Procedural Programming

It's Back? It Never Went Away

@KevlinHenney

Brycgstow

Bricstow


Bristow
DUNLEARY II 814

Bristol


procedure


procedural

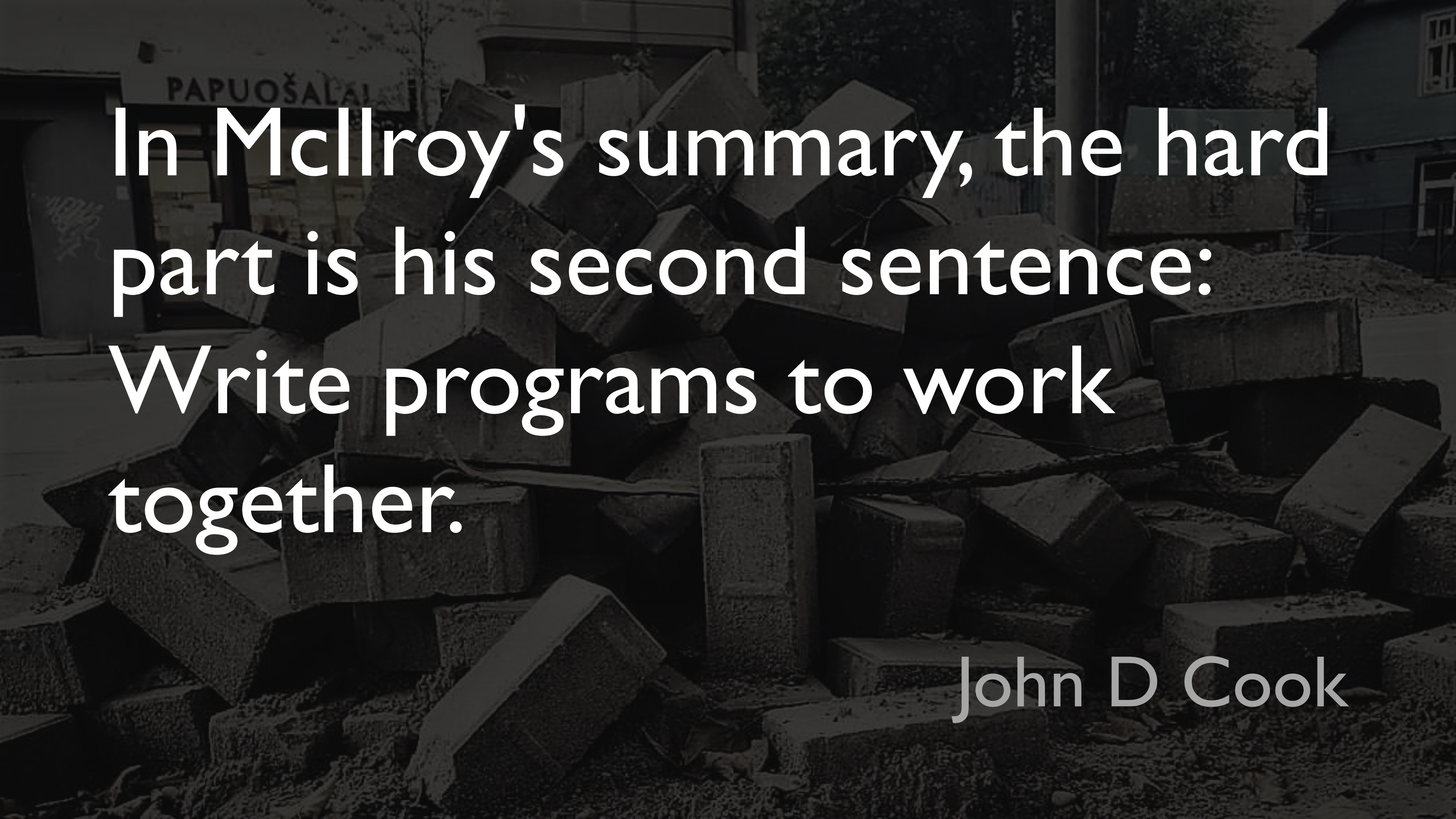

procedural?


μονόλιθος

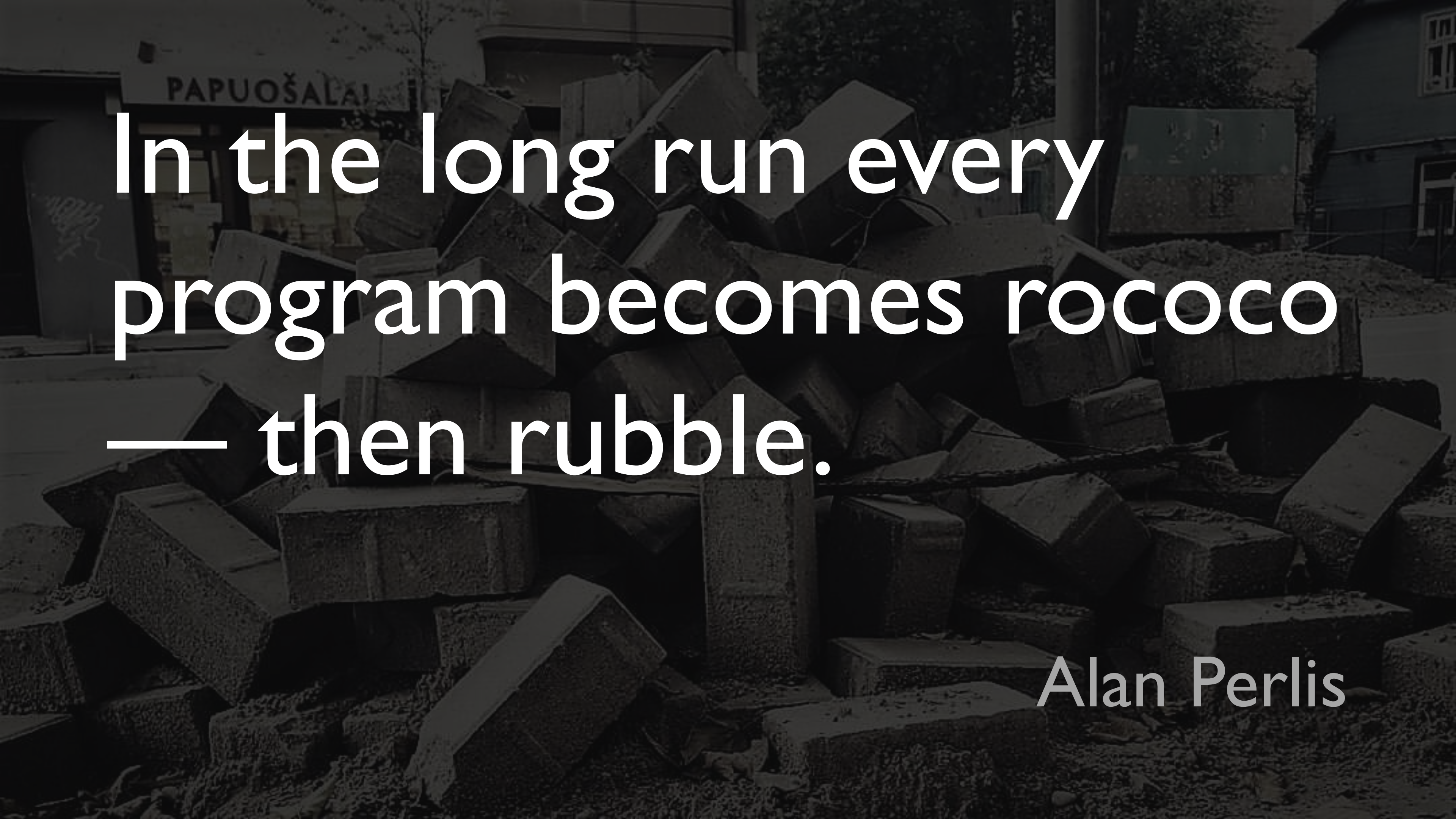
This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together.

Doug McIlroy


µservices

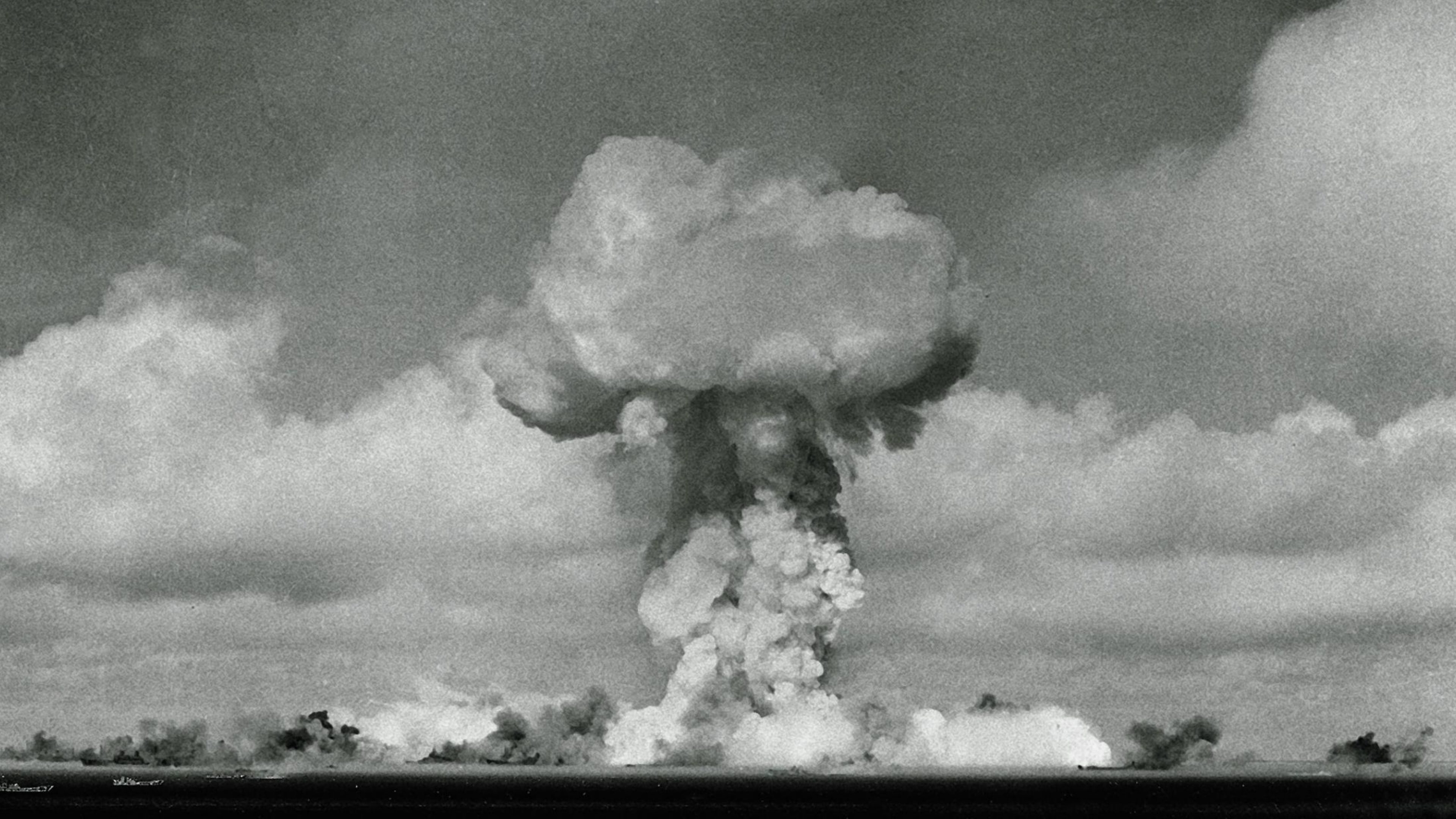

In McIlroy's summary, the hard part is his second sentence: Write programs to work together.
John D Cook


PAPUOŠALAI


PAPUOŠALAI
In the long run every program becomes rococo — then rubble.
Alan Perlis

1960s

1960s

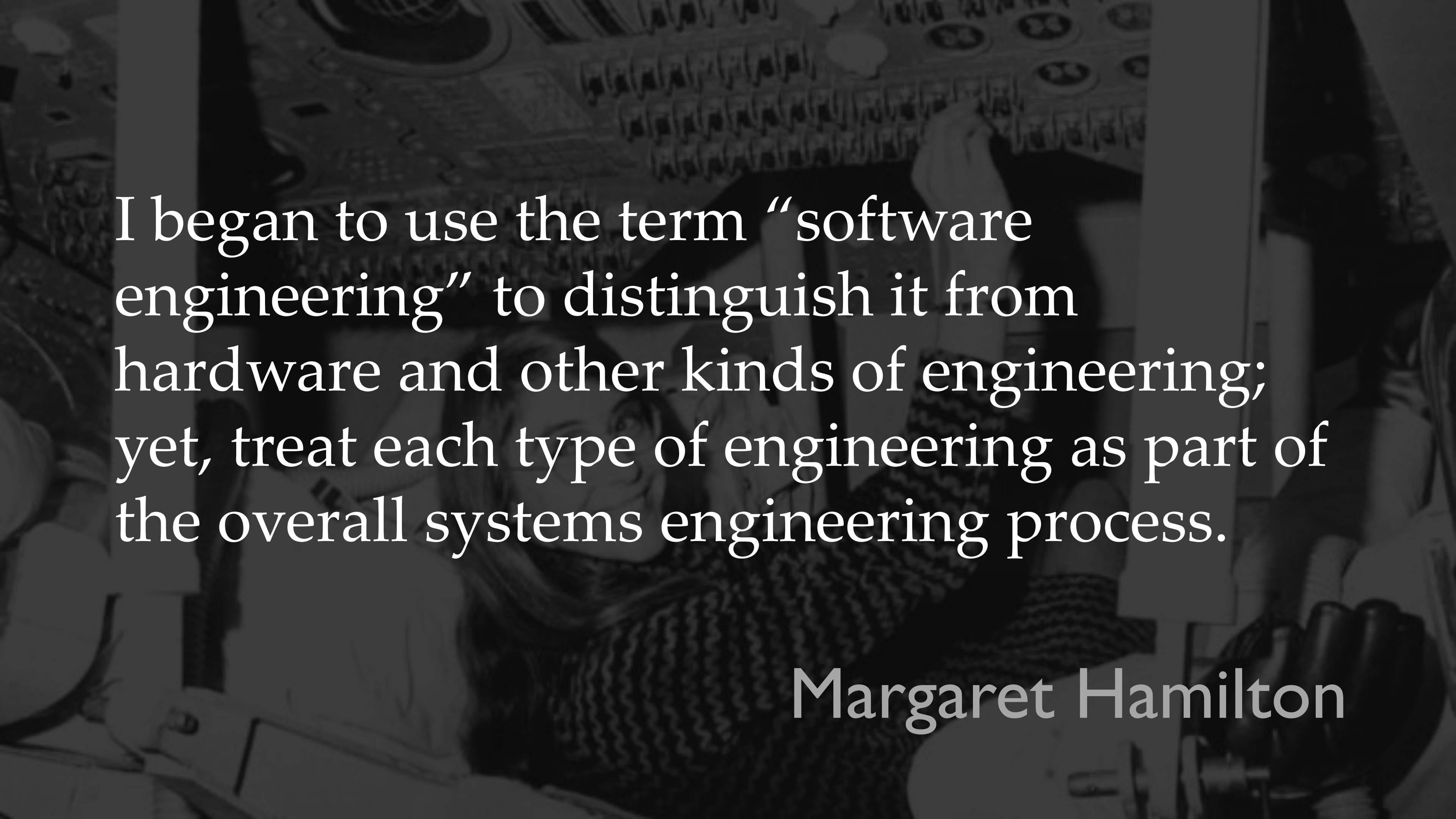


SOFTWARE ENGINEERING

Report on a conference sponsored by the

NATO SCIENCE COMMITTEE

Garmisch, Germany, 7th to 11th October 1968


I began to use the term "software engineering" to distinguish it from hardware and other kinds of engineering; yet, treat each type of engineering as part of the overall systems engineering process.
Margaret Hamilton


2001: A SPACE ODYSSEY

SOFTWARE ENGINEERING

Report on a conference sponsored by the

NATO SCIENCE COMMITTEE

Garmisch, Germany, 7th to 11th October 1968

SOFTWARE ENGINEERING

Define a subset of the system which is small enough to bring to an operational state [...] then build on that subsystem.

Report on a conference sponsored by the

NATO SCIENCE COMMITTEE

Garmisch, Germany, 7th to 11th October 1968

E E David

This strategy requires that the system be designed in modules which can be realized, tested, and modified independently, apart from conventions for intermodule communication.

