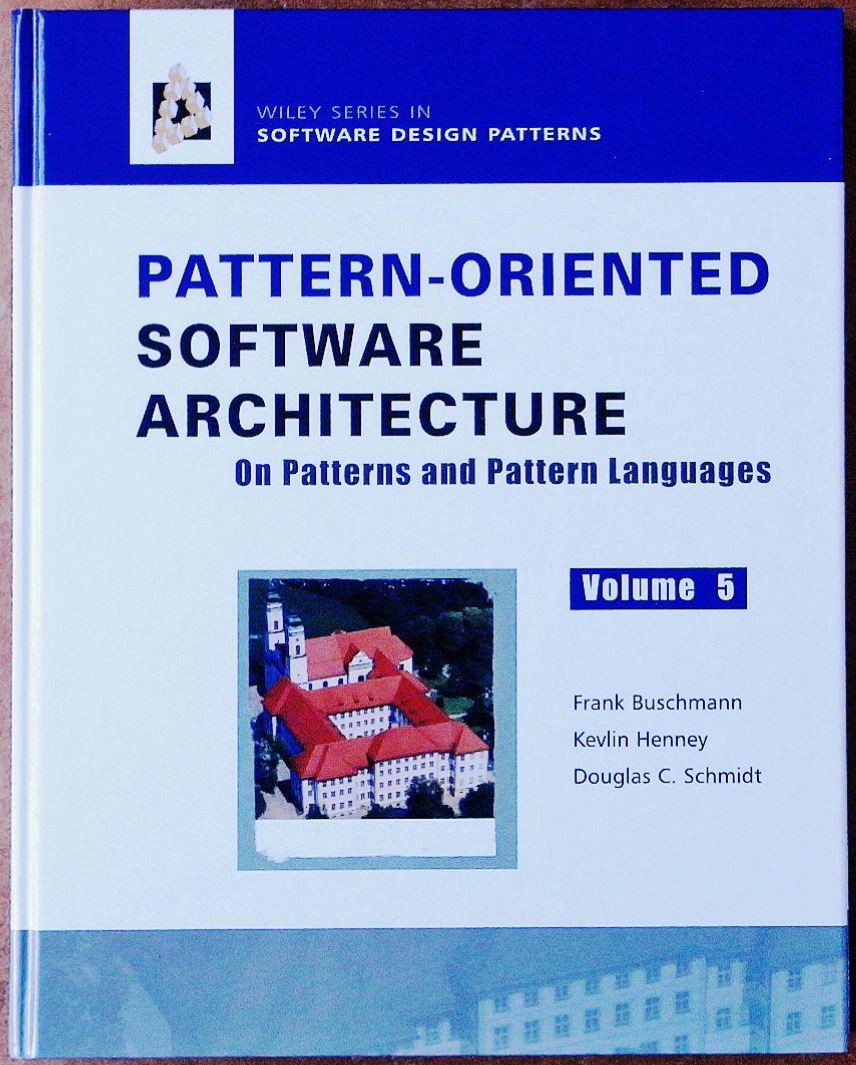
Fizzbuzzalooza

@KevlinHenney



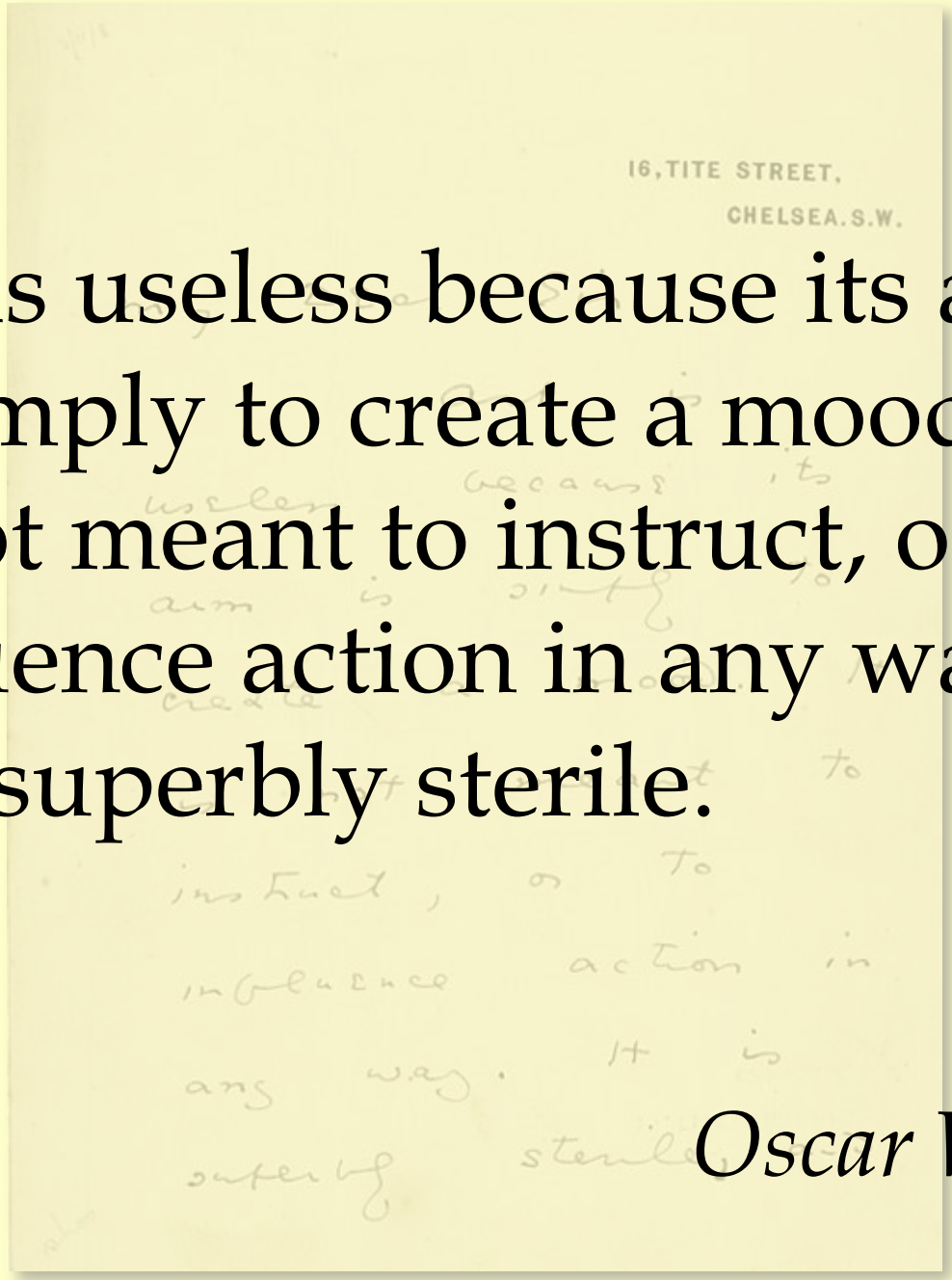
Art. Craft. Engineering. Science. These are the swirling muses of design patterns. Art and science are stories; craft and engineering are actions.

Wayne Cool

16, TITE STREET,
CHELSEA. S.W.

My Dear Sir

Art is useless because its aim is simply to create a mood. It is not meant to instruct, or to influence action in any way. It is utterly sterile, and

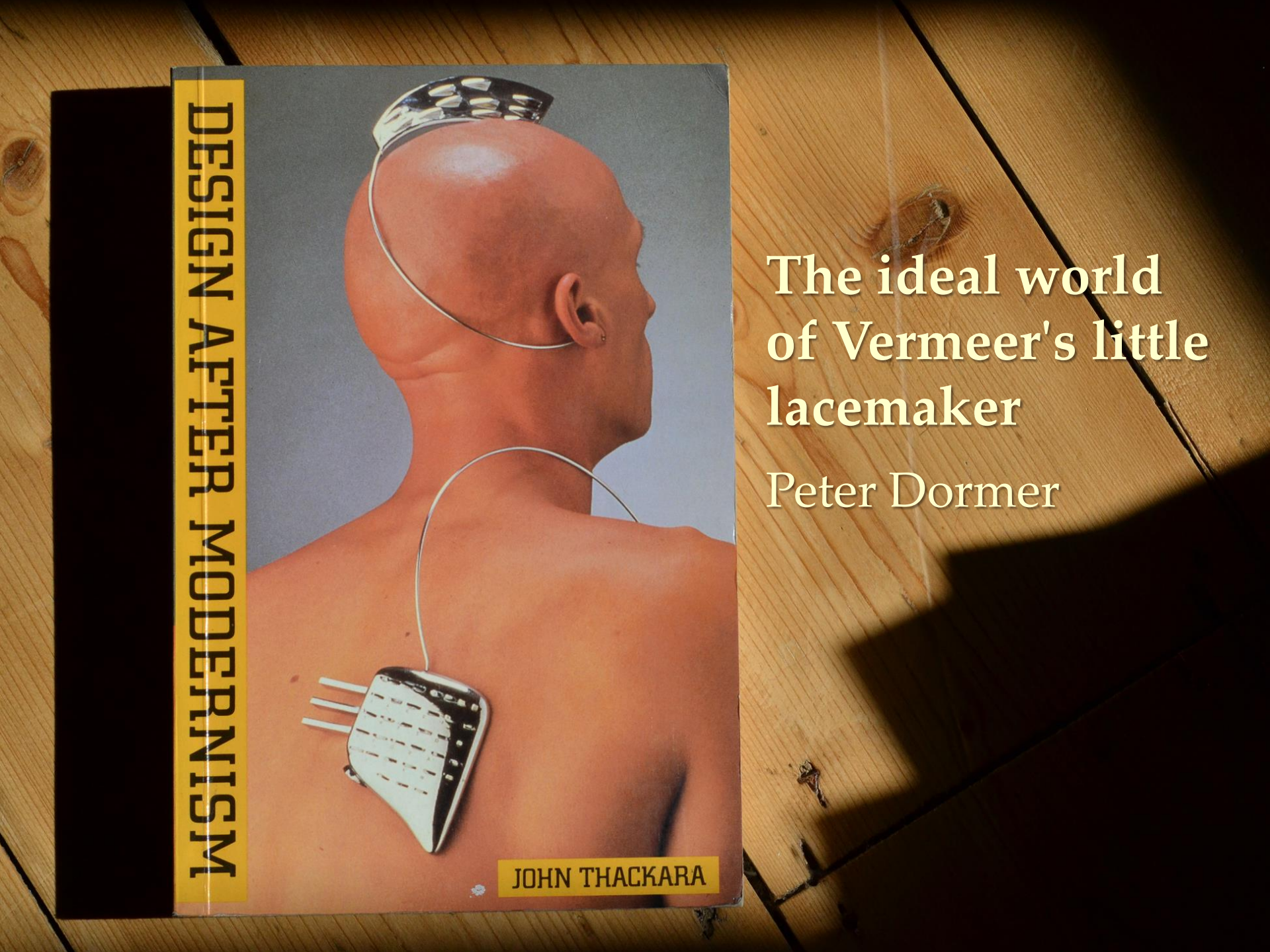


Art is useless because its aim is simply to create a mood. It is not meant to instruct, or to influence action in any way. It is superbly sterile.

