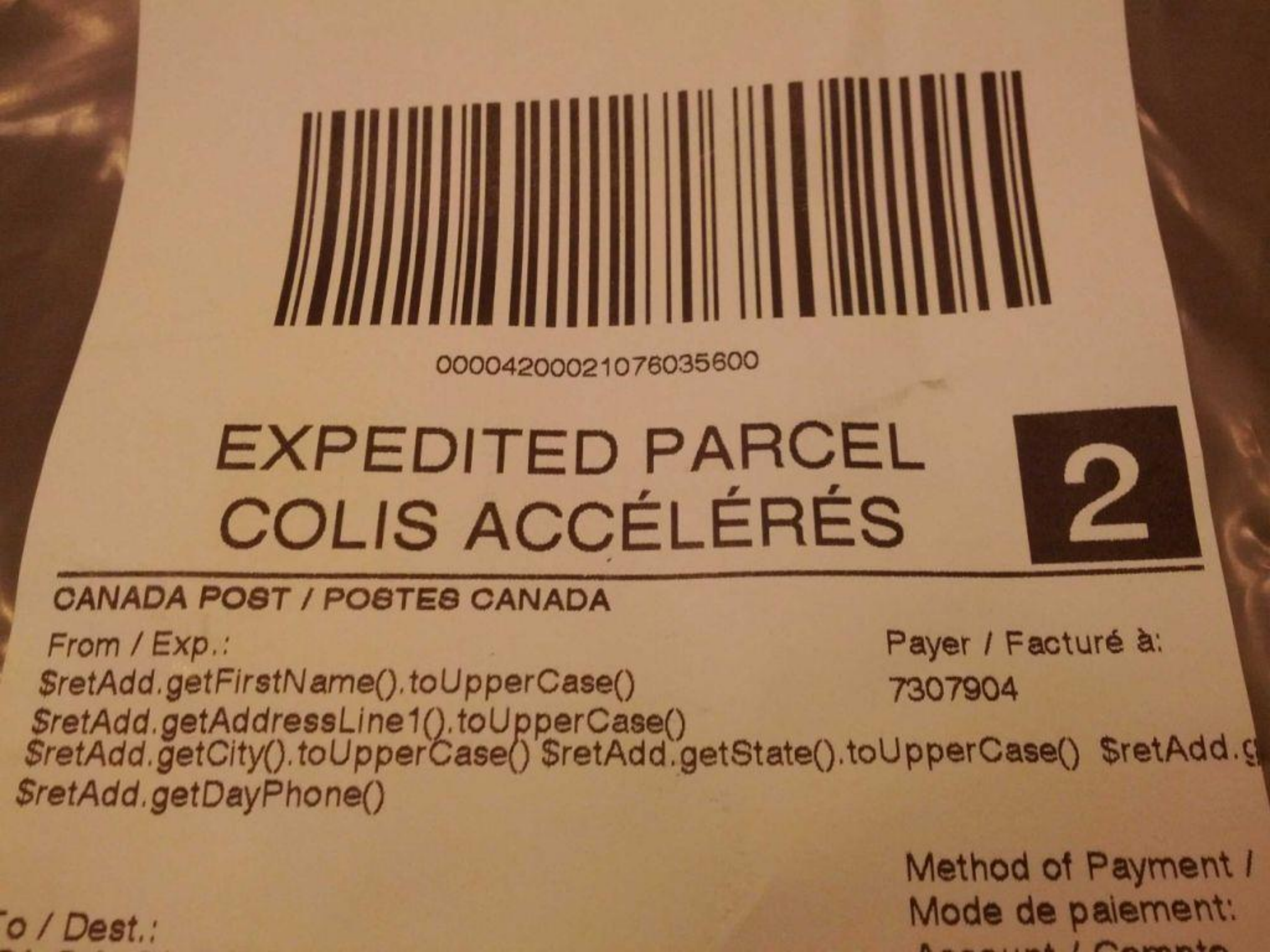
What We Talk About When We Talk About Unit Testing

@KevlinHenney

kevlin@curbralan.com

Write down the problem.
Think real hard.
Write down the solution.

The Feynman Problem-Solving Algorithm



00004200021076035600

EXPEDITED PARCEL
COLIS ACCÉLÉRÉS

2

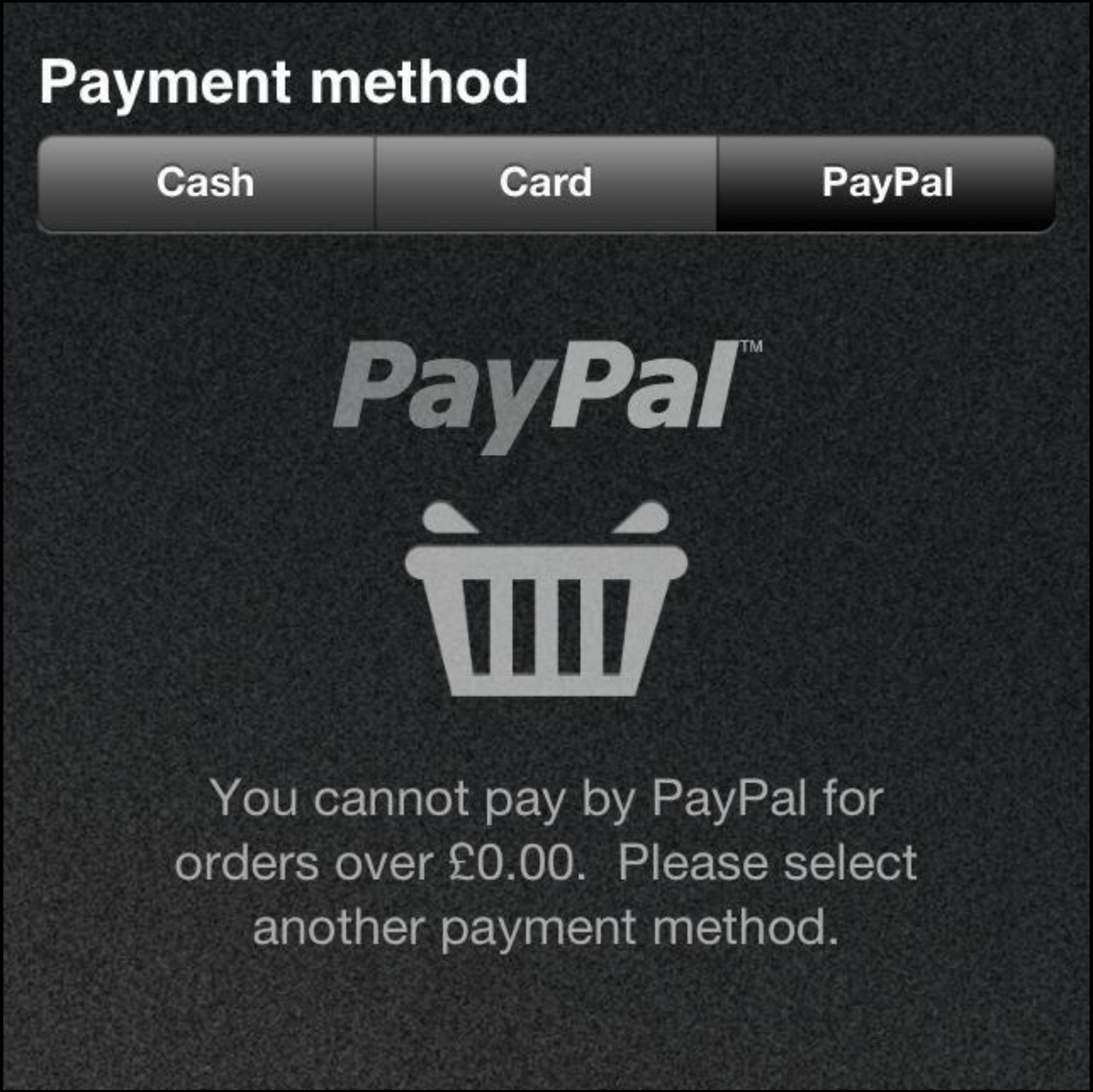
CANADA POST / POSTES CANADA

From / Exp.:
$retAdd.getFirstName().toUpperCase()
$retAdd.getAddressLine1().toUpperCase()
$retAdd.getCity().toUpperCase() $retAdd.getState().toUpperCase() $retAdd.g
$retAdd.getDayPhone()

Payer / Facturé à:
7307904

Method of Payment /
Mode de paiement:

To / Dest.:



Payment method

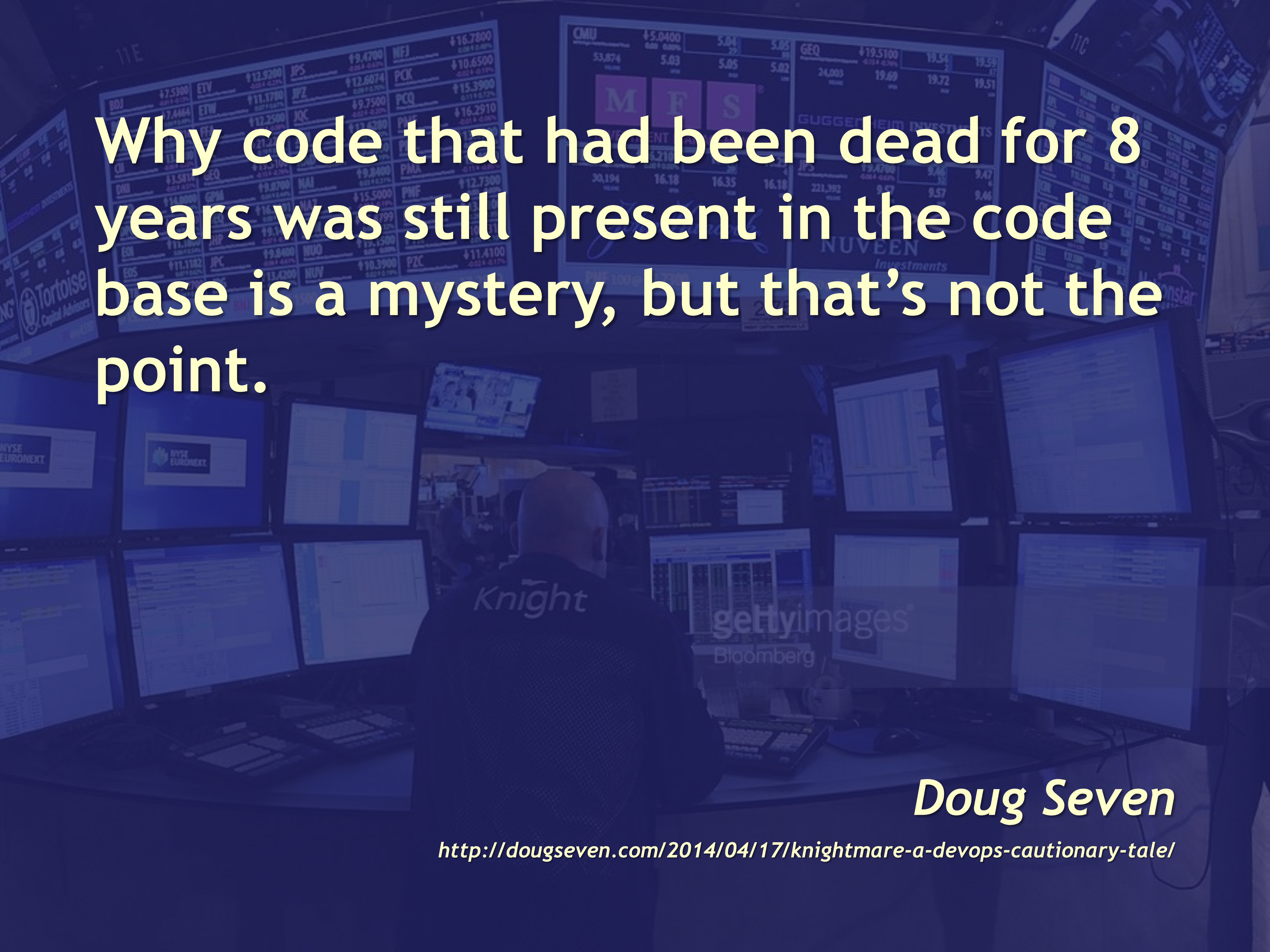
You cannot pay by PayPal for orders over £0.00. Please select another payment method.

11E
11C
MFS
INVESTMENT MANAGEMENT
GUGGENHEIM INVESTMENTS
John Hancock
NUVEEN
Investments
Tortoise
Capital Advisors
Nationstar
NYSE EURONEXT
2175
PNF 100@12.7300
BNJ 100@18.1500
MUH 300@17.2899
Knight

The update to SMARS was intended to replace old, unused code referred to as “Power Peg” — functionality that Knight hadn’t used in 8-years.