E E David

SOFTWARE ENGINEERING

The design process is an iterative one.

Report on a conference sponsored by the

NATO SCIENCE COMMITTEE

Garmisch, Germany, 7th to 11th October 1968

Andy Kinslow

SOFTWARE ENGINEERING

There are two classes of system designers. The first, if given five problems will solve them one at a time.

Andy Kinslow

Report on a conference sponsored by the

NATO SCIENCE COMMITTEE

Garmisch, Germany, 7th to 11th October 1968

The second will come back and announce that these aren't the real problems, and will eventually propose a solution to the single problem which underlies the original five.

Andy Kinslow

This is the ‘system type’ who is great during the initial stages of a design project. However, you had better get rid of him after the first six months if you want to get a working system.

Andy Kinslow

A software system can best be designed if the testing is interlaced with the designing instead of being used after the design.

Alan Perlis

```
proc is leap year = (int year) bool:
    skip;
```

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
Algol 68
Edited by
A. van Wijngaarden, B. J. Mailloux,
long
int
void
bool
char
union
short
struct

```
proc is leap year = (int year) bool:
    false;

[] proposition leap year spec =
(
    ("Years not divisible by 4 are not leap years",
     void: (assert (not is leap year (1967))))
);
```

```
mode proposition = struct (string name, proc void test);
```

```
proc is leap year = (int year) bool:
    false;

[] proposition leap year spec =
(
    ("Years not divisible by 4 are not leap years",
     void: (assert (not is leap year (1967))))
);





test (leap year spec)
```

```
mode proposition = struct (string name, proc void test);

proc test = ([] proposition spec) void:
    for entry from lwb spec to upb spec
    do
        print (name of spec [entry]);
        test of spec [entry];
        print (new line)
    od;
```

```
proc is leap year = (int year) bool:
    year mod 4 = 0;

[] proposition leap year spec =
(
    ("Years not divisible by 4 are not leap years",
     void: (assert (not is leap year (1967)))),
    ("Years divisible by 4 but not by 100 are leap years",
     void: (assert (is leap year (1968))))
);

test (leap year spec)
```

```
proc is leap year = (int year) bool:
    year mod 4 = 0 and year mod 100 /= 0;

[] proposition leap year spec =
(
    ("Years not divisible by 4 are not leap years",
     void: (assert (not is leap year (1967)))),
    ("Years divisible by 4 but not by 100 are leap years",
     void: (assert (is leap year (1968)))),
    ("Years divisible by 100 but not by 400 are not leap years",
     void: (assert (not is leap year (1900))))
);



test (leap year spec)
```

```
proc is leap year = (int year) bool:
    year mod 4 = 0 and year mod 100 /= 0 or year mod 400 = 0;

[] proposition leap year spec =
(
    ("Years not divisible by 4 are not leap years",
     void: (assert (not is leap year (1967)))),
    ("Years divisible by 4 but not by 100 are leap years",
     void: (assert (is leap year (1968)))),
    ("Years divisible by 100 but not by 400 are not leap years",
     void: (assert (not is leap year (1900)))),
    ("Years divisible by 400 are leap years",
     void: (assert (is leap year (2000))))
);

test (leap year spec)
```

LISP 1.5 Programmer's Manual

The Computation Center
and Research Laboratory of Electronics

Massachusetts Institute of Technology

```
proc is leap year = (int year) bool:
    year mod 4 = 0 and year mod 100 /= 0 or year mod 400 = 0;

[] proposition leap year spec =
(
    ("Years not divisible by 4 are not leap years",
     with (2018, 2001, 1967, 1), expect (false)),
    ("Years divisible by 4 but not by 100 are leap years",
     with (2016, 1984, 1968, 4), expect (true)),
    ("Years divisible by 100 but not by 400 are not leap years",
     with (2100, 1900, 100), expect (false)),
    ("Years divisible by 400 are leap years",
     with (2000, 1600, 400), expect (true))
);

test (is leap year, leap year spec)
```

```
mode expect = bool;
mode with = flex [1:0] int;
mode proposition = struct (string name, with inputs, expect result);
```

```
proc test = (proc (int) bool function, [] proposition spec) void:
    for entry from lwb spec to upb spec
    do
        print (name of spec [entry]);
        string report := "", separator := "  failed for ";
        [] int inputs = inputs of spec [entry];
        for value from lwb inputs to upb inputs
        do
            if
                bool expected = result of spec [entry];
                function (inputs [value]) /= expected
            then
                report +:= separator + whole(inputs[value], 0);
                separator := " "
            fi
        od;
        print (if report = "" then (new line) else (new line, report, new line) fi)
    od;
```

```
proc test = (proc (int) bool function, [] proposition spec) void:
    for entry from lwb spec to upb spec
    do
        print (name of spec [entry]);
        string report := "", separator := "  failed for ";
        [] int inputs = inputs of spec [entry];
        for value from lwb inputs to upb inputs
        do
            if
                bool expected = result of spec [entry];
                function (inputs [value]) /= expected
            then
                report +:= separator + whole(inputs[value], 0);
                separator := " "
            fi
        od;
        print (if report = "" then (new line) else (new line, report, new line) fi)
    od;
```

```
proc test = (proc (int) bool function, [] proposition spec) void:
    for entry from lwb spec to upb spec
    do
        print (name of spec [entry]);
        string report := "", separator := "  failed for ";
        [] int inputs = inputs of spec [entry];
        for value from lwb inputs to upb inputs
        do
            if
                bool expected = result of spec [entry];
                function (inputs [value]) /= expected
            then
                report +:= separator + whole(inputs[value], 0);
                separator := " "
            fi
        od;
        print (if report = "" then (new line) else (new line, report, new line) fi)
    od;
```

```
proc test = (proc (int) bool function, [] proposition spec) void:
    for entry from lwb spec to upb spec
    do
        print (name of spec [entry]);
        string report := "", separator := "  failed for ";
        [] int inputs = inputs of spec [entry];
        for value from lwb inputs to upb inputs
        do
            if
                bool expected = result of spec [entry];
                function (inputs [value]) /= expected
            then
                report +:= separator + whole(inputs[value], 0);
                separator := " "
            fi
        od;
        print (if report = "" then (new line) else (new line, report, new line) fi)
    od;
```

```
proc test = (proc (int) bool function, [] proposition spec) void:
    for entry from lwb spec to upb spec
    do
        print (name of spec [entry]);
        string report := "", separator := "  failed for ";
        [] int inputs = inputs of spec [entry];
        for value from lwb inputs to upb inputs
        do
            if
                bool expected = result of spec [entry];
                function (inputs [value]) /= expected
            then
                report +:= separator + whole(inputs[value], 0);
                separator := " "
            fi
        od;
        print (if report = "" then (new line) else (new line, report, new line) fi)
    od;
```

```
proc test = (proc (int) bool function, [] proposition spec) void:
    for entry from lwb spec to upb spec
    do
        print (name of spec [entry]);
        string report := "", separator := "  failed for ";
        [] int inputs = inputs of spec [entry];
        for value from lwb inputs to upb inputs
        do
            if
                bool expected = result of spec [entry];
                function (inputs [value]) /= expected
            then
                report +:= separator + whole(inputs[value], 0);
                separator := " "
            fi
        od;
        print (if report = "" then (new line) else (new line, report, new line) fi)
    od;
```

```
proc test = (proc (int) bool function, [] proposition spec) void:
    for entry from lwb spec to upb spec
    do
        print (name of spec [entry]);
        string report := "", separator := "  failed for ";
        [] int inputs = inputs of spec [entry];
        for value from lwb inputs to upb inputs
        do
            if
                bool expected = result of spec [entry];
                function (inputs [value]) /= expected
            then
                report +:= separator + whole(inputs[value], 0);
                separator := " "
            fi
        od;
        print (if report = "" then (new line) else (new line, report, new line) fi)
    od;
```

```
proc test = (proc (int) bool function, [] proposition spec) void:
    for entry from lwb spec to upb spec
    do
        print (name of spec [entry]);
        string report := "", separator := "  failed for ";
        [] int inputs = inputs of spec [entry];
        for value from lwb inputs to upb inputs
        do
            if
                bool expected = result of spec [entry];
                function (inputs [value]) /= expected
            then
                report +:= separator + whole(inputs[value], 0);
                separator := " "
            fi
        od;
        print ((report = "" | (new line) | (new line, report, new line)))
    od;
```

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

Alfred Aho

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

SOFTWARE ENGINEERING

There is no such question as testing things after the fact with simulation models, but that in effect the testing and the replacement of simulations with modules that are deeper and more detailed goes on with the simulation model controlling, as it were, the place and order in which these things are done.

NATO SCIENCE COMMITTEE

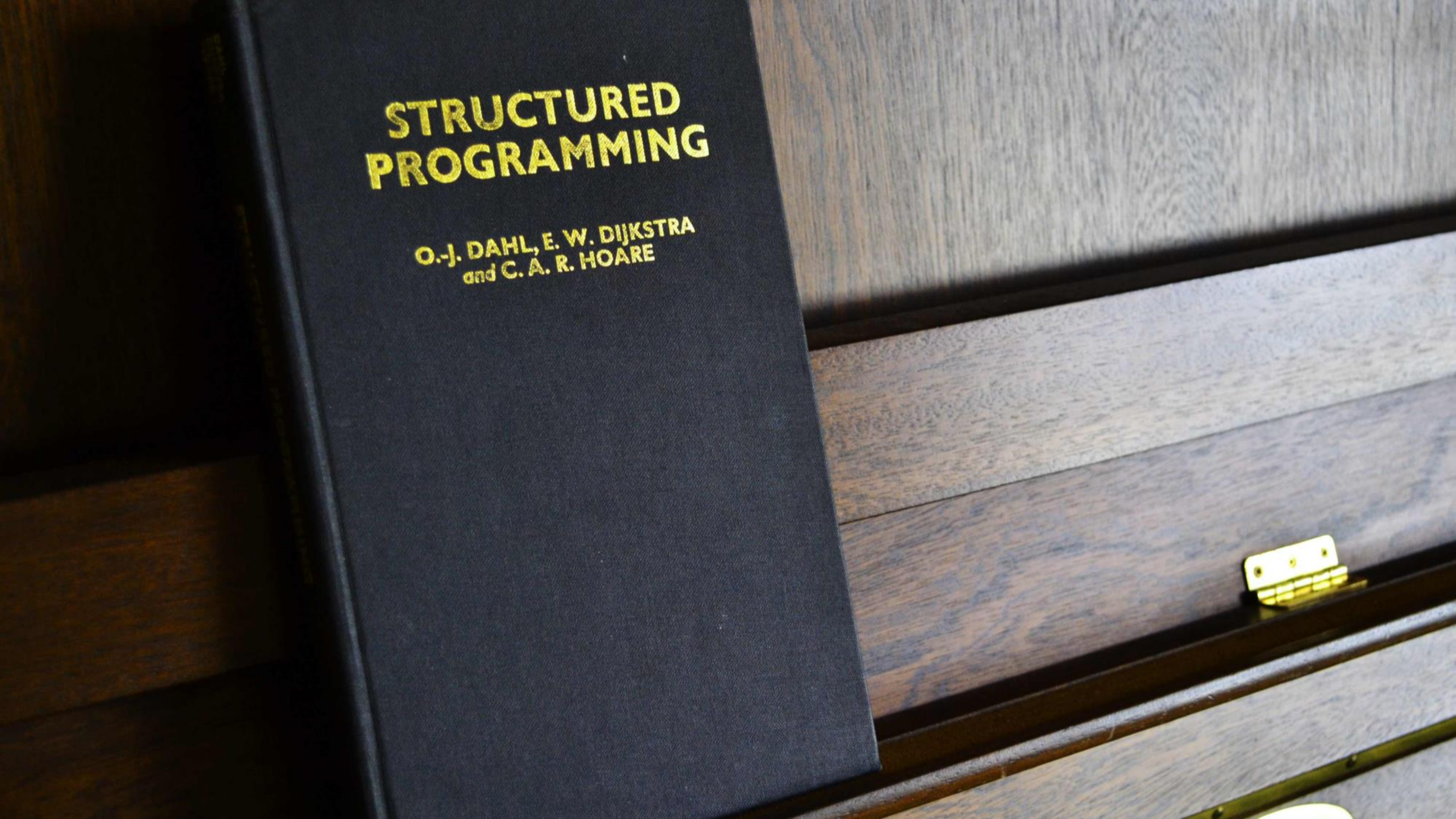
Garmisch, Germany, 7th to 11th Octobe

Alan Perlis

As design work progresses this simulation will gradually evolve into the real system.

The simulation is the design.

Tad B Pinkerton


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

goto

Letters to the Editor

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** $A1$ **else** $A2$), 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 jump-less 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


/ WordFriday
Adam Jacot de Boinod
I NEVER KNEW THERE WAS A WORD FOR IT
'Very funny'
Independent on Sunday
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
T. F. HOAD
word origins
the language report
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
Joie de vivre n.
A COMPREHENSIVE WORD-FINDING DICTIONARY
Chambers
COLLINS
REFERENCE DICTIONARY
MATHEMATICS
E. J. BOROWSKI AND J. M. BORWEIN
Oxford
Samuel Johnson's
LOCK
JEFFREY KACIRK
WORDS
BOTHER ME

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.

```
      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
```

```fortran
      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

```
      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
```

```
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.

```
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

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.

Tom Duff

```
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);
    }
}
```

break

$$
\begin{array}{lll}
\text{Ord } 1(V) & & \Rightarrow R \\
 & 0 & 0 \\
 & m \times \sigma & m \times \sigma
\end{array}
$$

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

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

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

continue

break

return

```
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
}
```

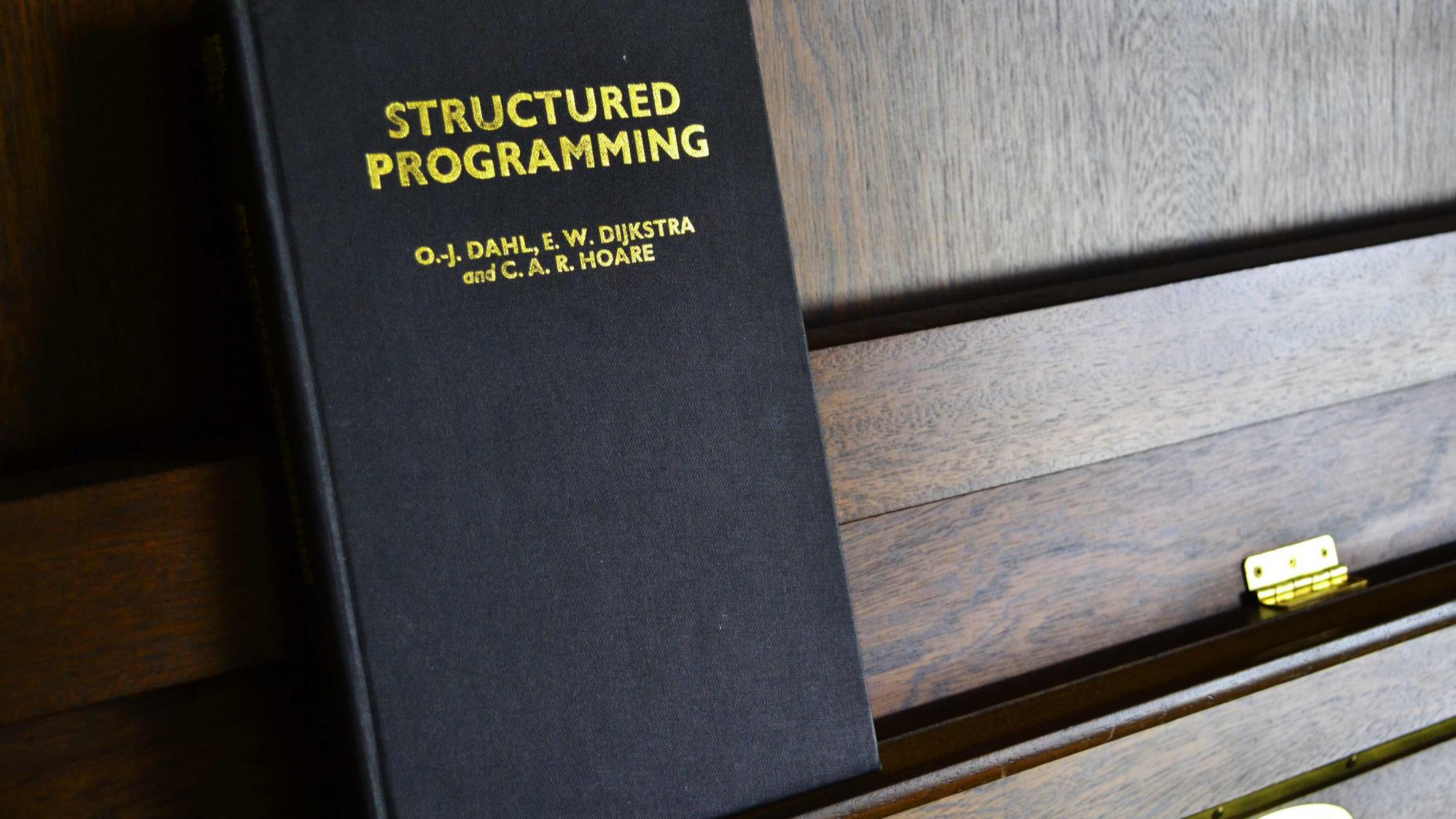
```
def isLeapYear(year)
{
    if (year % 400 == 0)
        return true
    if (year % 100 == 0)
        return false
    if (year % 4 == 0)
        return true
    return false
}
```

