FP vs. OOP: case studies in misunderstandings

Panicz Maciej Godek

godek.maciek@gmail.com

Functional Tricity, 27.06.2019

Definition of a paradigm

paradigm – a distinct set of concepts or thought patterns, including theories, research methods, postulates, and standards for what constitutes a legitimate contribution to a field. [Wikipedia]


1:25000
500M
1000M
1500M
2000M
1:24000
Новогербово
Васильевское
пр.Воронцово
пр.Богаево
пр.Крапивня
пр.Горки
пр.Петрово
пр.Белкино

Definition of functional programming

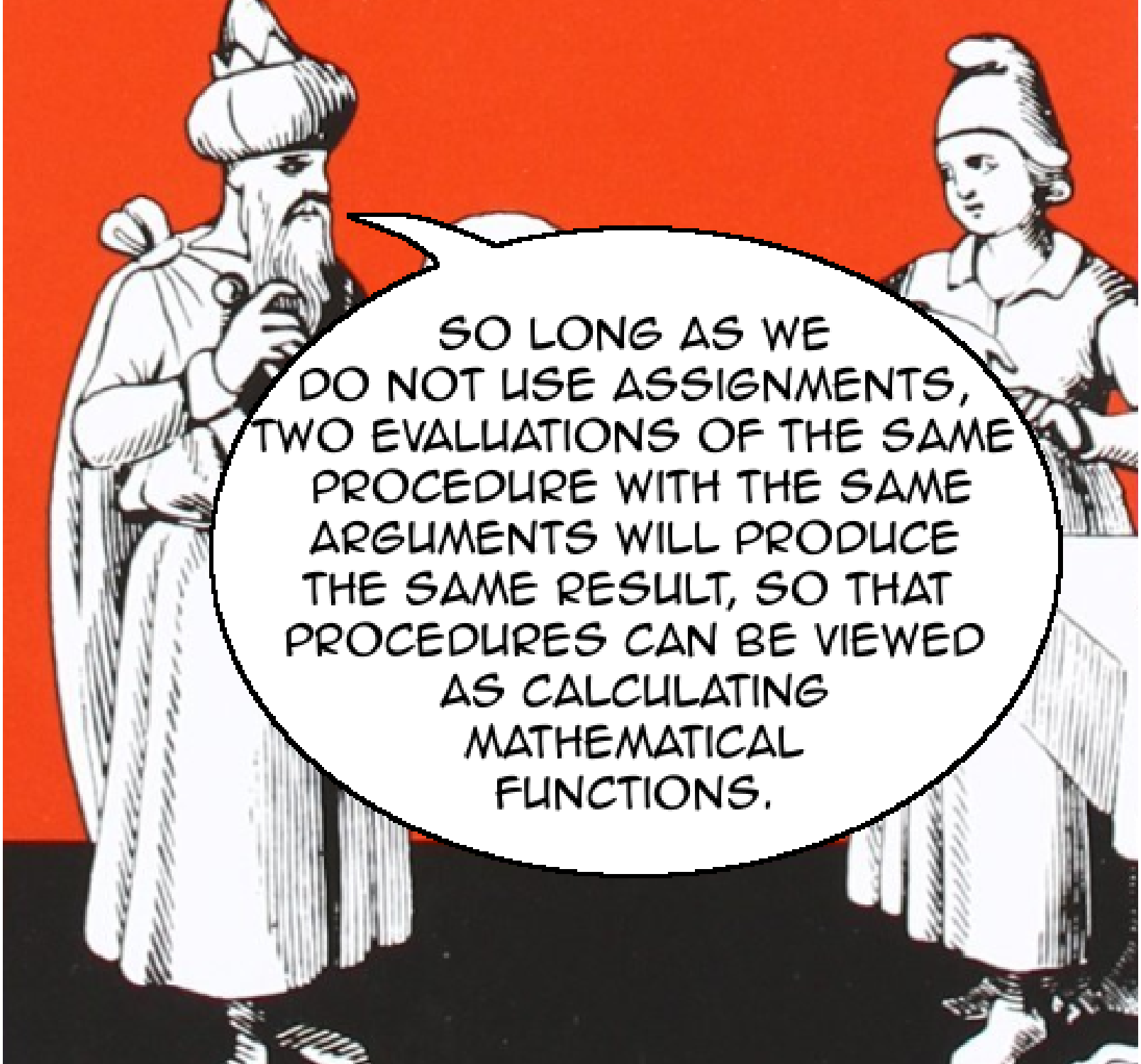
functional programming – a style of programming where a programmer is allowed only to define pure functions, by composing them from other pure functions

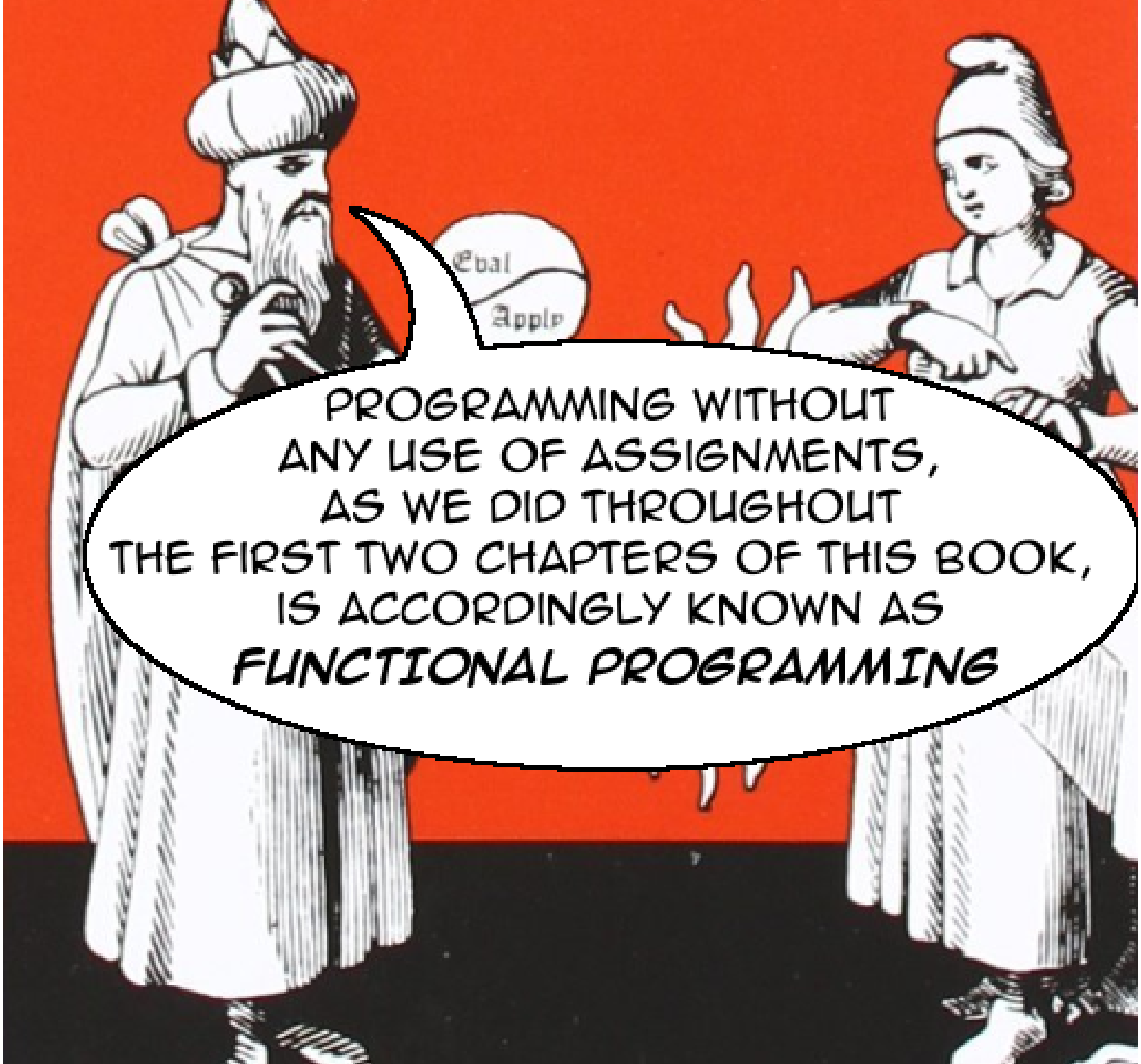

Eval
Apple


AS WE HAVE SEEN, THE SET! OPERATOR ENABLES US TO MODEL OBJECTS THAT HAVE LOCAL STATE.


HOWEVER, THIS ADVANTAGE COMES AT A PRICE.
Eval
Appl


OUR PROGRAMMING LANGUAGE CAN NO LONGER BE INTERPRETED IN TERMS OF THE SUBSTITUTION MODEL OF PROCEDURE APPLICATION [...]


SO LONG AS WE DO NOT USE ASSIGNMENTS, TWO EVALUATIONS OF THE SAME PROCEDURE WITH THE SAME ARGUMENTS WILL PRODUCE THE SAME RESULT, SO THAT PROCEDURES CAN BE VIEWED AS CALCULATING MATHEMATICAL FUNCTIONS.


Eval
Apply
PROGRAMMING WITHOUT ANY USE OF ASSIGNMENTS, AS WE DID THROUGHOUT THE FIRST TWO CHAPTERS OF THIS BOOK, IS ACCORDINGLY KNOWN AS
FUNCTIONAL PROGRAMMING


val
Appl

Referential opacity

```
(define (compile expression result)
 (match expression
   (`(,<*> ,a ,b)
    (let ((a+ (name a))
          (b+ (name b)))
      `(,@(compile a a+)
        ,@(compile b b+)
        (,result = ,a+ ,<*> ,b+))))
   (_
    '())))
```

Referential opacity

```
(define (compile expression result)
 (match expression
   (`(,<*> ,a ,b)
    (let ((a+ (name a))
          (b+ (name b)))
      `(,@(compile a a+)
        ,@(compile b b+)
        (,result = ,a+ ,<*> ,b+))))
   (_
    '())))
```

Referential opacity

```
(define (compile expression result)
 (match expression
   (`(,<*> ,a ,b)
    (let ((a+ (name a))
          (b+ (name b)))
      `(,@(compile a a+)
        ,@(compile b b+)
        (,result = ,a+ ,<*> ,b+))))
   (_
    '())))
```

Referential opacity

```
(define (compile expression result)
 (match expression
   (`(,<*> ,a ,b)
    (let ((a+ (name a))
          (b+ (name b)))
      `(,@(compile a a+)
        ,@(compile b b+)
        (,result = ,a+ ,<*> ,b+))))
   (_
    '())))
```

Referential opacity

```
(define (compile expression result)
 (match expression
   (`(,<*> ,a ,b)
    (let ((a+ (name a))
          (b+ (name b)))
      `(,@(compile a a+)
        ,@(compile b b+)
        (,result = ,a+ ,<*> ,b+))))
   (_
    '())))
```

Referential opacity

```
(define name
 (let ((counter 0))
   (lambda (expression)
     (cond ((symbol?  expression)
            expression)
           (else
            (set!  counter (+ counter 1))
            (symbol-append 't
              (number->symbol counter)))))))
```

Referential opacity

```
(define name
 (let ((counter 0))
   (lambda (expression)
     (cond ((symbol?  expression)
            expression)
           (else
            (set!  counter (+ counter 1))
            (symbol-append 't
              (number->symbol counter)))))))
```

Referential opacity

```
(define name
 (let ((counter 0))
   (lambda (expression)
     (cond ((symbol? expression)
            expression)
           (else
            (set! counter (+ counter 1))
            (symbol-append 't
              (number->symbol counter)))))))
```

Referential opacity

```
(define name
 (let ((counter 0))
   (lambda (expression)
     (cond ((symbol?  expression)
            expression)
           (else
            (set!  counter (+ counter 1))
            (symbol-append 't
              (number->symbol counter)))))))
```

Referential opacity

```
(compile '(* (+ x y) (/ z w)) 'result)
```

Referential opacity

```
(match '(* (+ x y) (/ z w))
 (`(,<*> ,a ,b)
  (let ((a+ (name a))
        (b+ (name b)))
    `(,@(compile a a+)
      ,@(compile b b+)
      (result = ,a+ ,<*> ,b+))))
 (_
  '()))
```

Referential opacity

```
(match '(* (+ x y) (/ z w))
 (`(,<*> ,a ,b)
  (let ((a+ (name a))
        (b+ (name b)))
    `(,@(compile a a+)
      ,@(compile b b+)
      (result = ,a+ ,<*> ,b+))))
 (_
  '()))
```

Referential opacity

```
(let ((a+ (name '(+ x y)))
      (b+ (name '(/ z w))))
  `(,@(compile '(+ x y) a+)
    ,@(compile '(/ z w) b+)
    (result = ,a+ * ,b+)))
```

Referential opacity

```
(let ((a+ (name '(+ x y)))
      (b+ (name '(/ z w))))
  `(,@(compile '(+ x y) a+)
    ,@(compile '(/ z w) b+)
    (result = ,a+ * ,b+)))
```

Referential opacity

```
(let ((a+ (name '(+ x y)))
      (b+ (name '(/ z w))))
  `(,@(compile '(+ x y) a+)
    ,@(compile '(/ z w) b+)
    (result = ,a+ * ,b+)))
```

Referential opacity

```
(let ((a+ 't1)
      (b+ 't2))
  `(,@(compile '(+ x y) a+)
    ,@(compile '(/ z w) b+)
    (result = ,a+ * ,b+)))
```

Referential opacity

```
`(,@(compile '(+ x y) 't1)
  ,@(compile '(/ z w) 't2)
   (result = t1 * t2))
```

Referential opacity

```
`(,@(compile '(+ x y) 't1)
  ,@(compile '(/ z w) 't2)
   (result = t1 * t2))
```

Referential opacity

```
'((t1 = x + y)
  (t2 = z / w)
  (result = t1 * t2))
```

Non-determinism

```
(define (shuffle l)
 (let ((n (length l)))
   (if (is n < 2)
       l
     (let ((left right
                  (split-at l
                     (+ (random (- n 1))
                        1))))
       (if (= (random 2) 1)
           `(,@(shuffle right)
             ,@(shuffle left))
         `(,@(shuffle left)
           ,@(shuffle right)))))))
```

Non-determinism

```
(define (shuffle l)
 (let ((n (length l)))
   (if (is n < 2)
       l
     (let ((left right
                  (split-at l
                     (+ (random (- n 1))
                        1))))
       (if (= (random 2) 1)
           `(,@(shuffle right)
             ,@(shuffle left))
         `(,@(shuffle left)
           ,@(shuffle right)))))))
```

Non-determinism

```
(define (shuffle l)
 (let ((n (length l)))
   (if (is n < 2)
       l
     (let ((left right
                  (split-at l
                     (+ (random (- n 1))
                        1))))
       (if (= (random 2) 1)
           `(,@(shuffle right)
             ,@(shuffle left))
         `(,@(shuffle left)
           ,@(shuffle right)))))))
```

Non-determinism

```
(define (shuffle l)
 (let ((n (length l)))
   (if (is n < 2)
       l
     (let ((left right
                  (split-at l
                     (+ (random (- n 1))
                        1))))
       (if (= (random 2) 1)
           `(,@(shuffle right)
             ,@(shuffle left))
         `(,@(shuffle left)
           ,@(shuffle right)))))))
```

Non-determinism

```
(define (shuffle l)
 (let ((n (length l)))
   (if (is n < 2)
       l
     (let ((left right
                  (split-at l
                     (+ (random (- n 1))
                        1))))
       (if (= (random 2) 1)
           `(,@(shuffle right)
             ,@(shuffle left))
         `(,@(shuffle left)
           ,@(shuffle right)))))))
```

Non-determinism

```
(shuffle '(a b c))
```

Non-determinism

```
(let ((n (length '(a b c))))
 (if (is n < 2)
     '(a b c)
   (let ((left right
              (split-at '(a b c)
                 (+ (random (- n 1))
                    1))))
     (if (= (random 2) 1)
         `(,@(shuffle right)
           ,@(shuffle left))
       `(,@(shuffle left)
         ,@(shuffle right)))))))
```

Non-determinism

```
(let ((n 3))
 (if (is n < 2)
     '(a b c)
   (let ((left right
              (split-at '(a b c)
                 (+ (random (- n 1))
                    1))))
     (if (= (random 2) 1)
         `(,@(shuffle right)
           ,@(shuffle left))
       `(,@(shuffle left)
         ,@(shuffle right))))))
```

Non-determinism

```
(if (is 3 < 2)
    '(a b c)
  (let ((left right
                (split-at '(a b c)
                   (+ (random (- 3 1))
                      1))))
    (if (= (random 2) 1)
        `(,@(shuffle right)
          ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right)))))
```

Non-determinism

```
(let ((left right
           (split-at '(a b c)
              (+ (random 2)
                 1))))
  (if (= (random 2) 1)
      `(,@(shuffle right)
        ,@(shuffle left))
    `(,@(shuffle left)
      ,@(shuffle right))))
```

Non-determinism

```
(let ((left right
           (split-at '(a b c)
              (+ (random 2)
                 1))))
  (if (= (random 2) 1)
      `(,@(shuffle right)
        ,@(shuffle left))
    `(,@(shuffle left)
      ,@(shuffle right))))
```

Non-determinism

```
(let* ((x (random 2))
       (left right
             (split-at '(a b c)
                 (+ x
                    1))))
   (if (= x 1)
       `(,@(shuffle right)
         ,@(shuffle left))
     `(,@(shuffle left)
       ,@(shuffle right))))
```

Non-determinism

```
(let ((left right
           (split-at '(a b c)
              (+ (random 2)
                 1))))
  (if (= (random 2) 1)
      `(,@(shuffle right)
        ,@(shuffle left))
    `(,@(shuffle left)
      ,@(shuffle right))))
```

Non-determinism

```
(let ((left right (split-at '(a b c)
                            (+ (random 2) 1))))
  (if (= (random 2) 1)
      `(,@(shuffle right)
        ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right))))
```


```
(let ((left right (split-at '(a b c)
                            (+ (random 2) 1))))
  (if (= (random 2) 1)
      `(,@(shuffle right)
        ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                            (+ (random 2) 1))))
  (if (= (random 2) 1)
      `(,@(shuffle right)
        ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                            (+ (random 2) 1))))
  (if (= (random 2) 1)
      `(,@(shuffle right)
        ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right))))
```

Non-determinism

```
(let ((left right (split-at '(a b c)
                            (+ 0 1))))
  (if (= 0 1)
      `(,@(shuffle right)
        ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                            (+ 0 1))))
  (if (= 1 1)
      `(,@(shuffle right)
        ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                            (+ 1 1))))
  (if (= 0 1)
      `(,@(shuffle right)
        ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                            (+ 1 1))))
  (if (= 1 1)
      `(,@(shuffle right)
        ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right))))
```

Non-determinism

```
(let ((left right (split-at '(a b c)
                            1)))
  (if (= 0 1)
      `(,@(shuffle right)
        ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                            1)))
  (if (= 1 1)
      `(,@(shuffle right)
        ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                            2)))
  (if (= 0 1)
      `(,@(shuffle right)
        ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                            2))
  (if (= 1 1)
      `(,@(shuffle right)
        ,@(shuffle left))
      `(,@(shuffle left)
        ,@(shuffle right))))
```

Non-determinism

```
`(,@(shuffle '(a))
  ,@(shuffle '(b c)))
```

```
`(,@(shuffle '(b c))
  ,@(shuffle '(a)))
```

```
`(,@(shuffle '(a b))
  ,@(shuffle '(c)))
```

```
`(,@(shuffle '(c))
  ,@(shuffle '(a b)))
```

Non-determinism

```
'(a b c)
'(a c b)
```

```
'(b c a)
'(c b a)
```

```
'(a b c)
'(b a c)
```

```
'(c a b)
'(c b a)
```

Non-determinism

```
'(a b c)
'(a c b)
```

```
'(b c a)
'(c b a)
```

```
'(a b c)
'(b a c)
```

```
'(c a b)
'(c b a)
```

Purity

```
main ::  IO ()
main = do {
  name <- getLine;
  putStrLn "Hello "++name;
}
```

FP vs OOP

$$2 + 2$$

```
(+) :: Num a => a -> a -> a
```

```
2.+(2)
```

FP vs OOP

$$2 + 2$$

```
(+) :: Num a => a -> a -> a
```

```
2.+(2)
```

FP vs OOP

$$2 + 2$$

```
(+) :: Num a => a -> a -> a
```

```
2.+(2)
```

Michael Feathers
@mfeathers

Obserwuj

OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.

