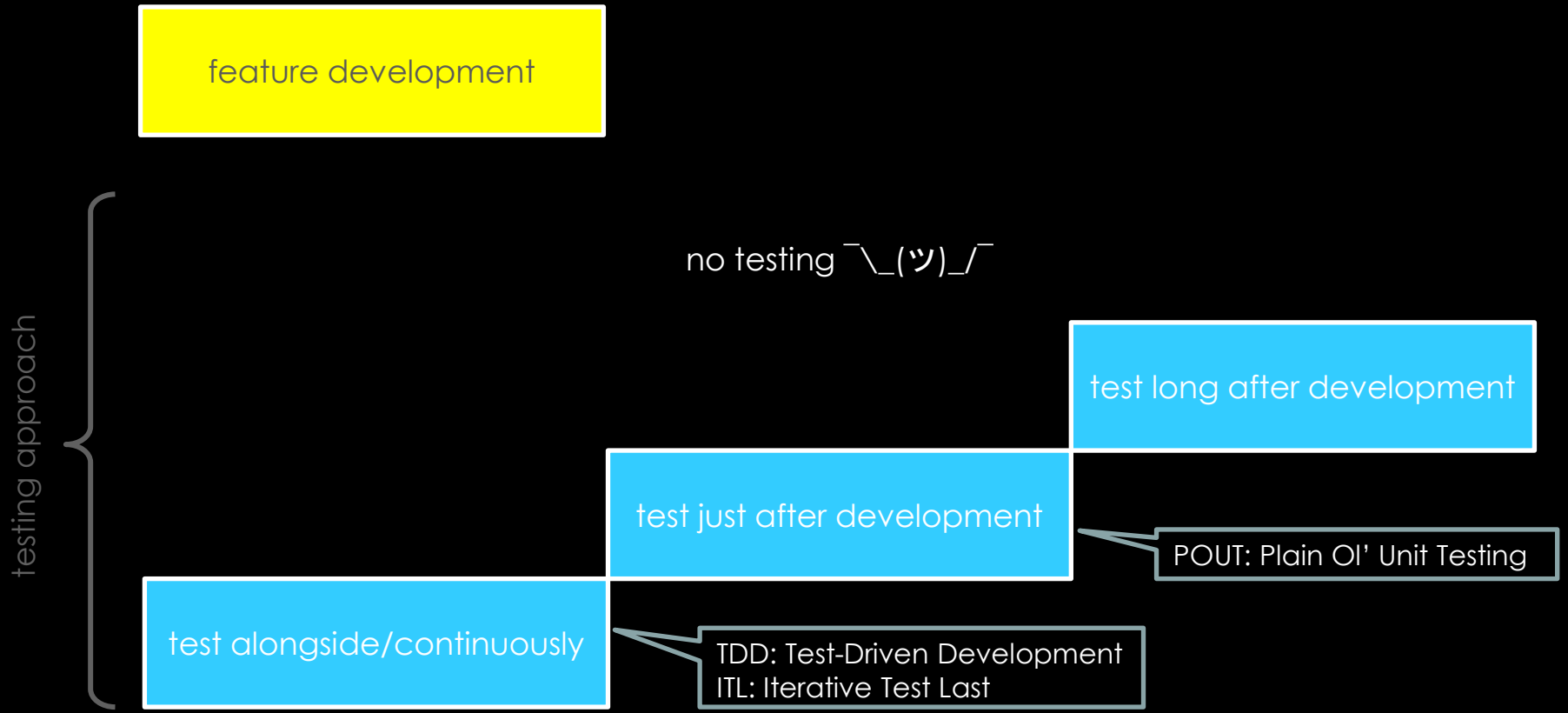
Never trust a test you haven't seen fail.



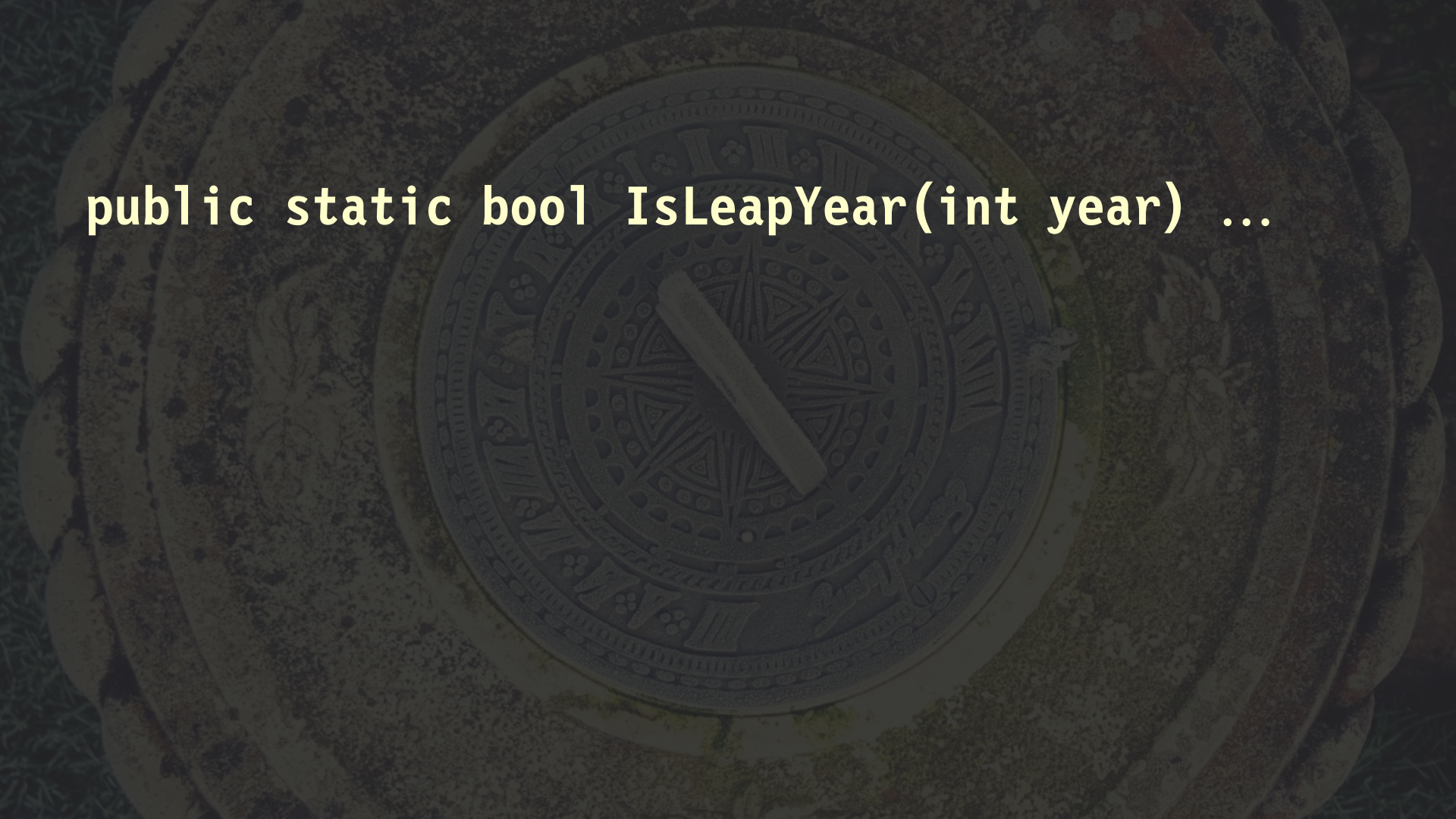
Marit van Dijk

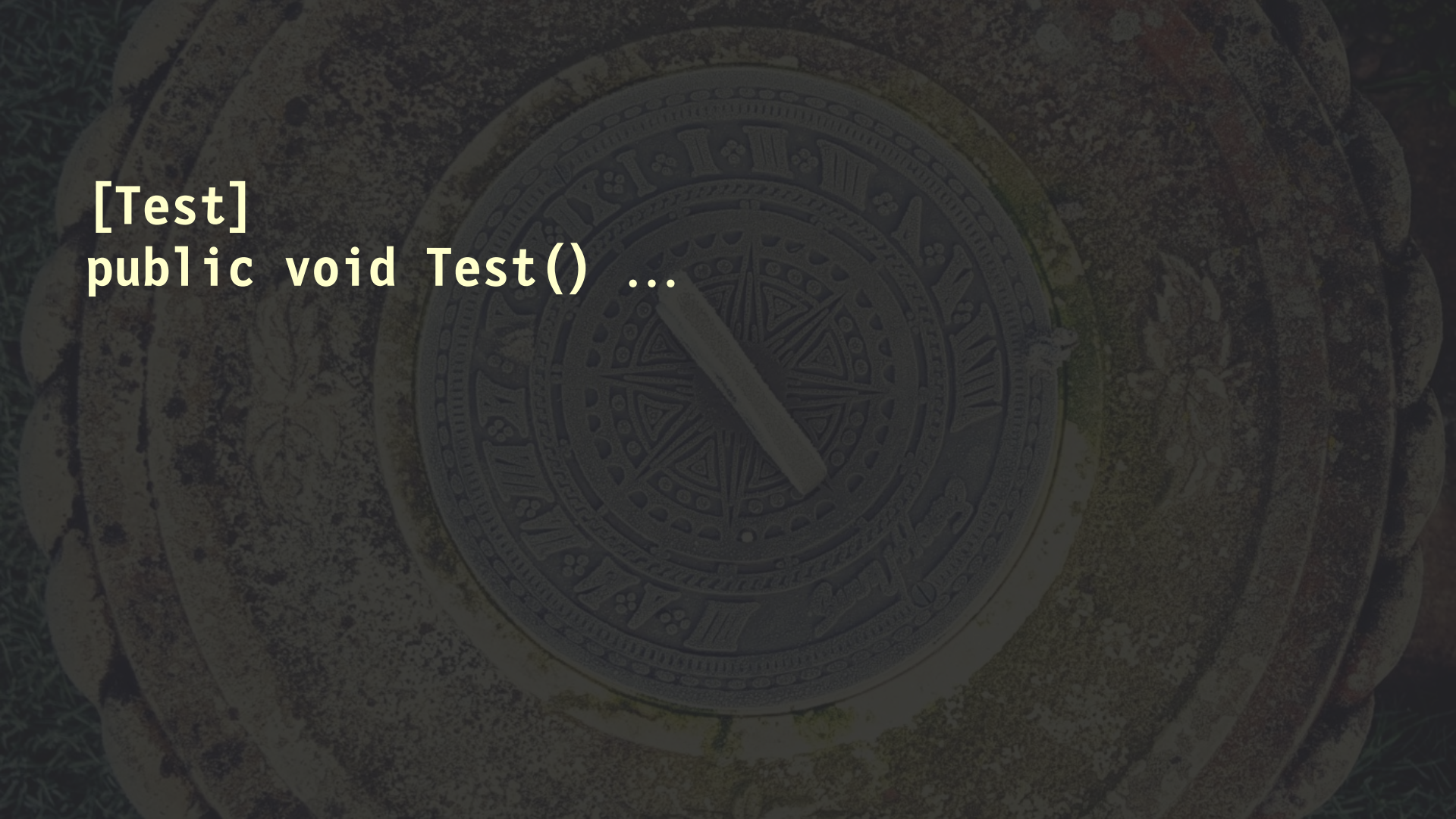
"Use Testing to Develop Better Software Faster"