```
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;
```

```haskell
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;
```


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.

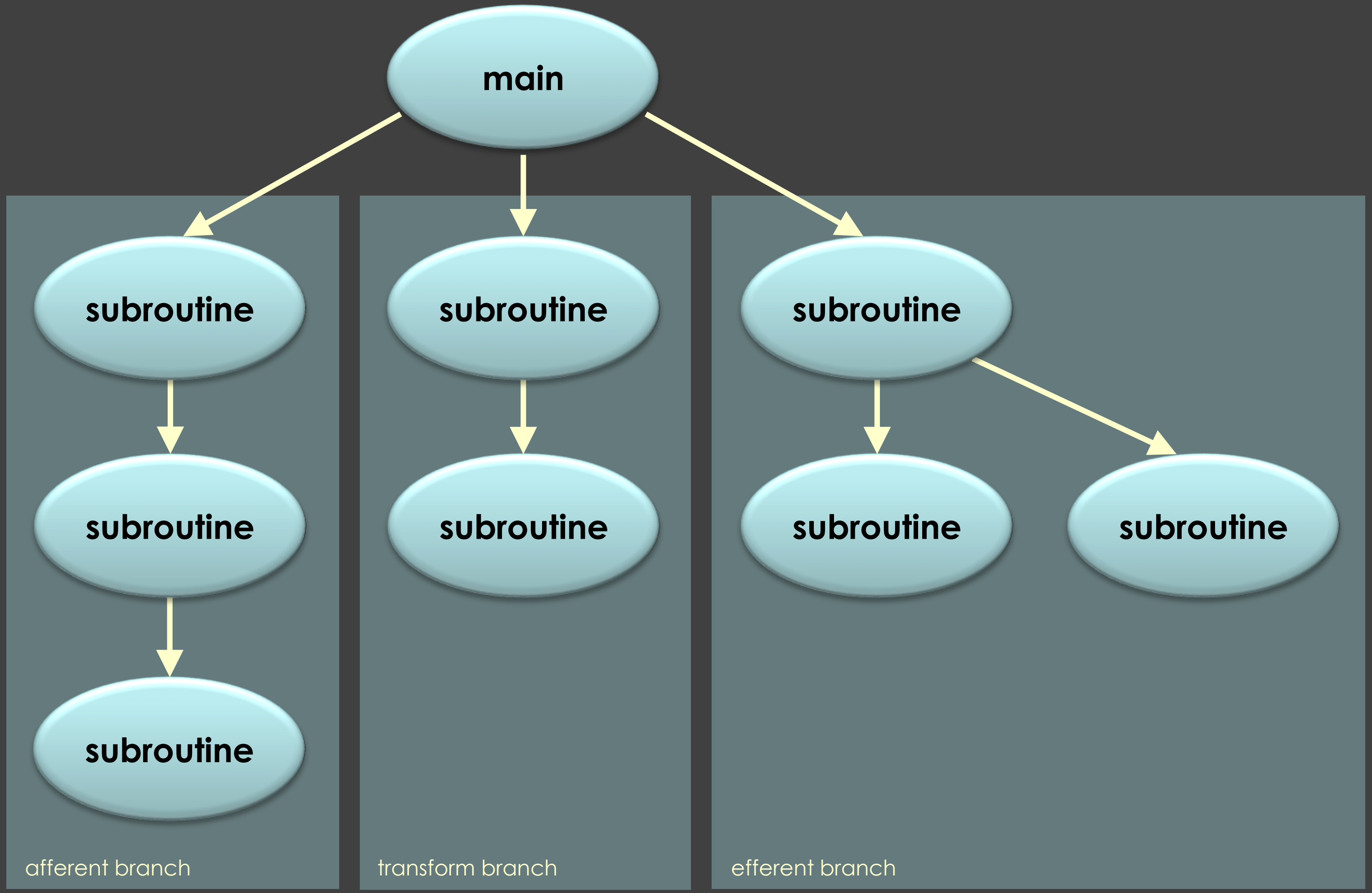
Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"

sequence
selection
iteration

Main Program and Subroutine

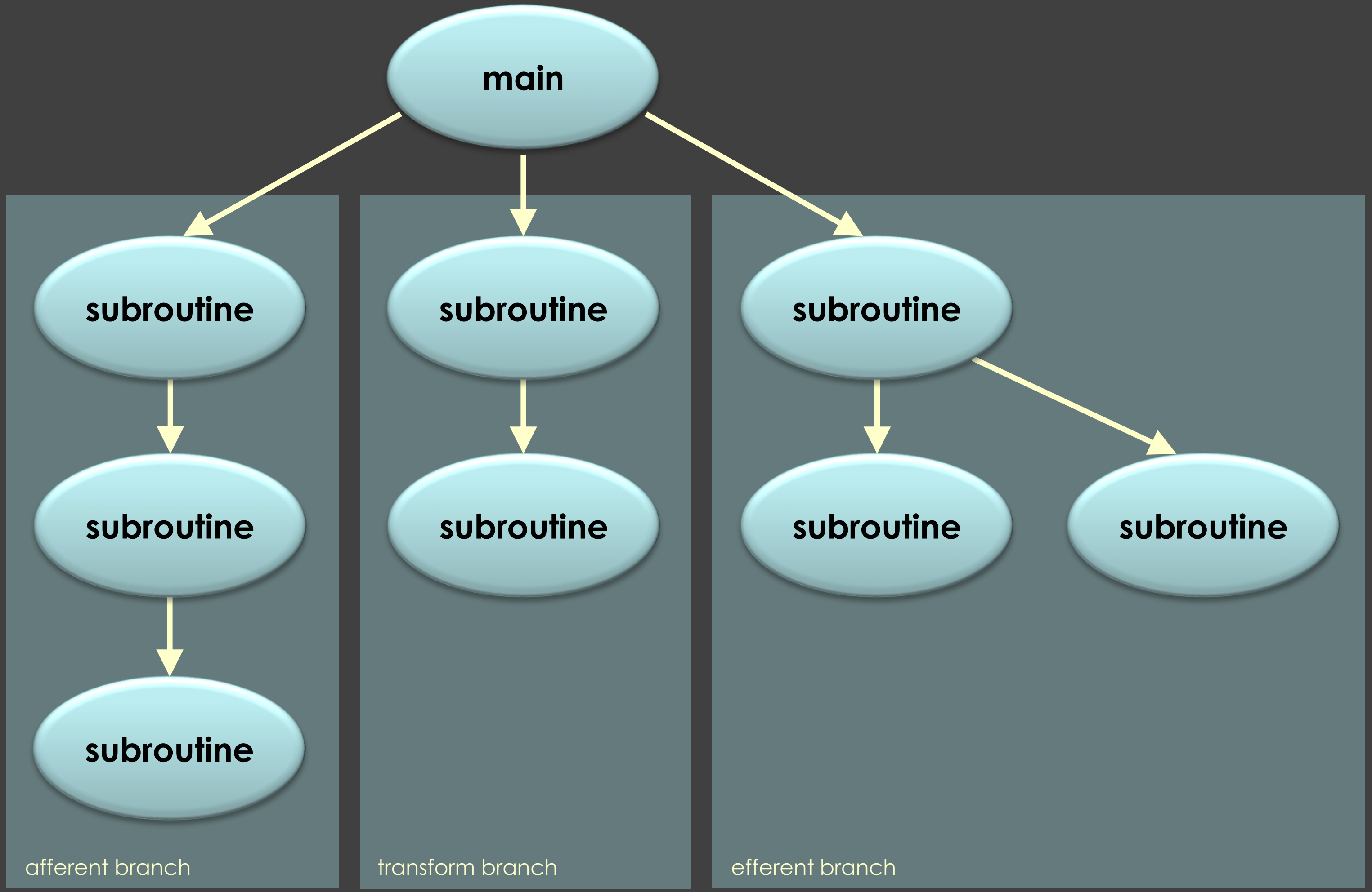
The goal is to decompose a program into smaller pieces to help achieve modifiability. A program is decomposed hierarchically.

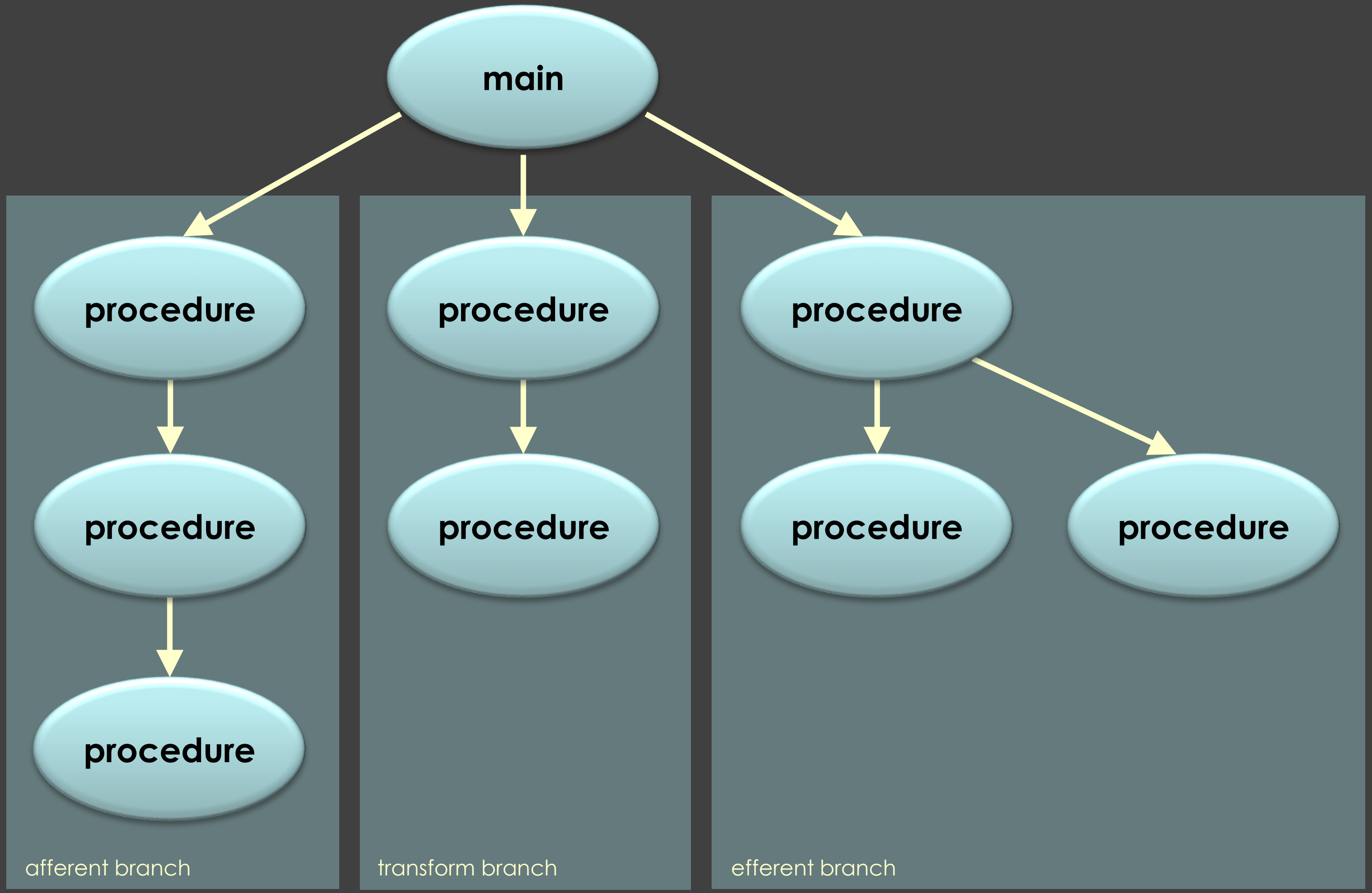
Len Bass, Paul Clements & Rick Kazman
Software Architecture in Practice

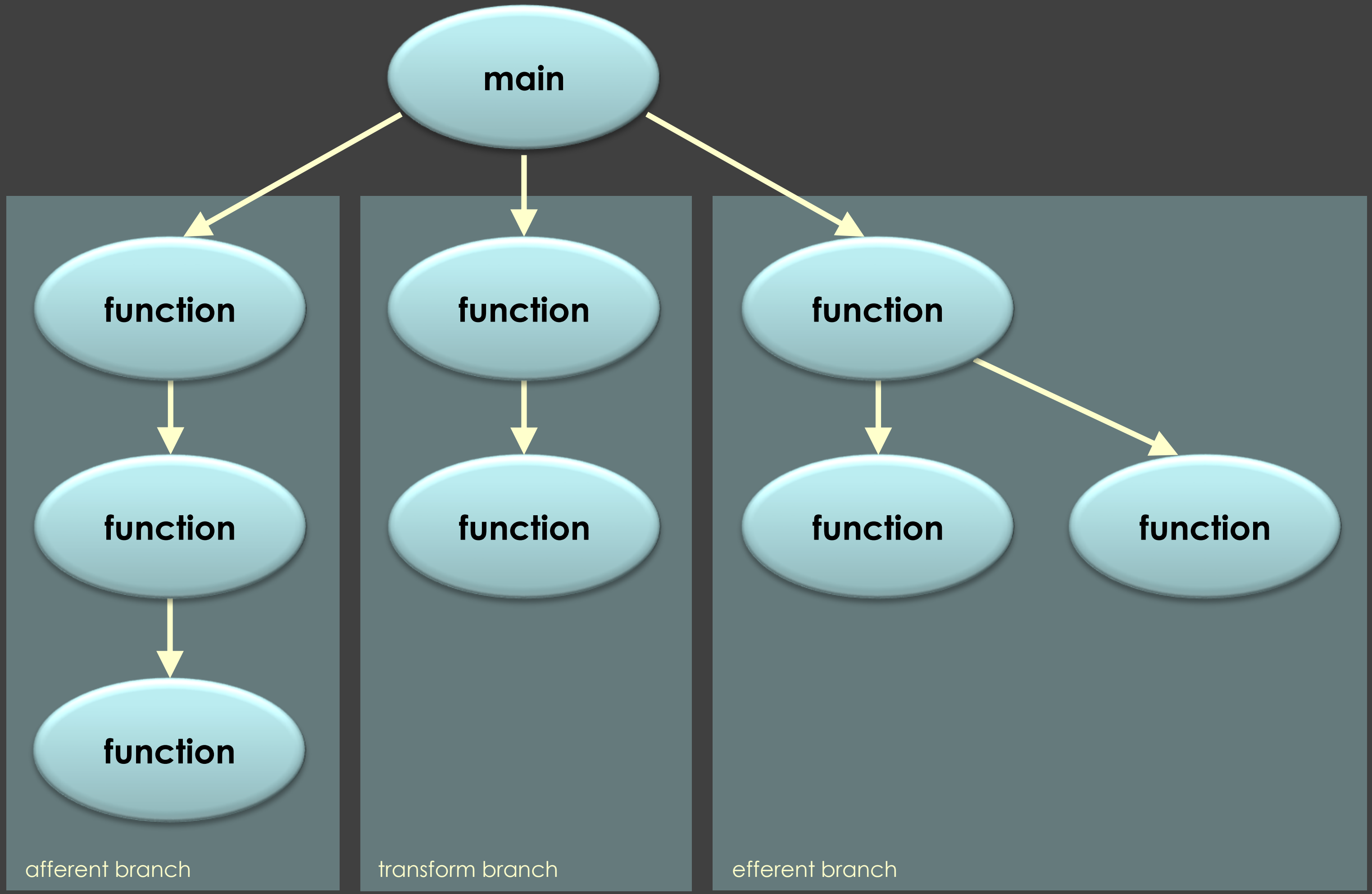

main
subroutine
subroutine
subroutine
subroutine
subroutine
subroutine
subroutine
subroutine
afferent branch
transform branch
efferent branch

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.

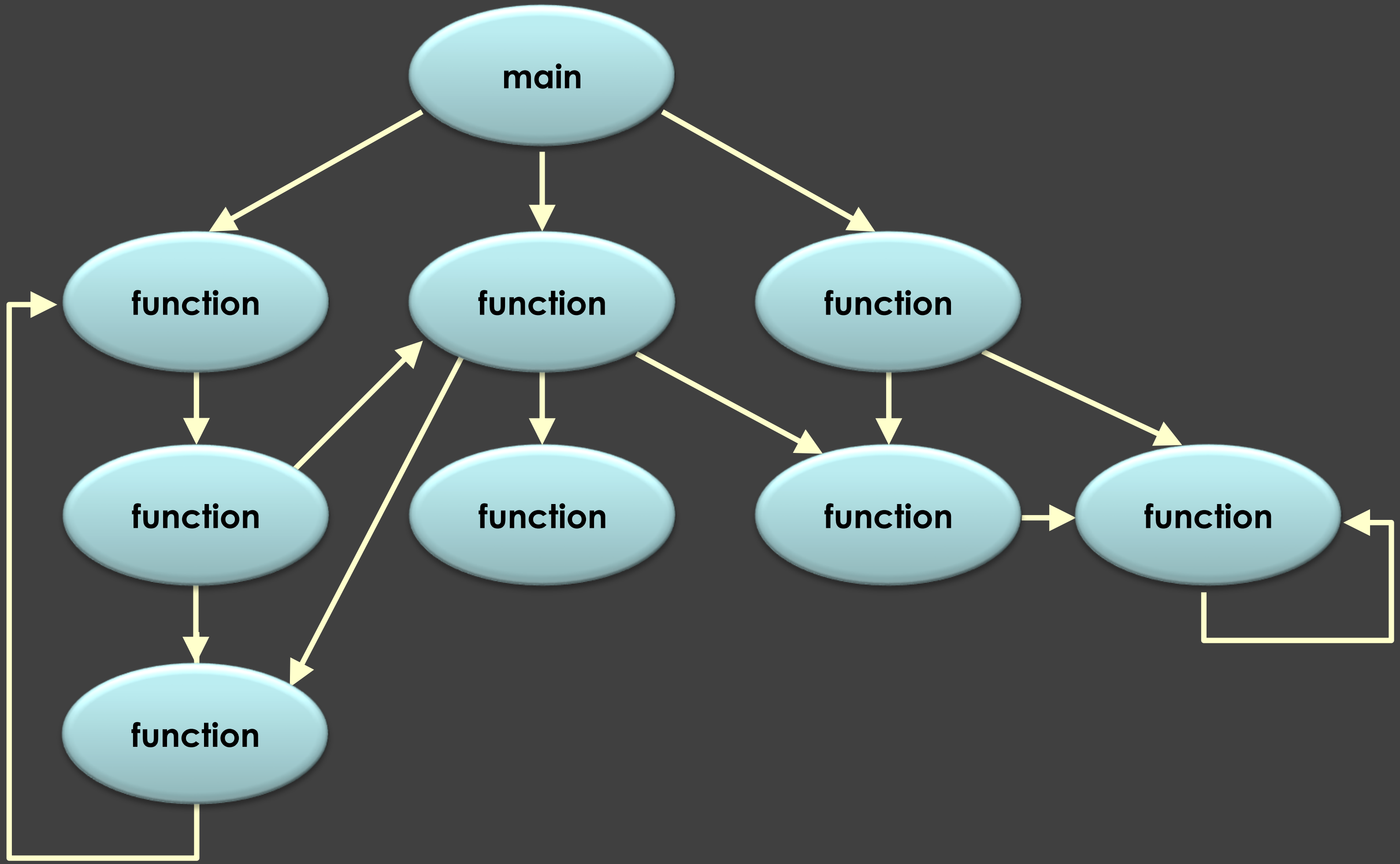
Len Bass, Paul Clements & Rick Kazman
Software Architecture in Practice


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

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

C A R Hoare

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

Alan Perlis

We propose [...] that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.

David L Parnas

"On the Criteria to Be Used in Decomposing Systems into Modules"

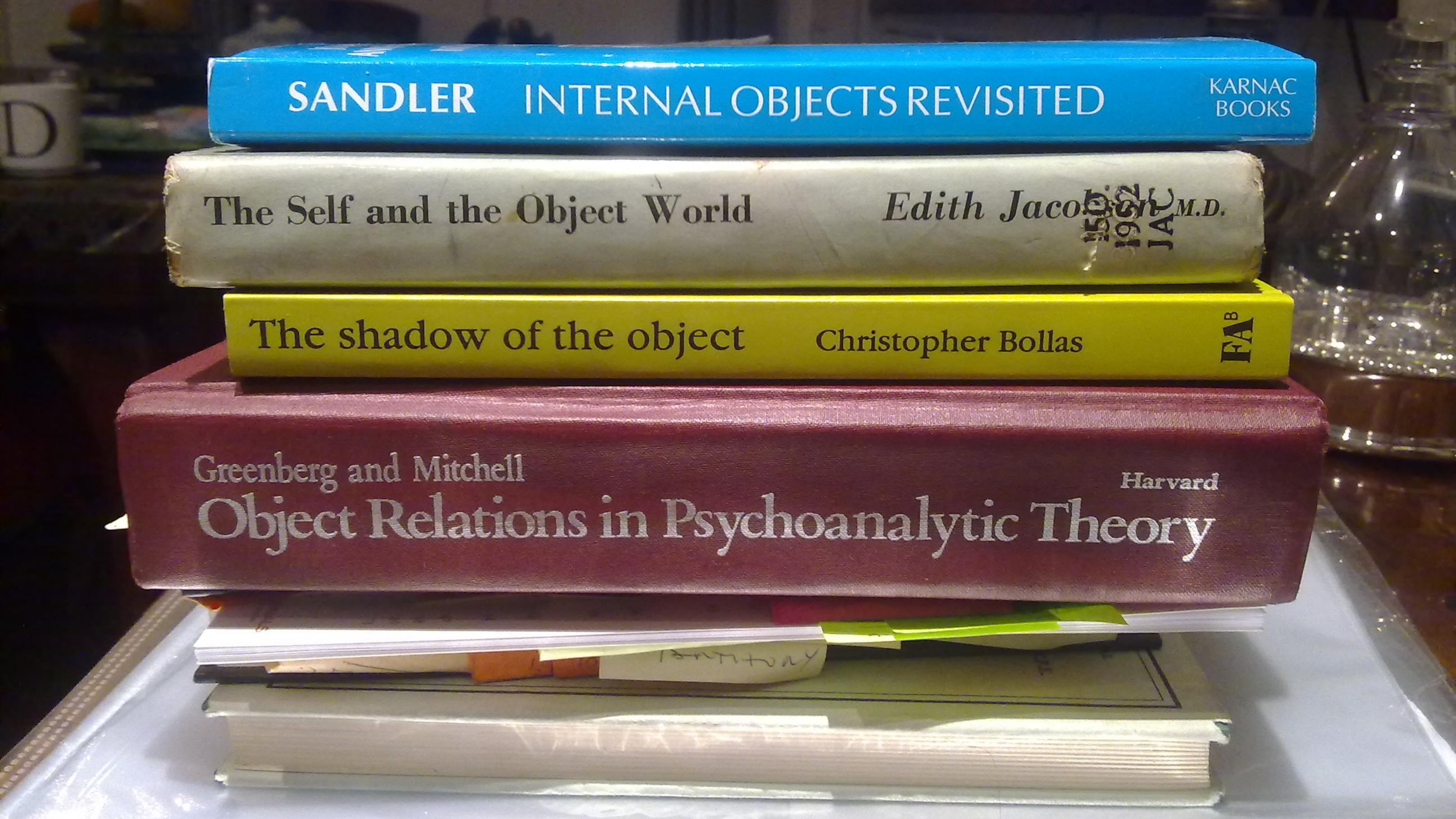
An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects.

Barbara Liskov
"Programming with Abstract Data Types"

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

Barbara Liskov

"Programming with Abstract Data Types"


SANDLER
INTERNAL OBJECTS REVISITED
KARNAC BOOKS
The Self and the Object World
M.D.
1962
JAC
The shadow of the object
Christopher Bollas
FAB
Greenberg and Mitchell
Object Relations in Psychoanalytic Theory
Harvard