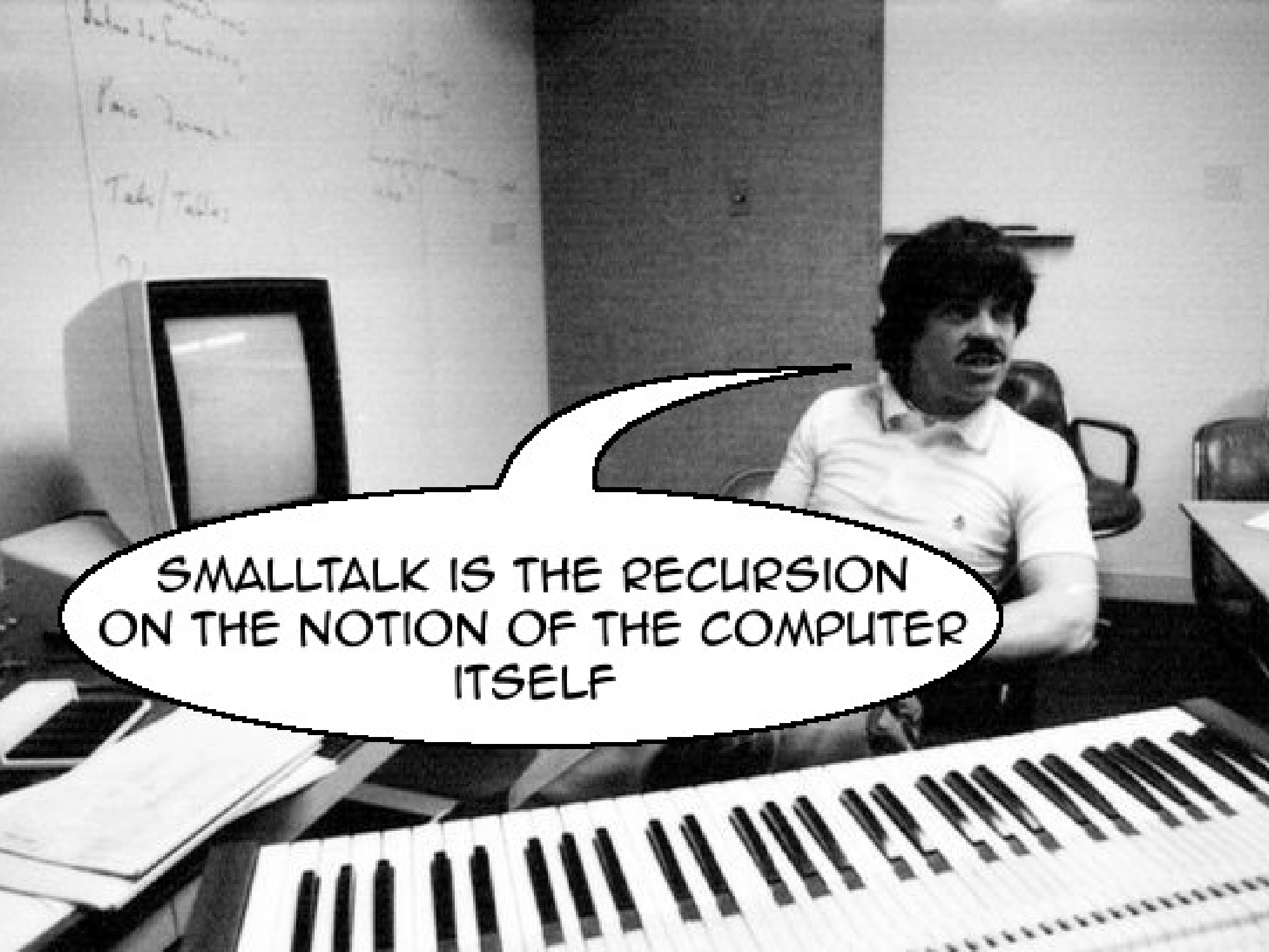
Przetłumacz tweeta

08:27 - 3 lis 2010

441 podań dalej **399** polubień

7 441 399


I COINED THE TERM "OBJECT ORIENTED PROGRAMING"


SMALLTALK IS THE RECURSION
ON THE NOTION OF THE COMPUTER
ITSELF


WE SEND AMBASSADORS,
RATHER THAN MESSAGES,
FOR IMPORTANT NEGOTIATIONS

The notion of the computer (live demo)

`1.rkt`

`https://github.com/panicz/sracket`

Recursion on the notion of the computer (live demo)

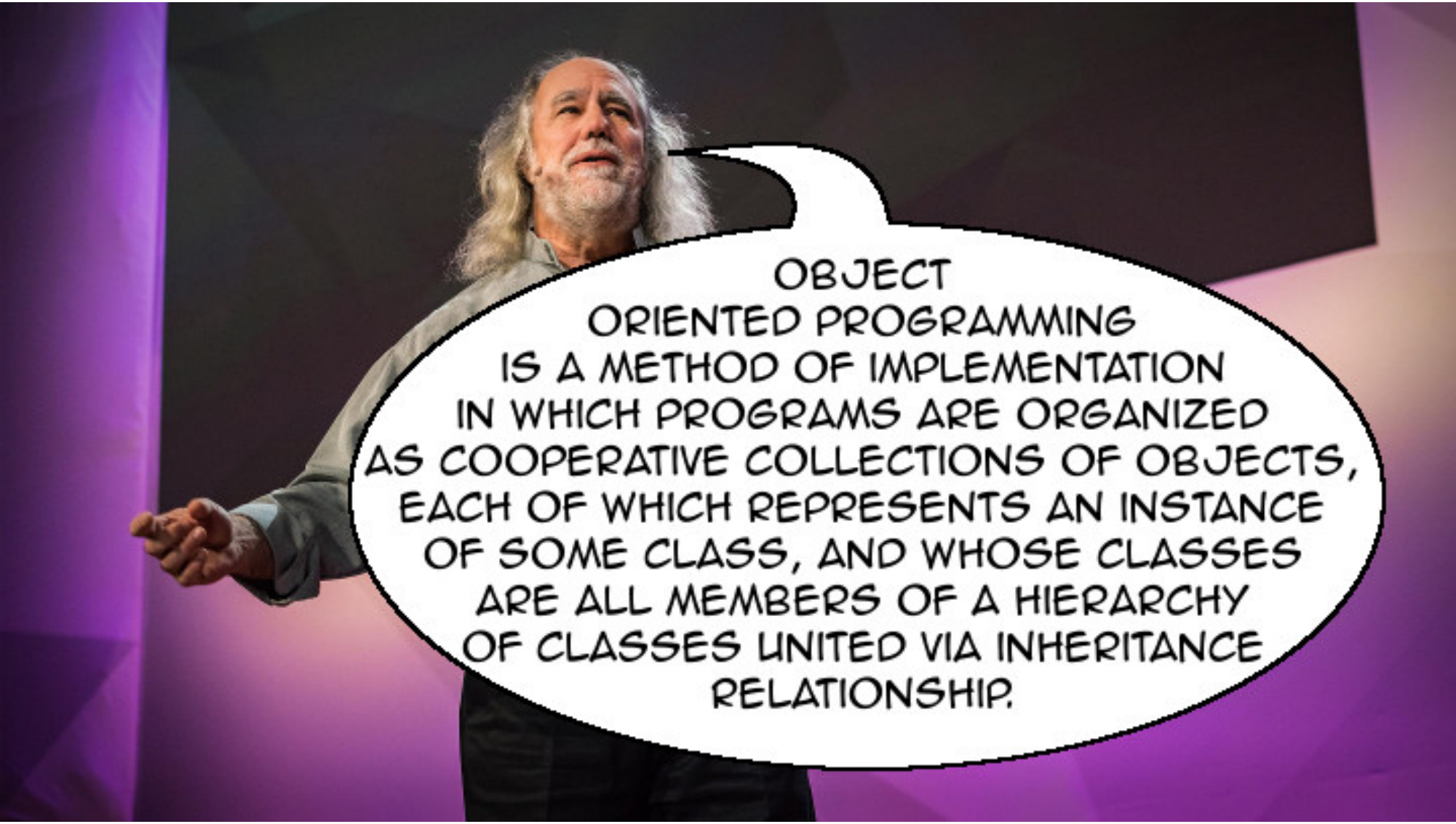
`2.rkt`

`https://github.com/panicz/sracket`

Beyond the computer (live demo)

`3.rkt`

`https://github.com/panicz/sracket`


OBJECT
ORIENTED PROGRAMMING
IS A METHOD OF IMPLEMENTATION
IN WHICH PROGRAMS ARE ORGANIZED
AS COOPERATIVE COLLECTIONS OF OBJECTS,
EACH OF WHICH REPRESENTS AN INSTANCE
OF SOME CLASS, AND WHOSE CLASSES
ARE ALL MEMBERS OF A HIERARCHY
OF CLASSES UNITED VIA INHERITANCE
RELATIONSHIP.

The four pillars of OOP

“Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

The four pillars of OOP

“Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

The four pillars of OOP

“Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

The four pillars of OOP

“Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

The four pillars of OOP

“Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

The four pillars of OOP

“Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

The four pillars of OOP

“Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

The four pillars of OOP

“Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

The four pillars of OOP

“Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

The four pillars of OOP

“Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

The four pillars of OOP

“Pillars”:

- abstraction
- ~~encapsulation~~
- inheritance
- ~~polymorphism~~

Principles (Booch et al.):

- abstraction
- ~~encapsulation~~
- ~~modularity~~
- ~~hierarchy~~

John Ⓐ De Goes
@jdegoes

Following

An abstraction is a set of types, operations on these types, and laws that give meaning to the operations across all contexts.

In functional programming, we call abstractions type classes (or sometimes modules), and contrast them with design patterns.

12:24 PM - 17 Sep 2018

24 Retweets **78** Likes

5 24 78

HIRE ERIC

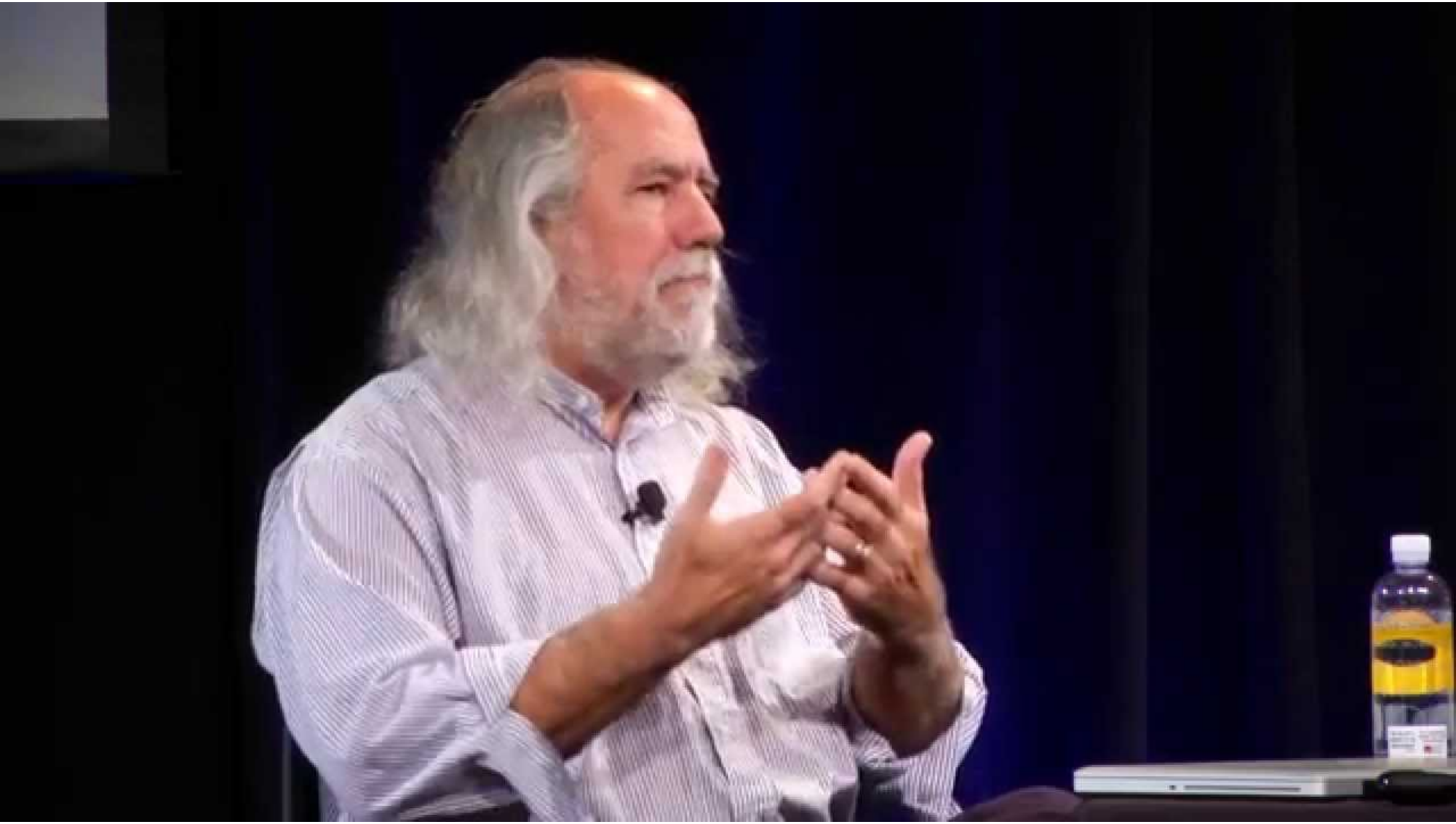
Professional Services

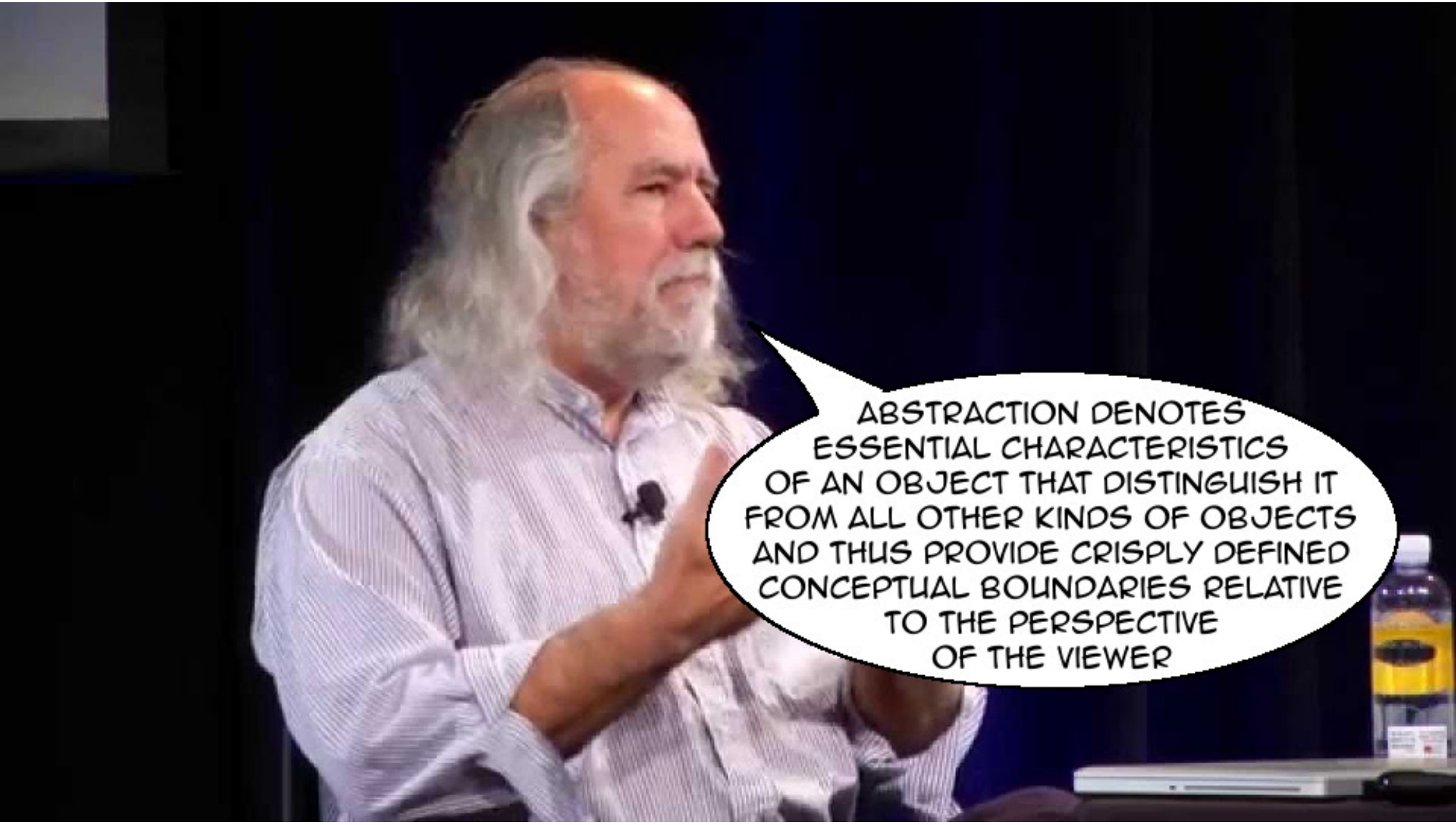
WHAT IS AN ABSTRACTION?

