ThoughtWorks®

Theory for practice

BEYOND PARADIGMS

Understand language features,
use the right patterns.

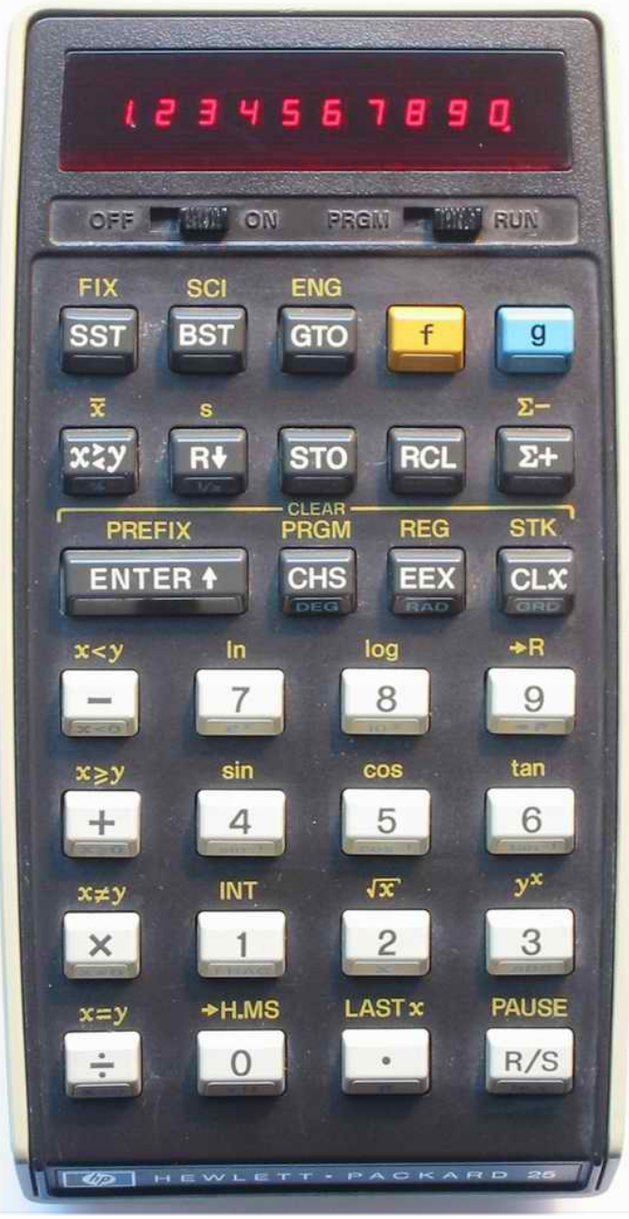
Luciano Ramalho
@ramalhoorg

ThoughtWorks®

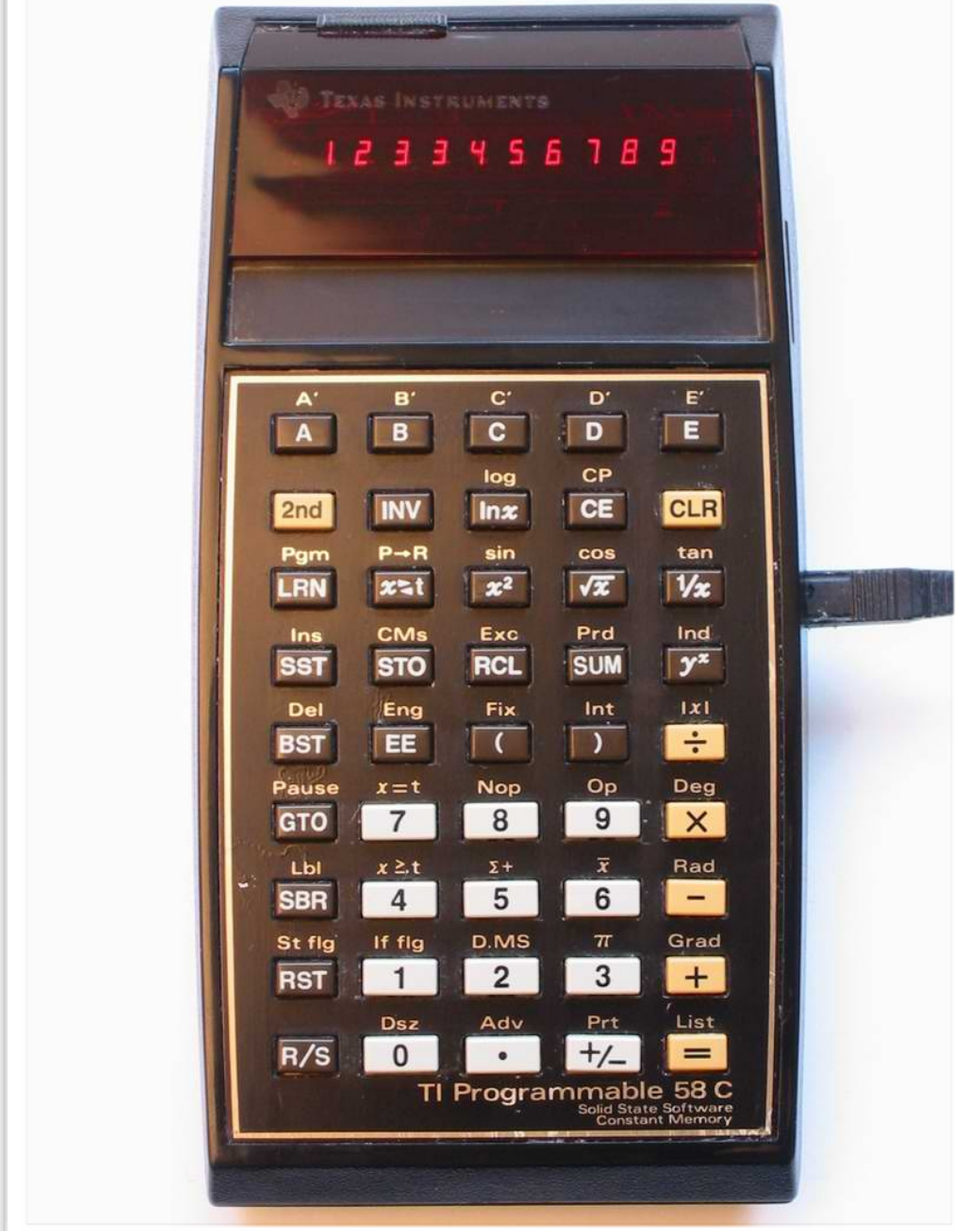
PARADIGMS

Programming language categories

CALCULATOR PROGRAMMING IS IMPERATIVE



HP-25

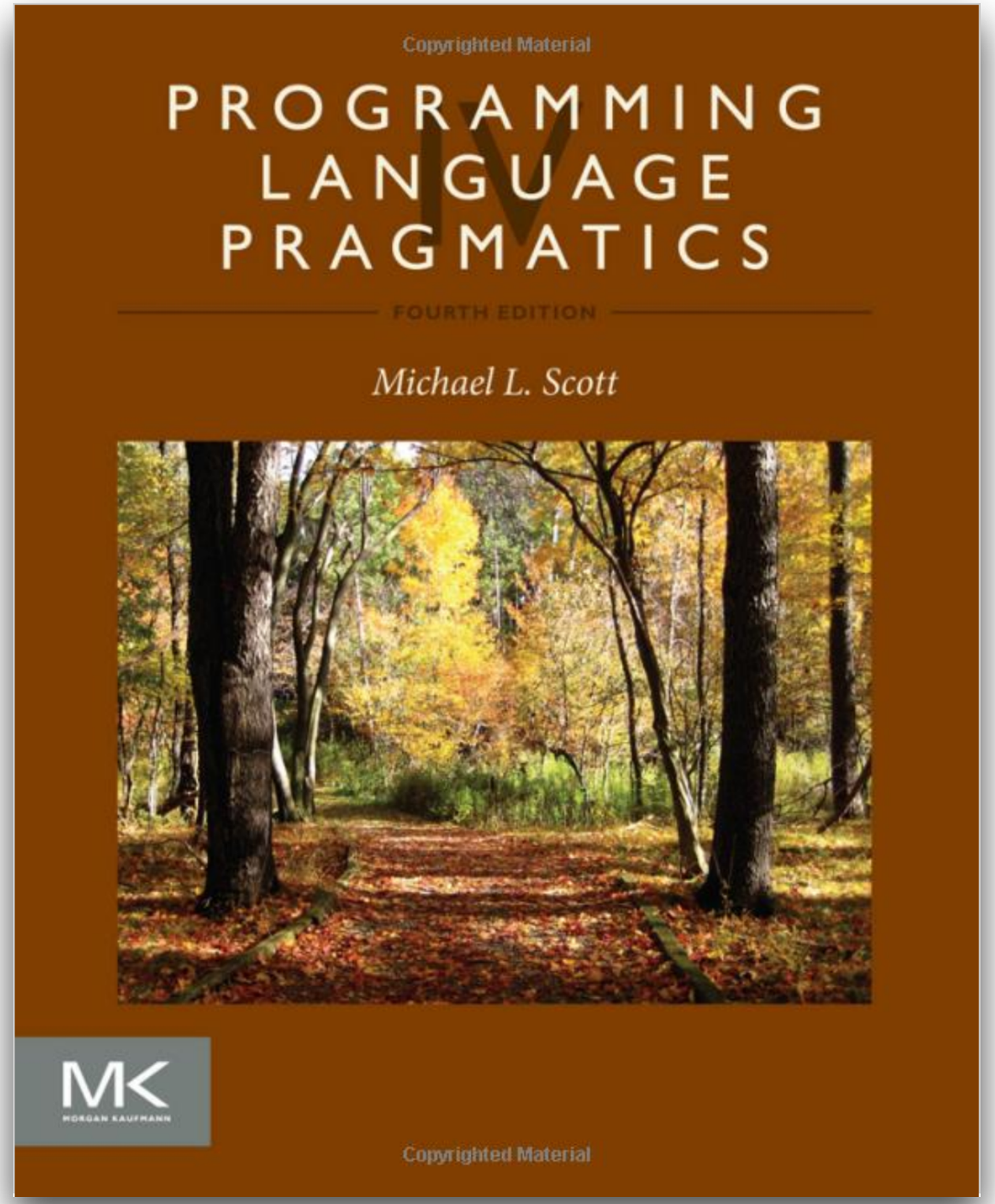


TI 58C

HP-25 CALCULATOR PROGRAMMING LANGUAGE

Visor		Intro-dução	X	Y	Z	T	
Linha	Código						
00							
01	14 11 04	f FIX 4					Apres
02	24 00	RCL 0	X				Apres
03	33	EEX	1. 00	X			
04	04	4	1. 04	X			
05	71	÷	$X/10^4$				Divid
06	24 01	RCL 1	V	$X/10^4$			
07	15 41	g x<0	V	$X/10^4$			V é n
08	13 11	GTO 11	V	$X/10^4$			Sim,
09	51	+	$V + X/10^4$				Não,
10	13 13	GTO 13	$V + X/10^4$				
11	21	x⇄y	$X/10^4$	V			V < 0
12	41	–	$V - X/10^4$				
13	74	R/S	V.X				V.X =
14	24 02	RCL 2	F	B			Quein
15	14 41	f x<y	F	B			Comb
16	13 34	GTO 34	F	B			Sim,
17	22	R↓	B			F	Não,
18	23 41 02	STO – 2	B			F	Subtra
19	05	5	5	B			Gravi

A SURVEY-STYLE PROGRAMMING LANGUAGES BOOK



Programming Language Pragmatics, 4th edition (2015)

Michael L. Scott

GCD ASM X86

Greatest common divisor in x86 Assembly

(Scott, 2015)

```
        pushl   %ebp                # \
        movl    %esp, %ebp          #  ) reserve space for local variables
        subl    $16, %esp           # /
        call    getint              # read
        movl    %eax, -8(%ebp)      # store i
        call    getint              # read
        movl    %eax, -12(%ebp)     # store j
A:      movl    -8(%ebp), %edi      # load i
        movl    -12(%ebp), %ebx     # load j
        cmpl    %ebx, %edi          # compare
        je      D                   # jump if i == j
        movl    -8(%ebp), %edi      # load i
        movl    -12(%ebp), %ebx     # load j
        cmpl    %ebx, %edi          # compare
        jle     B                   # jump if i < j
        movl    -8(%ebp), %edi      # load i
        movl    -12(%ebp), %ebx     # load j
        subl    %ebx, %edi          # i = i - j
        movl    %edi, -8(%ebp)      # store i
        jmp     C
B:      movl    -12(%ebp), %edi     # load j
        movl    -8(%ebp), %ebx      # load i
        subl    %ebx, %edi          # j = j - i
        movl    %edi, -12(%ebp)     # store j
C:      jmp     A
D:      movl    -8(%ebp), %ebx      # load i
        push    %ebx                # push i (pass to putint)
        call    putint              # write
        addl    $4, %esp            # pop i
        leave                       # deallocate space for local variables
        mov     $0, %eax            # exit status for program
        ret                         # return to operating system
```

Figure 1.7 Naive x86 assembly language for the GCD program.

GCD IN C, OCAML AND PROLOG

```c
int gcd(int a, int b) {                                    // C
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}
```