Oscar Wilde

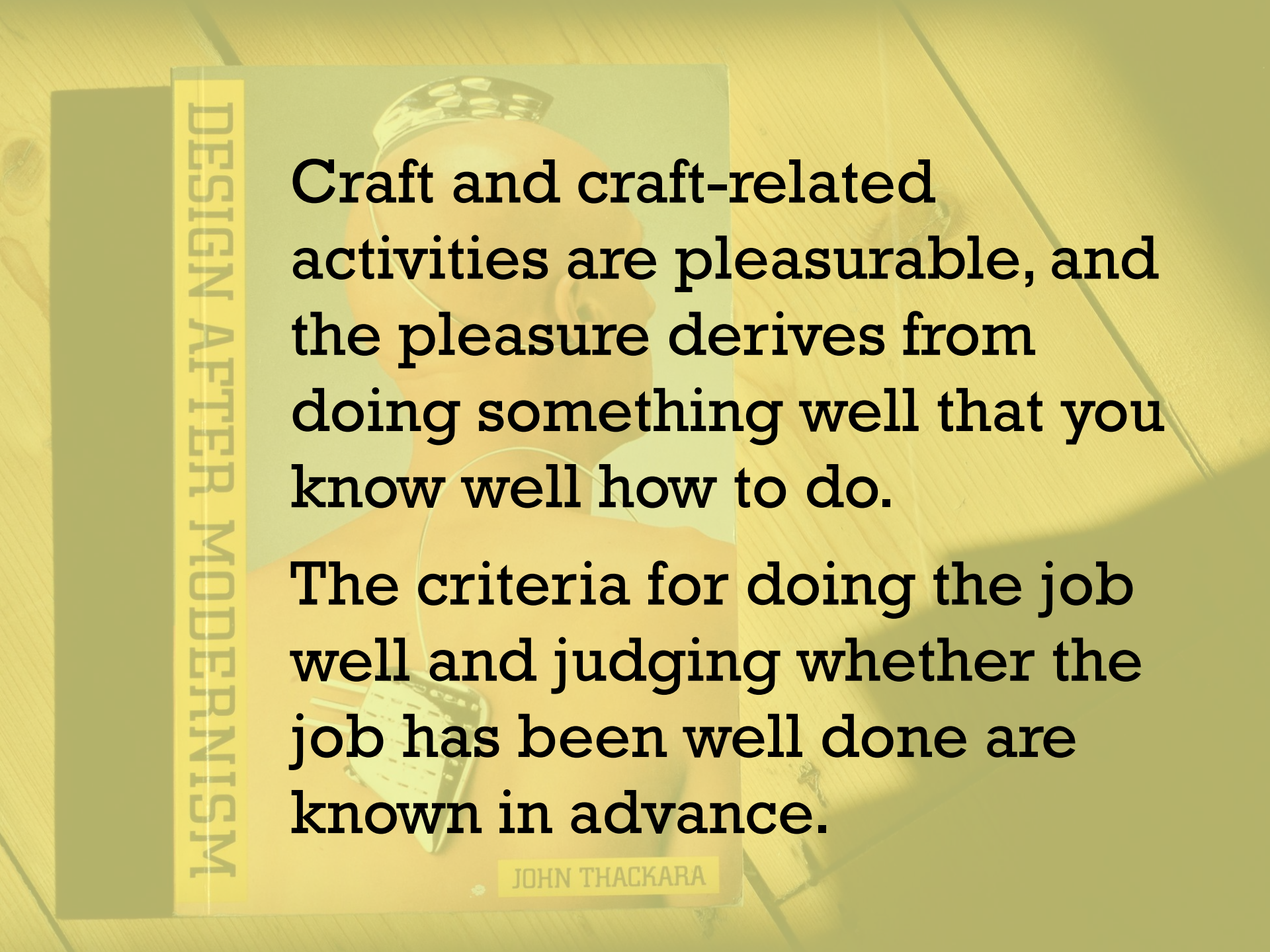
Skill without imagination is craftsmanship and gives us many useful objects such as wickerwork picnic baskets. Imagination without skill gives us modern art.

Tom Stoppard



The ideal world of Vermeer's little lacemaker

Peter Dormer



Craft and craft-related activities are pleasurable, and the pleasure derives from doing something well that you know well how to do.

The criteria for doing the job well and judging whether the job has been well done are known in advance.

Fizz buzz is a group word game for children to teach them about division.

http://en.wikipedia.org/wiki/Fizz_buzz

Players generally sit in a circle. The player designated to go first says the number "1", and each player thenceforth counts one number in turn. However, any number divisible by three is replaced by the word *fizz* and any divisible by five by the word *buzz*. Numbers divisible by both become *fizz buzz*. A player who hesitates or makes a mistake is eliminated from the game.

http://en.wikipedia.org/wiki/Fizz_buzz

Players generally sit in a circle. The player designated to go first says the number "1", and each player thenceforth counts one number in turn. However, **any number divisible by three** is replaced by the word ***fizz*** and **any divisible by five** by the word ***buzz***. Numbers **divisible by both** become ***fizz buzz***. A player who hesitates or makes a mistake is eliminated from the game.

http://en.wikipedia.org/wiki/Fizz_buzz

Players generally sit in a circle. The player designated to go first says the number "1", and each player **thenceforth** counts one number in turn. However, any number divisible by three is replaced by the word *fizz* and any divisible by five by the word *buzz*. Numbers divisible by both become *fizz buzz*. A player who hesitates or makes a mistake is eliminated from the game.

http://en.wikipedia.org/wiki/Fizz_buzz

Adults may play Fizz buzz as a drinking game, where making a mistake leads to the player having to make a drinking-related forfeit. [*citation needed*]

http://en.wikipedia.org/wiki/Fizz_buzz

Fizz buzz has been used as an interview screening device for computer programmers.

http://en.wikipedia.org/wiki/Fizz_buzz

Creating a list of the first 100 Fizz buzz numbers is a trivial problem for any would-be computer programmer.

http://en.wikipedia.org/wiki/Fizz_buzz

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

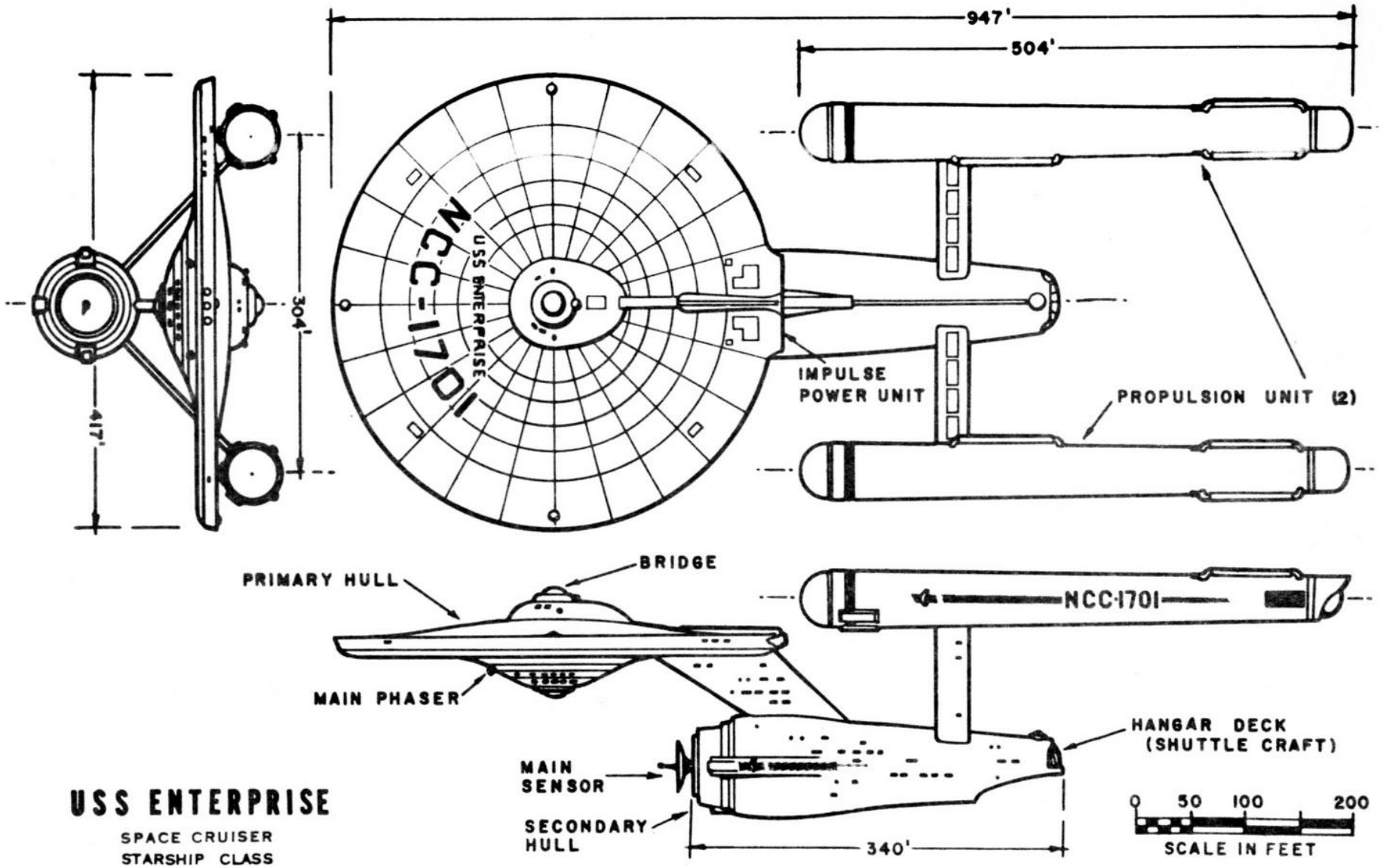
https://twitter.com/richardadalton/status/591534529086693376

FizzBuzz is also a popular Kata in the software craftsmanship movement.

http://en.wikipedia.org/wiki/Fizz_buzz

```
for (var i = 1; i <= 100; i++) {
    // For each iteration,
    // initialize an empty string
    var string = '';

    // If `i` is divisible through 3
    // without a rest, append `Fizz`
    if (i % 3 == 0) {
        string += 'Fizz';
    }

    // If `i` is divisible through 5
    // without a rest, append `Buzz`
    if (i % 5 == 0) {
        string += 'Buzz';
    }

    // If `string` is still empty,
    // `i` is not divisible by 3 or 5,
    // so use the number instead.
    if (string == '') {
        string += i;
    }

    // At the end of this iteration, print the string
    console.log(string);
}
```

http://en.wikipedia.org/wiki/Fizz_buzz



USS ENTERPRISE

SPACE CRUISER
STARSHIP CLASS

enterprise, *noun*

- a project or undertaking that is especially bold, complicated or arduous
- readiness to engage in undertakings of difficulty, risk, danger or daring
- a design of which the execution is attempted
- a commercial or industrial undertaking
- a firm, company or business
- a unit of economic organisation or activity

Concise Oxford English Dictionary · Oxford English Dictionary · Merriam-Webster's Collegiate Dictionary

enterprise, *noun*

- a project or undertaking that is especially bold, complicated or arduous
- readiness to engage in undertakings of difficulty, risk, danger or daring
- a design of which the execution is attempted
- a commercial or industrial undertaking
- a firm, company or business
- a unit of economic organisation or activity

Concise Oxford English Dictionary · Oxford English Dictionary · Merriam-Webster's Collegiate Dictionary

```
// Define constants
#define FIZZ "fizz"
#define BUZZ "buzz"
#define NULL 0
#define PI 3.14

std::string FizzBuzzCalculator(int n)
{
    // Declarations
    char buffer[1024];
    memset(buffer, 0, 1024);
    std::string retval;
    auto isDivisibleBy3 = [&]() { if(n % 3 == 0) { strcpy(buffer, FIZZ); return true; } return false; };
    auto isDivisibleBy5 = [&]() { if(n % 5 == 0) { strcat(buffer, BUZZ); return true; } return false; };

    // Check to see if argument is between 1 and 100, otherwise log and throw
    if(n < 1 || n > 100)
    {
        sprintf(buffer, "FizzBuzzCalculator, argument must be in range: %i", n);
        Log::Instance().Write(buffer);
        throw std::invalid_argument(buffer);
    }

    // Check if need to write number
    if (!isDivisibleBy3() & !isDivisibleBy5())
    {
        sprintf(buffer, "%i", n);
    }

    // Assign return value (retval) and return
    retval.assign(buffer, strlen(buffer));
    return retval;
}
```

```
// Define constants
#define FIZZ "fizz"
#define BUZZ "buzz"
#define NULL 0
#define PI 3.14

std::string FizzBuzzCalculator(int n)
{
    // Declarations
    char buffer[1024];
    memset(buffer, 0, 1024);
    std::string retval;
    auto isDivisibleBy3 = [&]() { if(n % 3 == 0) { strcpy(buffer, FIZZ); return true; } return false; };
    auto isDivisibleBy5 = [&]() { if(n % 5 == 0) { strcat(buffer, BUZZ); return true; } return false; };

    // Check to see if argument is between 1 and 100, otherwise log and throw
    if(n < 1 || n > 100)
    {
        sprintf(buffer, "FizzBuzzCalculator, argument must be in range: %i", n);
        Log::Instance().Write(buffer);
        throw std::invalid_argument(buffer);
    }

    // Check if need to write number
    if (!isDivisibleBy3() & !isDivisibleBy5())
    {
        sprintf(buffer, "%i", n);
    }

    // Assign return value (retval) and return
    retval.assign(buffer, strlen(buffer));
    return retval;
}
```

```
// Define constants
#define FIZZ "fizz"
#define BUZZ "buzz"
#define NULL 0
#define PI 3.14

std::string FizzBuzzCalculator(int n)
{
    // Declarations
    char buffer[1024];
    memset(buffer, 0, 1024);
    std::string retval;
    auto isDivisibleBy3 = [&]() { if(n % 3 == 0) { strcpy(buffer, FIZZ); return true; } return false; };
    auto isDivisibleBy5 = [&]() { if(n % 5 == 0) { strcat(buffer, BUZZ); return true; } return false; };

    // Check to see if argument is between 1 and 100, otherwise log and throw
    if(n < 1 || n > 100)
    {
        sprintf(buffer, "FizzBuzzCalculator, argument must be in range: %i", n);
        Log::Instance().Write(buffer);
        throw std::invalid_argument(buffer);
    }

    // Check if need to write number
    if (!isDivisibleBy3() & !isDivisibleBy5())
    {
        sprintf(buffer, "%i", n);
    }

    // Assign return value (retval) and return
    retval.assign(buffer, strlen(buffer));
    return retval;
}
```