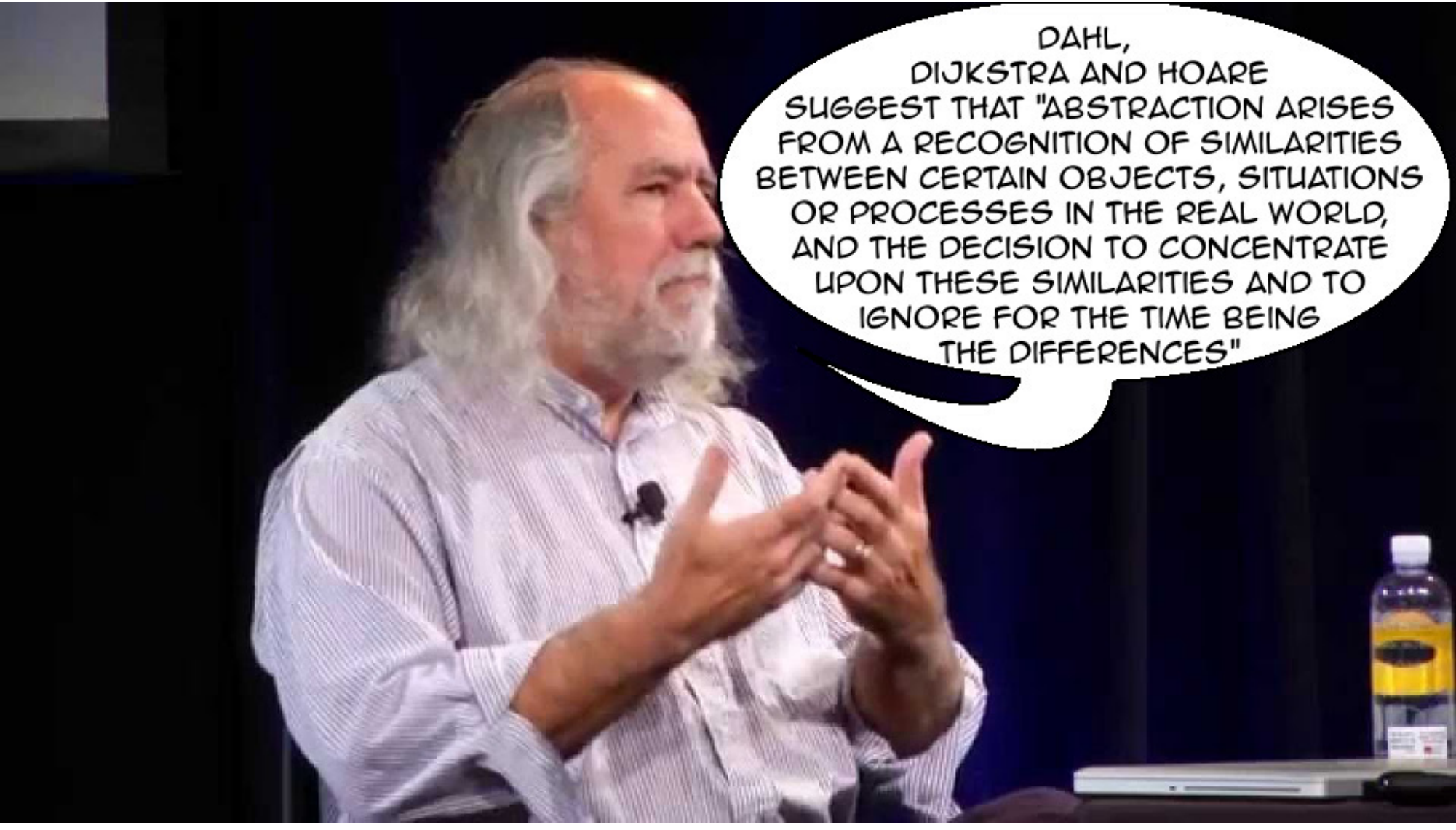
ERIC NORMAND · UPDATED AUGUST 15, 2018

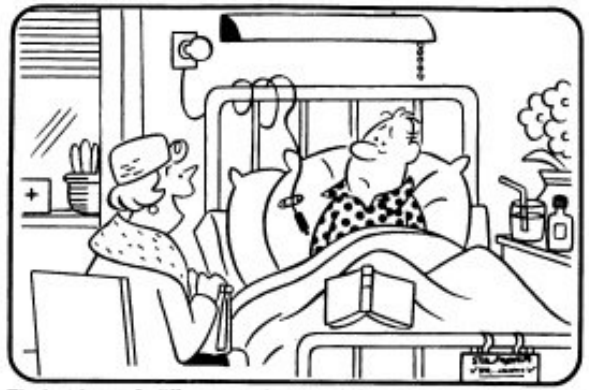
Summary: *We explore some of the background behind the meaning of the word abstraction and why we do it.*

For a term we use so much in our field, there are very few definitions of *abstraction*. And when I gave my talk called *Building Composable Abstractions*, that was a persistent question: what did I mean by *abstraction*? I'm going to be talking about abstractions a lot, both on this

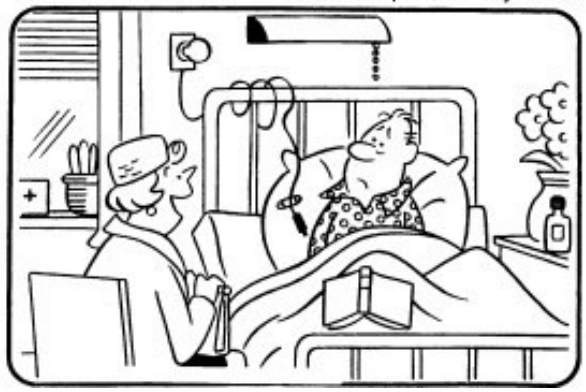



ABSTRACTION DENOTES ESSENTIAL CHARACTERISTICS OF AN OBJECT THAT DISTINGUISH IT FROM ALL OTHER KINDS OF OBJECTS AND THUS PROVIDE CRISPLY DEFINED CONCEPTUAL BOUNDARIES RELATIVE TO THE PERSPECTIVE OF THE VIEWER

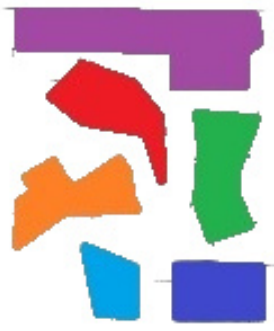

DAHL,
DIJKSTRA AND HOARE
SUGGEST THAT "ABSTRACTION ARISES
FROM A RECOGNITION OF SIMILARITIES
BETWEEN CERTAIN OBJECTS, SITUATIONS
OR PROCESSES IN THE REAL WORLD,
AND THE DECISION TO CONCENTRATE
UPON THESE SIMILARITIES AND TO
IGNORE FOR THE TIME BEING
THE DIFFERENCES"



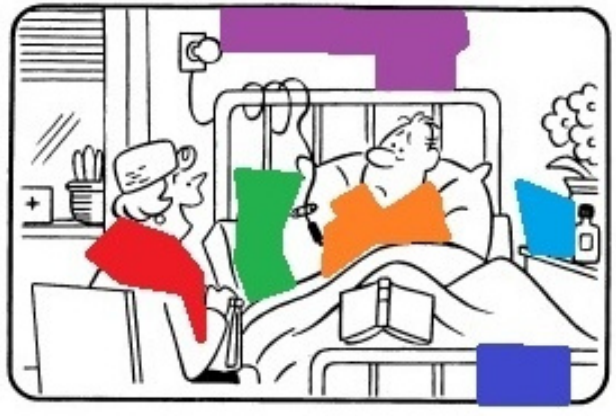
Find at least 6 differences between the panels -- If you can!



these holes



in



Definition of abstraction I

abstraction the process of intentionally omitting (leaving out) certain detail in some mental representation

Definition of abstraction II

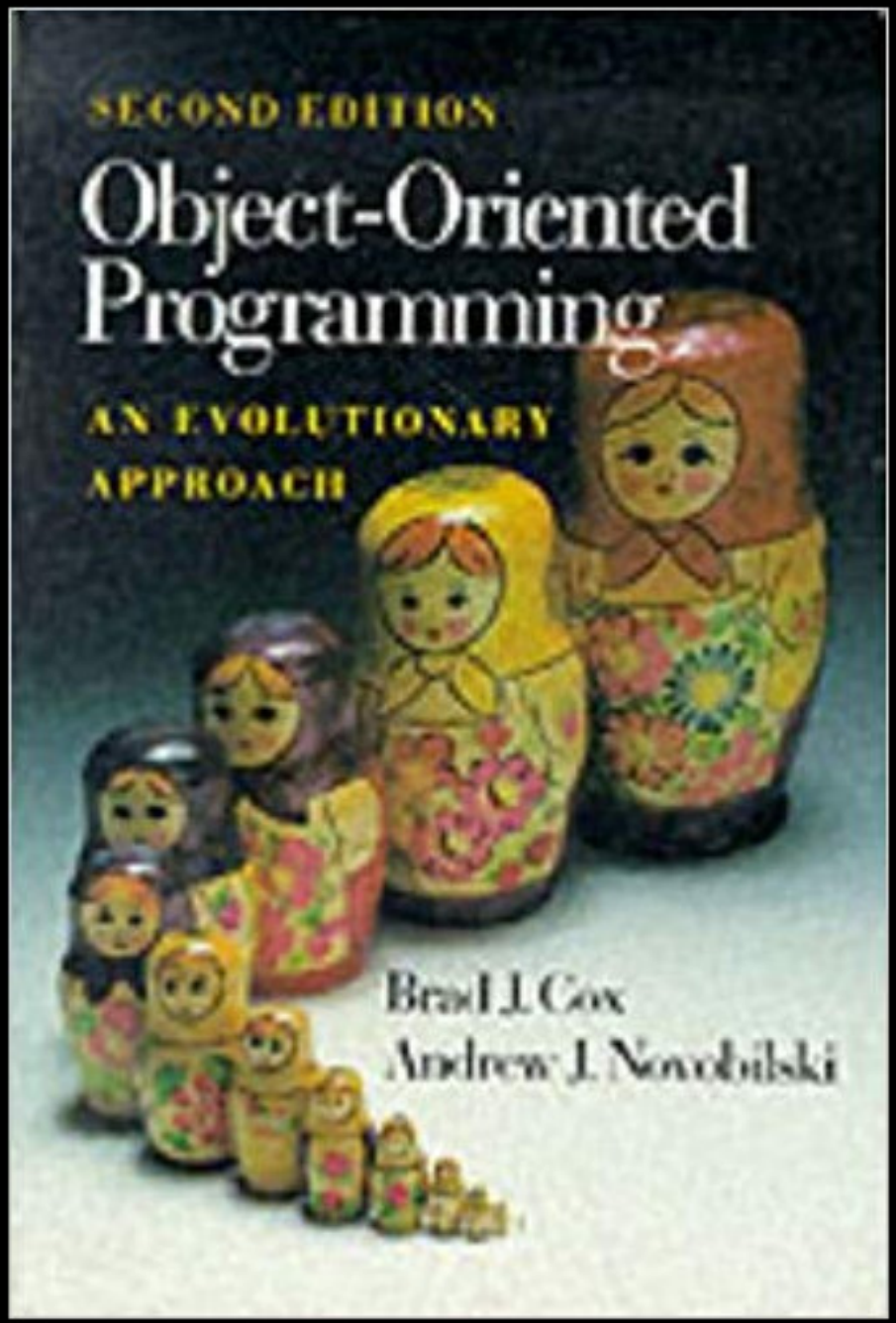
abstraction a mental representation which contains *holes* that can be filled with some content (concretized)

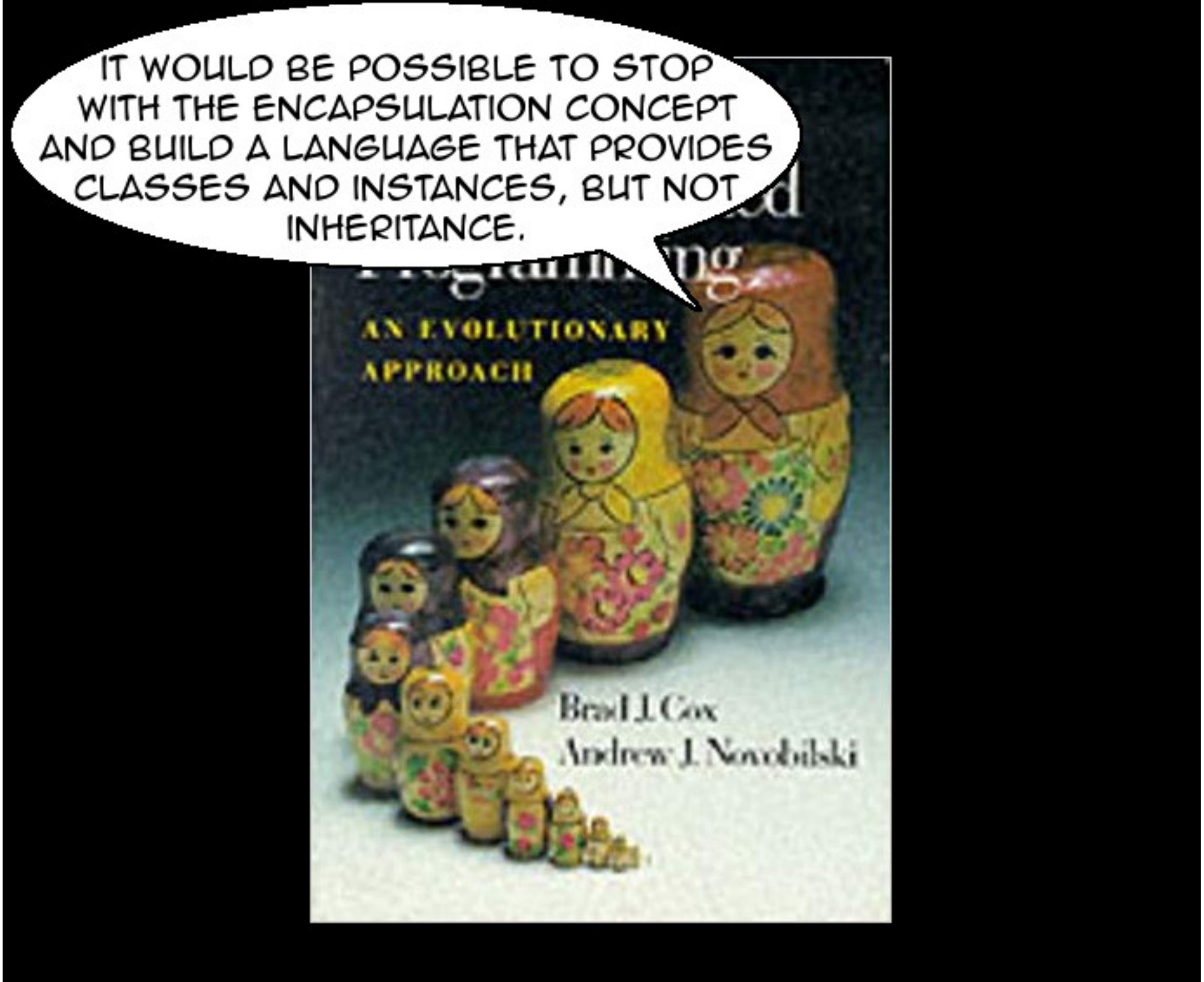

INHERITANCE [...] MEANS THAT YOU CAN SPECIFY THAT A CLASS IS A SPECIAL KIND OF ANOTHER CLASS [...].