```
let rec gcd a b =                                          (* OCaml *)
    if a = b then a
    else if a > b then gcd b (a - b)
         else gcd a (b - a)
```

```
gcd(A,B,G) :- A = B, G = A.                                % Prolog
gcd(A,B,G) :- A > B, C is A-B, gcd(C,B,G).
gcd(A,B,G) :- B > A, C is B-A, gcd(C,A,G).
```

Figure 1.2 The GCD algorithm in C (top), OCaml (middle), and Prolog (bottom). All three versions assume (without checking) that their inputs are positive integers.

GCD IN PYTHON

```python
def gcd(a, b):
    assert a > 0 and b > 0
    while a != b:
        if a > b:
            a -= b
        else:
            b -= a
    return a
```

Imperative style

```python
def gcd(a, b):
    assert a > 0 and b > 0
    if a == b:
        return a
    elif a > b:
        return gcd(b, a - b)
    else:
        return gcd(a, b - a)
```

Functional style

GCD IN PYTHON

Bad fit for Python: no tail-call optimisation

```python
def gcd(a, b):
    assert a > 0 and b > 0
    while a != b:
        if a > b:
            a -= b
        else:
            b -= a
    return a
```

Imperative style

```python
def gcd(a, b):
    assert a > 0 and b > 0
    if a == b:
        return a
    elif a > b:
        return gcd(b, a - b)
    else:
        return gcd(a, b - a)
```

Functional style

ONE CLASSIFICATION

1.2 The Programming Language Spectrum

Example 1.3

Classification of programming languages

The many existing languages can be classified into families based on their model of computation. Figure 1.1 shows a common set of families. The top-level division distinguishes between the *declarative* languages, in which the focus is on *what* the computer is to do, and the *imperative* languages, in which the focus is on *how* the computer should do it.■

declarative	
functional	Lisp/Scheme, ML, Haskell
dataflow	Id, Val
logic, constraint-based	Prolog, spreadsheets, SQL
imperative	
von Neumann	C, Ada, Fortran, …
object-oriented	Smalltalk, Eiffel, Java, …
scripting	Perl, Python, PHP, …

FIGURE 1.1 Classification of programming languages. Note that the categories are fuzzy and open to debate. In particular, it is possible for a functional language to be object-oriented, and many authors do not consider functional programming to be declarative.

Programming Language Pragmatics, 4th edition (2015)

Michael L. Scott

ONE CLASSIFICATION (ZOOM)

declarative	
functional	Lisp/Scheme, ML, Haskell
dataflow	Id, Val
logic, constraint-based	Prolog, spreadsheets, SQL
imperative	
von Neumann	C, Ada, Fortran, …
object-oriented	Smalltalk, Eiffel, Java, …
scripting	Perl, Python, PHP, …

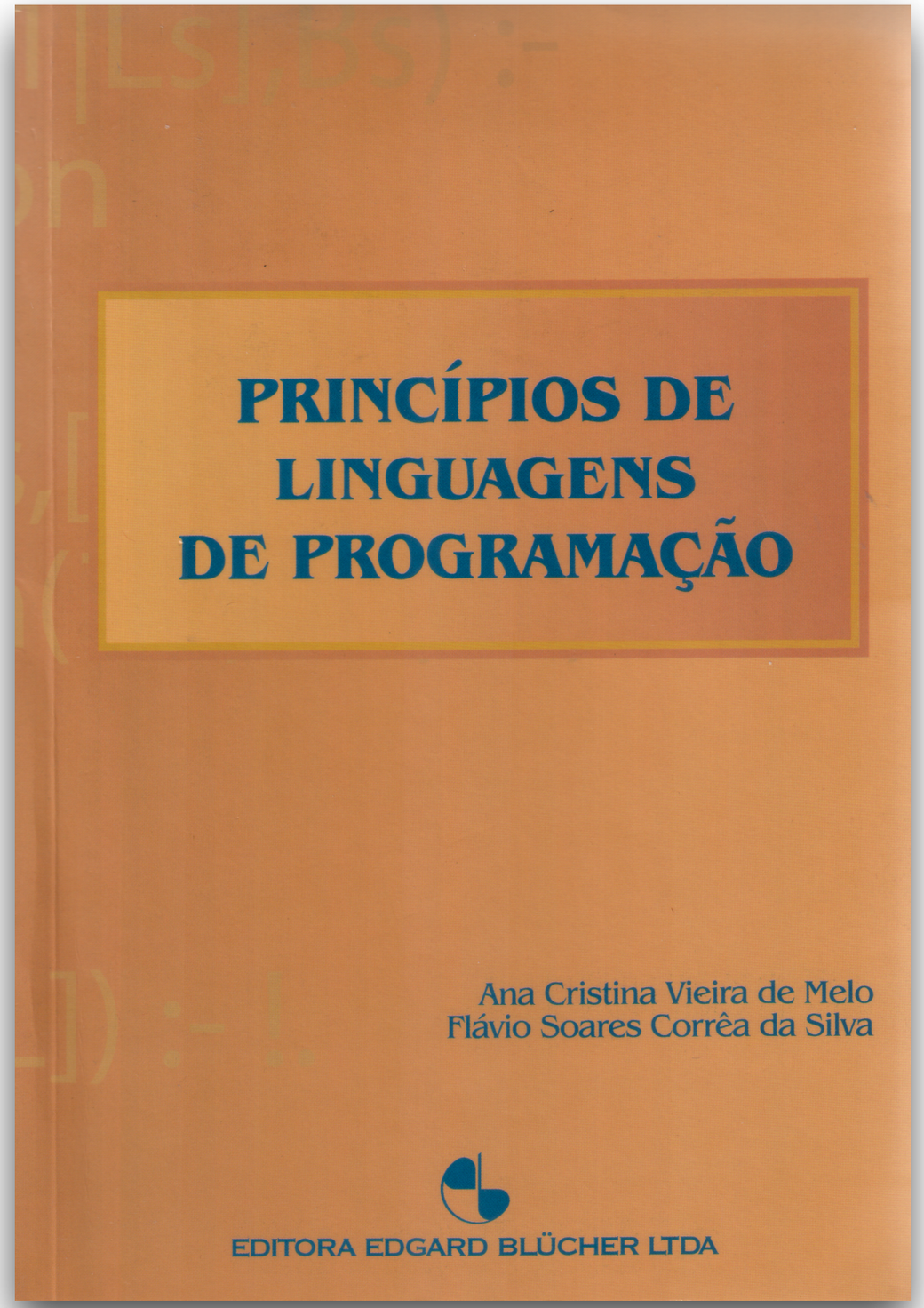
FIGURE 1.1 Classification of programming languages. Note that the categories are fuzzy and open to debate. In particular, it is possible for a functional language to be object-oriented, and many authors do not consider functional programming to be declarative.