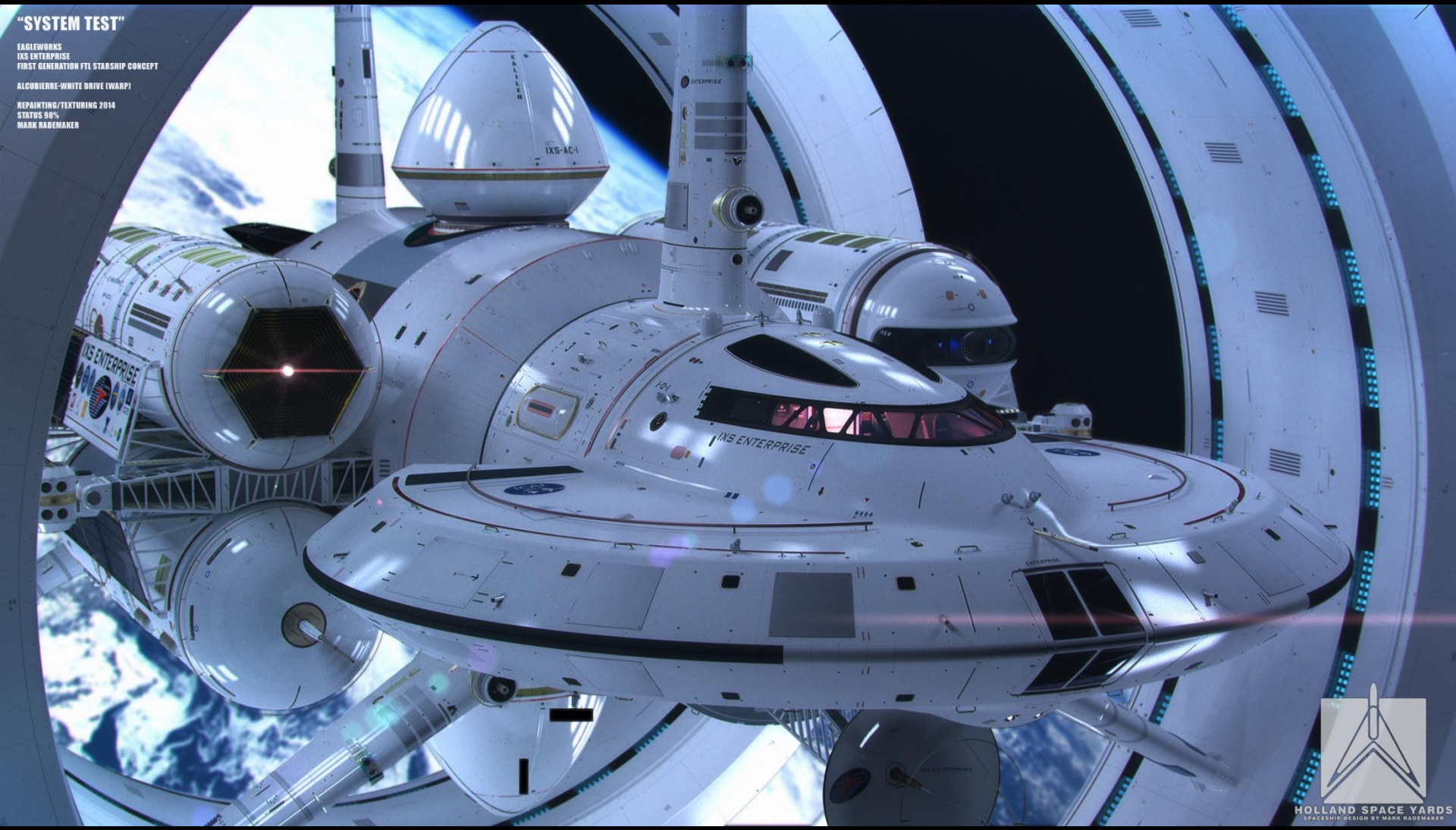
```
// Define constants
#define FIZZ "fizz"
#define BUZZ "buzz"
#define NULL 0
#define PI 3.14

std::string FizzBuzzCalculator(int n)
{
    // Declarations
    char buffer[1024];
    memset(buffer, 0, 1024);
    std::string retval;
    auto isDivisibleBy3 = [&]() { if(n % 3 == 0) { strcpy(buffer, FIZZ); return true; } return false; };
    auto isDivisibleBy5 = [&]() { if(n % 5 == 0) { strcat(buffer, BUZZ); return true; } return false; };

    // Check to see if argument is between 1 and 100, otherwise log and throw
    if(n < 1 || n > 100)
    {
        sprintf(buffer, "FizzBuzzCalculator, argument must be in range: %i", n);
        Log::Instance().Write(buffer);
        throw std::invalid_argument(buffer);
    }

    // Check if need to write number
    if (!isDivisibleBy3() & !isDivisibleBy5())
    {
        sprintf(buffer, "%i", n);
    }

    // Assign return value (retval) and return
    retval.assign(buffer, strlen(buffer));
    return retval;
}
```

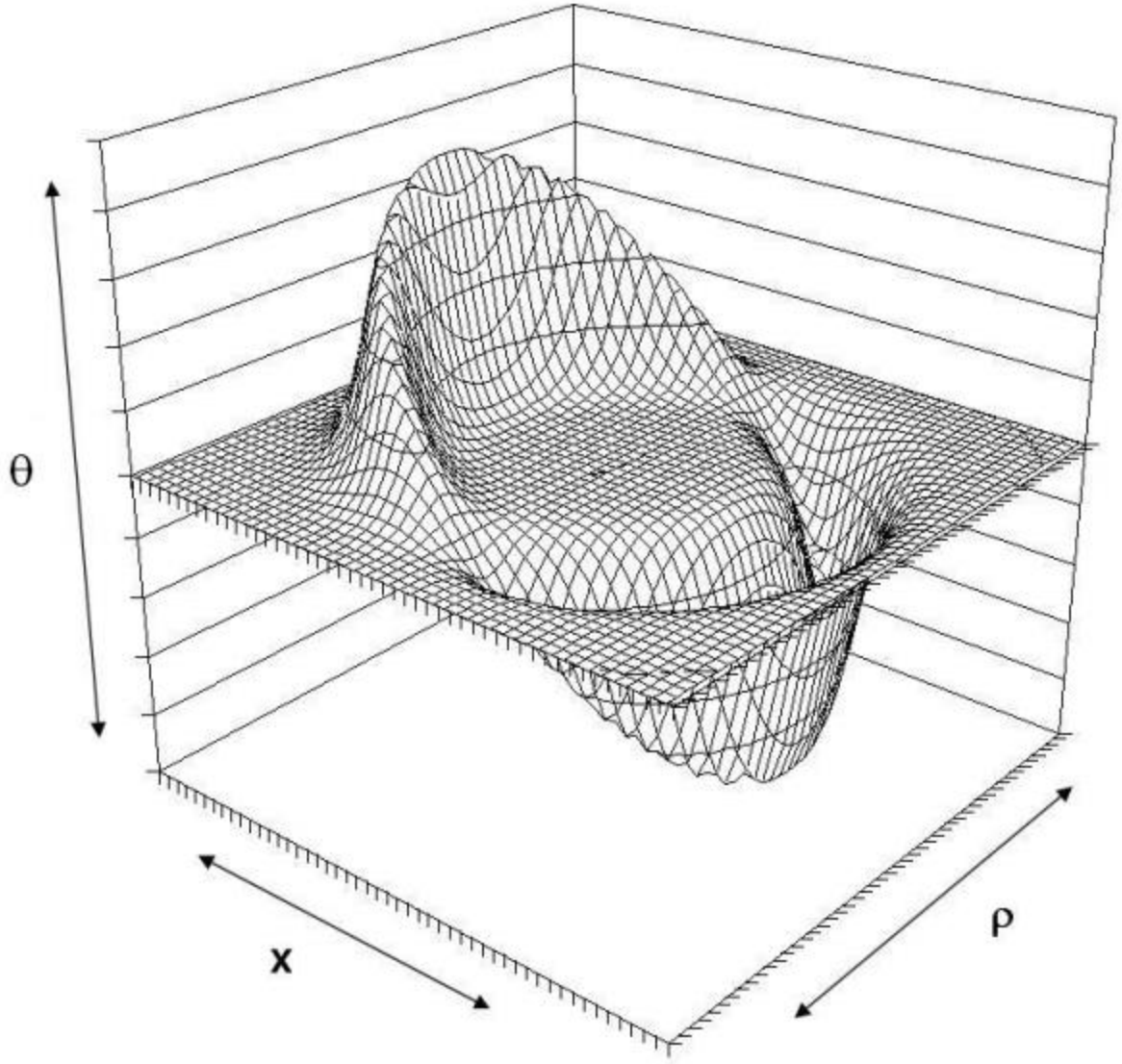
```
// Define constants
#define FIZZ "fizz"
#define BUZZ "buzz"
#define NULL 0
#define PI 3.14

std::string FizzBuzzCalculator(int n)
{
    // Declarations
    char buffer[1024];
    memset(buffer, 0, 1024);
    std::string retval;
    auto isDivisibleBy3 = [&]() { if(n % 3 == 0) { strcpy(buffer, FIZZ); return true; } return false; };
    auto isDivisibleBy5 = [&]() { if(n % 5 == 0) { strcat(buffer, BUZZ); return true; } return false; };

    // Check to see if argument is between 1 and 100, otherwise log and throw
    if(n < 1 || n > 100)
    {
        sprintf(buffer, "FizzBuzzCalculator, argument must be in range: %i", n);
        Log::Instance().Write(buffer);
        throw std::invalid_argument(buffer);
    }

    // Check if need to write number
    if (!isDivisibleBy3() & !isDivisibleBy5())
    {
        sprintf(buffer, "%i", n);
    }

    // Assign return value (retval) and return
    retval.assign(buffer, strlen(buffer));
    return retval;
}
```

```
// Define constants
#define FIZZ "fizz"
#define BUZZ "buzz"
#define NULL 0
#define PI 3.14

std::string FizzBuzzCalculator(int n)
{
    // Declarations
    char buffer[1024];
    memset(buffer, 0, 1024);
    std::string retval;
    auto isDivisibleBy3 = [&]() { if(n % 3 == 0) { strcpy(buffer, FIZZ); return true; } return false; };
    auto isDivisibleBy5 = [&]() { if(n % 5 == 0) { strcat(buffer, BUZZ); return true; } return false; };

    // Check to see if argument is between 1 and 100, otherwise log and throw
    if(n < 1 || n > 100)
    {
        sprintf(buffer, "FizzBuzzCalculator, argument must be in range: %i", n);
        Log::Instance().Write(buffer);
        throw std::invalid_argument(buffer);
    }

    // Check if need to write number
    if (!isDivisibleBy3() & !isDivisibleBy5())
    {
        sprintf(buffer, "%i", n);
    }

    // Assign return value (retval) and return
    retval.assign(buffer, strlen(buffer));
    return retval;
}
```

```
// Define constants
#define FIZZ "fizz"
#define BUZZ "buzz"
#define NULL 0
#define PI 3.14

std::string FizzBuzzCalculator(int n)
{
    // Declarations
    char buffer[1024];
    memset(buffer, 0, 1024);
    std::string retval;
    auto isDivisibleBy3 = [&]() { if(n % 3 == 0) { strcpy(buffer, FIZZ); return true; } return false; };
    auto isDivisibleBy5 = [&]() { if(n % 5 == 0) { strcat(buffer, BUZZ); return true; } return false; };

    // Check to see if argument is between 1 and 100, otherwise log and throw
    if(n < 1 || n > 100)
    {
        sprintf(buffer, "FizzBuzzCalculator, argument must be in range: %i", n);
        Log::Instance().Write(buffer);
        throw std::invalid_argument(buffer);
    }

    // Check if need to write number
    if (!isDivisibleBy3() & !isDivisibleBy5())
    {
        sprintf(buffer, "%i", n);
    }

    // Assign return value (retval) and return
    retval.assign(buffer, strlen(buffer));
    return retval;
}
```

USS ENTERPRISE
NCC-1701
NCC 1701


"SYSTEM TEST"
EAGLEWORKS
IXS ENTERPRISE
FIRST GENERATION FTL STARSHIP CONCEPT
ALCUBIERRE-WHITE DRIVE (WARP)
REPAINTING/TEXTURING 2014
STATUS 98%
MARK RADEMAKER
IXS-AC-1
IXS ENTERPRISE
HOLLAND SPACE YARDS
SPACESHIP DESIGN BY MARK RADEMAKER


θ
x
ρ


Enterprise
NASA
United States

README.md

FizzBuzzEnterpriseEdition

Enterprise software marks a special high-grade class of software that makes careful use of relevant software architecture design principles to build particularly customizable and extensible solutions to real problems. This project is an example of how the popular FizzBuzz game might be built were it subject to the high quality standards of enterprise software.