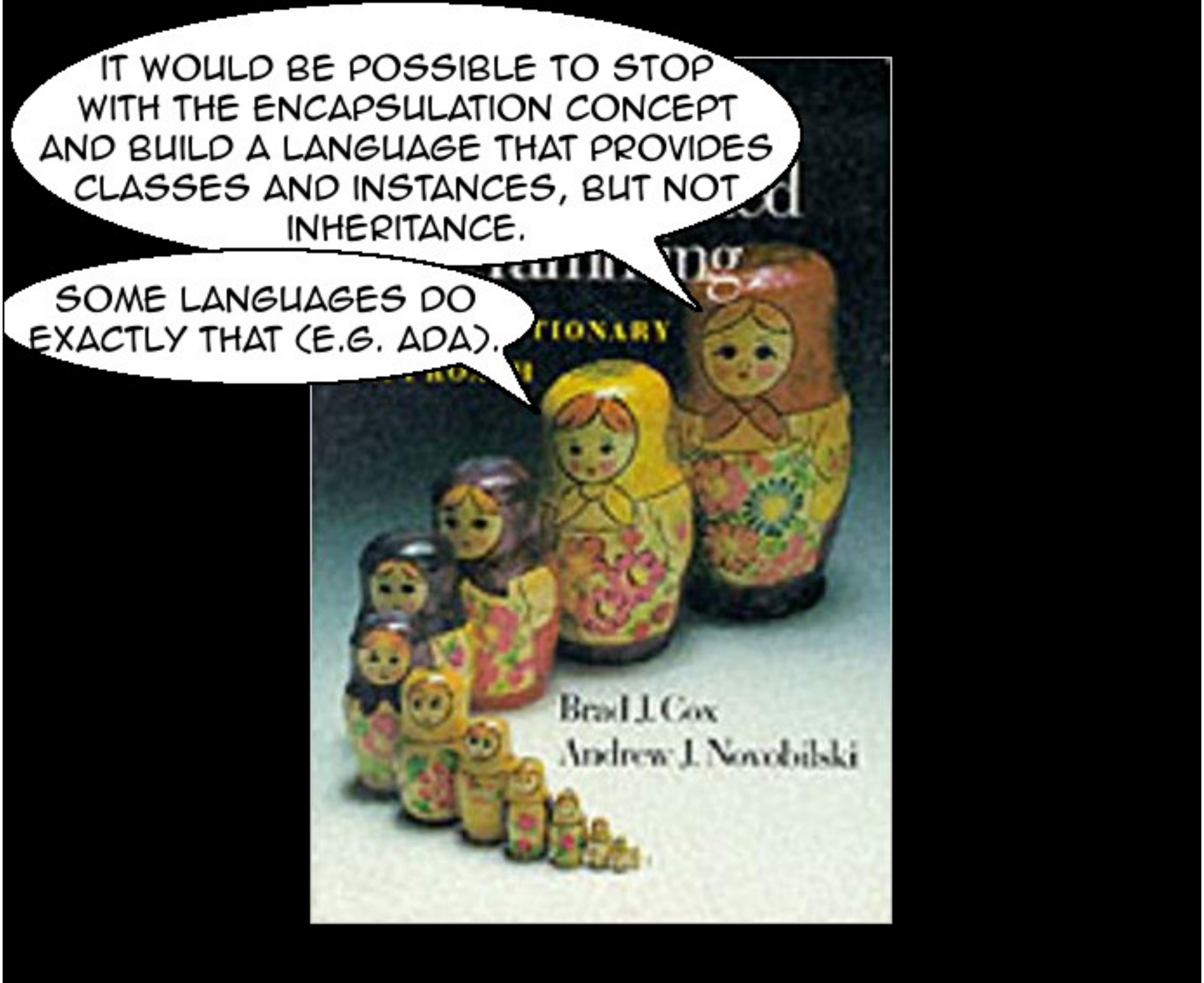

INHERITANCE [...] MEANS THAT YOU CAN SPECIFY THAT A CLASS IS A SPECIAL KIND OF ANOTHER CLASS [...].
THE UNDERLYING IDEA IS [...] THAT OF CLASSIFYING THINGS BY INCREASING DEGREES OF SPECIALIZATION

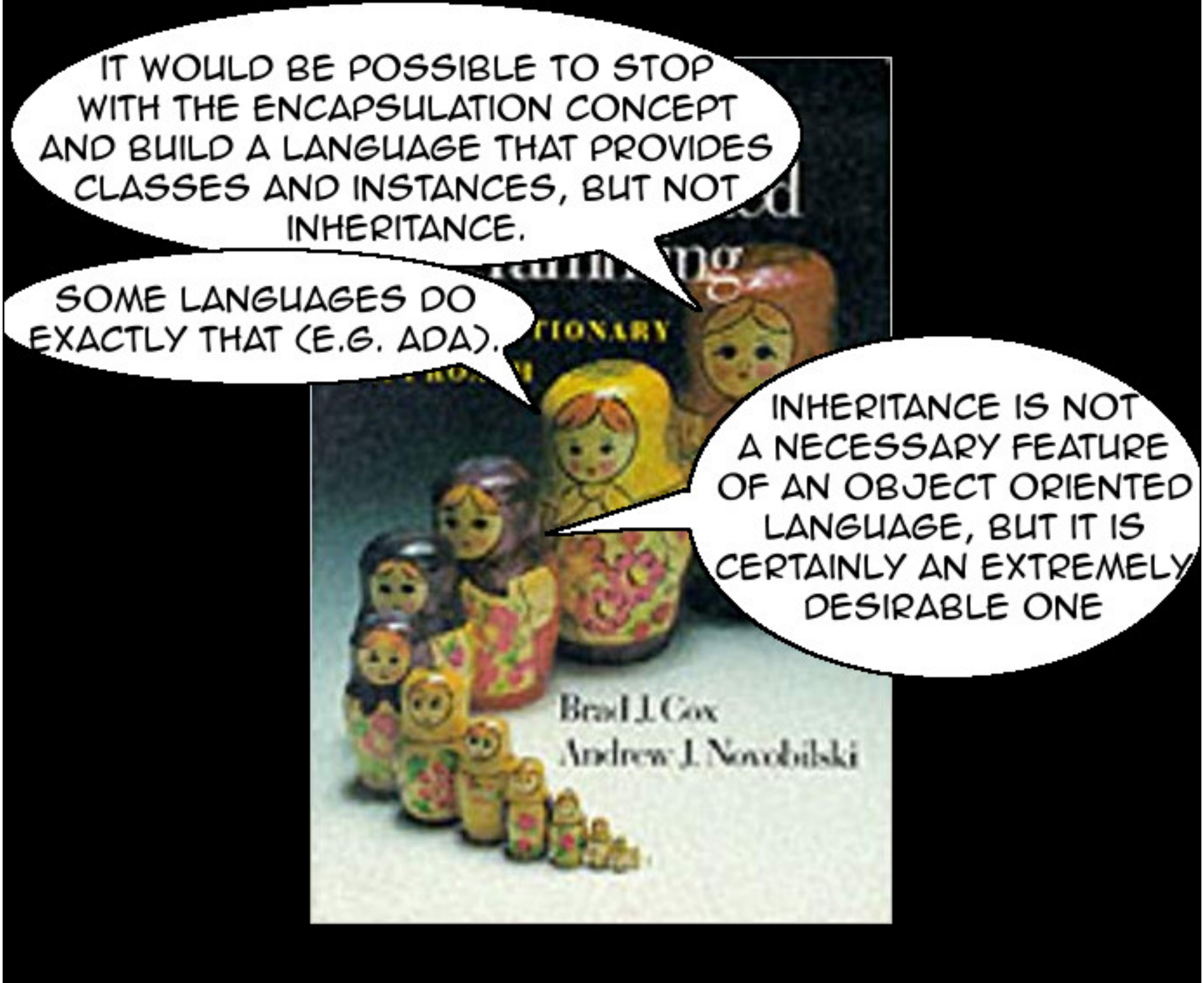

INHERITANCE [...] MEANS THAT YOU CAN SPECIFY THAT A CLASS IS A SPECIAL KIND OF ANOTHER CLASS [...].
THE UNDERLYING IDEA IS [...] THAT OF CLASSIFYING THINGS BY INCREASING DEGREES OF SPECIALIZATION
THE BOTANIST CARL VON LINNEE, BETTER KNOWN AS LINNEUS, POPULARIZED THIS WAY OF THINKING ABOUT PLANTS (AS WELL AS ANIMALS) IN THE EIGHTEENTH CENTURY. [...]


INHERITANCE [...] MEANS THAT YOU CAN SPECIFY THAT A CLASS IS A SPECIAL KIND OF ANOTHER CLASS [...].
THE UNDERLYING IDEA IS [...] THAT OF CLASSIFYING THINGS BY INCREASING DEGREES OF SPECIALIZATION
THE BOTANIST CARL VON LINNEE, BETTER KNOWN AS LINNEUS, POPULARIZED THIS WAY OF THINKING ABOUT PLANTS (AS WELL AS ANIMALS) IN THE EIGHTEENTH CENTURY. [...]
THE IDEA EXTENDED THE VARY WAY IN WHICH PEOPLE COULD THINK ABOUT THE WORLD


SECOND EDITION
Object-Oriented Programming
AN EVOLUTIONARY APPROACH
Brad J. Cox
Andrew J. Novobilski


IT WOULD BE POSSIBLE TO STOP WITH THE ENCAPSULATION CONCEPT AND BUILD A LANGUAGE THAT PROVIDES CLASSES AND INSTANCES, BUT NOT INHERITANCE.
AN EVOLUTIONARY APPROACH
Brad J. Cox
Andrew J. Novobilski

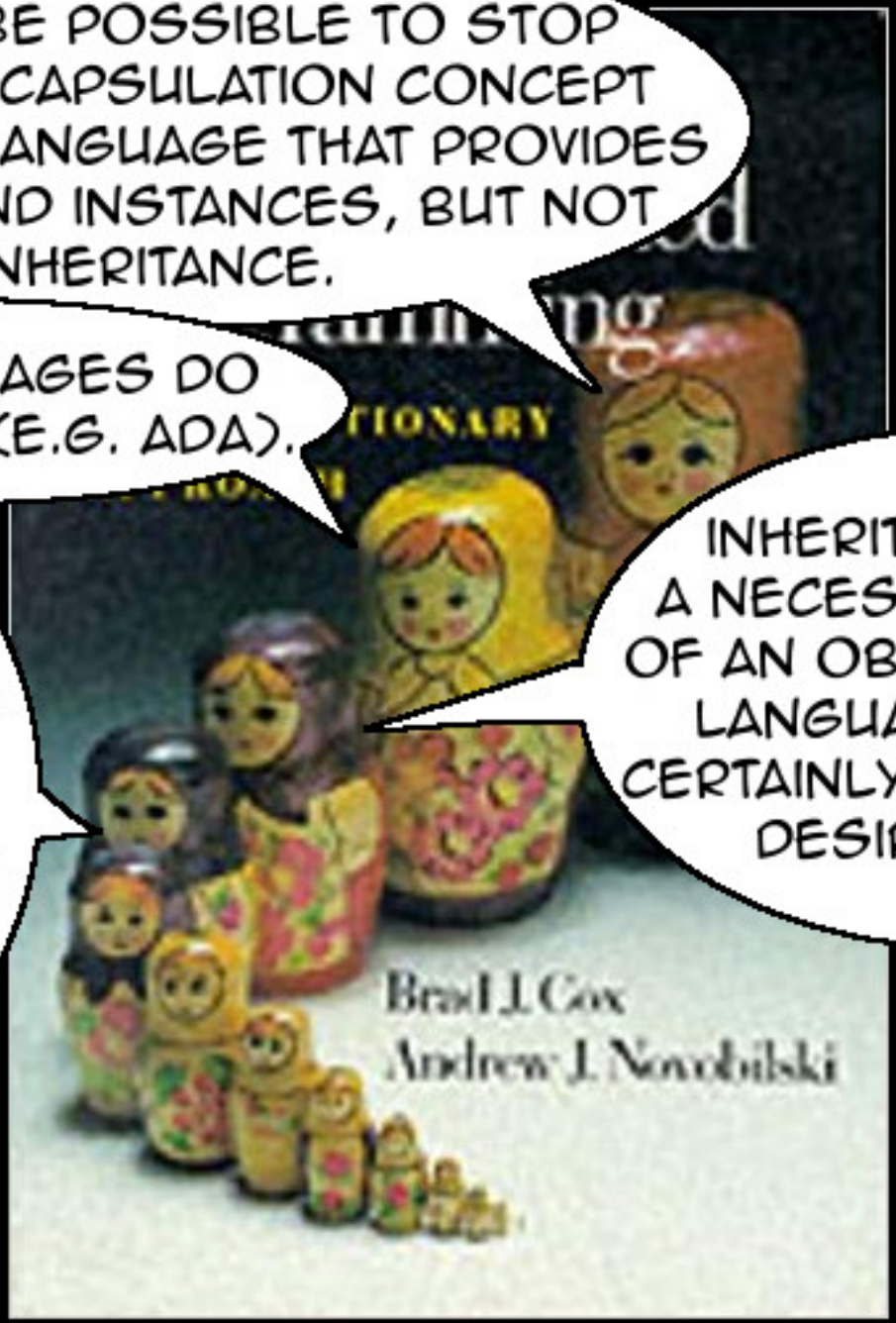

IT WOULD BE POSSIBLE TO STOP WITH THE ENCAPSULATION CONCEPT AND BUILD A LANGUAGE THAT PROVIDES CLASSES AND INSTANCES, BUT NOT INHERITANCE.
SOME LANGUAGES DO EXACTLY THAT (E.G. ADA).
Brad J. Cox
Andrew J. Novobilski


IT WOULD BE POSSIBLE TO STOP WITH THE ENCAPSULATION CONCEPT AND BUILD A LANGUAGE THAT PROVIDES CLASSES AND INSTANCES, BUT NOT INHERITANCE.
SOME LANGUAGES DO EXACTLY THAT (E.G. ADA).
INHERITANCE IS NOT A NECESSARY FEATURE OF AN OBJECT ORIENTED LANGUAGE, BUT IT IS CERTAINLY AN EXTREMELY DESIRABLE ONE
Brad J. Cox
Andrew J. Novobilski

IT WOULD BE POSSIBLE TO STOP WITH THE ENCAPSULATION CONCEPT AND BUILD A LANGUAGE THAT PROVIDES CLASSES AND INSTANCES, BUT NOT INHERITANCE.

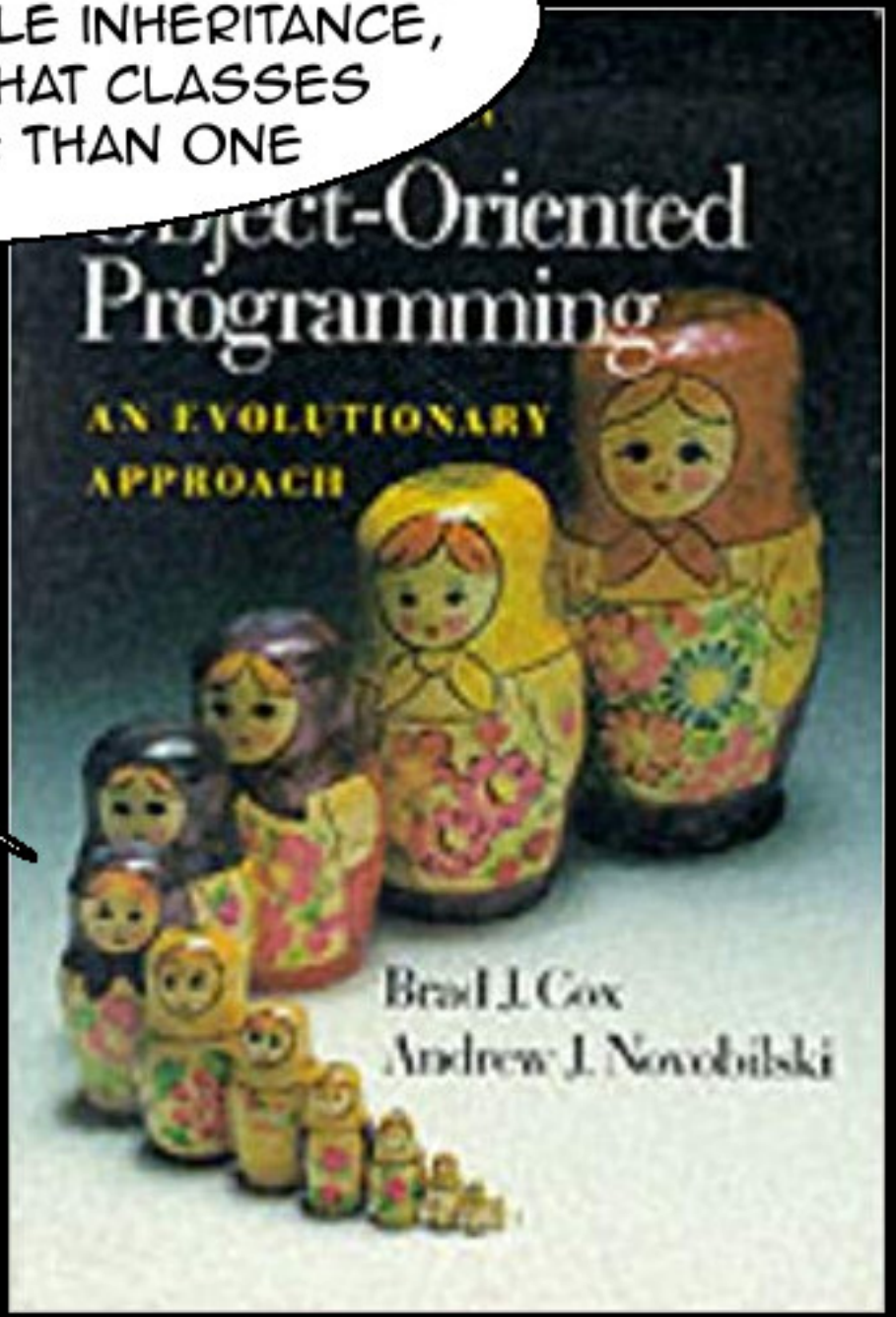
SOME LANGUAGES DO EXACTLY THAT (E.G. ADA).