ONE CLASSIFICATION (ZOOM)

declarative	
functional	Lisp/Scheme, ML, Haskell
dataflow	Id, Val
logic, constraint-based	Prolog, spreadsheets, SQL
imperative	
von Neumann	C, Ada, Fortran, …
object-oriented	Smalltalk, Eiffel, Java, …
scripting ???	Perl, Python, PHP, …

FIGURE 1.1 Classification of programming languages. Note that the categories are fuzzy and open to debate. In particular, it is possible for a functional language to be object-oriented, and many authors do not consider functional programming to be declarative.

ANOTHER BOOK



Princípios de Linguagens de Programação
(2003)

Ana Cristina Vieira de Melo

Flávio Soares Corrêa da Silva

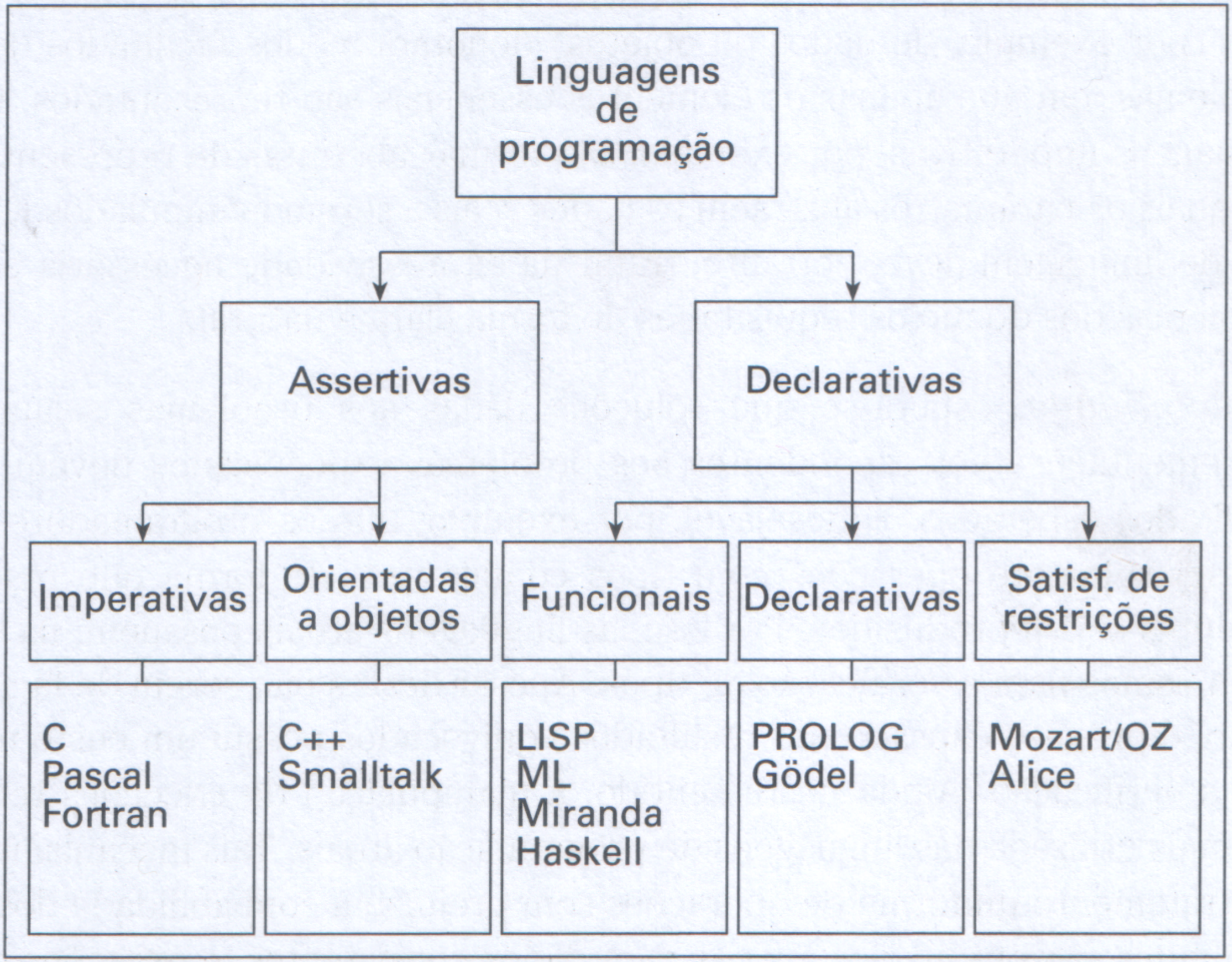


Figura 1 — Tipologia de linguagens de programação.

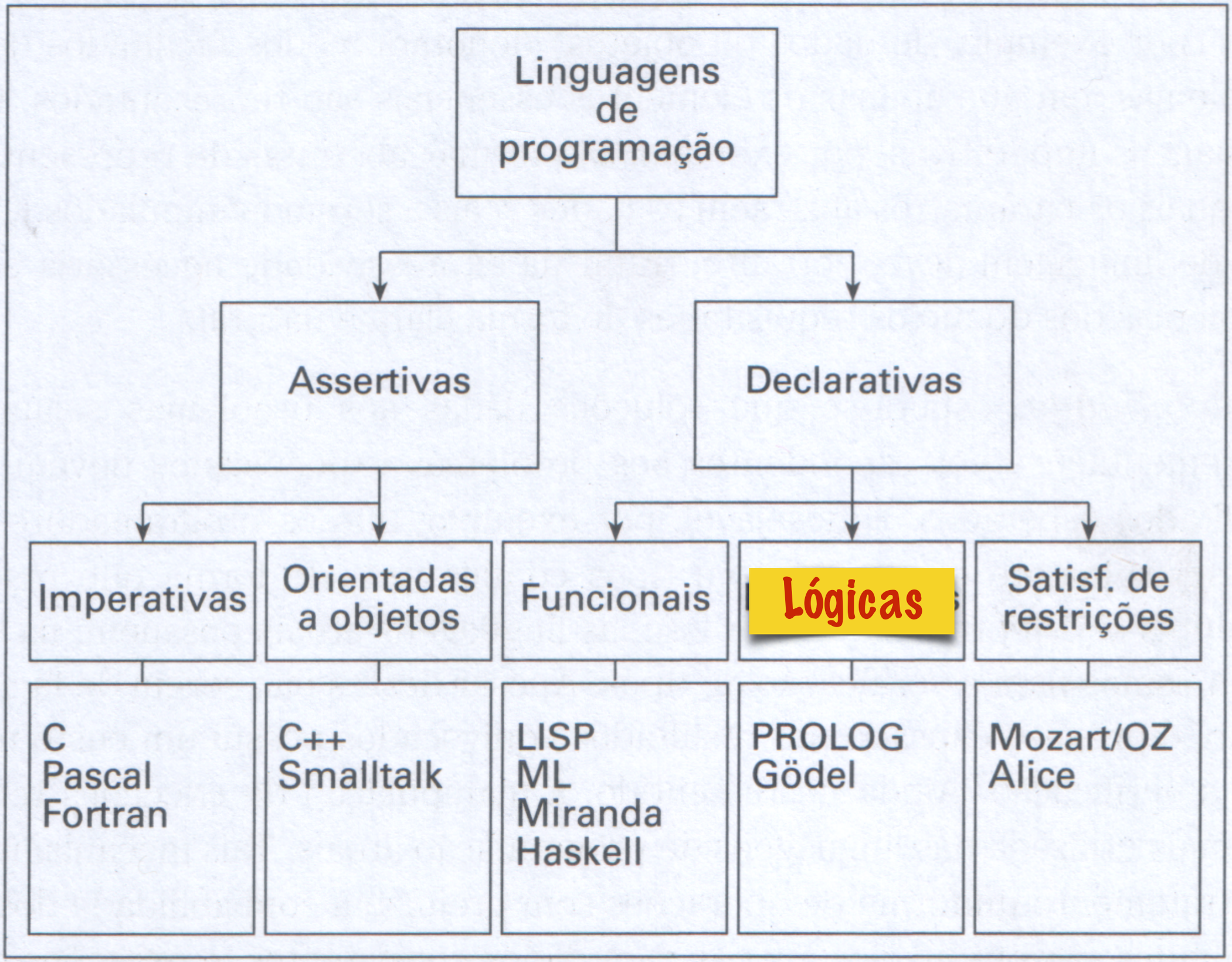


Figura 1 — Tipologia de linguagens de programação.

