Niklaus Wirth 80th-Birthday Symposium

ETH Zürich
20 February 2014

Stepwise refinement: From common sense to common practice

Carroll Morgan

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

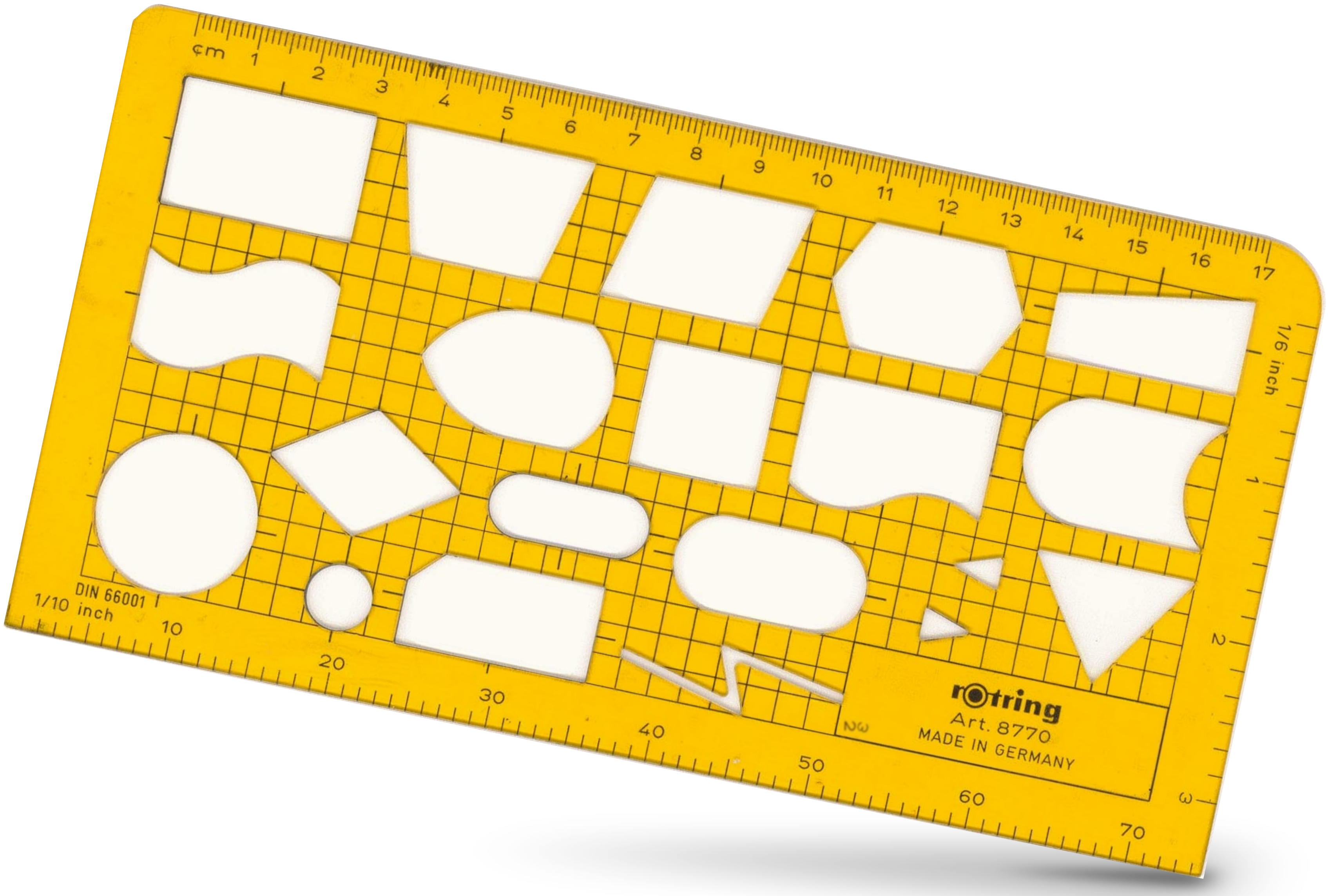
Disclaimer

Ⓜ Recommended for mature audiences

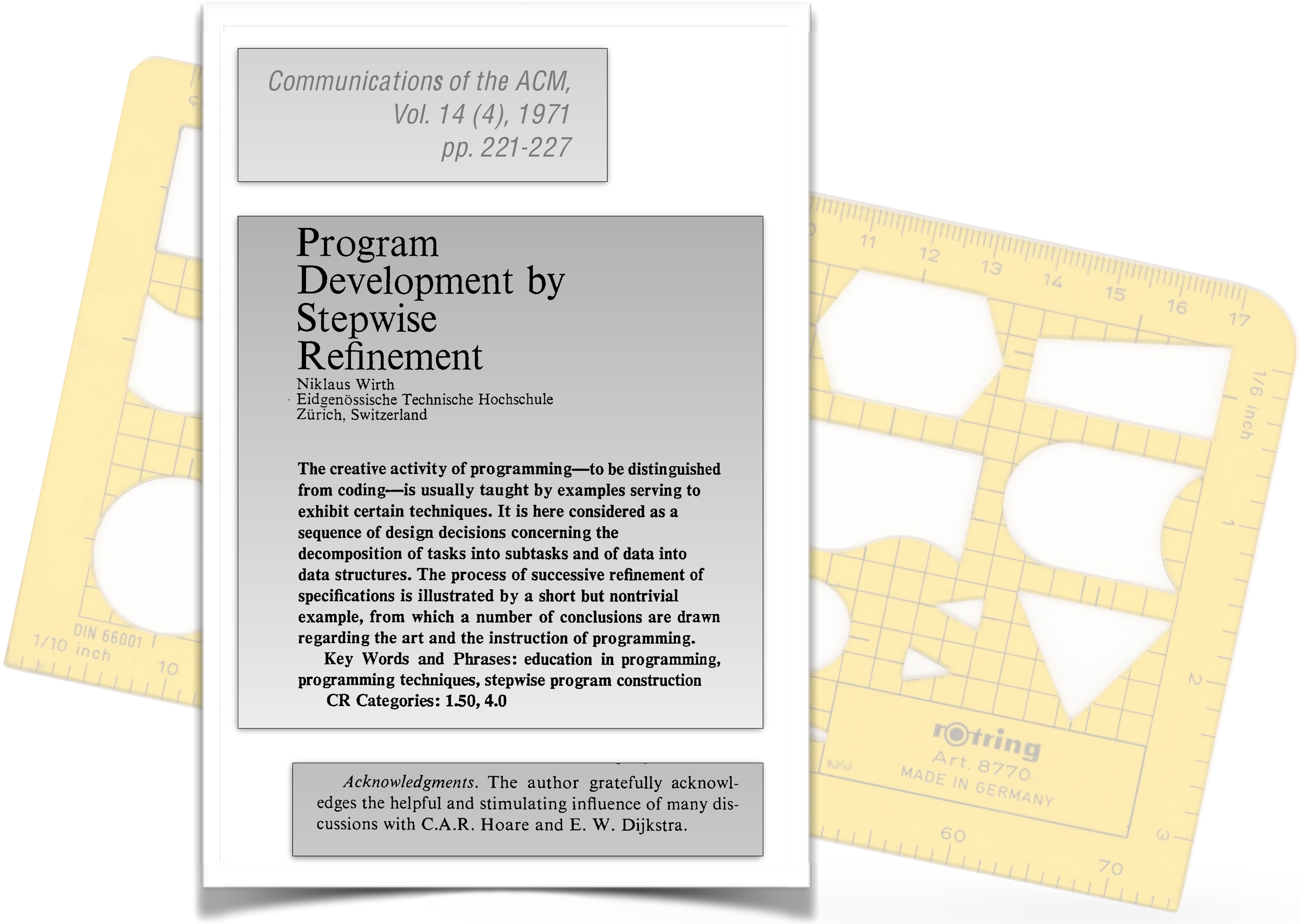
Although this talk is about Stepwise Refinement, and the contribution it makes to software engineering, it is not intended to suggest that Stepwise Refinement, alone, is the ideal.