ALSO, THE SIMPLE LINEAR SCHEME TO BE DESCRIBED THERE IS NOT THE ONLY WAY TO PROVIDE INHERITANCE



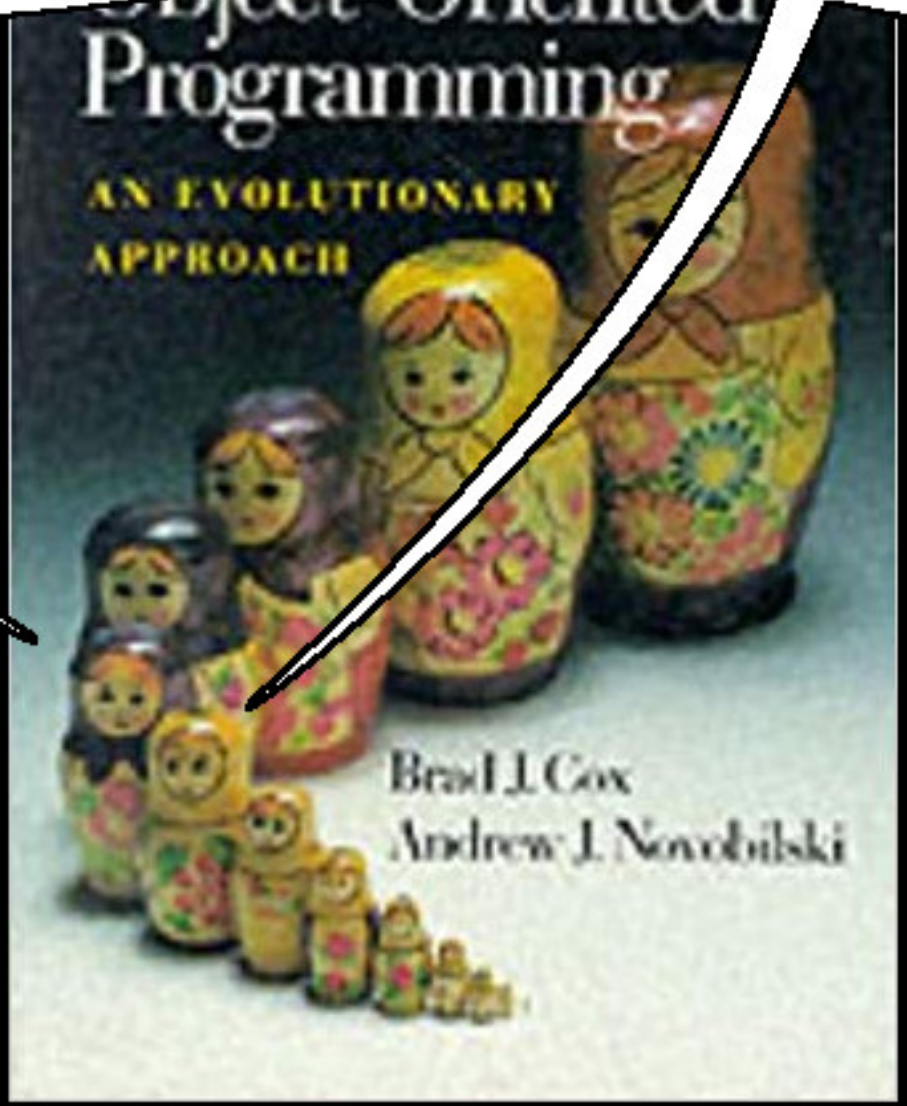
INHERITANCE IS NOT A NECESSARY FEATURE OF AN OBJECT ORIENTED LANGUAGE, BUT IT IS CERTAINLY AN EXTREMELY DESIRABLE ONE

FOR EXAMPLE, SOME LANGUAGES PROVIDE MULTIPLE INHERITANCE, WHICH MEANS THAT CLASSES CAN HAVE MORE THAN ONE SUPERCLASS



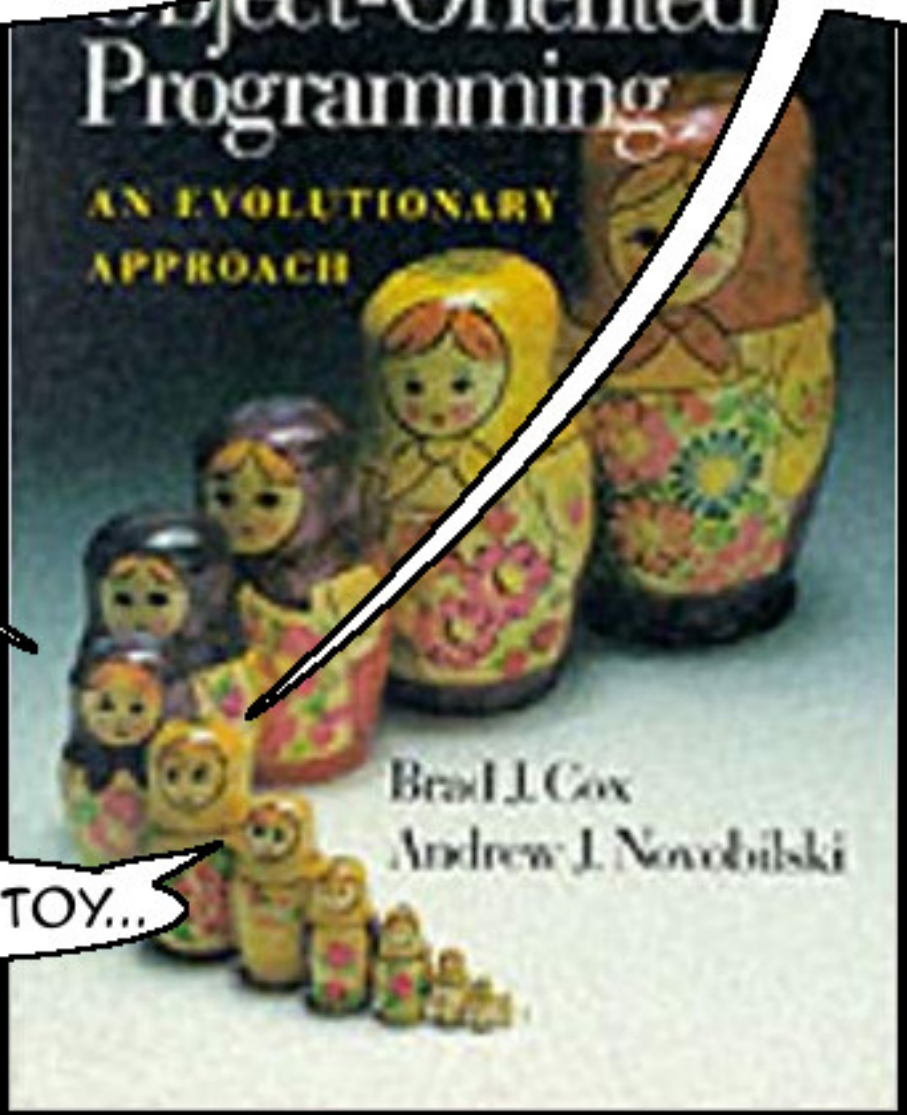
FOR EXAMPLE, SOME LANGUAGES PROVIDE MULTIPLE INHERITANCE, WHICH MEANS THAT CLASSES CAN HAVE MORE THAN ONE SUPERCLASS

THESE LANGUAGES PROVIDE MORE GENERALITY, SOLVING, FOR EXAMPLE, PROBLEMS LIKE EXPRESSING WHAT A TOY TRUCK IS

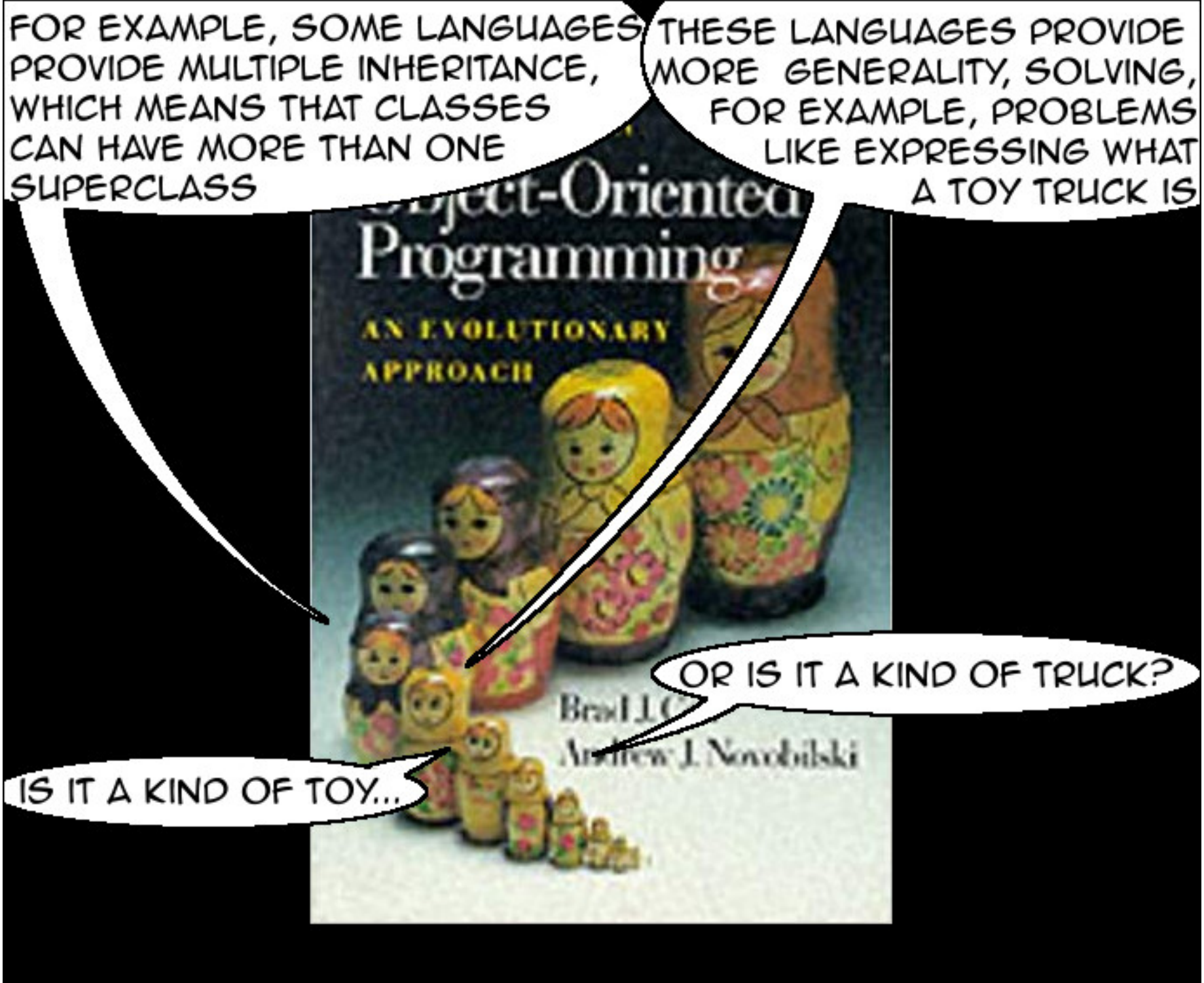


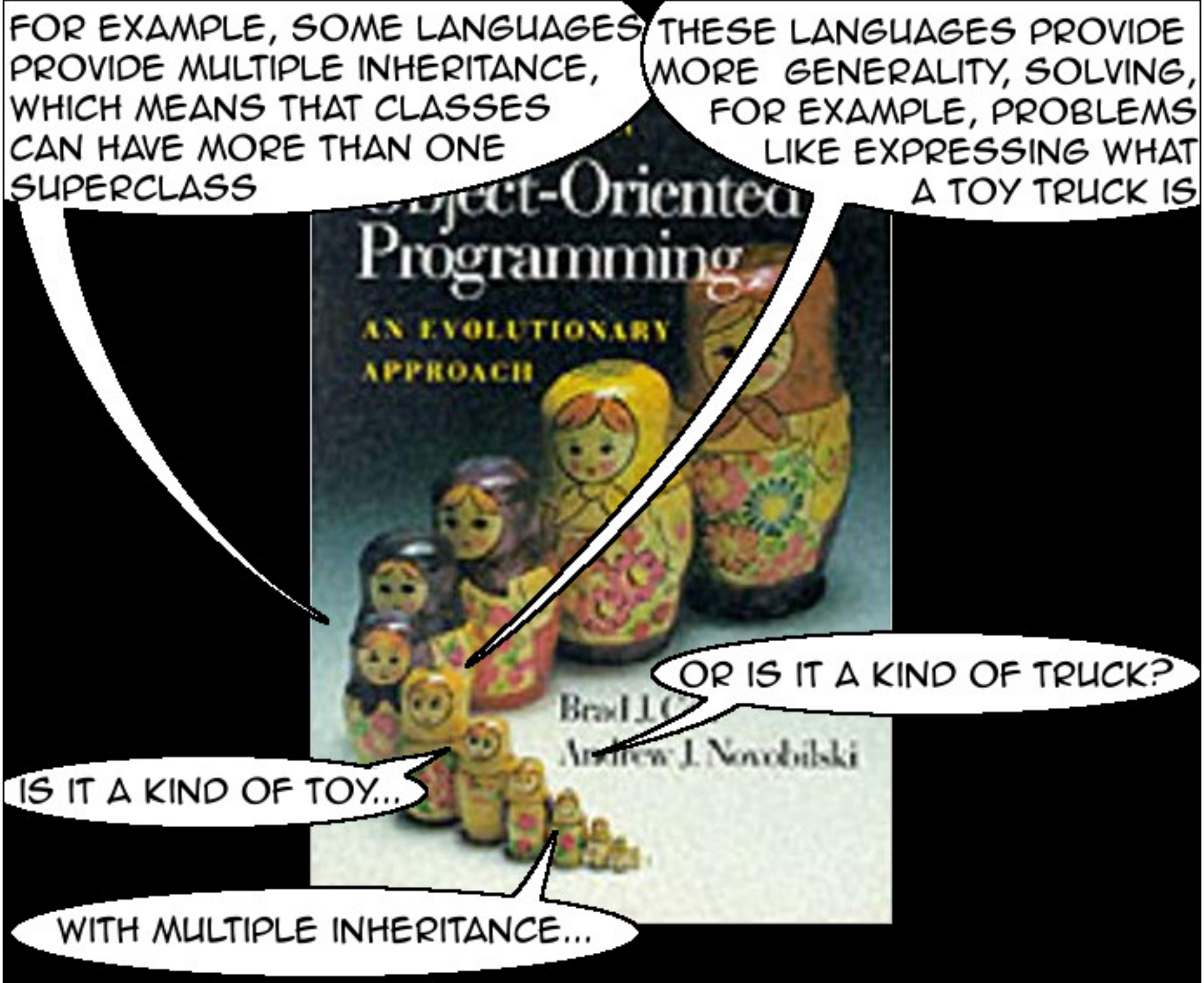
FOR EXAMPLE, SOME LANGUAGES PROVIDE MULTIPLE INHERITANCE, WHICH MEANS THAT CLASSES CAN HAVE MORE THAN ONE SUPERCLASS

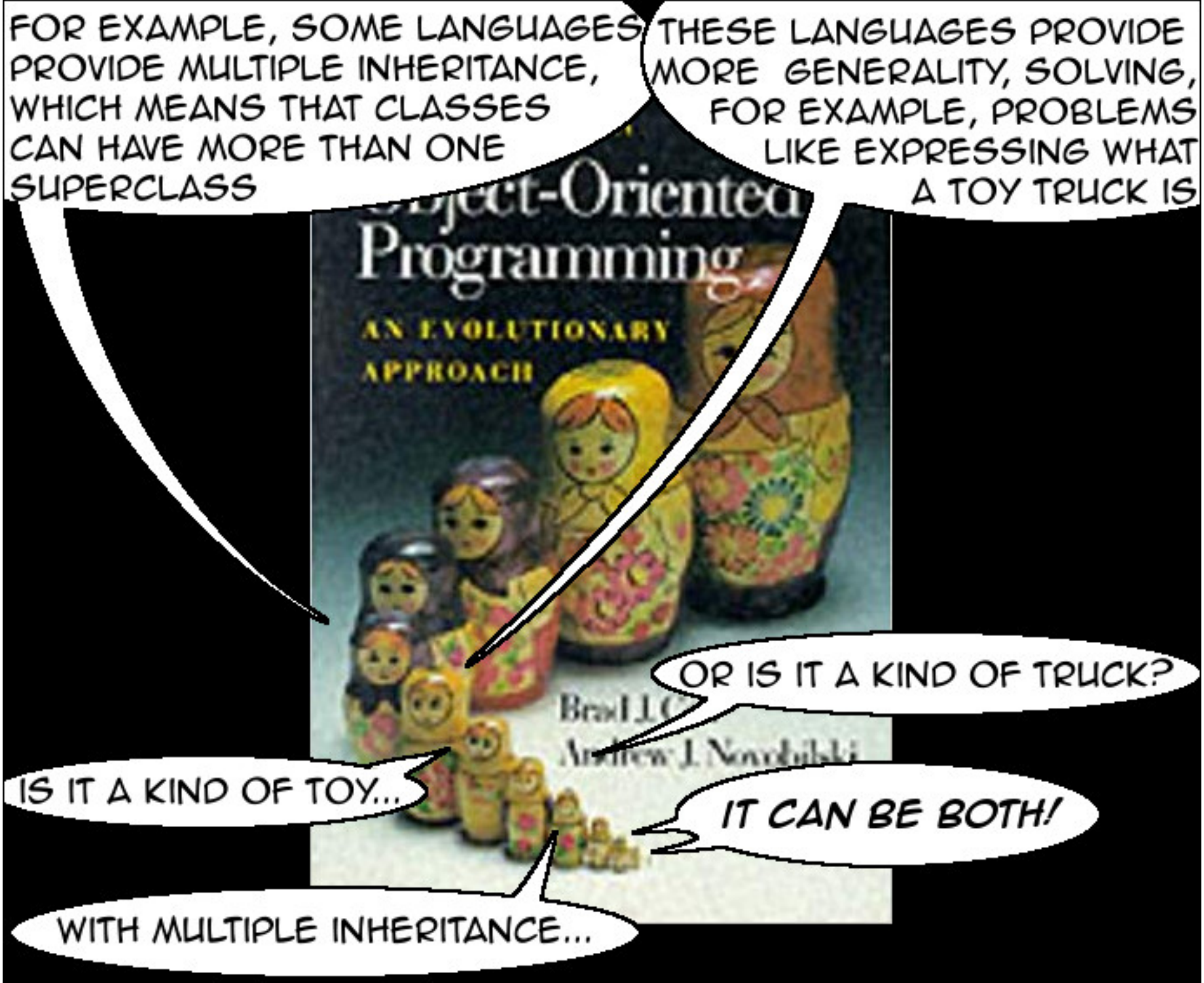
THESE LANGUAGES PROVIDE MORE GENERALITY, SOLVING, FOR EXAMPLE, PROBLEMS LIKE EXPRESSING WHAT A TOY TRUCK IS