THE LANGUAGE LIST

Firefox File Edit View History Bookmarks Tools Window Help 100% Wed 19:56

The Language List

www.info.univ-angers.fr/~gh/hilapr/langlist/langlist.htm 170% Search

The Language List

Collected Information On About 2500 Computer Languages, Past and Present.

Maintained by Bill Kinnersley

Welcome to The Language List! Early versions of this list were posted to comp.lang.misc beginning in 1991. Now a web site, our intention remains the same -- to become one of the most complete sources of information on computer programming languages ever assembled (or compiled :-).

The list does not pretend to be a definitive scholarly work. Its purpose is to collect and provide timely information in a rapidly growing field. Its accuracy and completeness depend to a great extent on the users of the Internet. If you know about a language that should be added, please share your knowledge.

Start a Search

LANGUAGES ARE MISSING…

Search the Language List

www.info.univ-angers.fr/~gh/hilapr/langlist/search.htm 170% Search

Search for a particular language entry:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Home

Not Found

The requested URL /~nkinners/LangList/Indexes/gindex.htm was not found on this server.

Apache/2.2.15 (Red Hat) Server at people.ku.edu Port 80

LANGUAGE CATEGORIES

Language Categories

Procedural Language

A language which states how to compute the result of a given problem. This term encompasses both imperative and functional languages.

Imperative Language

A language which operates by a sequence of commands that change the value of data elements. Imperative languages are typified by assignments and iteration.

Declarative Language

A language which operates by making descriptive statements about data, and relations between data. The algorithm is hidden in the semantics of the language. This category encompasses both applicative and logic languages. Examples of declarative features are set comprehensions and pattern-matching statements.

Applicative Language

A language that operates by application of functions to values, with no side effects. A functional language in the broad sense.

Functional Language

In the narrow sense, a functional language is one that operates by use of higher-order functions, building operators that manipulate functions directly without ever appearing to manipulate data. Example: FP.

LANGUAGE CATEGORIES (2)

Constraint Language

A language in which a problem is specified and solved by a series of constraining relationships.

Object-Oriented Language

A language in which data and the functions which access it are treated as a unit.

Concurrent Language

A concurrent language describes programs that may be executed in parallel. This may be either

- Multiprogramming: sharing one processor
- Multiprocessing: separate processors sharing one memory
- Distributed

Concurrent languages differ in the way that processes are created:

- Coroutines - control is explicitly transferred - examples are Simula I, SL5, BLISS and Modula-2.
- Fork/join - examples are PL/I and Mesa.
- Cobegin/coend - examples are ALGOL 68, CSP, Edison, Argus.
- Process declarations - examples are DP, SR, Concurrent Pascal, Modula, PLITS and Ada.

and the ways in which processes interact:

- Semaphores - ALGOL 68
- Conditional critical regions - Edison, DP, Argus
- Monitors - Concurrent Pascal, Modula
- Message passing - CSP, PLITS, Gypsy, Actors
- Remote procedure calls - DP, *Mod
    - Rendezvous - Ada, SR

LANGUAGE CATEGORIES (3)

- Message passing - CSP, PLITS, Gypsy, Actors
- Remote procedure calls - DP, *Mod
  - Rendezvous - Ada, SR
  - Atomic transactions - Argus

Fourth Generation Language (4GL)

A very high-level language. It may use natural English or visual constructs. Algorithms or data structures may be selected by the compiler.

Query Language

An interface to a database.

Specification Language

A formalism for expressing a hardware or software design.

Assembly Language

A symbolic representation of the machine language of a specific computer.

Intermediate Language

A language used as an intermediate stage in compilation. May be either text or binary.

Metalanguage

A language used for the formal description of another language.

LANGUAGE CATEGORIES (4)

Definitional Language

An applicative language containing assignments interpreted as definitions. Example: Lucid.

Single Assignment Language

An applicative language using assignments, with the convention that a variable may appear on the left side of an assignment only once within the portion of the program in which it is active.

Dataflow Language

A language suitable for use on a dataflow architecture. Necessary properties include freedom from side effects, and the equivalence of scheduling constraints with data dependencies. Examples: Val, Id, SISAL, Lucid.

Logic Language

A logic language deals with predicates or relationships p(X,Y). A program consists of a set of Horn clauses which may be:

- facts - p(X,Y) is true
- rules - p is true if q1 and q2 and ... qn are true
- queries - is g1 and g2 and ... gn true? (gi's are the goals.)

Further clauses are inferred using resolution. One clause is selected containing p as an assumption, another containing p as a consequence, and p is eliminated between them. If the two p's have different arguments they must be unified, using the substitution with the fewest constraints that makes them the same. Logic languages try alternative resolutions for each goal in succession, backtracking in a search for a common solution.

- OR-parallel logic languages try alternative resolutions in parallel
- AND-parallel logic languages try to satisfy several goals in parallel.

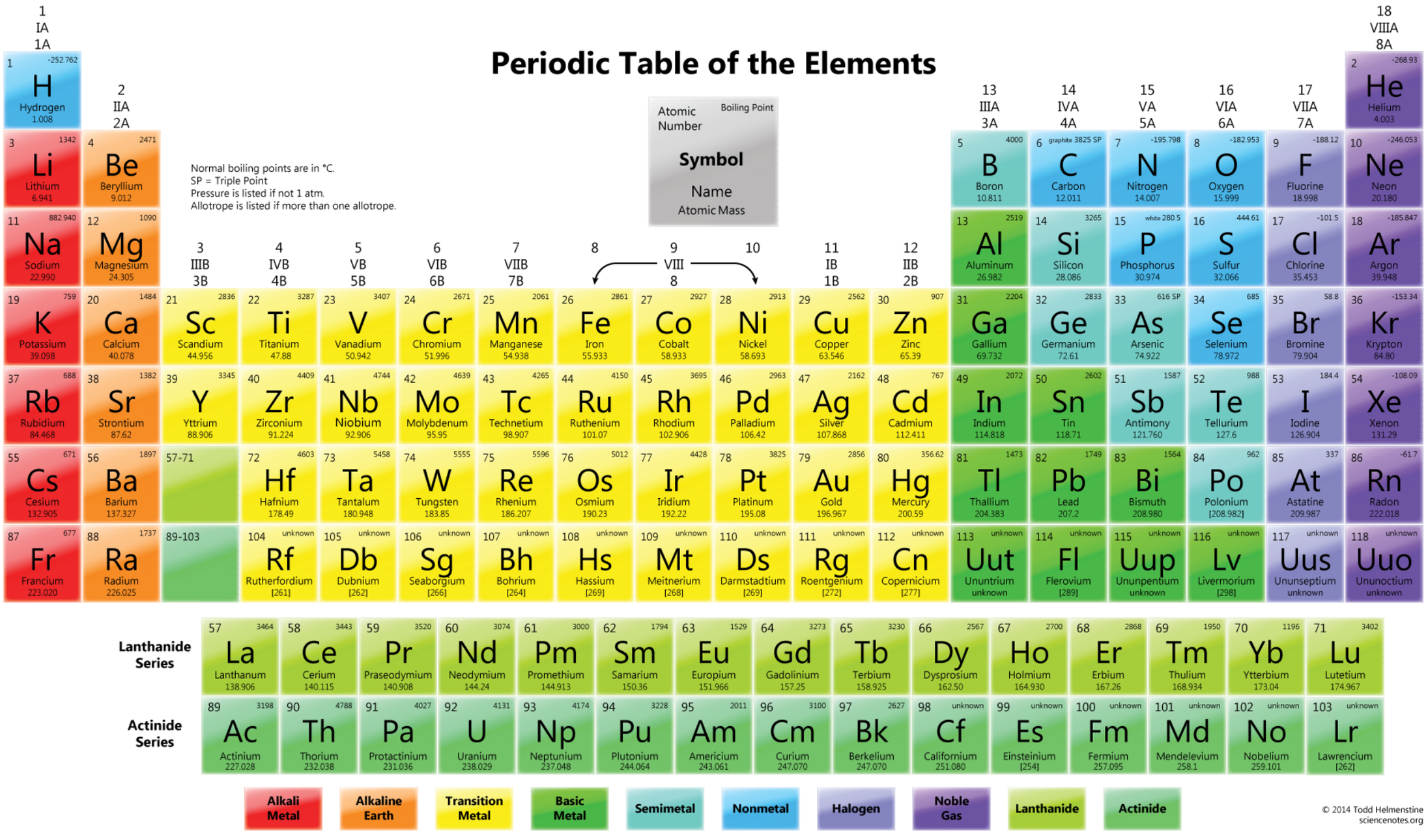
Constraint Language

ThoughtWorks®

CATEGORIES?

Ontologies are so 1900's

A CLASSIFICATION BASED ON HARD FACTS



A CLASSIFICATION BASED ON HARD FACTS?

"Noble" gases!?

Periodic Table of the Elements

Normal boiling points are in °C.
SP = Triple Point
Pressure is listed if not 1 atm.
Allotrope is listed if more than one allotrope.