FizzBuzz

FizzBuzz is a game that has gained in popularity as a programming assignment to weed out non-programmers during job interviews. The object of the assignment is less about solving it correctly according to the below rules and more about showing the programmer understands basic, necessary tools such as `if` -/ `else` -statements and loops. The rules of FizzBuzz are as follows:

For numbers 1 through 100,

- if the number is divisible by 3 print Fizz;
- if the number is divisible by 5 print Buzz;
- if the number is divisible by 3 and 5 (15) print FizzBuzz;
- else, print the number.

FizzBuzzEnterpriseEdition / src / main / java / com / seriouscompany / business / java / fizzbuzz / packagenamingpackage / interfaces / **factories** / +

Merge #127 — 2 comments

Dmitry-Me authored on 22 Jul 2014 — latest commit `aad80defc4`

..		
FizzBuzzOutputStrategyFactory.java	Adding a factory to create the output strategy so that we never get l...	a year ago
FizzBuzzSolutionStrategyFactory.j...	Problem solution should be instantiated through factory class to make...	2 years ago
IntegerPrinterFactory.java	Move sources	2 years ago
IntegerStringReturnerFactory.java	Move sources	2 years ago
IsEvenlyDivisibleStrategyFactory.java	Move sources	2 years ago
OutputGenerationContextVisitorFac...	Merge #127	8 months ago
StringPrinterFactory.java	Move sources	2 years ago
StringStringReturnerFactory.java	Move sources	2 years ago

https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition

All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.

Grady Booch

Architecture is the art of how to waste space.

Philip Johnson

Kevlin Henney
@KevlinHenney

Follow

Having experienced Enterprisation of OO — large classes named as Manager, Controller, Object, etc. — now dreading Enterprisation of FP.

6:42 AM - 12 Jun 2014

9 RETWEETS **6** FAVORITES

https://twitter.com/KevlinHenney/status/476962681636020224

Kevlin Henney
@KevlinHenney

Follow

Having experienced Enterprisation of OO — large classes named as Manager, Controller, Object, etc. — now dreading Enterprisation of FP.

6:42 AM - 12 Jun 2014

9 RETWEETS **6** FAVORITES

Kevlin Henney
@KevlinHenney

Follow

Enterprise FP: large functions with Function, Transform, Process, Calculate, etc. in their names... and lots of dependencies and side effects.

6:43 AM - 12 Jun 2014

7 RETWEETS **3** FAVORITES

https://twitter.com/KevlinHenney/status/476963022549032960

The Paradigms of Programming

Robert W. Floyd
Stanford University

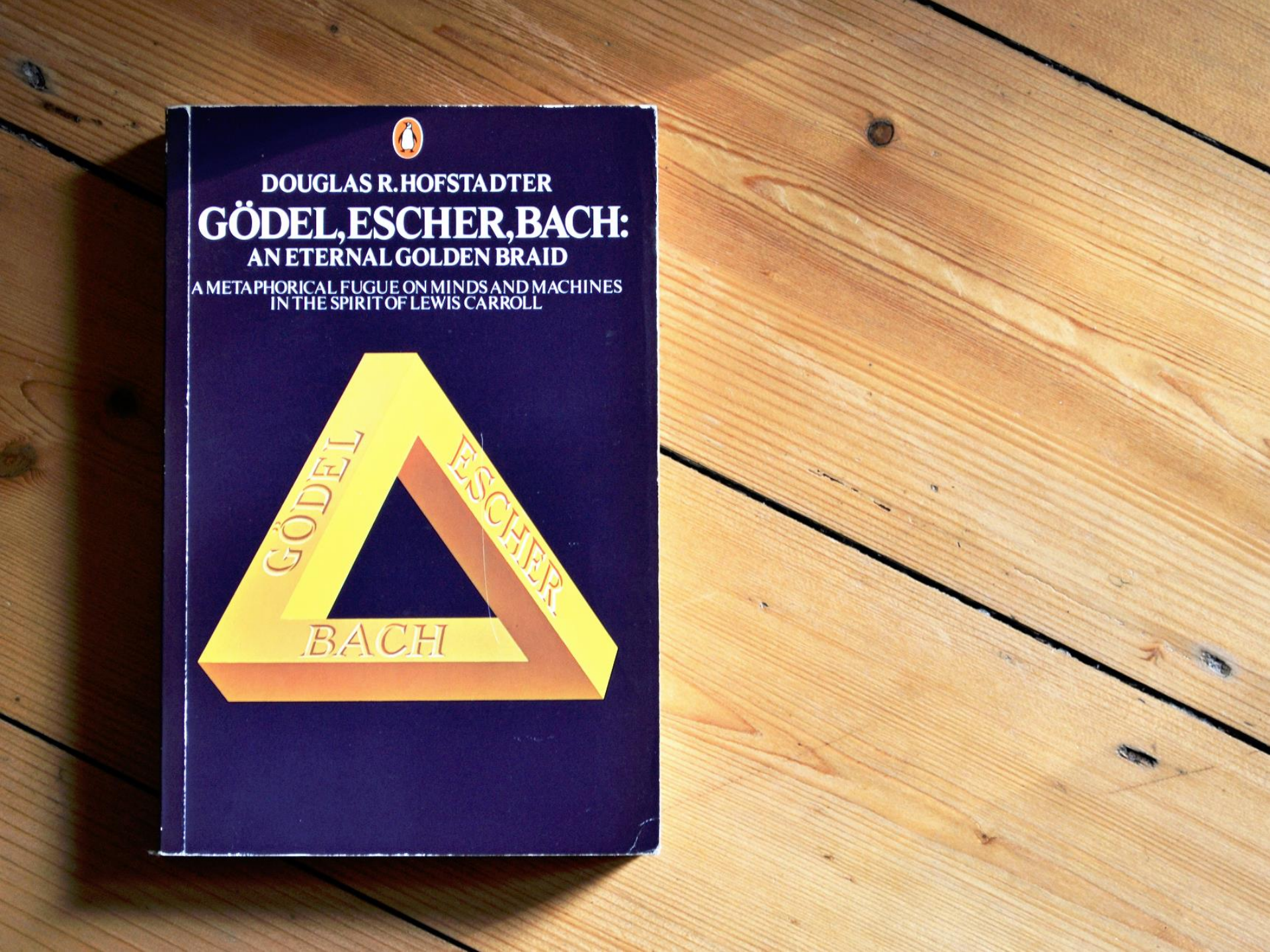
Paradigm(pæ·radim, −dəim) . . . [a. F. *paradigme*, ad. L. *paradigma*, a. Gr. παραδειγμα pattern, example, f. παραδεικνυ·ναι to exhibit beside, show side by side. . .]
1. A pattern, exemplar, example.
1752 J. Gill *Trinity* v. 91
The archetype, paradigm, exemplar, and idea, according to which all things were made.
From the Oxford English Dictionary.

Today I want to talk about the paradigms of programming, how they affect our success as designers of computer programs, how they should be taught, and how they should be embodied in our programming languages.

A familiar example of a paradigm of programming is the technique of *structured programming*, which appears to be the dominant paradigm in most current treatments of programming methodology. Structured programming, as formulated by Dijkstra [6], Wirth [27, 29], and Parnas [21], among others, consists of two phases.

In the first phase, that of top-down design, or stepwise refinement, the problem is decomposed into a very small number of simpler subproblems. In programming the solution of simultaneous linear equations, say, the first level of decomposition would be into a stage of triangularizing the equations and a following stage of back-substitution in the triangularized system. This gradual decomposition is continued until the subproblems that arise are simple enough to cope with directly. In the simultaneous equation example, the back substitution process would be further decomposed as a backwards iteration of a process which finds and stores the value of the *i*th variable from the *i*th equation. Yet further decomposition would yield a fully detailed algorithm.

I believe that the current state of the art of computer programming reflects inadequacies in our stock of paradigms, in our knowledge of existing paradigms, in the way we teach programming paradigms, and in the way our programming languages support, or fail to support, the paradigms of their user communities.


DOUGLAS R. HOFSTADTER
GÖDEL, ESCHER, BACH:
AN ETERNAL GOLDEN BRAID
A METAPHORICAL FUGUE ON MINDS AND MACHINES
IN THE SPIRIT OF LEWIS CARROLL
GÖDEL
ESCHER
BACH

Achilles: What I was driving [illegible] that exact order.

Tortoise: Oh, now I understand what you meant when you asked me "What sort of word is that?" The answer is that a philosopher by the name of "Willard Van Orman Quine" invented the operation, so I name it in his honor. However, I cannot go any further than this in my explanation. Why these particular five letters make up his name—not to mention why they occur in this particular order—is a question to which I have no ready answer. However, I'd be perfectly willing to go and—

Achilles: Please don't bother! I didn't really want to know everything about Quine's name. Anyway, now I know how to quine a phrase. It's quite amusing. Here's a quined phrase:

"IS A SENTENCE FRAGMENT" IS A SENTENCE FRAGMENT.

It's silly but all the same I enjoy it. You take a sentence fragment, quine it, and lo and behold, you've made a sentence! A true sentence, in this case.

Tortoise: How about quining the phrase "is a king with no subject"?

Achilles: A king without a subject would be—

Tortoise: —an anomaly, of course. Don't wander from the point. Let's have quines first, and kings afterwards!

Achilles: I'm to quine that phrase, am I? All right—

"IS A KING WITH NO SUBJECT" IS A KING WITH NO SUBJECT.

