
The Forgotten Art of Structured Programming
@KevlinHenney

Outline Slides

32


33

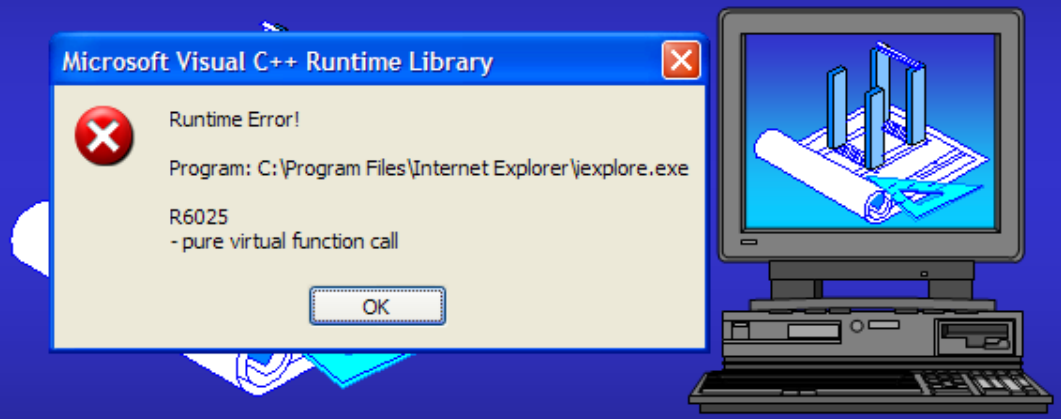

34


35


36


37


Combining Patterns



Pattern-Based Software Development

User Information

KH Kevlin Henney
kevlin@curbralan.com

Visual Studio profile

Sign out

⚠ TF400813: Resource not available for anonymous access. Client authentication required.
Reenter your credentials

Product Information

Visual Studio®
Professional 2013

License: MSDN Subscription
This product is licensed to:
kevlin@curbralan.com

This license will expire in 2147483647 days.

⚠ Your license has gone stale and must be updated. Check for an updated license to continue using this product.
Check for an updated license

License this product with a product key.
Change my product license

Ready to buy Visual Studio? Order online

Exit Visual Studio

0000420002107603560O

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

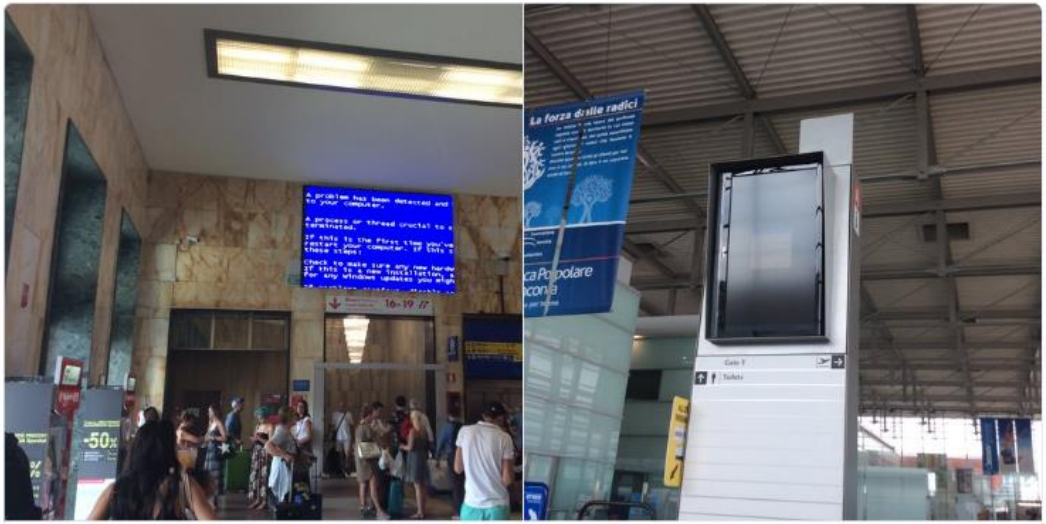
$retAdd.getDayPhone()

Payer / Facturé à:

7307904

Method of Payment /
Mode de paiement:

To / Dest.:



@tackline

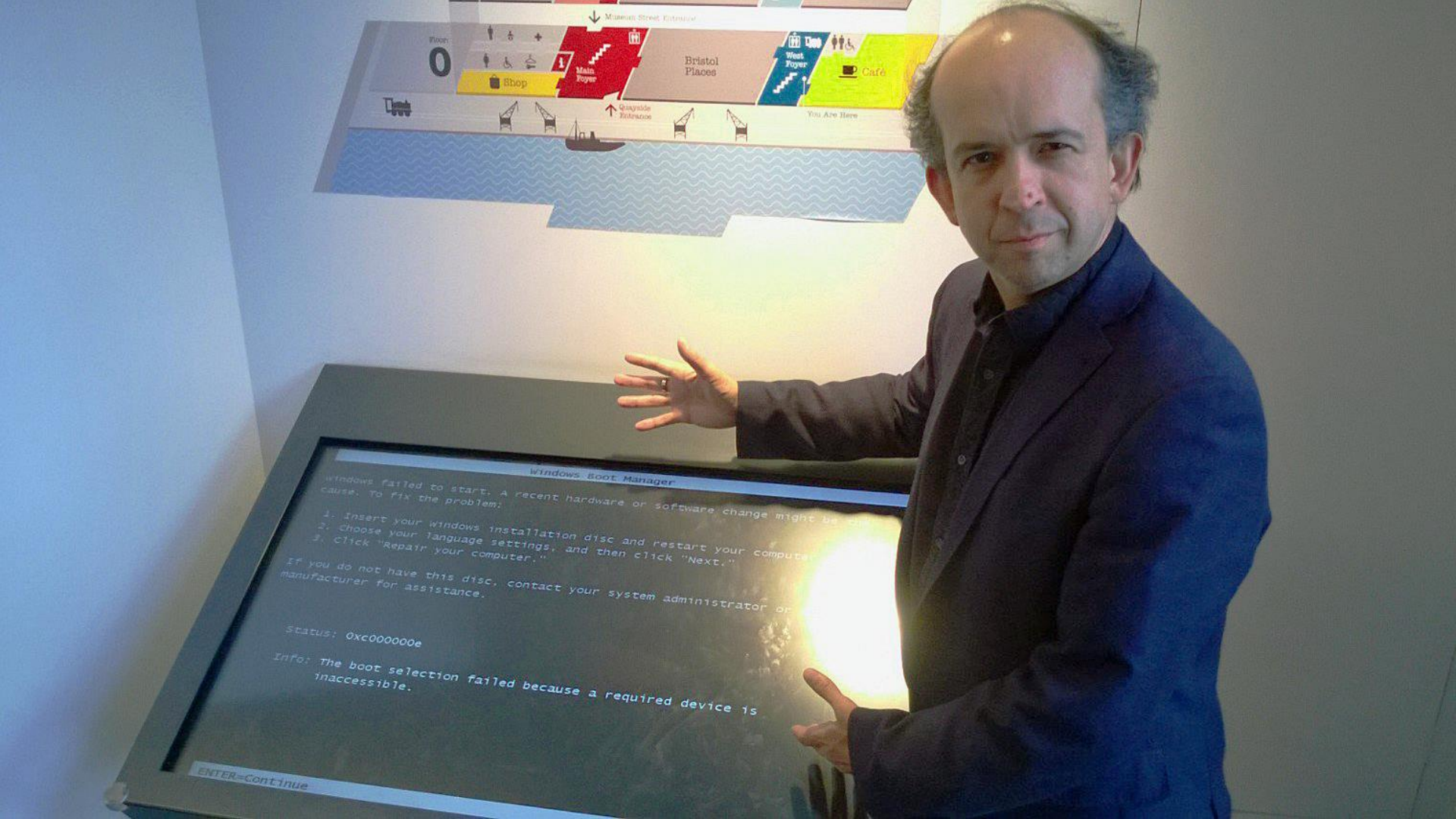
Follow

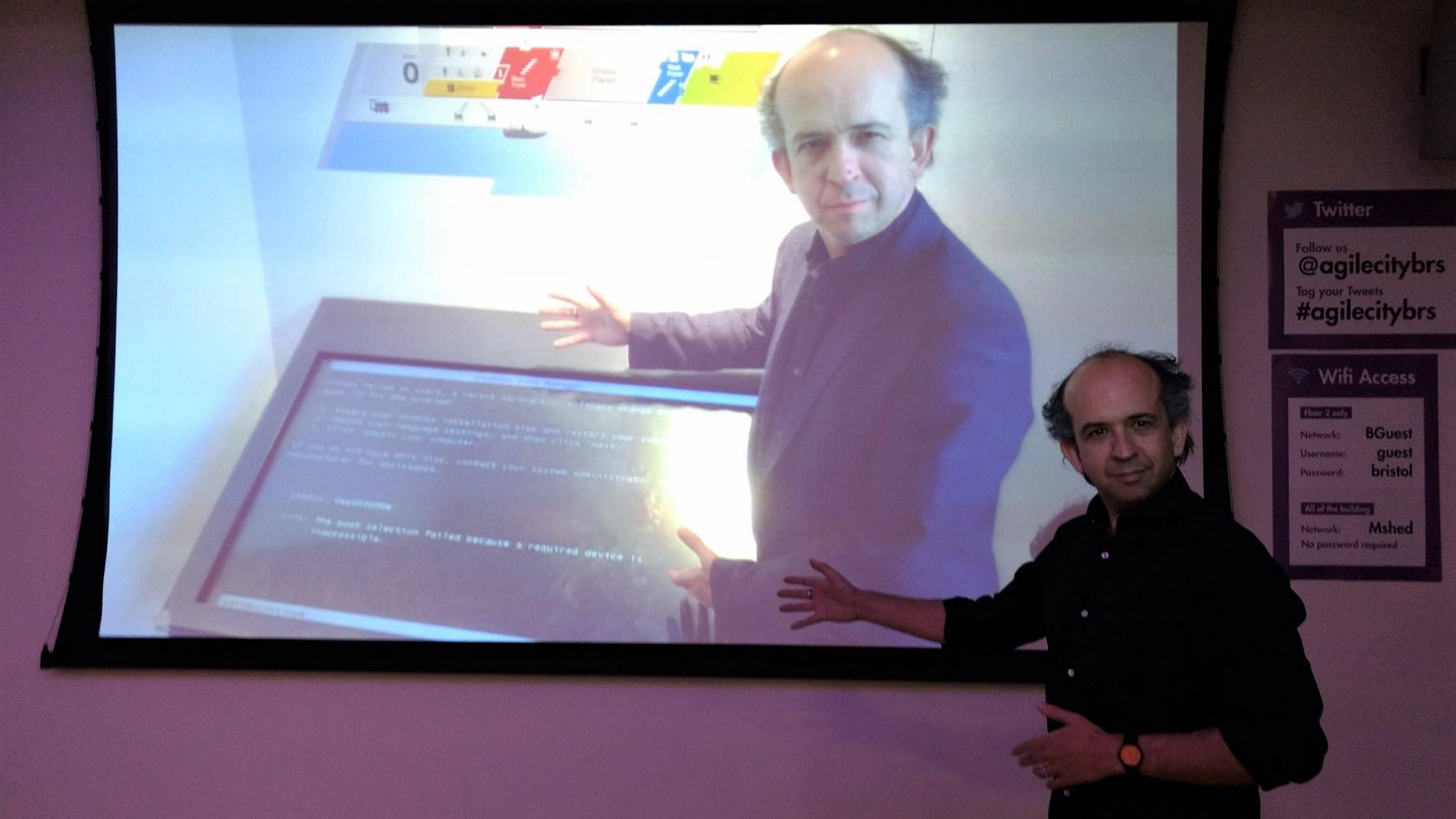
Arriving in Bologna, I saw a @KevlinHenney screen. Whilst queueing to leave Ancona another appeared as I waited.

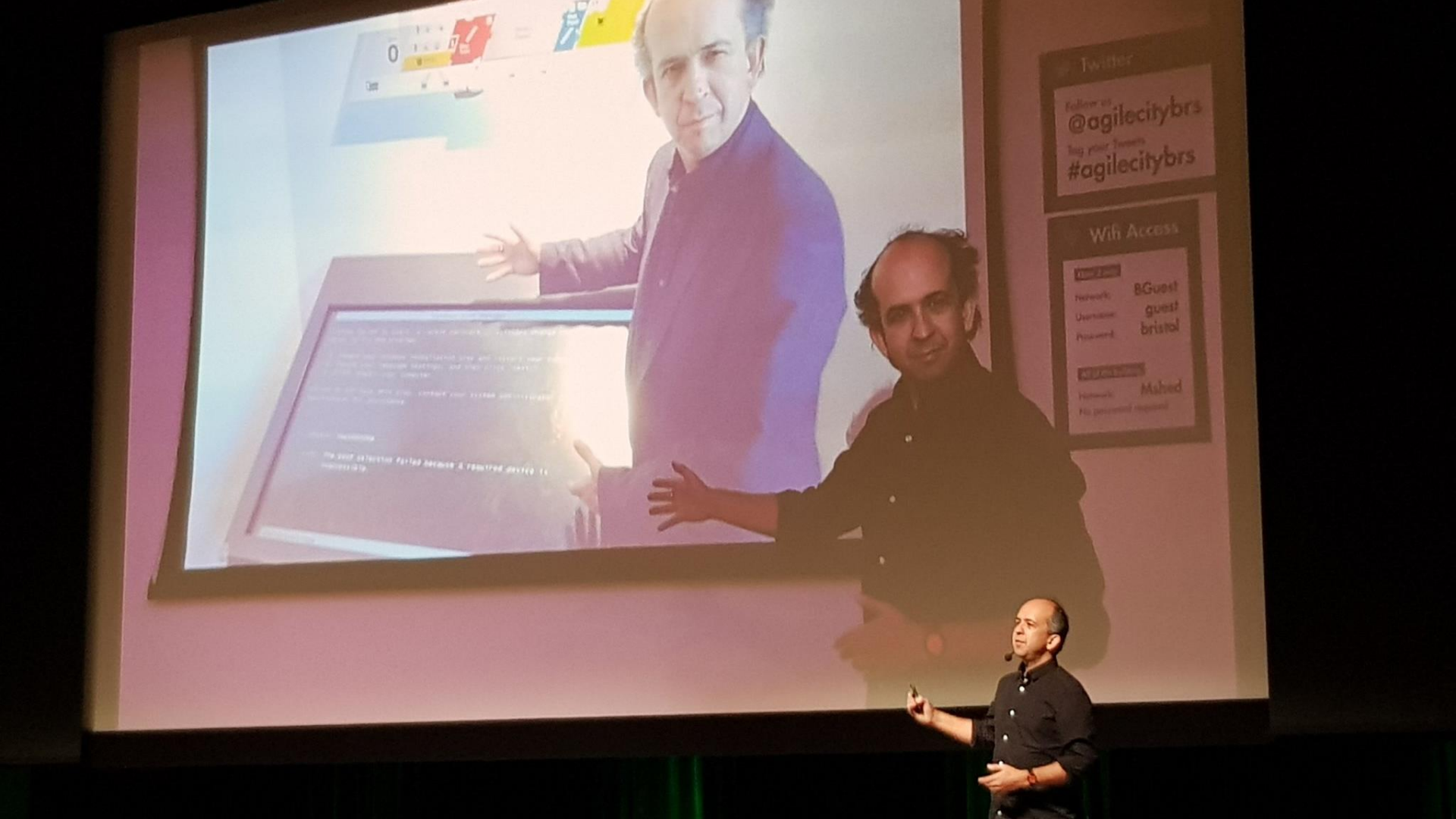
2:05 PM - 25 Jul 2016

3 2

https://twitter.com/tackline/status/757562488363843584


Floor
0
Shop
Main Foyer
Bristol Places
West Foyer
Café
Quayside Entrance
You Are Here
Windows Boot Manager
Windows failed to start. A recent hardware or software change might be
cause. To fix the problem:
1. Insert your windows installation disc and restart your computer
2. Choose your language settings, and then click "Next."
3. Click "Repair your computer."
If you do not have this disc, contact your system administrator or
manufacturer for assistance.
Status: 0xc000000e
Info: The boot selection failed because a required device is
inaccessible.
ENTER=Continue


Twitter
Follow us
@agilecitybrs
Tag your Tweets
#agilecitybrs
Wifi Access
Network: BGuest
Username: guest
Password: bristol
All of the building
Network: Mshed
No password required


Twitter
@agilecitybrs
#agilecitybrs
Wifi Access
BGuest
guest
bristol
Mahed


POLICE
PUBLIC
CALL
BOX
FREE
PUBLIC
PULL TO OPEN
SPACE
INVADERS
PLAYER
01

Our Reply

31 December 1969

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

SOFTWARE ENGINEERING

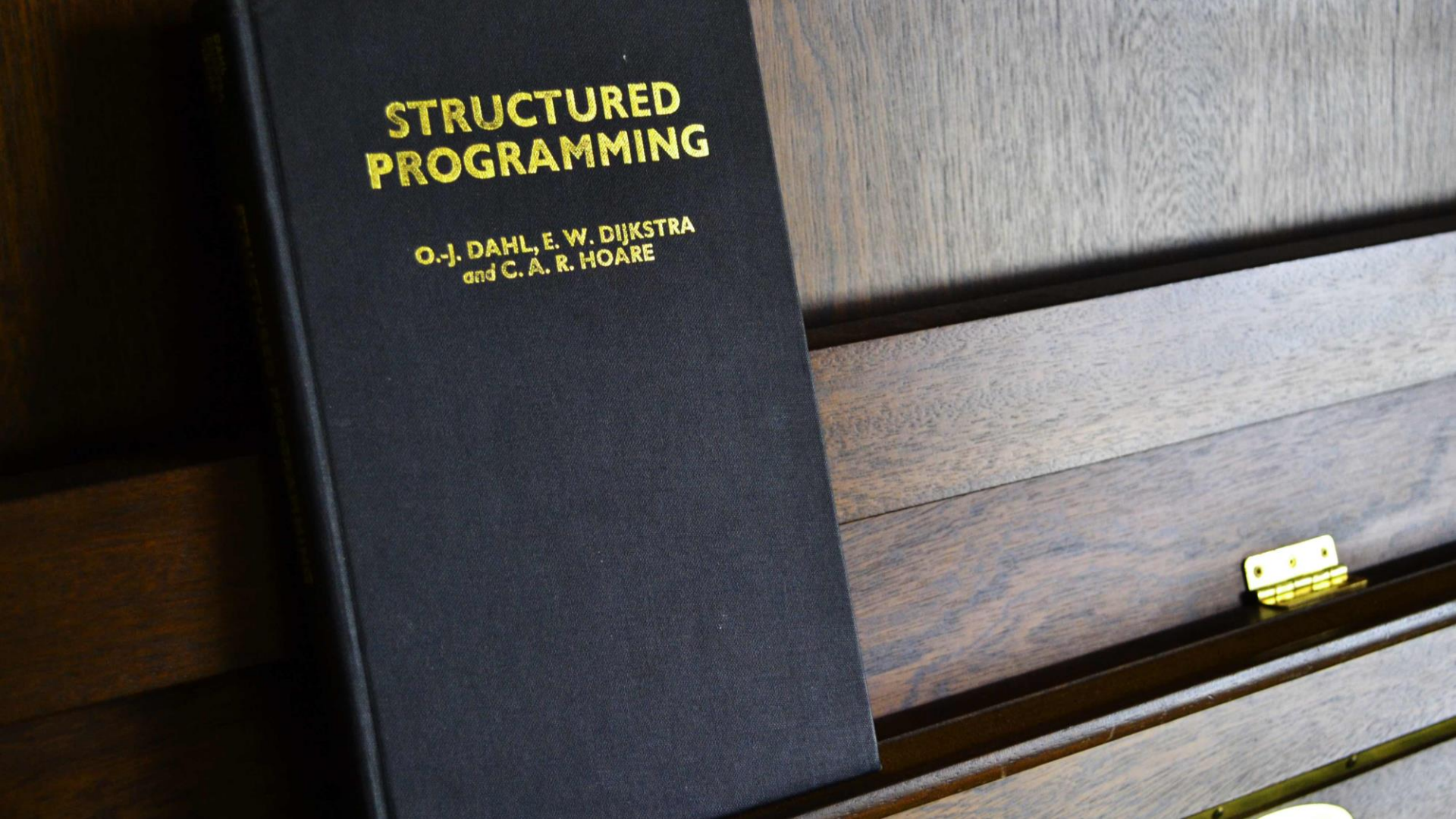
Report on a conference sponsored by the

NATO SCIENCE COMMITTEE

Garmisch, Germany, 7th to 11th October 1968


2001: A SPACE ODYSSEY


ATTENDEZ
PIETONS
RHUME
jacques borel


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

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.

The Paradigms of Programming

Robert W. Floyd
Stanford University

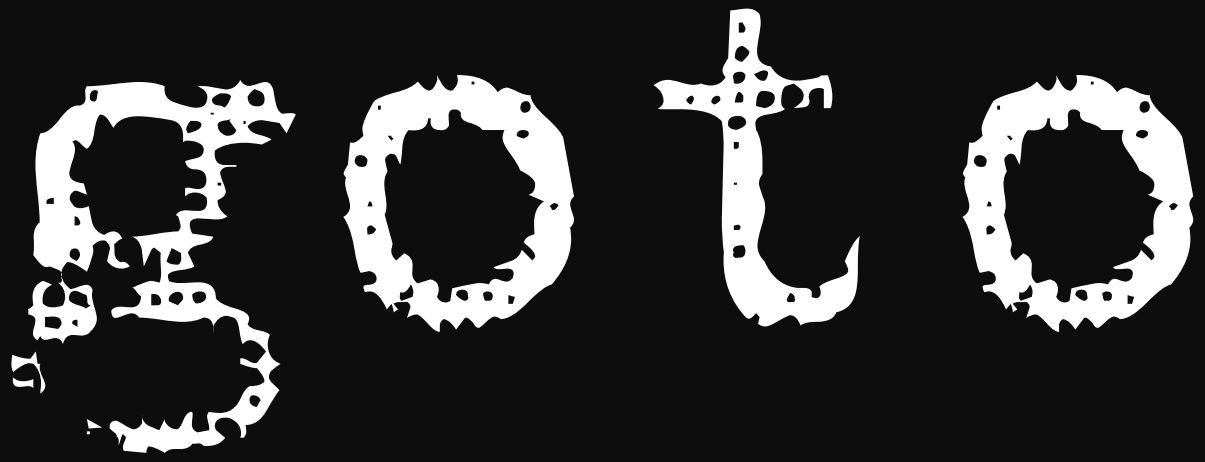
A familiar example of a paradigm of programming is the technique of *structured programming*, which appears to be the dominant paradigm in most current treatments of programming methodology.

Paradigm(pæ·radim, −dəim) . . . [a. F. *paradigme*, ad. . . .
. . . 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 [illegible] programming languages.