```
DEFINITION MODULE Stacks;
    TYPE Stack;
    PROCEDURE New(VAR self: Stack);
    PROCEDURE Delete(VAR self: Stack);
    PROCEDURE Push(self: Stack; top: ARRAY OF CHAR);
    PROCEDURE Pop(self: Stack);
    PROCEDURE Depth(self: Stack): CARDINAL;
    PROCEDURE Top(self: Stack; VAR top: ARRAY OF CHAR);
END Stacks.
```

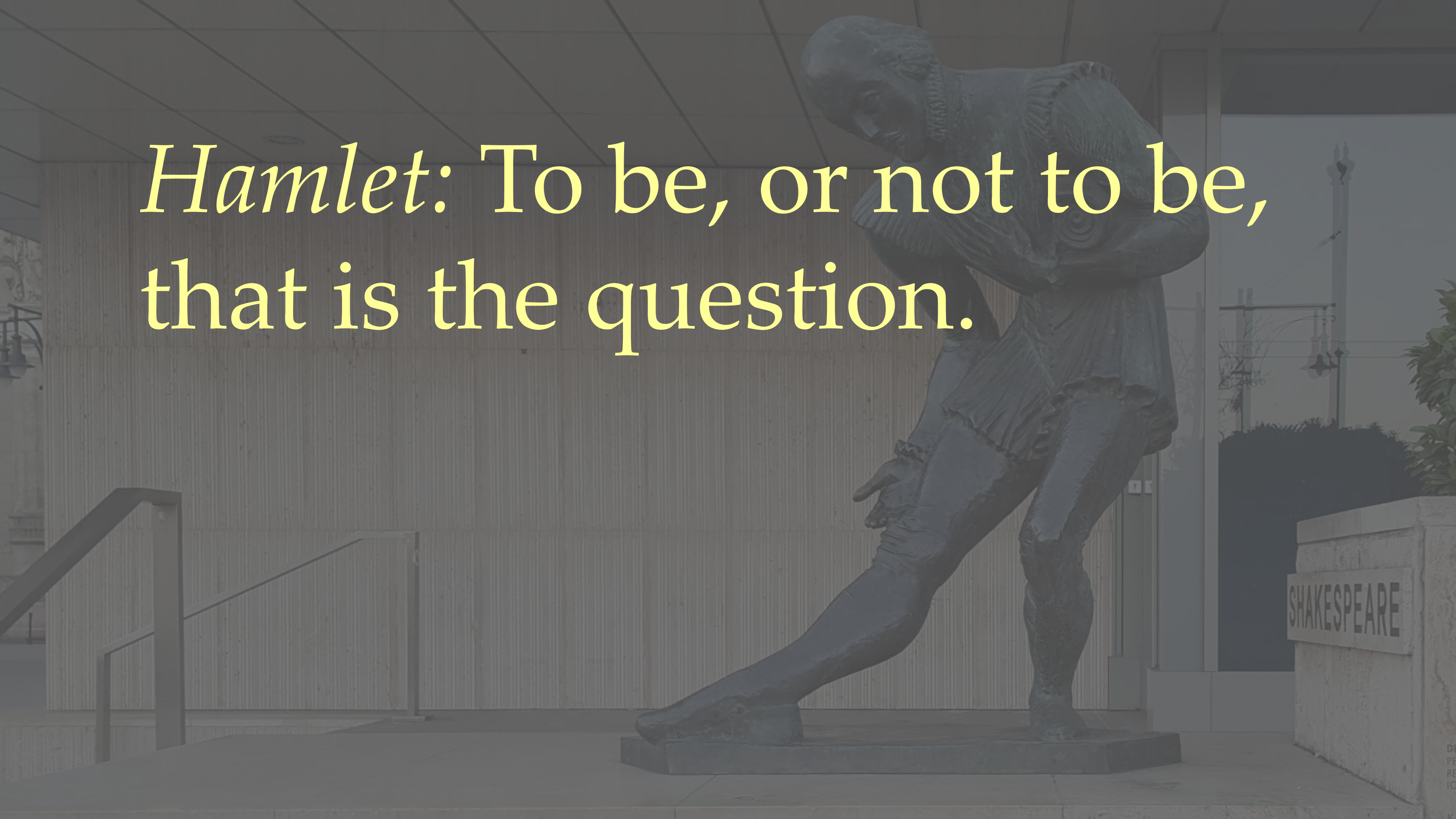
```
#ifdef __cplusplus
extern "C"
{
#endif

typedef struct stack stack;

stack * stack_new(void);
void stack_delete(stack *);

void stack_push(stack *, const char *);
void stack_pop(stack *);
size_t stack_depth(const stack *);
const char * stack_top(const stack *);

#ifdef __cplusplus
}
#endif
```

```
struct stack
{
    const char ** items;
    size_t depth;
};

stack * stack_new(void)
{
    stack * result = (stack *) malloc(sizeof(stack));
    result->items = (const char **) malloc(0);
    result->depth = 0;
    return result;
}

void stack_delete(stack * self)
{
    free(self->items);
    free(self);
}

void stack_push(stack * self, const char * new_top)
{
    self->items = (const char **) realloc(self->items, (self->depth + 1) * sizeof(char *));
    self->items[self->depth] = new_top;
    ++self->depth;
}

void stack_pop(stack * self)
{
    self->items = (const char **) realloc(self->items, (self->depth - 1) * sizeof(char *));
    --self->depth;
}

size_t stack_depth(const stack * self)
{
    return self->depth;
}

const char * stack_top(const stack * self)
{
    return self->items[self->depth - 1];
}
```

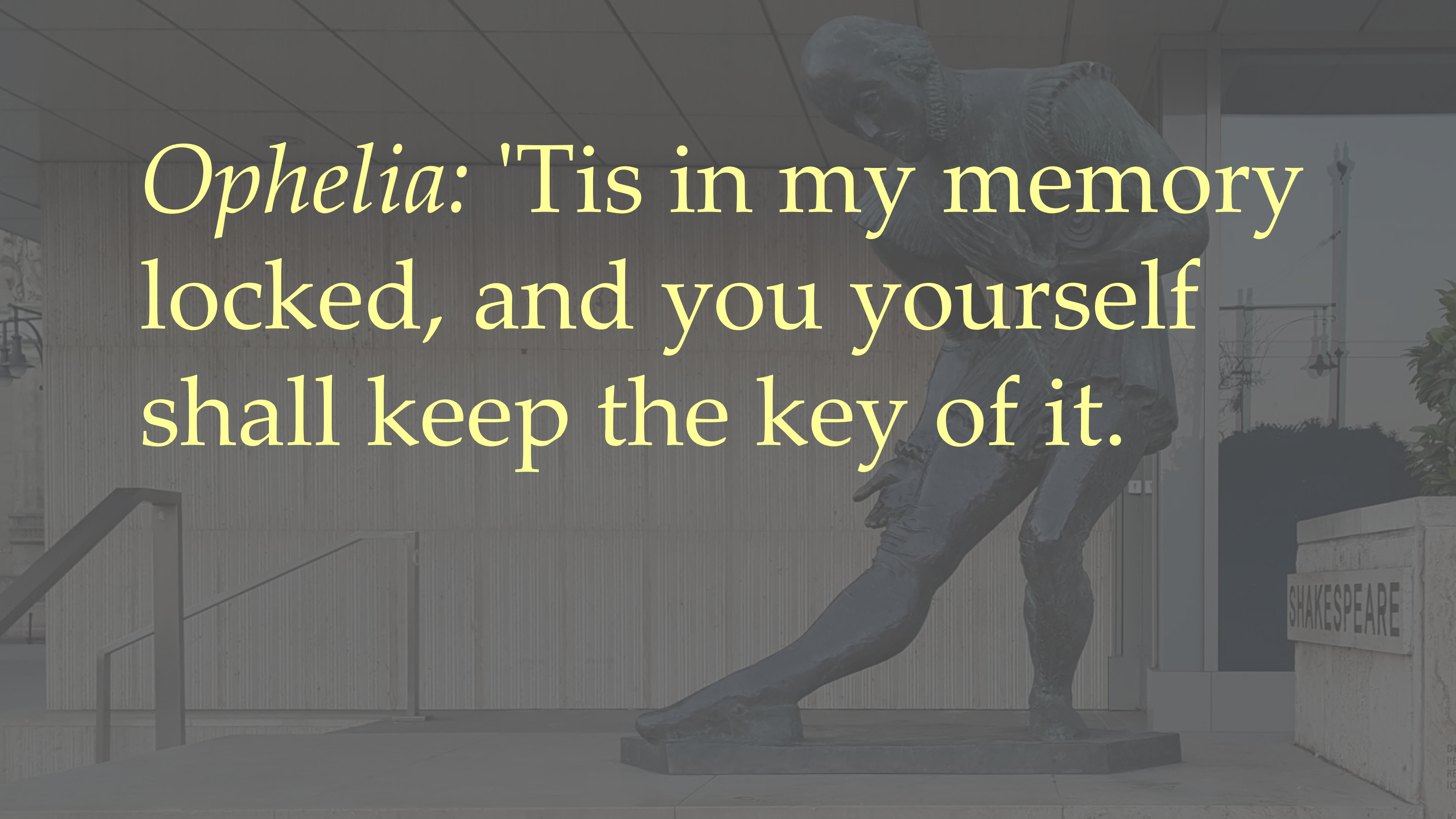
```
extern "C"
{
    struct stack
    {
        std::vector<std::string> items;
    };