Doug Seven

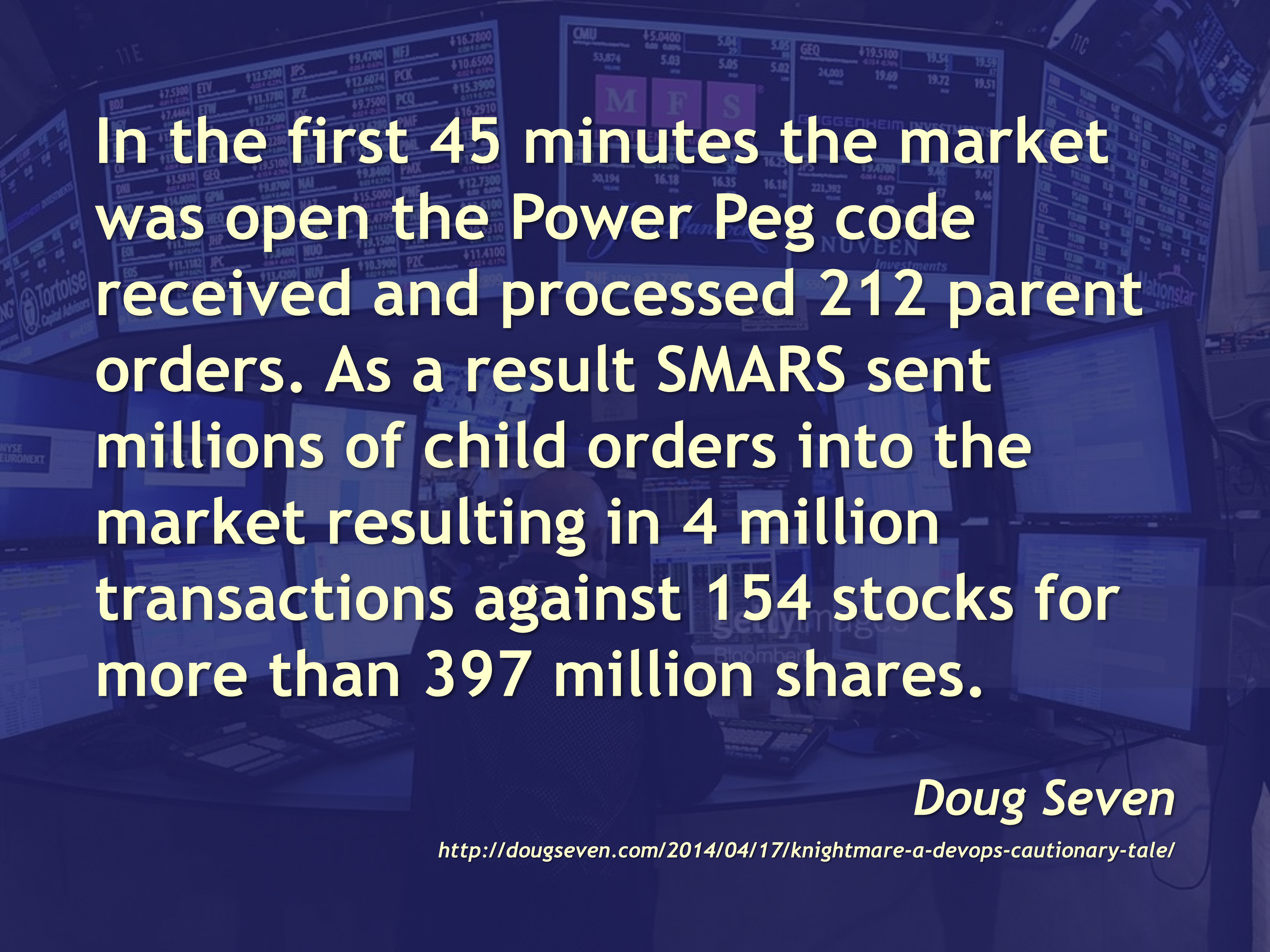
http://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/



Why code that had been dead for 8 years was still present in the code base is a mystery, but that’s not the point.

Doug Seven

http://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/



In the first 45 minutes the market was open the Power Peg code received and processed 212 parent orders. As a result SMARS sent millions of child orders into the market resulting in 4 million transactions against 154 stocks for more than 397 million shares.

Doug Seven

http://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/

Knight Capital Group realized a $460 million loss in 45 minutes.

Doug Seven

http://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/

RUD, *noun*

- Rapid Unscheduled Disassembly
- Rocket science and amateur rocketry jargon that's acronymous, euphemistic and explosively self-explanatory

Cité de l'espace
arianespace
esa
esa
cnes
cnes

The failure resulted in a loss of more than US$370 million.

http://en.wikipedia.org/wiki/Cluster_(spacecraft)

```
Juin      end if;
1996      L_M_DON_32 := TDB.T_ENTIER_32S ((1.0/C_M_LSB_DON) *
                                        G_M_INFO_DERIVE(T_ALG.E_DON))
          if L_M_DON_32 > 32767 then
             P_M_DERIVE(T_ALG.E_DON) := 16#7FFF#;
          elsif L_M_DON_32 < -32768 then
             P_M_DERIVE(T_ALG.E_DON) := 16#8000#;
          else
             P_M_DERIVE(T_ALG.E_DON) := UC_16S_EN_16NS(
                TDB.T_ENTIER_16S(L_M_DON_32));
          end if;

          P_M_DERIVE(T_ALG.E_DOE) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                                         ((1.0/C_M_LSB_DOE) *
                                         G_M_INFO_DERIVE(T_ALG.E_DOE)

          L_M_BV_32 := TDB.T_ENTIER_32S ((1.0/C_M_LSB_BV) *
                                        G_M_INFO_DERIVE(T_ALG.E_BV));
          if L_M_BV_32 > 32767 then
             P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
          elsif L_M_BV_32 < -32768 then
             P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
          else
             P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M
          end if;

501       P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                                        ((1.0/C_M_LSB_BH) *
                                        G_M_INFO_DERIVE(T_ALG.E_BH)))
       end LIRE_DERIVE;
    --$finprocedure

       --(
       procedure LIRE_SEUIL (P_M_SEUIL : out TDB.T_ENTIER_16NS) is
       --)
```

Um. What's the name of the word for things not being the same always. You know, I'm sure there is one. Isn't there?

There's must be a word for it... the thing that lets you know time is happening. Is there a word?

Change.

Oh. I was afraid of that.

Neil Gaiman
The Sandman

What experience and history teach is that nations and governments have never learned anything from history.

Georg Wilhelm Friedrich Hegel

Write down the problem.

Think real hard.

Write down the solution.

Check it.

assert

```
void * bsearch(
    const void * key,
    const void * base,
    size_t element_count,
    size_t element_size,
    int compare(const void * lhs, const void * rhs));
```

```
void * bsearch(
    const void * key,
    const void * base,
    size_t element_count,
    size_t element_size,
    int compare(const void * lhs, const void * rhs))
{
    ...
    void * result;
    ...
    assert(???); // What is the postcondition?
    return result;
}
```

```
void * bsearch(
    const void * key,
    const void * base,
    size_t element_count,
    size_t element_size,
    int compare(const void * lhs, const void * rhs))
{
    assert(???); // What is the precondition?
    void * result;
    ...
    assert(???); // What is the postcondition?
    return result;
}
```

#GoodLuck
WithThat

Test early.

Test often.

Test automatically.

Andrew Hunt and David Thomas
The Pragmatic Programmer

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

Martin Fowler

http://martinfowler.com/bliki/TestCoverage.html

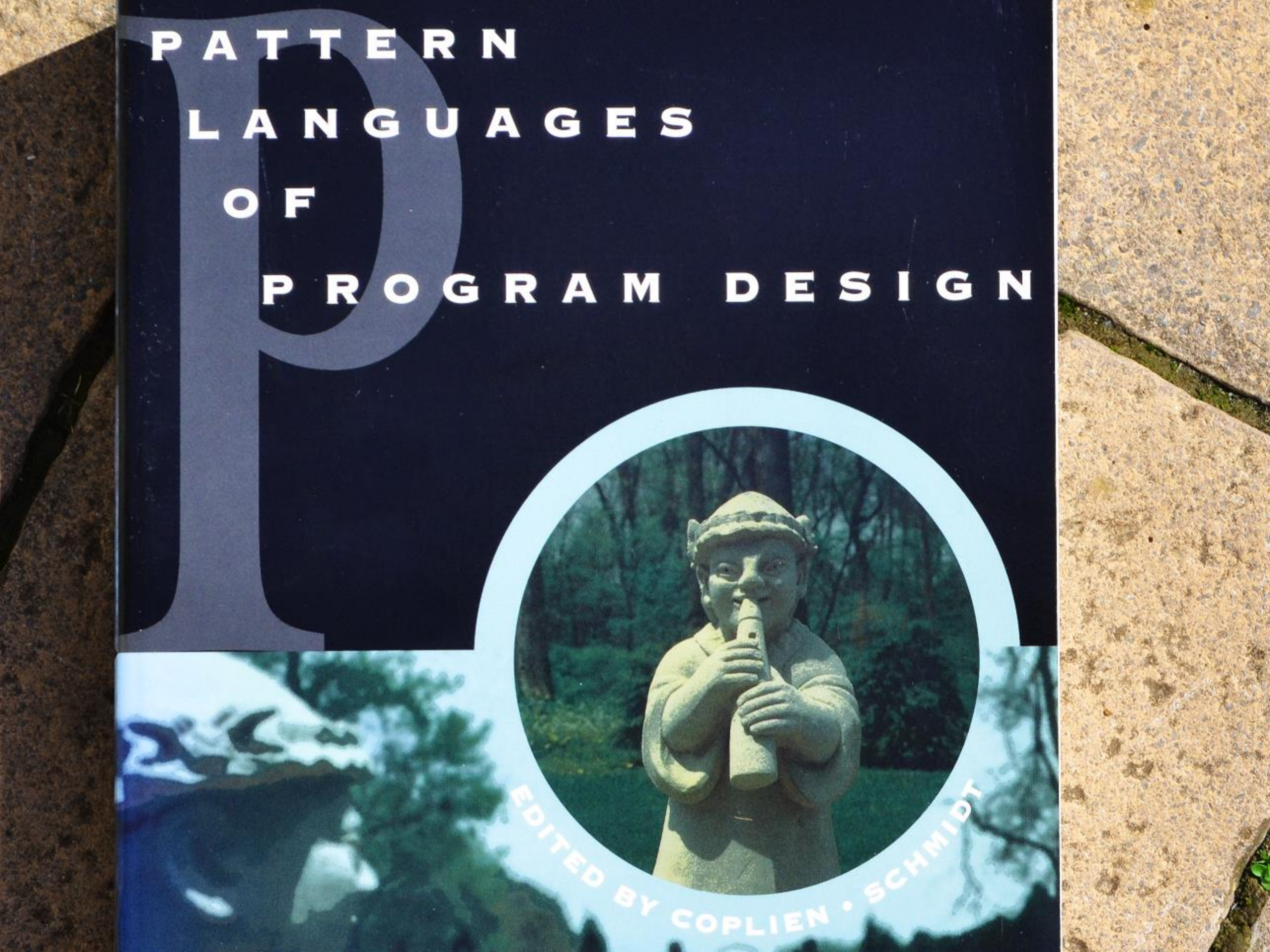
I expect a high level of coverage. Sometimes managers require one. There's a subtle difference.

Brian Marick
http://martinfowler.com/bliki/TestCoverage.html

Goodhart's law, *noun*

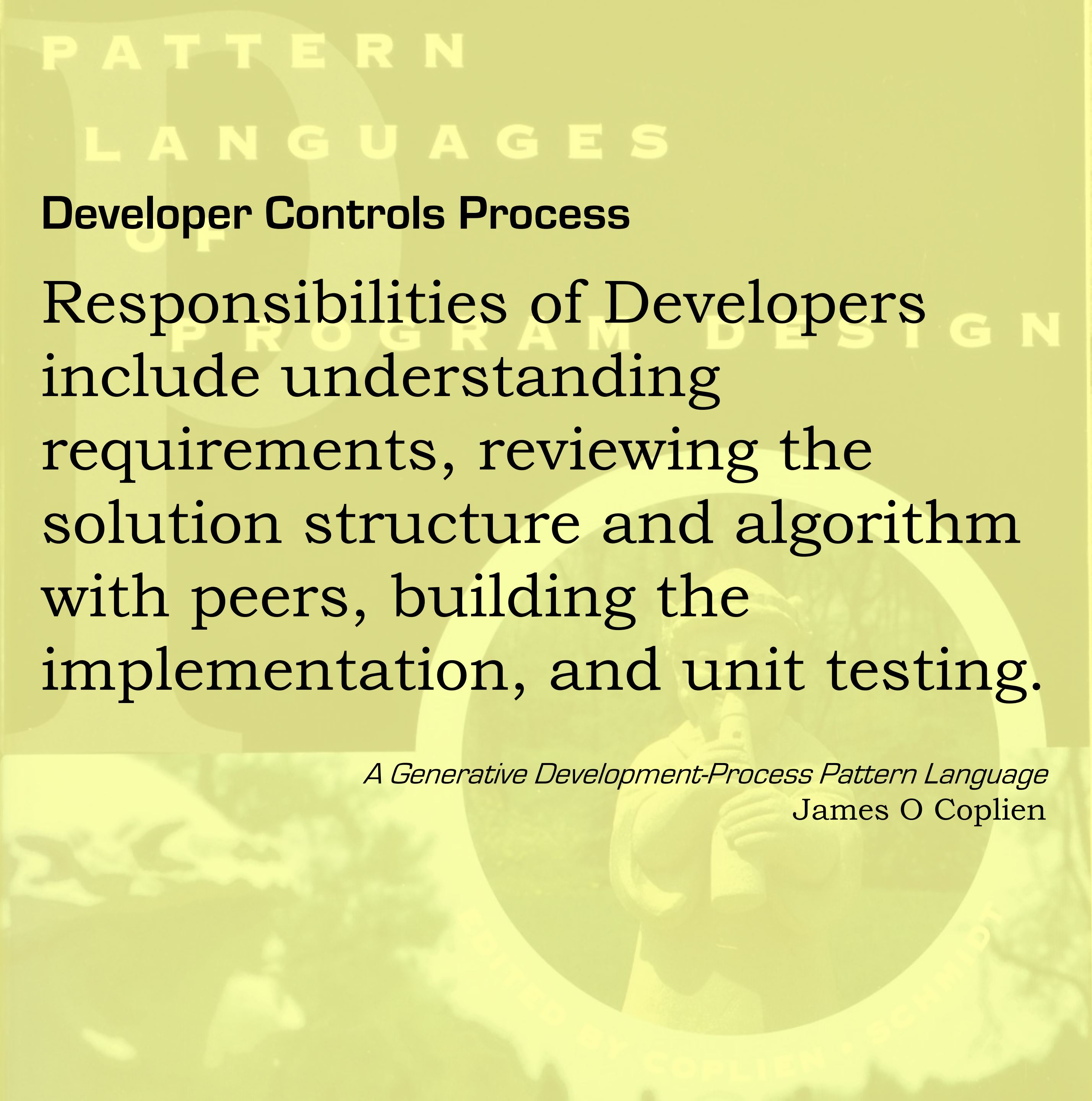
- Once a metric becomes a target, it loses its meaning as a measure.
- Named after Charles Goodhart, professor of economics at the LSE and former advisor to the Bank of England, who in 1975 observed that "Any observed statistical regularity will tend to collapse once pressure is placed upon it for control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

https://www.facebook.com/WordFriday/posts/320731068014857



PATTERN LANGUAGES OF PROGRAM DESIGN

EDITED BY COPLIEN • SCHMIDT



Developer Controls Process

Responsibilities of Developers include understanding requirements, reviewing the solution structure and algorithm with peers, building the implementation, and unit testing.