A familiar example of a paradigm of programming is the technique of *structured programming*, which ap[illegible] the [illegible] in most current [illegible] 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 [illegible] equations, say, the first level of decomposition would be into a stage of triangularizing the equations and a following stage of back-[illegible] 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 ith variable from the ith equation. Yet further decomposition would yield a fully detailed algorithm.


goto

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** *B* **then** *A*), alternative clauses (**if** *B* **then** *A*1 **else** *A*2), choice clauses as introduced by C. A. R. Hoare (case[i] of($A1, A2, \cdots, An$)), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1, B2 \rightarrow E2, \cdots, Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommendation to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the **go to** statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

REFERENCES:

1. WIRTH, NIKLAUS, AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June 1966), 413–432.
2. BÖHM, CORRADO, AND JACOPINI, GUISEPPE. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9* (May 1966), 366–371.

EDSGER W. DIJKSTRA
Technological University
Eindhoven, The Netherlands


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

/ WordFriday

snowclone, *noun*

- clichéd wording used as a template, typically originating in a single quote
- e.g., “X considered harmful”, “These aren't the Xs you're looking for”, “X is the new Y”, “It’s X, but not as we know it”, “No X left behind”, “It’s Xs all the way down”, “All your X are belong to us”

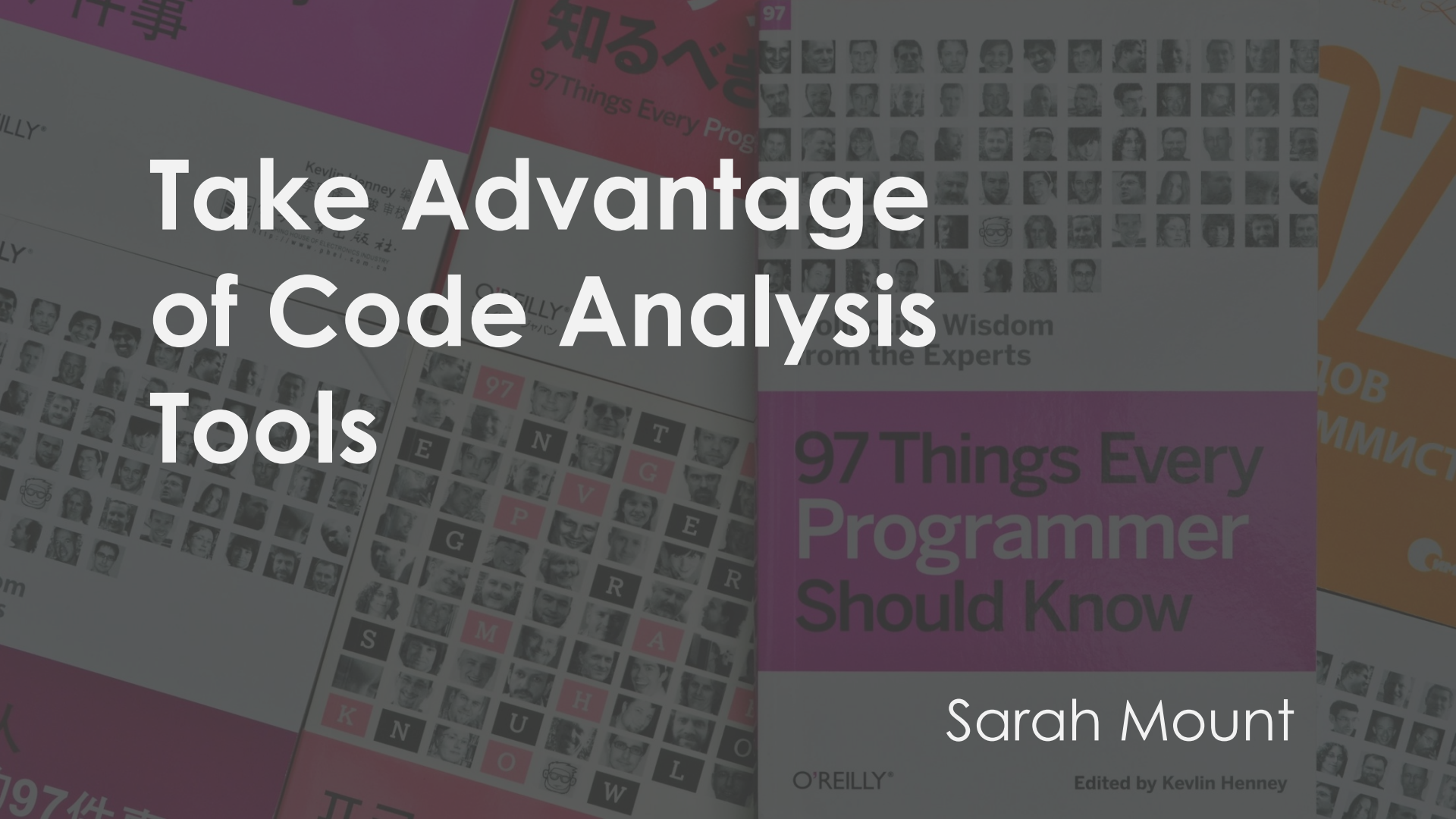
A Case against the GO TO Statement.

by Edsger W.Dijkstra
Technological University
Eindhoven, The Netherlands

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

```
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

Mike Bland
"Goto Fail, Heartbleed, and Unit Testing Culture"
https://martinfowler.com/articles/testing-culture.html



Take Advantage of Code Analysis Tools

Testing Is the Engineering Rigor of Software Development

Neal Ford

These bugs are as instructive as they were devastating: They were rooted in the same programmer optimism, overconfidence, and haste that strike projects of all sizes and domains.

Mike Bland
https://martinfowler.com/articles/testing-culture.html

These bugs arouse my passion because I've seen and lived the benefits of unit testing, and this strongly-imprinted experience compels me to reflect on how unit testing approaches could prevent defects as high-impact and high-profile as these SSL bugs.

Mike Bland
https://martinfowler.com/articles/testing-culture.html

```
int isleap(year)
int year;
{
    return year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
}
```

```
int isleap(year)
int year;
{
    if (year % 400 == 0)
        return 1;
    if (year % 100 == 0)
        return 0;
    if (year % 4 == 0)
        return 1;
    return 0;
}
```

```
int isleap(year)
int year;
{
    if (year % 400 == 0)
    {
        return 1;
    }
    if (year % 100 == 0)
    {
        return 0;
    }
    if (year % 4 == 0)
    {
        return 1;
    }
```

```
int isleap(year)
int year;
{
    if (year % 400 == 0)
    {
        return 1;
    }

    if (year % 100 == 0)
    {
        return 0;
    }

    if (year % 4 == 0)
    {
```

```
int isleap(year)
int year;
{
    if (year % 400 == 0)
        return 1;
    else if (year % 100 == 0)
        return 0;
    else if (year % 4 == 0)
        return 1;
    else
        return 0;
}
```

```
int isleap(year)
int year;
{
    if (year % 400 == 0)
        goto true;
    if (year % 100 == 0)
        goto false;
    if (year % 4 == 0)
        goto true;
false:
    return 0;
true:
    return 1;
}
```

```
int isleap(year)
int year;
{
    if (!(year % 400))
        goto true;
    if (!(year % 100))
        goto false;
    if (!(year % 4))
        goto true;
false:
    return 0;
true:
    return !0;
}
```

```
int isleap(year)
int year;
{
    if (!(year % 400))
true:   return !0;
    if (!(year % 100))
false:  return 0;
    if (!(year % 4))
        goto true;
    goto false;
}
```

```
      FUNCTION ISLEAP(YEAR)
      LOGICAL ISLEAP
      INTEGER YEAR
      IF (MOD(YEAR, 400) .EQ. 0) GOTO 20
      IF (MOD(YEAR, 100) .EQ. 0) GOTO 10
      IF (MOD(YEAR, 4) .EQ. 0) GOTO 20
10    ISLEAP = .FALSE.
      RETURN
20    ISLEAP = .TRUE.
      END
```

```
      FUNCTION ISLEAP(YEAR)
      LOGICAL ISLEAP
      INTEGER YEAR
      IF (MOD(YEAR, 400) .EQ. 0) GOTO 20
      IF (MOD(YEAR, 100) .EQ. 0) GOTO 10
      IF (MOD(YEAR, 4) .EQ. 0) GOTO 20
10    ISLEAP = .FALSE.
      RETURN
20    ISLEAP = .TRUE.
      RETURN
      END
```

```
      FUNCTION ISLEAP(YEAR)
      LOGICAL ISLEAP
      INTEGER YEAR
      IF (MOD(YEAR, 400) .EQ. 0) GOTO 20
      IF (MOD(YEAR, 100) .EQ. 0) GOTO 10
      IF (MOD(YEAR, 4) .EQ. 0) GOTO 20
10    ISLEAP = .FALSE.
      GOTO 30
20    ISLEAP = .TRUE.
30    RETURN
      END
```

```
      FUNCTION ISLEAP(YEAR)
      LOGICAL ISLEAP
      INTEGER YEAR
      IF (MOD(YEAR, 400) .EQ. 0) GOTO 20
      IF (MOD(YEAR, 100) .EQ. 0) GOTO 10
      IF (MOD(YEAR, 4) .EQ. 0) GOTO 20
10    ISLEAP = .FALSE.
      GOTO 30
20    ISLEAP = .TRUE.
      GOTO 30
30    RETURN
      END
```

```
      FUNCTION ISLEAP(YEAR)
      LOGICAL ISLEAP
      INTEGER YEAR
      IF (MOD(YEAR, 400) .EQ. 0) GOTO 20
      IF (MOD(YEAR, 100) .EQ. 0) GOTO 10
      IF (MOD(YEAR, 4) .EQ. 0) GOTO 20
10    ISLEAP = .FALSE.
      GOTO 30
20    ISLEAP = .TRUE.
      GOTO 30
30    CONTINUE
      RETURN
      END
```

```
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) THEN
        ISLEAP = .TRUE.
    ELSE IF (MOD(YEAR, 100) .EQ. 0) THEN
        ISLEAP = .FALSE.
    ELSE IF (MOD(YEAR, 4) .EQ. 0) THEN
        ISLEAP = .TRUE.
    ELSE
        ISLEAP = .FALSE.
    END IF
END
```

```
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) THEN
        ISLEAP = .TRUE.
    ELSE IF (MOD(YEAR, 100) .EQ. 0) THEN
        ISLEAP = .FALSE.
    ELSE IF (MOD(YEAR, 4) .EQ. 0) THEN
        ISLEAP = .TRUE.
    ELSE
        ISLEAP = .FALSE.
    END IF
END
```

A goto completely invalidates the high-level structure of the code.

Taligent's Guide to Designing Programs

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n=(count+7)/8;
    switch(count%8){
    case 0: do{ *to = *from++;
    case 7:     *to = *from++;
    case 6:     *to = *from++;
    case 5:     *to = *from++;
    case 4:     *to = *from++;
    case 3:     *to = *from++;
    case 2:     *to = *from++;
    case 1:     *to = *from++;
        }while(--n>0);
    }
}
```

```
send(to, from, count)
register short *to, *from;
register count
{
    register n=(count+7)/8;
    switch(count%8){
    case 0:     do{ *to = *from++;
    case 7:         *to = *from++;
    case 6:         *to = *from++;
    case 5:         *to = *from++;
    case 4:         *to = *from++;
    case 3:         *to = *from++;
    case 2:         *to = *from++;
    case 1:         *to = *from++;
        }while(--n>0);
    }
}
```

I feel a combination of pride and revulsion at this discovery.

Tom Duff

Many people have said that the worst feature of C is that switches don't break automatically before each case label. This code forms some sort of argument in that debate, but I'm not sure whether it's for or against.

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n=(count+7)/8;
    switch(count%8){
    case 0: do{ *to = *from++;
    case 7:     *to = *from++;
    case 6:     *to = *from++;
    case 5:     *to = *from++;
    case 4:     *to = *from++;
    case 3:     *to = *from++;
    case 2:     *to = *from++;
    case 1:     *to = *from++;
        }while(--n>0);
    }
}
```

Tom Duff

break

breaking bad

```
switch (line) {
case THING1:
    doit1();
    break;
case THING2:
    if (x == STUFF) {
       do_first_stuff();
       if (y == OTHER_STUFF)
           break;
       do_later_stuff();
    } /* coder meant to break to here... */
    initialize_modes_pointer();
    break;
default:
    processing();
} /* ...but actually broke to here! */
use_modes_pointer(); /* leaving the modes_pointer uninitialized */
```

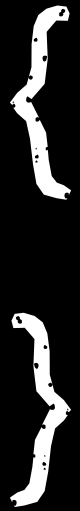
Peter van der Linden
Expert C Programming

break

break

brace

brace



continue

break

return

return

return

```
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) THEN
        ISLEAP = .TRUE.
    ELSE IF (MOD(YEAR, 100) .EQ. 0) THEN
        ISLEAP = .FALSE.
    ELSE IF (MOD(YEAR, 4) .EQ. 0) THEN
        ISLEAP = .TRUE.
    ELSE
        ISLEAP = .FALSE.
    END IF
END
```

```
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) THEN
        ISLEAP = .TRUE.
    ELSE IF (MOD(YEAR, 100) .EQ. 0) THEN
        ISLEAP = .FALSE.
    ELSE IF (MOD(YEAR, 4) .EQ. 0) THEN
        ISLEAP = .TRUE.
    ELSE
        ISLEAP = .FALSE.
    END IF
END
```

```
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) THEN
        ISLEAP = .TRUE.
        RETURN
    END IF
    IF (MOD(YEAR, 100) .EQ. 0) THEN
        ISLEAP = .FALSE.
        RETURN
    END IF
    IF (MOD(YEAR, 4) .EQ. 0) THEN
        ISLEAP = .TRUE.
        RETURN
    END IF
    ISLEAP = .FALSE.
END
```

```
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) THEN
        ISLEAP = .TRUE.
        RETURN
    END IF
    IF (MOD(YEAR, 100) .EQ. 0) THEN
        ISLEAP = .FALSE.
        RETURN
    END IF
    IF (MOD(YEAR, 4) .EQ. 0) THEN
        ISLEAP = .TRUE.
        RETURN
    END IF
    ISLEAP = .FALSE.
END
```

```
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) THEN
        ISLEAP = .TRUE.
        RETURN
    END IF
    IF (MOD(YEAR, 100) .EQ. 0) THEN
        ISLEAP = .FALSE.
        RETURN
    END IF
    IF (MOD(YEAR, 4) .EQ. 0) THEN
        ISLEAP = .TRUE.
        RETURN
    END IF
    ISLEAP = .FALSE.
END
```

```
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) THEN
        ISLEAP = .TRUE.
        RETURN
    ELSE IF (MOD(YEAR, 100) .EQ. 0) THEN
        ISLEAP = .FALSE.
        RETURN
    ELSE IF (MOD(YEAR, 4) .EQ. 0) THEN
        ISLEAP = .TRUE.
        RETURN
    ELSE
        ISLEAP = .FALSE.
        RETURN
    END IF
END
```

```
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) THEN
        ISLEAP = .TRUE.
        RETURN
    ELSE IF (MOD(YEAR, 100) .EQ. 0) THEN
        ISLEAP = .FALSE.
        RETURN
    ELSE IF (MOD(YEAR, 4) .EQ. 0) THEN
        ISLEAP = .TRUE.
        RETURN
    ELSE
        ISLEAP = .FALSE.
        RETURN
    END IF
END
```

```
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) THEN
        ISLEAP = .TRUE.
    ELSE IF (MOD(YEAR, 100) .EQ. 0) THEN
        ISLEAP = .FALSE.
    ELSE IF (MOD(YEAR, 4) .EQ. 0) THEN
        ISLEAP = .TRUE.
    ELSE
        ISLEAP = .FALSE.
    END IF
END
```

```
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) THEN
        ISLEAP = .TRUE.
    ELSE IF (MOD(YEAR, 100) .EQ. 0) THEN
        ISLEAP = .FALSE.
    ELSE IF (MOD(YEAR, 4) .EQ. 0) THEN
        ISLEAP = .TRUE.
    ELSE
        ISLEAP = .FALSE.
    END IF
END
```

```
def isLeapYear(year)
{
    return year % 4 == 0 && year % 100 != 0 || year % 400 == 0
}
```

```
def isLeapYear(year)
{
    year % 4 == 0 && year % 100 != 0 || year % 400 == 0
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        return true
    else if (year % 100 == 0)
        return false
    else if (year % 4 == 0)
        return true
    else
        return false
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        return true
    if (year % 100 == 0)
        return false
    if (year % 4 == 0)
        return true
    return false
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        true
    else if (year % 100 == 0)
        false
    else if (year % 4 == 0)
        true
    else
        false
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        return true
    if (year % 100 == 0)
        return false
    if (year % 4 == 0)
        return true
    return false
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        true
    else if (year % 100 == 0)
        false
    else if (year % 4 == 0)
        true
    else
        false
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        return true
    if (year % 100 == 0)
        return false
    if (year % 4 == 0)
        return true
    false
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        true
    else if (year % 100 == 0)
        false
    else if (year % 4 == 0)
        true
    else
        false
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        return true
    if (year % 100 == 0)
        return false
    if (year % 4 == 0)
        return true
    return false
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        ...
    else if (year % 100 == 0)
        ...
    else if (year % 4 == 0)
        ...
    else
        ...
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        ...
    if (year % 100 == 0)
        ...
    if (year % 4 == 0)
        ...
    return false
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        true
    else if (year % 100 == 0)
        false
    else if (year % 4 == 0)
        true
    else
        false
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        return true
    if (year % 100 == 0)
        return false
    if (year % 4 == 0)
        return true
    return false
}
```

```
def isLeapYear(year)
{
    if (year % 400 == 0)
        true
    else if (year % 100 == 0)
        false
    else if (year % 4 == 0)
        true
    else
        false
}
```

```
proc is leap year = (int year) bool:
    if year mod 400 = 0 then
        true
    elif year mod 100 = 0 then
        false
    elif year mod 4 = 0 then
        true
    else
        false
    fi;
```

Revised Report on the Algorithmic Language

Algol 68

Edited by
A. van Wijngaarden, B. J. Mailloux,
J. E. L. Peck, C. H. A. Koster, M. Sintzoff


Revised Report
on the Algorithmic Language
Algol 68
int
long
void
bool
char
union
short
struct
Edited by
A. van Wijngaarden, B. J. Mailloux,

```
isLeapYear year =
    if year `mod` 400 == 0 then
        True
    else if year `mod` 100 == 0 then
        False
    else if year `mod` 4 == 0 then
        True
    else
        False
```

```
function IsLeapYear(Year: Integer): Boolean;
begin
    if Year mod 400 = 0 then
        IsLeapYear := True
    else if Year mod 100 = 0 then
        IsLeapYear := False
    else if Year mod 4 = 0 then
        IsLeapYear := True
    else
        IsLeapYear := False
end;
```

```
func IsLeapYear(year int) bool {
	if year%400 == 0 {
		return true
	} else if year%100 == 0 {
		return false
	} else if year%4 == 0 {
		return true
	} else {
		return false
	}
}
```

```
func IsLeapYear(year int) (result bool) {
	if year%400 == 0 {
		result = true
	} else if year%100 == 0 {
		result = false
	} else if year%4 == 0 {
		result = true
	} else {
		result = false
	}
	return
}
```

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

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

```
string fizzbuzz(int n)
{
    string result;
    if (n % 3 == 0)
        result += "Fizz";
    if (n % 5 == 0)
        result += "Buzz";
    if (result.empty())
        result = to_string(n);
    return result;
}
```

```
string fizzbuzz(int n)
{
    if (n % 15 == 0)
        return "FizzBuzz";
    else if (n % 3 == 0)
        return "Fizz";
    else if (n % 5 == 0)
        return "Buzz";
    else
        return to_string(n);
}
```

```
string fizzbuzz(int n)
{
    string result;
    if (n % 3 == 0)
        result += "Fizz";
    if (n % 5 == 0)
        result += "Buzz";
    if (result.empty())
        result = to_string(n);
    return result;
}
```

```
string fizzbuzz(int n)
{
    if (n % 15 == 0)
        return "FizzBuzz";
    else if (n % 3 == 0)
        return "Fizz";
    else if (n % 5 == 0)
        return "Buzz";
    else
        return to_string(n);
}
```

```
string fizzbuzz(int n)
{
    string result;
    if (n % 3 == 0)
        ...
    if (n % 5 == 0)
        ...
    if (result.empty())
        ...
    return result;
}
```

```
string fizzbuzz(int n)
{
    if (n % 15 == 0)
        ...
    else if (n % 3 == 0)
        ...
    else if (n % 5 == 0)
        ...
    else
        ...
}
```

```
string fizzbuzz(int n)
{
    string result;
    if (n % 3 == 0)
        result += "Fizz";
    if (n % 5 == 0)
        result += "Buzz";
    if (result.empty())
        result = to_string(n);
    return result;
}
```

```
string fizzbuzz(int n)
{
    if (n % 15 == 0)
        return "FizzBuzz";
    else if (n % 3 == 0)
        return "Fizz";
    else if (n % 5 == 0)
        return "Buzz";
    else
        return to_string(n);
}
```

```
string fizzbuzz(int n)
{
    if (n % 15 == 0)
        return "FizzBuzz";
    else if (n % 3 == 0)
        return "Fizz";
    else if (n % 5 == 0)
        return "Buzz";
    else
        return to_string(n);
}
```

```
string fizzbuzz(int n)
{
    return
        (n % 15 == 0) ?
            "FizzBuzz" :
        (n % 3  == 0) ?
            "Fizz" :
        (n % 5  == 0) ?
            "Buzz" :
            to_string(n);
}
```

The default action is executed only if some previous actions were not executed.

Maciej Piróg
“FizzBuzz in Haskell by Embedding a Domain-Specific Language”