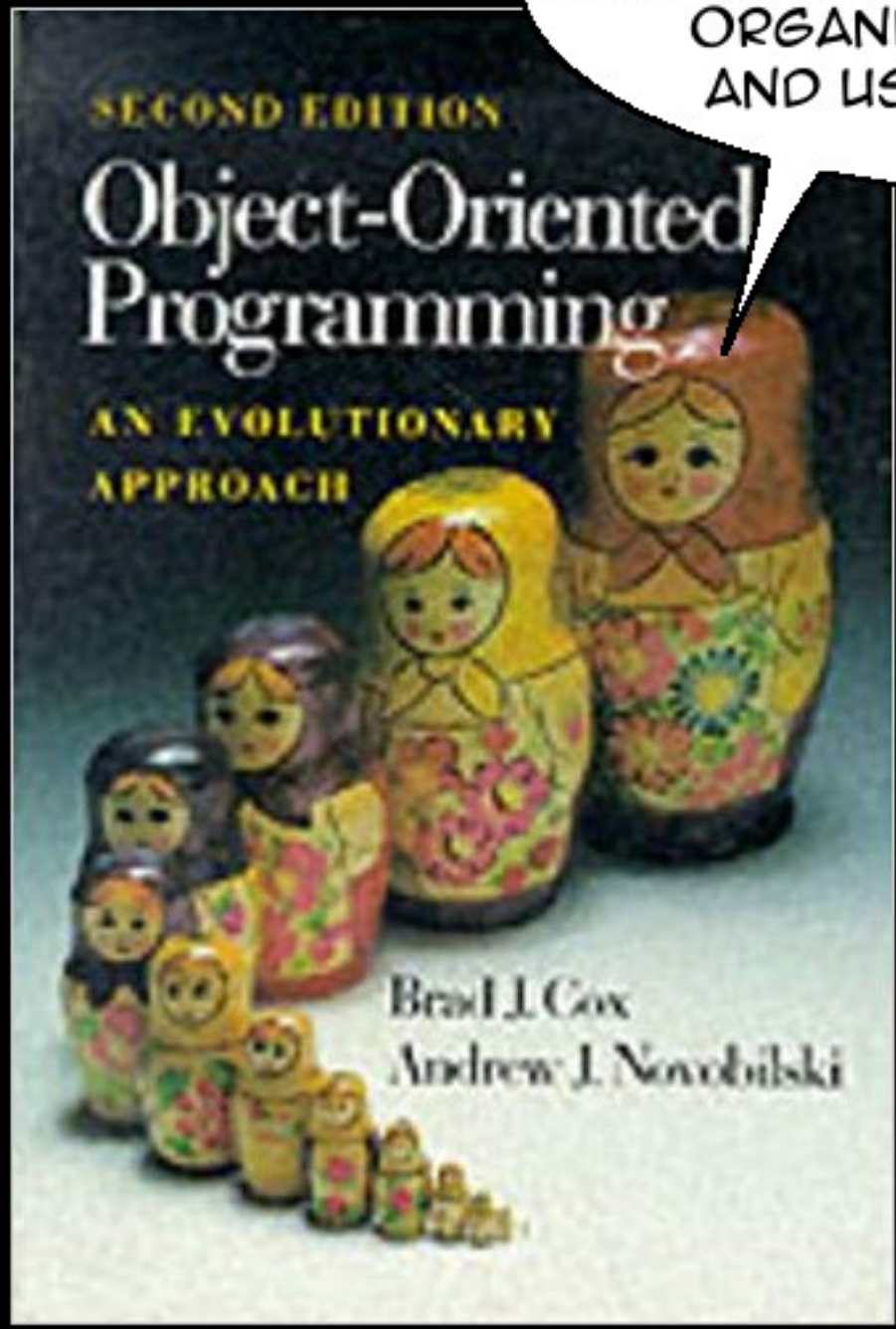


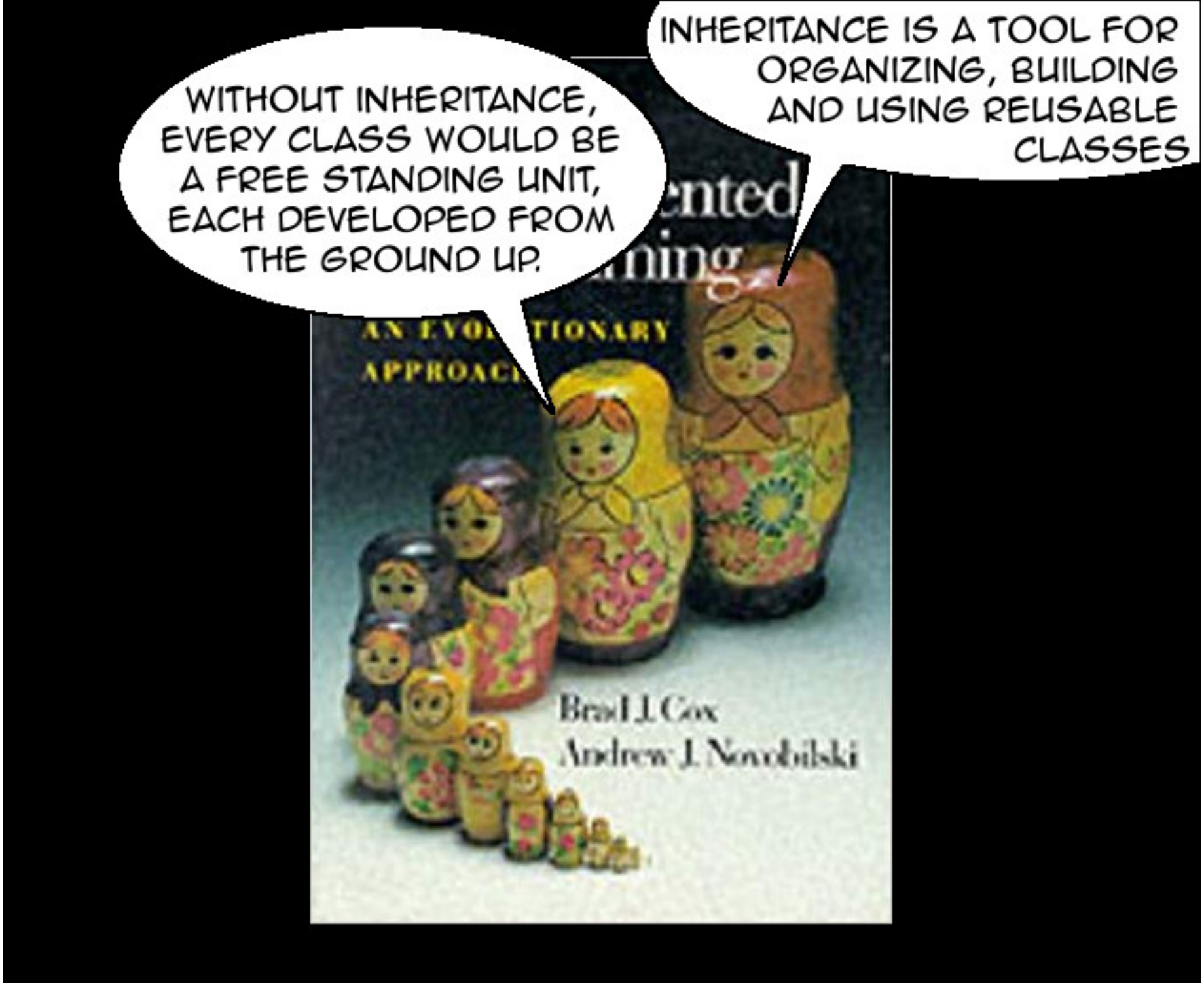
IS IT A KIND OF TOY...

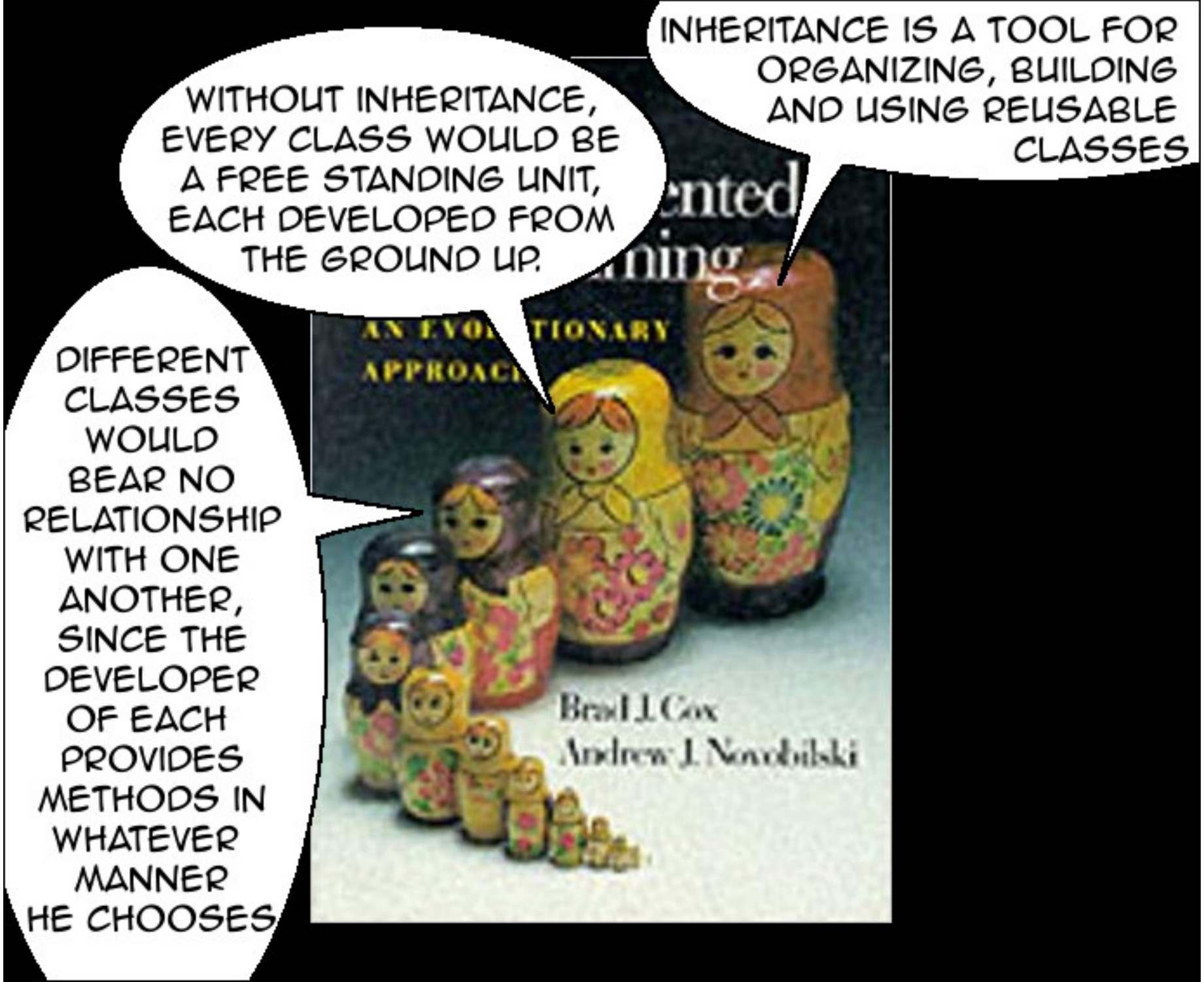

FOR EXAMPLE, SOME LANGUAGES PROVIDE MULTIPLE INHERITANCE, WHICH MEANS THAT CLASSES CAN HAVE MORE THAN ONE SUPERCLASS
THESE LANGUAGES PROVIDE MORE GENERALITY, SOLVING, FOR EXAMPLE, PROBLEMS LIKE EXPRESSING WHAT A TOY TRUCK IS
Object-Oriented Programming
AN EVOLUTIONARY APPROACH
Brad J. C
Andrew J. Novobilski
OR IS IT A KIND OF TRUCK?
IS IT A KIND OF TOY...

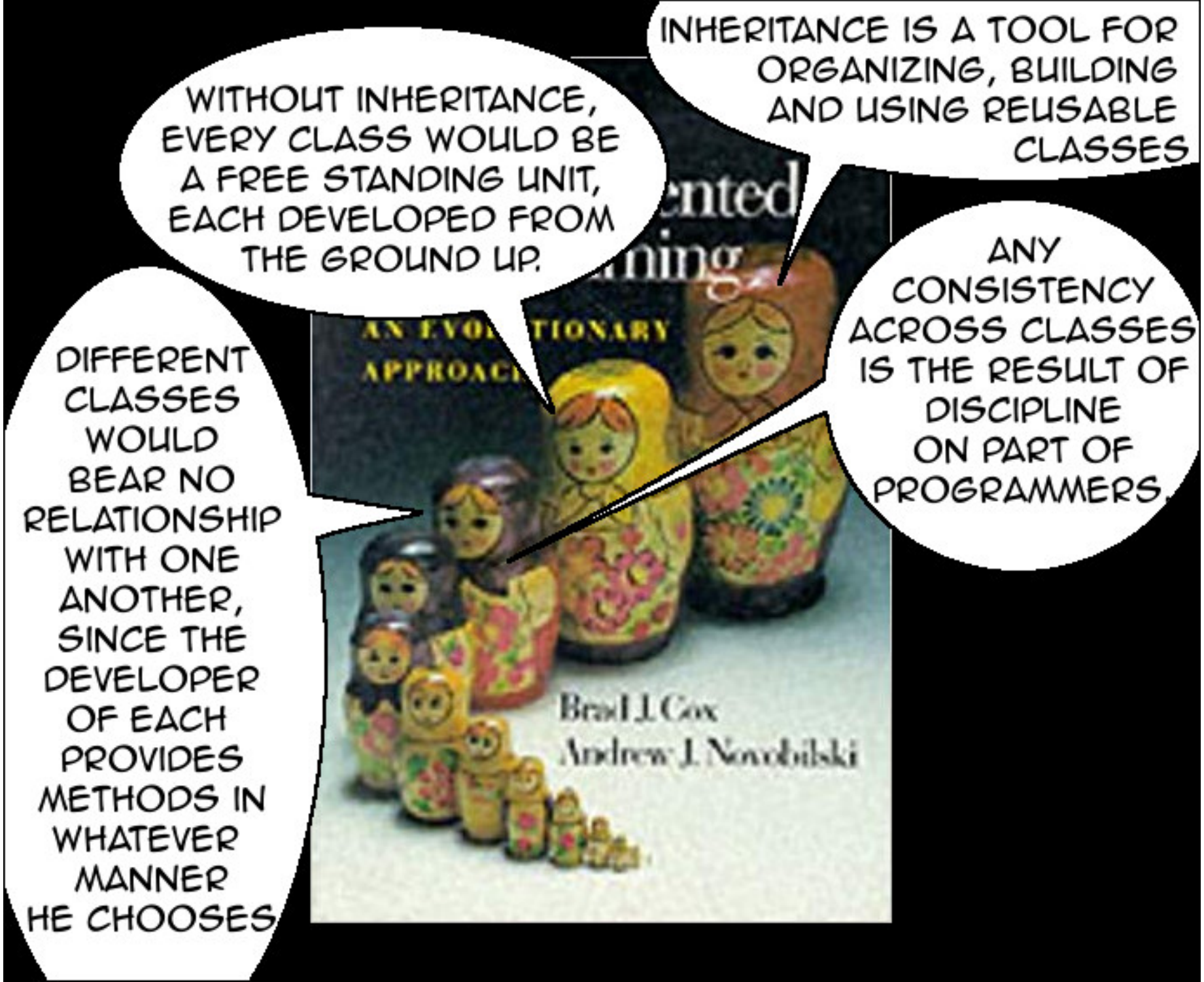

FOR EXAMPLE, SOME LANGUAGES PROVIDE MULTIPLE INHERITANCE, WHICH MEANS THAT CLASSES CAN HAVE MORE THAN ONE SUPERCLASS
THESE LANGUAGES PROVIDE MORE GENERALITY, SOLVING, FOR EXAMPLE, PROBLEMS LIKE EXPRESSING WHAT A TOY TRUCK IS
Object-Oriented Programming
AN EVOLUTIONARY APPROACH
Andrew J. Novobilski
OR IS IT A KIND OF TRUCK?
IS IT A KIND OF TOY...
WITH MULTIPLE INHERITANCE...


FOR EXAMPLE, SOME LANGUAGES PROVIDE MULTIPLE INHERITANCE, WHICH MEANS THAT CLASSES CAN HAVE MORE THAN ONE SUPERCLASS
THESE LANGUAGES PROVIDE MORE GENERALITY, SOLVING, FOR EXAMPLE, PROBLEMS LIKE EXPRESSING WHAT A TOY TRUCK IS
Object-Oriented Programming
AN EVOLUTIONARY APPROACH
OR IS IT A KIND OF TRUCK?
IS IT A KIND OF TOY...
IT CAN BE BOTH!
WITH MULTIPLE INHERITANCE...


INHERITANCE IS A TOOL FOR ORGANIZING, BUILDING AND USING REUSABLE CLASSES
SECOND EDITION
Object-Oriented Programming
AN EVOLUTIONARY APPROACH
Brad J. Cox
Andrew J. Novobilski


WITHOUT INHERITANCE, EVERY CLASS WOULD BE A FREE STANDING UNIT, EACH DEVELOPED FROM THE GROUND UP.
INHERITANCE IS A TOOL FOR ORGANIZING, BUILDING AND USING REUSABLE CLASSES
Brad J. Cox
Andrew J. Novobilski

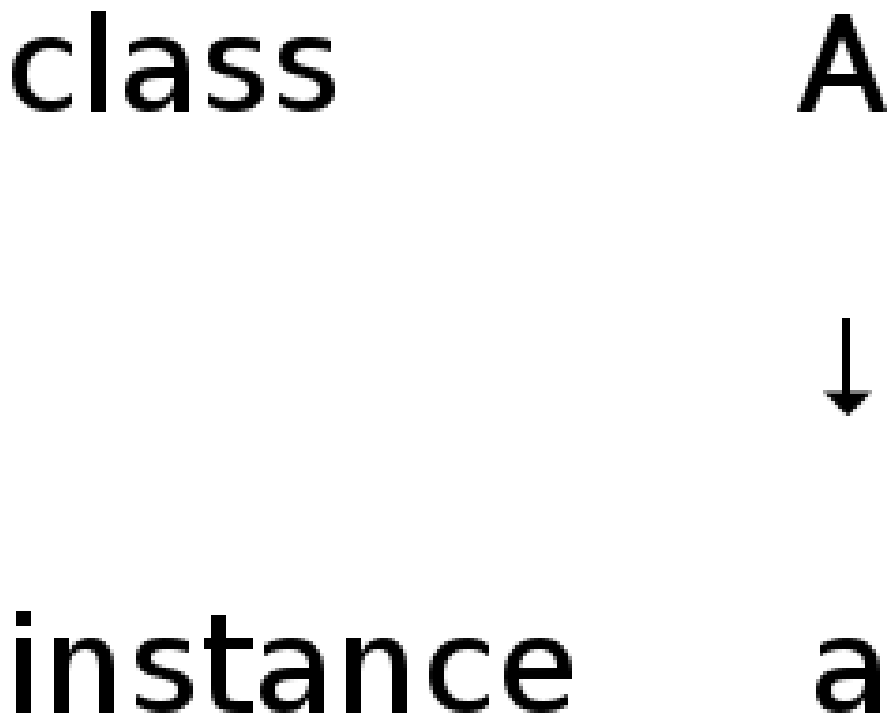

INHERITANCE IS A TOOL FOR ORGANIZING, BUILDING AND USING REUSABLE CLASSES
WITHOUT INHERITANCE, EVERY CLASS WOULD BE A FREE STANDING UNIT, EACH DEVELOPED FROM THE GROUND UP.
DIFFERENT CLASSES WOULD BEAR NO RELATIONSHIP WITH ONE ANOTHER, SINCE THE DEVELOPER OF EACH PROVIDES METHODS IN WHATEVER MANNER HE CHOOSES
Brad J. Cox
Andrew J. Novobilski


INHERITANCE IS A TOOL FOR ORGANIZING, BUILDING AND USING REUSABLE CLASSES
WITHOUT INHERITANCE, EVERY CLASS WOULD BE A FREE STANDING UNIT, EACH DEVELOPED FROM THE GROUND UP.
ANY CONSISTENCY ACROSS CLASSES IS THE RESULT OF DISCIPLINE ON PART OF PROGRAMMERS
DIFFERENT CLASSES WOULD BEAR NO RELATIONSHIP WITH ONE ANOTHER, SINCE THE DEVELOPER OF EACH PROVIDES METHODS IN WHATEVER MANNER HE CHOOSES
Brad J. Cox
Andrew J. Novobilski


INHERITANCE IS A TOOL FOR ORGANIZING, BUILDING AND USING REUSABLE CLASSES
WITHOUT INHERITANCE, EVERY CLASS WOULD BE A FREE STANDING UNIT, EACH DEVELOPED FROM THE GROUND UP.
ANY CONSISTENCY ACROSS CLASSES IS THE RESULT OF DISCIPLINE ON PART OF PROGRAMMERS
DIFFERENT CLASSES WOULD BEAR NO RELATIONSHIP WITH ONE ANOTHER, SINCE THE DEVELOPER OF EACH PROVIDES METHODS IN WHATEVER MANNER HE CHOOSES
INHERITANCE MAKES IT POSSIBLE TO DEFINE NEW SOFTWARE IN THE SAME WAY WE INTRODUCE A NEW CONCEPT TO A NEWCOMMER...