A Generative Development-Process Pattern Language
James O Coplien

When you write unit tests, TDD-style or after your development, you scrutinize, you think, and often you prevent problems without even encountering a test failure.

Michael Feathers
"The Flawed Theory Behind Unit Testing"
http://michaelfeathers.typepad.com/michael_feathers_blog/2008/06/the-flawed-theo.html

Very many people say "TDD" when they really mean, "I have good unit tests" ("I have GUTs"?). Ron Jeffries tried for years to explain what this was, but we never got a catch-phrase for it, and now TDD is being watered down to mean GUTs.

Alistair Cockburn
"The modern programming professional has GUTs"
http://alistair.cockburn.us/The+modern+programming+professional+has+GUTs

```
size_t ints_to_csv(
    const int * to_write, size_t how_many,
    char * output, size_t length);
```

```
size_t ints_to_csv(
    const int * to_write, size_t how_many, char * output, size_t length)
{
    size_t result = 0;
    if(length != 0)
    {
        if(how_many == 0)
        {
            output[0] = '\0';
        }
        else
        {
            for(size_t which = 0; which != how_many && result != length; ++which)
            {
                result +=
                    snprintf(
                        output + result, length - result,
                        which == 0 ? "%i" : ",%i",
                        to_write[which]);
            }
            result = result > length - 1 ? length - 1 : result;
        }
    }
    return result;
}
```

```
extern "C" size_t ints_to_csv(
    const int * to_write, size_t how_many, char * output, size_t length)
{
    size_t result = 0;

    if(length != 0)
    {
        output[length - 1] = '\0';
        std::ostrstream buffer(output, length - 1);

        for(size_t which = 0; which != how_many; ++which)
            buffer << (which == 0 ? "" : ",") << to_write[which];

        buffer << std::ends;
        result = std::strlen(output);
    }

    return result;
}
```

test → function

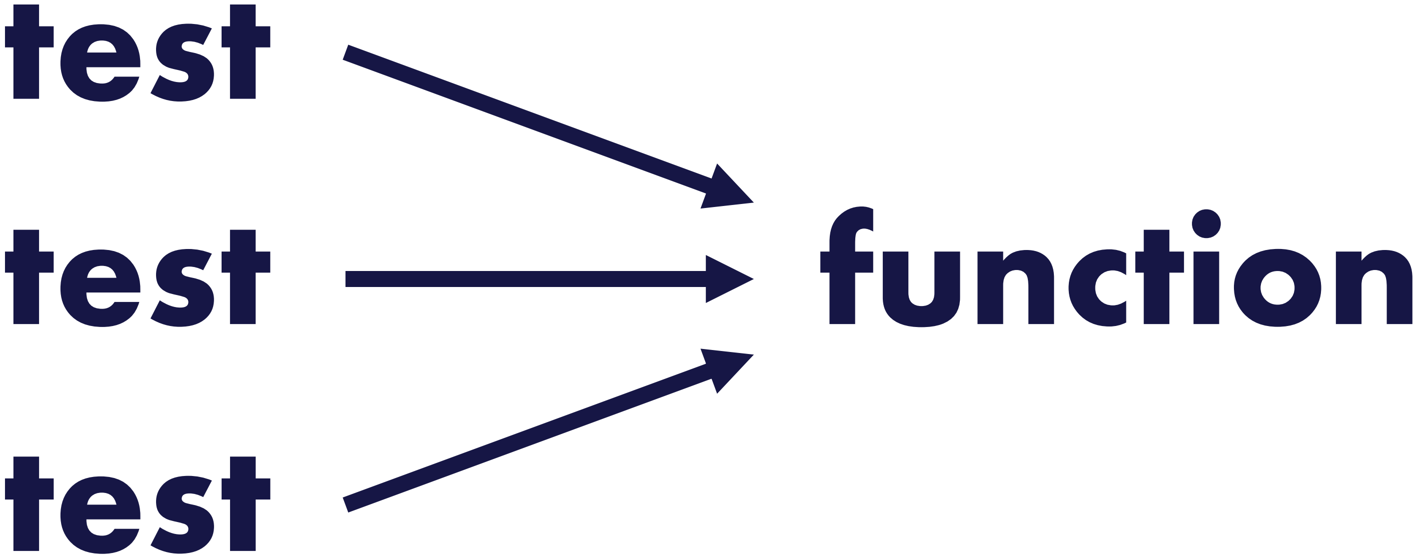
```
void test_ints_to_csv()
{
    size_t written = ints_to_csv(NULL, 0, NULL, 0);
    assert(written == 0);
    const int input[] = { 42 };
    written = ints_to_csv(input, 1, NULL, 0);
    assert(written == 0);
    char output[3] = "+++";
    written = ints_to_csv(NULL, 0, output, sizeof output);
    assert(written == 0);
    assert(output[0] == '\0');
    memcpy(output, "+++", sizeof output);
    written = ints_to_csv(input, 1, output, sizeof output);
    assert(written == 2);
    assert(strcmp(output, "42") == 0);
    ...
}
```

```
void test_ints_to_csv()
{
    // No values from null to null output writes nothing
    size_t written = ints_to_csv(NULL, 0, NULL, 0);
    assert(written == 0);

    // Value to null output writes nothing
    const int input[] = { 42 };
    written = ints_to_csv(input, 1, NULL, 0);
    assert(written == 0);

    // No values to sufficient output writes empty
    char output[3] = "+++";
    written = ints_to_csv(NULL, 0, output, sizeof output);
    assert(written == 0);
    assert(output[0] == '\0');

    // Positive value to sufficient output writes value without sign
    memcpy(output, "+++", sizeof output);
    written = ints_to_csv(input, 1, output, sizeof output);
    assert(written == 2);
    assert(strcmp(output, "42") == 0);
    ...
}
```

```
void test_ints_to_csv()
{
    // No values from null to null output writes nothing
    {
        size_t written = ints_to_csv(NULL, 0, NULL, 0);
        assert(written == 0);
    }
    // Value to null output writes nothing
    {
        const int input[] = { 42 };
        size_t written = ints_to_csv(input, 1, NULL, 0);
        assert(written == 0);
    }
    // No values to sufficient output writes empty
    {
        char output[3] = "+++";
        size_t written = ints_to_csv(NULL, 0, output, sizeof output);
        assert(written == 0);
        assert(output[0] == '\0');
    }
    // Positive value to sufficient output writes value without sign
    {
        const int input[] = { 42 };
        char output[3] = "+++";
        size_t written = ints_to_csv(input, 1, output, sizeof output);
        assert(written == 2);
        assert(strcmp(output, "42") == 0);
    }
```

```c
void No_values_from_null_to_null_output_writes_nothing()
{
    size_t written = ints_to_csv(NULL, 0, NULL, 0);
    assert(written == 0);
}
void Value_to_null_output_writes_nothing()
{
    const int input[] = { 42 };
    size_t written = ints_to_csv(input, 1, NULL, 0);
    assert(written == 0);
}
void No_values_to_sufficient_output_writes_empty()
{
    char output[3] = "+++";
    size_t written = ints_to_csv(NULL, 0, output, sizeof output);
    assert(written == 0);
    assert(output[0] == '\0');
}
void Positive_value_to_sufficient_output_writes_value_without_sign()
{
    const int input[] = { 42 };
    char output[3] = "+++";
    size_t written = ints_to_csv(input, 1, output, sizeof output);
    assert(written == 2);
    assert(strcmp(output, "42") == 0);
}
void Negative_value_to_sufficient_output_writes_value_with_sign()
{
    const int input[] = { -42 };
    char output[4] = "++++";
    size_t written = ints_to_csv(input, 1, output, sizeof output);
    assert(written == 3);
    assert(strcmp(output, "-42") == 0);
}
void Value_to_insufficient_output_writes_truncated_value()
{
    const int input[] = { 42 };
    char output[2] = "++";
    size_t written = ints_to_csv(input, 1, output, sizeof output);
    assert(written == 1);
    assert(strcmp(output, "4") == 0);
}
void Multiple_values_to_sufficient_output_writes_comma_separated_values()
{
    const int input[] = { 42, -273, 0, 7 };
    char output[12] = "++++++++++++";
    size_t written = ints_to_csv(input, 4, output, sizeof output);
    assert(written == 11);
    assert(strcmp(output, "42,-273,0,7") == 0);
}
void Multiple_values_to_insufficient_output_writes_truncated_value_sequence()
{
    const int input[] = { 42, -273, 0, 7 };
    char output[9] = "+++++++++";
    size_t written = ints_to_csv(input, 4, output, sizeof output);
    assert(written == 8);
    assert(strcmp(output, "42,-273,") == 0);
}
```

```
void No_values_from_null_to_null_output_writes_nothing()
{
    ...
}
void Value_to_null_output_writes_nothing()
{
    ...
}
void No_values_to_sufficient_output_writes_empty()
{
    ...
}
void Positive_value_to_sufficient_output_writes_value_without_sign()
{
    ...
}
void Negative_value_to_sufficient_output_writes_value_with_sign()
{
    ...
}
void Value_to_insufficient_output_writes_truncated_value()
{
    ...
}
void Multiple_values_to_sufficient_output_writes_comma_separated_values()
{
    ...
}
void Multiple_values_to_insufficient_output_writes_truncated_value_sequence()
{
    ...
}
```



test

test

test

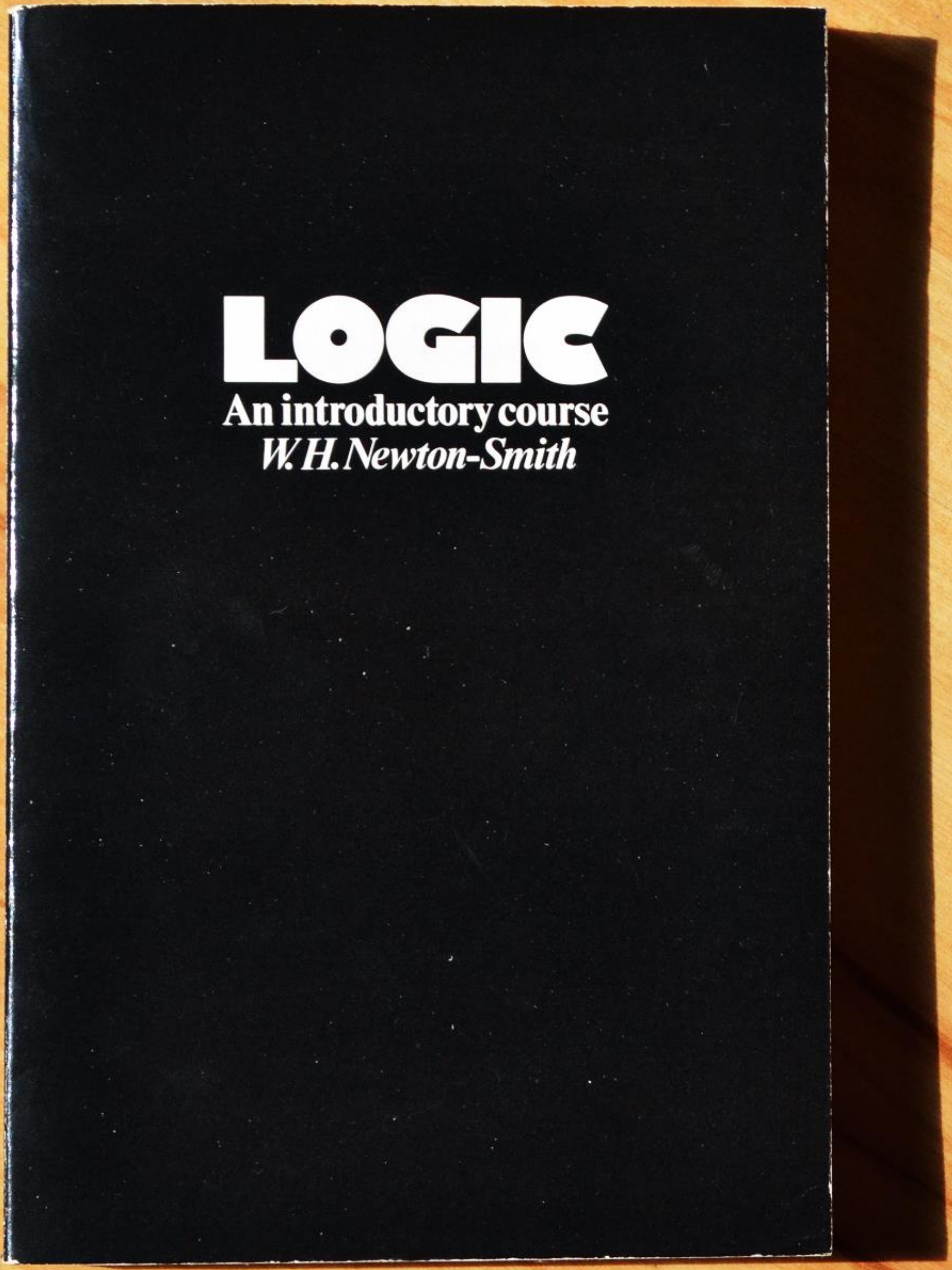
function

```
size_t ints_to_csv(
    const int * to_write, size_t how_many,
    char * output, size_t length);
```

- *No values from null to null output writes nothing*
- *Value to null output writes nothing*
- *No values to sufficient output writes empty*
- *Positive value to sufficient output writes value without sign*
- *Negative value to sufficient output writes value with sign*
- *Value to insufficient output writes truncated value*
- *Multiple values to sufficient output writes comma separated values*
- *Multiple values to insufficient output writes truncated value sequence*

Tests that are not written with their role as specifications in mind can be very confusing to read. The difficulty in understanding what they are testing can greatly reduce the velocity at which a codebase can be changed.