Atomic Number | Boiling Point
Symbol
Name
Atomic Mass

1 IA 1A	2 IIA 2A	3 IIIB 3B	4 IVB 4B	5 VB 5B	6 VIB 6B	7 VIIB 7B	8	9 VIII 8	10	11 IB 1B	12 IIB 2B	13 IIIA 3A	14 IVA 4A	15 VA 5A	16 VIA 6A	17 VIIA 7A	18 VIIIA 8A
1 -252.762 H Hydrogen 1.008																	2 -268.93 He Helium 4.003
3 1342 Li Lithium 6.941	4 2471 Be Beryllium 9.012											5 4000 B Boron 10.811	6 graphite 3825 SP C Carbon 12.011	7 -195.798 N Nitrogen 14.007	8 -182.953 O Oxygen 15.999	9 F Fluorine 18.998	10 -246.053 Ne Neon 20.180
11 882.940 Na Sodium 22.990	12 1090 Mg Magnesium 24.305											13 2519 Al Aluminum 26.982	14 3265 Si Silicon 28.086	15 white 280.5 P Phosphorus 30.974	16 444.61 S Sulfur 32.066	17 Cl Chlorine 35.453	18 -185.847 Ar Argon 39.948
19 759 K Potassium 39.098	20 1484 Ca Calcium 40.078	21 2836 Sc Scandium 44.956	22 3287 Ti Titanium 47.88	23 3407 V Vanadium 50.942	24 2671 Cr Chromium 51.996	25 2061 Mn Manganese 54.938	26 2861 Fe Iron 55.933	27 2927 Co Cobalt 58.933	28 2913 Ni Nickel 58.693	29 2562 Cu Copper 63.546	30 907 Zn Zinc 65.39	31 2204 Ga Gallium 69.732	32 2833 Ge Germanium 72.61	33 616 SP As Arsenic 74.922	34 685 Se Selenium 78.972	35 Br Bromine 79.904	36 -153.34 Kr Krypton 84.80
37 688 Rb Rubidium 84.468	38 1382 Sr Strontium 87.62	39 3345 Y Yttrium 88.906	40 4409 Zr Zirconium 91.224	41 4744 Nb Niobium 92.906	42 4639 Mo Molybdenum 95.95	43 4265 Tc Technetium 98.907	44 4150 Ru Ruthenium 101.07	45 3695 Rh Rhodium 102.906	46 2963 Pd Palladium 106.42	47 2162 Ag Silver 107.868	48 767 Cd Cadmium 112.411	49 2072 In Indium 114.818	50 2602 Sn Tin 118.71	51 1587 Sb Antimony 121.760	52 988 Te Tellurium 127.6	53 I Iodine 126.904	54 -108.09 Xe Xenon 131.29
55 671 Cs Cesium 132.905	56 1897 Ba Barium 137.327	57-71	72 4603 Hf Hafnium 178.49	73 5458 Ta Tantalum 180.948	74 5555 W Tungsten 183.85	75 5596 Re Rhenium 186.207	76 5012 Os Osmium 190.23	77 4428 Ir Iridium 192.22	78 3825 Pt Platinum 195.08	79 2856 Au Gold 196.967	80 356.62 Hg Mercury 200.59	81 1473 Tl Thallium 204.383	82 1749 Pb Lead 207.2	83 1564 Bi Bismuth 208.980	84 962 Po Polonium [208.982]	85 At Astatine 209.987	86 -61.7 Rn Radon 222.018
87 677 Fr Francium 223.020	88 1737 Ra Radium 226.025	89-103	104 unknown Rf Rutherfordium [261]	105 unknown Db Dubnium [262]	106 unknown Sg Seaborgium [266]	107 unknown Bh Bohrium [264]	108 unknown Hs Hassium [269]	109 unknown Mt Meitnerium [268]	110 unknown Ds Darmstadtium [269]	111 unknown Rg Roentgenium [272]	112 unknown Cn Copernicium [277]	113 unknown Uut Ununtrium unknown	114 unknown Fl Flerovium [289]	115 unknown Uup Ununpentium unknown	116 unknown Lv Livermorium [298]	117 Uus Ununseptium unknown	118 unknown Uuo Ununoctium unknown

Lanthanide Series	57 3464 La Lanthanum 138.906	58 3443 Ce Cerium 140.115	59 3520 Pr Praseodymium 140.908	60 3074 Nd Neodymium 144.24	61 3000 Pm Promethium 144.913	62 1794 Sm Samarium 150.36	63 1529 Eu Europium 151.966	64 3273 Gd Gadolinium 157.25	65 3230 Tb Terbium 158.925	66 2567 Dy Dysprosium 162.50	67 2700 Ho Holmium 164.930	68 2868 Er Erbium 167.26	69 1950 Tm Thulium 168.934	70 1196 Yb Ytterbium 173.04	71 3402 Lu Lutetium 174.967
Actinide Series	89 3198 Ac Actinium 227.028	90 4788 Th Thorium 232.038	91 4027 Pa Protactinium 231.036	92 4131 U Uranium 238.029	93 4174 Np Neptunium 237.048	94 3228 Pu Plutonium 244.064	95 2011 Am Americium 243.061	96 3100 Cm Curium 247.070	97 2627 Bk Berkelium 247.070	98 unknown Cf Californium 251.080	99 unknown Es Einsteinium [254]	100 unknown Fm Fermium 257.095	101 unknown Md Mendelevium 258.1	102 unknown No Nobelium 259.101	103 unknown Lr Lawrencium [262]

Alkali Metal | Alkaline Earth | Transition Metal | Basic Metal | Semimetal | Nonmetal | Halogen | Noble Gas | Lanthanide | Actinide

“Ontology is overrated.”

Clay Shirky

ThoughtWorks®

A BETTER APPROACH

Fundamental Features of Programming Languages

TEACHING PROGRAMMING LANGUAGE THEORY

Teaching Programming Langua ×

cs.brown.edu/~sk/Publications/Papers/Published/sk-teach-pl-post-linnaean/

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

Abstract

Programming language "paradigms" are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them? This paper argues that we should abandon this method of teaching languages, offers an alternative, reconciles an important split in programming language education, and describes a textbook that explores these matters.

Comment

The book discussed in this paper is available here.

Paper

PDF

A PAPER PRESENTING THE APPROACH

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

Abstract

Programming language "paradigms" are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them? This paper argues that we should abandon this method of teaching languages, offers an alternative, reconciles an important split in programming language education, and describes a textbook that explores these matters.

A PAPER PRESENTING THE APPROACH

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

Abstract

Programming language "paradigms" are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them? This paper argues that we should abandon this method of teaching languages, offers an alternative, reconciles an important split in programming language education, and describes a textbook that explores these matters.

A PAPER PRESENTING THE APPROACH

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

Abstract

Programming language "paradigms" are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them? This paper argues that we should abandon this method of teaching languages, offers an alternative, reconciles an important split in programming language education, and describes a textbook that explores these matters.

A PAPER PRESENTING THE APPROACH

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

Abstract

Programming language "paradigms" are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them? This paper argues that we should abandon this method of teaching languages, offers an alternative, reconciles an important split in programming language education, and describes a textbook that explores these matters.

THEORY IN PRACTICE WITH RACKET (A SCHEME DIALECT)

Programming Languages: Appl ×

cs.brown.edu/courses/cs173/2012/book/

On this page:

Version Second Edition

Programming Languages: Application and Interpretation

by Shriram Krishnamurthi

THEORY IN PRACTICE WITH PASCAL (1990)

Programming Languages: Sam | Goodreads | Programming Lan | site:norvig.com kamin - Google

Secure https://www.goodreads.com/book/show/2082799.Programming_Languages?from_search=true

goodreads Home My Books Browse Community Search books

Read

My rating:

Programming Languages: An Interpreter-Based Approach

by Samuel N. Kamin

5.0 · Rating details · 1 Rating · 0 Reviews

GET A COPY

Amazon BR | Online Stores | Book Links

Hardcover, 640 pages
Published January 1st 1990 by Addison Wesley Publishing Company
More Details... edit details

Recommend It | Stats | Recent Status Updates

FRIEND REVIEWS

Recommend This Book None of your friends have reviewed this book yet.

READER Q&A

Ask the Goodreads community a question about Programming Languages

Ask anything about the book

Be the first to ask a question about Programming Languages

LISTS WITH THIS BOOK

GENRES

Science > Computer Science 1 user

See top shelves...

BOOKS BY SAMUEL N. KAMIN

More...