My personal view is that every person programs in his or her own way, and that we (as teachers and as researchers) can contribute to that by providing a selection of conceptual tools, from which people can choose and with which they can improve their understanding and intellectual control of their own programs however they might have been constructed.

At the beginning of the 1970's



Then, in 1971… Stepwise Refinement



Communications of the ACM,
Vol. 14 (4), 1971
pp. 221-227

Program Development by Stepwise Refinement

Niklaus Wirth
Eidgenössische Technische Hochschule
Zürich, Switzerland

The creative activity of programming—to be distinguished from coding—is usually taught by examples serving to exhibit certain techniques. It is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures. The process of successive refinement of specifications is illustrated by a short but nontrivial example, from which a number of conclusions are drawn regarding the art and the instruction of programming.

Key Words and Phrases: education in programming, programming techniques, stepwise program construction
CR Categories: 1.50, 4.0

Acknowledgments. The author gratefully acknowledges the helpful and stimulating influence of many discussions with C.A.R. Hoare and E. W. Dijkstra.

Improvement of pieces leads to improvement of the whole

The 8 Queens Problem

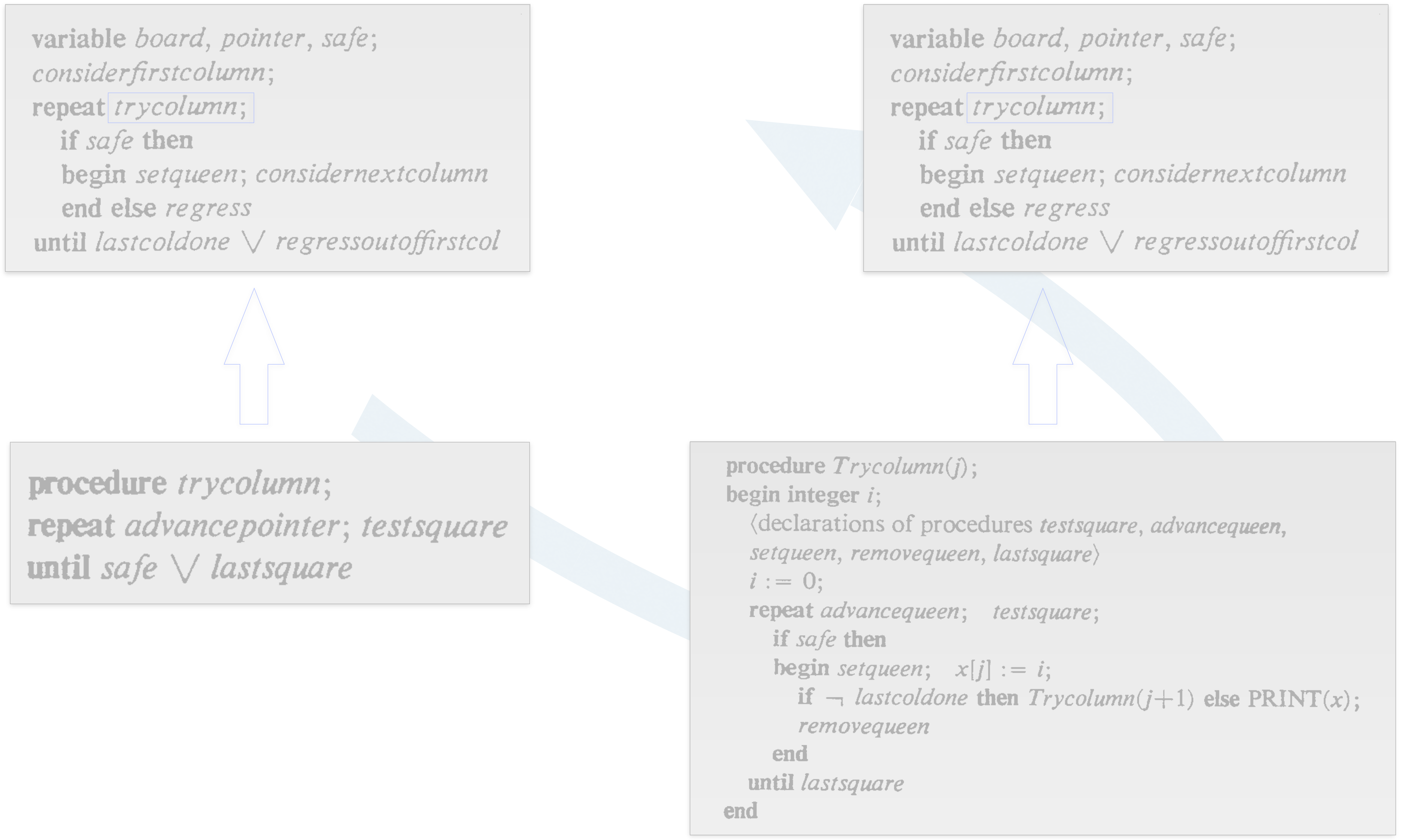
```
variable board, pointer, safe;
considerfirstcolumn;
repeat trycolumn;
   if safe then
   begin setqueen; considernextcolumn
   end else regress
until lastcoldone ∨ regressoutoffirstcol
```

```
procedure trycolumn;
repeat advancepointer; testsquare
until safe ∨ lastsquare
```

```
variable board, pointer, safe;
considerfirstcolumn;
repeat trycolumn;
   if safe then
   begin setqueen; considernextcolumn
   end else regress
until lastcoldone ∨ regressoutoffirstcol
```

```
procedure Trycolumn(j);
begin integer i;
   ⟨declarations of procedures testsquare, advancequeen,
   setqueen, removequeen, lastsquare⟩
   i := 0;
   repeat advancequeen;   testsquare;
      if safe then
      begin setqueen;   x[j] := i;
         if ¬ lastcoldone then Trycolumn(j+1) else PRINT(x);
         removequeen
      end
   until lastsquare
end
```