INHERITANCE IS A TOOL FOR ORGANIZING, BUILDING AND USING REUSABLE CLASSES
WITHOUT INHERITANCE, EVERY CLASS WOULD BE A FREE STANDING UNIT, EACH DEVELOPED FROM THE GROUND UP.
ANY CONSISTENCY ACROSS CLASSES IS THE RESULT OF DISCIPLINE ON PART OF PROGRAMMERS
DIFFERENT CLASSES WOULD BEAR NO RELATIONSHIP WITH ONE ANOTHER, SINCE THE DEVELOPER OF EACH PROVIDES METHODS IN WHATEVER MANNER HE CHOOSES
INHERITANCE MAKES IT POSSIBLE TO DEFINE NEW SOFTWARE IN THE SAME WAY WE INTRODUCE A NEW CONCEPT TO A NEWCOMMER...
BY COMPARING IT WITH SOMETHING THAT IS ALREADY FAMILIAR

class A

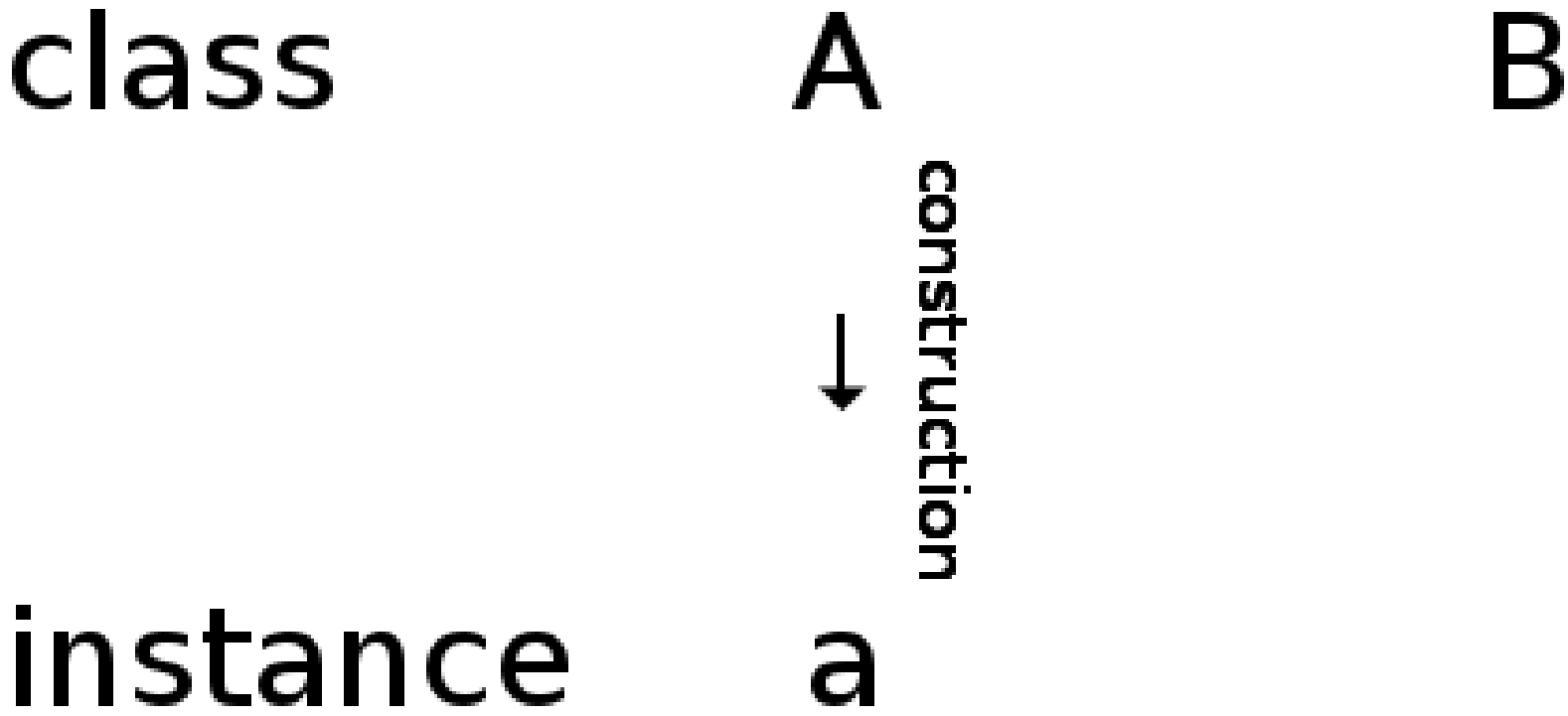

class
A
instance
a

class A

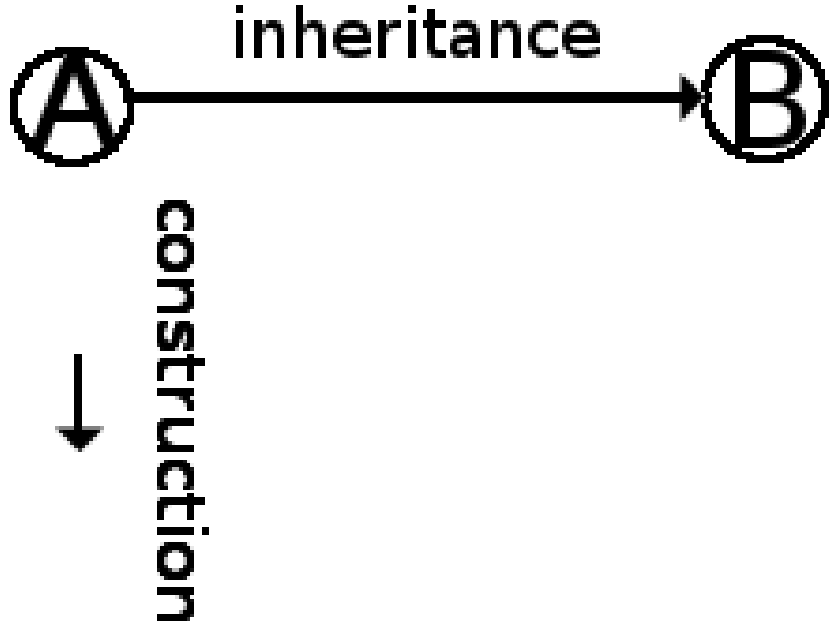
construction

↓

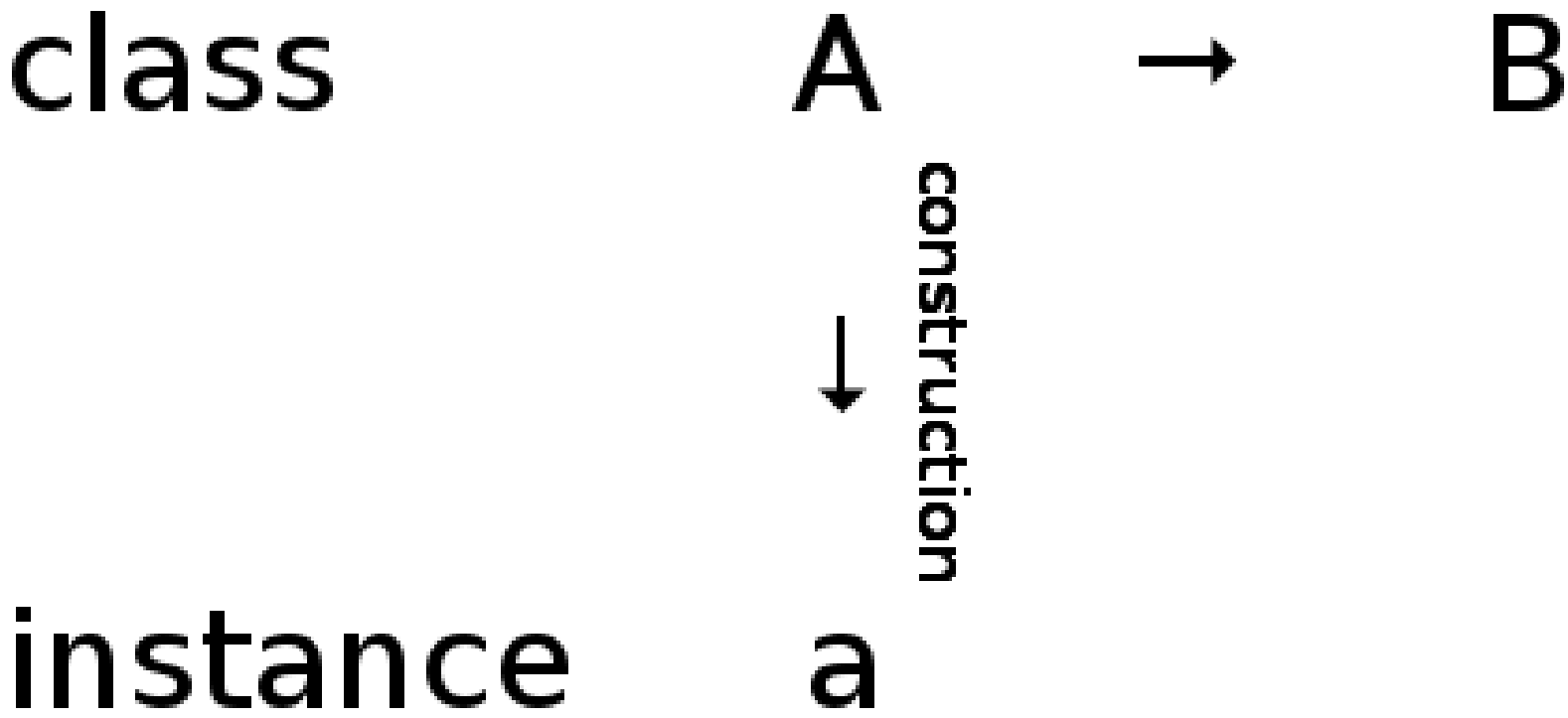
instance a

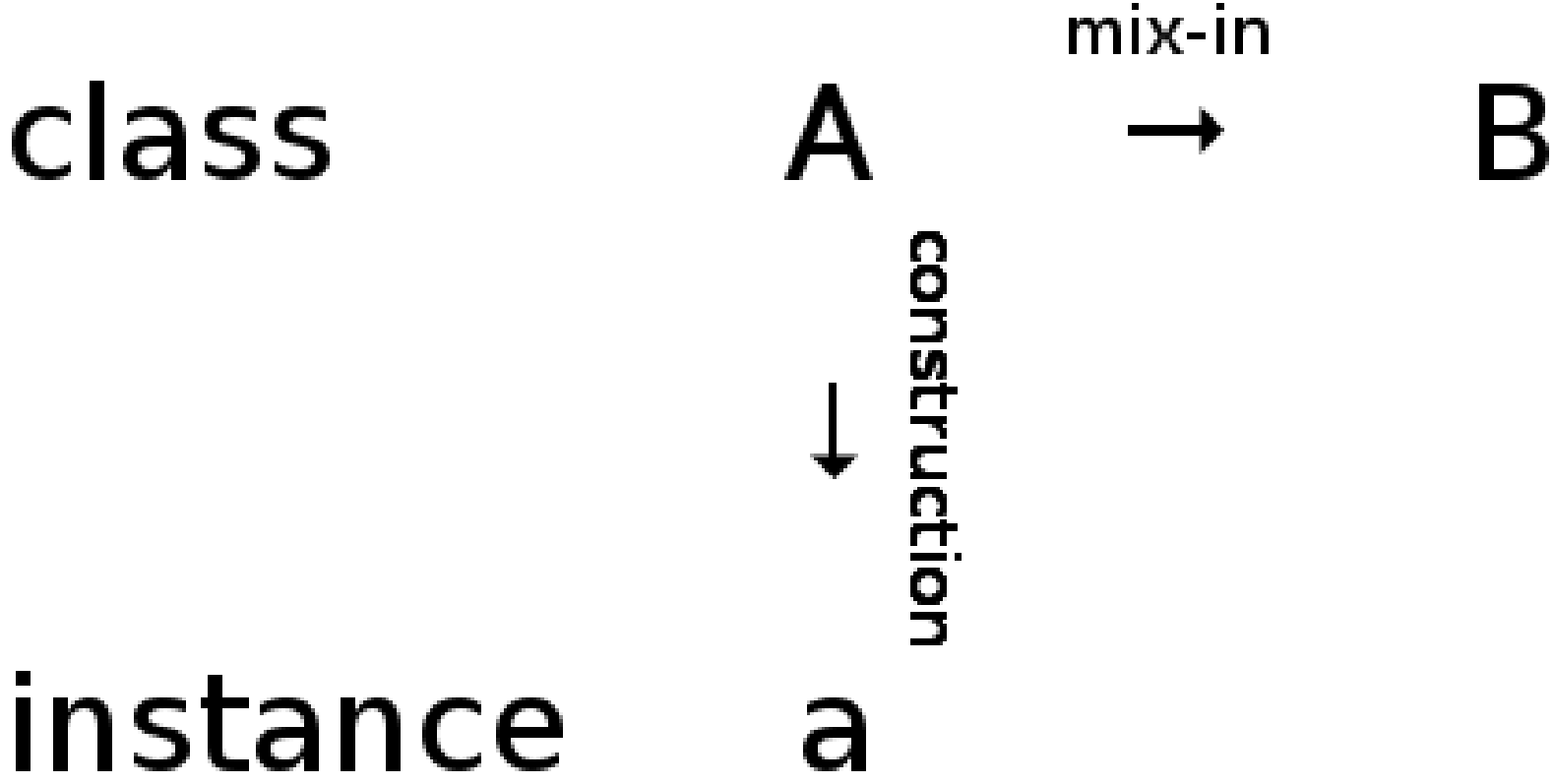

class
A
B
construction
instance
a

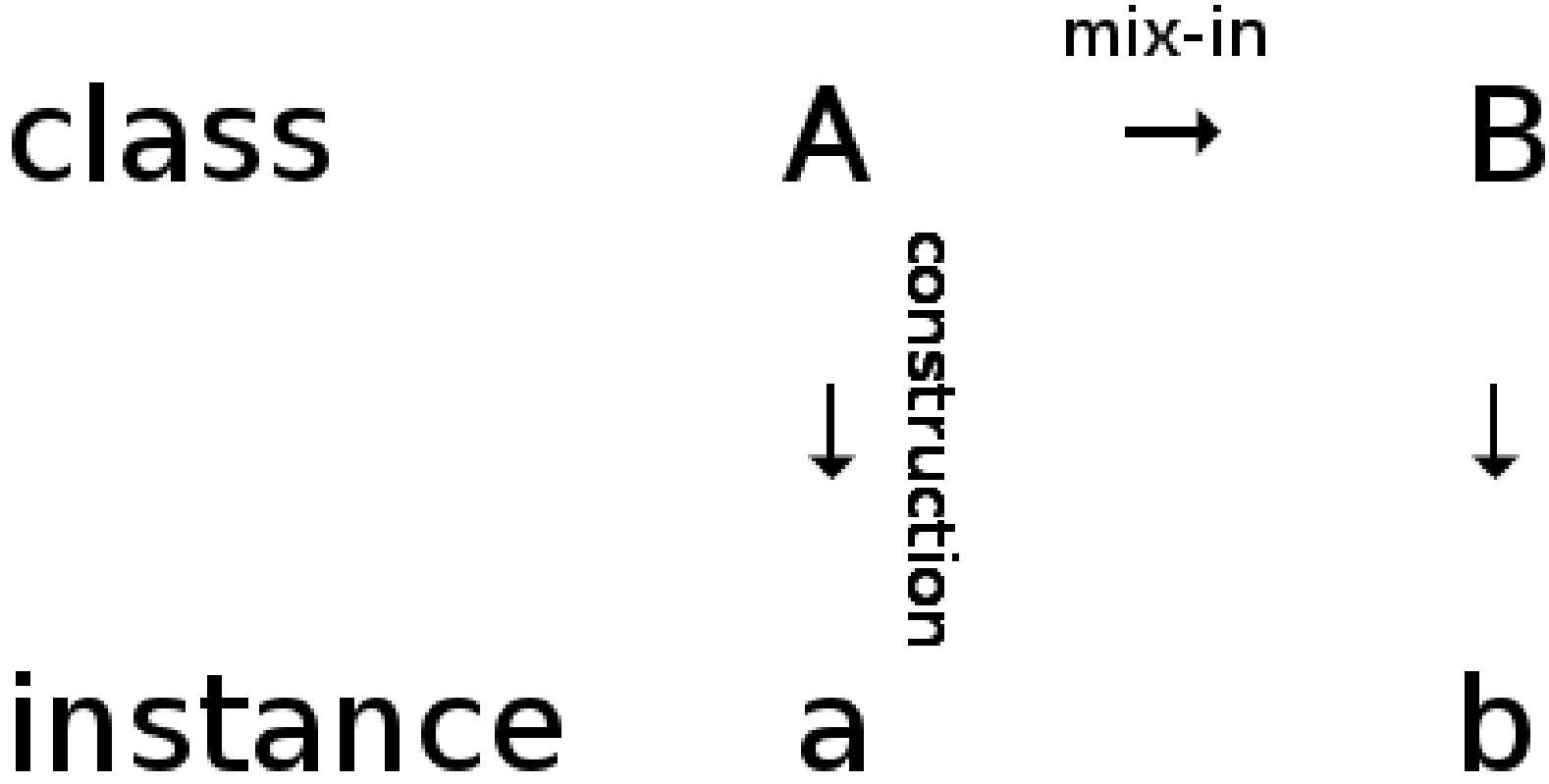
class



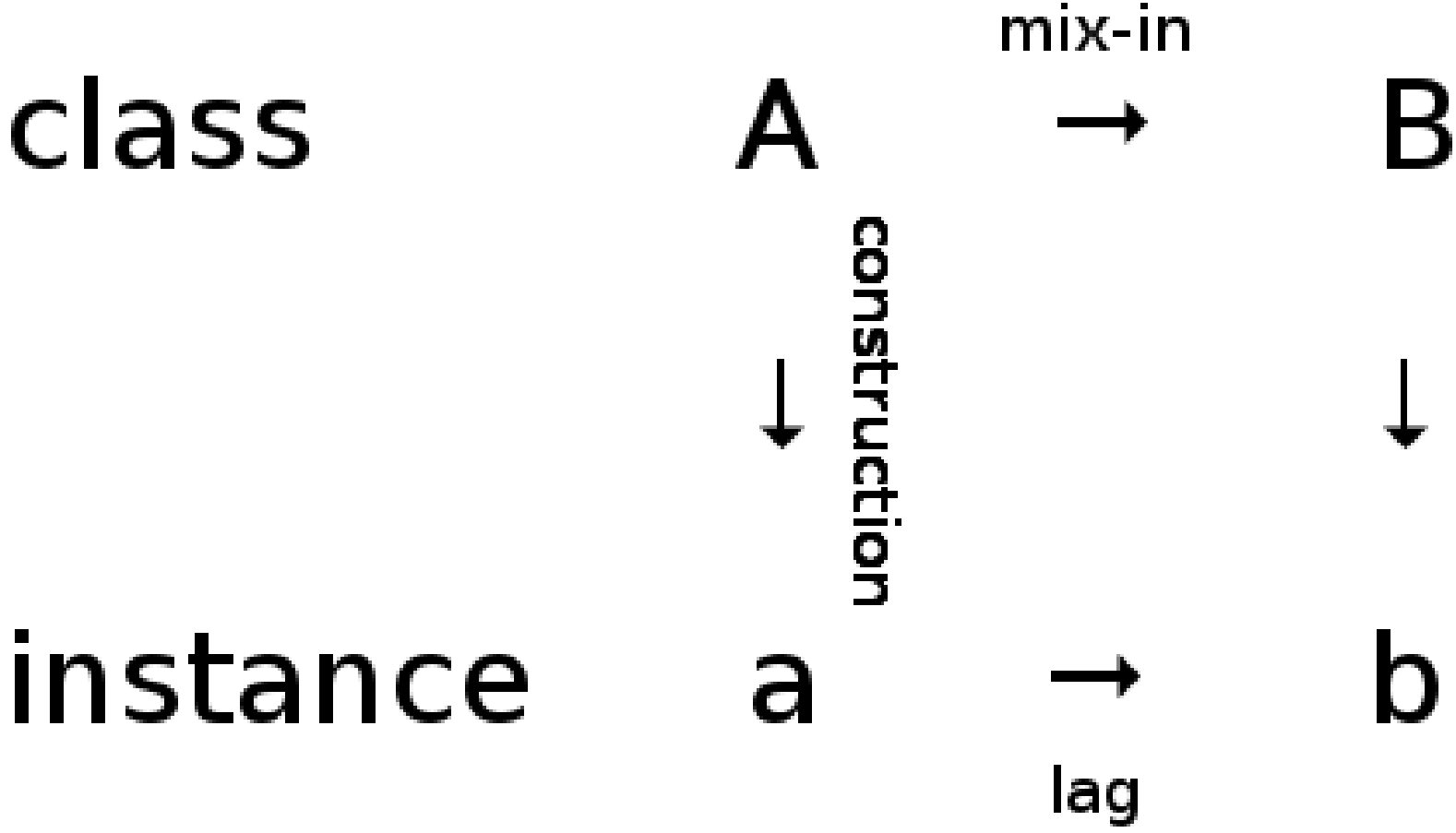
instance a


class
A
→
B
construction
instance
a


mix-in
class
A
B
construction
instance
a


mix-in
class
A
B
construction
instance
a
b

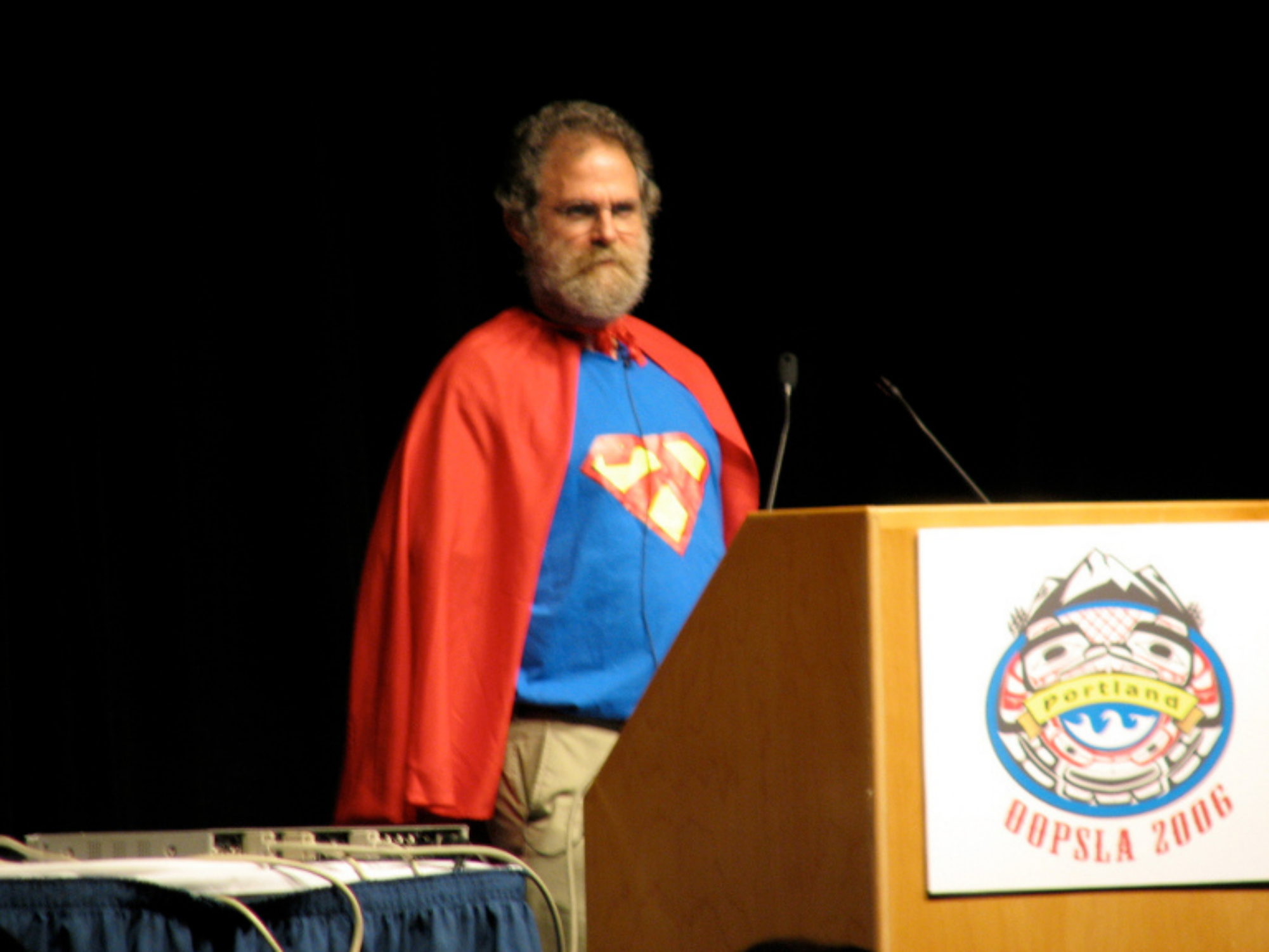

mix-in
class
A
B
construction
instance
a
b

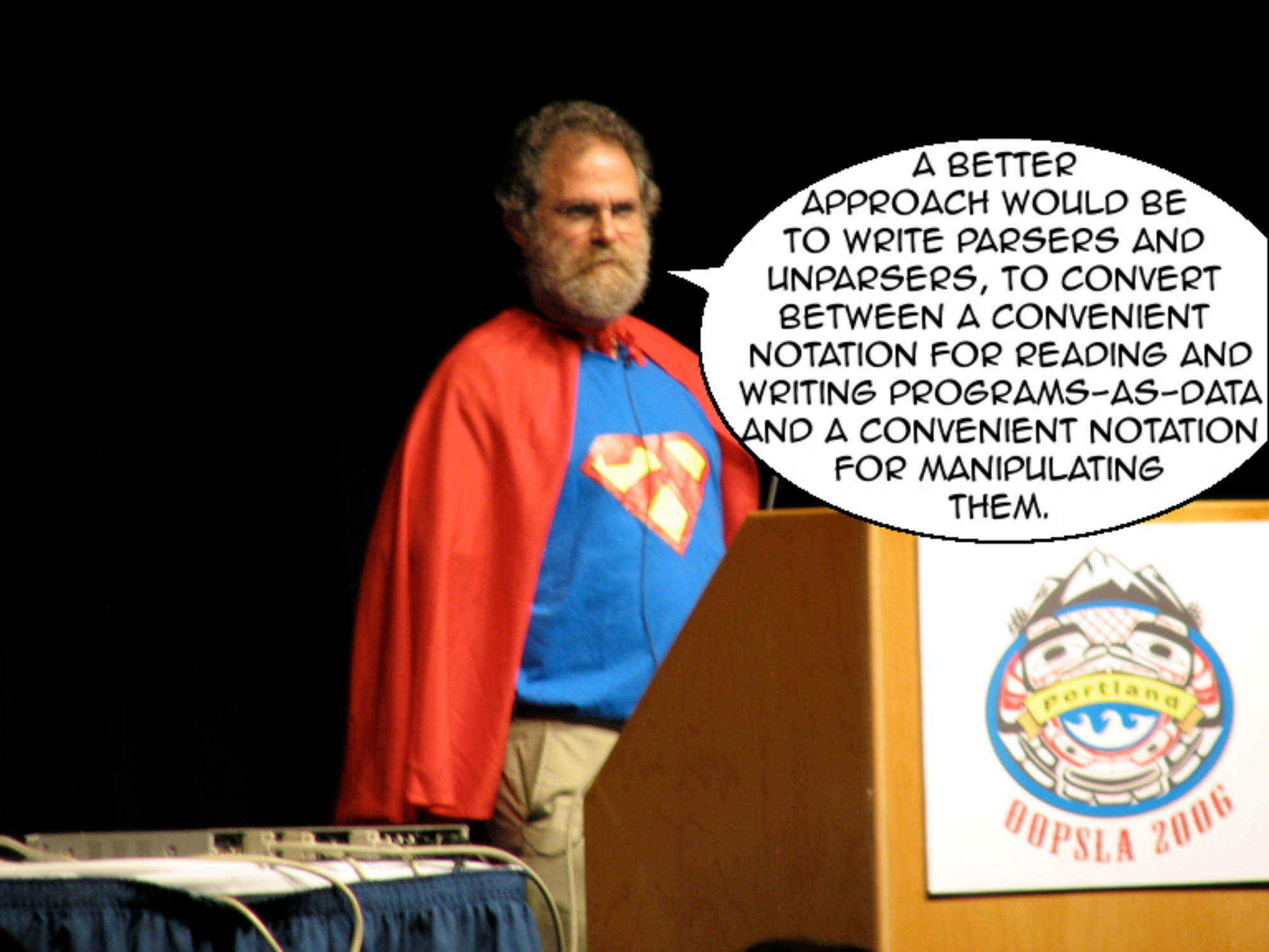

mix-in
class
A
B
construction
instance
a
b
lag

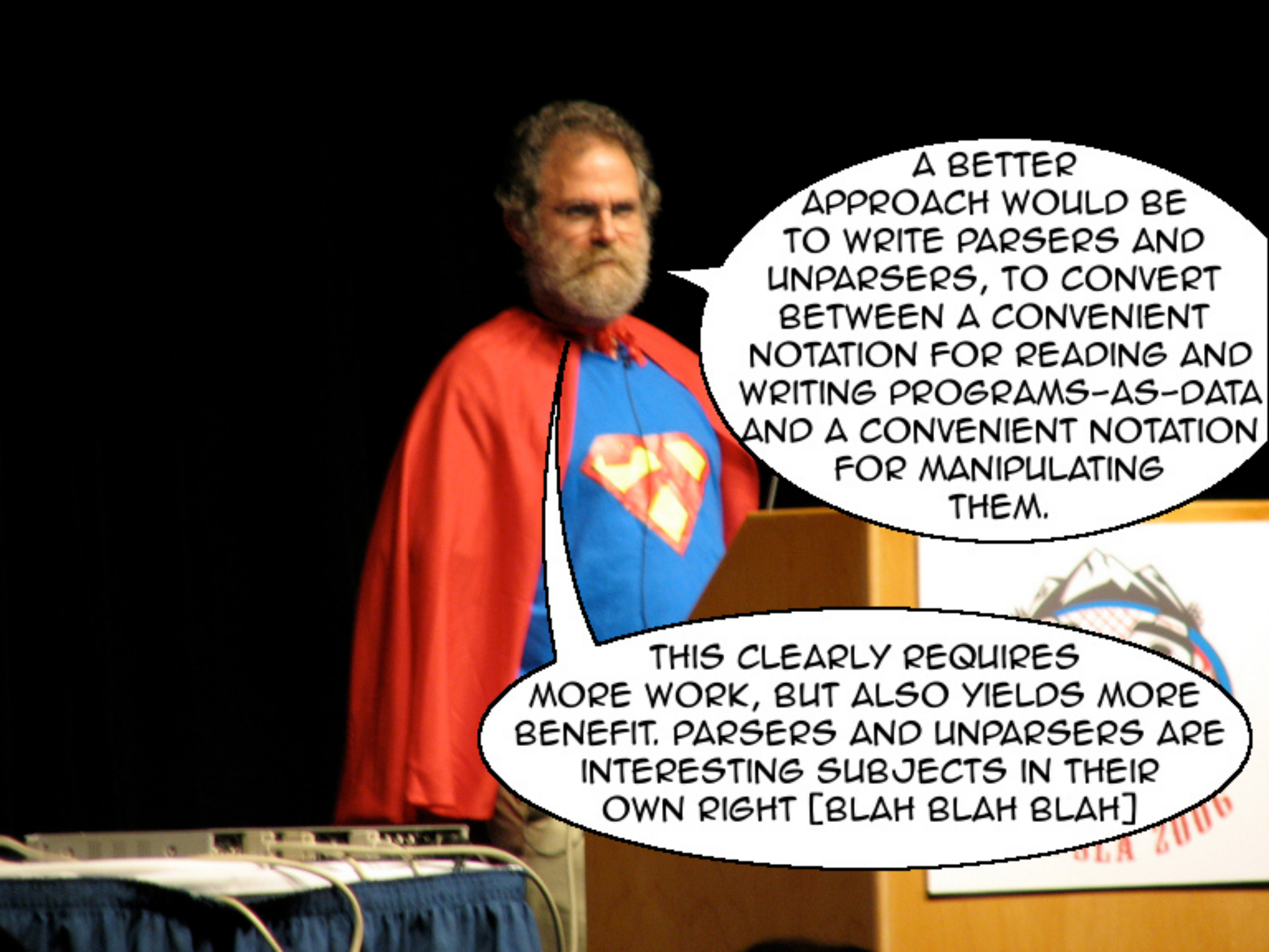
Lagging (live demo)

`4.rkt`

`https://github.com/panicz/sracket`


portland
OOPSLA 2006


A BETTER APPROACH WOULD BE TO WRITE PARSERS AND UNPARSERS, TO CONVERT BETWEEN A CONVENIENT NOTATION FOR READING AND WRITING PROGRAMS-AS-DATA AND A CONVENIENT NOTATION FOR MANIPULATING THEM.
portland
OOPSLA 2006


A BETTER APPROACH WOULD BE TO WRITE PARSERS AND UNPARSERS, TO CONVERT BETWEEN A CONVENIENT NOTATION FOR READING AND WRITING PROGRAMS-AS-DATA AND A CONVENIENT NOTATION FOR MANIPULATING THEM.