ThoughtWorks®

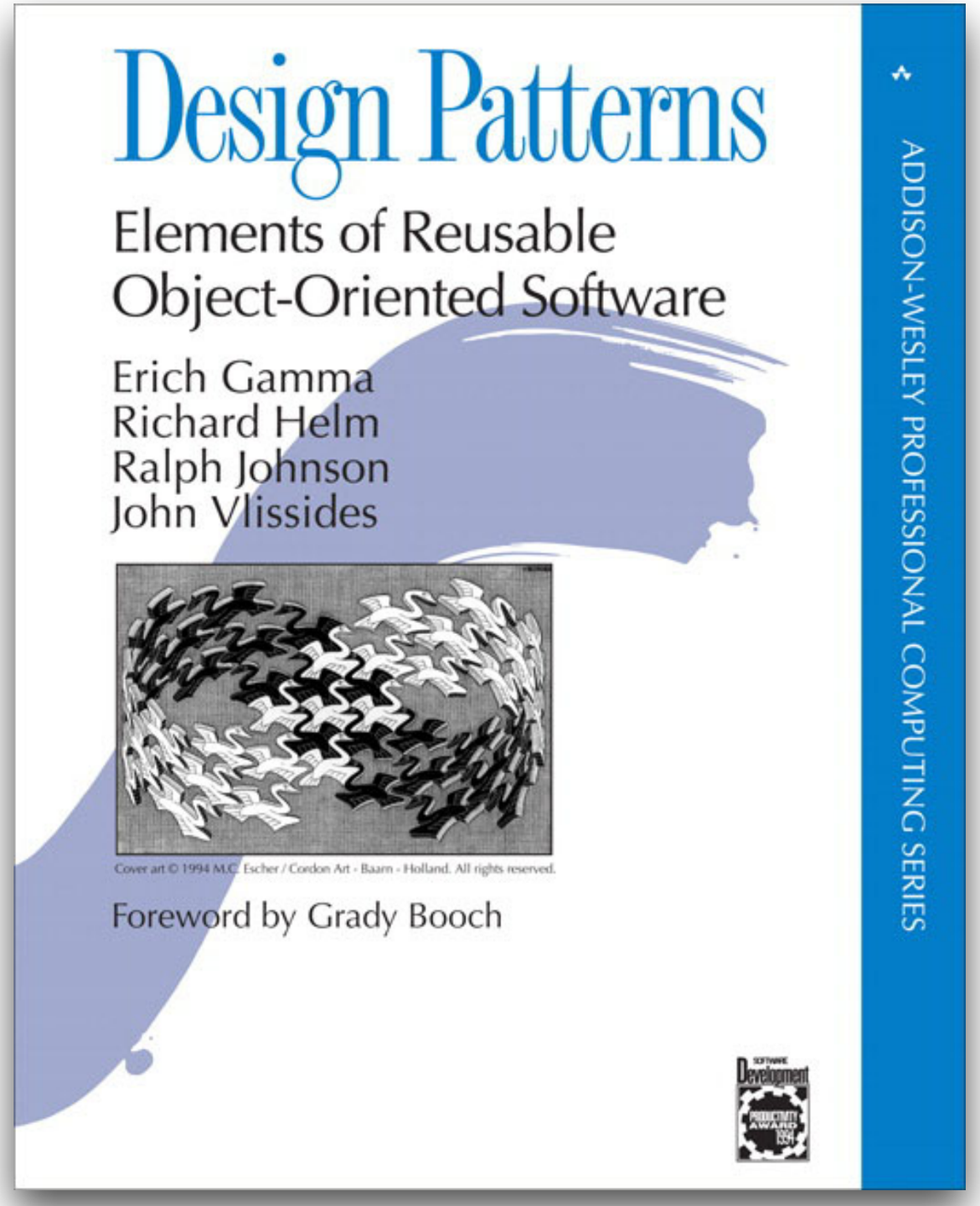
DESIGN PATTERNS

When languages fall short

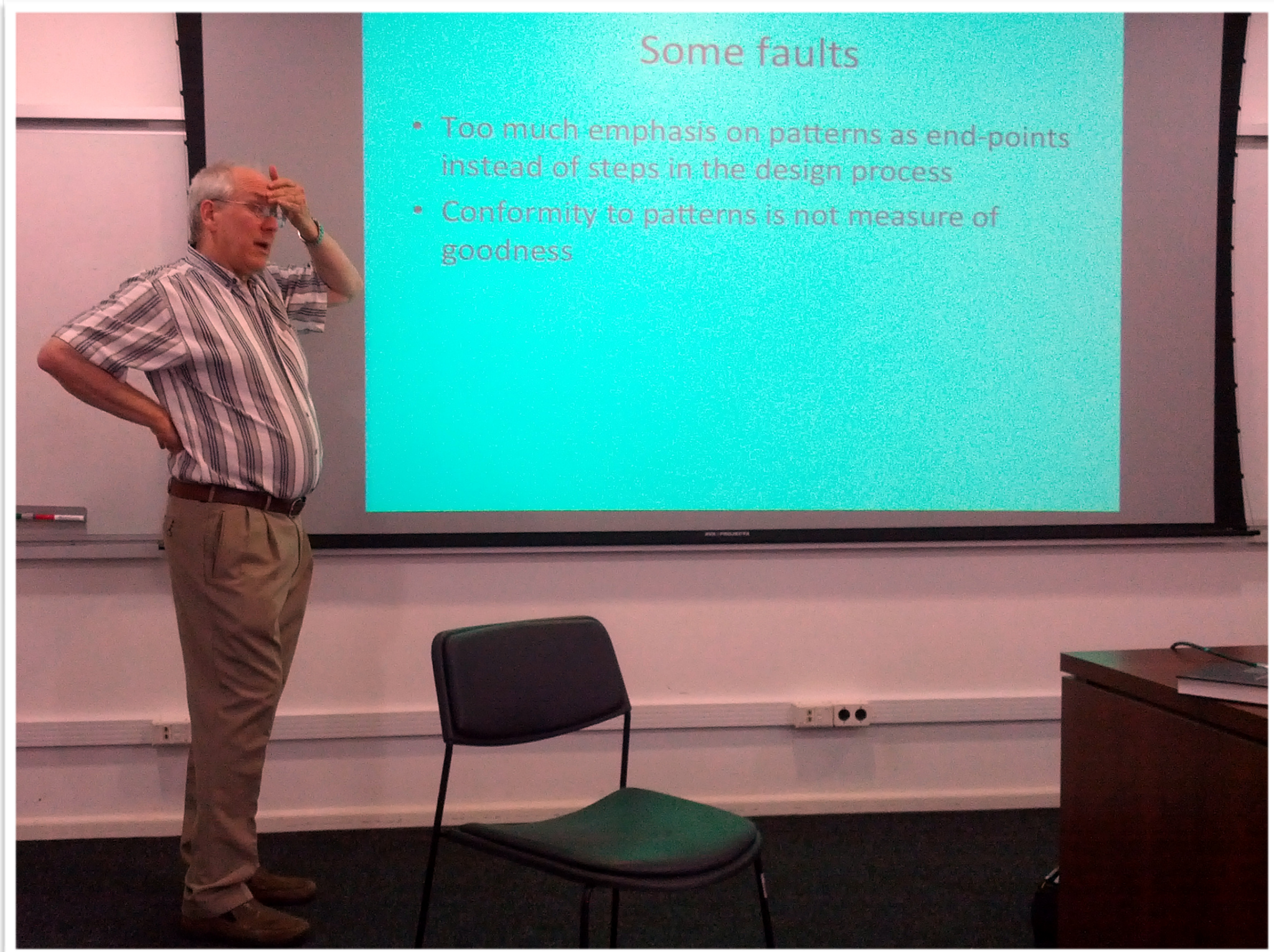
GOF: CLASSIC BOOK BY THE “GANG OF FOUR”

Design Patterns:
Elements of Reusable
Object-Oriented
Software (1995)

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



FAULTS IN THE SPREAD OF PATTERNS



NOT EVERY PATTERN IS UNIVERSAL

The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance," "Encapsulation," and "Polymorphism." Similarly, some of our patterns are supported directly by the less common object-oriented languages. CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor.[2]

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley, 1995), p. 4.

NOT EVERY PATTERN IS UNIVERSAL

The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance," "Encapsulation," and "Polymorphism." Similarly, some of our patterns are supported directly by the less common object-oriented languages. CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor.[2]

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley, 1995), p. 4.

NOT EVERY PATTERN IS UNIVERSAL

The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance," "Encapsulation," and "Polymorphism." Similarly, some of our patterns are supported directly by the less common object-oriented languages. CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor.[2]

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley, 1995), p. 4.

Design Patterns in Dynamic Programming

Peter Norvig
Chief Designer, Adaptive Systems
Harlequin Inc.

(2) Design Patterns in Dynamic Languages

- Dynamic Languages have fewer language limitations
  Less need for bookkeeping objects and classes
  Less need to get around class-restricted design
- Study of the *Design Patterns* book:
  16 of 23 patterns have qualitatively simpler implementation in Lisp or Dylan than in C++ for at least some uses of each pattern
- Dynamic Languages encourage new designs
  We will see some in Part (3)

(2) Design Patterns in Dynamic Languages

- Dynamic Languages have fewer language limitations
  Less need for bookkeeping objects and classes
  Less need to get around class-restricted design
- Study of the *Design Patterns* book:
  16 of 23 patterns have qualitatively simpler implementation in Lisp or Dylan than in C++ for at least some uses of each pattern
- Dynamic Languages encourage new designs
  We will see some in Part (3)

Design Patterns in Dylan or Lisp

16 of 23 patterns are either invisible or simpler, due to:

- First-class types (6): `Abstract-Factory, Flyweight, Factory-Method, State, Proxy, Chain-Of-Responsibility`
- First-class functions (4): `Command, Strategy, Template-Method, Visitor`
- Macros (2): `Interpreter, Iterator`
- Method Combination (2): `Mediator, Observer`
- Multimethods (1): `Builder`
- Modules (1): `Facade`

ThoughtWorks®

FEATURES

Core features, not mere syntax

SAMPLE FEATURES ✖ LANGUAGES

	Common Lisp
First-class functions	✔
First-class types	✔
Iterators	*
Variable model	reference
Type checking	dynamic
Type expression	structural

SAMPLE FEATURES ✖ LANGUAGES

	Common Lisp
Functions as objects	✔
Classes as objects	✔
Iterators	*
Variable model	reference
Type checking	dynamic
Type expression	structural

SAMPLE FEATURES ✖ LANGUAGES

	Common Lisp	C
First-class functions	✔	*
First-class types	✔	
Iterators	*	
Variable model	reference	value*
Type checking	dynamic	static
Type expression	structural	nominal

SAMPLE FEATURES ✖ LANGUAGES

	Common Lisp	C	Java
First-class functions	✔	*	✔
First-class types	✔		
Iterators	*		✔
Variable model	reference	value*	value and reference
Type checking	dynamic	static	static
Type expression	structural	nominal	nominal

SAMPLE FEATURES ✖ LANGUAGES

	Common Lisp	C	Java	Python
First-class functions	✔	*	✔	✔
First-class types	✔			✔
Iterators	*		✔	✔
Variable model	reference	value*	value and reference	reference
Type checking	dynamic	static	static	dynamic
Type expression	structural	nominal	nominal	structural