    stack * stack_new()
    {
        return new stack;
    }

    void stack_delete(stack * self)
    {
        delete self;
    }

    void stack_push(stack * self, const char * new_top)
    {
        self->items.push_back(new_top);
    }

    void stack_pop(stack * self)
    {
        self->items.pop_back();
    }

    size_t stack_depth(const stack * self)
    {
        return self->items.size();
    }

    const char * stack_top(const stack * self)
    {
        return self->items.back().c_str();
    }
}
```


SHAKESPEARE

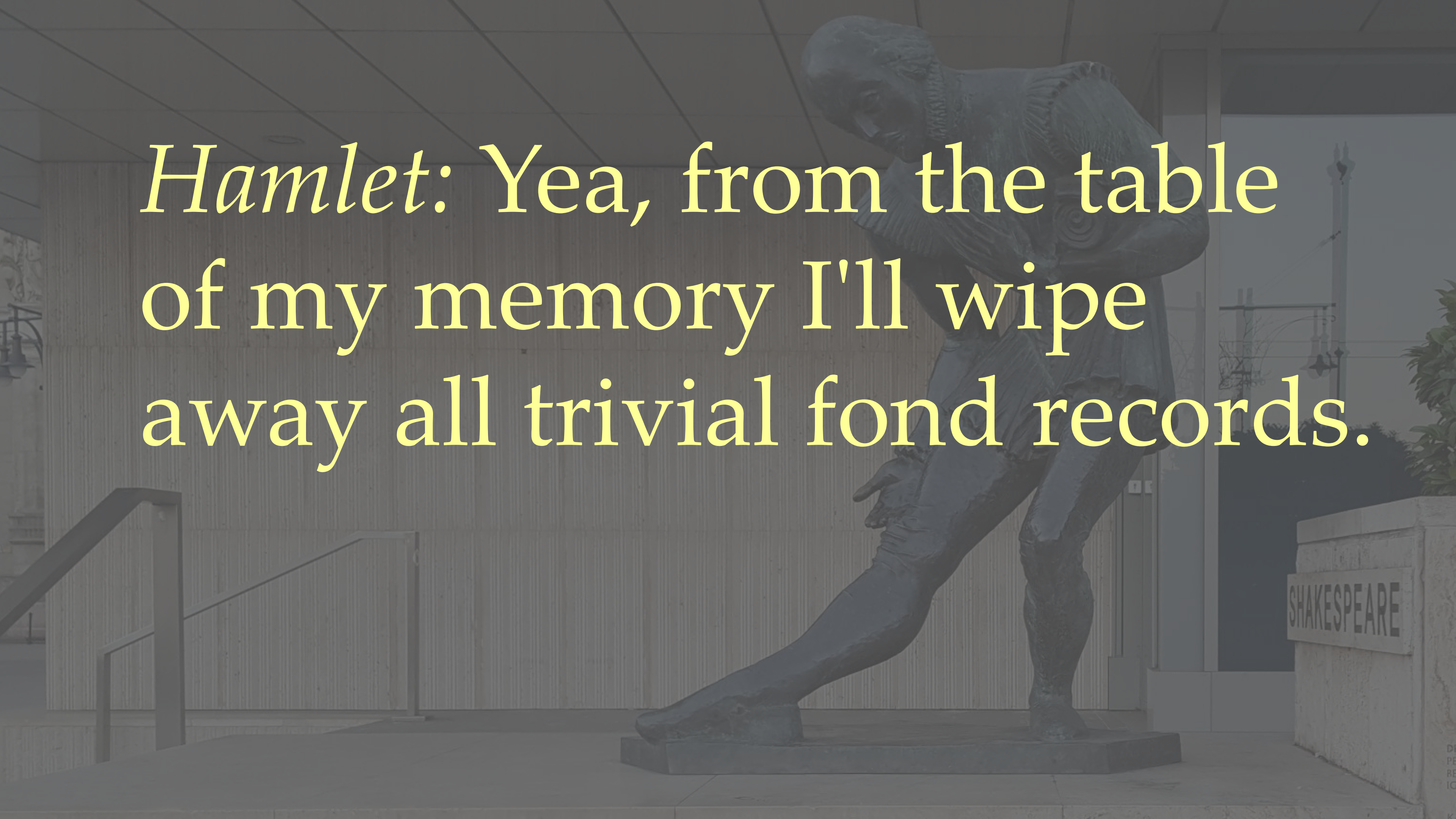
Hamlet: To be, or not to be, that is the question.

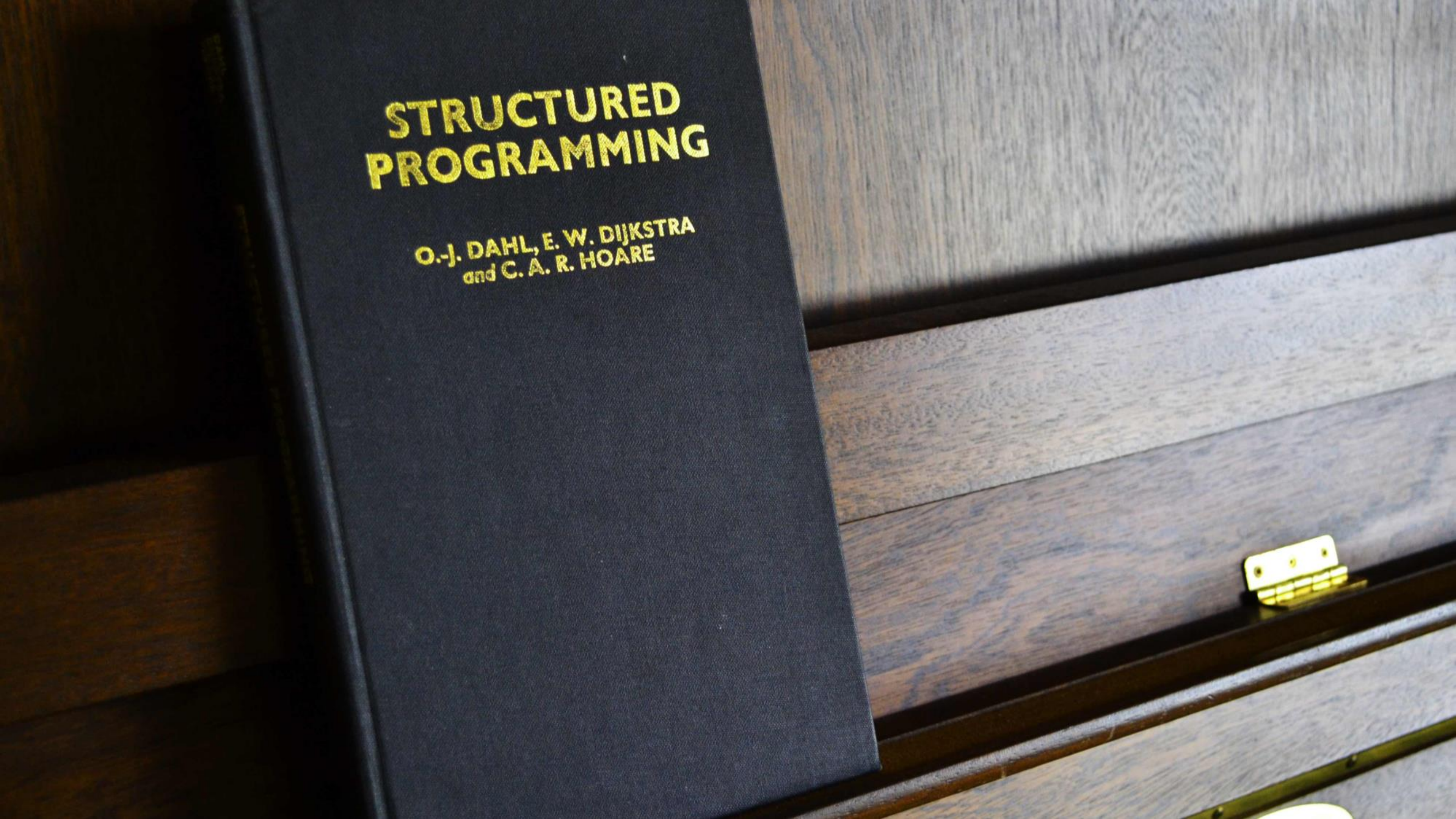


Ophelia: 'Tis in my memory locked, and you yourself shall keep the key of it.



Hamlet: Yea, from the table of my memory I'll wipe away all trivial fond records.

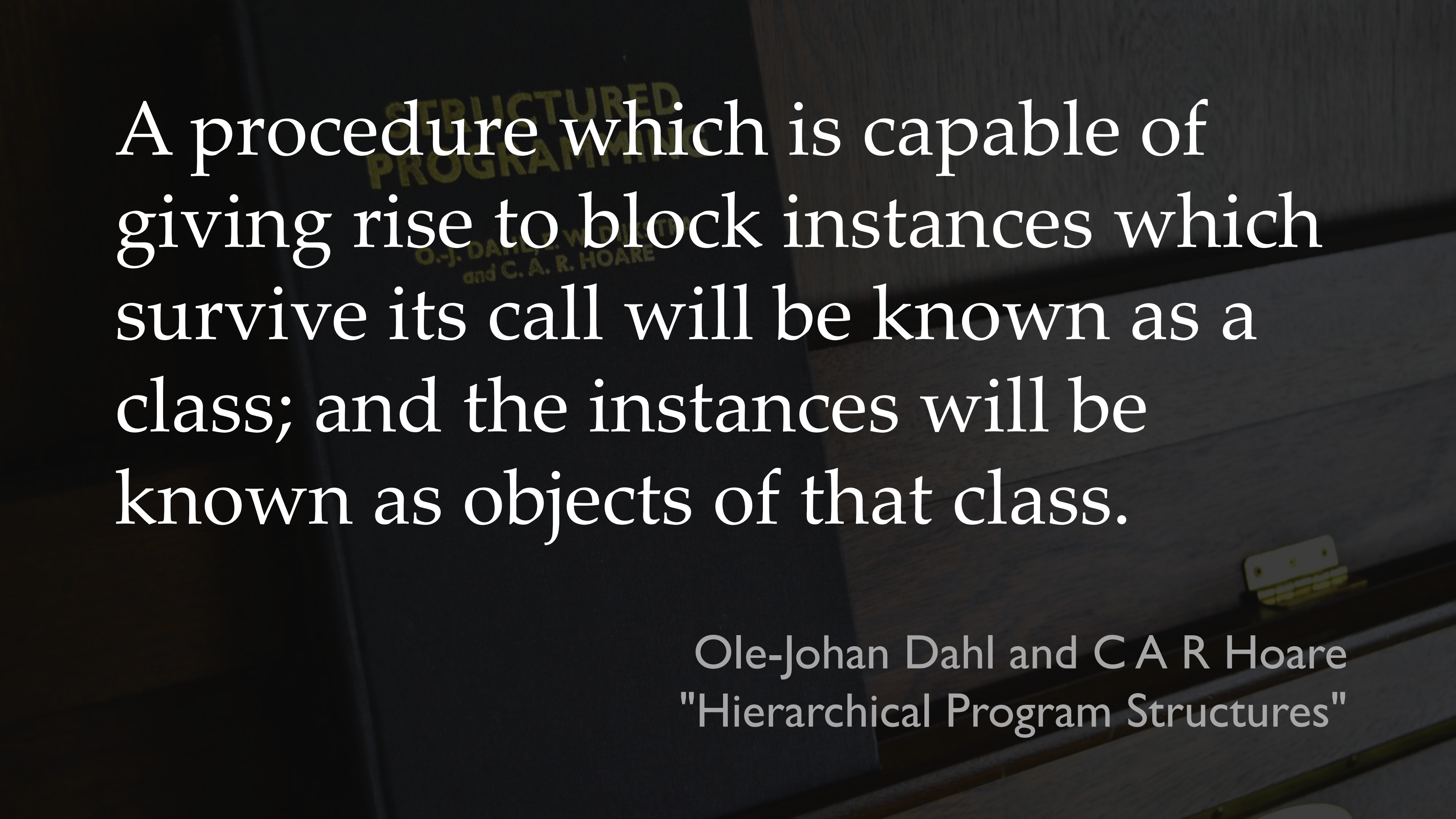



STRUCTURED
PROGRAMMING
O.-J. DAHL, E. W. DIJKSTRA
and C. A. R. HOARE

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"

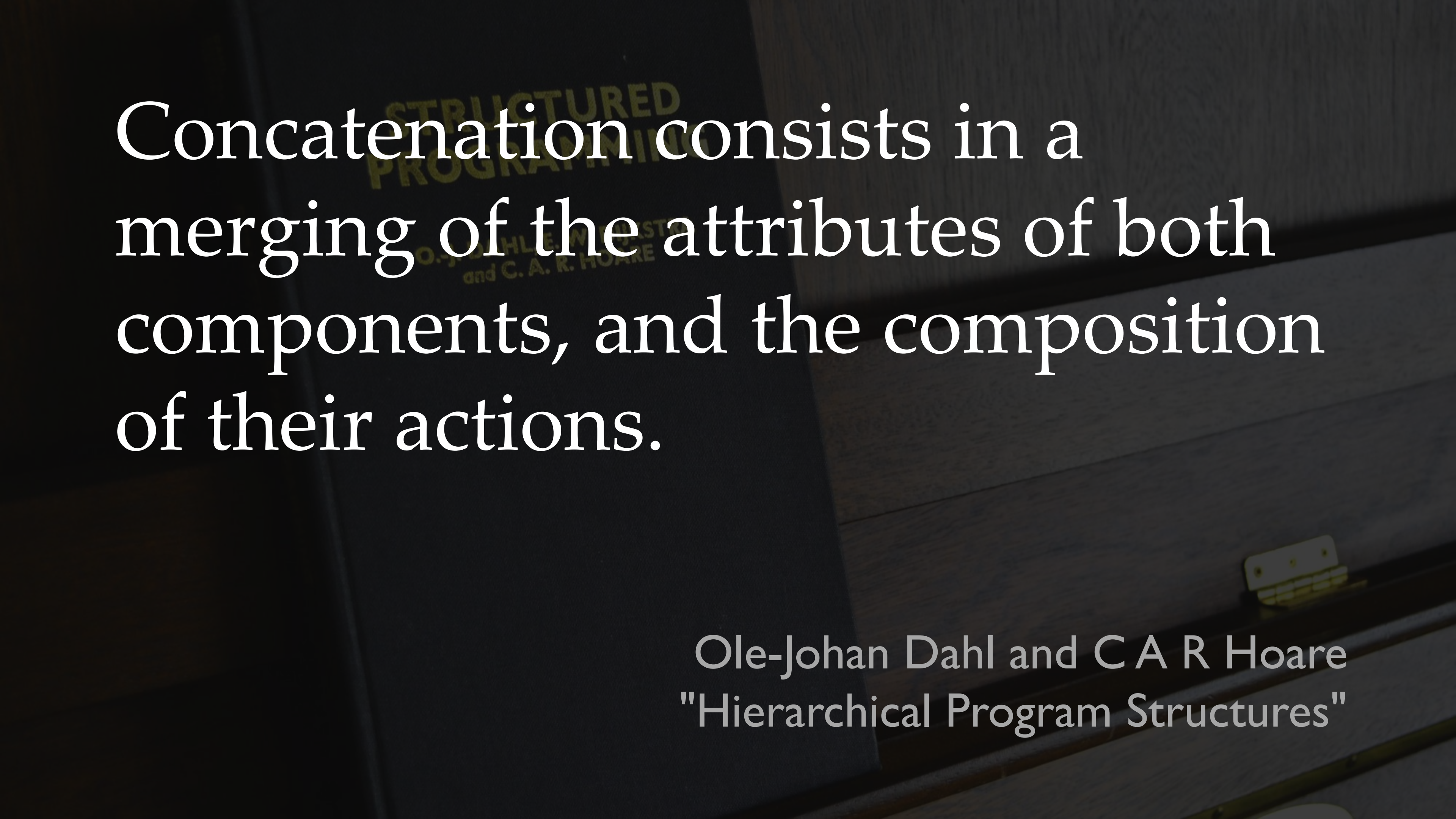
```
class Stack(capacity);
    integer capacity;
begin
    ref(Book) array books(1:capacity);
    integer count;

    procedure Push(top); ...
    procedure Pop; ...
    boolean procedure IsEmpty; ...
    boolean procedure IsFull; ...
    integer procedure Depth; ...
    ref(Book) procedure Top; ...

    count := 0
end;
```

```
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.