```
string fizzbuzz(int n)
{
    static const string fizzed[] { "", "Fizz" };
    static const string buzzed[] { "", "Buzz" };
    const string result =
        fizzed[n % 3 == 0] + buzzed[n % 5 == 0];
    return result.empty() ? to_string(n) : result;
}
```

The default action is executed only if some previous actions were not executed.

We ask if we can accomplish this without having to check the conditions for the previous actions twice; in other words, if we can make the control flow follow the information flow without loosing modularity.

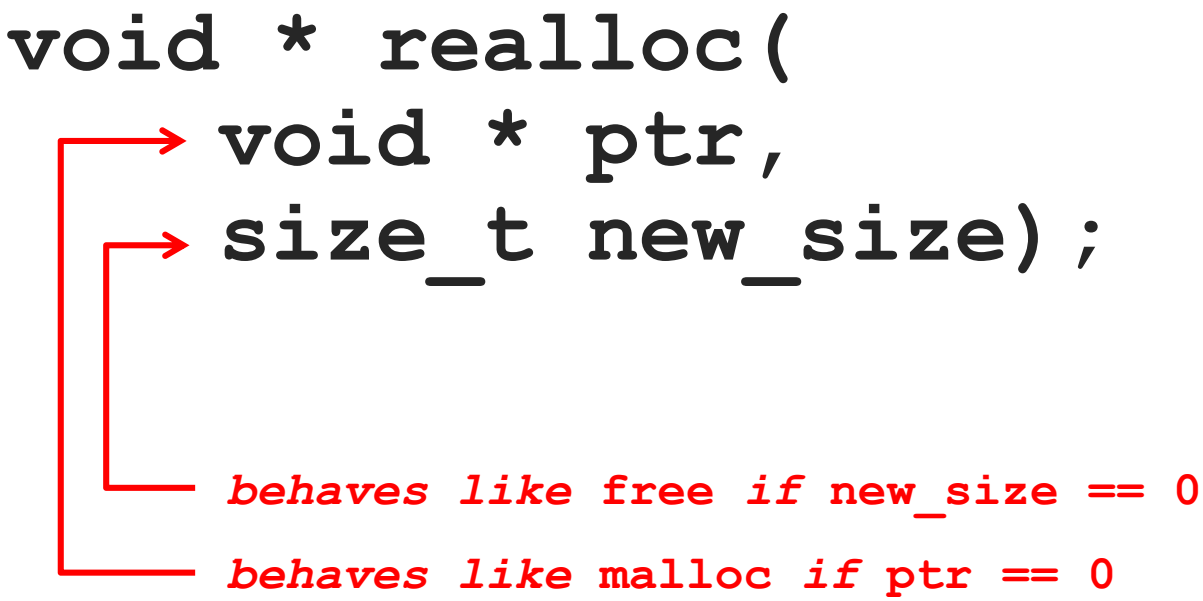
Maciej Piróg
“FizzBuzz in Haskell by Embedding a Domain-Specific Language”

```
string fizzbuzz(int n)
{
    auto fizz =
        [=](function<string(string)> f)
        {
            return
                n % 3 == 0
                    ? [=](auto) { return "Fizz" + f(""); }
                    : f;
        };
    auto buzz =
        [=](function<string(string)> f)
        {
            return
                n % 5 == 0
                    ? [=](auto) { return "Buzz" + f(""); }
                    : f;
        };
    auto id = [](auto s) { return s; };
    return fizz(buzz(id))(to_string(n));
}
```

```
string fizzbuzz(int n)
{
    auto test =
        [=](auto d, auto s, function<string(string)> f)
        {
            return
                n % d == 0
                    ? [=](string) { return s + f(""); }
                    : f;
        };
    auto fizz = bind(test, 3, "Fizz", _1);
    auto buzz = bind(test, 5, "Buzz", _1);
    auto id = [](auto s) { return s; };
    return fizz(buzz(id))(to_string(n));
}
```

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

Oscar Wilde

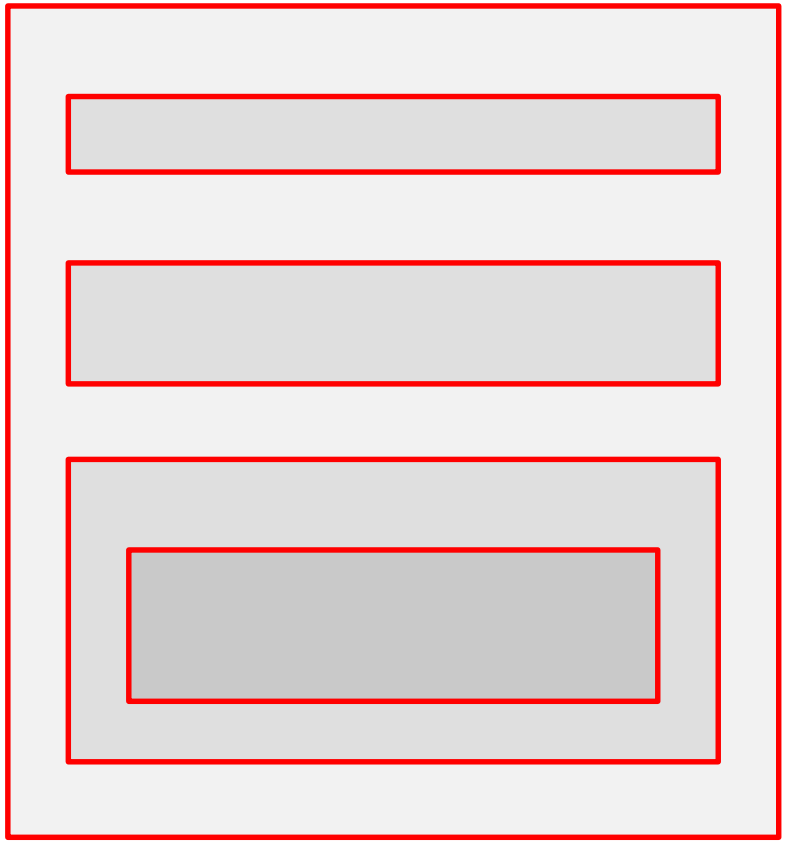

void * realloc(
void * ptr,
size_t new_size);
behaves like free if new_size == 0
behaves like malloc if ptr == 0

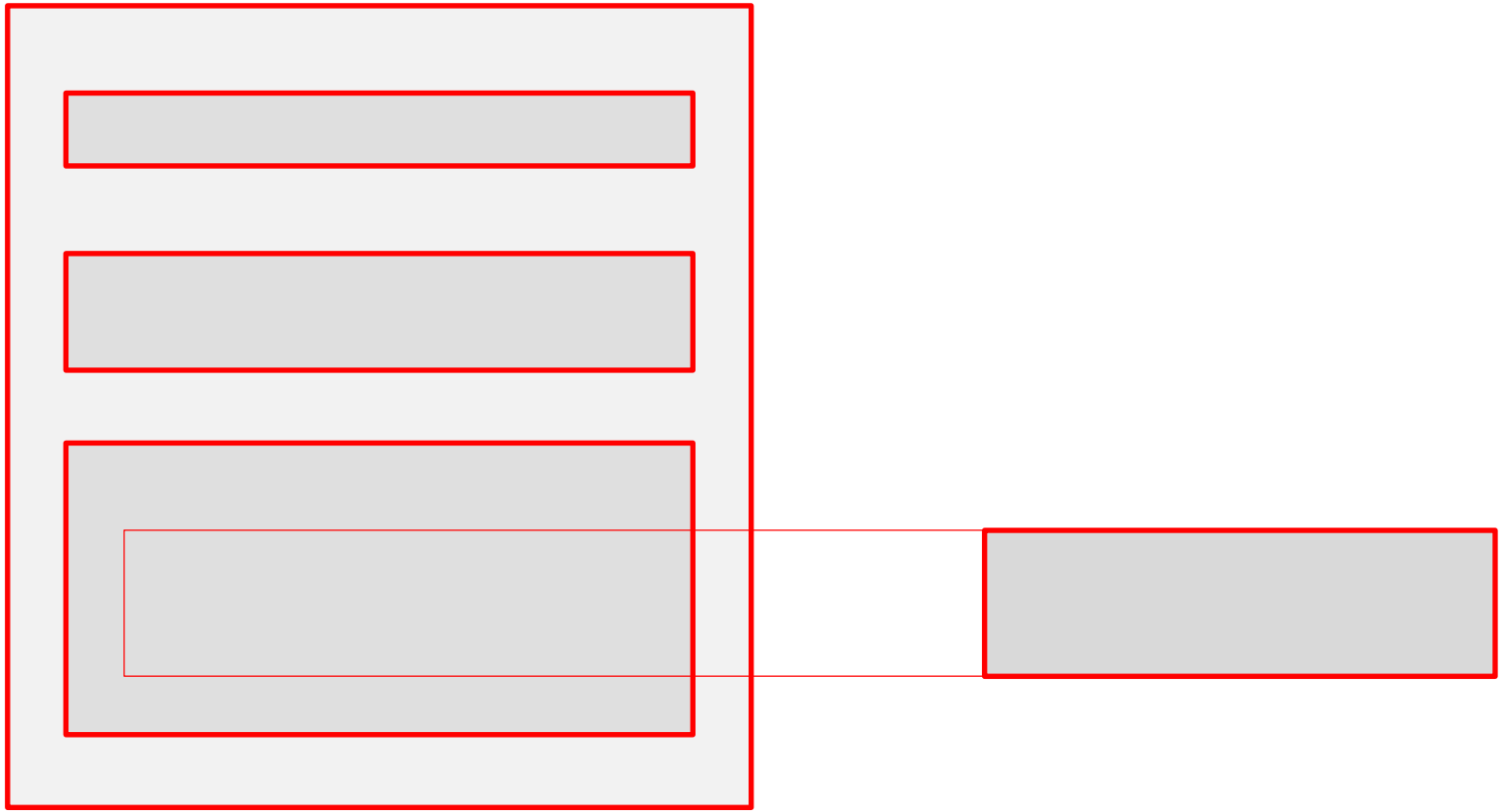
```
void * realloc(void * ptr, size_t new_size)
{
    if (!ptr)
    {
        return malloc(new_size);
    }
    if (new_size == 0)
    {
        free(ptr);
        return nullptr;
    }
    if (ptr == _Resize(ptr, new_size) && _Size(ptr) >= new_size)
    {
        return ptr;
    }
    void * ptr_ptr = malloc(new_size);
    if (!ptr_new)
    {
        return nullptr;
    }
    memcpy(ptr_new, ptr, std::min(_Size(ptr), new_size));
    free(ptr);
    return ptr_new;
}
```

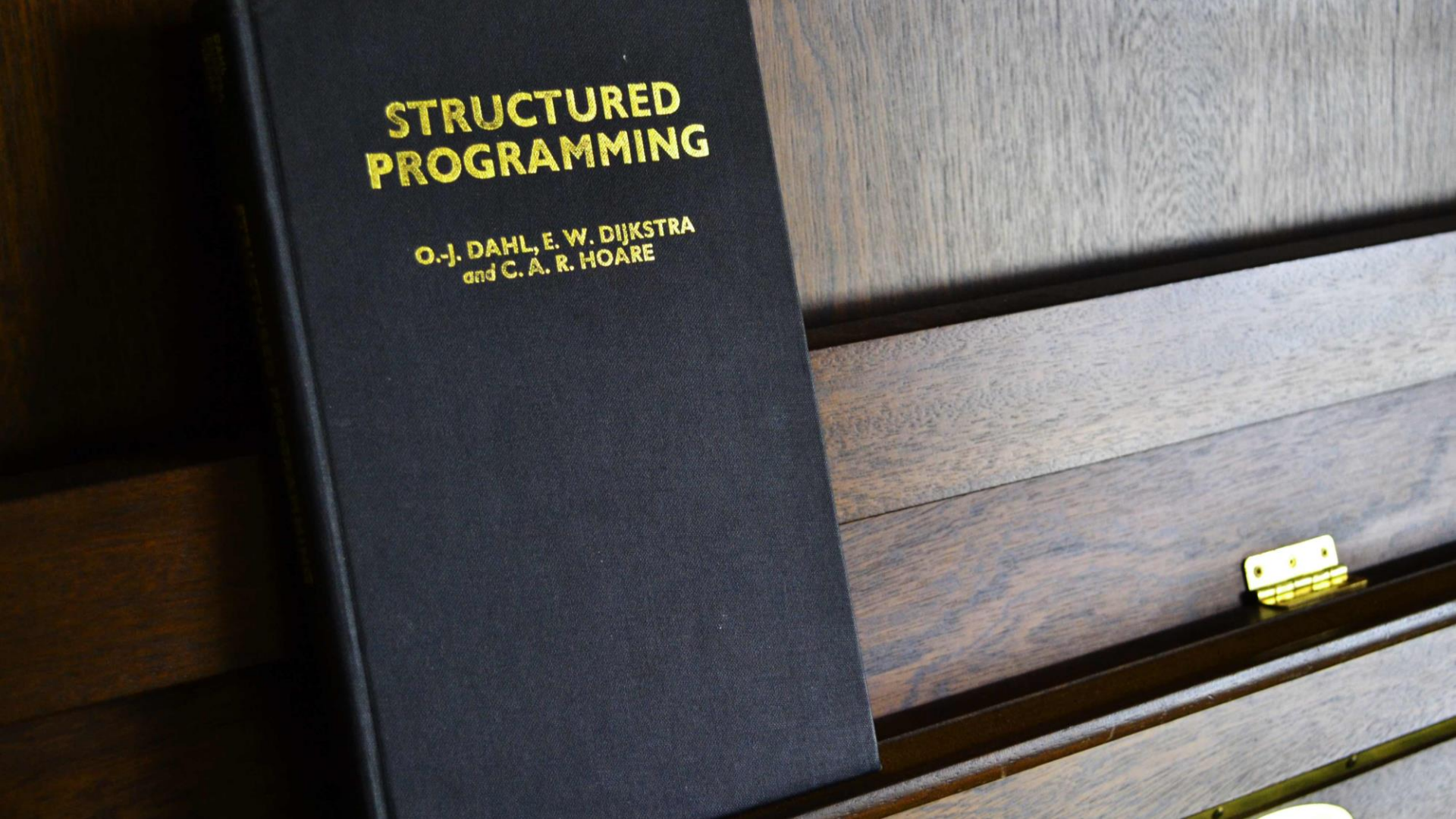
```
void * realloc(void * ptr, size_t new_size)
{
    if (!ptr)
    {
        ptr = malloc(new_size);
    }
    else if (new_size == 0)
    {
        free(ptr);
        ptr = nullptr;
    }
    else if (ptr != _Resize(ptr, new_size) || _Size(ptr) < new_size)
    {
        if (void * ptr_new = malloc(new_size))
        {
            memcpy(ptr_new, ptr, std::min(_Size(ptr), new_size));
            free(ptr);
            ptr = ptr_new;
        }
    }
    return ptr;
}
```

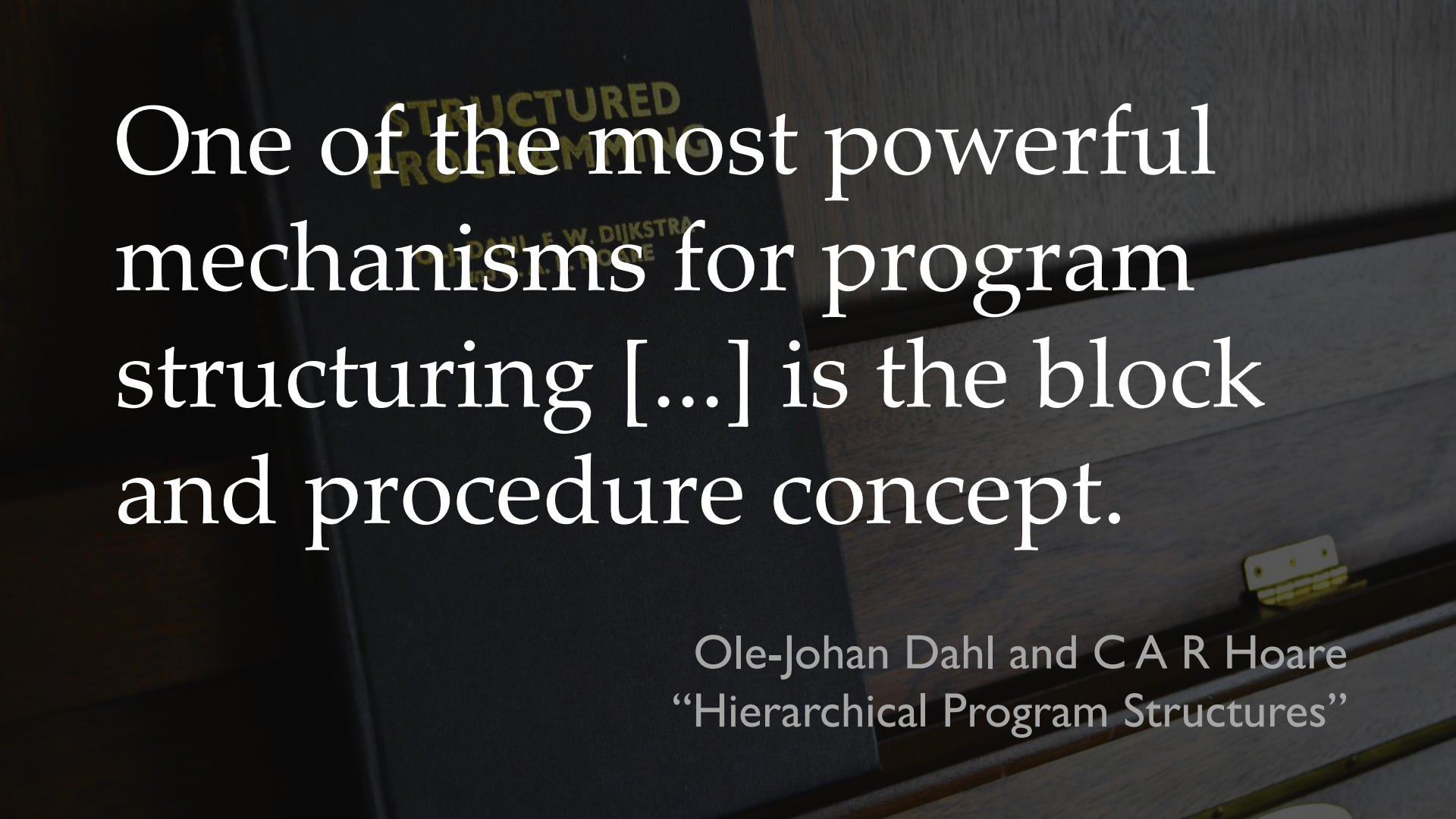
```
void * realloc(void * ptr, size_t new_size)
{
    if (!ptr)
    {
        ptr = malloc(new_size);
    }
    else if (new_size == 0)
    {
        free(ptr);
        ptr = nullptr;
    }
    else if (ptr != _Resize(ptr, new_size) || _Size(ptr) < new_size)
    {
        if (void * ptr_new = malloc(new_size))
        {
            memcpy(ptr_new, ptr, std::min(_Size(ptr), new_size));
            free(ptr);
            ptr = ptr_new;
        }
    }
    return ptr;
}
```