It seems to me that it might make more sense if it said "sentence"

As a result of this research, I created a language called HQ9+. HQ9+ is a very simple language consisting of four operations: H, Q, 9, and +. These operations can be used to create any of the types of example programs described above. They work as follows:

```
H  Prints "Hello, world!"
Q  Prints the entire text of the source code file.
9  Prints the complete canonical lyrics to "99 Bottles of Beer on the Wall"
+  Increments the accumulator.
```

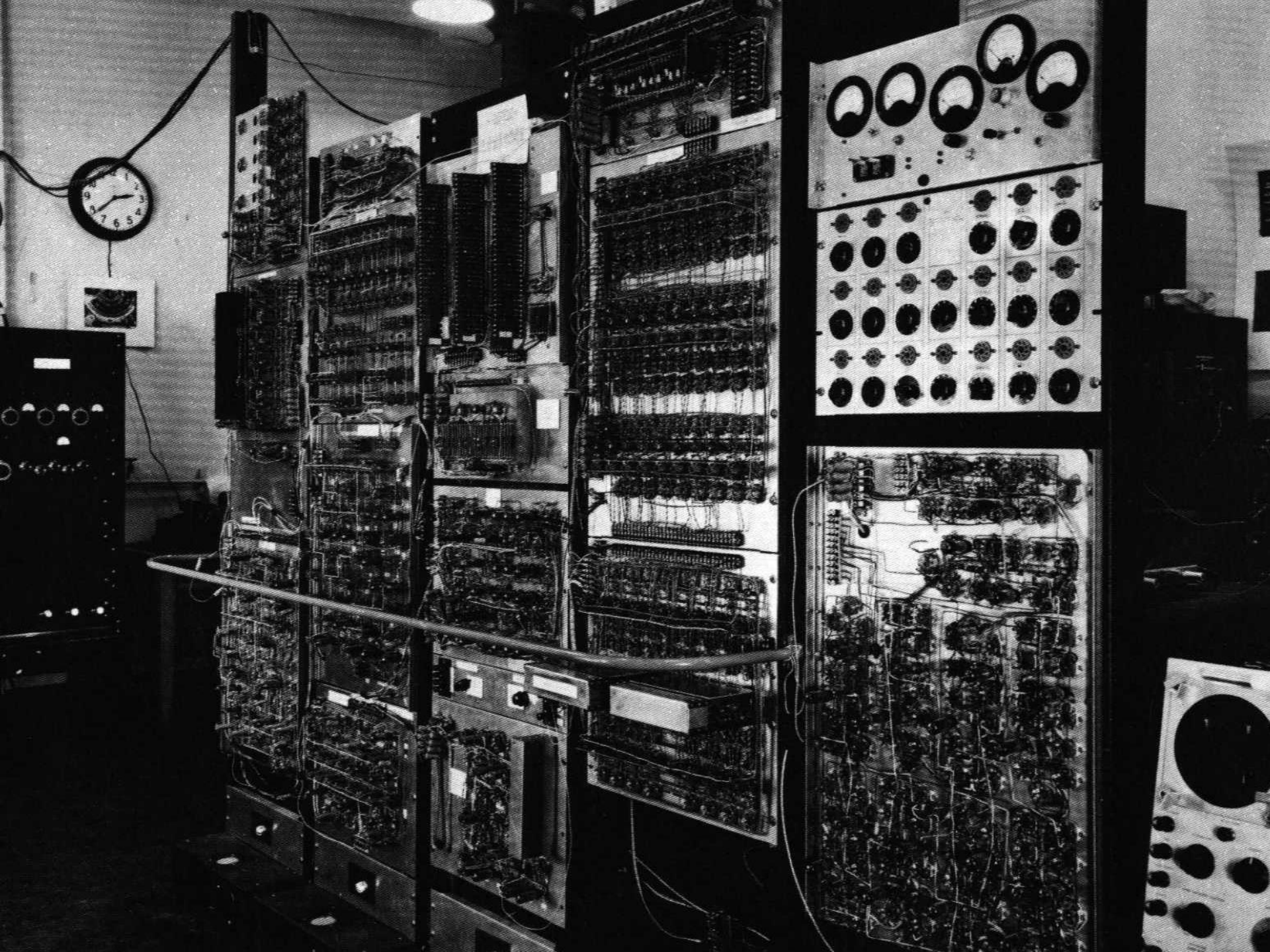
HQ9+ is very simple, but allows you to do some things that are very difficult in other languages. For example, here is a program that creates four – count 'em, four – copies of itself on the screen:

```
qqqq
```

This produces:

```
qqqq
qqqq
qqqq
qqqq
```

Wow! Wasn't that straightforward?

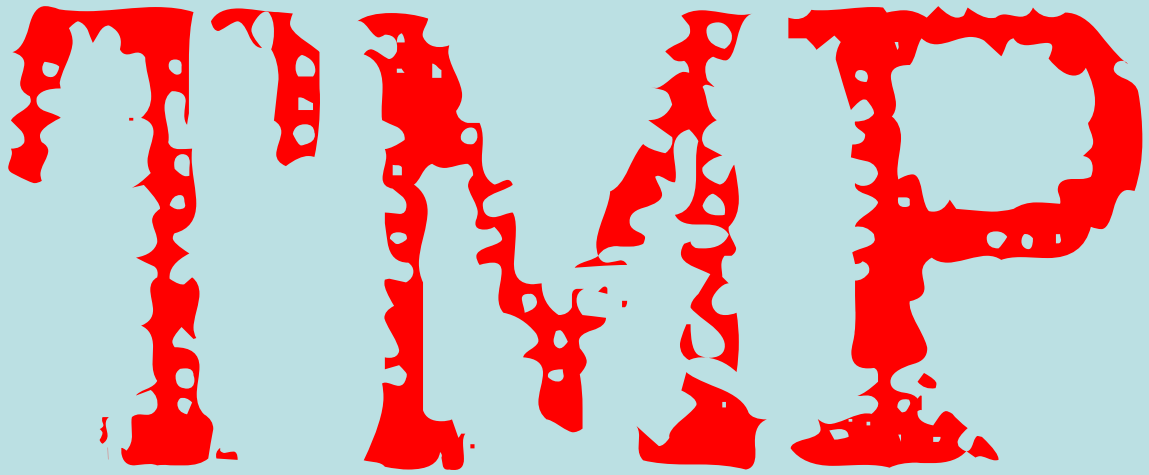


T123SE60S#S*S&S@SP100SP10SP5SP3S
P1SQSPSBSFSISUSZSPSP1SPSPSO34SO3
5SO36ST49SA50SA41ST50SA50SU51SS4
OSE62SA40SS41SE73ST51SO34SO45SO4
6SO48SO48ST49SA50SS39SE75SA39SS4
1SE86ST51SO34SO44SO47SO48SO48ST4
9SA51SG53SO33ST49SA50SS37SA41SG9
8SZSO43SO43ST49ST52SA50SS38SG109
ST51SA52SA41ST52SA51SE101ST49SA5
2SS41SG117SA41SL512ST52SO52ST49S
A51SL512ST52SO52SE53SXS

Olve Maudal

http://www.pvv.org/~oma/FizzBuzz_EDSAC_ACCU_Apr2015.pdf

```
// Erwin Unruh, untitled program,
// ANSI X3J16-94-0075/ISO WG21-462, 1994.

template <int i>
struct D
{
    D(void *);
    operator int();
};

template <int p, int i>
struct is_prime
{
    enum { prim = (p%i) && is_prime<(i>2?p:0), i>::prim };
};

template <int i>
struct Prime_print
{
    Prime_print<i-1>    a;
    enum { prim = is_prime<i,i-1>::prim };
    void f() { D<i> d = prim; }
};

struct is_prime<0,0> { enum { prim = 1 }; };
struct is_prime<0,1> { enum { prim = 1 }; };
struct Prime_print<2>
{
    enum { prim = 1 };
    void f() { D<2> d = prim; }
};

void foo()
{
    Prime_print<10> a;
}

// output:
// unruh.cpp 30: conversion from enum to D<2> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<3> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<5> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<7> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<11> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<13> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<17> requested in Prime_print
// unruh.cpp 30: conversion from enum to D<19> requested in Prime_print
```


TMP

```
struct Fizz{};
struct Buzz{};
struct FizzBuzz{};

template<int i>
struct RunFizzBuzz
{
   typedef vector<int_<i> > Number;

   typedef typename if_c<(i % 3 == 0) && (i % 5 == 0), FizzBuzz,
                     typename if_c<i % 3 == 0, Fizz,
                              typename if_c<i % 5 == 0, Buzz, Number>::type>::type >::type t1;

   typedef typename push_back<typename RunFizzBuzz<i - 1>::ret, t1>::type ret;
};

template<>
struct RunFizzBuzz<0> // Terminate the recusion.
{
   typedef vector<int_<0> > ret;
};

int main()
{
   typedef RunFizzBuzz<100>::ret::compilation_error_here res;
}
```

```
\Main.cpp(36) : error C2039: 'compilation_error_here' : is not a member of
 'boost::mpl::vector101 <SNIP long argument list>'
    with
    [
          T0=boost::mpl::int_<0>,
          T1=boost::mpl::vector<boost::mpl::int_<1>>,
          T2=boost::mpl::vector<boost::mpl::int_<2>>,
          T3=Fizz,
          T4=boost::mpl::vector<boost::mpl::int_<4>>,
          T5=Buzz,
          T6=Fizz,
          T7=boost::mpl::vector<boost::mpl::int_<7>>,
          T8=boost::mpl::vector<boost::mpl::int_<8>>,
          T9=Fizz,
          T10=Buzz,
          T11=boost::mpl::vector<boost::mpl::int_<11>>,
          T12=Fizz,
          T13=boost::mpl::vector<boost::mpl::int_<13>>,
          T14=boost::mpl::vector<boost::mpl::int_<14>>,
          T15=FizzBuzz,
          <SNIP of elements 16 - 95>
          T96=Fizz,
          T97=boost::mpl::vector<boost::mpl::int_<97>>,
          T98=boost::mpl::vector<boost::mpl::int_<98>>,
          T99=Fizz,
          T100=Buzz
    ]
```

http://www.adampetersen.se/articles/fizzbuzz.htm

```
enum fizzbuzzed
{
    fizz = -2,
    buzz = -1,
    fizzbuzz = 0,
    first = 1,
    last = 100
};

constexpr fizzbuzzed fizzbuzz_of(int n)
{
    return
        n % 3 == 0 && n % 5 == 0 ? fizzbuzz :
        n % 3 == 0 ? fizz :
        n % 5 == 0 ? buzz :
        fizzbuzzed(n);
}
```

```
enum fizzbuzzed
{
    fizz     = -2,
    buzz     = -1,
    fizzbuzz = 0,
    first    = 1,
    last     = 100
};

constexpr fizzbuzzed fizzbuzz_of(int n)
{
    return
        n % 3 == 0 && n % 5 == 0 ? fizzbuzz :
        n % 3 == 0               ? fizz :
        n % 5 == 0               ? buzz :
                                   fizzbuzzed(n);
}
```

```
enum class fizzbuzzed
{
    fizz     = -2,
    buzz     = -1,
    fizzbuzz = 0,
    first    = 1,
    last     = 100
};

constexpr fizzbuzzed fizzbuzz(int n)
{
    return
        n % 3 == 0 && n % 5 == 0 ? fizzbuzzed::fizzbuzz :
        n % 3 == 0               ? fizzbuzzed::fizz :
        n % 5 == 0               ? fizzbuzzed::buzz :
                                   fizzbuzzed(n);
}
```

Signal-to-noise ratio (often abbreviated SNR or S/N) is a measure used in science and engineering that compares the level of a desired signal to the level of background noise.