Many developments followed during that decade



```
variable board, pointer, safe;
considerfirstcolumn;
repeat trycolumn;
   if safe then
   begin setqueen; considernextcolumn
   end else regress
until lastcoldone $\vee$ regressoutoffirstcol
```

```
variable board, pointer, safe;
considerfirstcolumn;
repeat trycolumn;
   if safe then
   begin setqueen; considernextcolumn
   end else regress
until lastcoldone $\vee$ regressoutoffirstcol
```

```
procedure trycolumn;
repeat advancepointer; testsquare
until safe $\vee$ lastsquare
```

```
procedure Trycolumn(j);
begin integer i;
   ⟨declarations of procedures testsquare, advancequeen,
   setqueen, removequeen, lastsquare⟩
   $i := 0$;
   repeat advancequeen;   testsquare;
      if safe then
      begin setqueen;   $x[j] := i$;
         if $\neg$ lastcoldone then Trycolumn($j+1$) else PRINT($x$);
         removequeen
      end
   until lastsquare
end
```

By 1980, two important ideas had emerged.

non-determinism
as a specification tool

e.g. Abrial
Dijkstra
Jones...

"Design your program as a member of a family of programs that it might have been."

(stepwise) refinement
as a mathematical relation

e.g. Back
Hoare
Meertens...

"...the open-endedness of stepwise refinement has been achieved by introducing correctness of refinement as a binary relation..."

... *and many others.*

By 1980, two important ideas had emerged.

non-determinism
as a specification tool

e.g. Abrial
Dijkstra
Jones...

"Design your program as a member of a family of programs that it might have been."

(stepwise) refinement
as a mathematical relation

e.g. Back
Hoare
Meertens...

"...the open-endedness of stepwise refinement has been achieved by introducing correctness of refinement as a binary relation..."

... *and many others.*

Principles of the refinement relation

Legislative: How do you determine what the rules of refinement should be?

Judicial: How do you determine whether those rules have been followed?

legislative discussed later

Principles of refinement: judicial

The refinement relation is determined by a vocabulary of (un)desirable observations, the *terms of reference*.[1]

A specification *S* is refined by implementation *I* just when every desirable observation that can be made of *S* can be made of *I* also.

The terms of reference are determined by social, legal and political concerns.

This is *not* mathematics.

[1] Think of a judicial inquiry into whether implementation *I* meets its specification *S*.

Principles of refinement: a cynical view[1]

A software business has as its primary goal to make money for its owners. In order to do this (a corollary thus) it must strive

1. to keep its customers happy, and
2. to stay out of court.

If the business does end up in court, it strives

3. to win the case.

How can the business hope to check every possible observation within the *ToR* that the court will apply (and over which ToR it has no control)?

[1] We are not however considering businesses that charge a fee to fix their own mistakes.

Principles of refinement: common sense

Use software-development practices that are *guaranteed* to preserve all the desirable properties.

It's common sense that this cannot be done by enumeration of those properties: indeed there might be infinitely many of them.

Instead you describe the properties, and ensure your practices preserve every property so-described.

This *is* mathematics.[1]

[1] You need a mathematician to tell you how to do this; but you do not need to be a mathematician in order to follow her advice. You don't need to be an aeronautical engineer to fly an Airbus: a pilot's (considerable) skills are of another kind.

look at it the other way 'round

Principles of refinement: seen negatively

The refinement relation is determined by a vocabulary of undesirable observations…

A specification *S* is shown *not* to be refined by kludge *K* just when an actual undesirable observation of *K* has occurred in operation…[1]

…and it can be proved that such an observation could never be made of *S*.[2]

Any (legal) means whatsoever can be used to determine which tests might cause undesirable behaviour. But they must actually be carried out.

[1] Use a *forensic specialist* for this, an expert witness.
[2] Use a *mathematician* for this, also an expert witness but of a different kind.

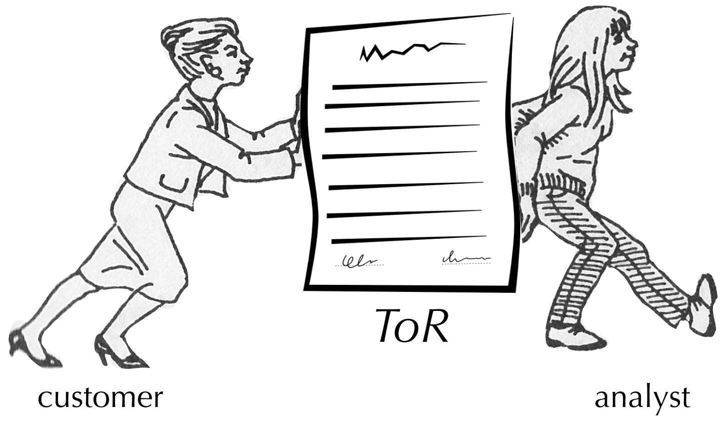
Judicial vs. legislative

Above were opinions on *achieving* refinement and, on the other hand, on *refuting* it "in court."