```
void * realloc(void * ptr, size_t new_size)
{
    if (!ptr)
    {
        ptr = malloc(new_size);
    }
    else if (new_size == 0)
    {
        free(ptr);
        ptr = nullptr;
    }
    else if (ptr != _Resize(ptr, new_size) || _Size(ptr) < new_size)
    {
        if (void * ptr_new = malloc(new_size))
        {
            memcpy(ptr_new, ptr, std::min(_Size(ptr), new_size));
            free(ptr);
            ptr = ptr_new;
        }
    }
    return ptr;
}
```






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



One of the most powerful mechanisms for program structuring [...] is the block and procedure concept.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"

$$
\begin{array}{l|ll}
 & \mathrm{Ord}\ 1(V) & \Rightarrow R \\
 & 0 & 0 \\
 & m \times \sigma & m \times \sigma
\end{array}
$$

$$
\begin{array}{l|ll}
 & V & \Rightarrow Z \\
V & 0 & 0 \\
S & m \times \sigma & m \times \sigma
\end{array}
$$

$$
\begin{array}{l|l}
 & W1(m-1)\left[\begin{array}{l|ll}
Z \Rightarrow Z & i & \Rightarrow \varepsilon \\
0\quad 1 & & \\
\sigma\quad \sigma & 1.n & 1.n
\end{array}\right. \\
V & \left. W\left[\begin{array}{l|ll}
\varepsilon \geq 0 \overset{\rightarrow}{\cdot} & Z < Z \overset{\rightarrow}{\cdot} & \left[\begin{array}{l|l}
Z \Rightarrow Z & \varepsilon - 1 \Rightarrow \varepsilon \\
0\quad 0 & \\
\varepsilon\quad \varepsilon + 1 & \\
\sigma\quad \sigma &
\end{array}\right] \\
1\quad 0 & & \\
\quad \varepsilon & & \\
\sigma\quad \sigma & \overline{Z < Z} \overset{\rightarrow}{\cdot} & \left[\begin{array}{l|l}
Z \Rightarrow Z & \mathrm{Fin}^3 \\
0\quad 0 & \\
\quad \varepsilon + 1 & \\
\sigma\quad \sigma &
\end{array}\right]
\end{array}\right] \\
K & \left.\begin{array}{l}
\varepsilon \quad = -1 \overset{\rightarrow}{\cdot} Z \Rightarrow Z \\
\qquad\quad 1\quad 0 \\
\qquad\quad\ \ 0 \\
1.n\quad 1.n\quad \sigma\quad \sigma
\end{array}\right] \\
S &
\end{array}
$$

$$
\begin{array}{l|ll}
 & Z & \Rightarrow R \\
V & 0 & 0 \\
S & m \times \sigma & m \times \sigma
\end{array}
$$

sequence

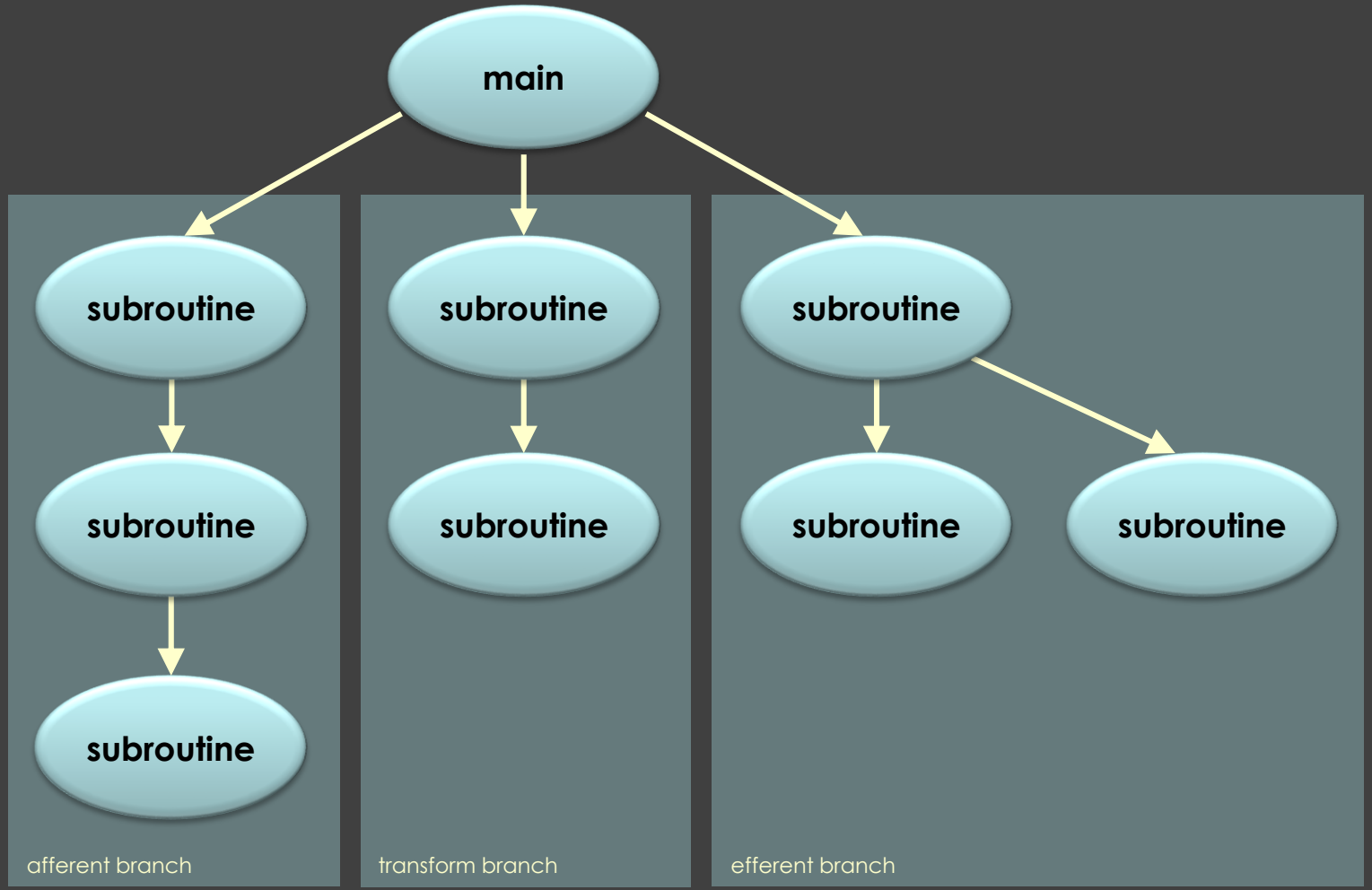
selection

iteration

Main Program and Subroutine

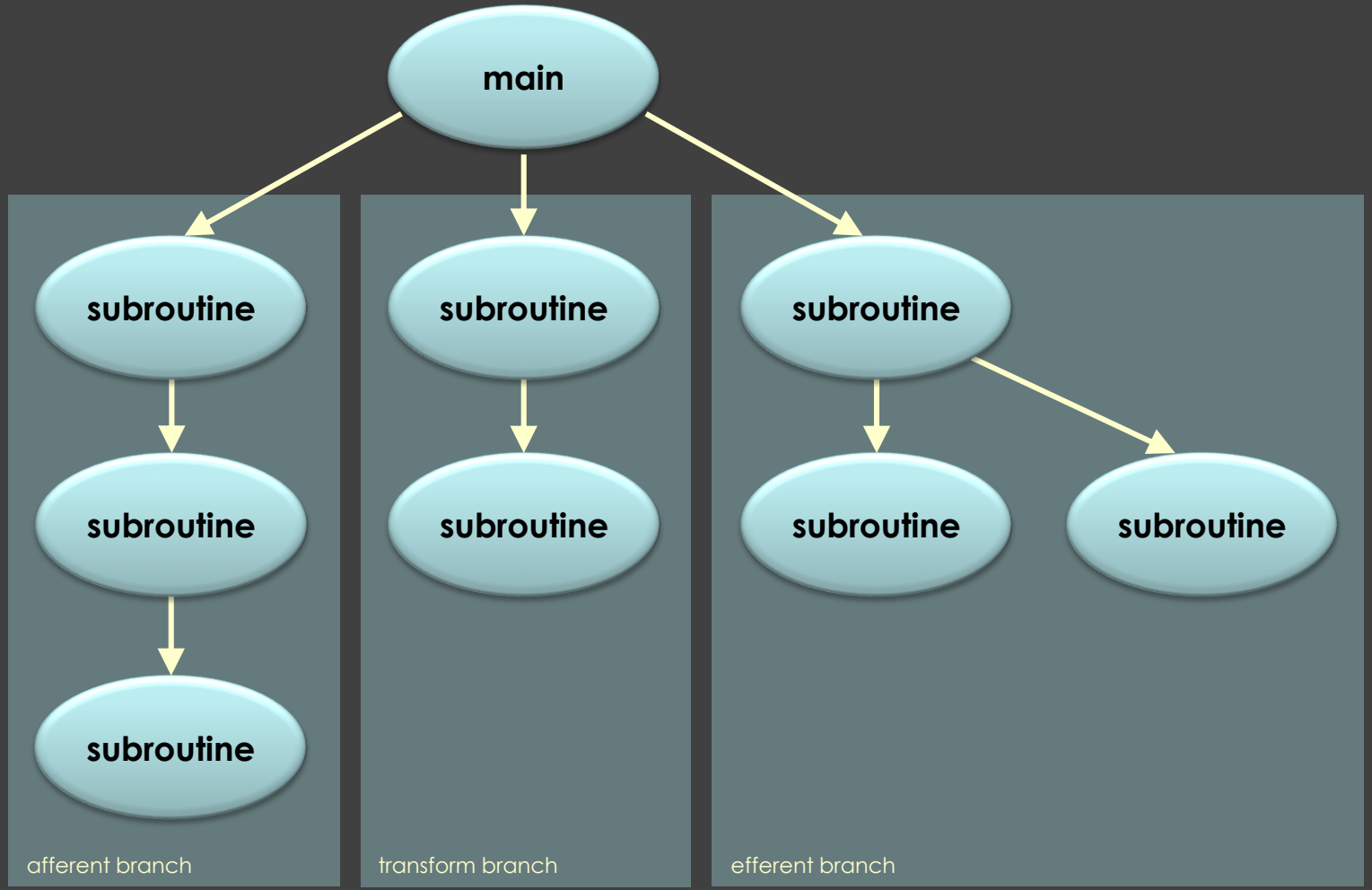
The goal is to decompose a program into smaller pieces to help achieve modifiability. A program is decomposed hierarchically.

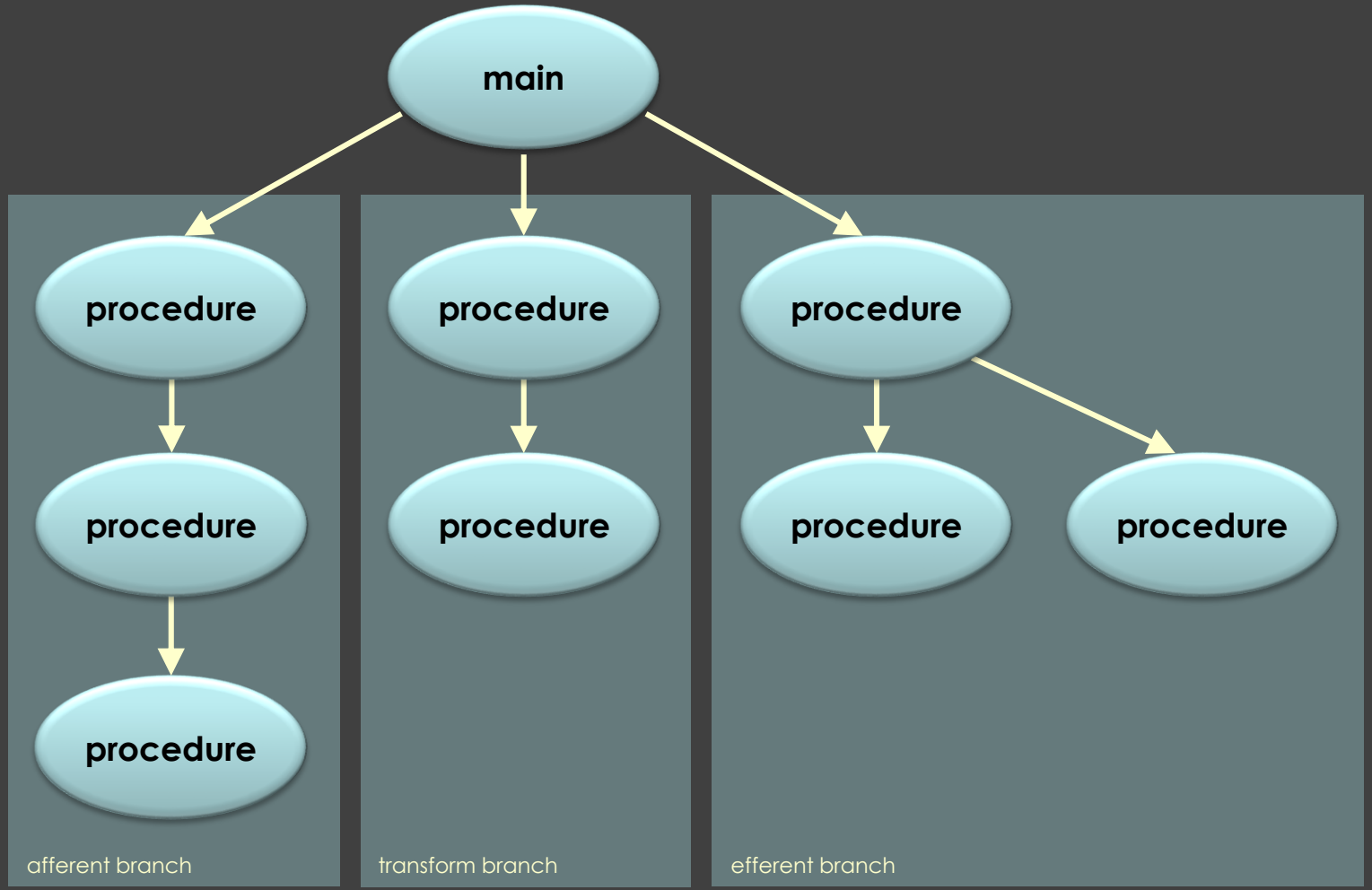
Len Bass, Paul Clements & Rick Kazman
Software Architecture in Practice

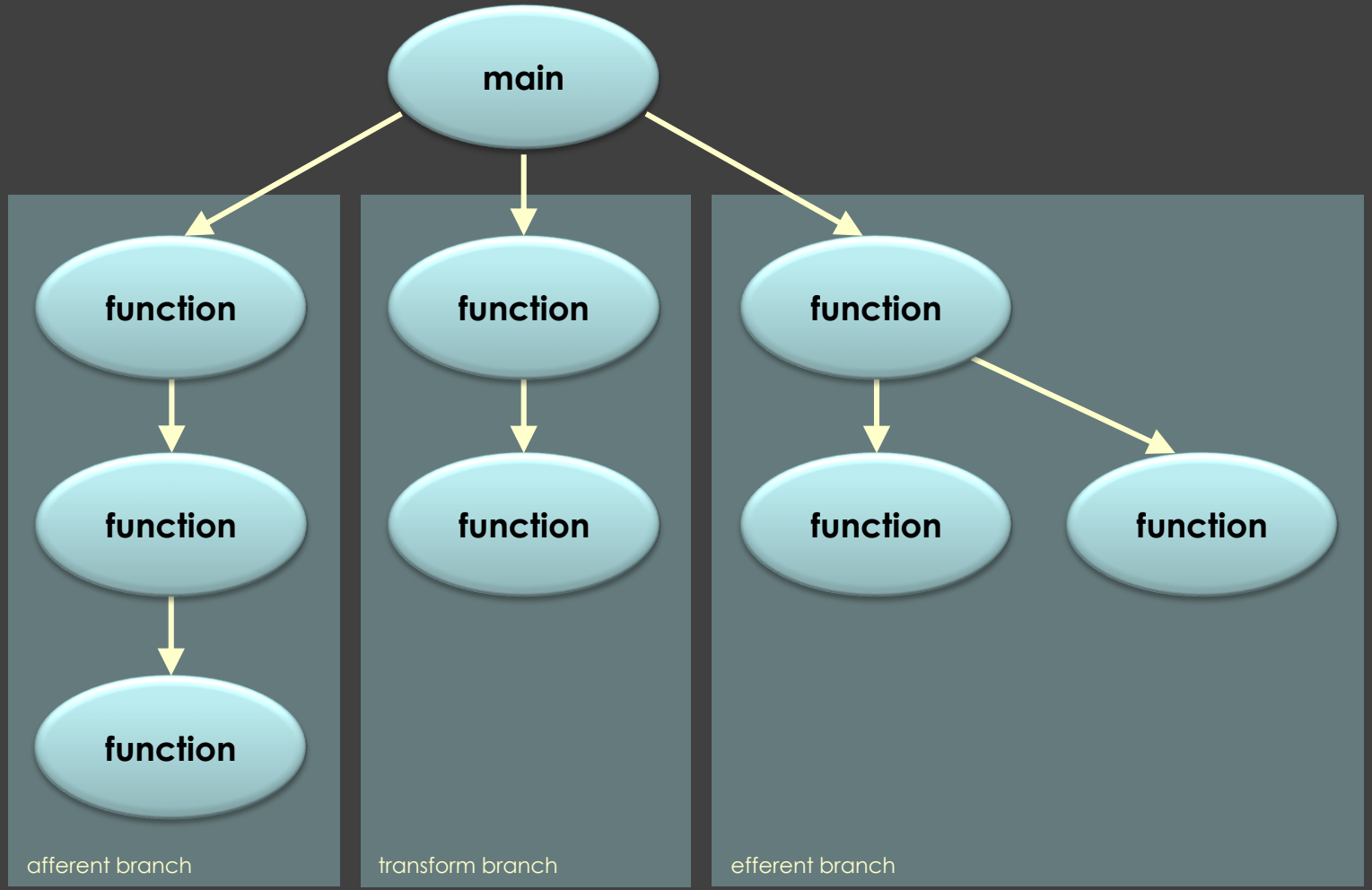

main
subroutine
subroutine
subroutine
subroutine
subroutine
subroutine
subroutine
subroutine
afferent branch
transform branch
efferent branch

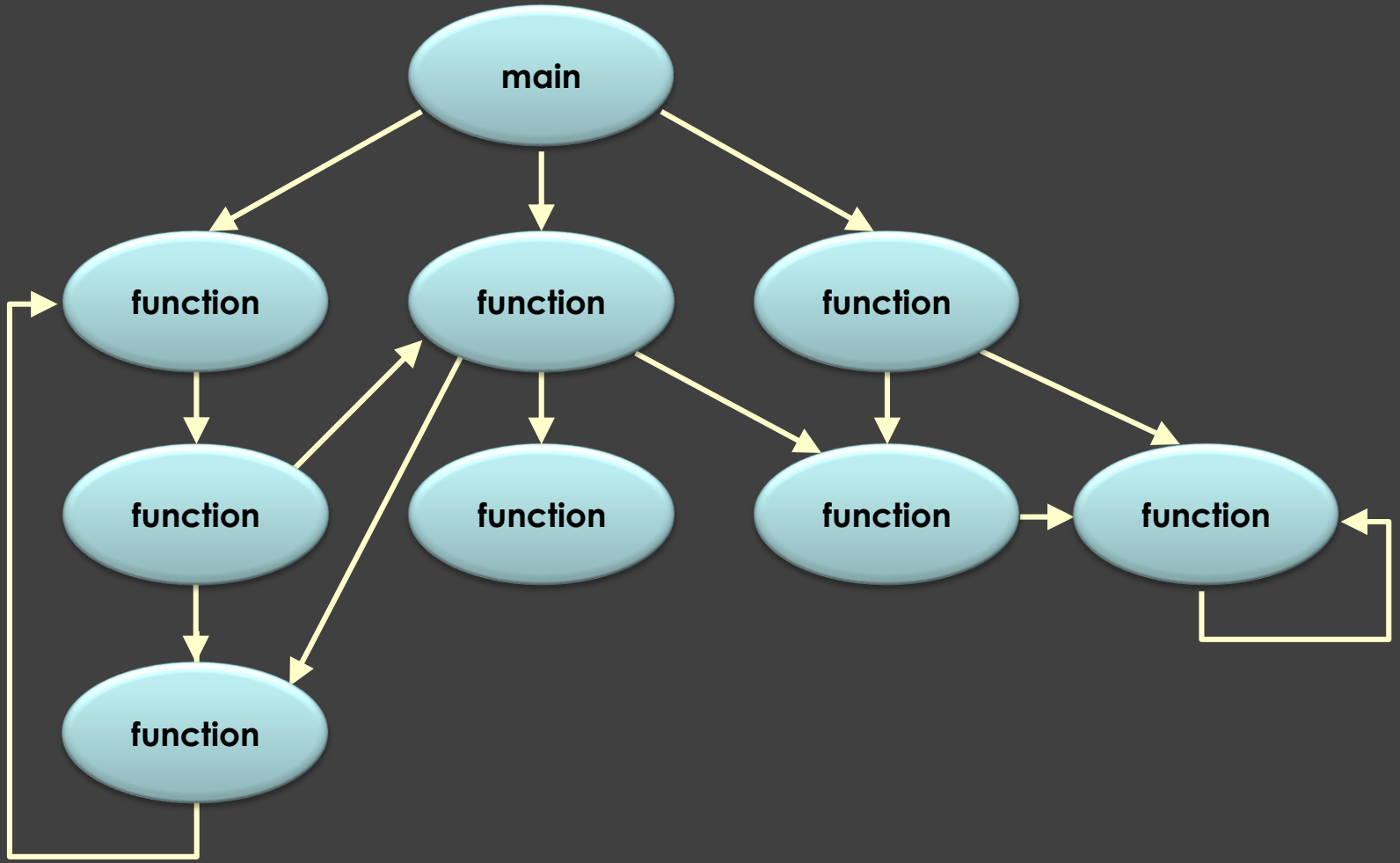
There is typically a single thread of control and each component in the hierarchy gets this control (optionally along with some data) from its parent and passes it along to its children.

Len Bass, Paul Clements & Rick Kazman
Software Architecture in Practice


main
subroutine
subroutine
subroutine
subroutine
subroutine
subroutine
subroutine
subroutine
afferent branch
transform branch
efferent branch


main
procedure
procedure
procedure
procedure
procedure
procedure
procedure
procedure
afferent branch
transform branch
efferent branch


main
function
function
function
function
function
function
function
function
afferent branch
transform branch
efferent branch

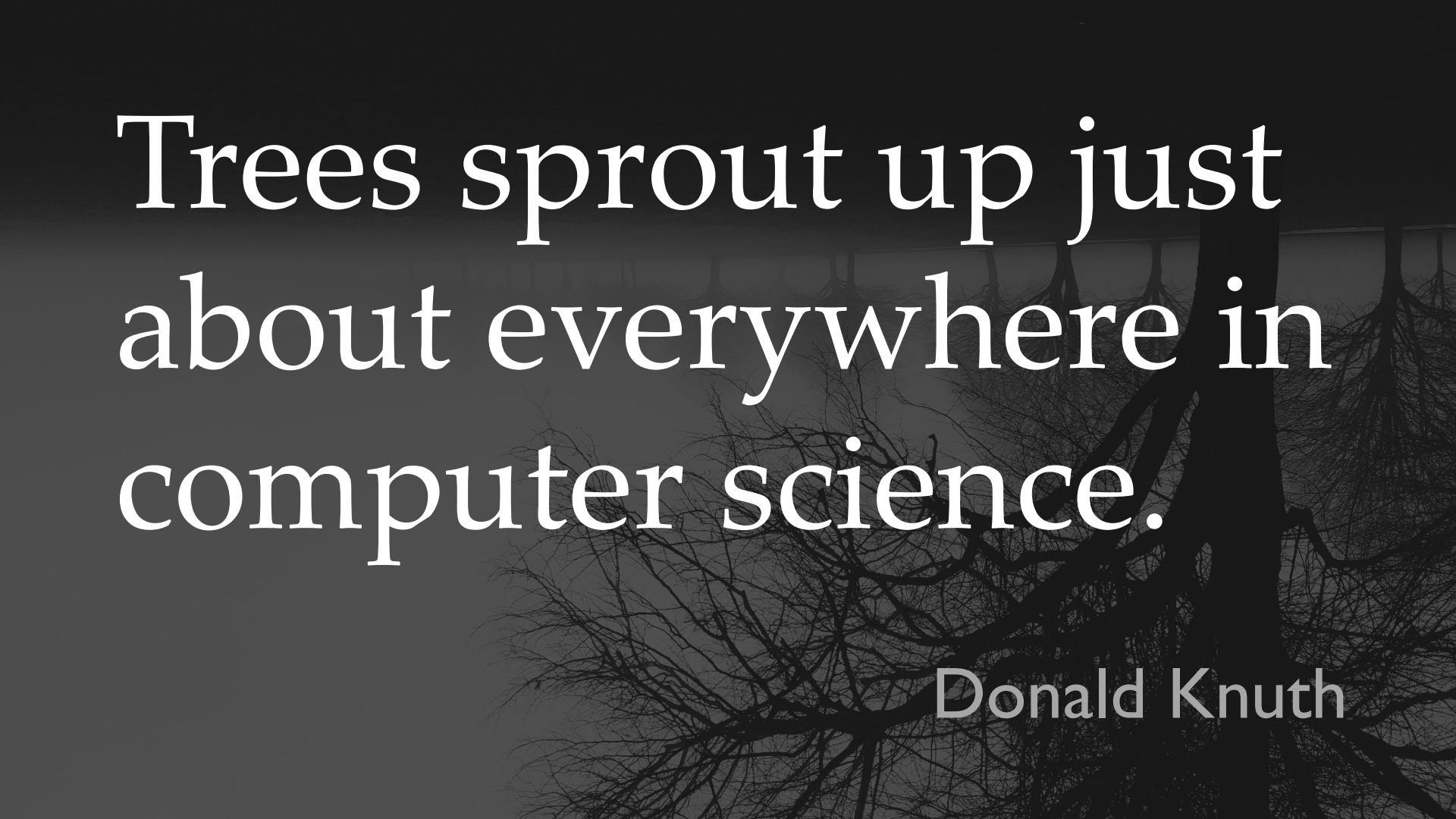

main
function
function
function
function
function
function
function
function

Everything should be built top-down, except the first time.

Alan Perlis

You cannot teach beginners top-down programming, because they don't know which end is up.

C A R Hoare


Trees sprout up just about everywhere in computer science.
Donald Knuth

'Michael Jackson's best work ever.' Tom DeMarco

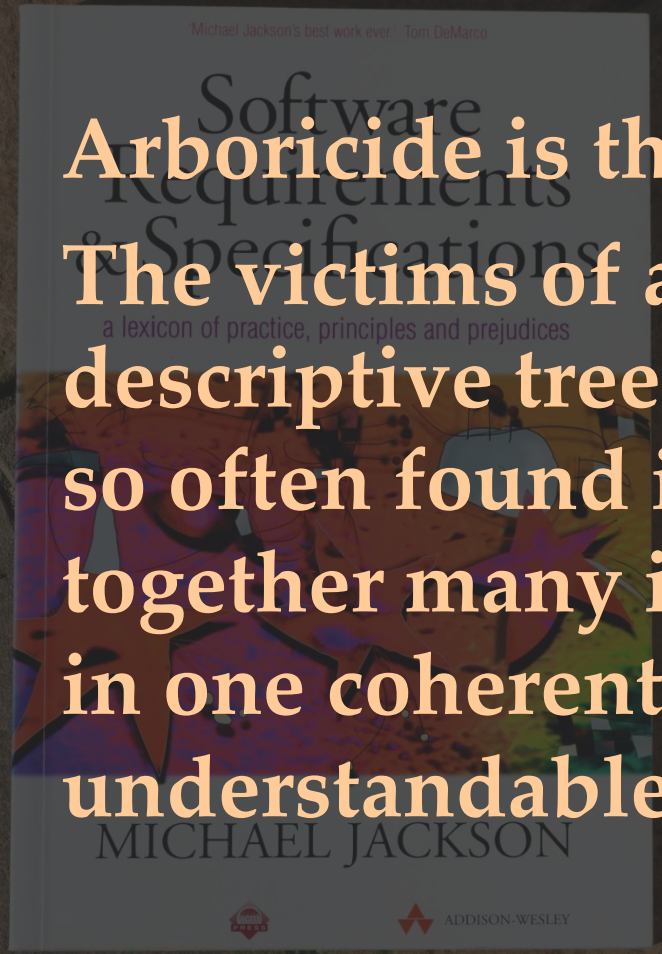
Software Requirements & Specifications

a lexicon of practice, principles and prejudices

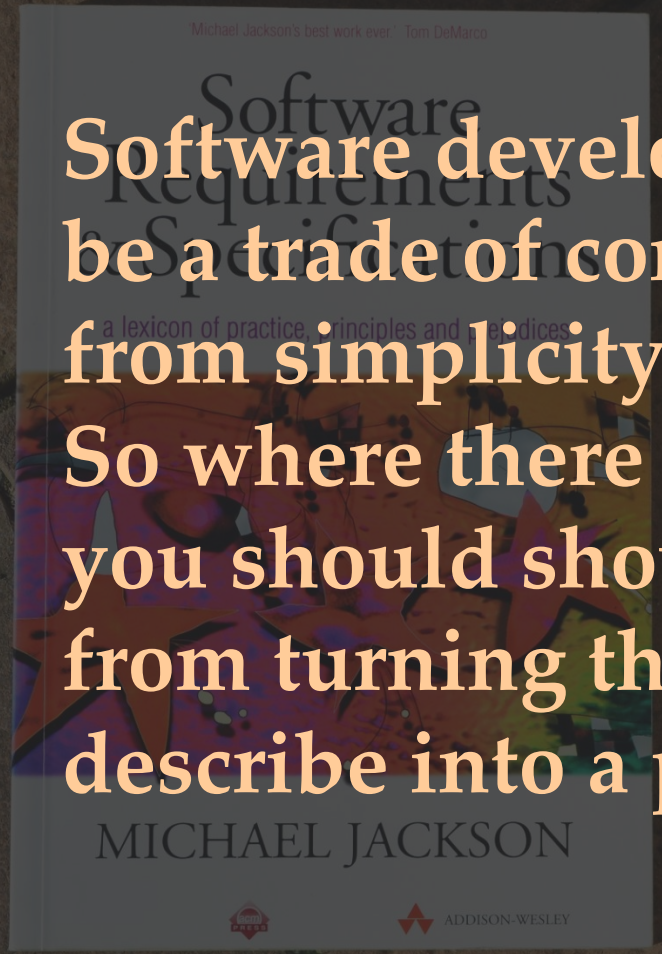
MICHAEL JACKSON

ACM PRESS

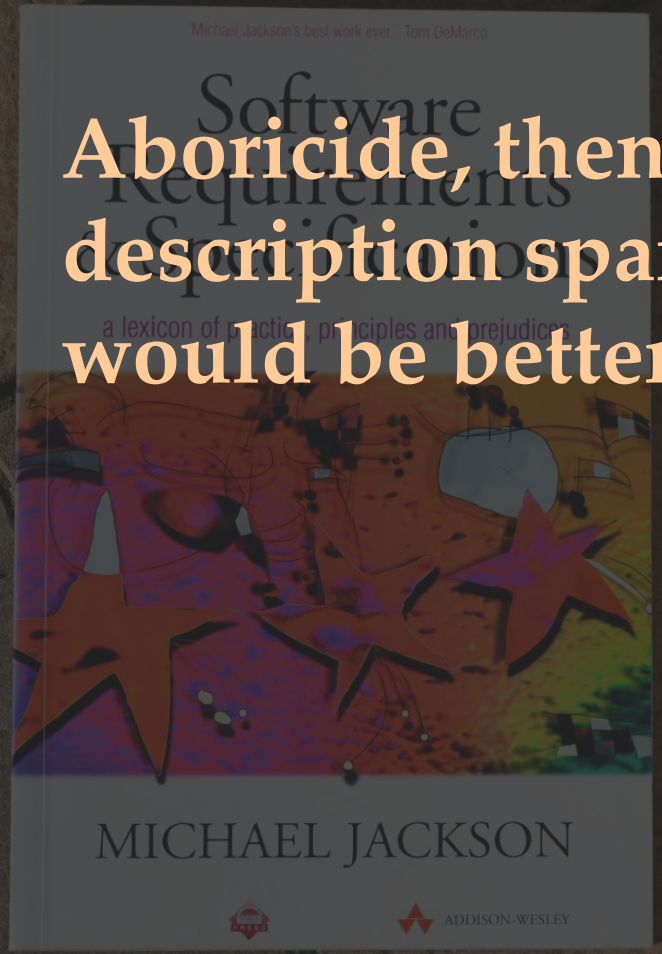
ADDISON-WESLEY



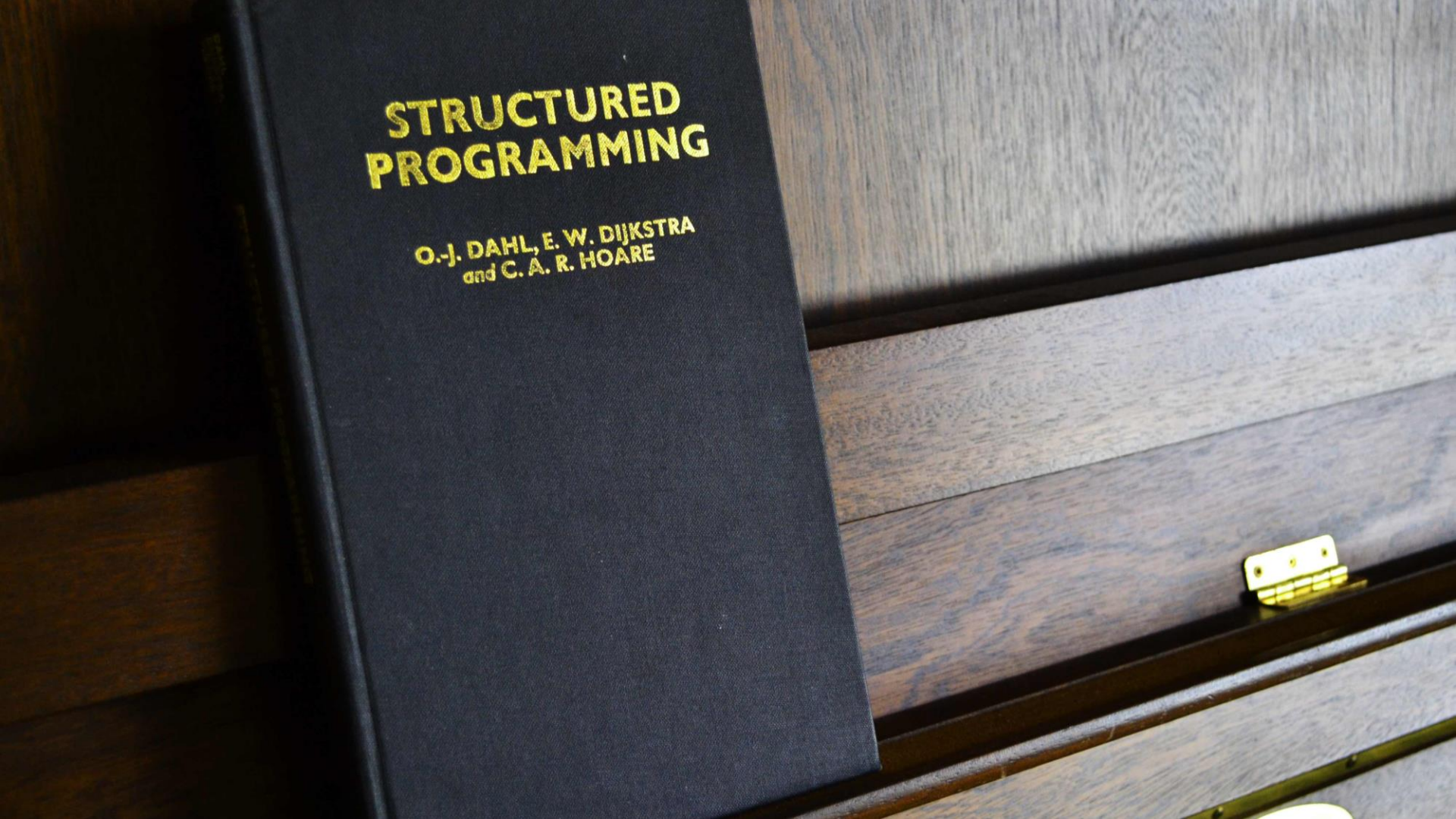
Arboricide is the murder of trees. The victims of arboricide are the descriptive tree structures that are so often found in software, holding together many individual elements in one coherent and immediately understandable harmony.

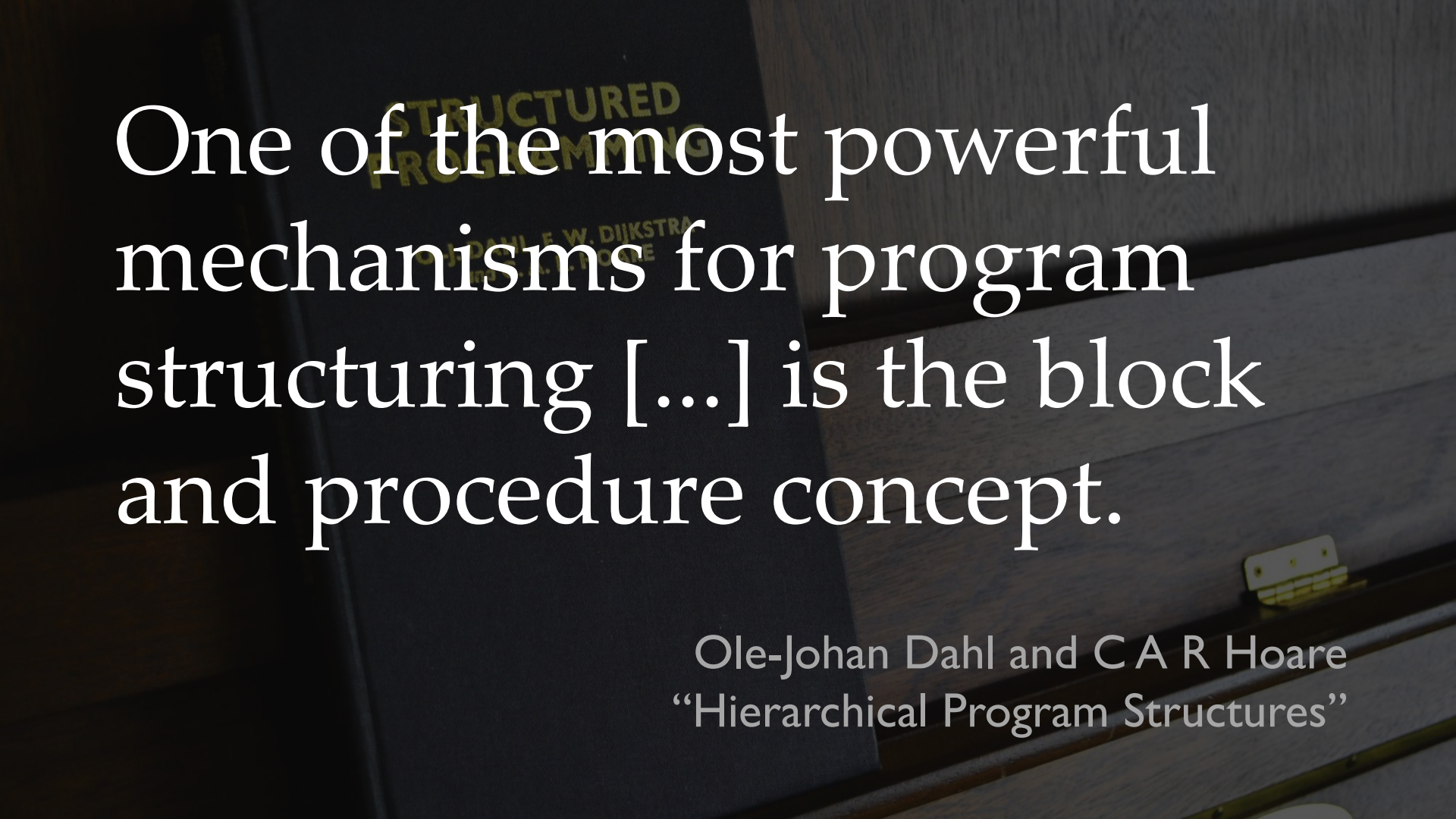


Software development should not be a trade of constructing difficulty from simplicity. Quite the contrary. So where there are trees to be shown you should show them, and refrain from turning the relationships they describe into a puzzle.



Aboricide, then, is using a smaller description span when a larger one would be better.


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



One of the most powerful mechanisms for program structuring [...] is the block and procedure concept.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"