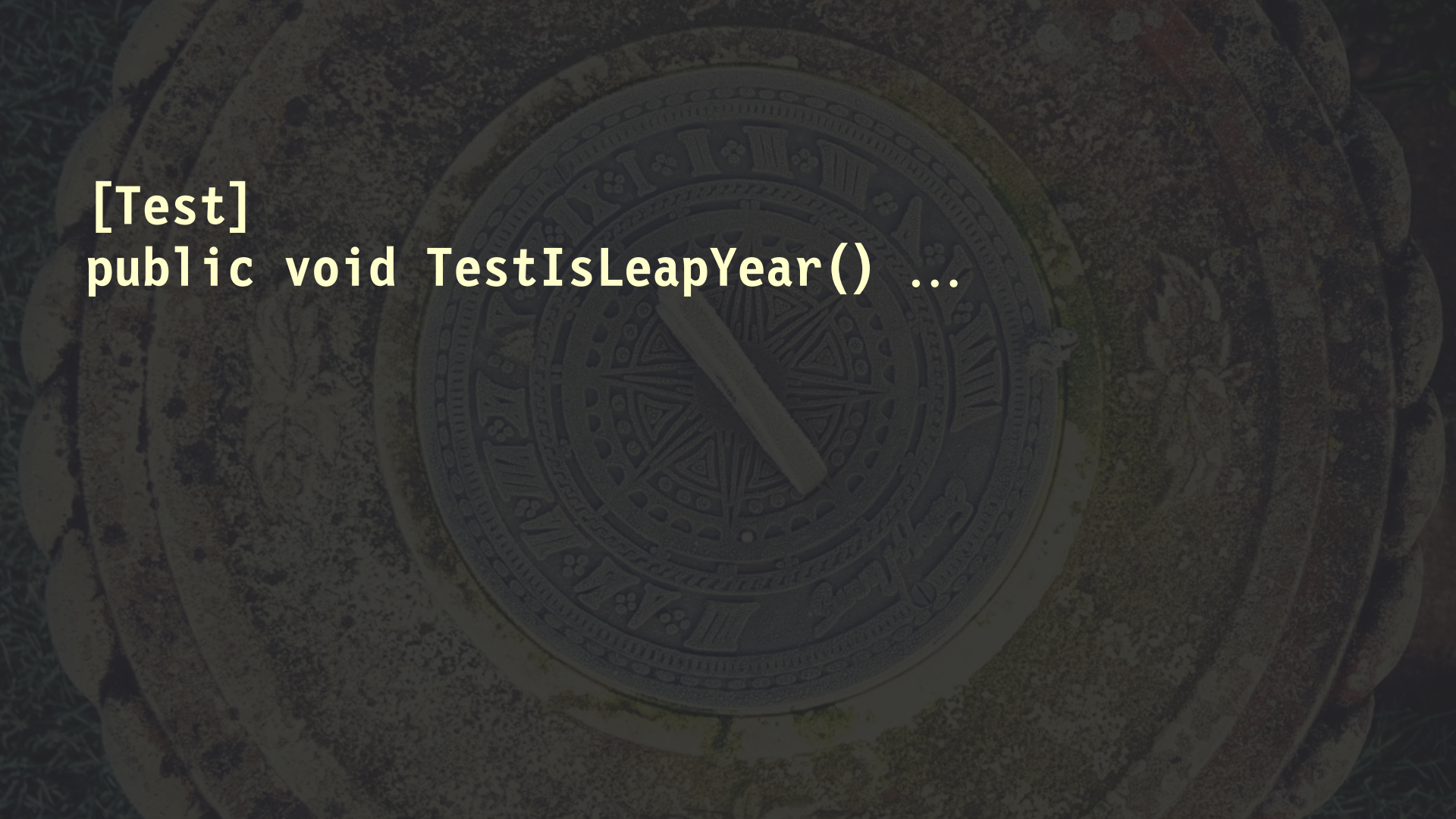
medium.com/97-things/use-testing-to-develop-better-software-faster-9dd2616543d3


feature development
testing approach
no testing ¯_(ツ)_/¯
test long after development
test just after development
POUT: Plain Ol' Unit Testing
test alongside/continuously
TDD: Test-Driven Development
ITL: Iterative Test Last




public static bool IsLeapYear(int year) ...


[Test]
public void Test() ...


[Test]
public void TestIsLeapYear() ...

```
[Test]
public void TestIsLeapYearIsCorrect() ...
```

```
[Test]
public void Test1() ...
[Test]
public void Test2() ...
```

```
[Test]
public void TestLeapYears() ...
[Test]
public void TestNonLeapYears() ...
```

```
[Test]
public void TestLeapYears()
{
    Assert.IsTrue(IsLeapYear(2020));
    Assert.IsTrue(IsLeapYear(2000));
}
[Test]
public void TestNonLeapYears() ...
```

```
[Test]
public void Test2020IsALeapYear() ...
[Test]
public void Test2000IsALeapYear() ...
[Test]
public void Test2022IsNotALeapYear() ...
[Test]
public void Test1900IsNotALeapYear() ...
```

```
[Test]
public void _2020IsALeapYear() ...
[Test]
public void _2000IsALeapYear() ...
[Test]
public void _2022IsNotALeapYear() ...
[Test]
public void _1900IsNotALeapYear() ...
```

```
[Test]
public void _2020_is_a_leap_year() ...
[Test]
public void _2000_is_a_leap_year() ...
[Test]
public void _2022_is_not_a_leap_year() ...
[Test]
public void _1900_is_not_a_leap_year() ...
```

```
_2020_is_a_leap_year

_2000_is_a_leap_year

_2022_is_not_a_leap_year

_1900_is_not_a_leap_year
```

```
_2016_is_a_leap_year

_2400_is_a_leap_year

_2019_is_not_a_leap_year

_2100_is_not_a_leap_year
```

```
A_year_divisible_by_4_is_a_leap_year

A_year_divisible_by_400_is_a_leap_year

A_year_not_divisible_by_4_is_not_a_leap_year

A_year_divisible_by_100_is_not_a_leap_year
```

```
A_year_not_divisible_by_4_is_not_a_leap_year

A_year_divisible_by_4_is_a_leap_year

A_year_divisible_by_100_is_not_a_leap_year

A_year_divisible_by_400_is_a_leap_year
```

```
A_year_not_divisible_by_4_is_not_a_leap_year

A_year_divisible_by_4_but_not_by_100_is_a_leap_ye

A_year_divisible_by_100_but_not_by_400_is_not_a_

A_year_divisible_by_400_is_a_leap_year
```

```
A_year_not_divisible_by_4_is_not_a_leap_year

A_year_divisible_by_4_but_not_by_100_is_a_leap_year

A_year_divisible_by_100_but_not_by_400_is_not_a_leap_year

A_year_divisible_by_400_is_a_leap_year
```

```
IsLeapYear_YearNotDivisibleBy4_ReturnsFalse

IsLeapYear_YearDivisibleBy4ButNotBy100_ReturnsTrue

IsLeapYear_YearDivisibleBy100ButNotBy400_ReturnsFalse

IsLeapYear_YearDivisibleBy400_ReturnsTrue
```

```
A_year_not_divisible_by_4_should_not_be_a_leap_year

A_year_divisible_by_4_but_not_by_100_should_not_be_a_leap_y

A_year_divisible_by_100_but_not_by_400_should_not_be_a_leap

A_year_divisible_by_400_should_not_be_a_leap_year
```

```
A_year_not_divisible_by_4_must_not_be_a_leap_year

A_year_divisible_by_4_but_not_by_100_must_not_be_a_leap_yea

A_year_divisible_by_100_but_not_by_400_must_not_be_a_leap_y

A_year_divisible_by_400_must_not_be_a_leap_year
```

```
A_year_not_divisible_by_4_is_not_a_leap_year

A_year_divisible_by_4_but_not_by_100_is_a_leap_year

A_year_divisible_by_100_but_not_by_400_is_not_a_leap_year

A_year_divisible_by_400_is_a_leap_year
```

```
public static bool IsLeapYear(int year)
{
    return
        year % 4 == 0 &&
        year % 100 != 0 ||
        year % 400 == 0;
}
```

```
A_year_not_divisible_by_4_is_not_a_leap_year

A_year_divisible_by_4_but_not_by_100_is_a_leap_year

A_year_divisible_by_100_but_not_by_400_is_not_a_leap_year

A_year_divisible_by_400_is_a_leap_year
```

```
public static bool IsLeapYear(int year)
{
    return year % 4 == 0;
}
```

```
A_year_not_divisible_by_4_is_not_a_leap_year

A_year_divisible_by_4_but_not_by_100_is_a_leap_year

A_year_divisible_by_100_but_not_by_400_is_not_a_leap_year

A_year_divisible_by_400_is_a_leap_year
```

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.