But how are the *refinement-defining* terms of reference determined in the first place?

There are two pressures, acting from either side: the software business wants ToR that are cheap to achieve; the customer wants ToR that protect her from disappointment.

Usually these do not agree.

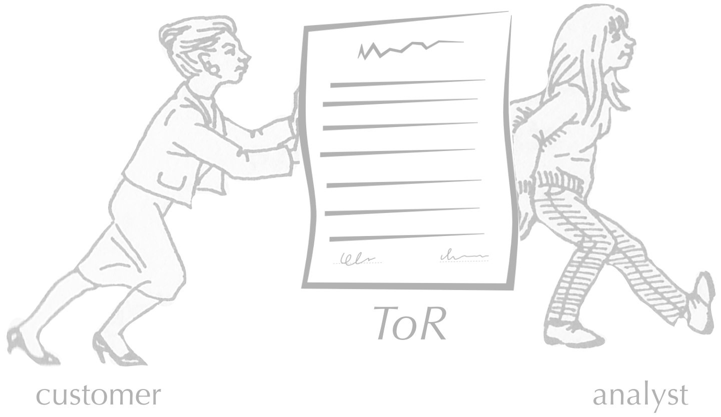


Business ToR vs. client ToR

One might think that the business would strive for weaker ToR (cheaper to achieve), and the customer for stronger (more protection). But actually this is not so. Weaker ToR's are not necessarily easier to achieve, at least for large projects.

A business using *stepwise refinement* will need ToR's that can be managed during that process, that can be achieved piecewise and then maintained.

Usually these do not agree.



ToR's that were too weak: two examples

traces

1980: from Milner's CCS (actually a ToR for equality)

An intuitively appealing definition of *observational equivalence* of concurrent processes is

$P \approx Q$ just when for all traces tr
$P \overset{tr}{\Rightarrow} P'$ implies $Q \overset{tr}{\Rightarrow} Q'$ for some $Q' \approx P'$,
and vice versa.

But it does *not* follow that $\mathcal{C}(P) \approx \mathcal{C}(Q)$, if the context $\mathcal{C}$ contains external choice.

Milner fixed this.

ToR's that were too weak: two examples

Hoare triples

1983: related to Kozen's PPDL (ToR for refinement)

An intuitively appealing definition of *refinement* of probabilistic programs might be

$P \sqsubseteq Q$ just when for all preconditions *pre* and postconditions *post*

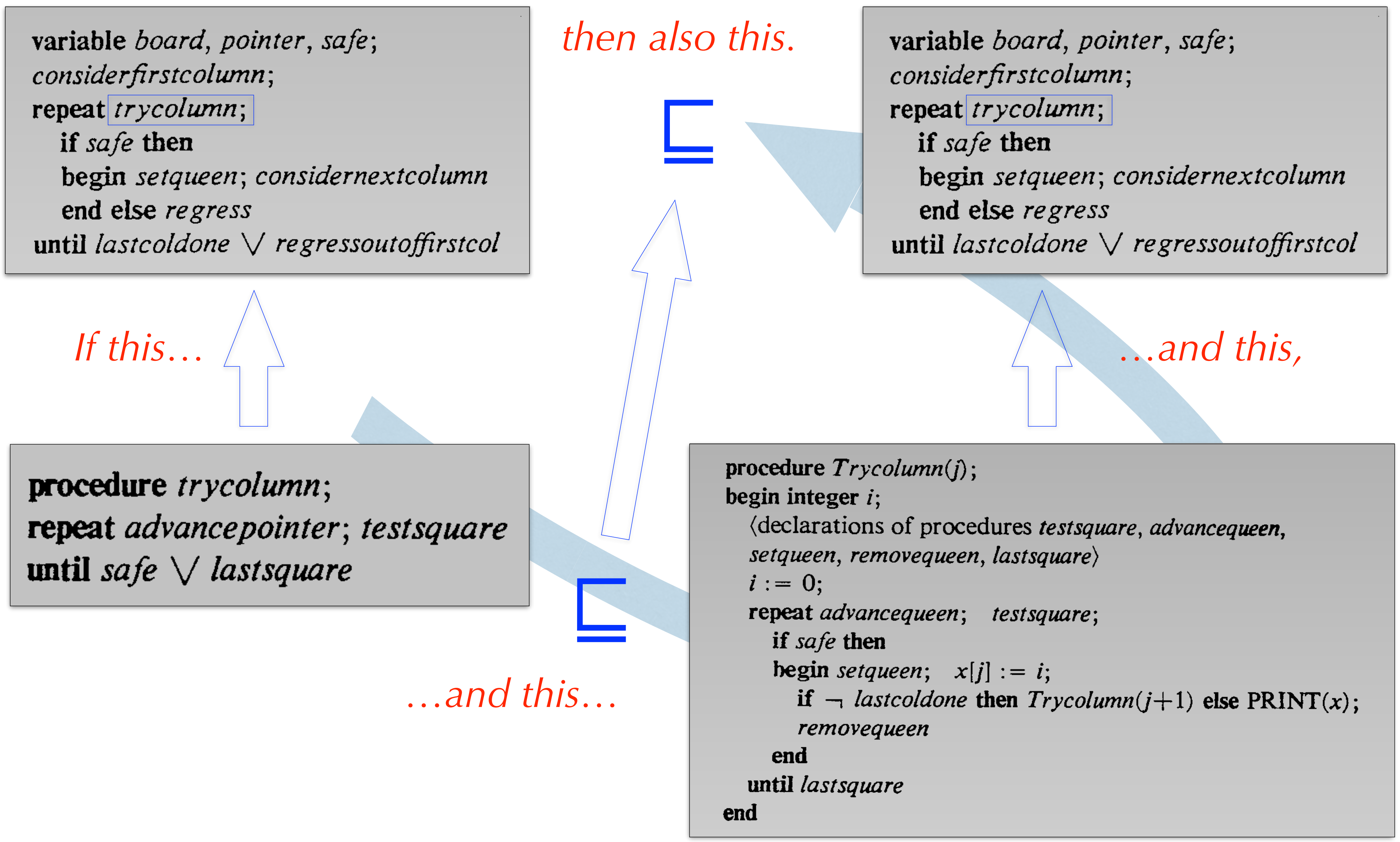
$$\begin{array}{rl} & \text{the probability that } \{pre\}\ P\ \{post\} \\ \leq & \text{the probability that } \{pre\}\ Q\ \{post\}. \end{array}$$

But it does *not* follow that $\mathcal{C}(P) \sqsubseteq \mathcal{C}(Q)$, if the context $\mathcal{C}$ contains nondeterministic choice.