```
begin
    ref(Book) array books(1:capacity);
    integer count;
    procedure Push(top); ...
    procedure Pop; ...
    boolean procedure IsEmpty; ...
    boolean procedure IsFull; ...
    integer procedure Depth; ...
    ref(Book) procedure Top; ...
    count := 0
end;
```

A procedure which is capable of giving rise to block instances which survive its call will be known as a class; and the instances will be known as objects of that class.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"

```
class Stack(capacity);
    integer capacity;
begin
    ref(Book) array books(1:capacity);
    integer count;
    procedure Push(top); ...
    procedure Pop; ...
    boolean procedure IsEmpty; ...
    boolean procedure IsFull; ...
    integer procedure Depth; ...
    ref(Book) procedure Top; ...
    count := 0
end;
```

λ-calculus was the first object-oriented language.

William Cook

"On Understanding Data Abstraction, Revisited"

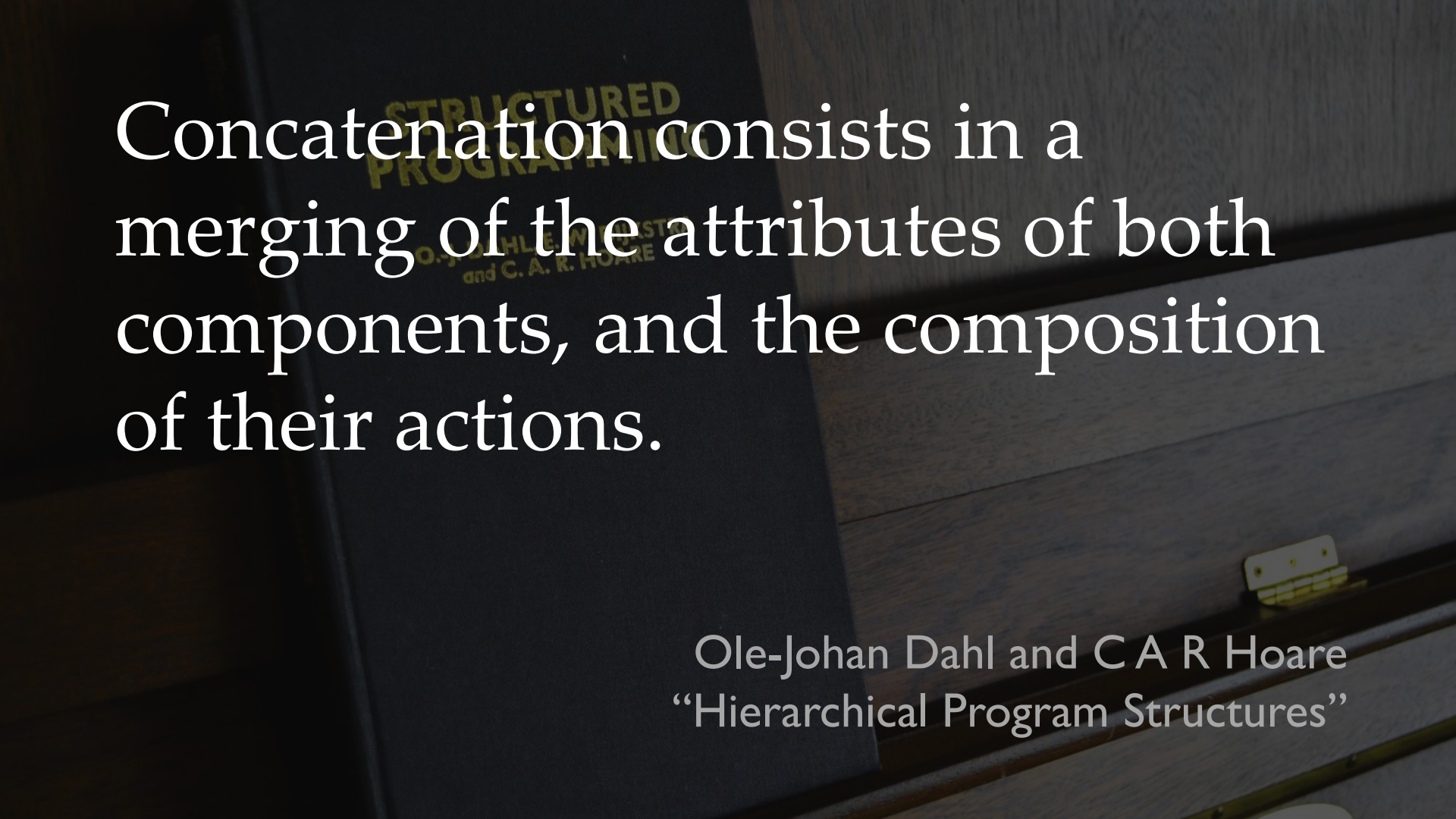
https://dl.acm.org/citation.cfm?id=1640133

```
const newStack = () => {
    const items = []
    return {
        depth: () => items.length,
        top: () => items[0],
        pop: () => { items.shift() },
        push: newTop => { items.unshift(newTop) },
    }
}
```

```
const newStack = () => {
    const items = []
    return {
        depth: () => items.length,
        top: () => items[items.length - 1],
        pop: () => { items.pop() },
        push: newTop => { items.push(newTop) },
    }
}
```

Concatenation is an operation defined between two classes *A* and *B*, or a class *A* and a block *C*, and results in the formation of a new class or block.

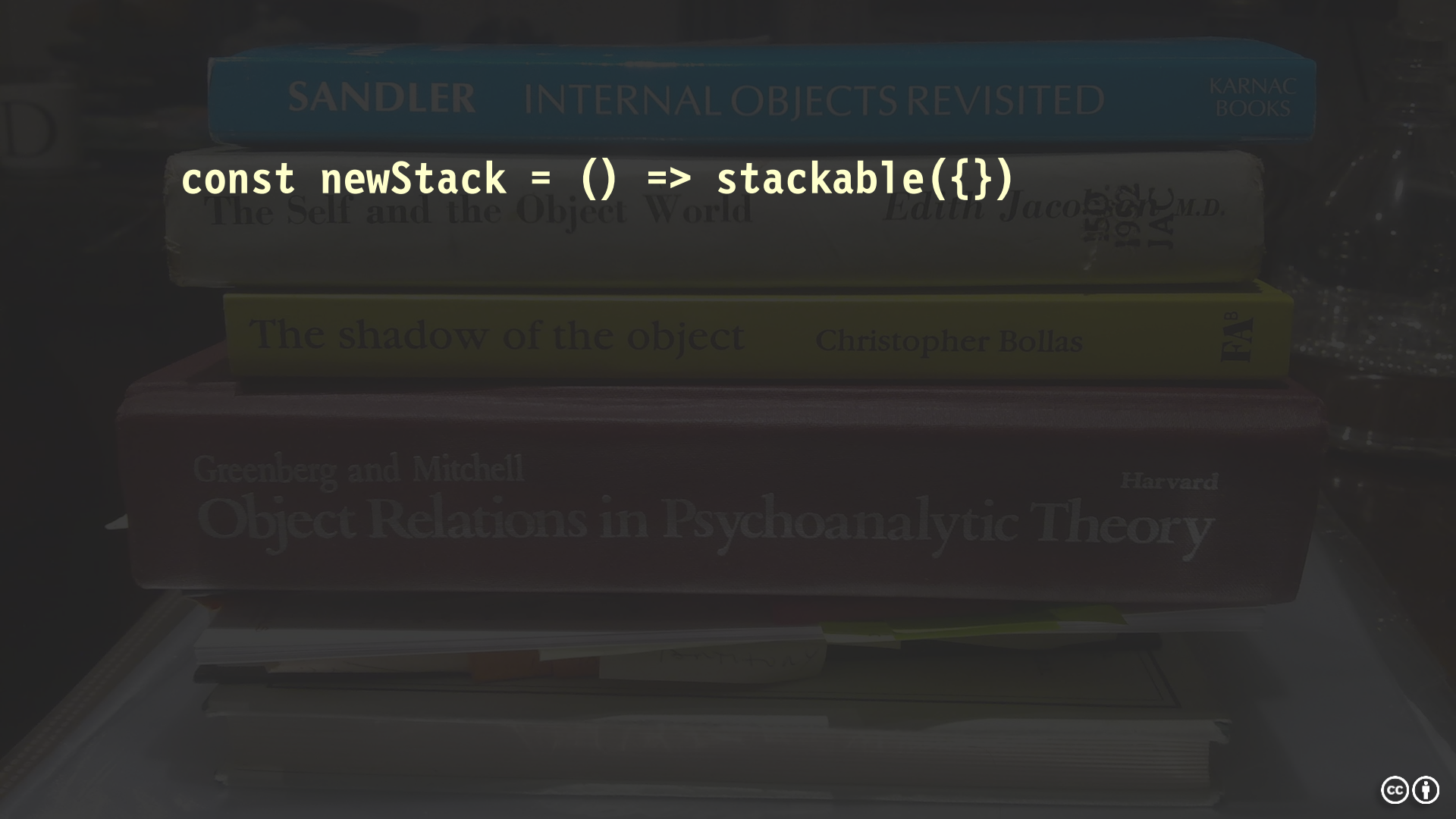
Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"



Concatenation consists in a merging of the attributes of both components, and the composition of their actions.

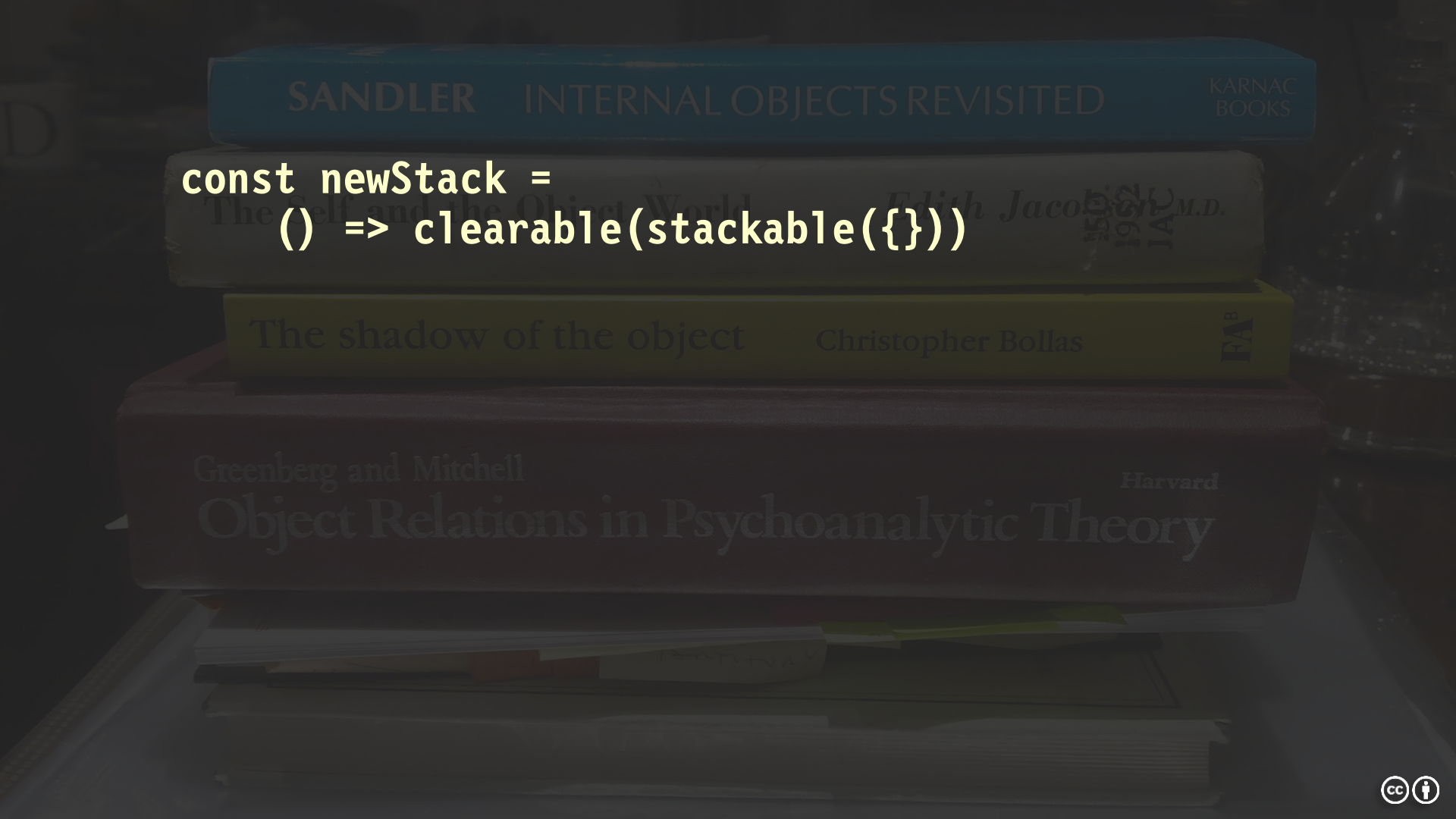
Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"