Signal-to-noise ratio is sometimes used informally to refer to the ratio of useful information to false or irrelevant data in a conversation or exchange.

http://en.wikipedia.org/wiki/Signal_to_noise_ratio

```
char * fizzbuzz(int n)
{
    char * result = (char *) malloc(20);
    result[0] = '\0';
    if(n % 3 == 0)
    {
        strcat(result, "fizz");
    }
    if(n % 5 == 0)
    {
        strcat(result, "buzz");
    }
    if(result[0] == '\0')
    {
        sprintf(result, "%i", n);
    }
    return result;
}
```

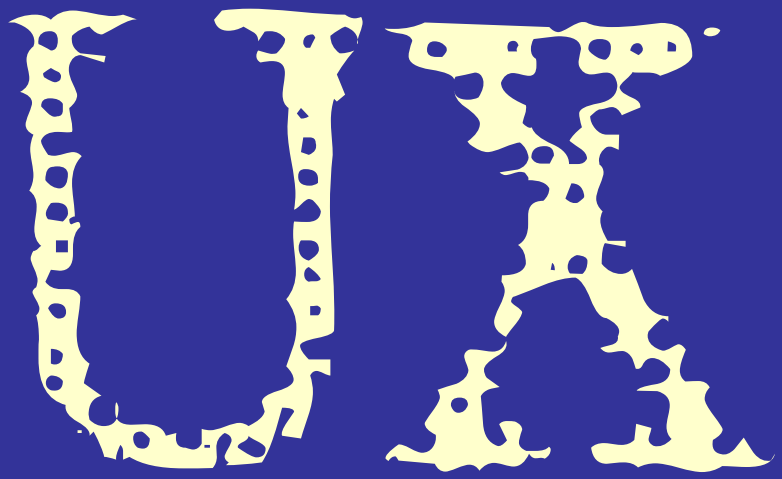
```
char * fizzbuzz(int n)
{
    char * result = (char *) calloc(20, 1);
    if(n % 3 == 0)
    {
        strcat(result, "fizz");
    }
    if(n % 5 == 0)
    {
        strcat(result, "buzz");
    }
    if(result[0] == '\0')
    {
        sprintf(result, "%i", n);
    }
    return result;
}
```

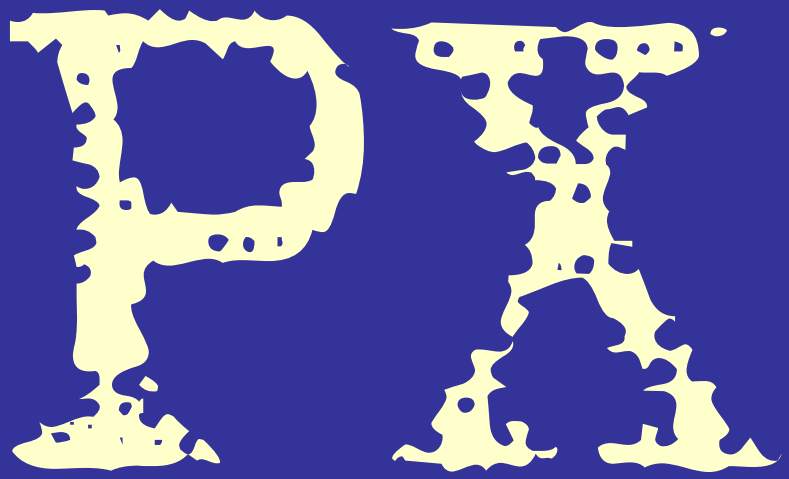
```
char * fizzbuzz(int n)
{
    char * result = (char *) calloc(20, 1);
    strcat(result, n % 3 == 0 ? "fizz" : "");
    strcat(result, n % 5 == 0 ? "buzz" : "");
    if(result[0] == '\0')
        sprintf(result, "%i", n);
    return result;
}
```

```
char * fizzbuzz(int n)
{
    char * result = (char *) malloc(20);
    strcpy(result, n % 3 == 0 ? "fizz" : "");
    strcat(result, n % 5 == 0 ? "buzz" : "");
    if(result[0] == '\0')
        sprintf(result, "%i", n);
    return result;
}
```

```
char * fizzbuzz(int n)
{
    assert(n >= 1 && n <= 100);
    char * result = (char *) malloc(20);
    strcpy(result, n % 3 == 0 ? "fizz" : "");
    strcat(result, n % 5 == 0 ? "buzz" : "");
    if(result[0] == '\0')
        sprintf(result, "%i", n);
    return result;
}
```

```
char * fizzbuzz(char * result, size_t size, int n)
{
    strncpy(result, n % 3 == 0 ? "fizz" : "", size);
    strncat(result, n % 5 == 0 ? "buzz" : "", size);
    if(result[0] == '\0')
        snprintf(result, size, "%i", n);
    return result;
}
```


UX


PX

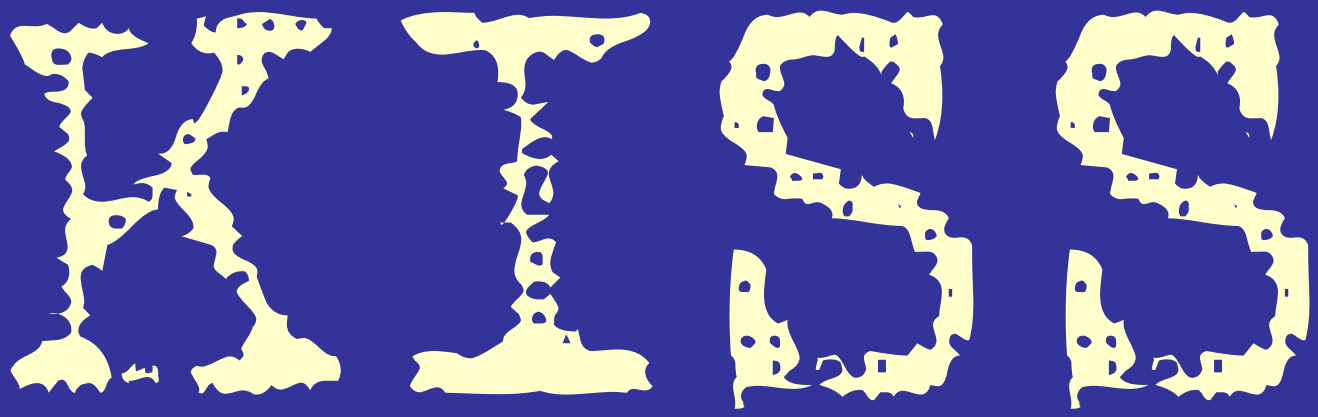
```
struct fizzbuzzed
{
    const char * value[2];
};

struct fizzbuzzed fizzbuzz(int n)
{
    struct fizzbuzzed digits =
    {
        "\0""\0""1""\0""2""\0""3""\0""4""\0"
        "5""\0""6""\0""7""\0""8""\0""9\0" + (n / 10) * 2,
        "0""\0""1""\0""2""\0""3""\0""4""\0"
        "5""\0""6""\0""7""\0""8""\0""9" + (n % 10) * 2
    };
    struct fizzbuzzed words =
    {
        n % 3 == 0 ? "fizz" : "", n % 5 == 0 ? "buzz" : ""
    };
    return
        words.value[0][0] || words.value[1][0] ?
            words : digits;
}
```

```
typedef struct fizzbuzzed
{
    const char * value[2];
} fizzbuzzed;

fizzbuzzed fizzbuzz(int n)
{
    fizzbuzzed digits =
    {
        "\0""\0""1""\0""2""\0""3""\0""4""\0"
        "5""\0""6""\0""7""\0""8""\0""9\0" + (n / 10) * 2,
        "0""\0""1""\0""2""\0""3""\0""4""\0"
        "5""\0""6""\0""7""\0""8""\0""9" + (n % 10) * 2
    };
    fizzbuzzed words =
    {
        n % 3 == 0 ? "fizz" : "", n % 5 == 0 ? "buzz" : ""
    };
    return
        words.value[0][0] || words.value[1][0] ?
            words : digits;
}
```

Omit needless words.

William Strunk and E B White
The Elements of Style

```
typedef struct fizzbuzzed
{
    char value[9];
} fizzbuzzed;

fizzbuzzed fizzbuzz(int n)
{
    fizzbuzzed result;
    strcpy(result.value, n % 3 == 0 ? "fizz" : "");
    strcat(result.value, n % 5 == 0 ? "buzz" : "");
    if(result.value[0] == '\0')
        sprintf(result.value, "%i", n);
    return result;
}
```


KISS

```
=IF(AND(MOD(ROW(),3)=
0,MOD(ROW(),5)=0),"Fizz
Buzz",IF(MOD(ROW(),3)=
0,"Fizz",IF(MOD(ROW(),5)
=0,"Buzz",ROW())))
```

Excel is the world's most popular functional language.

Simon Peyton-Jones

```
=IF(AND(MOD(ROW(), 3) = 0,
        MOD(ROW(), 5) = 0),
    "FizzBuzz",
    IF(MOD(ROW(), 3) = 0,
        "Fizz",
        IF(MOD(ROW(), 5) = 0,
            "Buzz",
            ROW())))
```

```
data FizzBuzzed =
    Fizz | Buzz | FizzBuzz | Number Int
    deriving Show

fizzBuzz :: Int -> FizzBuzzed
fizzBuzz n | n `mod` 15 == 0 = FizzBuzz
           | n `mod` 3  == 0 = Fizz
           | n `mod` 5  == 0 = Buzz
           | otherwise       = Number n
```

```
fizzbuzz(N) when N rem 15 == 0 -> fizzbuzz;
fizzbuzz(N) when N rem 3  == 0 -> fizz;
fizzbuzz(N) when N rem 5  == 0 -> buzz;
fizzbuzz(N)                    -> N.
```

```
fizzbuzz(N) ->
    if
        N rem 15 == 0 -> fizzbuzz;
        N rem 3  == 0 -> fizz;
        N rem 5  == 0 -> buzz;
        true          -> N
    end.
```

Multithreading is just one damn thing after, before, or simultaneous with another.

Andrei Alexandrescu

```
fizzbuzzer() ->
    receive
        {User, N} when N >= 1, N =< 100 ->
            User ! fizzbuzz(N),
            fizzbuzzer();
        {User, _} ->
            User ! error,
            fizzbuzzer()
    end.
```

Actor-based concurrency is just one damn message after another.

Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes when it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are sytem components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

```
(1..100) |
  %{
    $fizzed = if($_ % 3 -eq 0) {"Fizz"}
    $buzzed = if($_ % 5 -eq 0) {"Buzz"}
    $fizzbuzzed = $fizzed + $buzzed
    if($fizzbuzzed) {$fizzbuzzed} else {$_}
  }
```

```
$0 % 3 == 0                    { printf "Fizz" }
$0 % 5 == 0                    { printf "Buzz" }
$0 % 3 != 0 && $0 % 5 != 0 { printf $0     }
                               { printf "\n"   }
```

```
echo {1..100} | tr ' ' '\n' | awk '
$0 % 3 == 0                    { printf "Fizz" }
$0 % 5 == 0                    { printf "Buzz" }
$0 % 3 != 0 && $0 % 5 != 0 { printf $0     }
                               { printf "\n"   }
'
```

```
echo {1..100} | tr ' ' '\n' | awk '
$0 % 3 == 0                    { printf "Fizz" }
$0 % 5 == 0                    { printf "Buzz" }
$0 % 3 != 0 && $0 % 5 != 0 { printf $0     }
                               { printf "\n"   }
' | diff - expected && echo Pass
```

Would you do anything differently in the development of AWK looking back?

One of the things that I would have done differently is instituting rigorous testing as we started to develop the language. We initially created AWK as a 'throw-away' language, so we didn't do rigorous quality control as part of our initial implementation.

I mentioned to you earlier that there was a person who wrote a CAD system in AWK. The reason he initially came to see me was to report a bug in the AWK compiler. He was very testy with me saying I had wasted three weeks of his life, as he had been looking for a bug in his own code only to discover that it was a bug in the AWK compiler! I huddled with Brian Kernighan after this, and we agreed we really need to do something differently in terms of quality control. So 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 from then on, first had to write a test for the new feature.

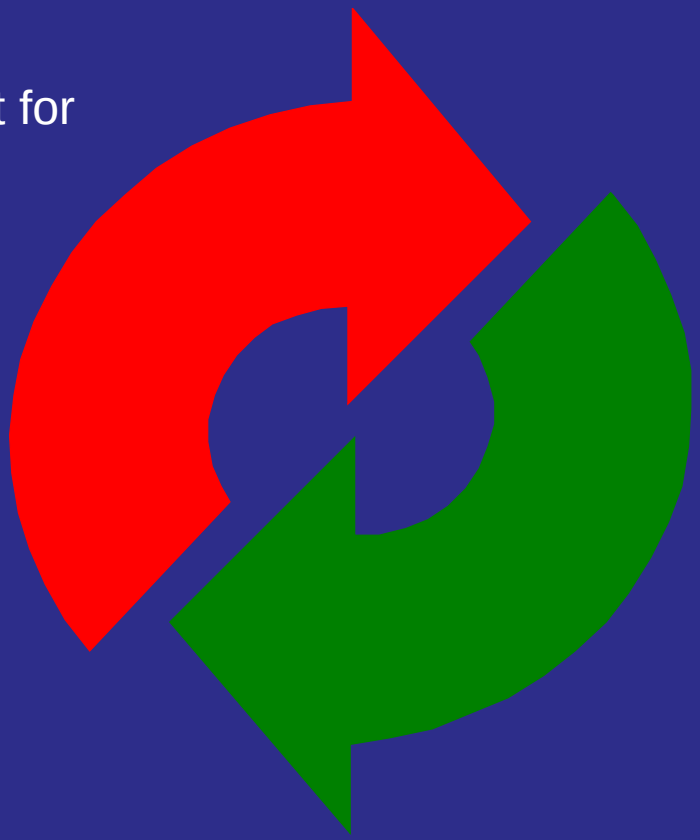
http://www.computerworld.com.au/article/216844/a-z_programming_languages_awk/