Kozen side-stepped this.

All this is important because…

Stepwise refinement doesn't work otherwise.



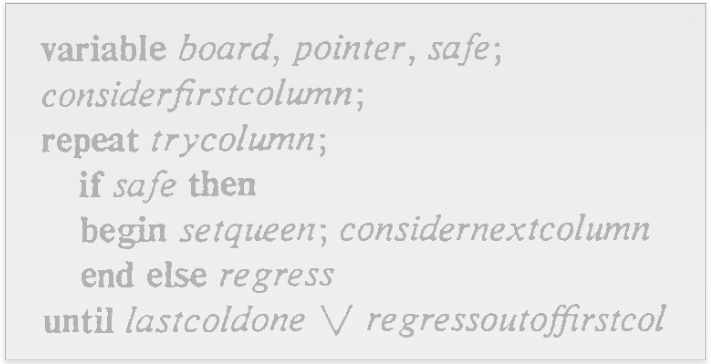
```
variable board, pointer, safe;
considerfirstcolumn;
repeat trycolumn;
   if safe then
   begin setqueen; considernextcolumn
   end else regress
until lastcoldone ∨ regressoutoffirstcol
```

```
procedure trycolumn;
repeat advancepointer; testsquare
until safe ∨ lastsquare
```

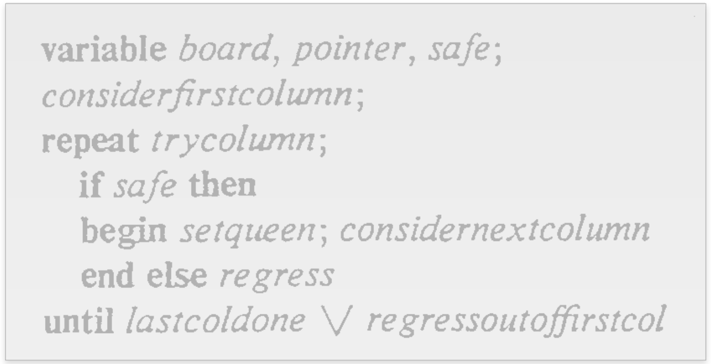
```
variable board, pointer, safe;
considerfirstcolumn;
repeat trycolumn;
   if safe then
   begin setqueen; considernextcolumn
   end else regress
until lastcoldone ∨ regressoutoffirstcol
```

```
procedure Trycolumn(j);
begin integer i;
   ⟨declarations of procedures testsquare, advancequeen,
   setqueen, removequeen, lastsquare⟩
   i := 0;
   repeat advancequeen;   testsquare;
      if safe then
      begin setqueen;   x[j] := i;
         if ¬ lastcoldone then Trycolumn(j+1) else PRINT(x);
         removequeen
      end
   until lastsquare
end
```

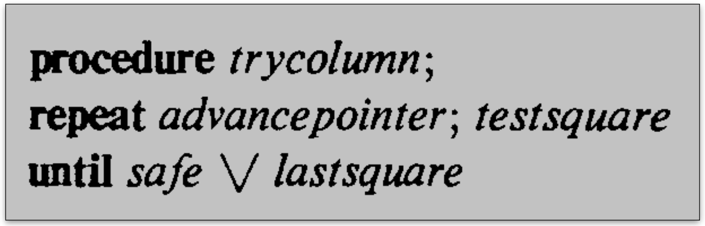
It should be possible to do this, in isolation



```
variable board, pointer, safe;
considerfirstcolumn;
repeat trycolumn;
   if safe then
   begin setqueen; considernextcolumn
   end else regress
until lastcoldone ∨ regressoutoffirstcol
```

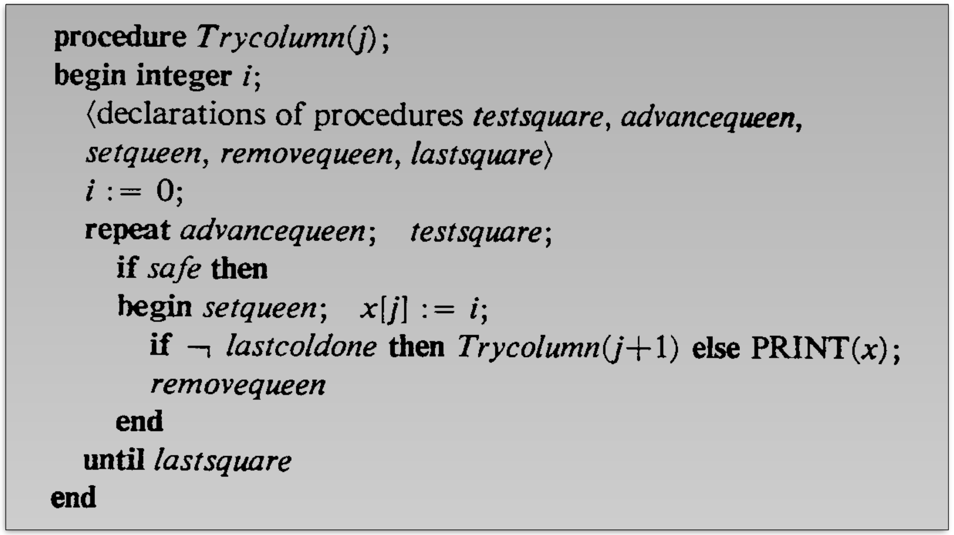


```
variable board, pointer, safe;
considerfirstcolumn;
repeat trycolumn;
   if safe then
   begin setqueen; considernextcolumn
   end else regress
until lastcoldone ∨ regressoutoffirstcol
```