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

Nat Pryce & Steve Freeman

Are your tests really driving your development?

```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year
    {
        [Test]
        public void if_it_is_divisible_by_4_but_not_by_100() ...
        [Test]
        public void if_it_is_divisible_by_400() ...
    }
    public class A_year_is_not_a_leap_year
    {
        [Test]
        public void if_it_is_not_divisible_by_4() ...
        [Test]
        public void if_it_is_divisible_by_100_but_not_by_400() ...
    }
}
```

```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year
    {
        [Test]
        public void if_it_is_divisible_by_4_but_not_by_100()
        [Test]
        public void if_it_is_divisible_by_400()
    }
    public class A_year_is_not_a_leap_year
    {
        [Test]
        public void if_it_is_not_divisible_by_4()
        [Test]
        public void if_it_is_divisible_by_100_but_not_by_400()
    }
}
```

```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year
    {
        [Test]
        public void if_it_is_divisible_by_4_but_not_by_100()
        {
            Assert.IsTrue(IsLeapYear(2020));
        }
        [Test]
        public void if_it_is_divisible_by_400() ...
    }
    public class A_year_is_not_a_leap_year ...
}
```

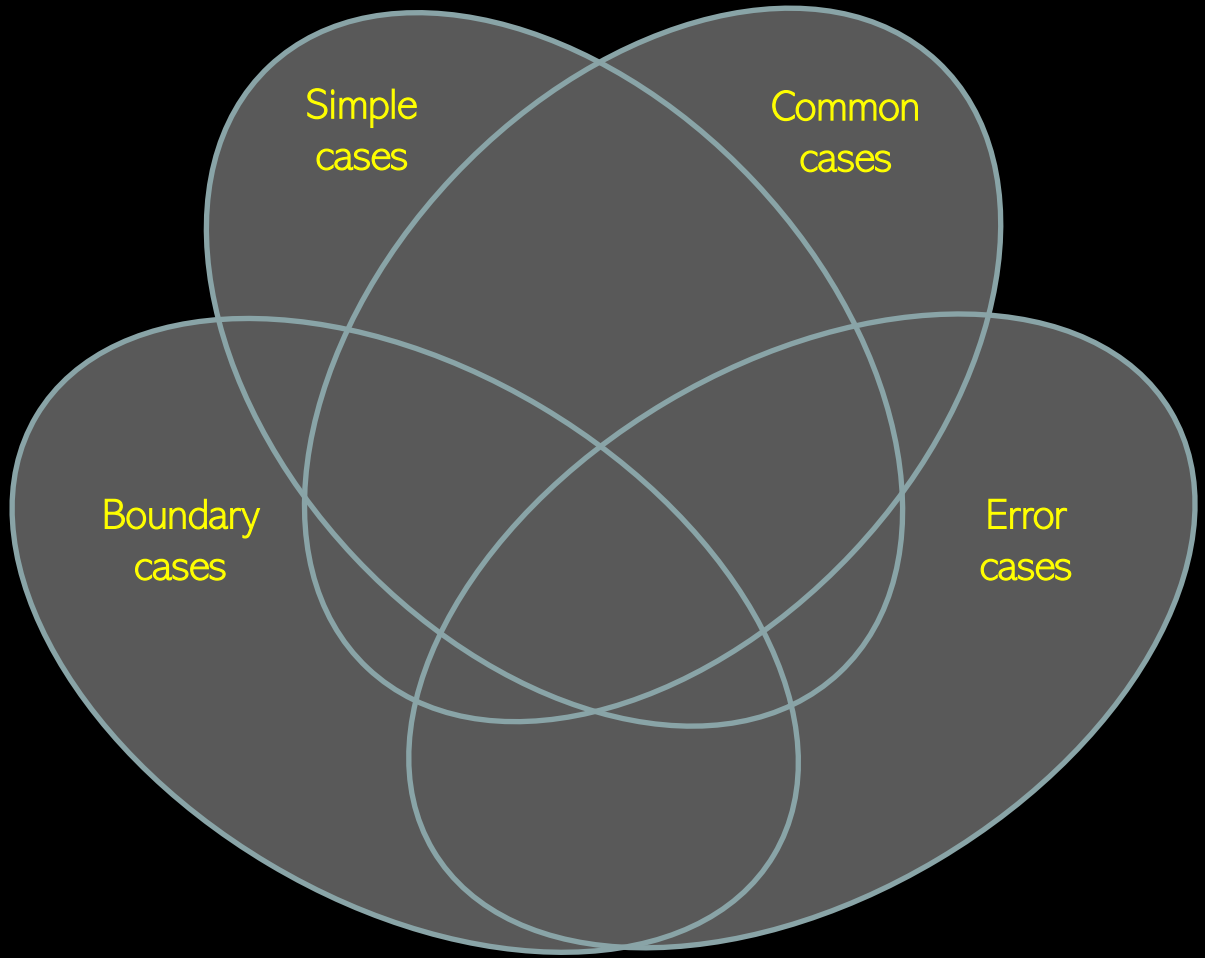
```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year
    {
        [Test]
        public void if_it_is_divisible_by_4_but_not_by_100(
            [Values(2020, 2016, 1984, 4)] int year)
        {
            Assert.IsTrue(IsLeapYear(year));
        }
        [Test]
        public void if_it_is_divisible_by_400() ...
    }
    public class A_year_is_not_a_leap_year ...
}
```

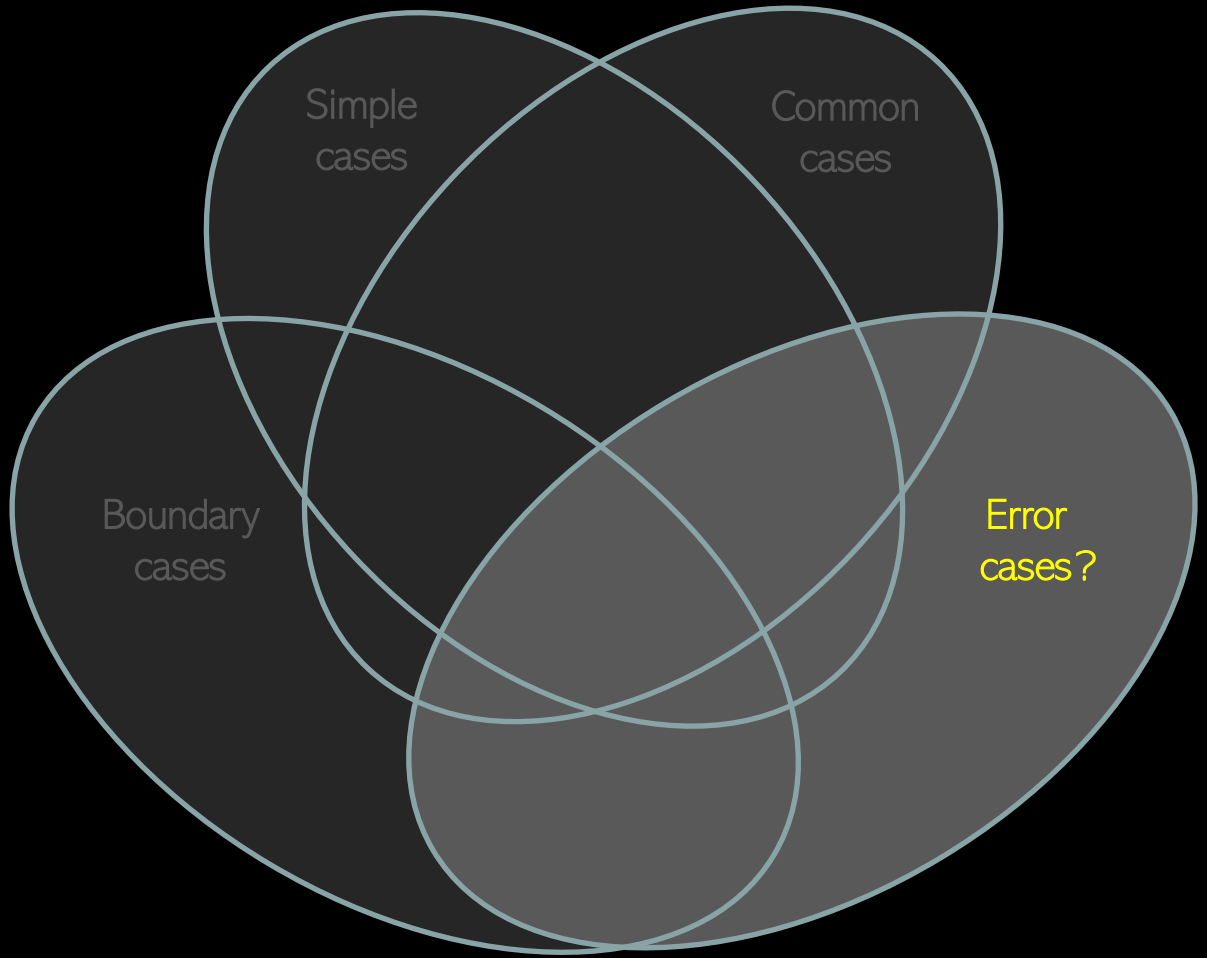
```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year
    {
        [Test]
        public void if_it_is_divisible_by_4_but_not_by_100(
            [Values(2020, 2016, 1984, 4)] int year)
        {
            Assert.IsTrue(IsLeapYear(year));
        }
        [Test]
        public void if_it_is_divisible_by_400() ...
    }
    public class A_year_is_not_a_leap_year ...
}
```

```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year
    {
        [Test]
        public void if_it_is_divisible_by_4_but_not_by_100(
            [Values(2020, 2016, 1984, 4)] int year)
        {
            Assert.IsTrue(IsLeapYear(year));
        }
        [Test]
        public void if_it_is_divisible_by_400(
            [Range(400, 4000, 400)] int year)
        {
            Assert.IsTrue(IsLeapYear(year));
        }
    }
    public class A_year_is_not_a_leap_year ...
}
```

```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year
    {
        [Test]
        public void if_it_is_divisible_by_4_but_not_by_100(
            [Values(2020, 2016, 1984, 4)] int year) ...
        [Test]
        public void if_it_is_divisible_by_400(
            [Range(400, 4000, 400)] int year) ...
    }

    public class A_year_is_not_a_leap_year
    {
        [Test]
        public void if_it_is_not_divisible_by_4(
            [Values(2022, 2019, 1999, 1)] int year) ...
        [Test]
        public void if_it_is_divisible_by_100_but_not_by_400(
            [Values(2100, 1900, 100)] int year) ...
    }
}
```