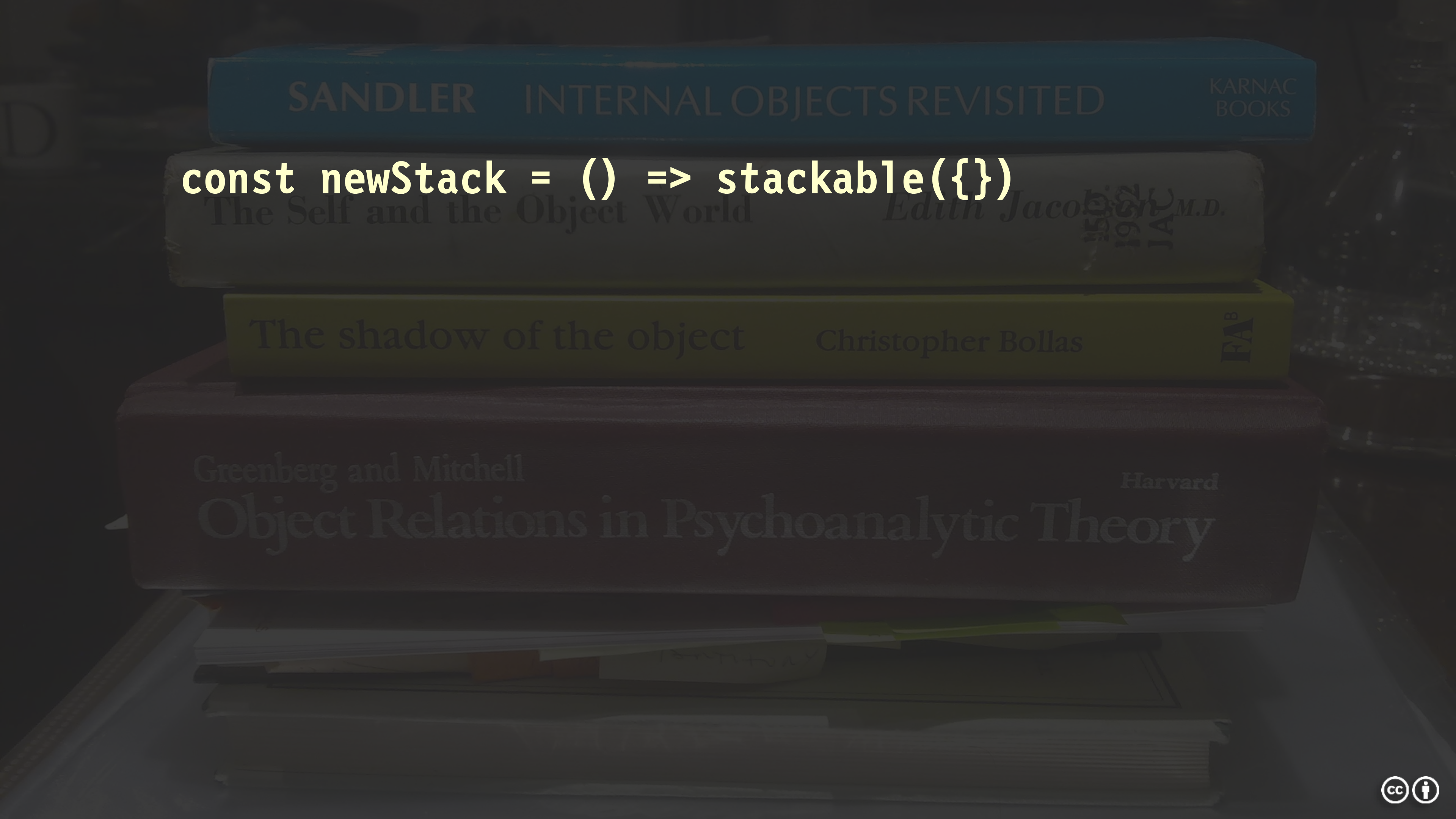
Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"



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

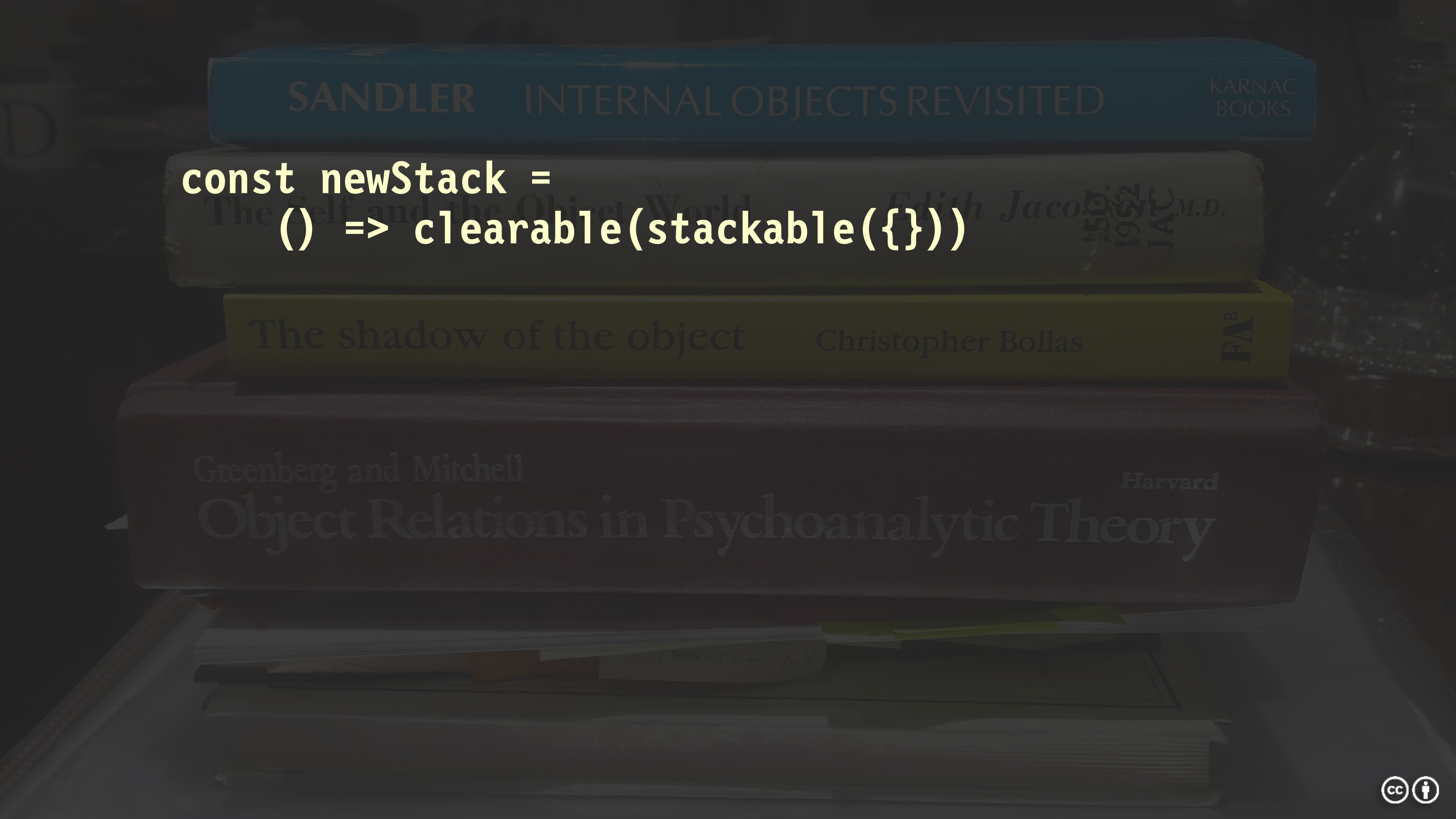
Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"

```
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('ACCU')
            stack.push('2018')
            stack.push('2018')

            assert(stack.depth() === 3)
            assert(stack.top() === '2018')
        },
    ...
}
```

```
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('ACCU')
            stack.push('2018')
            stack.push('2018')

            assert(stack.depth() === 3)
            assert(stack.top() === '2018')
        },
    ...
}
```

```
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('ACCU')
            stack.push('2018')
            stack.push('2018')

            assert(stack.depth() === 3)
            assert(stack.top() === '2018')
        },
    ...
}
```

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"

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

Make — A Program for Maintaining Computer Programs

S. I. Feldman

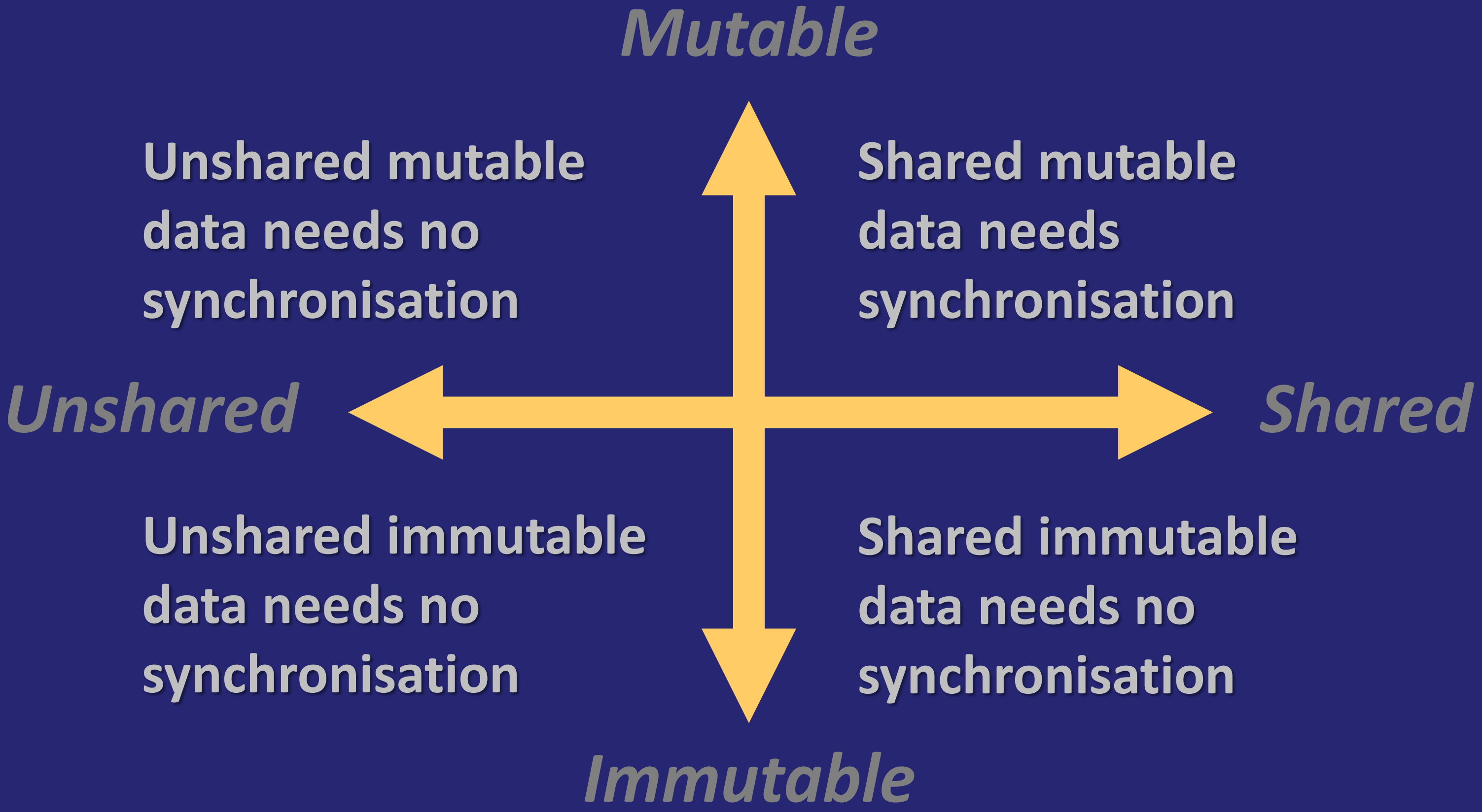
ABSTRACT

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *Make* the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the *Make* command will create the proper files simply, correctly, and with a minimum amount of effort.

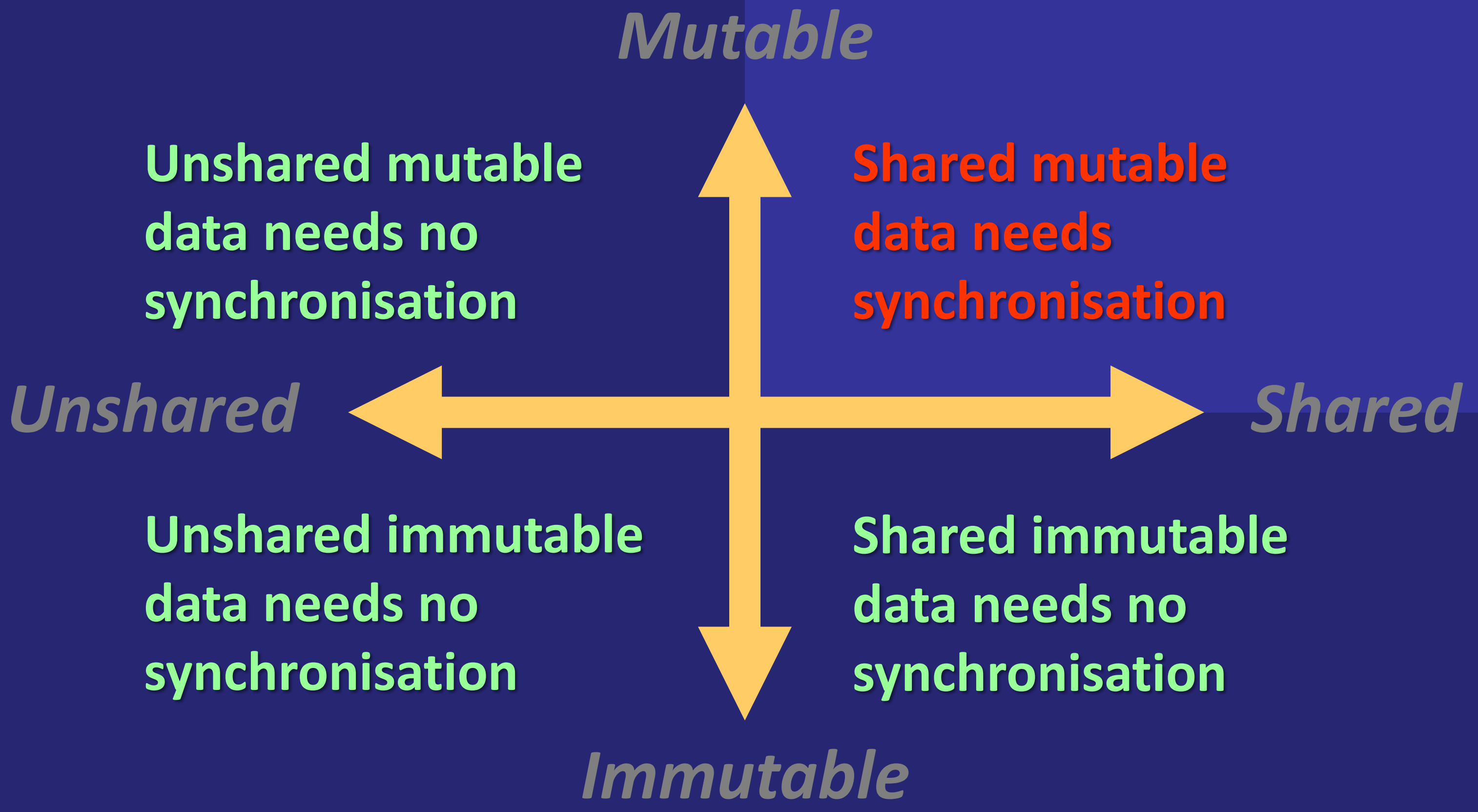
The basic operation of *Make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *Make* does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