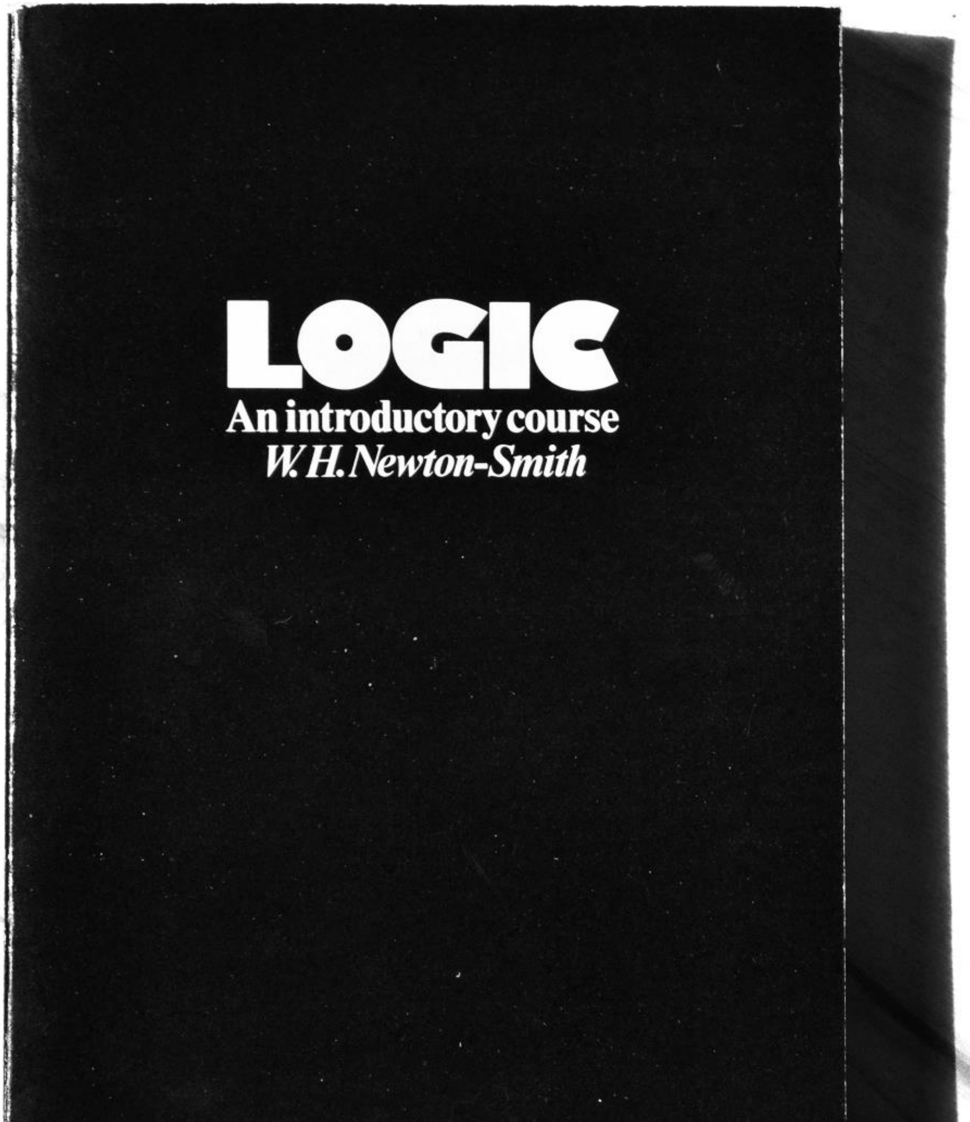
Nat Pryce and Steve Freeman
"Are Your Tests Really Driving Your Development?"



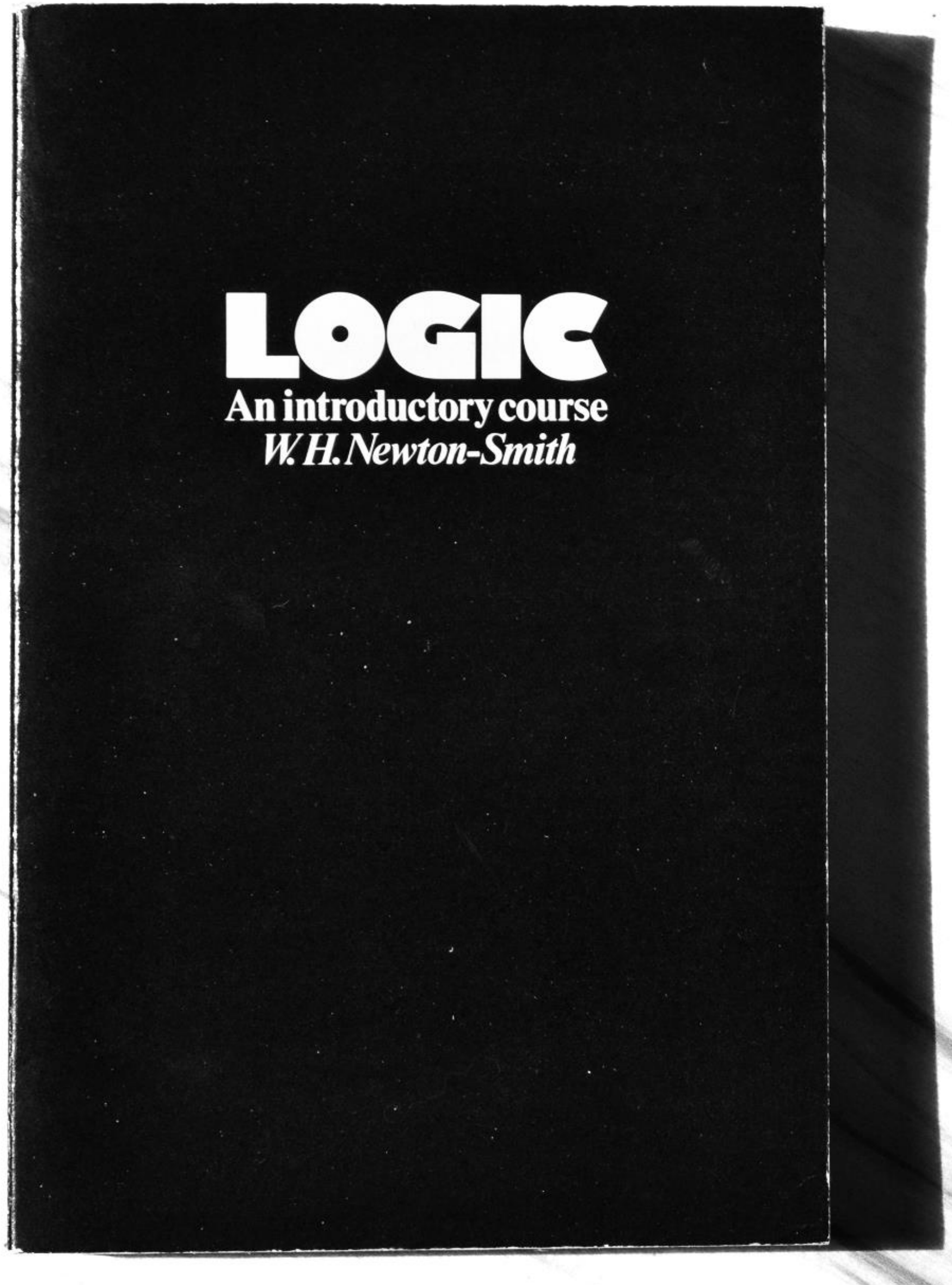
LOGIC

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.

```
public static bool IsLeapYear(int year) ...
```

```
YearsNotDivisibleBy4...

YearsDivisibleBy4ButNotBy100...

YearsDivisibleBy100ButNotBy400...

YearsDivisibleBy400...
```

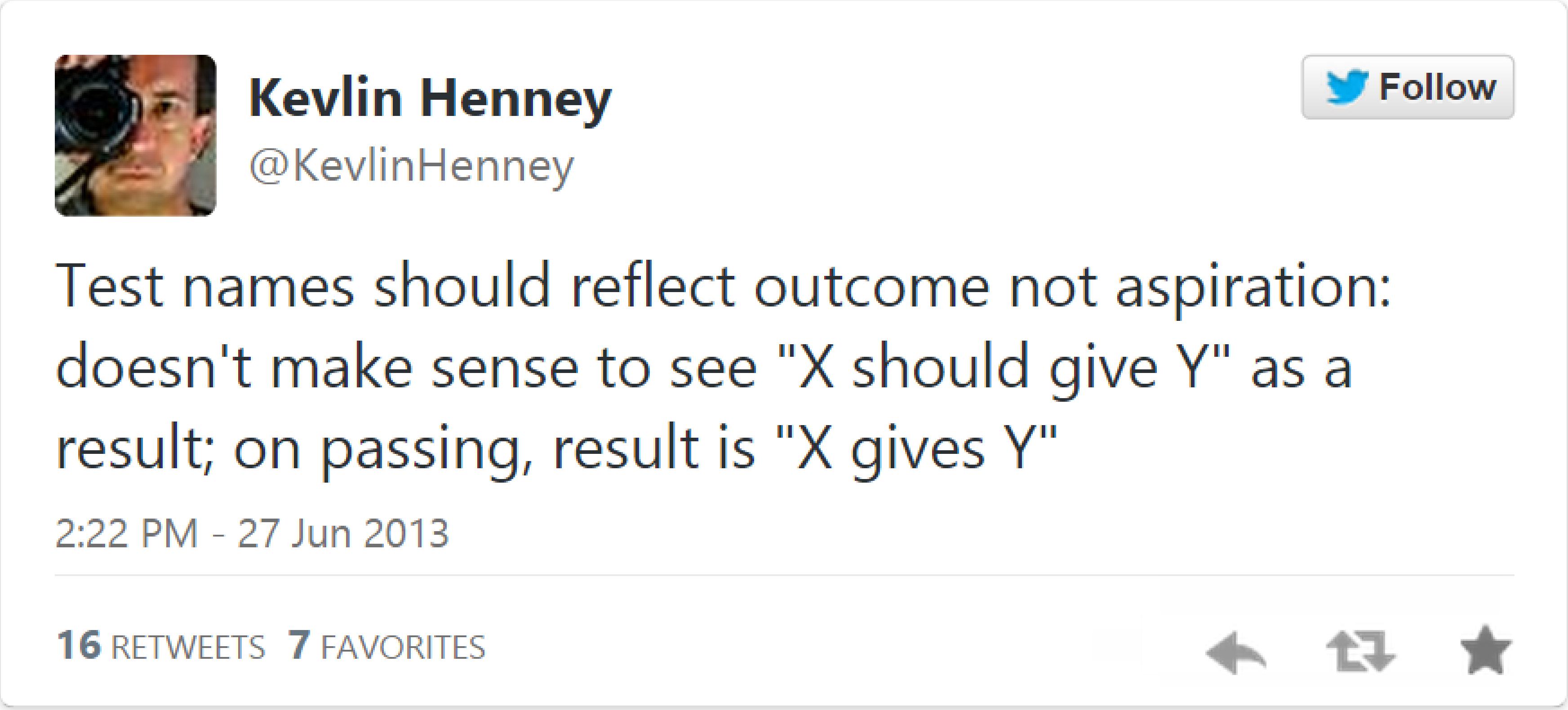
```
Years_not_divisible_by_4_...

Years_divisible_by_4_but_not_by_100_...

Years_divisible_by_100_but_not_by_400_...

Years_divisible_by_400_...
```

```
Years_not_divisible_by_4_should_not_be_leap_years

Years_divisible_by_4_but_not_by_100_should_be_leap_years

Years_divisible_by_100_but_not_by_400_should_not_be_leap_years

Years_divisible_by_400_should_be_leap_years
```

Make definite assertions. Avoid tame, colourless, hesitating, noncommittal language.

Note [...] that when a sentence is made stronger, it usually becomes shorter. Thus brevity is a by-product of vigour.

William Strunk and E B White
The Elements of Style



Kevlin Henney
@KevlinHenney

Test names should reflect outcome not aspiration: doesn't make sense to see "X should give Y" as a result; on passing, result is "X gives Y"

2:22 PM - 27 Jun 2013

16 RETWEETS **7** FAVORITES

https://twitter.com/KevlinHenney/status/350242801868484608

```
Years_not_divisible_by_4_are_not_leap_years

Years_divisible_by_4_but_not_by_100_are_leap_years

Years_divisible_by_100_but_not_by_400_are_not_leap_years

Years_divisible_by_400_are_leap_years
```

```
Years_not_divisible_by_4_are_not_leap_years

Years_divisible_by_4_but_not_by_100_are_leap_years

Years_divisible_by_100_but_not_by_400_are_not_leap_years

Years_divisible_by_400_are_leap_years
```

Years_not_divisible_by_4_are_not_leap_years

Years_divisible_by_4_but_not_by_100_are_leap_years

Years_divisible_by_100_but_not_by_400_are_not_leap_years

Years_divisible_by_400_are_leap_years

A test case should be just that: it should correspond to a single case.