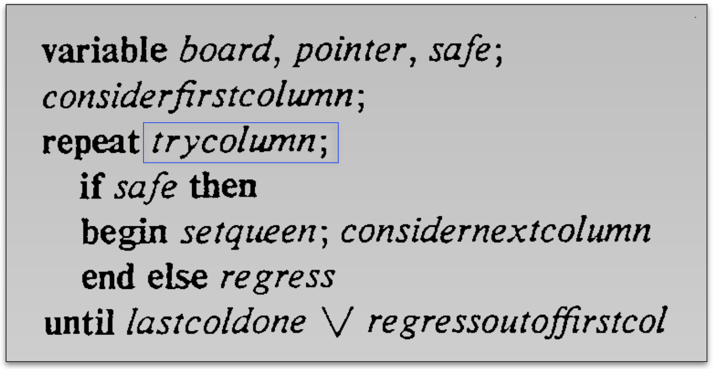
```
procedure trycolumn;
repeat advancepointer; testsquare
until safe ∨ lastsquare
```

$\sqsubseteq$



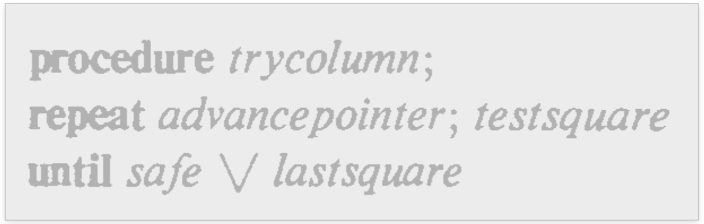
```
procedure Trycolumn(j);
begin integer i;
   ⟨declarations of procedures testsquare, advancequeen,
   setqueen, removequeen, lastsquare⟩
   i := 0;
   repeat advancequeen;   testsquare;
      if safe then
      begin setqueen;   x[j] := i;
         if ¬ lastcoldone then Trycolumn(j+1) else PRINT(x);
         removequeen
      end
   until lastsquare
end
```

…without knowing about this.

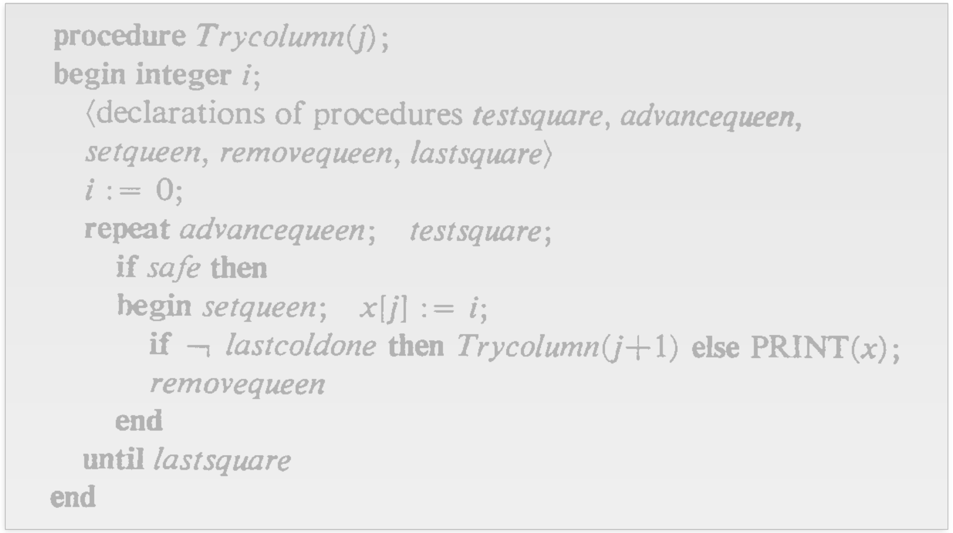


```
variable board, pointer, safe;
considerfirstcolumn;
repeat trycolumn;
   if safe then
   begin setqueen; considernextcolumn
   end else regress
until lastcoldone $\vee$ regressoutoffirstcol
```

It's just common sense:
and the mathematics in
the background is only
a means to this end.



```
procedure trycolumn;
repeat advancepointer; testsquare
until safe $\vee$ lastsquare
```



```
procedure Trycolumn(j);
begin integer i;
   ⟨declarations of procedures testsquare, advancequeen,
   setqueen, removequeen, lastsquare⟩
   $i := 0$;
   repeat advancequeen;   testsquare;
      if safe then
      begin setqueen;   $x[j] := i$;
         if $\neg$ lastcoldone then Trycolumn($j+1$) else PRINT($x$);
         removequeen
      end
   until lastsquare
end
```

Compositional closure: a *legislative* technique

"is blatantly violated by"

Determine by public consultation a relation $\not\preccurlyeq$ so that "everyone agrees" that if $S \not\preccurlyeq K$ then K cannot possibly be considered by any reasonable person to be an implementation of S — for example

- *S always terminates, but K can get into an infinite loop, or* ------ *sequential*
- *S will never do trace tr, but K might or* ------ *concurrent*
- *K is more likely to abort than S is.* ------ *probabilistic*

The (complementary) relation $\preccurlyeq$ does *not* have to be preserved by context.

1983: de Nicola, Hennessy

Compositional closure: a *legislative* technique

Define $P \sqsubseteq Q$ so that

If $P \sqsubseteq Q$ then $\mathcal{C}(P) \preccurlyeq \mathcal{C}(Q)$ for all contexts $\mathcal{C}$ and

If $P \not\sqsubseteq Q$ then $\mathcal{C}(P) \not\preccurlyeq \mathcal{C}(Q)$ for some context $\mathcal{C}$.

That is, from $\preccurlyeq$ and a description of all $\mathcal{C}$'s the mathematicians find a $\sqsubseteq$ that the developers can use so that

If they *do* use it, their customers will be happy in all contexts; and, Not too weak.

If they *don't* use it, there is a context in which a customer will be unhappy. Not too strong.

Just right:
and unique.