August 15, 1978


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

Mutable

Unshared mutable data needs no synchronisation

Shared mutable data needs synchronisation

Unshared

Shared

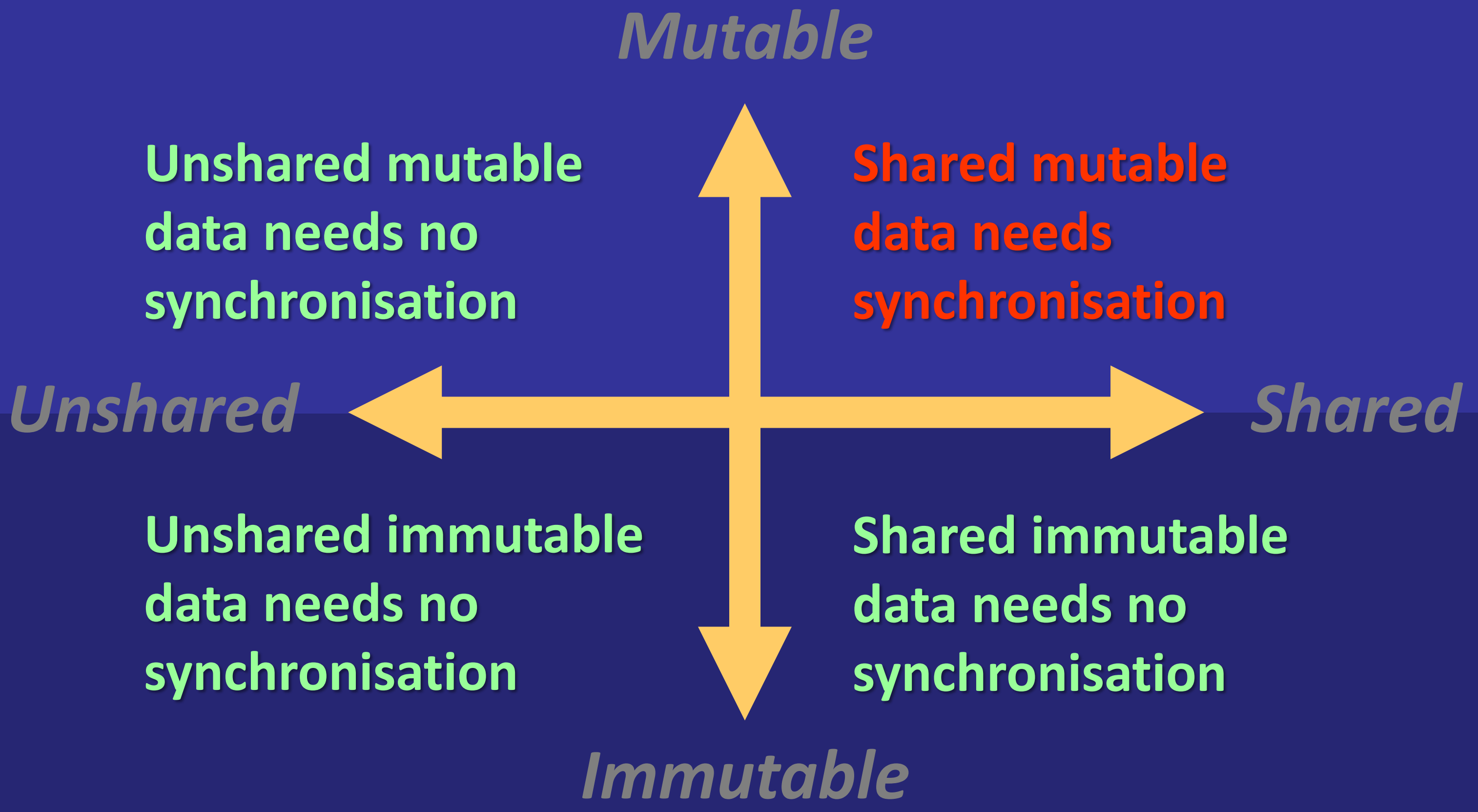
Unshared immutable data needs no synchronisation

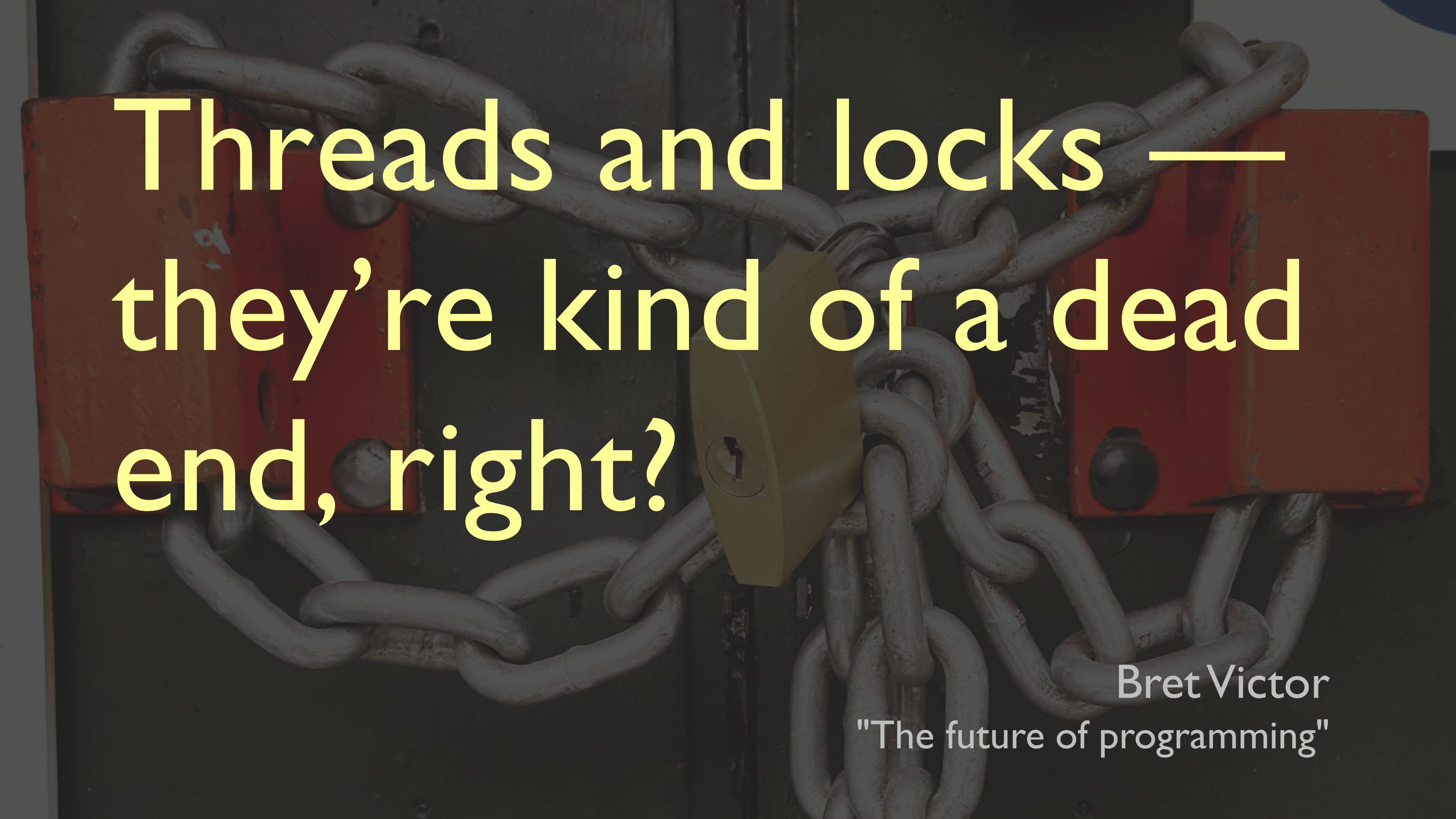
Shared immutable data needs no synchronisation

Immutable

Procedural Comfort Zone

Procedural Discomfort Zone




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, we should just, like, pack up and go home, 'cause we've clearly failed as an engineering field.

Bret Victor
"The future of programming"

Procedural Comfort Zone

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 computation model allows programmers to build a single computational activity: a single-threaded, step-at-a-time computation.

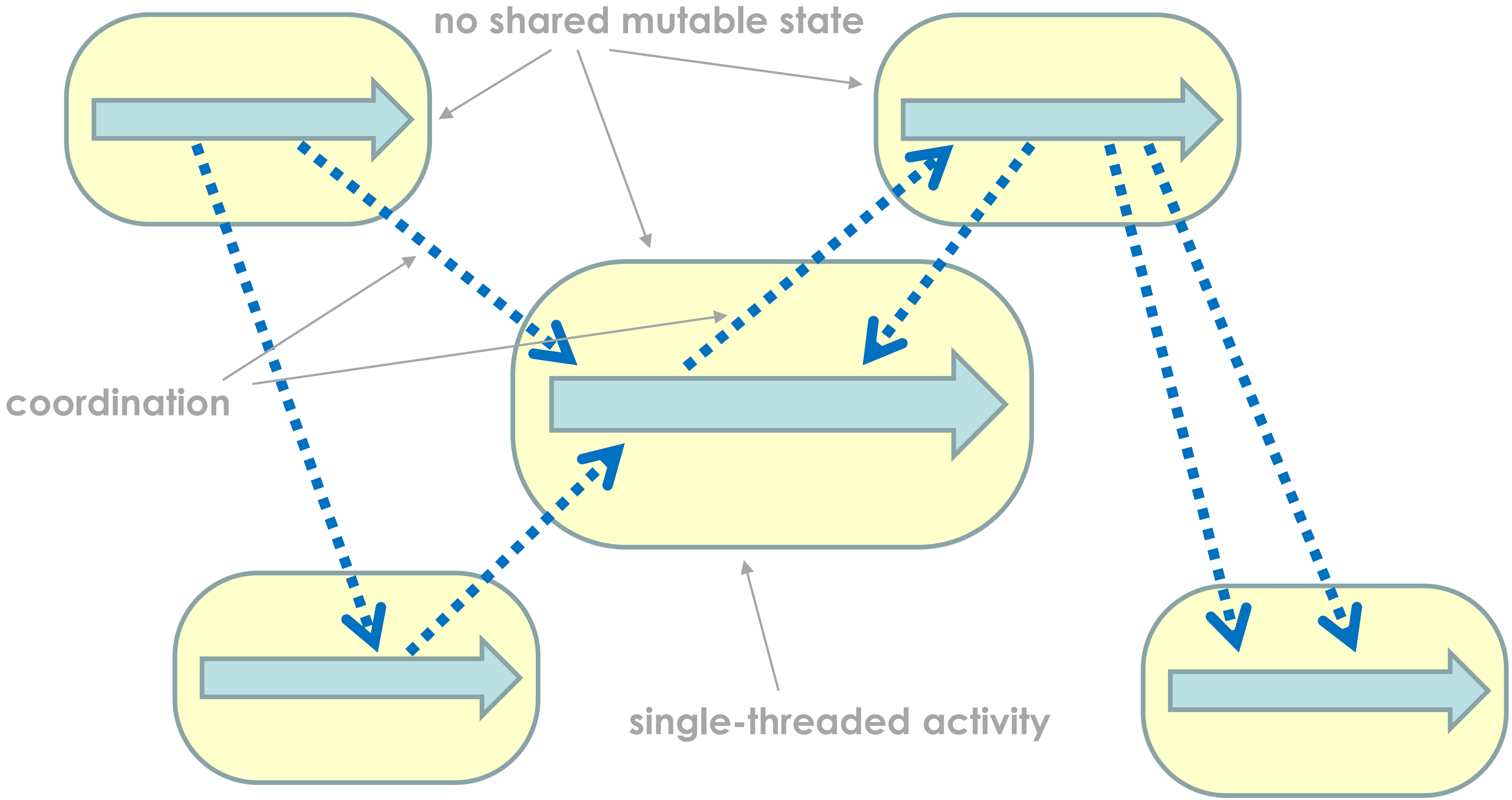
David Gelernter + Nicholas Carriero

"Coordination Languages and their Significance"

The coordination model is the glue that binds separate activities into an ensemble.

David Gelernter + Nicholas Carriero

"Coordination Languages and their Significance"


no shared mutable state
coordination
single-threaded activity

Summary--what's most important.

To put my strongest concerns in a nutshell:

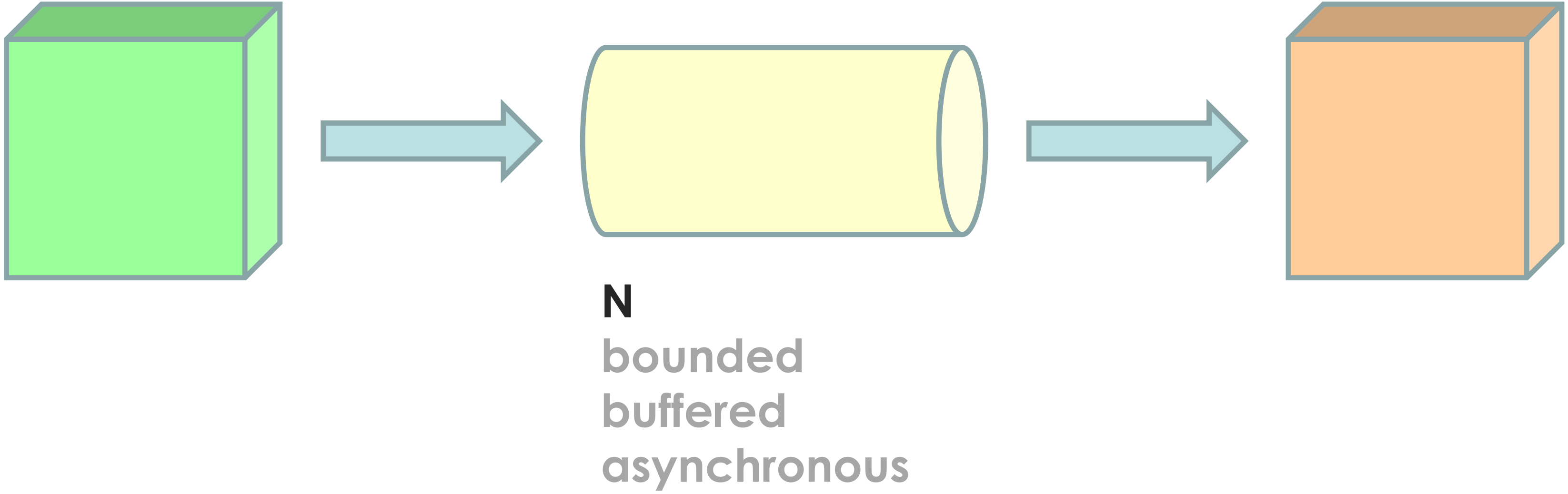
1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes when it becomes necessary to massage data in another way. This is the way of IO also.

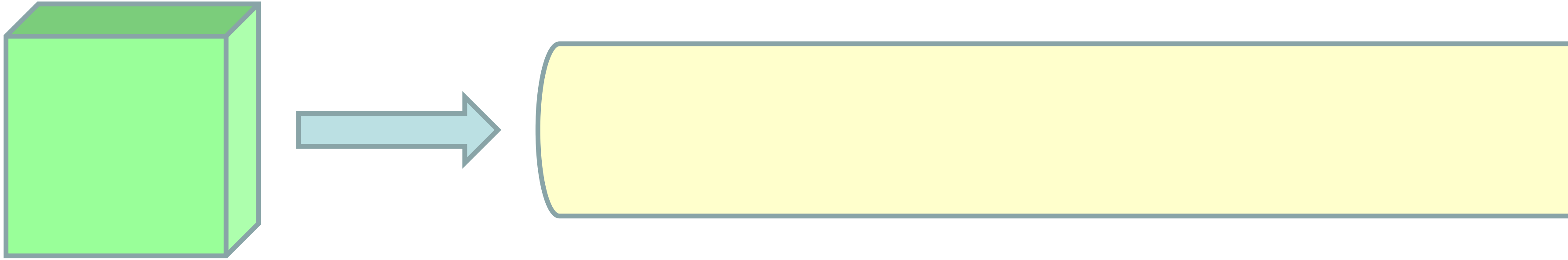
2. Our loader should be able to do link-loading and controlled establishment.