Red

Write a failing test for a new feature

Green

Write enough code to pass the test



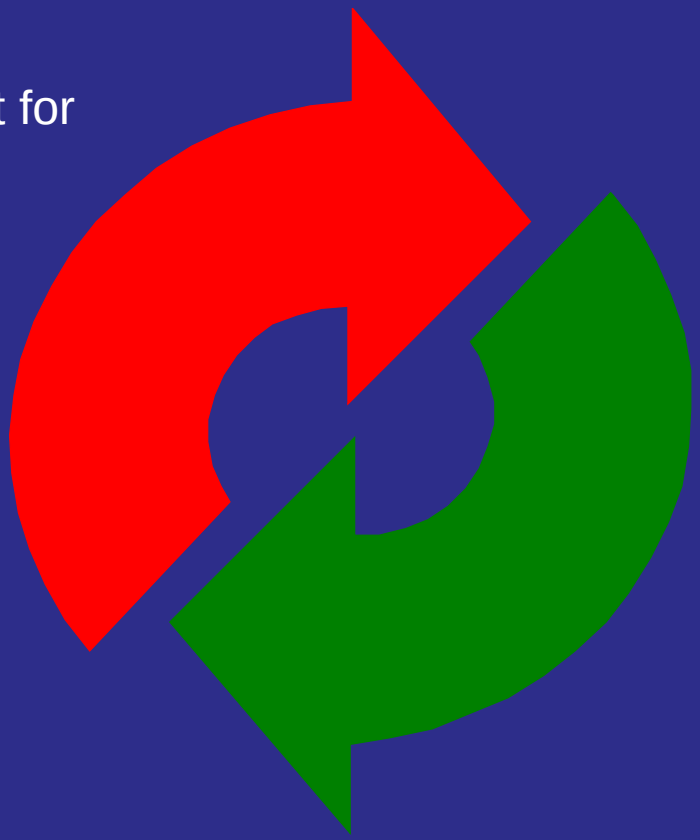
Refactor

Refine code and tests

Test-First Cycle

Red

Write a failing test for a new feature



Green

Write enough code to pass the test

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Robert "Uncle Bob" Martin
"The Three Laws of TDD"
http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

```
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(8) == 8
assert fizzbuzz(9) == 'Fizz'
assert fizzbuzz(10) == 'Buzz'
assert fizzbuzz(11) == 11
assert fizzbuzz(12) == 'Fizz'
assert fizzbuzz(13) == 13
assert fizzbuzz(14) == 14
assert fizzbuzz(15) == 'FizzBuzz'
...
assert fizzbuzz(98) == 98
assert fizzbuzz(99) == 'Fizz'
assert fizzbuzz(100) == 'Buzz'
```

```
def fizzbuzz(n):
    if n == 1: return 1
    if n == 2: return 2
    if n == 3: return 'Fizz'
    if n == 4: return 4
    if n == 5: return 'Buzz'
    if n == 6: return 'Fizz'
    if n == 7: return 7
    if n == 8: return 8
    if n == 9: return 'Fizz'
    if n == 10: return 'Buzz'
    if n == 11: return 11
    if n == 12: return 'Fizz'
    if n == 13: return 13
    if n == 14: return 14
    if n == 15: return 'FizzBuzz'
    ...
    if n == 98: return 98
    if n == 99: return 'Fizz'
    if n == 100: return 'Buzz'
```

DRY?

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

switch

```
def fizzbuzz(n):
    if n == 1: return 1
    if n == 2: return 2
    if n == 3: return 'Fizz'
    if n == 4: return 4
    if n == 5: return 'Buzz'
    if n == 6: return 'Fizz'
    if n == 7: return 7
    if n == 8: return 8
    if n == 9: return 'Fizz'
    if n == 10: return 'Buzz'
    if n == 11: return 11
    if n == 12: return 'Fizz'
    if n == 13: return 13
    if n == 14: return 14
    if n == 15: return 'FizzBuzz'
    ...
    if n == 98: return 98
    if n == 99: return 'Fizz'
    if n == 100: return 'Buzz'
```

```
def fizzbuzz(n):
    return {
        1: lambda: 1,
        2: lambda: 2,
        3: lambda: 'Fizz',
        4: lambda: 4,
        5: lambda: 'Buzz',
        6: lambda: 'Fizz',
        7: lambda: 7,
        8: lambda: 8,
        9: lambda: 'Fizz',
        10: lambda: 'Buzz',
        11: lambda: 11,
        12: lambda: 'Fizz',
        13: lambda: 13,
        14: lambda: 14,
        15: lambda: 'FizzBuzz',
        ...
        98: lambda: 98,
        99: lambda: 'Fizz',
        100: lambda: 'Buzz'
    }[n]()
```

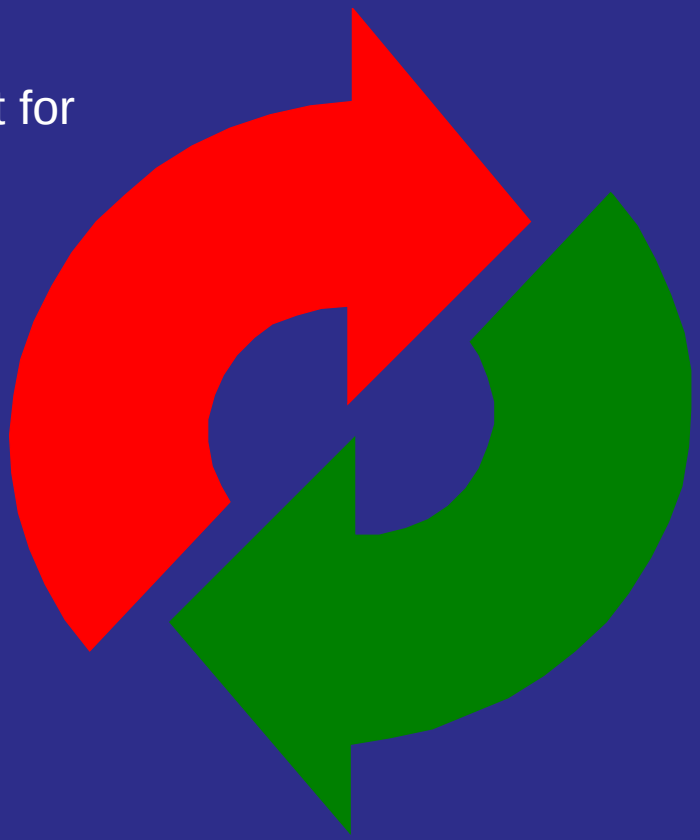
```
def fizzbuzz(n):
    return {
        1: 1,
        2: 2,
        3: 'Fizz',
        4: 4,
        5: 'Buzz',
        6: 'Fizz',
        7: 7,
        8: 8,
        9: 'Fizz',
        10: 'Buzz',
        11: 11,
        12: 'Fizz',
        13: 13,
        14: 14,
        15: 'FizzBuzz',
        ...
        98: 98,
        99: 'Fizz',
        100: 'Buzz'
    }[n]
```

Red

Write a failing test for a new feature

Green

Write enough code to pass the test



Refactor

Refine code and tests

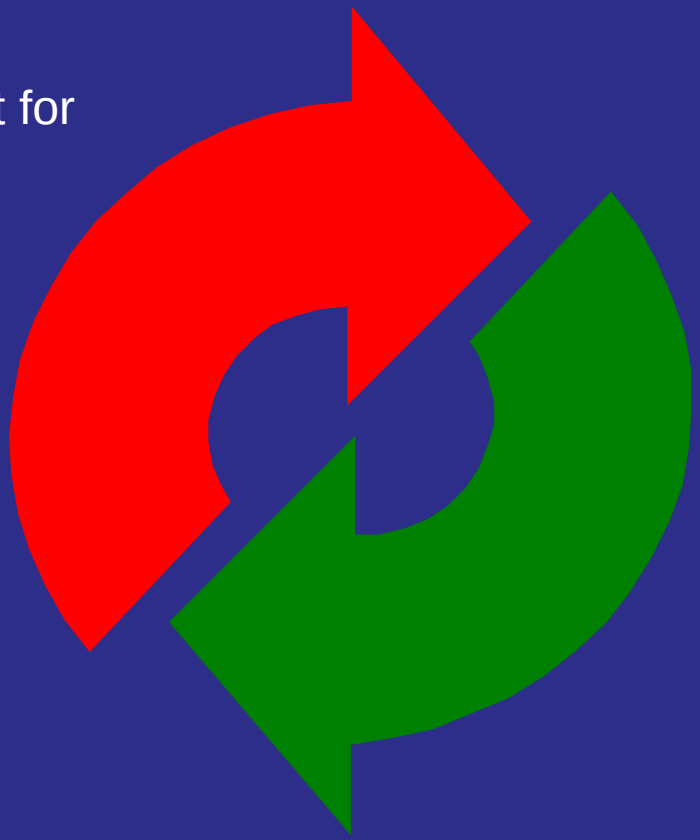
Test-First Cycle

Red

Write a failing test for a new feature

Green

Write enough code to pass the test



Refactor!

Refine code and tests

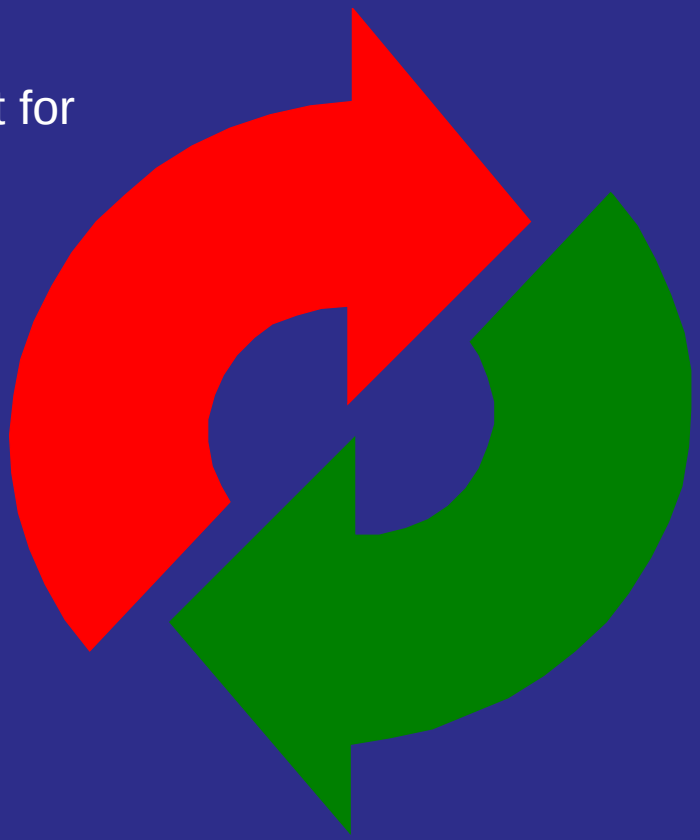
Test-First Cycle

Red

Write a failing test for a new feature

Green

Write enough code to pass the test



Refactor?