SAMPLE FEATURES ✖ LANGUAGES

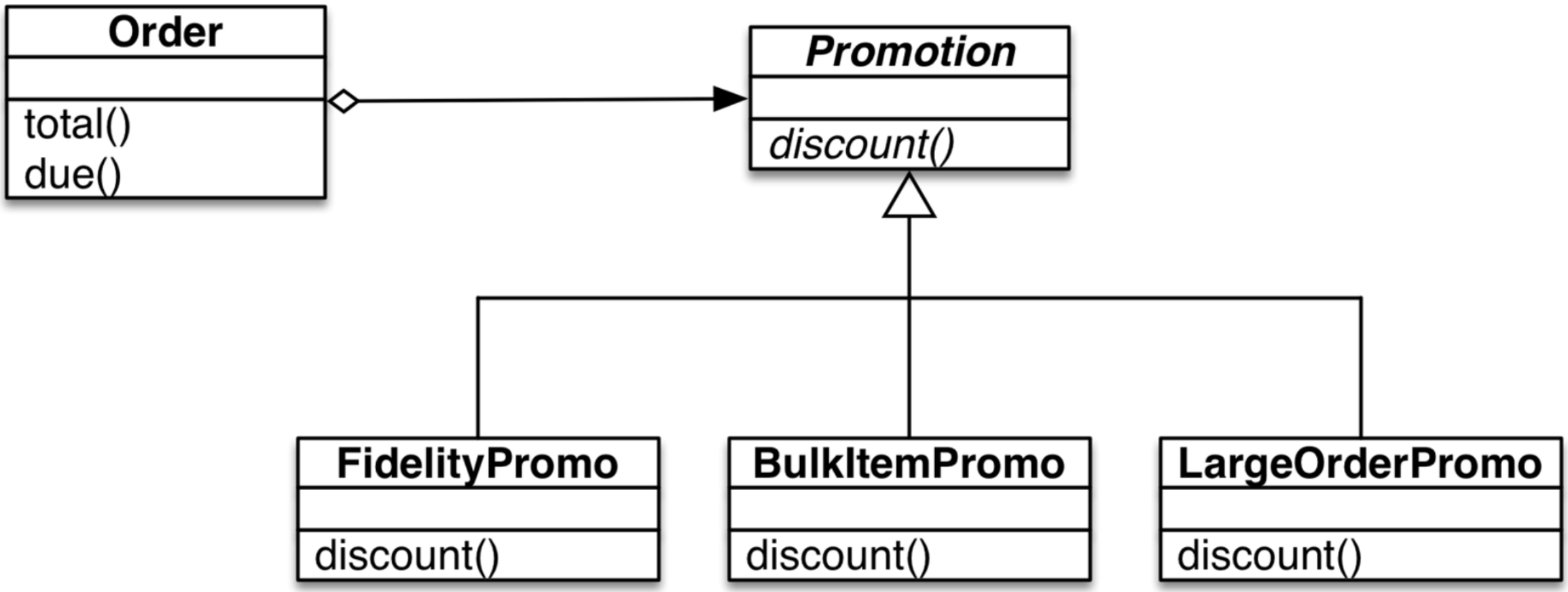
	Common Lisp	C	Java	Python	Go
First-class functions	✔	*	✔	✔	✔
First-class types	✔			✔	
Iterators	*		✔	✔	*
Variable model	reference	value*	value and reference	reference	value* and reference
Type checking	dynamic	static	static	dynamic	static
Type expression	structural	nominal	nominal	structural	structural

ThoughtWorks®

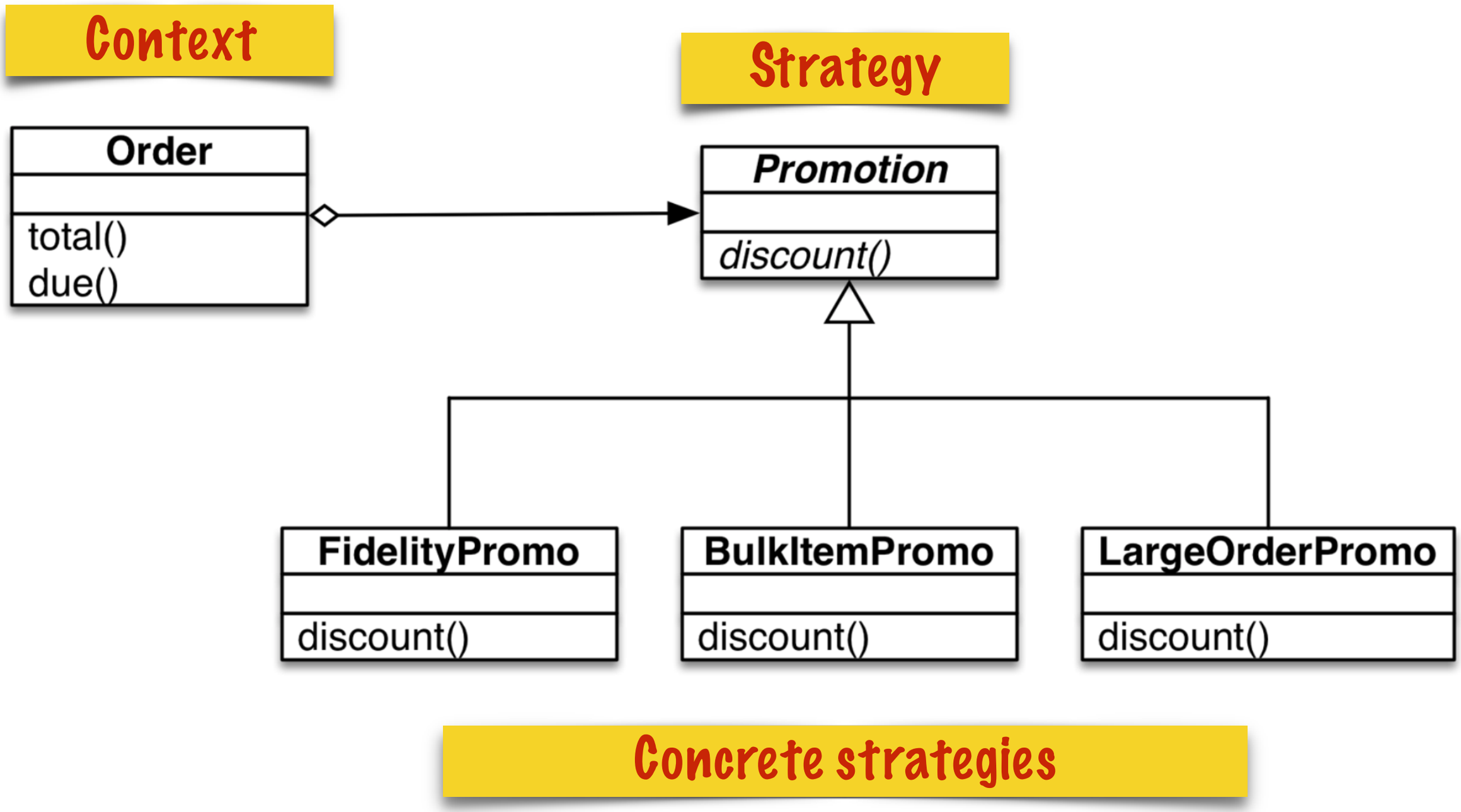
STRATEGY IN PYTHON

Leveraging Python's fundamental features

CHOOSE AN ALGORITHM AT RUN TIME



CHOOSE AN ALGORITHM AT RUN TIME



DOCTESTS FOR CONTEXT AND ONE CONCRETE STRATEGY

```
Create two customers, with and without "fidelity points":

    >>> ann = Customer('Ann Smith', 1100)  # ❶
    >>> joe = Customer('John Doe', 0)

Create a shopping cart with some fruits:

    >>> cart = [LineItem('banana', 4, .5),  # ❷
    ...         LineItem('apple', 10, 1.5),
    ...         LineItem('watermellon', 5, 5.0)]

The `FidelityPromo` only gives a discount to Ann:

    >>> Order(joe, cart, FidelityPromo())
    <Order total: 42.00 due: 42.00>
    >>> Order(ann, cart, FidelityPromo())
    <Order total: 42.00 due: 39.90>
```

Instance of Strategy is given to Order constructor

DOCTESTS FOR TWO ADDITIONAL CONCRETE STRATEGIES

```
The `BulkItemPromo` gives a discount for items with 20+ units:

    >>> banana_cart = [LineItem('banana', 30, .5),  # ❺
    ...                LineItem('apple', 10, 1.5)]
    >>> Order(joe, banana_cart, BulkItemPromo())  # ❻
    <Order total: 30.00 due: 28.50>

`LargeOrderPromo` gives a discount for orders with 10+ items:

    >>> long_order = [LineItem(str(item_code), 1, 1.0) # ❼
    ...               for item_code in range(10)]
    >>> Order(joe, long_order, LargeOrderPromo())  # ❽
    <Order total: 10.00 due: 9.30>
```

VARIATIONS OF STRATEGY IN PYTHON

Classic implementation using ABC

First-class function implementation

Parameterised closure implementation

Parameterised callable implementation

CLASSIC STRATEGY: THE CONTEXT CLASS

Strategy is given to constructor

```
class Order:  # the Context

    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = list(cart)
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        fmt = '<Order total: {:.2f} due: {:.2f}>'
```