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 sytem components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964


N
bounded
buffered
asynchronous

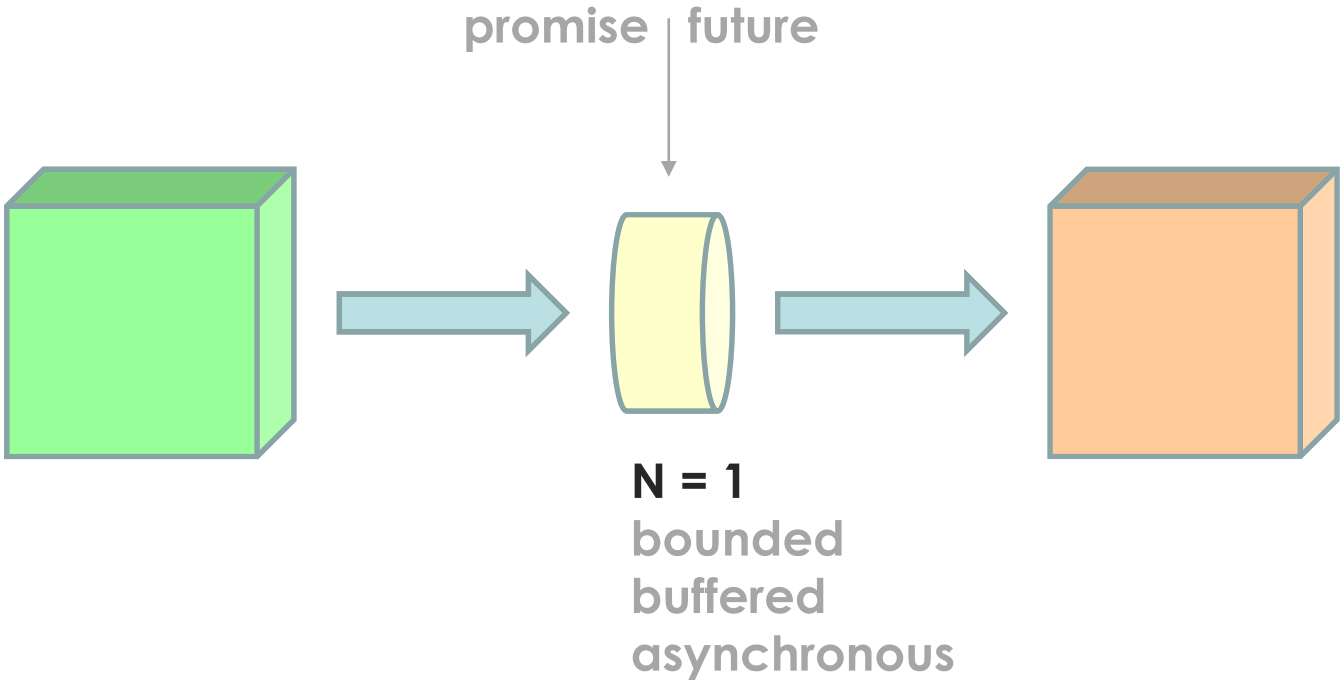


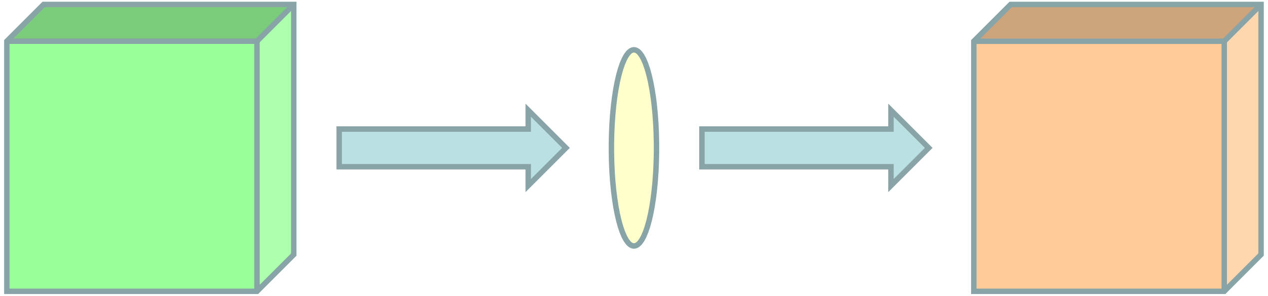
$N = \infty$

unbounded

buffered

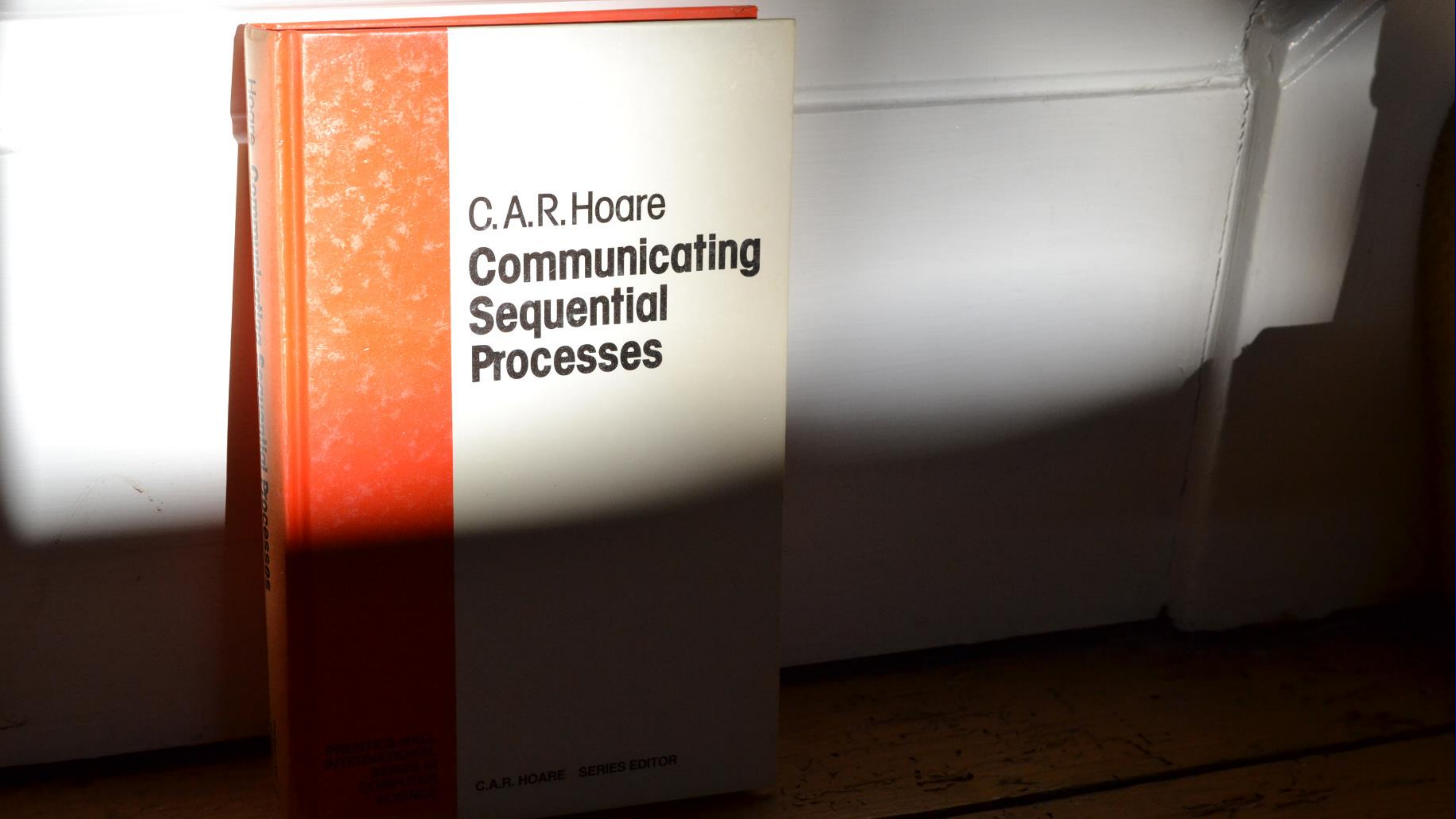
asynchronous

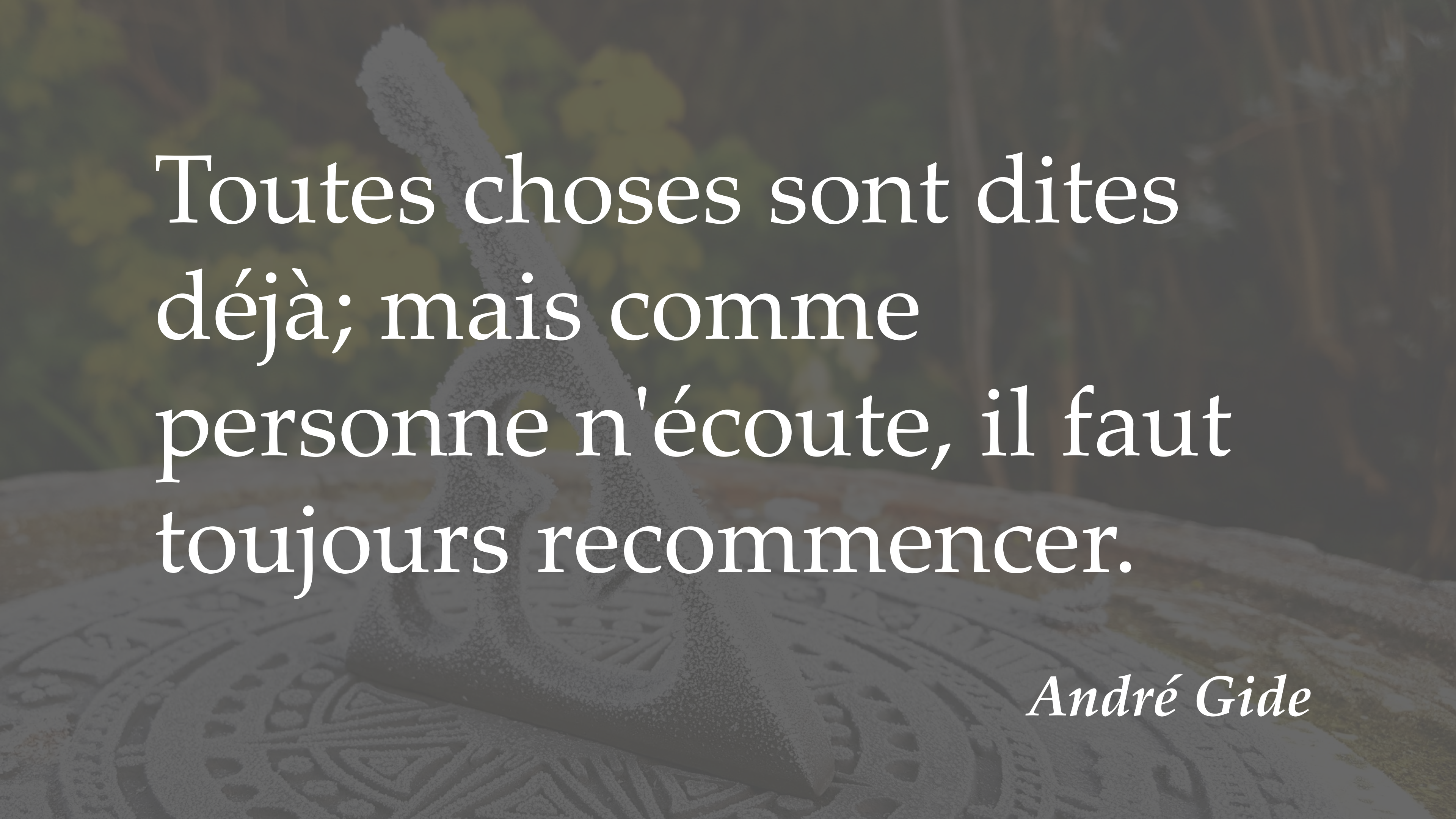

promise
future
N = 1
bounded
buffered
asynchronous

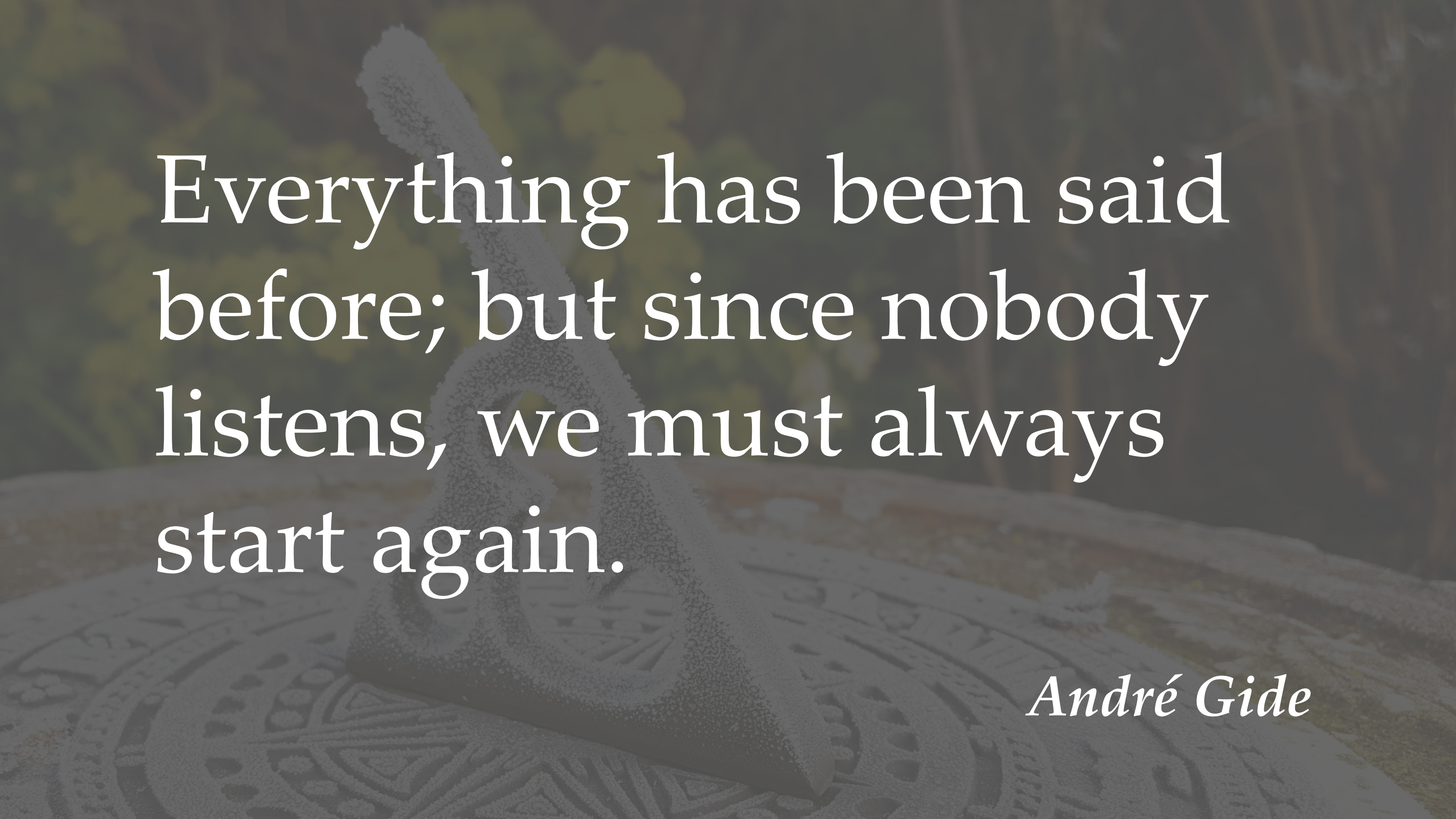


N = 0

bounded
unbuffered
synchronous


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


Toutes choses sont dites déjà; mais comme personne n'écoute, il faut toujours recommencer.
André Gide


Everything has been said before; but since nobody listens, we must always start again.
André Gide