THIS CLEARLY REQUIRES MORE WORK, BUT ALSO YIELDS MORE BENEFIT. PARSERS AND UNPARSERS ARE INTERESTING SUBJECTS IN THEIR OWN RIGHT [BLAH BLAH BLAH]

Conversation with Jon Harrop

SICP is a seriously out of date book [...]

Conversation with Jon Harrop

SICP is a seriously out of date book [...]

I'm not sure whether parsing would actually bring any value to the topics that were covered by SICP. How would it be helpful to their logic circuit simulator, or to the picture language? How would it help with symbolic differentiation? How would they justify introduction of syntax? “Hey, listen, we have this straightforward way of representing expressions, and although we could get used to it and go on with our presentation of ideas, it looks a bit alien to us, so we will devote the next chapter to parsing, so that we could be writing it so that it would resemble things that we are more accustomed to”?

Conversation with Jon Harrop

SICP is a seriously out of date book [...]

I'm not sure whether parsing would actually bring any value to the topics that were covered by SICP. How would it be helpful to their logic circuit simulator, or to the picture language? How would it help with symbolic differentiation? How would they justify introduction of syntax? “Hey, listen, we have this straightforward way of representing expressions, and although we could get used to it and go on with our presentation of ideas, it looks a bit alien to us, so we will devote the next chapter to parsing, so that we could be writing it so that it would resemble things that we are more accustomed to”?

What a bizarre question. Syntax is defacto standard. Syntax is everywhere, in every major language. Rather than asking for justification for the inclusion of syntax you should be trying to justify its exclusion.

LIMB (live demo)

`5.rkt`

`https://github.com/panicz/sracket`