Refine code and tests

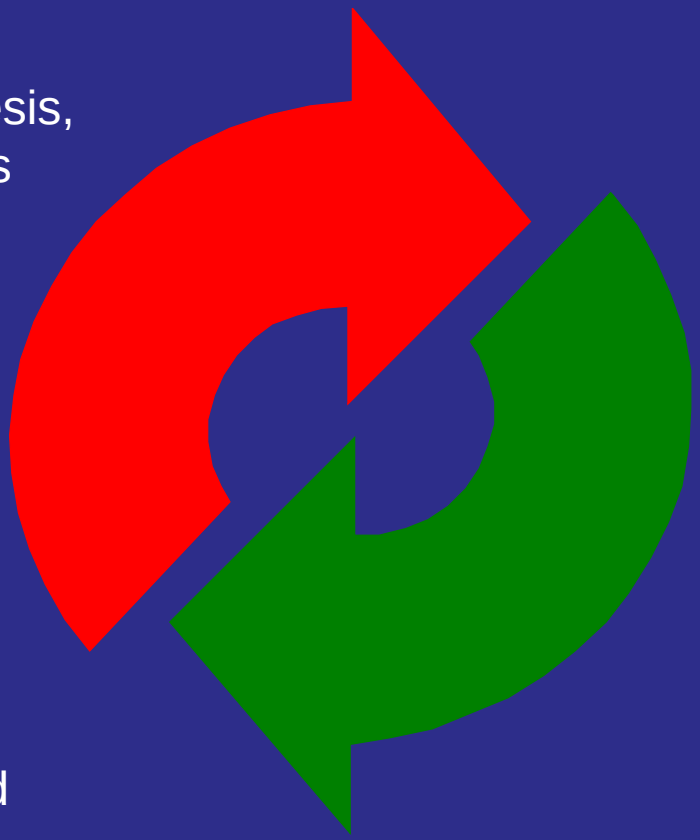
Test-First Cycle

Plan

Establish hypothesis, goal or work tasks

Do

Carry out plan



Act

Revise approach or artefacts based on study

Study

Review what has been done against plan (a.k.a. *Check*)

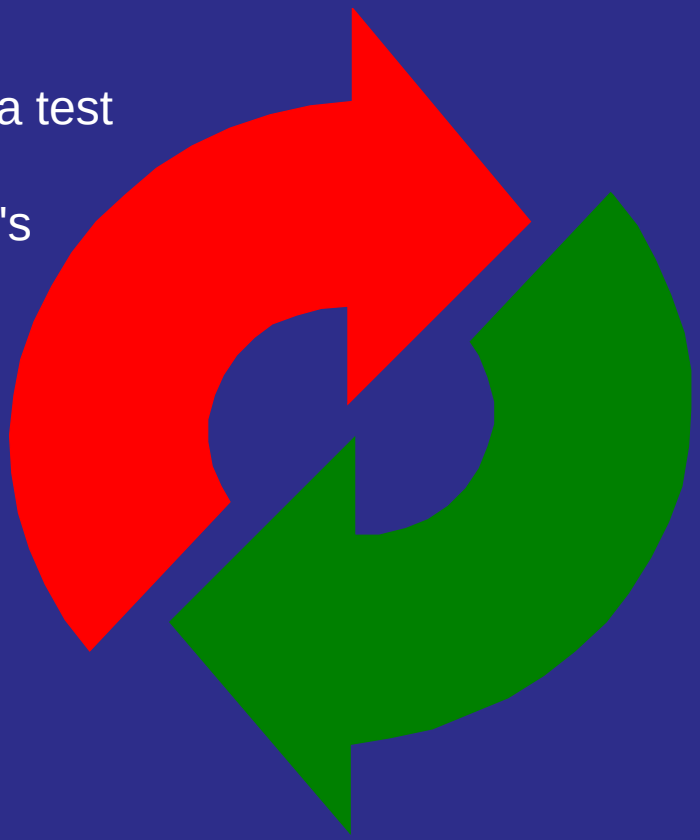
Deming's PDSA Cycle

Write

Create or extend a test case for new behaviour — as it's new, the test fails

Realise

Implement so that the test passes



Refactor

Make it so!

Reflect

Is there something in the code or tests that could be improved?

Test-First Cycle

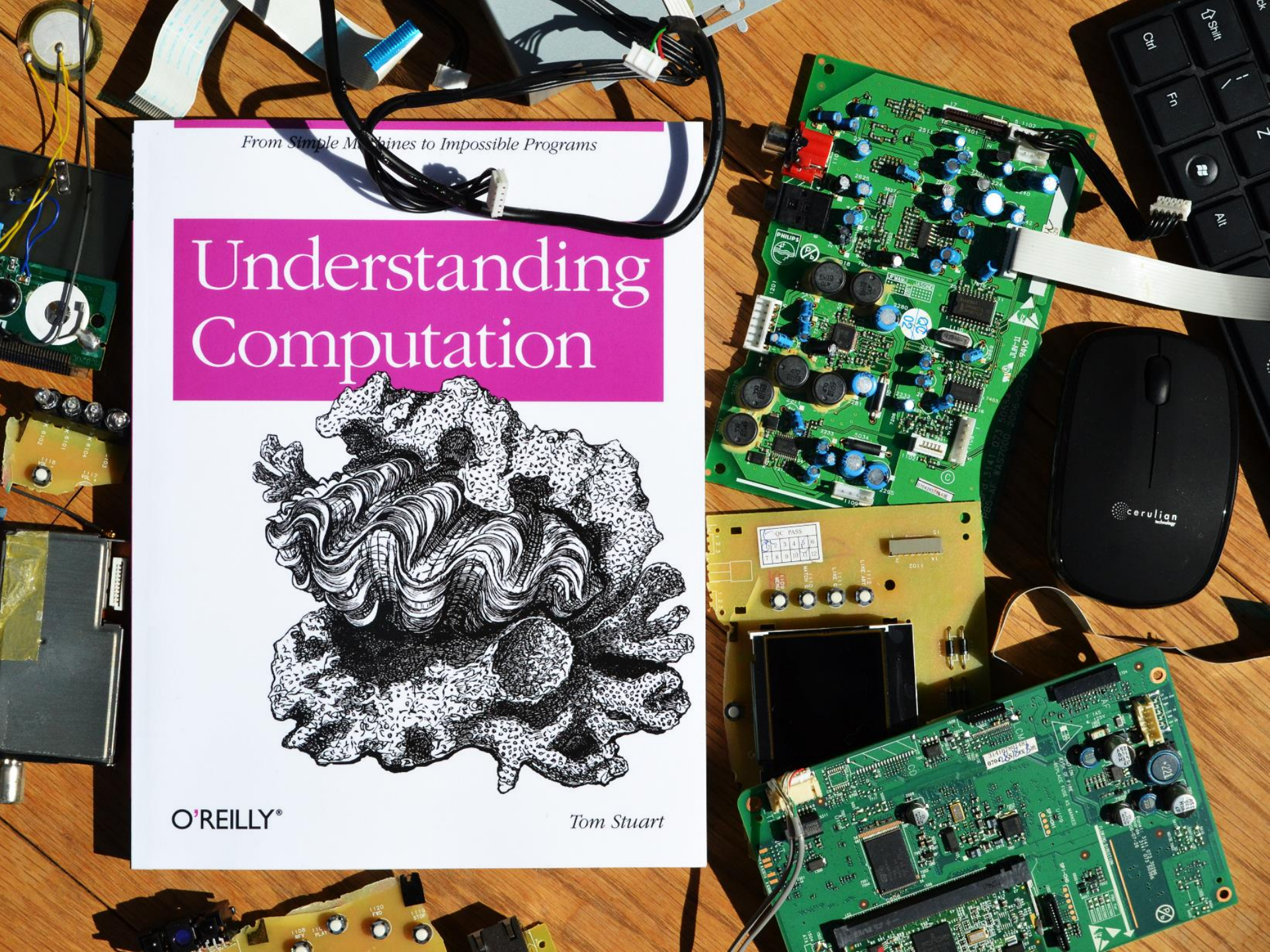
```
def fizzbuzz(n):
    fizzes = [''] + (([''] * 2) + ['Fizz']) * 33 + ['']
    buzzes = [''] + (([''] * 4) + ['Buzz']) * 20
    numbers = list(map(str, range(0, 101)))
    return fizzes[n] + buzzes[n] or numbers[n]
```

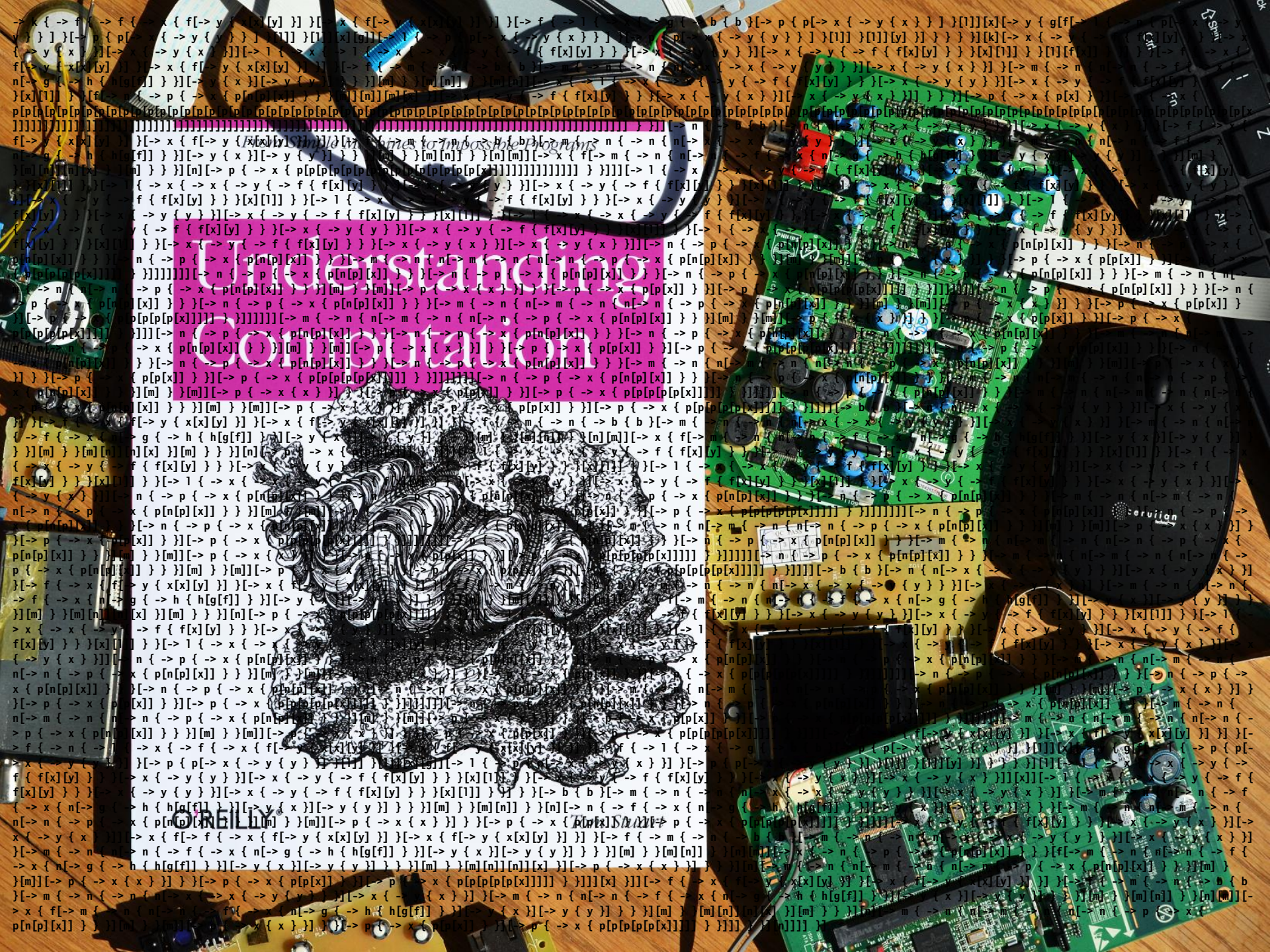
```
def fizzbuzz(n):
    return (
        'FizzBuzz' if n in range(0, 101, 3 * 5) else
        'Fizz' if n in range(0, 101, 3) else
        'Buzz' if n in range(0, 101, 5) else
        str(n))
```

```
fizzer = lambda n: 'Fizz' * (n % 3 == 0)
buzzer = lambda n: 'Buzz' * (n % 5 == 0)
number = lambda n: str(n) * (n % 3 != 0 and n % 5 != 0)
fizzbuzz = lambda n: fizzer(n) + buzzer(n) + number(n)
```

```
fizzbuzz = (
    lambda n:
        'Fizz' * (n % 3 == 0) +
        'Buzz' * (n % 5 == 0) +
        str(n) * (n % 3 != 0 and n % 5 != 0))
```

```
MAP[RANGE[ONE][HUNDRED]][-> n {
  IF[IS_ZERO[MOD[n][FIFTEEN]]][
    FIZZBUZZ
  ][IF[IS_ZERO[MOD[n][THREE]]][
    FIZZ
  ][IF[IS_ZERO[MOD[n][FIVE]]][
    BUZZ
  ][
    TO_DIGITS[n]
  ]]]
}]
```


From Simple Machines to Impossible Programs
Understanding Computation
O'REILLY®
Tom Stuart


Understanding
Concatenation
O'REILLY

A work of art is the unique result of a unique temperament.

Oscar Wilde