Simple cases
Common cases
Boundary cases
Error cases


Simple cases
Common cases
Boundary cases
Error cases?

```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year ...
    public class A_year_is_not_a_leap_year ...
    public class A_year_is_not_supported ...
}
```

```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year ...
    public class A_year_is_not_a_leap_year ...
    public class A_year_is_not_supported
    {
        [Test]
        public void if_it_is_0()
        {
            Assert.Catch<ArgumentException>(() => IsLeapYear(0));
        }
    }
}
```

```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year ...
    public class A_year_is_not_a_leap_year ...
    public class A_year_is_not_supported
    {
        [Test]
        public void if_it_is_0()
        {
            Assert.Catch<ArgumentException>(() => IsLeapYear(0));
        }
        [Test]
        public void if_it_is_negative(
            [Values(-1, -4, -100, -400)] int year)
        {
            Assert.Catch<ArgumentException>(() => IsLeapYear(year));
        }
    }
}
```

```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year ...
    public class A_year_is_not_a_leap_year ...
    public class A_year_is_not_supported
    {
        [Test]
        public void if_it_is_0()
        {
            Assert.Catch<ArgumentException>(() => IsLeapYear(0));
        }
        [Test]
        public void if_it_is_negative(
            [Values(-1, -4, -100, -400, int.MinValue)] int year)
        {
            Assert.Catch<ArgumentException>(() => IsLeapYear(year));
        }
    }
}
```

```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year ...
    public class A_year_is_not_a_leap_year ...
    public class A_year_is_supported ...
    public class A_year_is_not_supported ...
}
```

```
namespace Leap_year_spec
{
    public class A_year_is_a_leap_year ...

    public class A_year_is_not_a_leap_year ...

    public class A_year_is_supported
    {
        [Test]
        public void if_it_is_positive(
            [Values(1, int.MaxValue)] int year)
        {
            Assert.DoesNotThrow(() => IsLeapYear(year));
        }
    }

    public class A_year_is_not_supported ...
}
```

```
namespace Leap_year_spec ...
    public class A_year_is_a_leap_year ...
        public void if_it_is_divisible_by_4_but_not_by_400(...) ...
        public void if_it_is_divisible_by_400(...) ...
    public class A_year_is_not_a_leap_year ...
        public void if_is_not_divisible_by_4(...) ...
        public void if_it_is_divisible_by_100_but_not_by_400(...) ...
    public class A_year_is_supported ...
        public void if_it_is_positive(...) ...
    public class A_year_is_not_supported ...
        public void if_it_is_0() ...
        public void if_it_is_negative(...) ...
```

```
namespace Leap_year_spec
    public class A_year_is_a_leap_year
        public void if_it_is_divisible_by_4_but_not_by_400()
        public void if_it_is_divisible_by_400()
    public class A_year_is_not_a_leap_year
        public void if_is_not_divisible_by_4()
        public void if_it_is_divisible_by_100_but_not_by_400()
    public class A_year_is_supported
        public void if_it_is_positive()
    public class A_year_is_not_supported
        public void if_it_is_0()
        public void if_it_is_negative()
```