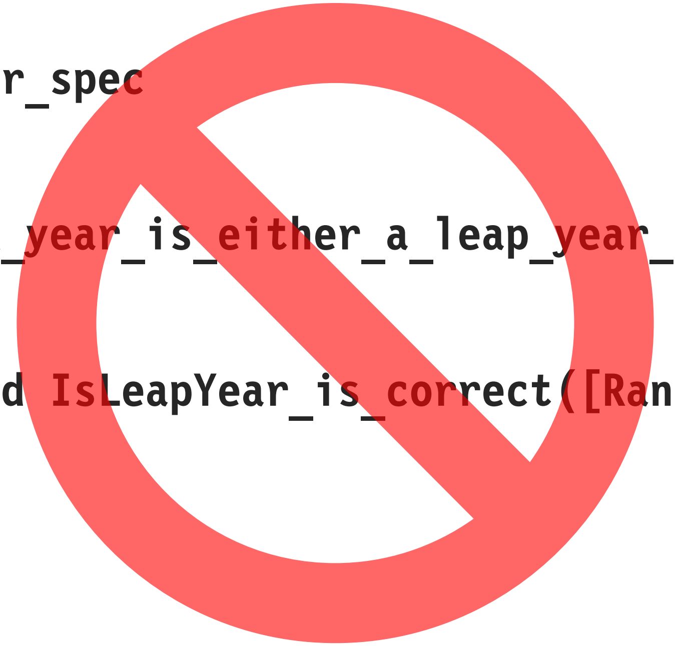
```
namespace Leap_year_spec
{
    [TestFixture]
    public class A_year_is_a_leap_year
    {
        [Test] public void If_it_is_divisible_by_4_but_not_by_100(...) ...
        [Test] public void If_it_is_divisible_by_400(...) ...
    }
    [TestFixture]
    public class A_year_is_not_a_leap_year
    {
        [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
{
    [TestFixture]
    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(...) ...
    }
    [TestFixture]
    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
{
    [TestFixture]
    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(…) …
    }
    [TestFixture]
    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
{
    [TestFixture]
    public class A_year_is_a_leap_year
    {
        [Test]
        public void If_it_is_divisible_by_4_but_not_by_100(
            [Values(2012, 1984, 4)] int year)
        {
            Assert.IsTrue(IsLeapYear(year));
        }
        [Test]
        public void If_it_is_divisible_by_400(
            [Range(400, 2400, 400)] int year)
        {
            Assert.IsTrue(IsLeapYear(year));
        }
    }
    [TestFixture]
    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
{
    [TestFixture]
    public class A_year_is_a_leap_year
    {
        [Test]
        public void If_it_is_divisible_by_4_but_not_by_100(
            [Values(2012, 1984, 4)] int year)
        {
            Assert.IsTrue(IsLeapYear(year));
        }
        [Test]
        public void If_it_is_divisible_by_400(
            [Range(400, 2400, 400)] int year)
        {
            Assert.IsTrue(IsLeapYear(year));
        }
    }
    [TestFixture]
    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
{
    [TestFixture]
    public class A_year_is_either_a_leap_year_or_not
    {
        [Test]
        public void IsLeapYear_is_correct([Range(1, 10000)] int year) ...
    }
}
```

```
namespace Leap_year_spec
{
    [TestFixture]
    public class A_year_is_either_a_leap_year_or_not
    {
        [Test]
        public void IsLeapYear_is_correct([Range(1, 10000)] int year)
        {
            Assert.AreEqual(
                year % 4 == 0 && year % 100 != 0 || year % 400 == 0,
                IsLeapYear(year));
        }
    }
}
```

```
namespace Leap_year_spec
{
    [TestFixture]
    public class A_year_is_either_a_leap_year_or_not
    {
        [Test]
        public void IsLeapYear_is_correct([Range(1, 10000)] int year)
        {
            Assert.AreEqual(LeapYearExpectation(year), IsLeapYear(year));
        }
        public static bool LeapYearExpectation(int year)
        {
            return year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
        }
    }
}
```

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

All happy families are alike; each unhappy family is unhappy in its own way.

Leo Tolstoy
Anna Karenina

```
proposition isbn_spec[] =
{
    ...
    "Test validation", []
    {
        CATCH(isbn("97805968094857"), isbn::malformed);
        CATCH(isbn("978059680948"), isbn::malformed);
        CATCH(isbn("978Q5968=9485"), isbn::malformed);
        CATCH(isbn("9780596809486"), isbn::malformed);
    },
    ...
};
```

```
struct proposition
{
    std::string name;
    std::function<void()> run;
};

struct failure
{
    const char * expression;
    int line;
};

template<typename Propositions>
void test(const Propositions & to_test)
{
    for(auto & test : to_test)
    {
        try
        {
            std::cout << test.name << std::flush;
            test.run();
            std::cout << "\n";
        }
        catch(failure & caught)
        {
            std::cout << " failed:\n  " << caught.expression << "\n  at line " << caught.line << "\n";
        }
    }
}

#define ASSERT(condition) void((condition) ? 0 : throw failure({ "ASSERT(" #condition ")", __LINE__ }))

#define CATCH(expression, exception) \
    try \
    { \
        (expression); \
        throw failure({ "CATCH(" #expression ", " #exception ")", __LINE__ }); \
    } \
    catch (exception &) \
    { \
    } \
    catch (...) \
    { \
        throw failure({ "CATCH(" #expression ", " #exception ")", __LINE__ }); \
    }
```

```
proposition isbn_spec[] =
{
    ...
    "Test validation", []
    {
        CATCH(isbn("97805968094857"), isbn::malformed);
        CATCH(isbn("978059680948"), isbn::malformed);
        CATCH(isbn("978Q5968=9485"), isbn::malformed);
        CATCH(isbn("9780596809486"), isbn::malformed);
    },
    ...
};
```

```
proposition isbn_spec[] =
{
    ...
    "Test validation", []
    {
        CATCH(isbn("97805968094857"), isbn::malformed);
        CATCH(isbn("978059680948"), isbn::malformed);
        CATCH(isbn("978Q5968=9485"), isbn::malformed);
        CATCH(isbn("9780596809486"), isbn::malformed);
    },
    ...
};
```

```
proposition isbn_spec[] =
{
    ...
    "Test validation works", []
    {
        CATCH(isbn("97805968094857"), isbn::malformed);
        CATCH(isbn("978059680948"), isbn::malformed);
        CATCH(isbn("978Q5968=9485"), isbn::malformed);
        CATCH(isbn("9780596809486"), isbn::malformed);
    },
    ...
};
```

```
proposition isbn_spec[] =
{
    ...
    "Test validation works", []
    {
        CATCH(isbn("97805968094857"), isbn::malformed);
        CATCH(isbn("978059680948"), isbn::malformed);
        CATCH(isbn("978Q5968=9485"), isbn::malformed);
        CATCH(isbn("9780596809486"), isbn::malformed);
    },
    ...
};
```

```
proposition isbn_spec[] =
{
    ...
    "ISBNs with more than 13 digits are malformed", []
    {
        CATCH(isbn("97805968094857"), isbn::malformed);
    },
    "ISBNs with fewer than 13 digits are malformed", []
    {
        CATCH(isbn("978059680948"), isbn::malformed);
    },
    "ISBNs with non-digits are malformed", []
    {
        CATCH(isbn("978Q5968=9485"), isbn::malformed);
    },
    "ISBNs with an incorrect check digit are malformed", []
    {
        CATCH(isbn("9780596809486"), isbn::malformed);
    },
    ...
};
```

```
proposition isbn_spec[] =
{
    ...
    "ISBNs with more than 13 digits are malformed", []
    {
        CATCH(isbn("97805968094857"), isbn::malformed);
    },
    "ISBNs with fewer than 13 digits are malformed", []
    {
        CATCH(isbn("978059680948"), isbn::malformed);
    },
    "ISBNs with non-digits are malformed", []
    {
        CATCH(isbn("978Q5968=9485"), isbn::malformed);
    },
    "ISBNs with an incorrect check digit are malformed", []
    {
        CATCH(isbn("9780596809486"), isbn::malformed);
    },
    ...
};
```

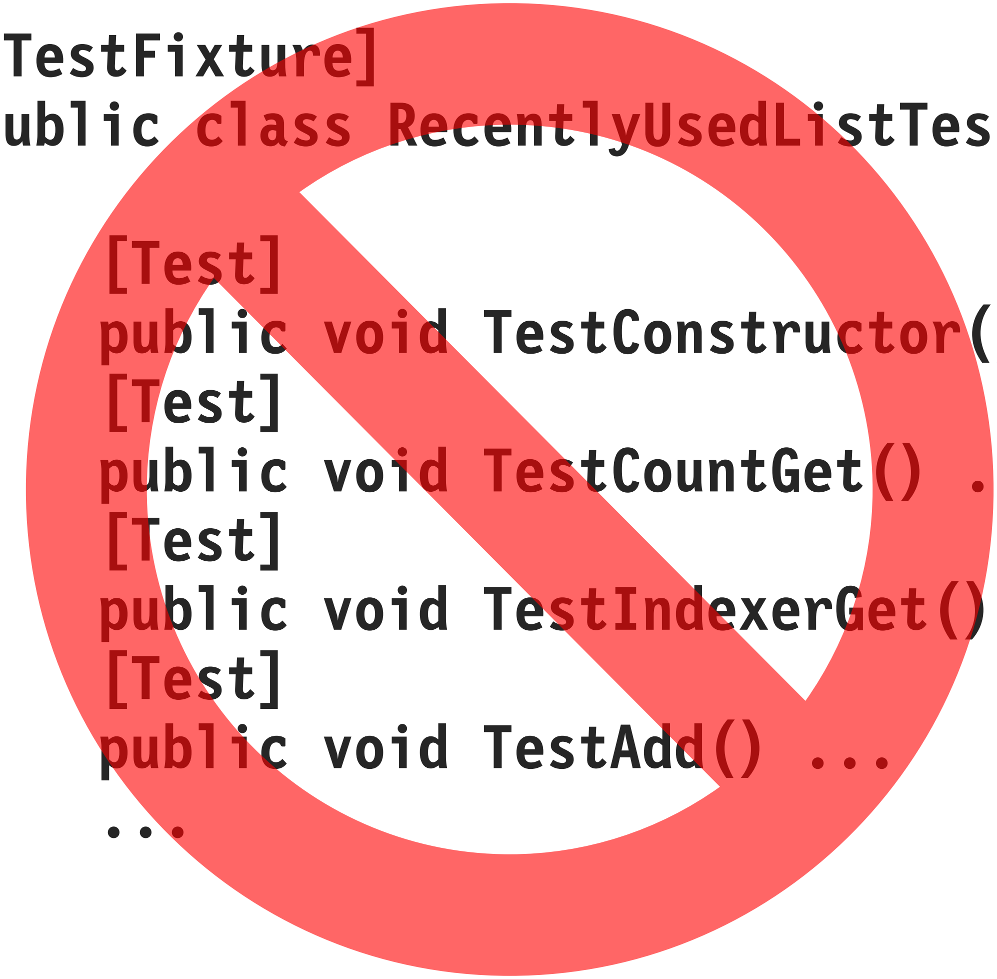
Validation is not a behaviour; the consequence of validation is.

test → method

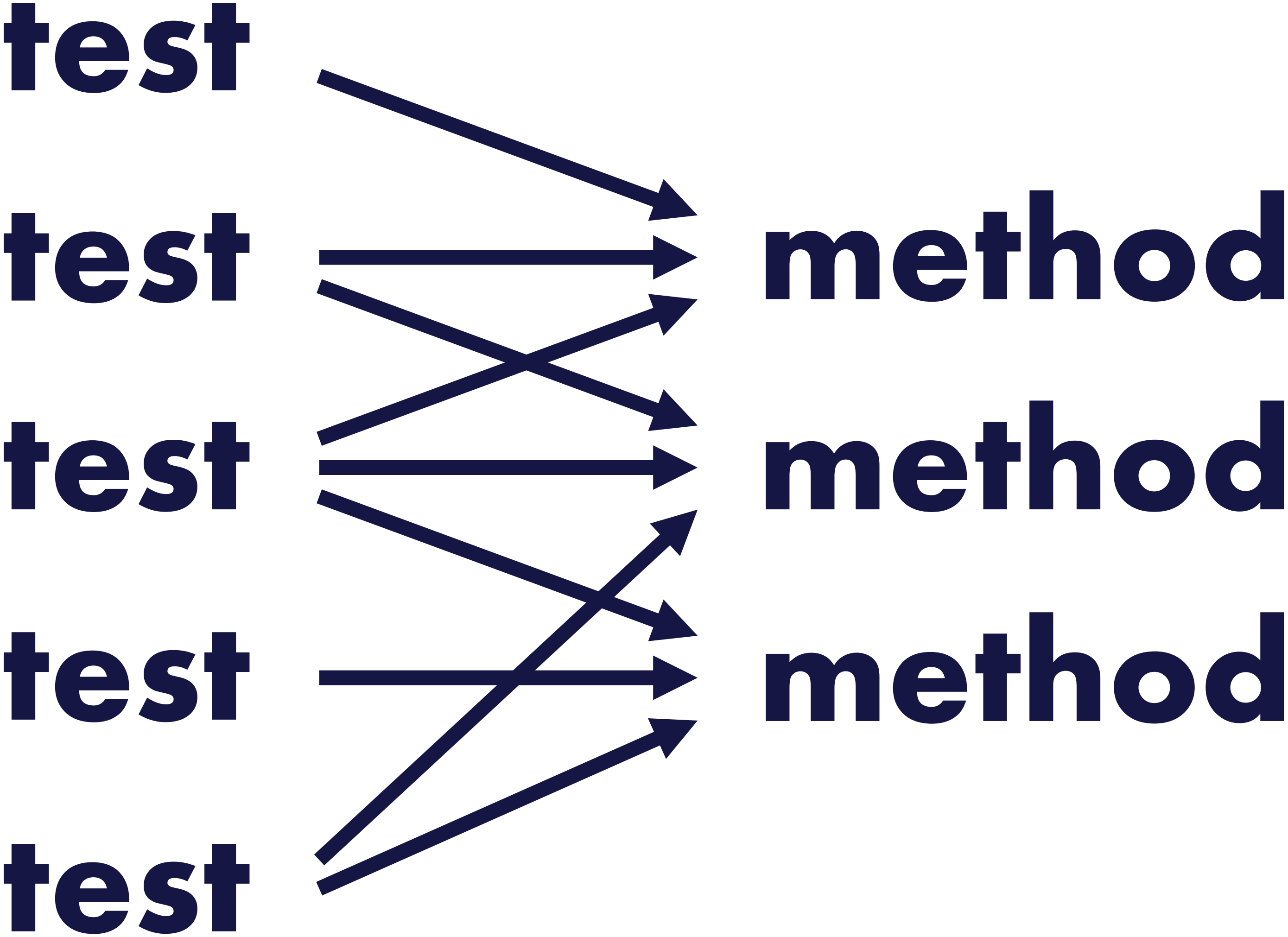
test → method

test → method

```
public class RecentlyUsedList
{
    ...
    public RecentlyUsedList() ...
    public int Count
    {
        get...
    }
    public string this[int index]
    {
        get...
    }
    public void Add(string newItem) ...
    ...
}
```



```
[TestFixture]
public class RecentlyUsedListTests
{
    [Test]
    public void TestConstructor() ...
    [Test]
    public void TestCountGet() ...
    [Test]
    public void TestIndexerGet() ...
    [Test]
    public void TestAdd() ...
    ...
}
```