```
const stackable = base => {
    const items = []
    return Object.assign(base, {
        depth: () => items.length,
        top: () => items[items.length - 1],
        pop: () => { items.pop() },
        push: newTop => { items.push(newTop) },
    })
}
```




```
const newStack = () => stackable({})
```

```
const clearable = base => {
    return Object.assign(base, {
        clear: () => {
            while (base.depth())
                base.pop()
        },
    })
}
```




```
const newStack =
    () => clearable(stackable({}))
```

```
const newStack =
    () => compose(clearable, stackable)({})

const compose = (...funcs) =>
    arg => funcs.reduceRight(
        (composed, func) => func(composed), arg)
```

Concept Hierarchies

The construction principle involved is best called *abstraction;* we concentrate on features common to many phenomena, and we abstract *away* features too far removed from the conceptual level at which we are working.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra.

Barbara Liskov
"Data Abstraction and Hierarchy"

What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"

```
const nonDuplicateTop = base => {
    const push = base.push
    return Object.assign(base, {
        push: newTop => {
            if (base.top() !== newTop)
                push(newTop)
        },
    })
}
```

```
tests = {
    ...
    'A non-empty stack becomes deeper by retaining a pushed item as its top':
        () => {
            const stack = newStack()

            stack.push('C++/C')
            stack.push('2019')
            stack.push('2019')

            assert(stack.depth() === 3)
            assert(stack.top() === '2019')
        },
    ...
}
```

```
const newStack =
  () => compose(clearable, stackable)({})

tests = {
    ...
    'A non-empty stack becomes deeper by retaining a pushed item as its top':
        () => {
            const stack = newStack()

            stack.push('C++/C')
            stack.push('2019')
            stack.push('2019')

            assert(stack.depth() === 3)
            assert(stack.top() === '2019')
        },
    ...
}
```

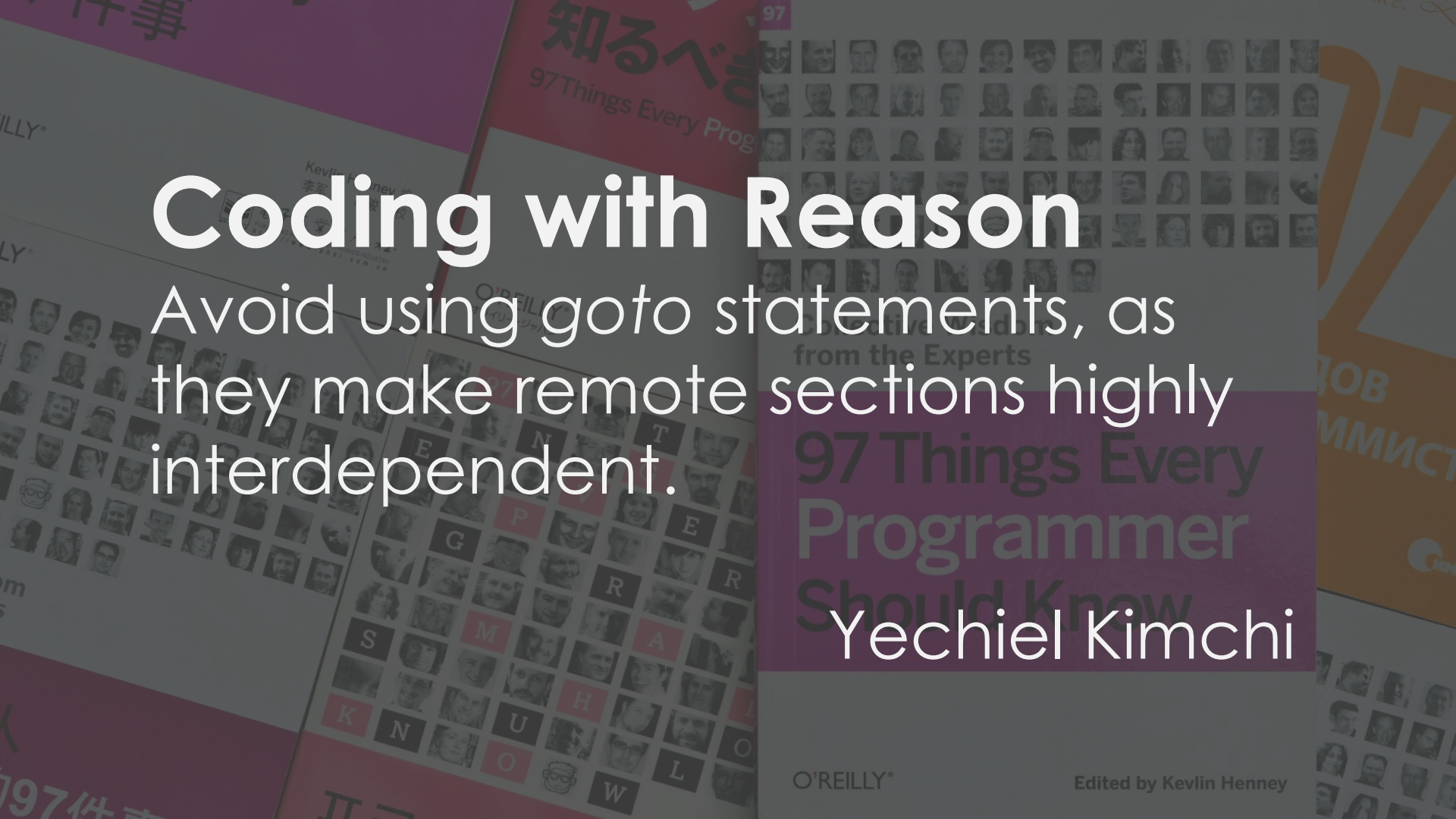
```
const newStack =
  () => compose(nonDuplicateTop, clearable, stackable)({})

tests = {
    ...
    'A non-empty stack becomes deeper by retaining a pushed item as its top':
        () => {
            const stack = newStack()
            stack.push('C++/C')
            stack.push('2019')
            stack.push('2019')
            assert(stack.depth() === 3)
            assert(stack.top() === '2019')
        },
    ...
}
```

What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"

reasonable

reason



Coding with Reason

Avoid using *goto* statements, as they make remote sections highly interdependent.

Yechiel Kimchi


Coding with Reason
Avoid using modifiable global variables, as they make all sections that use them dependent.
Yechiel Kimchi
97 Things Every Programmer Should Know
from the Experts
O'REILLY
Edited by Kevlin Henney

```
tests = {
    ...
    'A non-empty stack becomes deeper by retaining a pushed item as its top':
        () => {
            const stack = newStack()

            stack.push('C++/C')
            stack.push('2019')
            stack.push('2019')

            assert(stack.depth() === 3)
            assert(stack.top() === '2019')
        },
    ...
}
```

```
tests = {
    ...
    'A non-empty stack becomes deeper by retaining a pushed item as its top':
        () => {
            const stack = newStack()        // Arrange

            stack.push('C++/C')             // Act
            stack.push('2019')
            stack.push('2019')

            assert(stack.depth() === 3)     // Assert
            assert(stack.top() === '2019')
        },
    ...
}
```

```
tests = {
    ...
    'A non-empty stack becomes deeper by retaining a pushed item as its top':
        () => {
            const stack = newStack()        // Given

            stack.push('C++/C')             // When
            stack.push('2019')
            stack.push('2019')