LEVEL
DRIVE
OUTPUT
FUZZBOX
INPUT
9v dc
COOLTONE



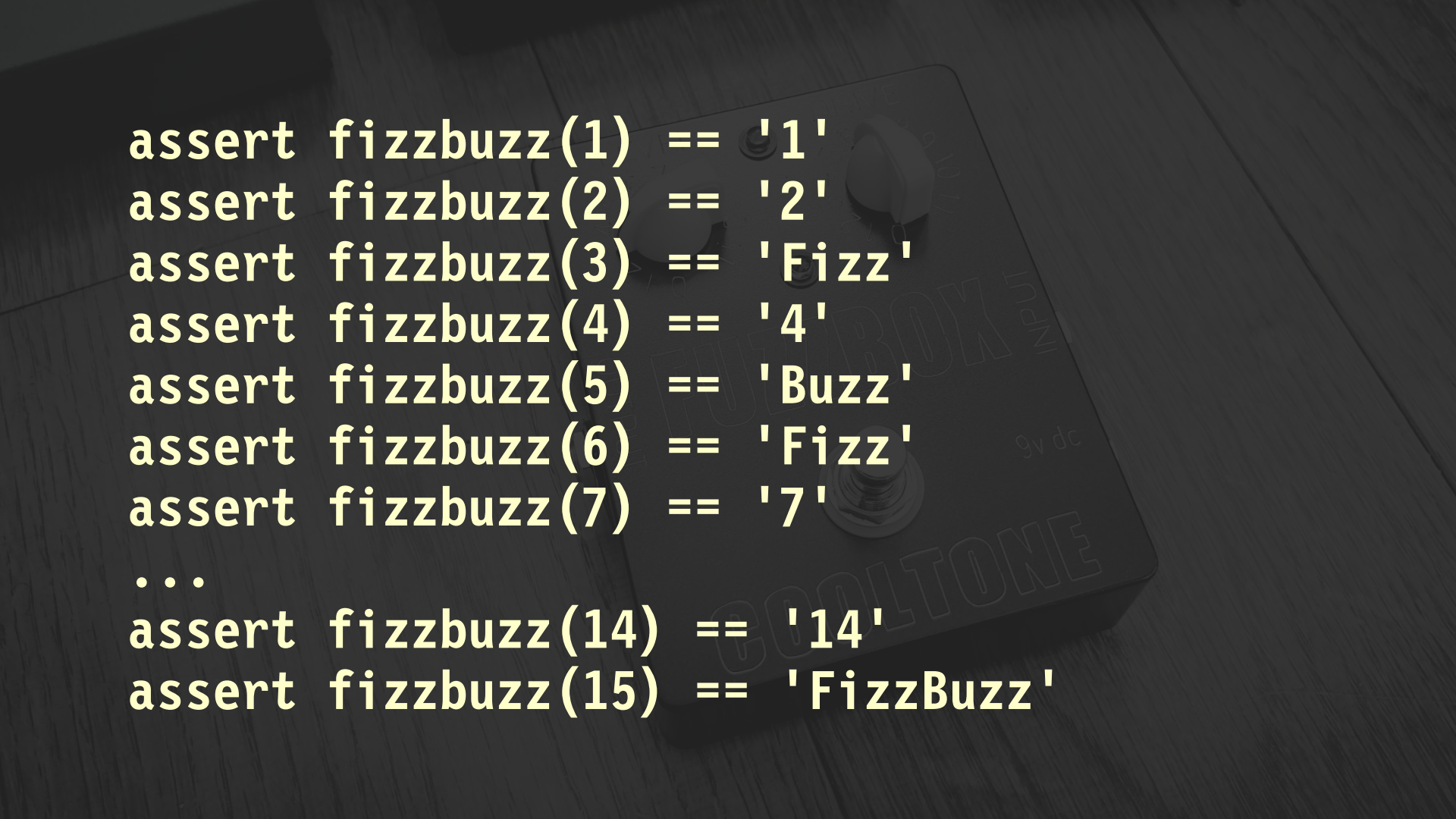
Richard Dalton
@richardadalton

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

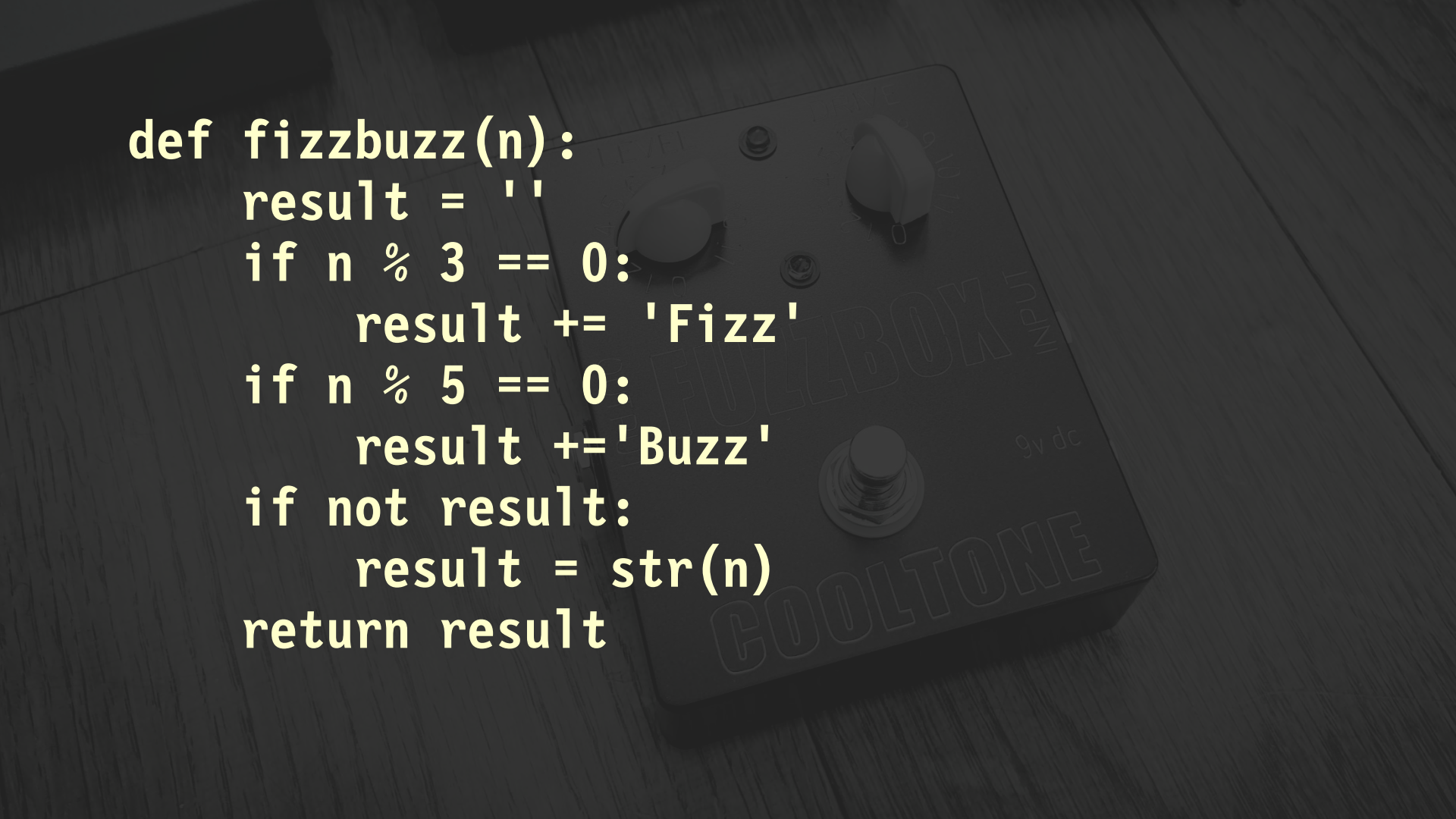
10:29 AM · Apr 24, 2015

13 15 Share this Tweet

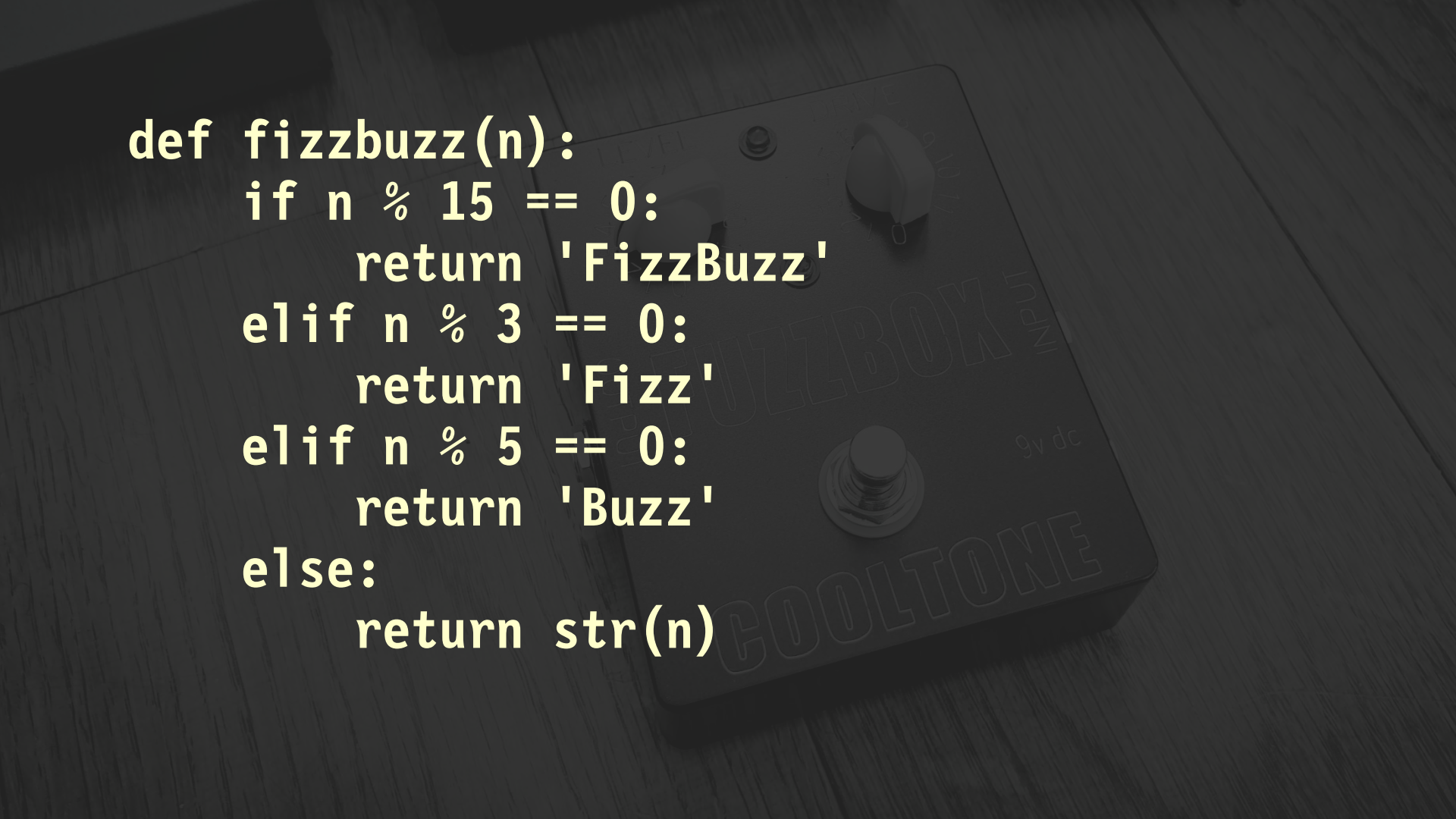
twitter.com/richardadalton/status/591534529086693376



```
assert fizzbuzz(1) == '1'
assert fizzbuzz(2) == '2'
assert fizzbuzz(3) == 'Fizz'
assert fizzbuzz(4) == '4'
assert fizzbuzz(5) == 'Buzz'
assert fizzbuzz(6) == 'Fizz'
assert fizzbuzz(7) == '7'
...
assert fizzbuzz(14) == '14'
assert fizzbuzz(15) == 'FizzBuzz'
```




```
def fizzbuzz(n):
    result = ''
    if n % 3 == 0:
        result += 'Fizz'
    if n % 5 == 0:
        result +='Buzz'
    if not result:
        result = str(n)
    return result
```



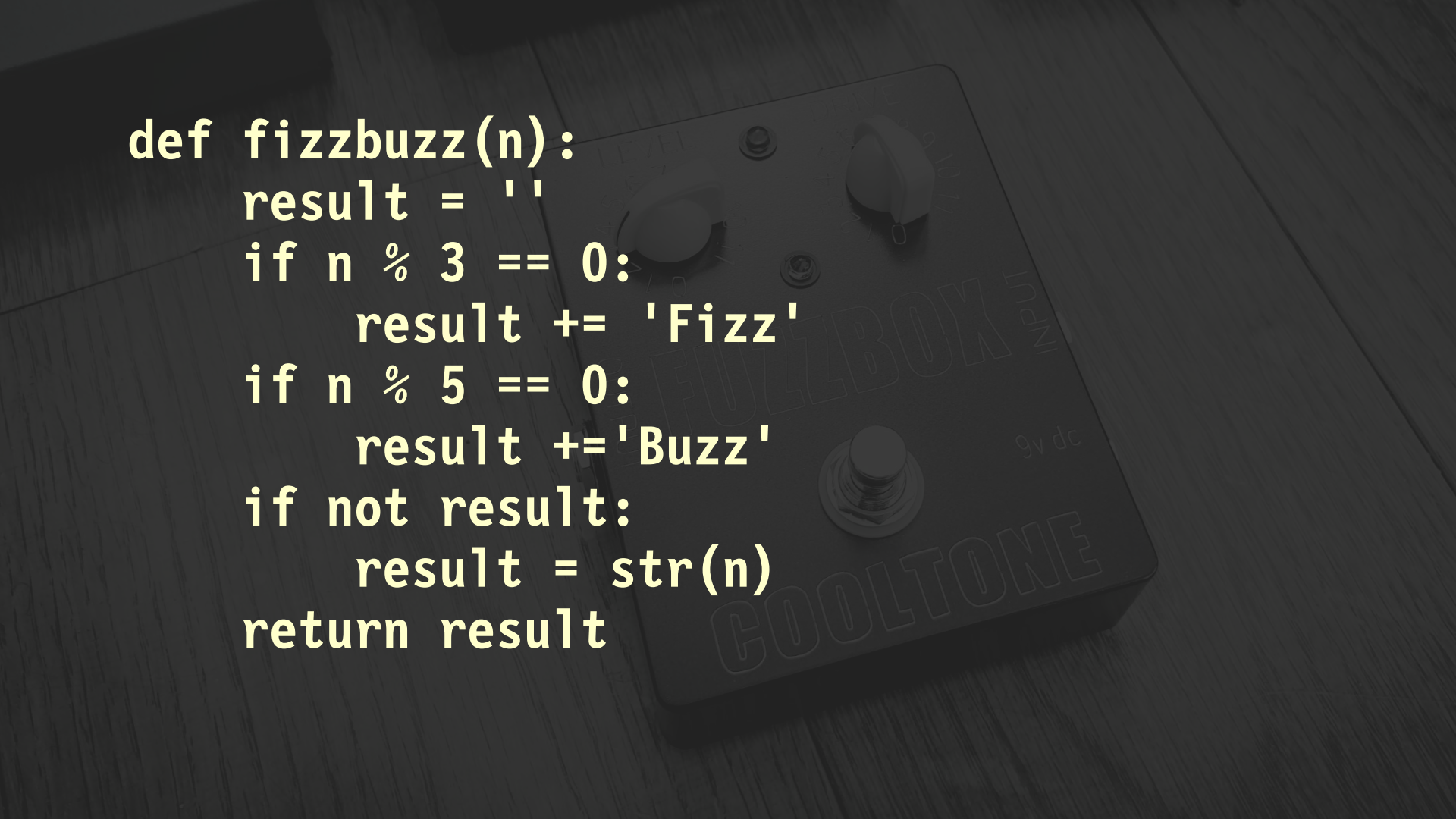

```
def fizzbuzz(n):
    if n % 15 == 0:
        return 'FizzBuzz'
    elif n % 3 == 0:
        return 'Fizz'
    elif n % 5 == 0:
        return 'Buzz'
    else:
        return str(n)
```