test

test

test

test

test

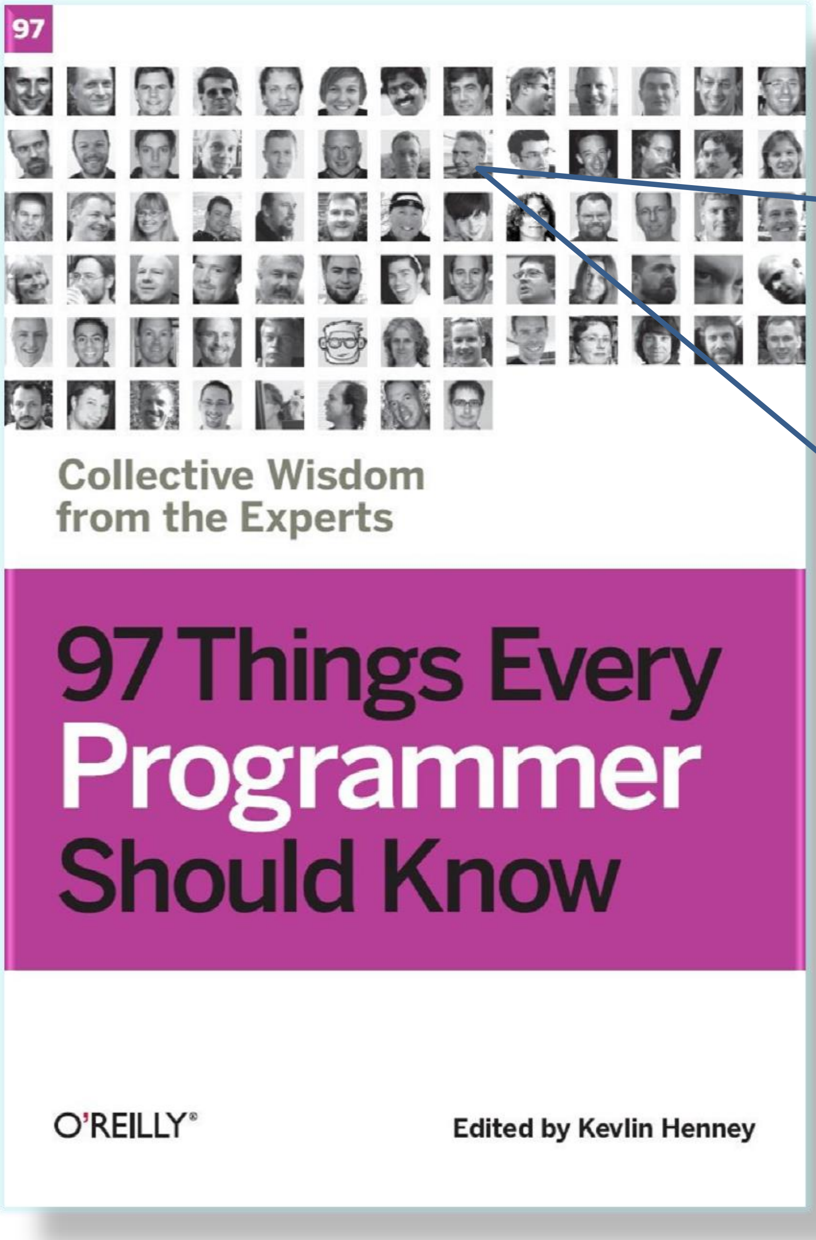
method

method

method

```
namespace RecentlyUsedList_spec
{
    [TestFixture]
    public class A_new_list
    {
        [Test] public void Is_empty() …
    }
    [TestFixture]
    public class An_empty_list
    {
        [Test] public void Retains_a_single_addition() …
        [Test] public void Retains_unique_additions_in_stack_order() …
    }
    [TestFixture]
    public class A_non_empty_list
    {
        [Test] public void Is_unchanged_when_head_item_is_readded() …
        [Test] public void Moves_non_head_item_to_head_when_it_is_readded() …
    }
    [TestFixture]
    public class Any_list_rejects
    {
        [Test] public void Addition_of_null_items() …
        [Test] public void Indexing_past_its_end() …
        [Test] public void Negative_indexing() …
    }
}
```

```
namespace RecentlyUsedList_spec
{
    [TestFixture]
    public class A_new_list
    {
        [Test] public void Is_empty() ...
    }
    [TestFixture]
    public class An_empty_list
    {
        [Test] public void Retains_a_single_addition() ...
        [Test] public void Retains_unique_additions_in_stack_order() ...
    }
    [TestFixture]
    public class A_non_empty_list
    {
        [Test] public void Is_unchanged_when_head_item_is_readded() ...
        [Test] public void Moves_non_head_item_to_head_when_it_is_readded() ...
    }
    [TestFixture]
    public class Any_list_rejects
    {
        [Test] public void Addition_of_null_items() ...
        [Test] public void Indexing_past_its_end() ...
        [Test] public void Negative_indexing() ...
    }
}
```


97
Collective Wisdom from the Experts
97 Things Every Programmer Should Know
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. They describe how the code works. 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

Different usage scenarios will have slightly different versions of each of these.

Gerard Meszaros
"Write Tests for People"

```
namespace RecentlyUsedList_spec
{
    [TestFixture]
    public class A_new_list ...
    [TestFixture]
    public class An_empty_list
    {
        [Test]
        public void Retains_a_single_addition(
            [Values("Oxford", "Bristol", "London")] string addend)
        {
            var items = new RecentlyUsedList();   // Given...

            items.Add(addend);                    // When...

            Assert.AreEqual(1, items.Count);      // Then...
            Assert.AreEqual(addend, list[0]);
        }
        [Test] public void Retains_unique_additions_in_stack_order() ...
    }
    [TestFixture]
    public class A_non_empty_list ...
    [TestFixture]
    public class Any_list_rejects ...
}
```

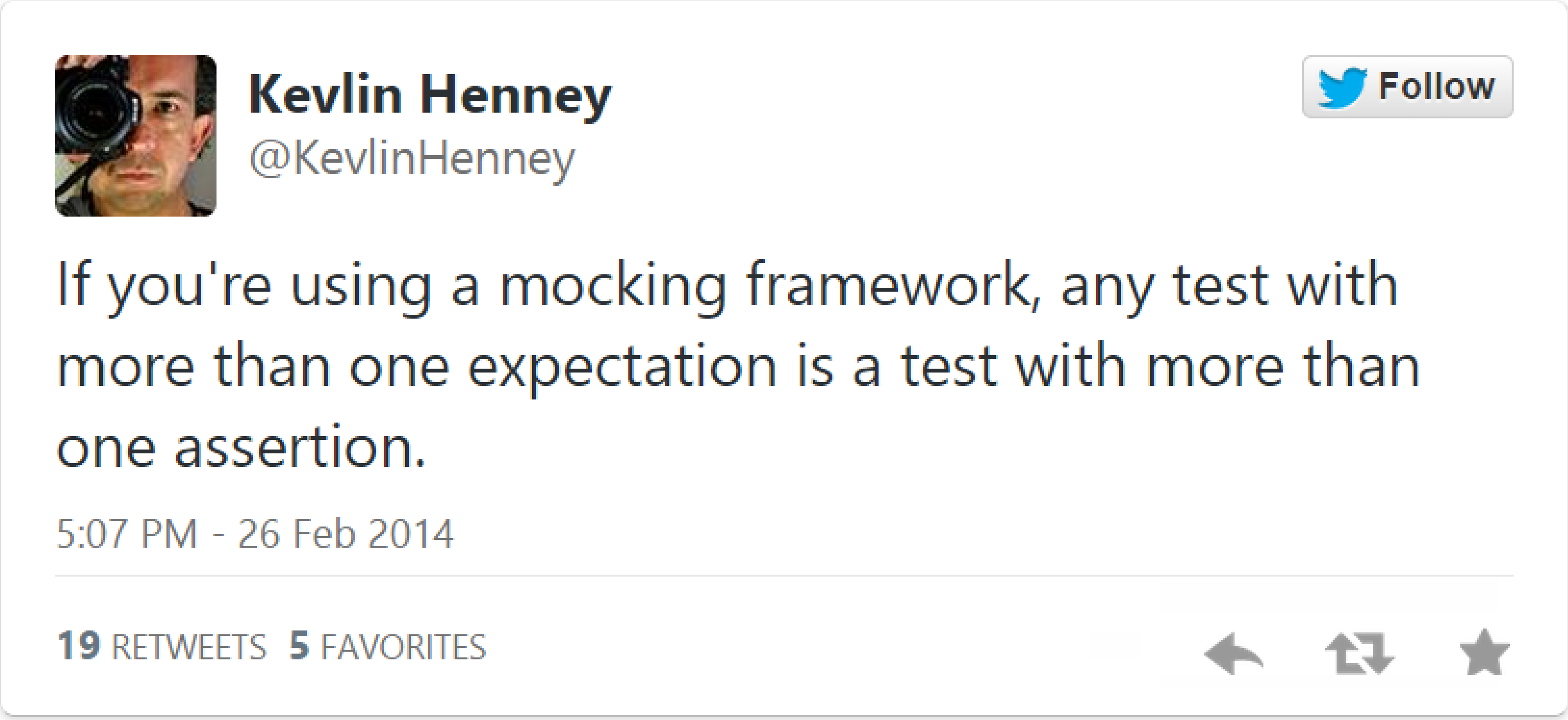
One of the things that Osherove warns against is multiple asserts in unit tests.

Owen Pellegrin
http://www.owenpellegrin.com/blog/testing/how-do-you-solve-multiple-asserts/

Proper unit tests should fail for exactly one reason, that’s why you should be using one assert per unit test.

http://rauchy.net/oapt/

```
string[] itinerary = ...;
string[] expected =
{
    "London", "Bristol", "Oslo"
};
Assert.AreEqual(expected, itinerary);
Assert.DoesNotThrow(() =>
{
    string[] itinerary = ...;
    string[] expected = ...;

    Assert.IsNotNull(itinerary);
    Assert.AreEqual(3, itinerary.Length);
    Assert.AreEqual("London", itinerary[0]);
    Assert.AreEqual("Bristol", itinerary[1]);
    Assert.AreEqual("Oslo", itinerary[2]);
});
```



Kevlin Henney
@KevlinHenney

If you're using a mocking framework, any test with more than one expectation is a test with more than one assertion.

5:07 PM - 26 Feb 2014

19 RETWEETS **5** FAVORITES

https://twitter.com/KevlinHenney/status/438707024067825664

```
new Expectations()
{{
    ...
}}
```

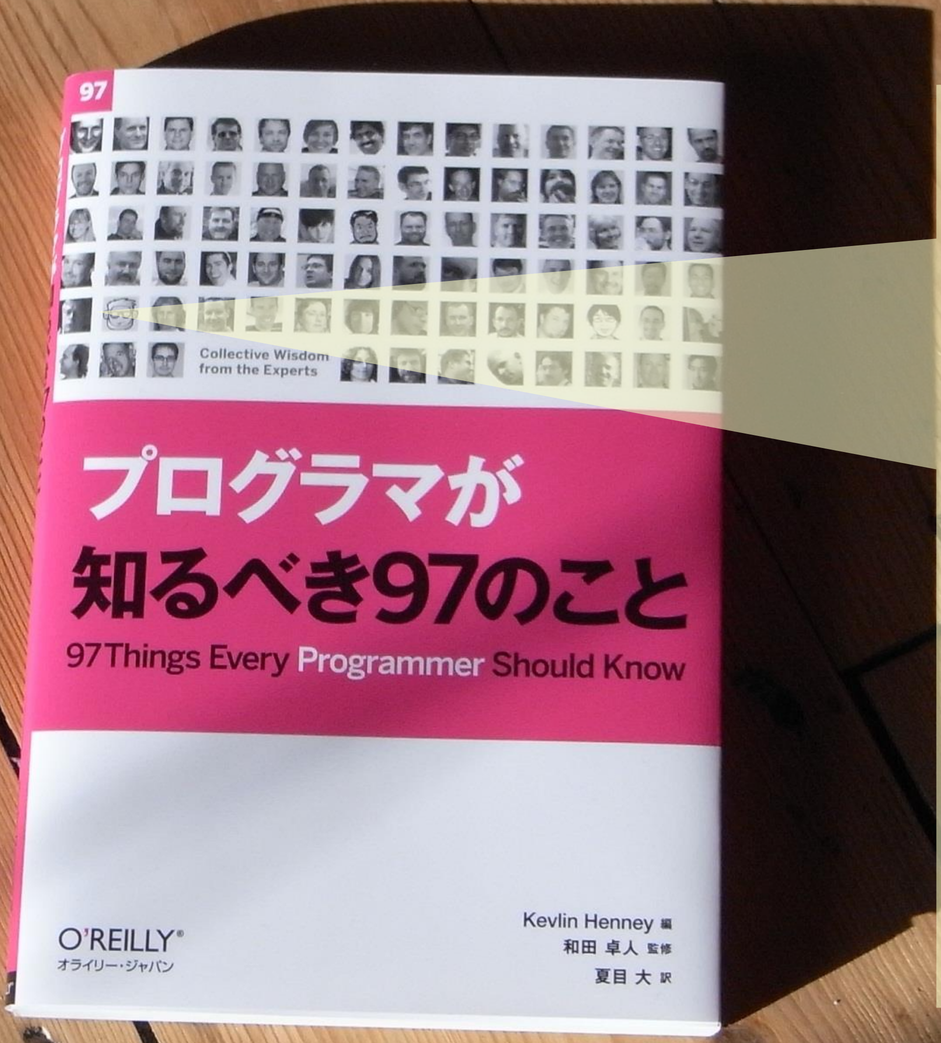
```
@Test(expected=...)
public void ...()
{
    ...
}
```

```
def ...()
{
    ...
    expect:
    ...
}
```

My guideline is usually that you test one logical *concept* per test. You can have multiple asserts on the same *object*. They will usually be the same concept being tested.