            assert(stack.depth() === 3)     // Then
            assert(stack.top() === '2019')
        },
    ...
}
```

jasongorman
@jasongorman

Fun Fact: "Given... when... then..." is what we call a Hoare Triple en.wikipedia.org/wiki/Hoare_log…

7:42 PM - 3 Mar 2015

17 16

$$\{P\}\ Q\ \{R\}$$

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming, proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

A5 $(r - y) + y \times (1 + q)$

$= (r - y) + (y \times 1 + y \times q)$

A9 $= (r - y) + (y + y \times q)$

A3 $= ((r - y) + y) + y \times q$

A6 $= r + y \times q$ provided $y \leqslant r$

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to *nonnegative* numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and later have been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming' proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

$$P\{Q\}R$$

of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

A5 $(r - y) + y \times (1 + q)$

$= (r - y) + (y \times 1 + y \times q)$

A9 $= (r - y) + (y + y \times q)$

A3 $= (r - y) + y) + y \times q$

A6 $= r + y \times q$ provided $y \leqslant r$

The axioms A1 to A9 are of course true of the traditional infinite set of integers of mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to *nonnegative* numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

An Axiomatic Basis for Computer Programming

If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion.

KEY WORDS AND PHRASES: axiomatic method, theory of programming' proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

A5 $= (r - y) + (y \times 1 + y \times q)$

$= (r - y) + y \times (1 + q)$

A6 $= r + y \times q$ provided $y \leqslant r$

The axioms A1 to A9 are of course true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to *nonnegative* numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

sequence

selection

iteration

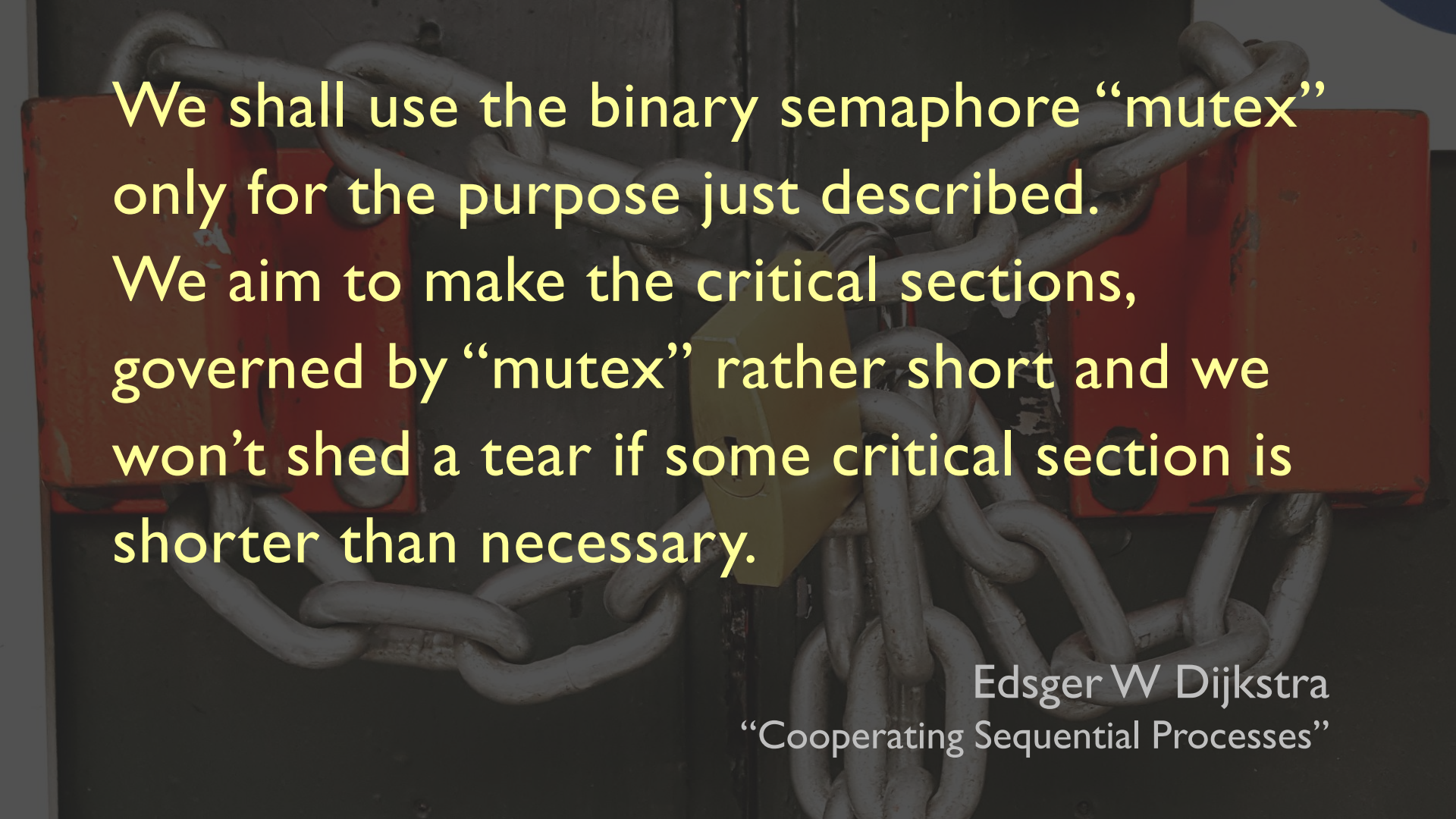
λ Calrissian
@mattpodwysocki

OH: "take me down to concurrency city where green pretty is grass the girls the and are"

9:30 PM - 24 Oct 2013

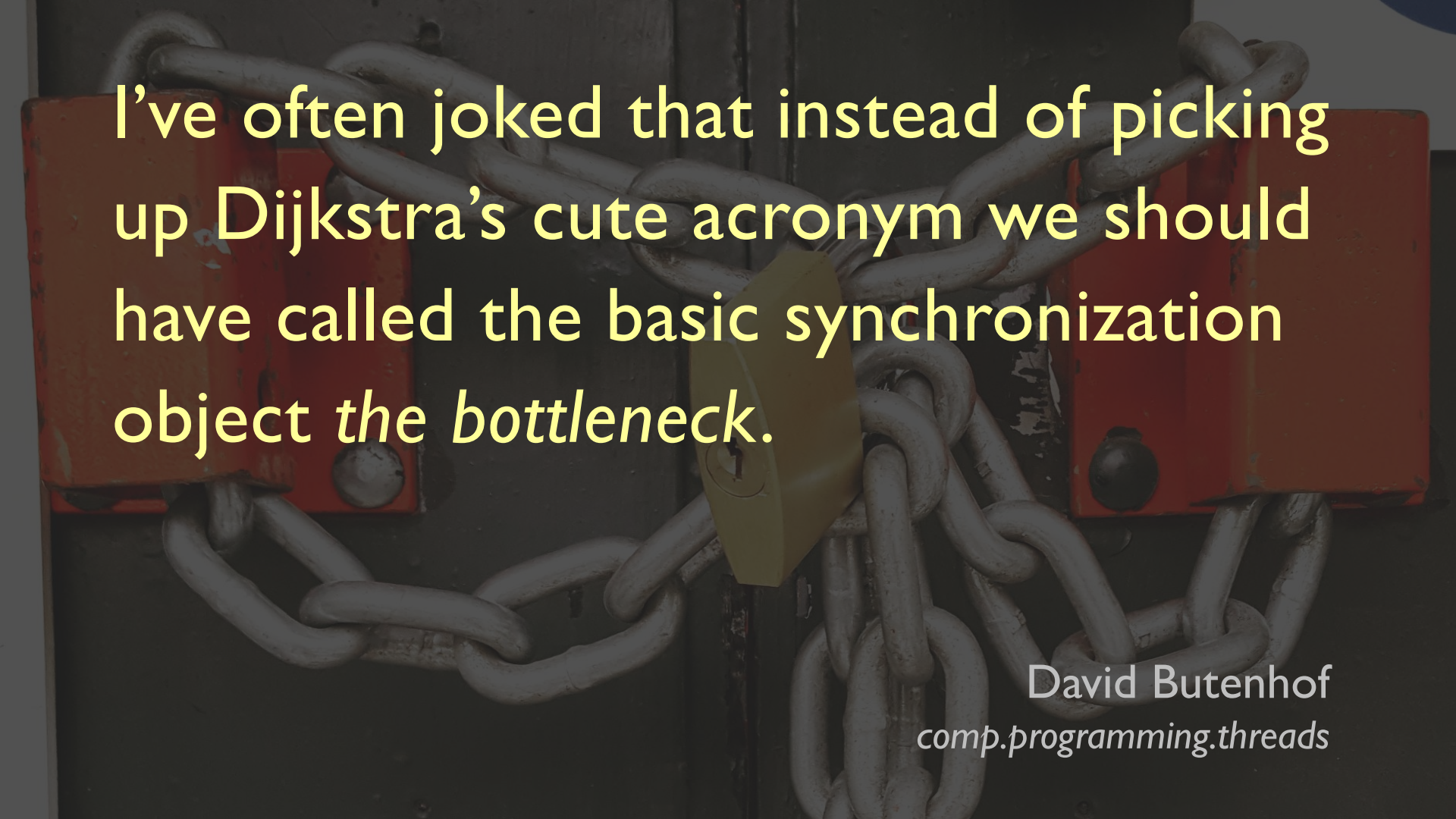
1,417 843

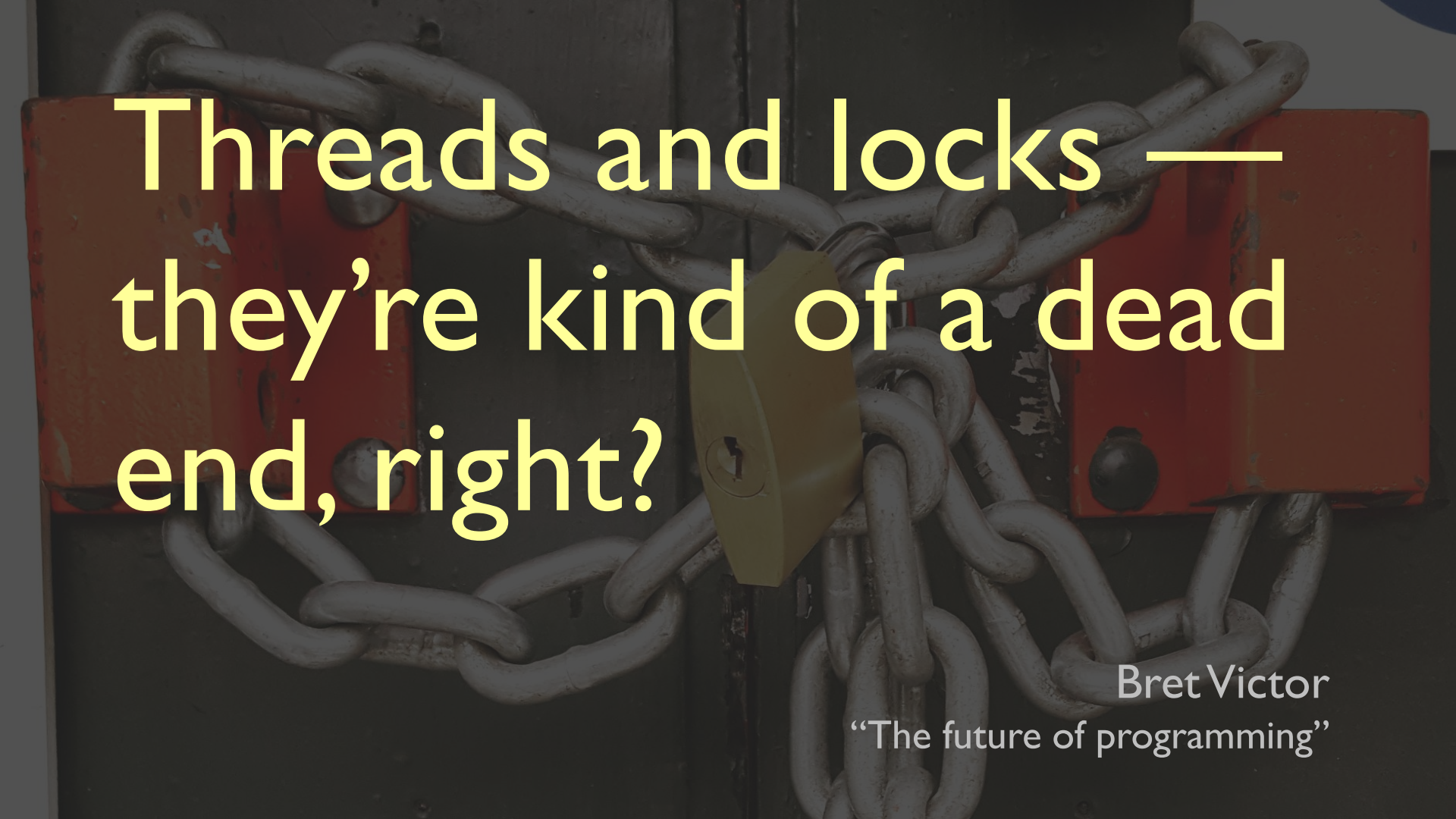
https://twitter.com/mattpodwysocki/status/393474697699921921

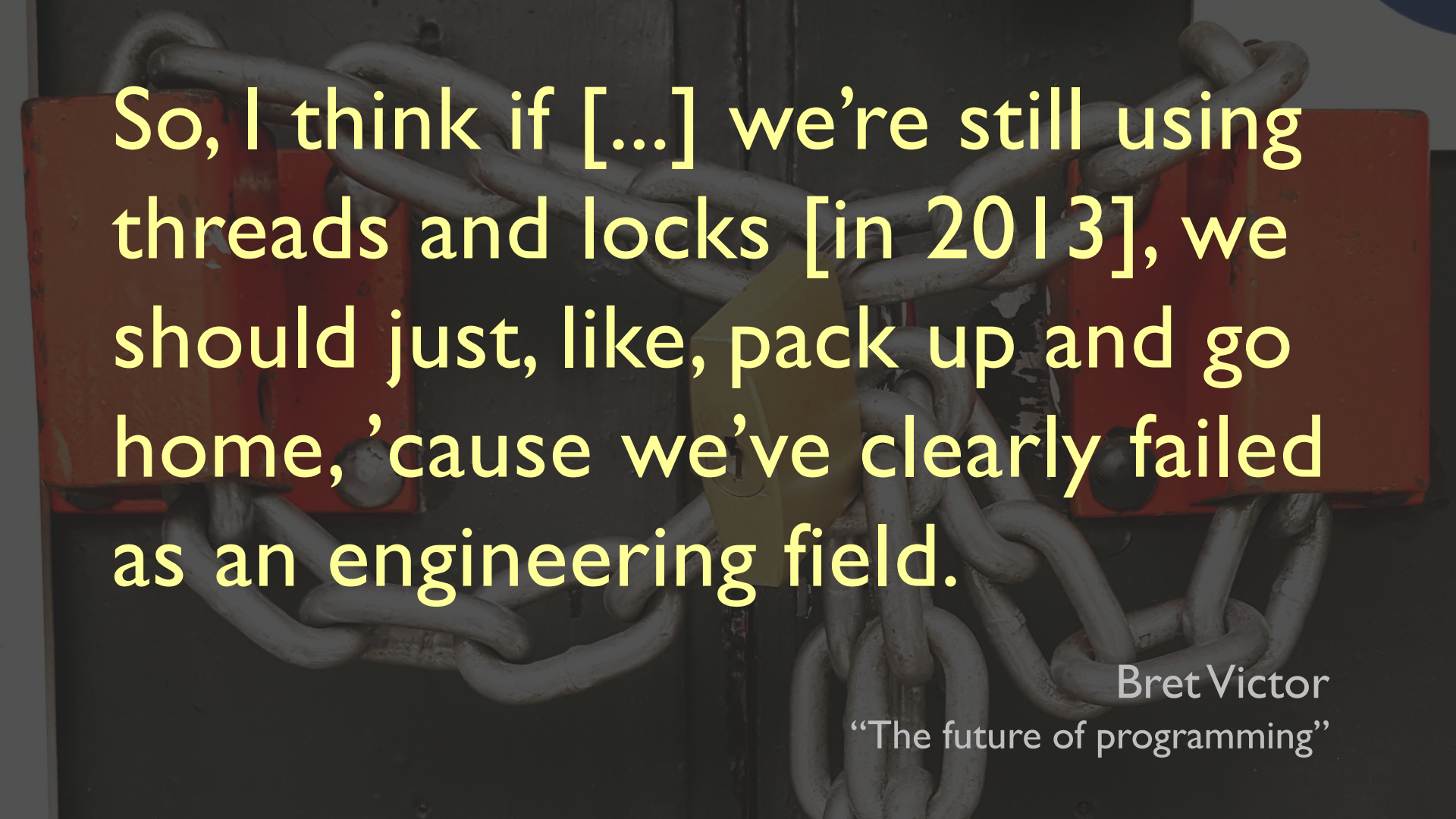


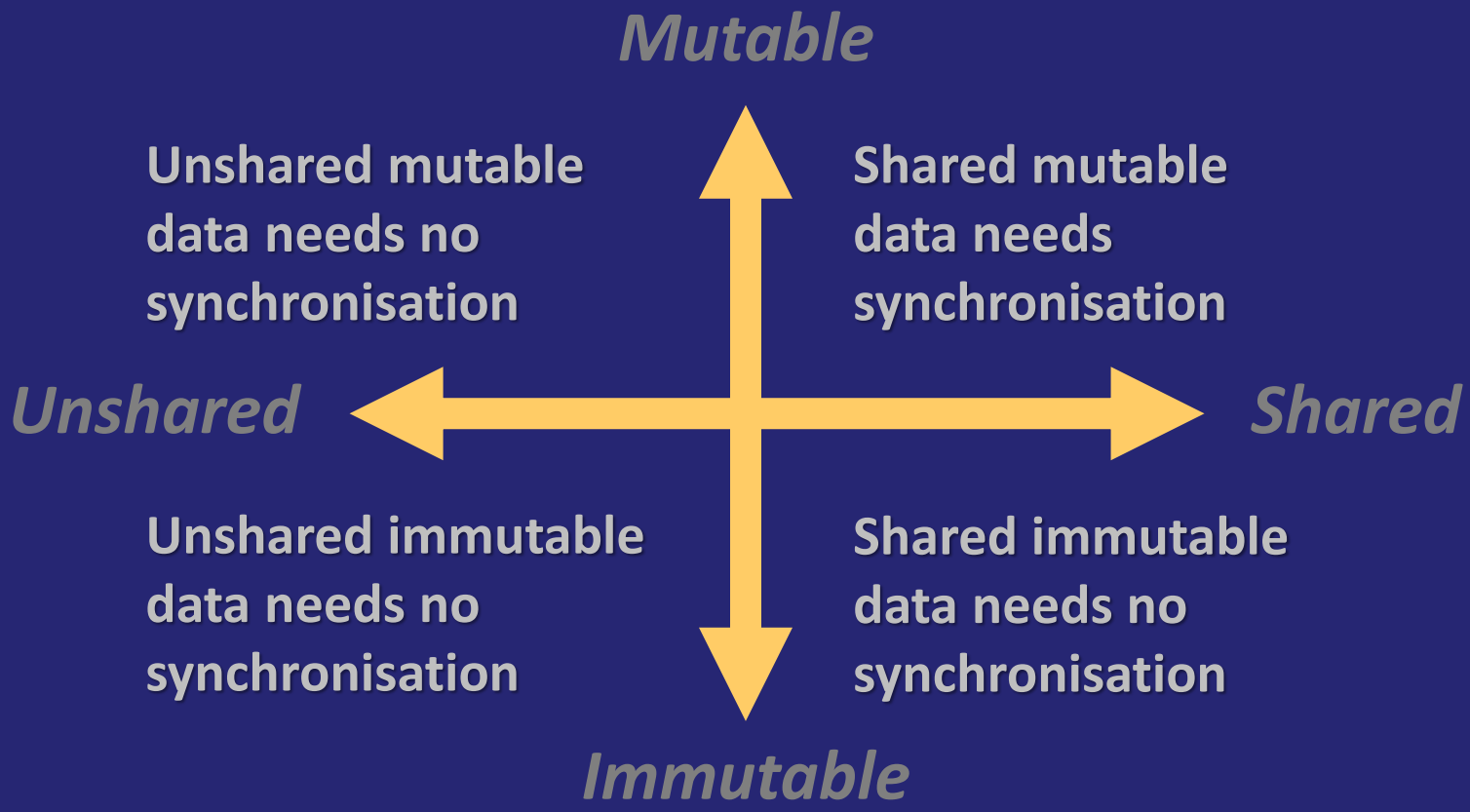
We shall use the binary semaphore “mutex” only for the purpose just described.
We aim to make the critical sections, governed by “mutex” rather short and we won’t shed a tear if some critical section is shorter than necessary.