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

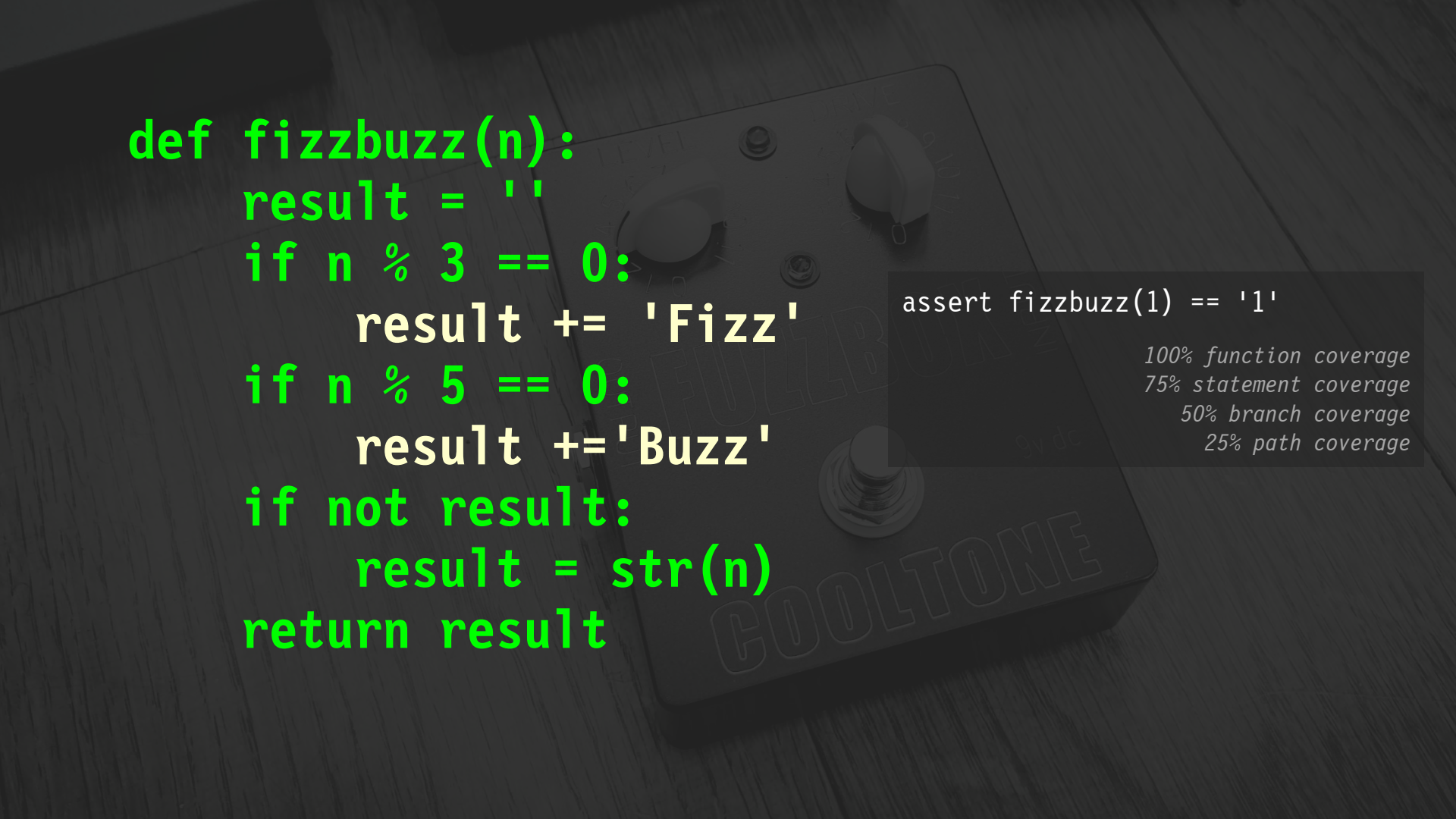
Brian Marick

martinfowler.com/bliki/TestCoverage.html

coverage



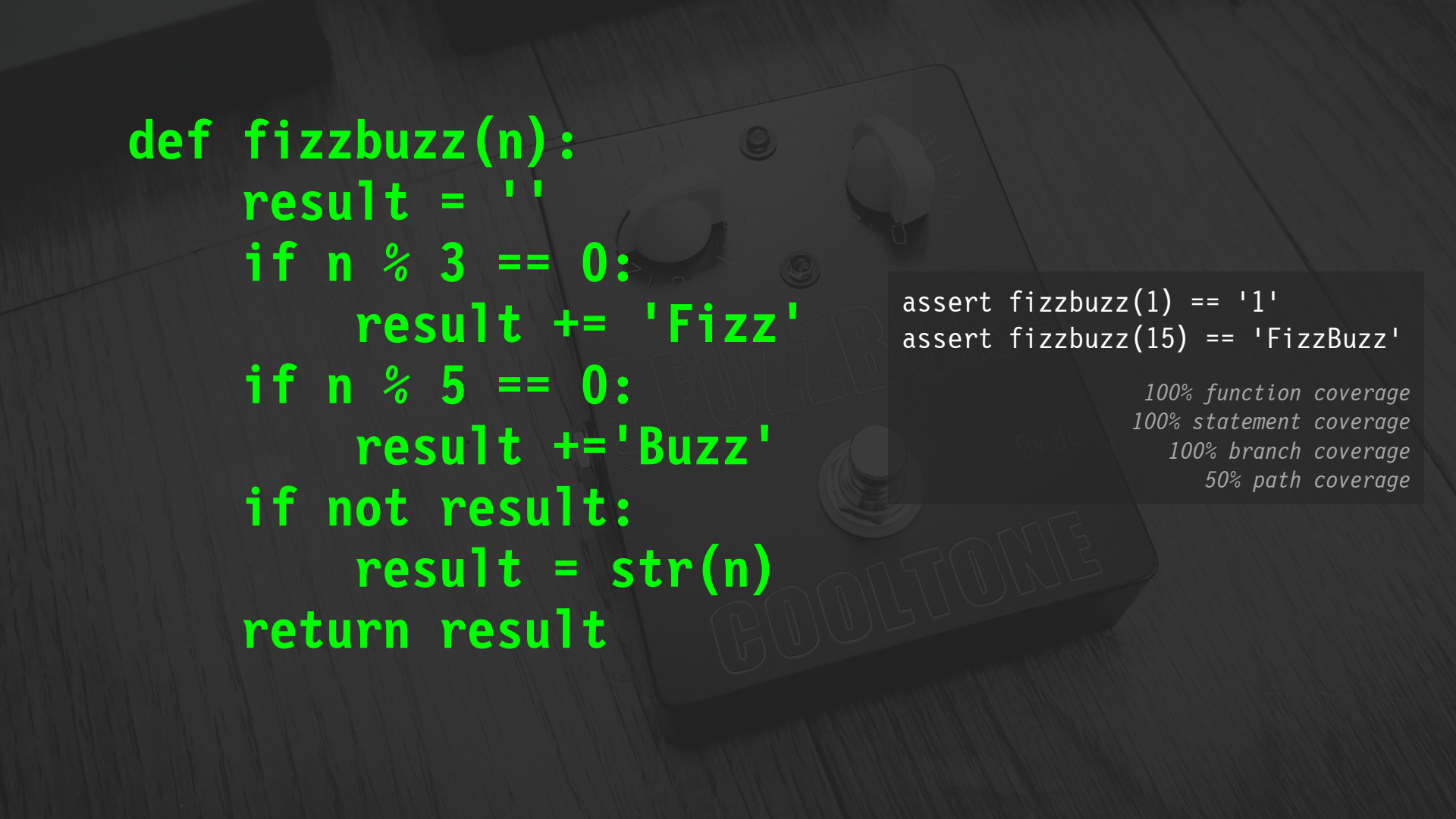

```
def fizzbuzz(n):
    result = ''
    if n % 3 == 0:
        result += 'Fizz'
    if n % 5 == 0:
        result +='Buzz'
    if not result:
        result = str(n)
    return result
```




```
def fizzbuzz(n):
    result = ''
    if n % 3 == 0:
        result += 'Fizz'
    if n % 5 == 0:
        result +='Buzz'
    if not result:
        result = str(n)
    return result
```

```
assert fizzbuzz(1) == '1'
```

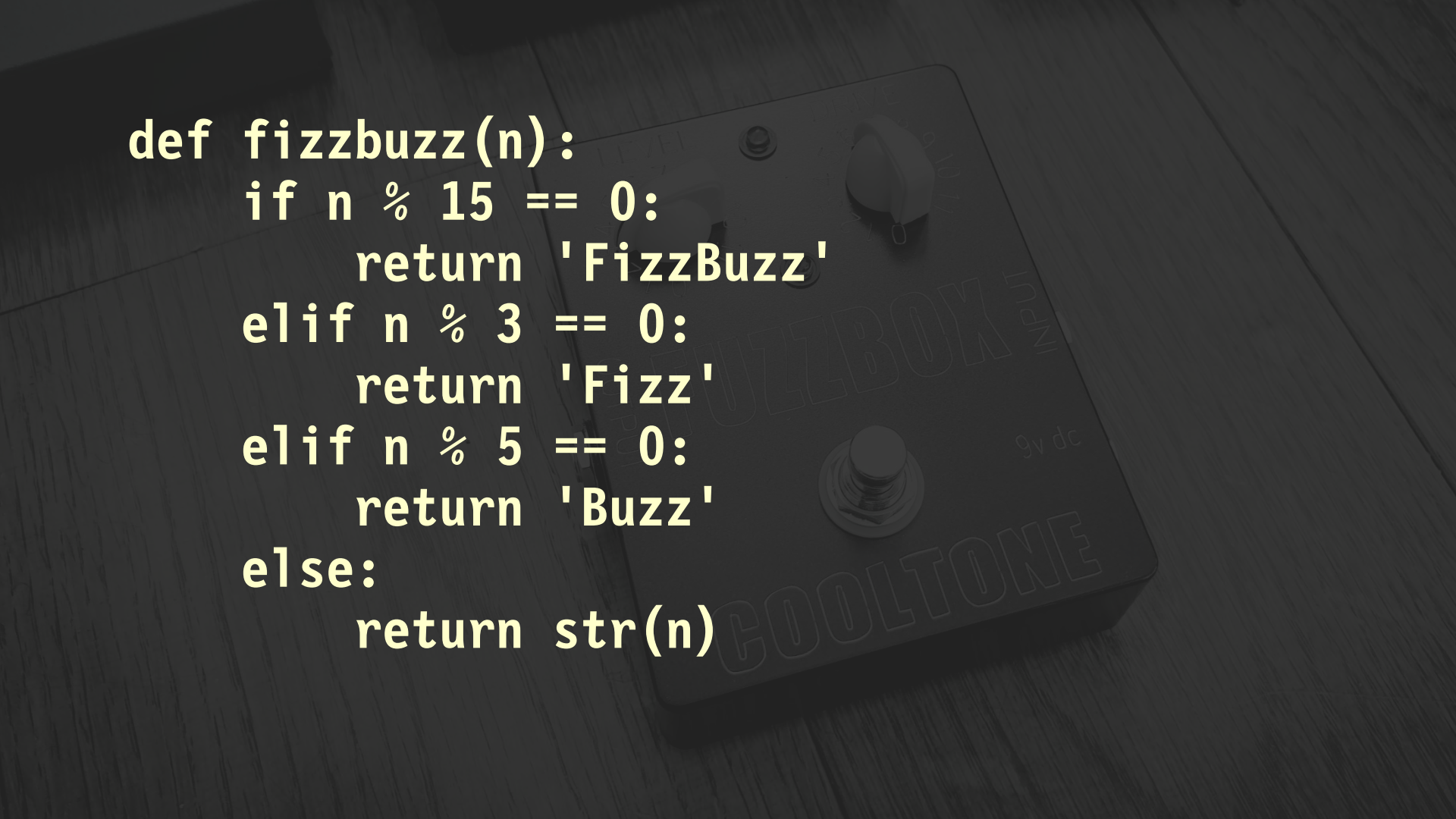
100% function coverage
75% statement coverage
50% branch coverage
25% path coverage