Compositional closure: a *legislative* technique

Define $P \sqsubseteq Q$ so that

If $P \sqsubseteq Q$ then $\mathcal{C}(P) \preccurlyeq \mathcal{C}(Q)$ for all contexts $\mathcal{C}$ and

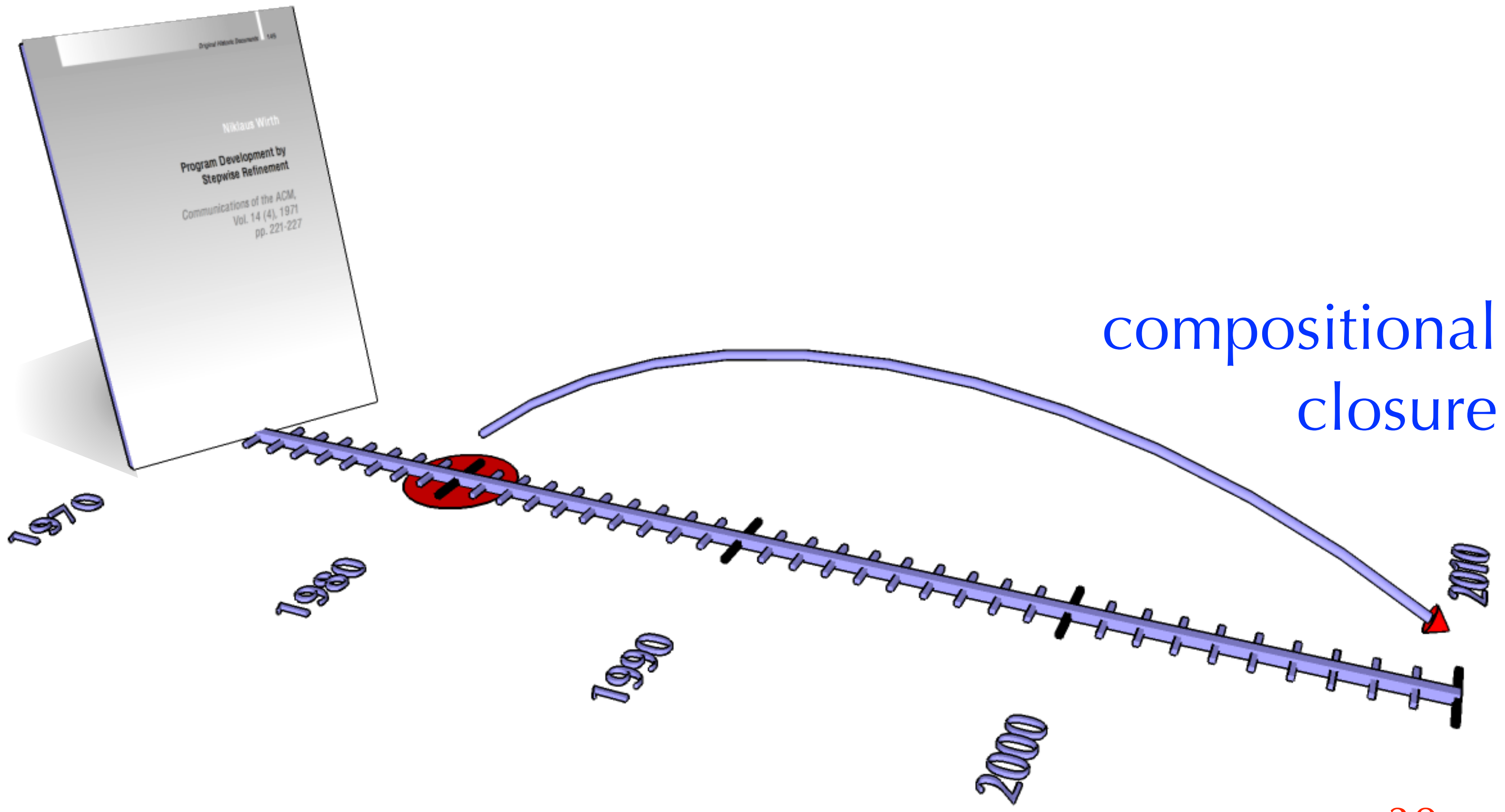
If $P \not\sqsubseteq Q$ then $\mathcal{C}(P) \not\preccurlyeq \mathcal{C}(Q)$ for some context $\mathcal{C}$.

That is, from $\preccurlyeq$ and a description of all $\mathcal{C}$'s the mathematicians find a $\sqsubseteq$ that the developers can use so that

Follow the refinement rules, and your implementation is safe in all contexts.

Break the rules, and there is a context is which your implementation breaks. *Guaranteed*.

Fast-forward to 2010


Original Historic Documents 145

Niklaus Wirth

Program Development by Stepwise Refinement

Communications of the ACM, Vol. 14 (4), 1971 pp. 221-227

compositional closure

30 years later

What should the ToR be for security?

2010: ***No consensus***.

- The chance of guessing the secret in one try must not increase (Rényi min-entropy)?
- The Shannon-entropy of the secret must not decrease?
- The average number of incorrect guesses must not decrease?
- The number of guesses needed to have 50% chance of being correct should not decrease?
- …

This is security in the broad sense of “keeping data secret,” including e.g. noninterference, and is more general than merely cryptography.

Compositional closure of Rényi

Use this one as the starting point $\preccurlyeq$,

- The chance of guessing the secret in one try must not increase (Rényi min-entropy),

since it's agreed by the "public" to be reasonable; apply compositional closure to *synthesise* from it a refinement order $\sqsubseteq$ for quantitative information flow.

The mathematicians do the synthesis from $\mathscr{C}$; the business never sees that synthesis. The business applies the order $\sqsubseteq$; the customer never sees it being applied.

The customer ultimately is happy because of that order she never sees, because the $\preccurlyeq$ she *does* see is achieved.

Meanwhile, on the top half of the world and independently

Generalise the same order $\preccurlyeq$

- The chance of guessing the secret in one try must not increase (Rényi min-entropy),

in a way suggested by Landauer and Redmond's Lattice of Information,[1] then formulate and investigate the so-called Coriaceous Conjecture.[2]

The Paris procedure was different (from Sydney's), not guaranteed to produce a unique definition. And yet…

They obtained **exactly the same order** as had been found in the southern hemisphere by unique synthesis.