Edsger W Dijkstra
“Cooperating Sequential Processes”

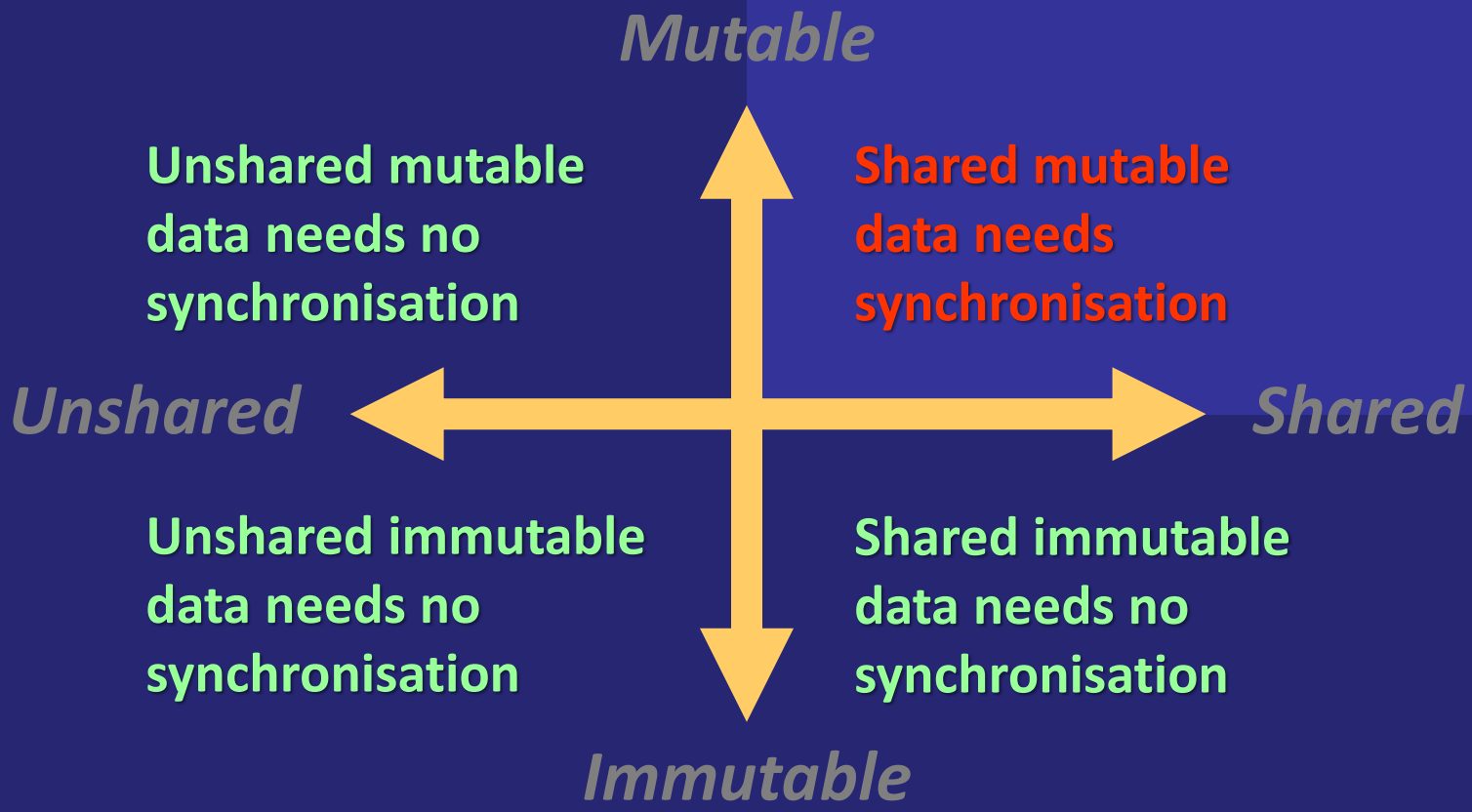

I've often joked that instead of picking up Dijkstra's cute acronym we should have called the basic synchronization object *the bottleneck*.
David Butenhof
comp.programming.threads


Threads and locks —
they're kind of a dead
end, right?
Bret Victor
"The future of programming"

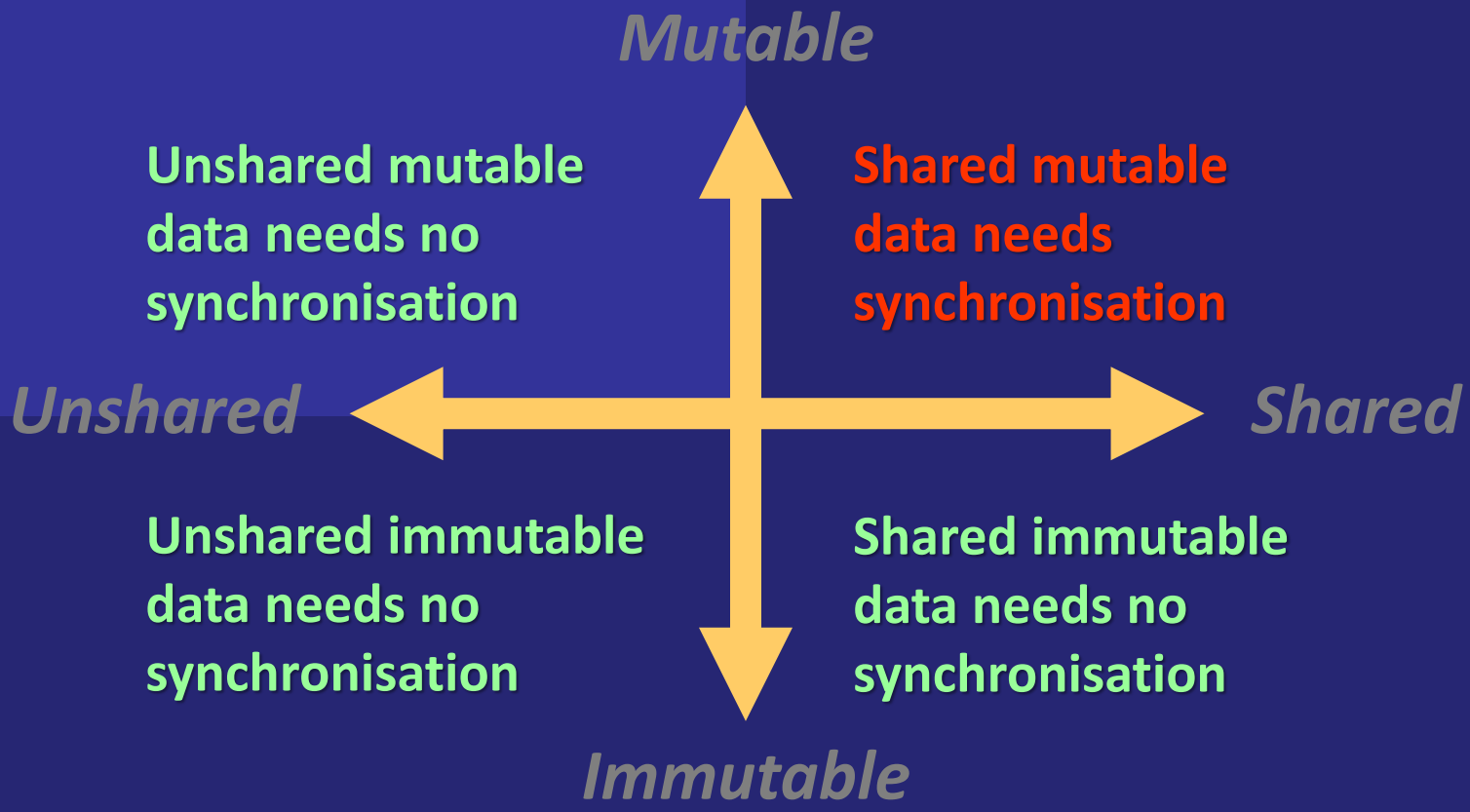

So, I think if [...] we're still using threads and locks [in 2013], we should just, like, pack up and go home, 'cause we've clearly failed as an engineering field.
Bret Victor
"The future of programming"


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

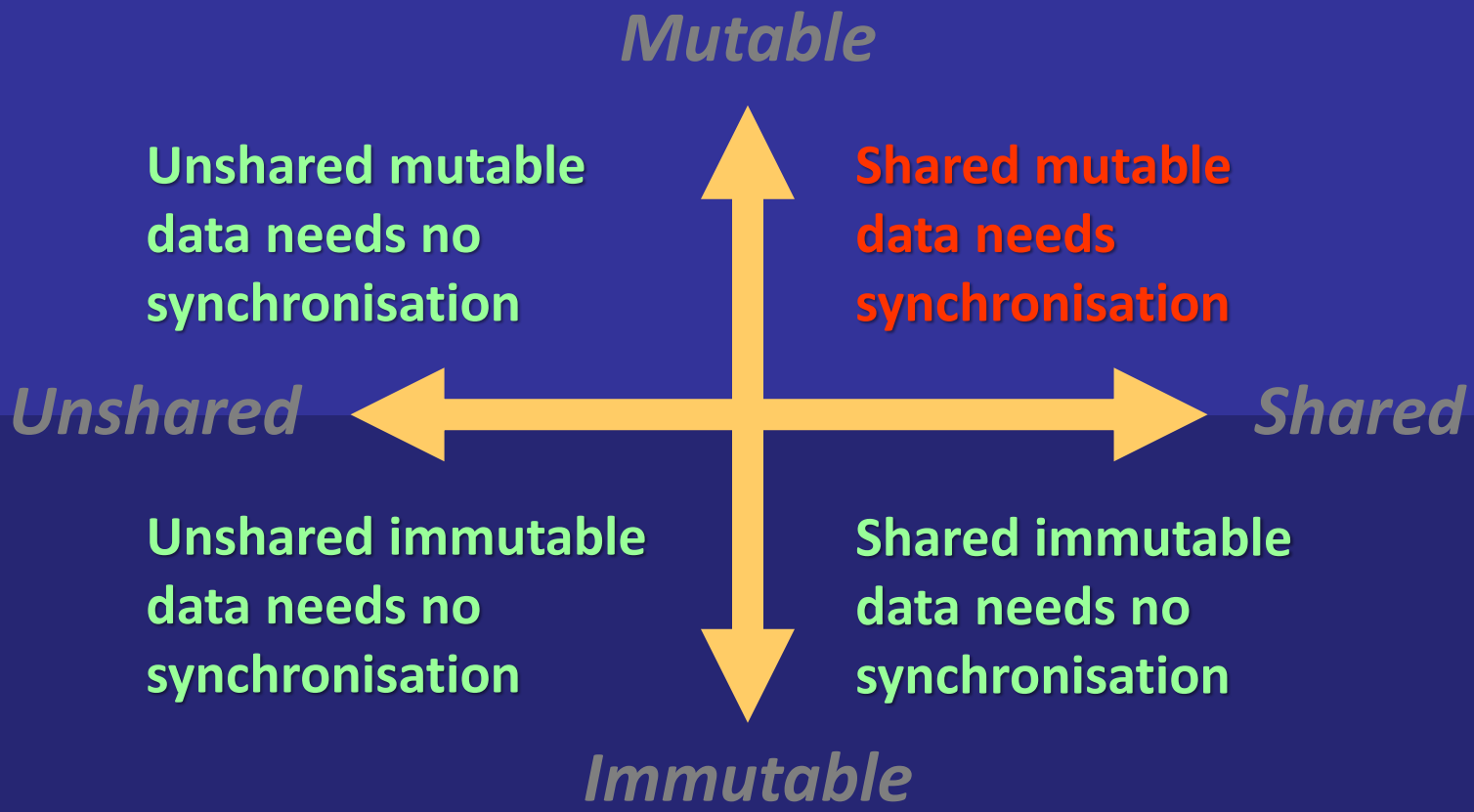


Procedural Comfort Zone

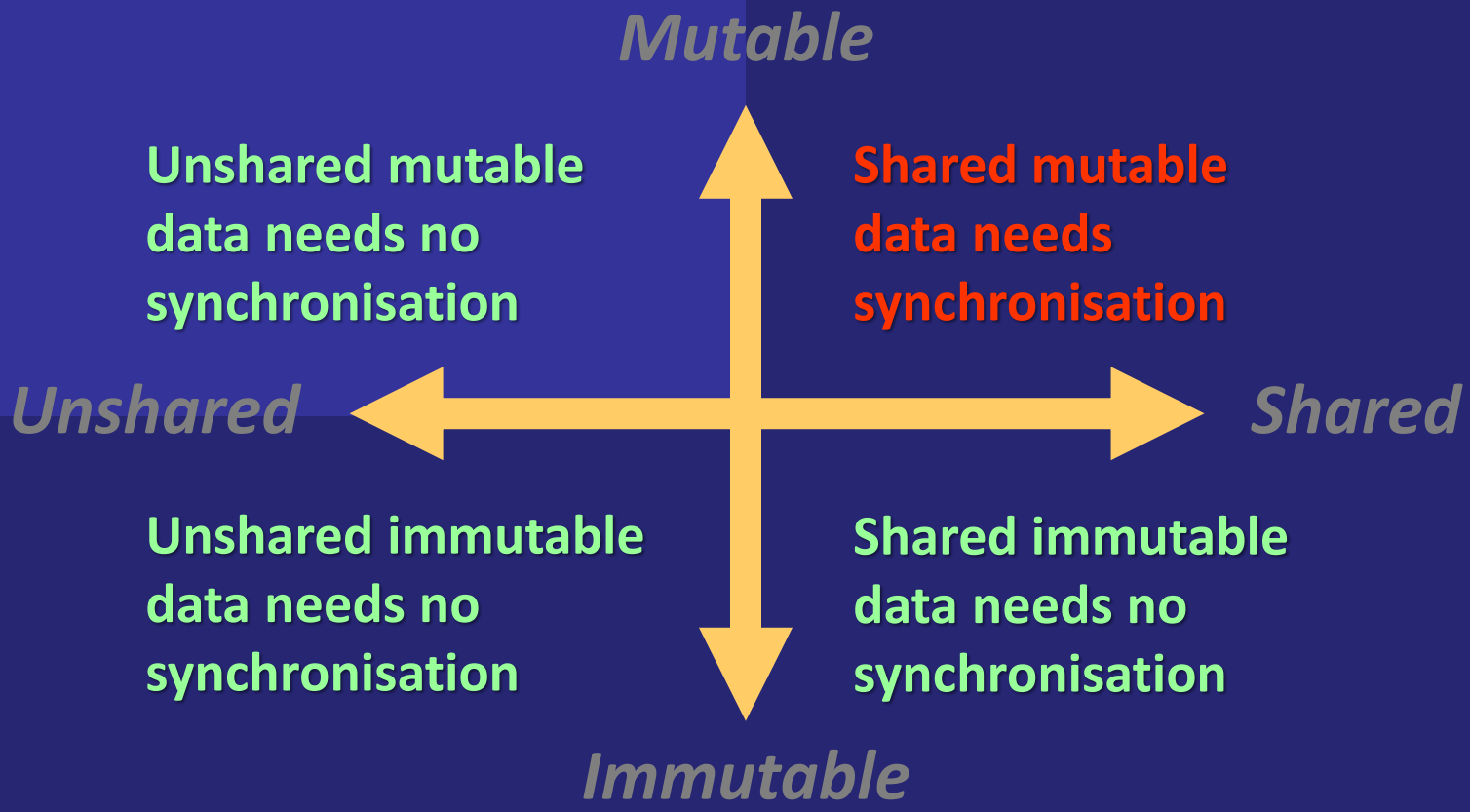


Procedural Comfort Zone

Procedural Discomfort Zone



Procedural Comfort Zone



We can build a complete programming model out of two separate pieces—the **computation model** and the **coordination model.**

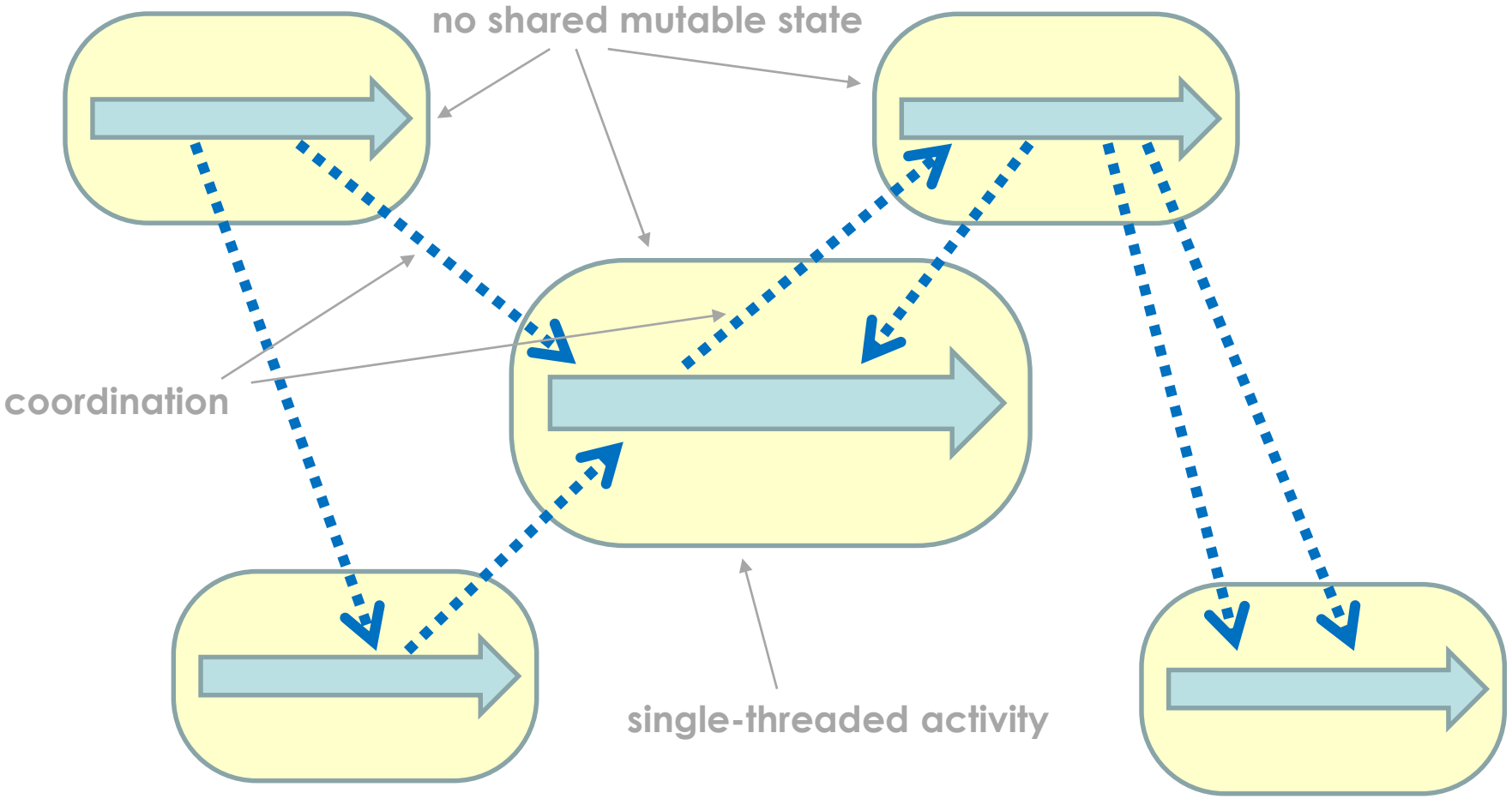
David Gelernter + Nicholas Carriero

“Coordination Languages and their Significance”

Algorithms + Data Structures = Programs

Niklaus Wirth

Coordination +
Computation =
Programs


no shared mutable state
coordination
single-threaded activity

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.

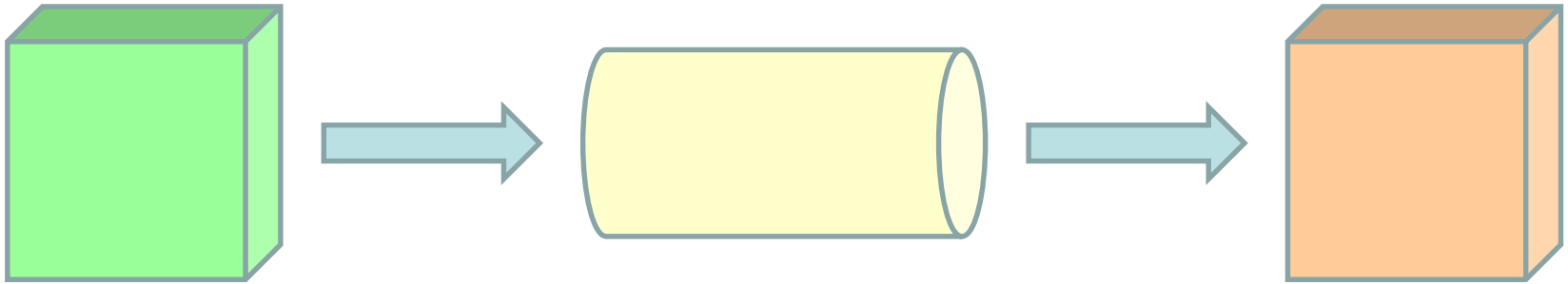
3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

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

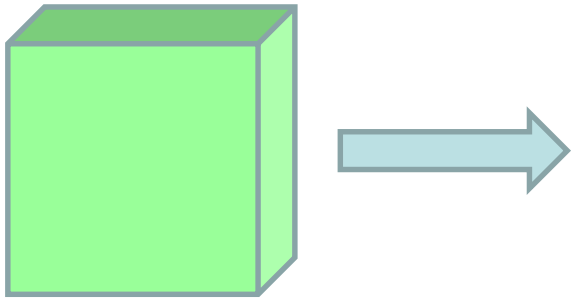
M. D. McIlroy
Oct. 11, 1964

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


CHECK
LOOP
LEVEL
STOP/UNDO
OUTPUT A
OUTPUT B
INPUT A
INPUT B
Loop Station
RC-1
E.LEVEL
F.BACK
D.TIME
HOLD
MODE
OUTPUT
INPUT
DIRECT OUT
Digital Delay
DD-3
E.LEVEL
RATE
DEPTH
FILTER
CUT FLAT
HIGH
LOW
OUTPUT A (MONO)
INPUT
OUTPUT B
Chorus Ensemble
CE-5
OUTPUT
INPUT
Distortion
DS-1X
FG6LL A20317
FG6ILLBS A20317
Fender

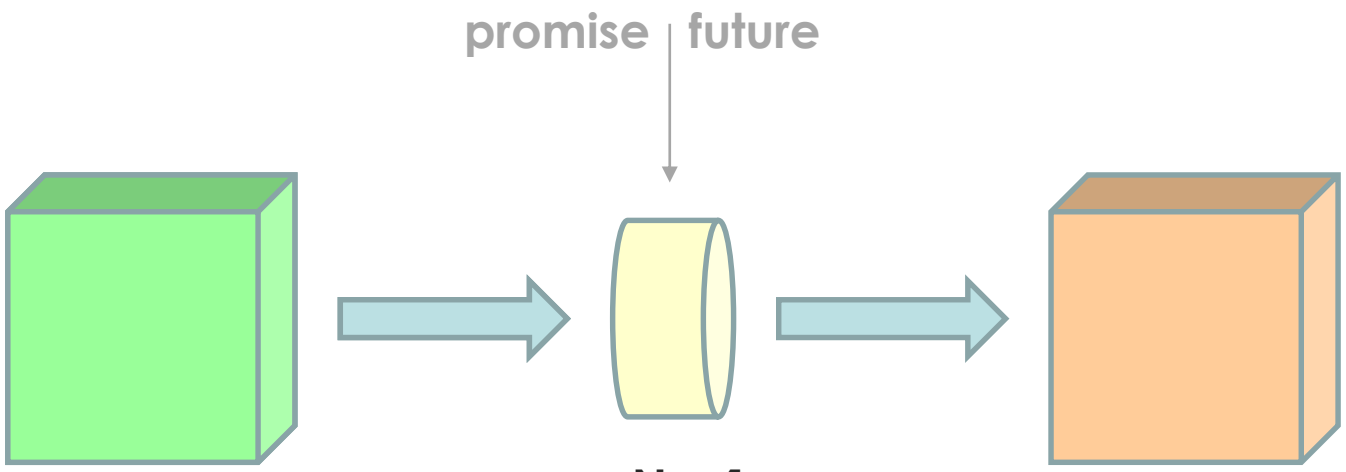


N
bounded
buffered
asynchronous

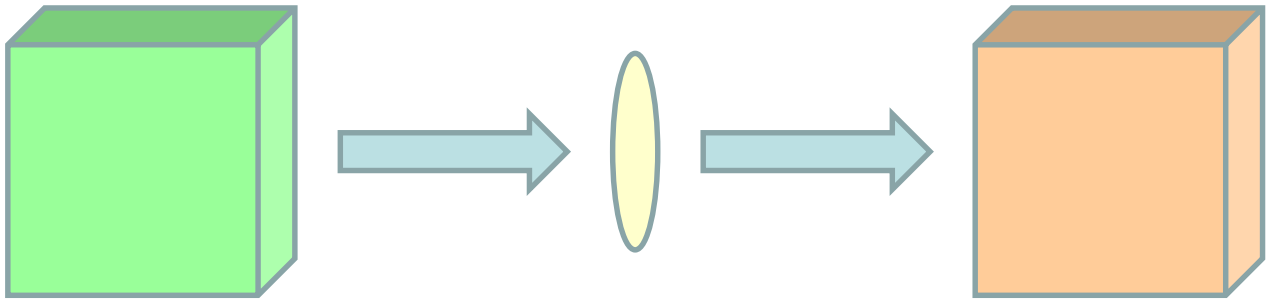


N = ∞

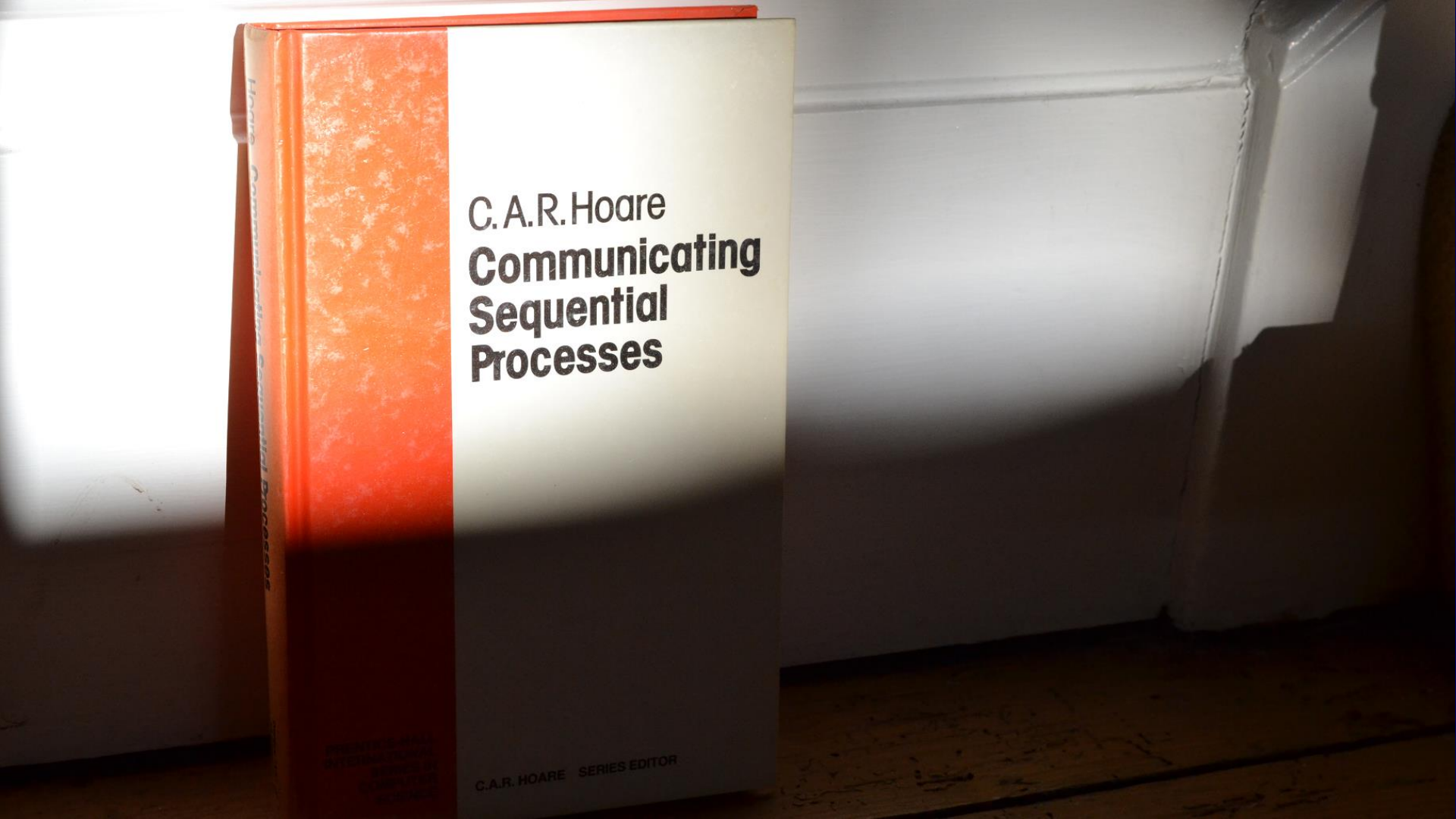
unbounded
buffered
asynchronous

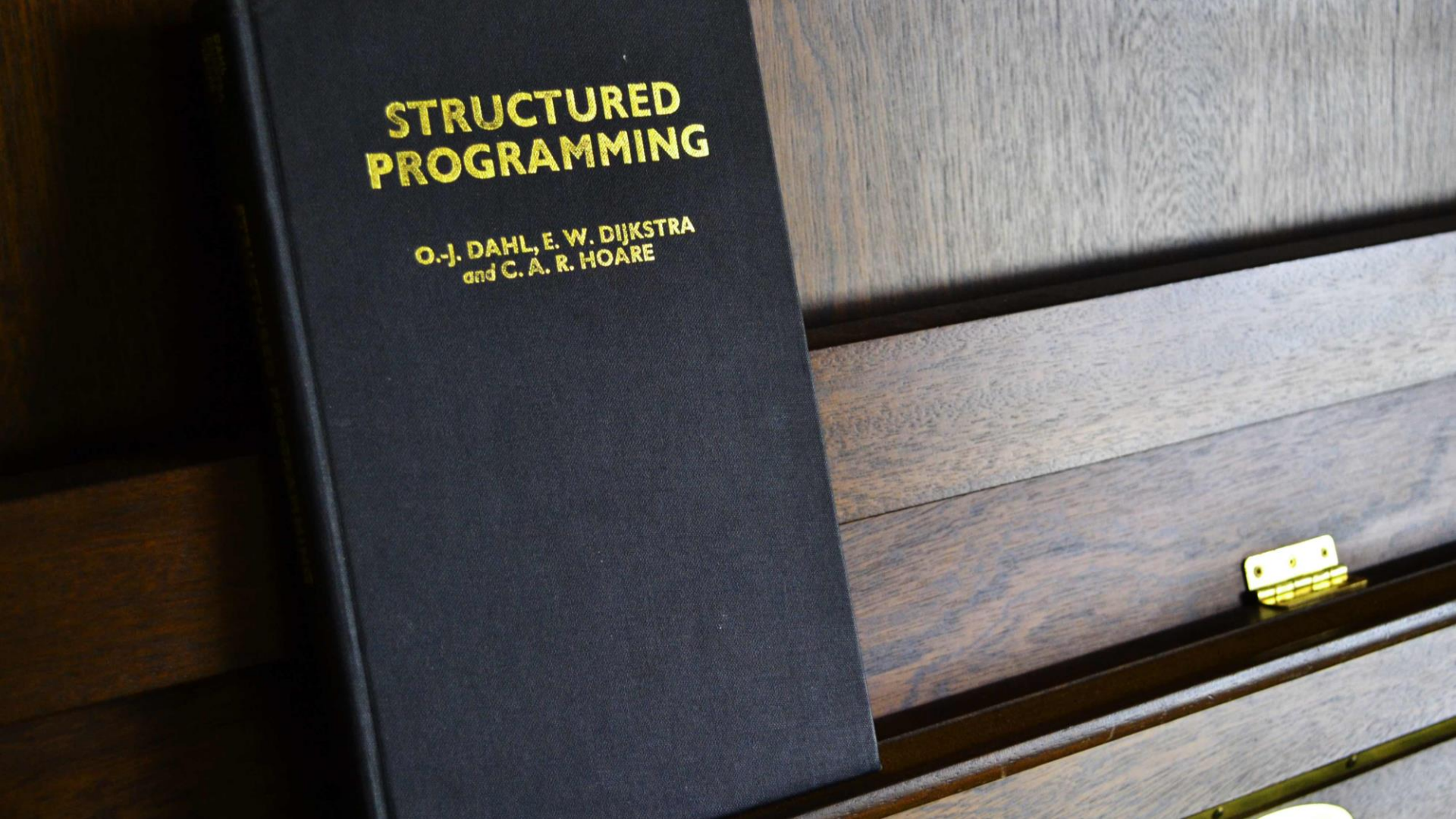


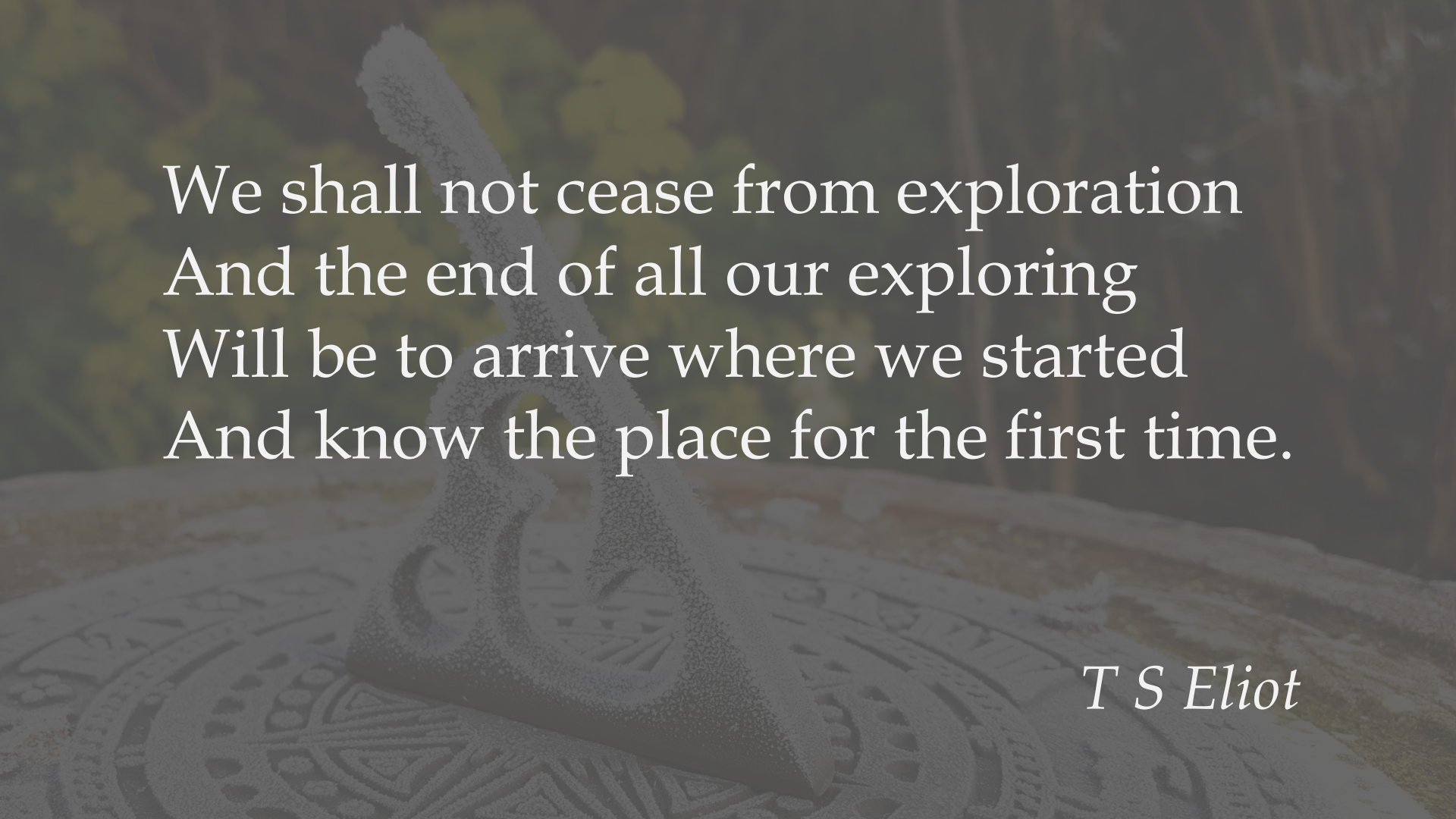
N = 1
bounded
buffered
asynchronous



N = 0
bounded
unbuffered
synchronous


C. A. R. Hoare
Communicating Sequential Processes
C.A.R. HOARE SERIES EDITOR


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


We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.
T S Eliot