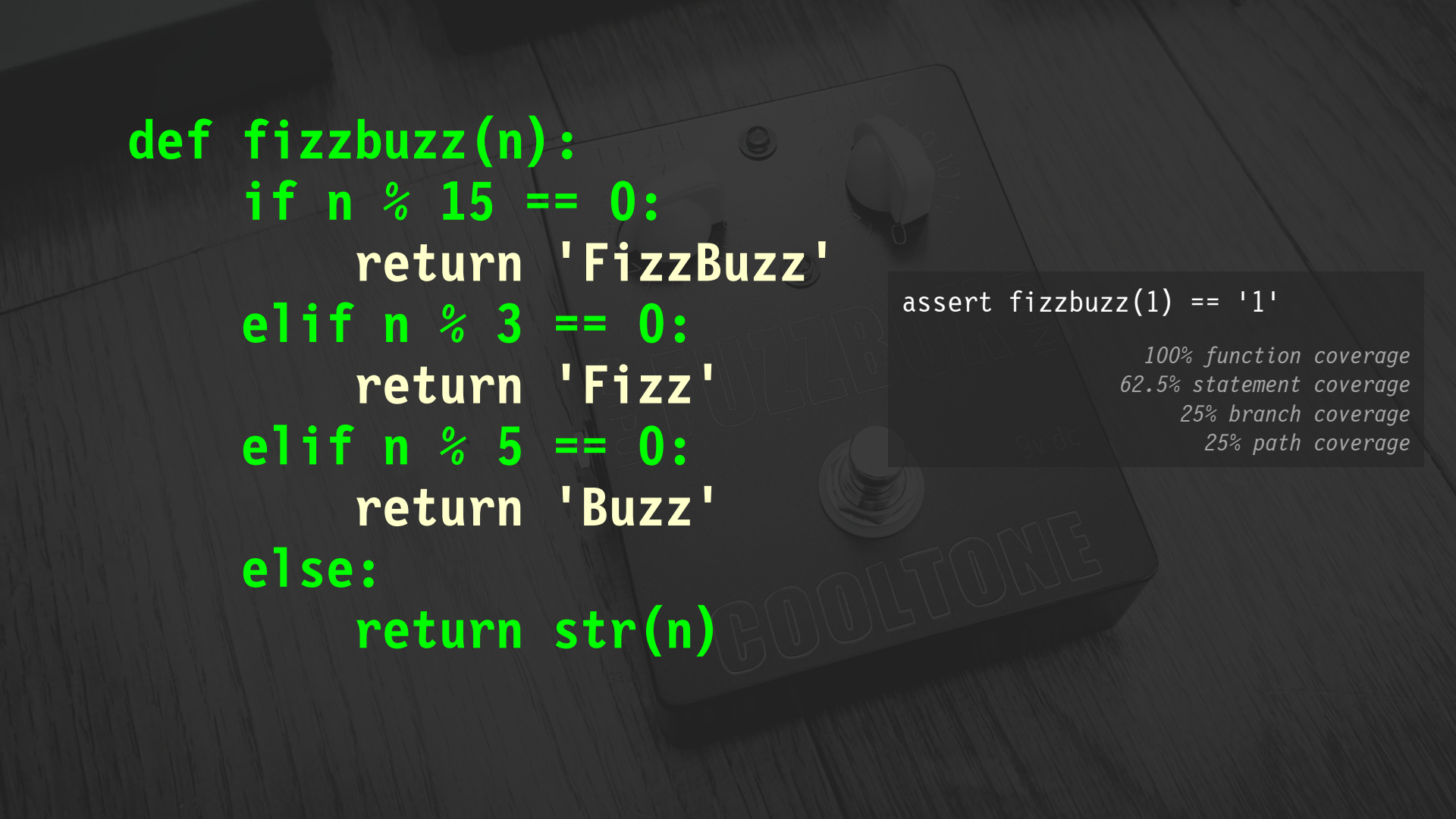

```
def fizzbuzz(n):
    result = ''
    if n % 3 == 0:
        result += 'Fizz'
    if n % 5 == 0:
        result +='Buzz'
    if not result:
        result = str(n)
    return result
```

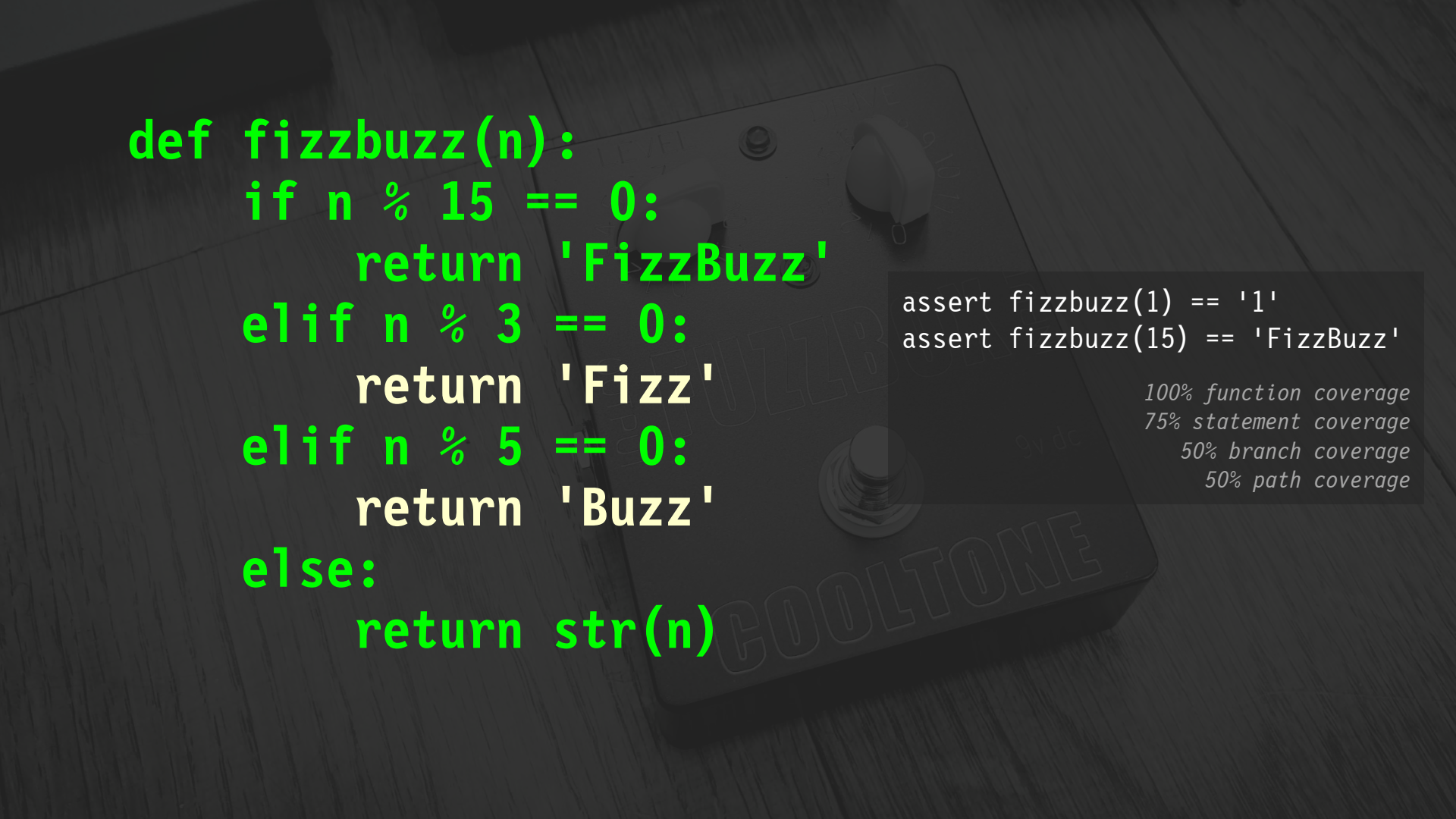
```
assert fizzbuzz(1) == '1'
assert fizzbuzz(15) == 'FizzBuzz'
```

100% function coverage
100% statement coverage
100% branch coverage
50% path coverage




```
def fizzbuzz(n):
    if n % 15 == 0:
        return 'FizzBuzz'
    elif n % 3 == 0:
        return 'Fizz'
    elif n % 5 == 0:
        return 'Buzz'
    else:
        return str(n)
```