[1] From 1993: the generalisation of *Lattice of Inf.* was thus a 20-year-old problem.
[2] Try Googling the *Coriaceous Conjecture*.

A triumph of *common sense*

Principles that were articulated so clearly, *so long ago*, are still guiding researchers today.

Why are they not guiding practitioners?

Why are they not *common practice*?

Why do I claim they are not?

Most undergraduates "these days"…

- Can't do static reasoning — actually, they
- Don't even know what "static reasoning" means.
- Don't understand *abstraction* as a concept (even if they practise it accidentally).
- Don't know what an invariant is.

and yet…

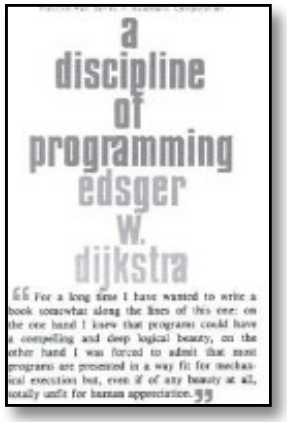
- are *ecstatic* if taught these ideas taught informally — thus demonstrating (1) that indeed they did not know them before, and (2) that they can learn them now.[1]

[1] Google *(In-)formal methods: the lost art.*

The *XXX* of Programming

It's more than a "lost art" — it's a lost *opportunity!*

To a long line of ground-breaking texts,


a discipline of programming edsger w. dijkstra

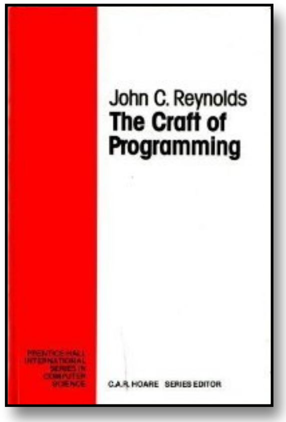
"For a long time I have wanted to write a book somewhat along the lines of this one: on the one hand I knew that programs could have a compelling and deep logical beauty, on the other hand I was forced to admit that most programs are presented in a way fit for mechanical execution but, even if of any beauty at all, totally unfit for human appreciation."

TEXTS AND MONOGRAPHS IN COMPUTER SCIENCE

THE SCIENCE OF PROGRAMMING

David Gries

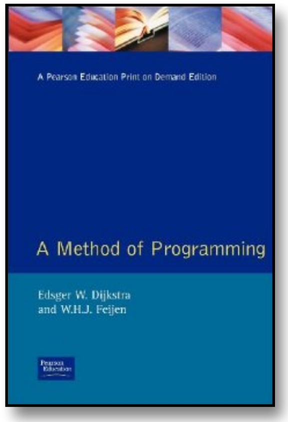
Springer


John C. Reynolds

The Craft of Programming

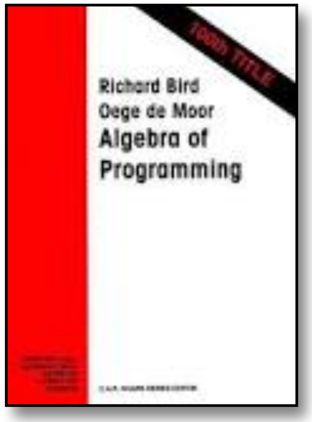
PRENTICE-HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE

C.A.R. HOARE SERIES EDITOR


A Pearson Education Print on Demand Edition

A Method of Programming

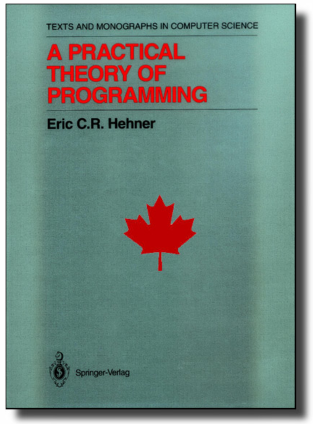
Edsger W. Dijkstra and W.H.J. Feijen


Richard Bird

Oege de Moor

Algebra of Programming

C.A.R. HOARE SERIES EDITOR


TEXTS AND MONOGRAPHS IN COMPUTER SCIENCE

A PRACTICAL THEORY OF PROGRAMMING

Eric C.R. Hehner

Springer-Verlag

add one more…

The Pain of Programming

Give the students exercises that they must struggle to complete, and whose solutions will not satisfy them. Make them suffer.

Only then give them the conceptual tools to do the same job with pleasure, elegance and satisfaction.

These tools are not theories, formulae and proofs: they are *ways of thinking*; and they can be taught initially with *natural language, pictures and informal reasoning*.

Only if you put them in a position where they <u>ask</u> for conceptual help with their programming, will they <u>appreciate</u> what you give them.

From common sense to common practice: the importance of *who* does *what*

Mathematicians synthesise refinement relations, and discover software-development theories; but they (as a rule) don’t have to apply them.

Programmers use those theories to build systems; but they don’t have to create the theories themselves.

Clients use the systems that programmers create; but they don’t have to know how they’re built.

From common sense to common practice: the importance of *who* does *what*

Mathematicians discover theories.

Programmers use theories.

Customers benefit from theories.

Common practice? We can still succeed.

Even though it has been 43 years since Niklaus Wirth's paper (and the many other influential papers from that time), it is not too late to make sure that *everyone* gets the benefit that they convey: mathematicians, programmers, customers.

Each role has different needs: don't confuse them.

Condition our students in particular, who later become our practitioners, to *want* to use we we know they need… and only then show them how.

They will appreciate it — in the end.

Kids "these days" think they know everything.

It's not easy…

When I was a boy of fourteen, my father was so ignorant I could hardly stand to have the old man around.

But when I got to be twenty-one, I was astonished at how much he had learned in seven years.

Mark Twain: c19th American writer.

It’s not easy…

When I started learning about Stepwise Refinement, my teachers were so ignorant that I could hardly stand to go to lectures.

But by the end of the course I was astonished at how much they had learned in those four years.

Mark Twain: c19th American writer, adapted.

And yet it can be done.