Roy Osherove

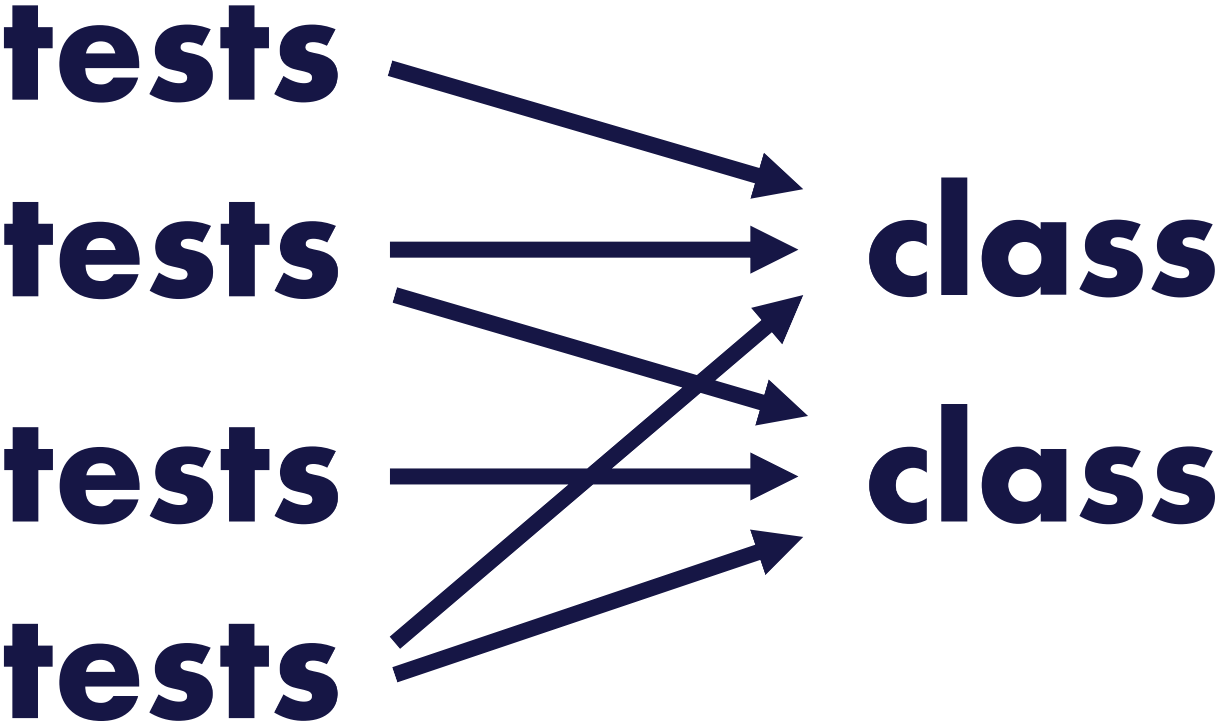
http://www.owenpellegrin.com/blog/testing/how-do-you-solve-multiple-asserts/


97
Collective Wisdom from the Experts
プログラマが知るべき97のこと
97 Things Every Programmer Should Know
O'REILLY®
オライリー・ジャパン
Kevlin Henney 編
和田 卓人 監修
夏目 大 訳

One of the most foundational principles of good design is:

> Gather together those things that change for the same reason, and separate those things that change for different reasons.

This principle is often known as the *single responsibility principle*, or SRP. In short, it says that a subsystem, module, class, or even a function, or a test should not have more than one reason to change.



tests

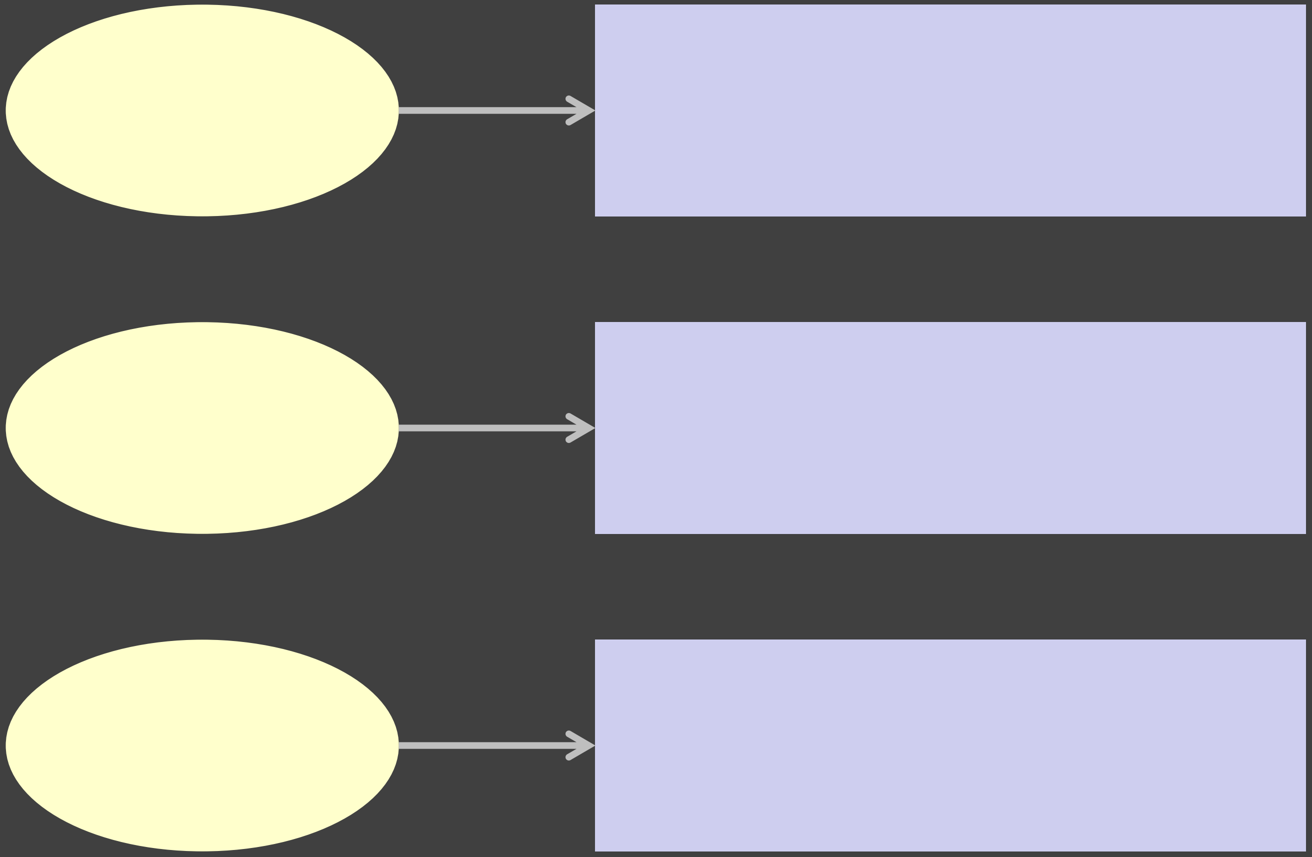
tests

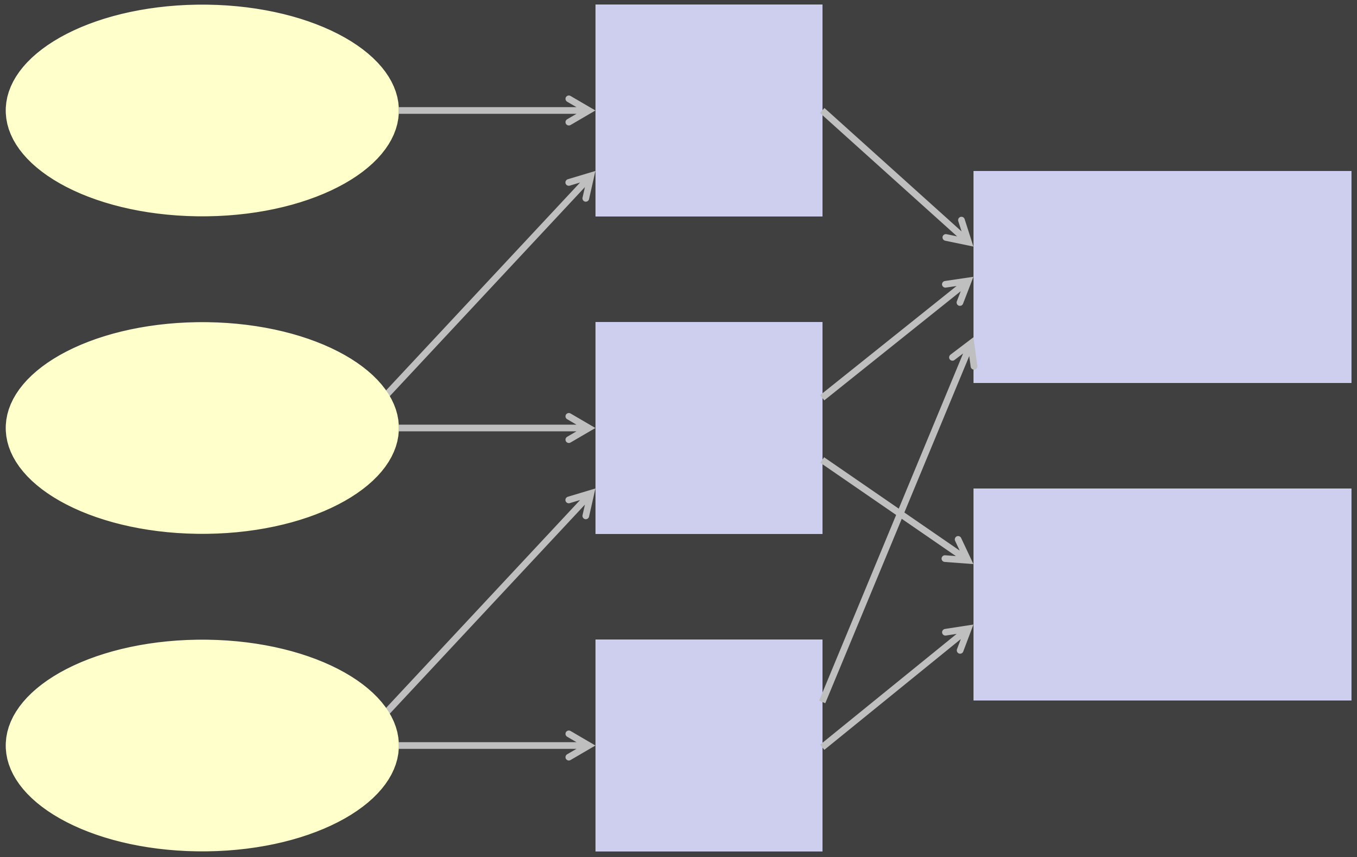
tests

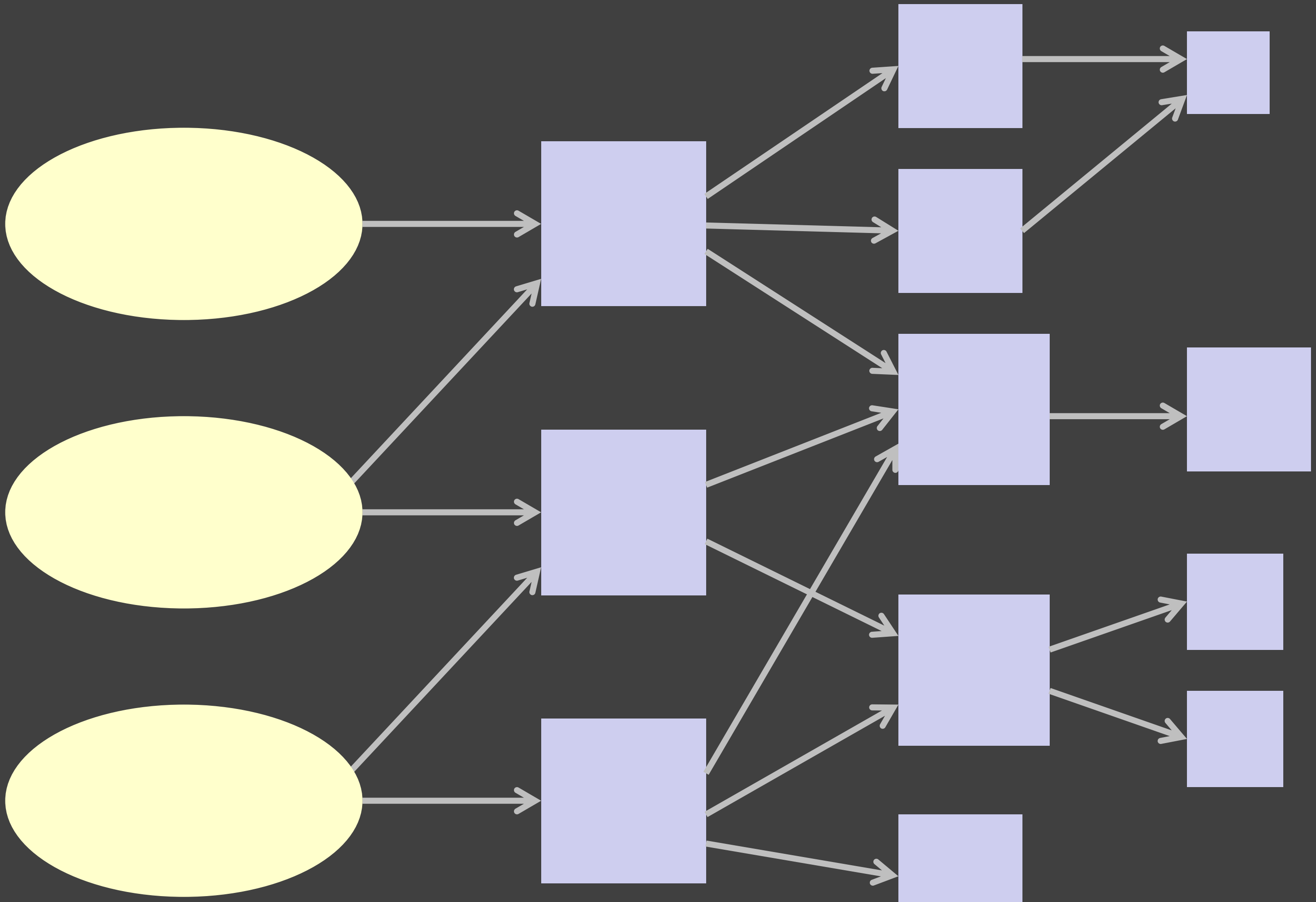
tests

class

class







A test is not a unit test if:

- **It talks to the database**
- **It communicates across the network**
- **It touches the file system**
- **It can't run at the same time as any of your other unit tests**
- **You have to do special things to your environment (such as editing config files) to run it.**

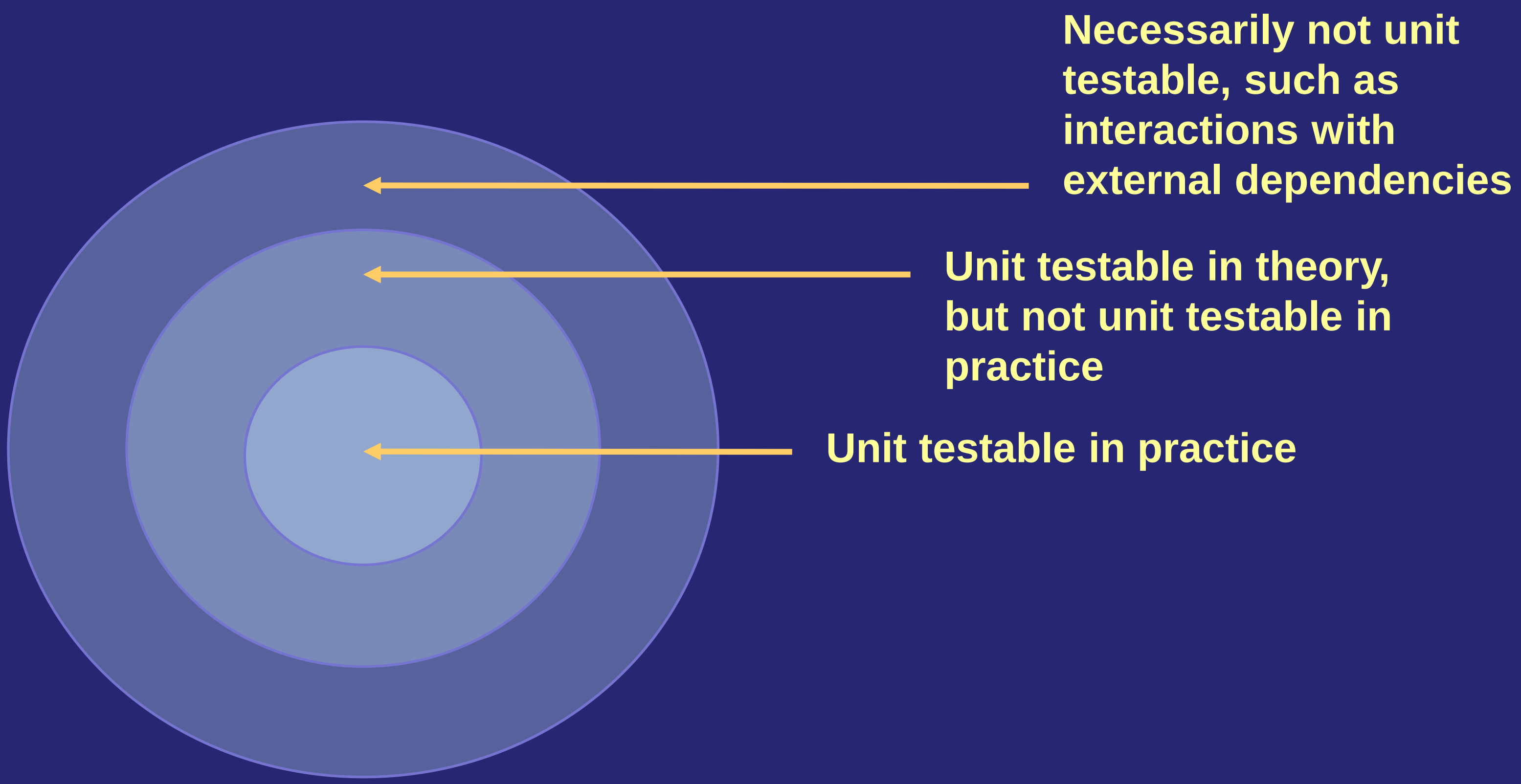
Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes.

Michael Feathers
"A Set of Unit Testing Rules"
http://www.artima.com/weblogs/viewpost.jsp?thread=126923

A unit test is a test of behaviour whose success or failure is wholly determined by the correctness of the test and the correctness of the unit under test.

Kevlin Henney

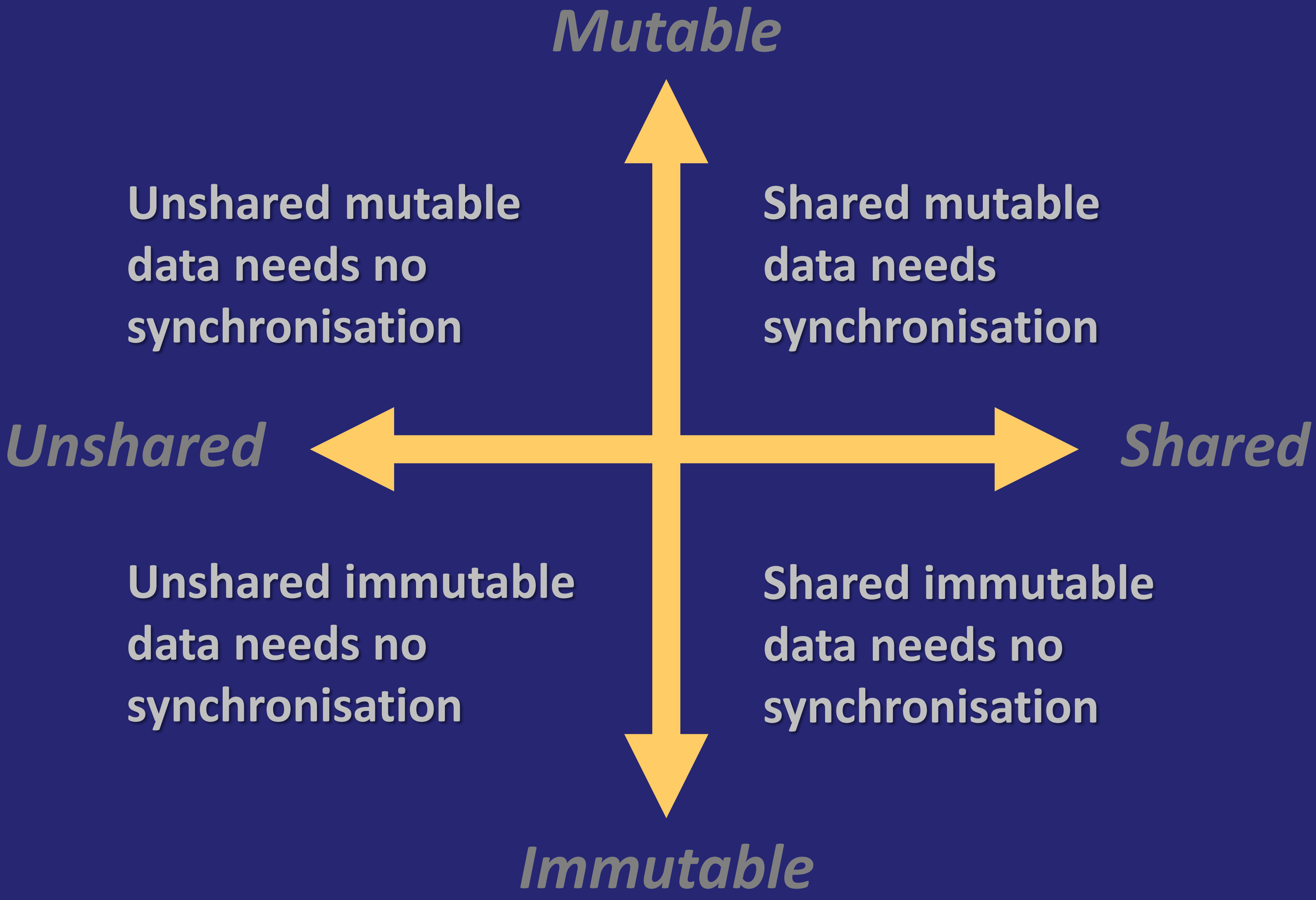
http://www.theregister.co.uk/2007/07/28/what_are_your_units/



Necessarily not unit testable, such as interactions with external dependencies

Unit testable in theory, but not unit testable in practice

Unit testable in practice



Mutable

Unshared mutable data needs no synchronisation

Shared mutable data needs synchronisation

Unshared

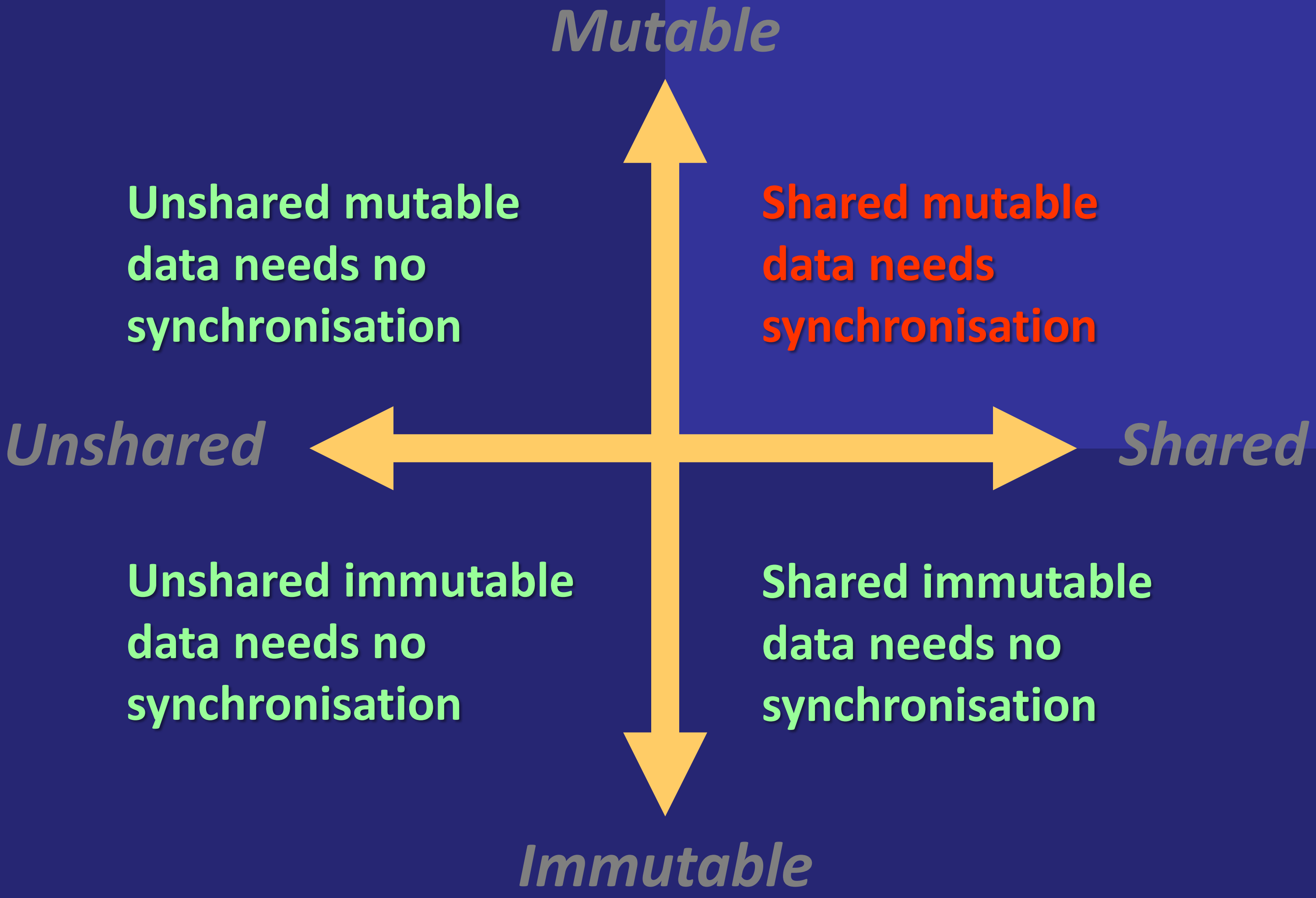
Shared

Unshared immutable data needs no synchronisation

Shared immutable data needs no synchronisation

Immutable

The Synchronisation Quadrant



Mutable

Unshared mutable data needs no synchronisation

Shared mutable data needs synchronisation

Unshared

Shared

Unshared immutable data needs no synchronisation

Shared immutable data needs no synchronisation

Immutable

The real value of tests is not that they detect bugs in the code but that they detect inadequacies in the methods, concentration, and skills of those who design and produce the code.

C A R Hoare