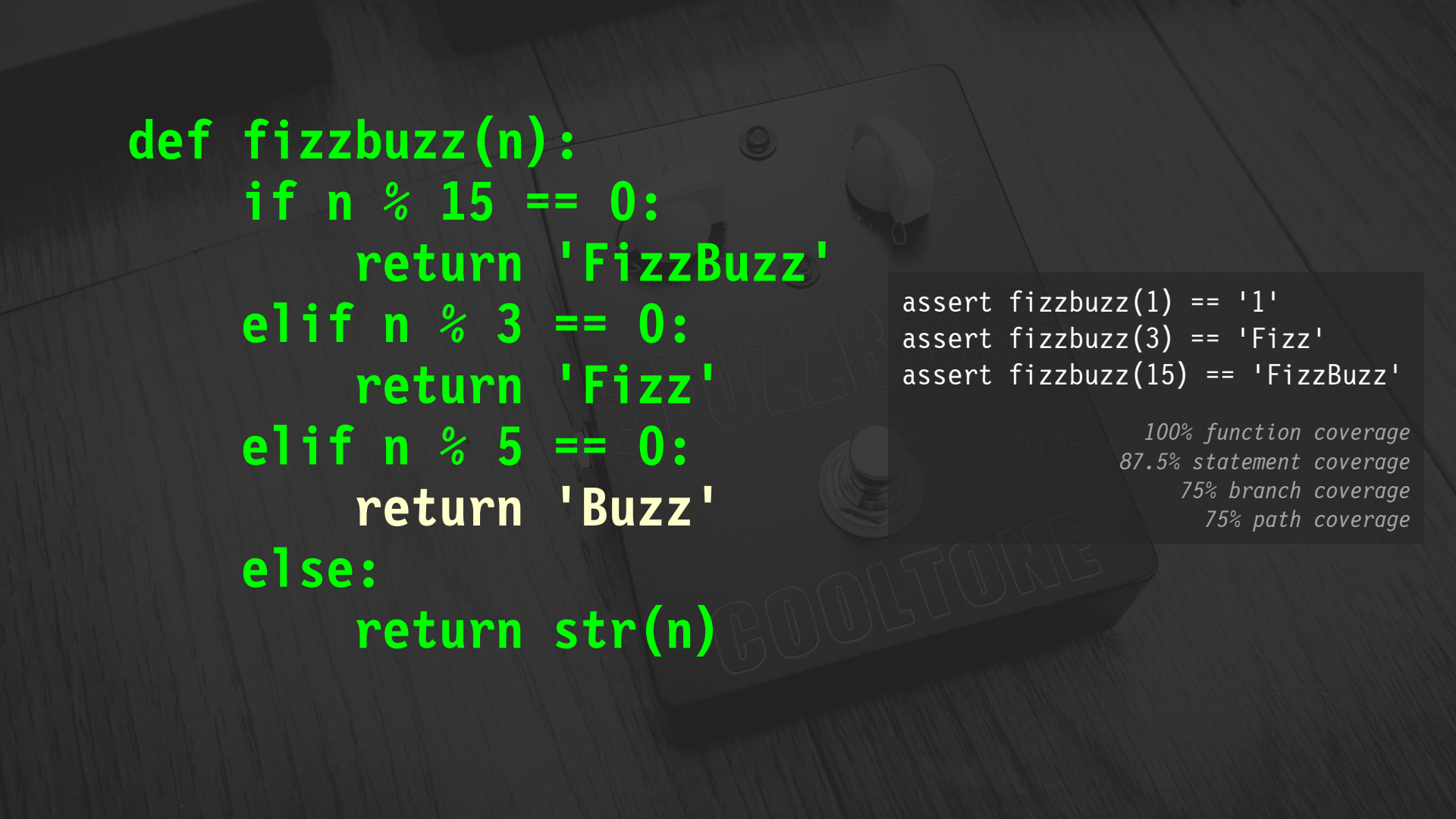

def fizzbuzz(n):
if n % 15 == 0:
return 'FizzBuzz'
elif n % 3 == 0:
return 'Fizz'
elif n % 5 == 0:
return 'Buzz'
else:
return str(n)
assert fizzbuzz(1) == '1'
100% function coverage
62.5% statement coverage
25% branch coverage
25% path coverage




```
def fizzbuzz(n):
    if n % 15 == 0:
        return 'FizzBuzz'
    elif n % 3 == 0:
        return 'Fizz'
    elif n % 5 == 0:
        return 'Buzz'
    else:
        return str(n)
```

```
assert fizzbuzz(1) == '1'
assert fizzbuzz(15) == 'FizzBuzz'
```

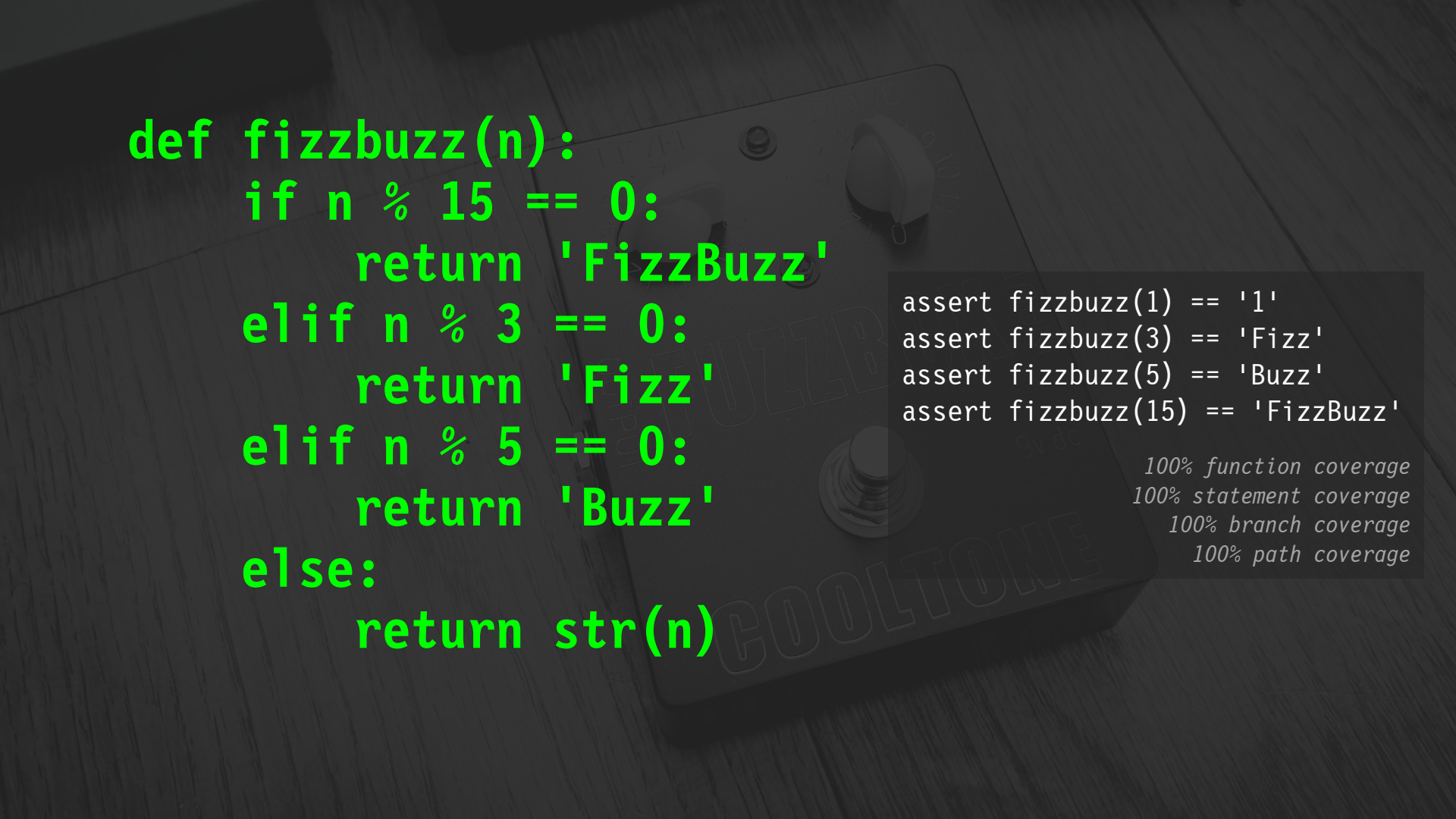
100% function coverage
75% statement coverage
50% branch coverage
50% path coverage




```python
def fizzbuzz(n):
    if n % 15 == 0:
        return 'FizzBuzz'
    elif n % 3 == 0:
        return 'Fizz'
    elif n % 5 == 0:
        return 'Buzz'
    else:
        return str(n)
```

```python
assert fizzbuzz(1) == '1'
assert fizzbuzz(3) == 'Fizz'
assert fizzbuzz(15) == 'FizzBuzz'
```

100% function coverage
87.5% statement coverage
75% branch coverage
75% path coverage


def fizzbuzz(n):
if n % 15 == 0:
return 'FizzBuzz'
elif n % 3 == 0:
return 'Fizz'
elif n % 5 == 0:
return 'Buzz'
else:
return str(n)
assert fizzbuzz(1) == '1'
assert fizzbuzz(3) == 'Fizz'
assert fizzbuzz(5) == 'Buzz'
assert fizzbuzz(15) == 'FizzBuzz'
100% function coverage
100% statement coverage
100% branch coverage
100% path coverage

The first principle is that you must not fool yourself — and you are the easiest person to fool.

Richard Feynman

every result is 'Fizz', 'Buzz', 'FizzBuzz' or a decimal string
every decimal result corresponds to its ordinal position
every third result contains 'Fizz'
every fifth result contains 'Buzz'
every fifteenth result is 'FizzBuzz'
the ordinal position of every 'Fizz' result is divisible by 3
the ordinal position of every 'Buzz' result is divisible by 5
the ordinal position of every 'FizzBuzz' result is divisible by 15

```
result = [fizzbuzz(n) for n in range(1, 101)]
properties = [
    all(a in {'Fizz', 'Buzz', 'FizzBuzz'} or a.isdecimal() for a in result),
    all(int(a) == n for n, a in enumerate(result, 1) if a.isdecimal()),
    all('Fizz' in a for a in result[2::3]),
    all('Buzz' in a for a in result[4::5]),
    all(a == 'FizzBuzz' for a in result[14::15]),
    all(n % 3 == 0 for n, a in enumerate(result, 1) if a == 'Fizz'),
    all(n % 5 == 0 for n, a in enumerate(result, 1) if a == 'Buzz'),
    all(n % 15 == 0 for n, a in enumerate(result, 1) if a == 'FizzBuzz')
]
assert all(properties)
```

Algorithms + Data Structures = Programs

Niklaus Wirth

Structure + Interpretation = Programs