Strategy is used here

STRATEGY ABC AND A CONCRETE STRATEGY

```python
class Promotion(ABC):  # the Strategy: an Abstract Base Class

    @abstractmethod
    def discount(self, order):
        """Return discount as a positive dollar amount"""
```

```python
class FidelityPromo(Promotion):  # first Concrete Strategy
    """5% discount for customers with 1000 or more fidelity points"""

    def discount(self, order):
        return order.total() * .05 if order.customer.fidelity >= 1000 else 0
```

TWO CONCRETE STRATEGIES

```
class BulkItemPromo(Promotion):  # second Concrete Strategy
    """10% discount for each LineItem with 20 or more units"""

    def discount(self, order):
        discount = 0
        for item in order.cart:
            if item.quantity >= 20:
                discount += item.total() * .1
        return discount


class LargeOrderPromo(Promotion):  # third Concrete Strategy
    """7% discount for orders with 10 or more distinct items"""

    def discount(self, order):
        distinct_items = {item.product for item in order.cart}
        if len(distinct_items) >= 10:
            return order.total() * .07
        return 0
```

ThoughtWorks®

FIRST-CLASS FUNCTION STRATEGY

CONTEXT: STRATEGY FUNCTION AS ARGUMENT

Strategy function is passed to Order constructor

```
>>> banana_cart = [LineItem('banana', 30, .5),
...                LineItem('apple', 10, 1.5)]
>>> Order(joe, banana_cart, bulk_item_promo)
<Order total: 30.00 due: 28.50>
>>> long_order = [LineItem(str(item_code), 1, 1.0)
...               for item_code in range(10)]
>>> Order(joe, long_order, large_order_promo)
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, large_order_promo)
<Order total: 42.00 due: 42.00>
```

CONTEXT: INVOKING THE STRATEGY FUNCTION

classic_strategy.py ↔ strategy.py

```
        self.customer = customer
        self.cart = list(cart)
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        fmt = '<Order total: {:.2f} due: {:.2f}>'
        return fmt.format(self.total(), self.due())


ss Promotion(ABC):  # the Strategy: an Abstract B

 @abstractmethod
```

```
        self.customer = customer
        self.cart = list(cart)
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total()
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion(self)
        return self.total() - discount

    def __repr__(self):
        fmt = '<Order total: {:.2f} due: {:.
        return fmt.format(self.total(), self.

fidelity_promo(order):
```

CONCRETE STRATEGIES AS FUNCTIONS

classic_strategy.py ↔ strategy.py

```
-class Promotion(ABC):  # the Strategy: an Abstrac    +def fidelity_promo(order):
-
-    @abstractmethod
-    def discount(self, order):
-        """Return discount as a positive dollar a
-
-
-class FidelityPromo(Promotion):  # first Concrete
    """5% discount for customers with 1000 or mor        """5% discount for customers with 100
-
-    def discount(self, order):
        return order.total() * .05 if order.custo        return order.total() * .05 if order.c


-class BulkItemPromo(Promotion):  # second Concret    +def bulk_item_promo(order):
    """10% discount for each LineItem with 20 or        """10% discount for each LineItem wit
-
-    def discount(self, order):
        discount = 0                                    discount = 0
        for item in order.cart:                         for item in order.cart:
            if item.quantity >= 20:                         if item.quantity >= 20:
                discount += item.total() * .1                   discount += item.total() * .1
        return discount                                 return discount
```

CONCRETE STRATEGIES AS FUNCTIONS (2)

classic_strategy.py ↔ strategy.py

```
-class BulkItemPromo(Promotion):  # second Concret
     """10% discount for each LineItem with 20 or
-
-    def discount(self, order):
         discount = 0
         for item in order.cart:
             if item.quantity >= 20:
                 discount += item.total() * .1
         return discount


-class LargeOrderPromo(Promotion):  # third Concre
     """7% discount for orders with 10 or more dis
-
-    def discount(self, order):
         distinct_items = {item.product for item i
         if len(distinct_items) >= 10:
             return order.total() * .07
         return 0
```

```
+def bulk_item_promo(order):
     """10% discount for each LineItem wit
     discount = 0
     for item in order.cart:
         if item.quantity >= 20:
             discount += item.total() * .1
     return discount


+def large_order_promo(order):
     """7% discount for orders with 10 or
     distinct_items = {item.product for it
     if len(distinct_items) >= 10:
         return order.total() * .07
     return 0
```

ThoughtWorks®

PARAMETERISED STRATEGY WITH CLOSURE

PARAMETERISED STRATEGY IMPLEMENTED AS CLOSURE

Promo is called with discount percent value

```
>>> banana_cart = [LineItem('banana', 30, .5),
...                LineItem('apple', 10, 1.5)]
>>> Order(joe, banana_cart, bulk_item_promo(10))
<Order total: 30.00 due: 28.50>
>>> long_order = [LineItem(str(item_code), 1, 1.0)
...                for item_code in range(10)]
>>> Order(joe, long_order, large_order_promo(7))
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, large_order_promo(7))
<Order total: 42.00 due: 42.00>
```

PARAMETERISED STRATEGY IMPLEMENTED AS CLOSURE

Outer function gets percent argument

```
def bulk_item_promo(percent):
    """discount for each LineItem with 20 or more units"""
    def discounter(order):
        discount = 0
        for item in order.cart:
            if item.quantity >= 20:
                discount += item.total() * percent/100.0
        return discount
    return discounter
```

Inner function carries percent binding in its closure

LAMBDA: SHORTCUT TO DECLARE INNER FUNCTION

strategy.py ↔ strategy_param.py

```
- def fidelity_promo(order):
-     """5% discount for customers with 1000 or mor
-     return order.total() * .05 if order.customer.
+ def fidelity_promo(percent):
+     """discount for customers with 1000 o
+     return lambda order: (order.total() *
+                           if order.custom

- def bulk_item_promo(order):
-     """10% discount for each LineItem with 20 or
+ def bulk_item_promo(percent):
+     """discount for each LineItem with 20
      discount = 0
      for item in order.cart:
          if item.quantity >= 20:
-             discount += item.total() * .1
+             discount += item.total()
      return discount
+ return discounter

- def large_order_promo(order):
-     """7% discount for orders with 10 or more dis
+ def large_order_promo(percent):
+     """discount for orders with 10 or mor
+     def discounter(order):
      distinct_items = {item.product for item in or
      if len(distinct_items) >= 10:
-         return order.total() * .07
+         return order.total() * percen
      return 0
```

ThoughtWorks®

PARAMETERISED STRATEGY WITH CALLABLE

PARAMETERISED STRATEGY IMPLEMENTED AS CLOSURE

Promo is instantiated with discount percent value

```
>>> banana_cart = [LineItem('banana', 30, .5),
...                LineItem('apple', 10, 1.5)]
>>> Order(joe, banana_cart, BulkItemPromo(10))
<Order total: 30.00 due: 28.50>
>>> long_order = [LineItem(str(item_code), 1, 1.0)
...                for item_code in range(10)]
>>> Order(joe, long_order, LargeOrderPromo(7))
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, LargeOrderPromo(7))
<Order total: 42.00 due: 42.00>
```

PROMOTION AS A CALLABLE CLASS

strategy_param.py ↔ strategy_param2.py

```
- def fidelity_promo(percent):
-     """discount for customers with 1000 or m
-     return lambda order: (order.total() * pe
-                           if order.customer.

- def bulk_item_promo(percent):
-     """discount for each LineItem with 20 or
-     def discounter(order):
```

```
+ class Promotion():
+     """compute discount for order"""
+
+     def __init__(self, percent):
+         self.percent = percent
+
+     def __call__(self, order):
+         raise NotImplementedError("Subclass re
+
+
+ class FidelityPromo(Promotion):
+     """discount for customers with 1000 or mor
+
+     def __call__(self, order):
+         if order.customer.fidelity >= 1000:
+             return order.total() * self.percen
+         return 0


+ class BulkItemPromo(Promotion):
+     """discount for each LineItem with 20 or m
+
+     def __call__(self, order):
          discount = 0
```

CONCRETE STRATEGIES AS CALLABLE CONCRETE CLASSES

strategy_param.py ↔ strategy_param2.py

```
- def bulk_item_promo(percent):                    + class BulkItemPromo(Promotion):
      """discount for each LineItem with 20 or more       """discount for each LineItem with 20
-     def discounter(order):                        +
                                                   +     def __call__(self, order):
          discount = 0                                         discount = 0
          for item in order.cart:                              for item in order.cart:
              if item.quantity >= 20:                              if item.quantity >= 20:
-                 discount += item.total() * percer    +                   discount += item.total()
          return discount                                      return discount
-     return discounter

- def large_order_promo(percent):                  + class LargeOrderPromo(Promotion):
      """discount for orders with 10 or more distir       """discount for orders with 10 or mor
-     def discounter(order):                        +
                                                   +     def __call__(self, order):
          distinct_items = {item.product for item i            distinct_items = {item.product fo
          if len(distinct_items) >= 10:                        if len(distinct_items) >= 10:
-             return order.total() * percent / 100.    +               return order.total() * self.p
          return 0                                             return 0
-     return discounter
```

ThoughtWorks®

WHICH IS MORE IDIOMATIC?

Classes x functions

WHICH IS MORE IDIOMATIC?

Classic strategy *feels* too verbose for Python*

First-class functions are very common in the standard library

- The sorted built-in key argument is one example.

* Yes, this is subjective. I am talking about style!

WHICH IS MORE IDIOMATIC — WITH PARAMETER?

Use of closure is common in Python

- nonlocal was added in Python 3 to support it better

Callable objects are uniquely Pythonic

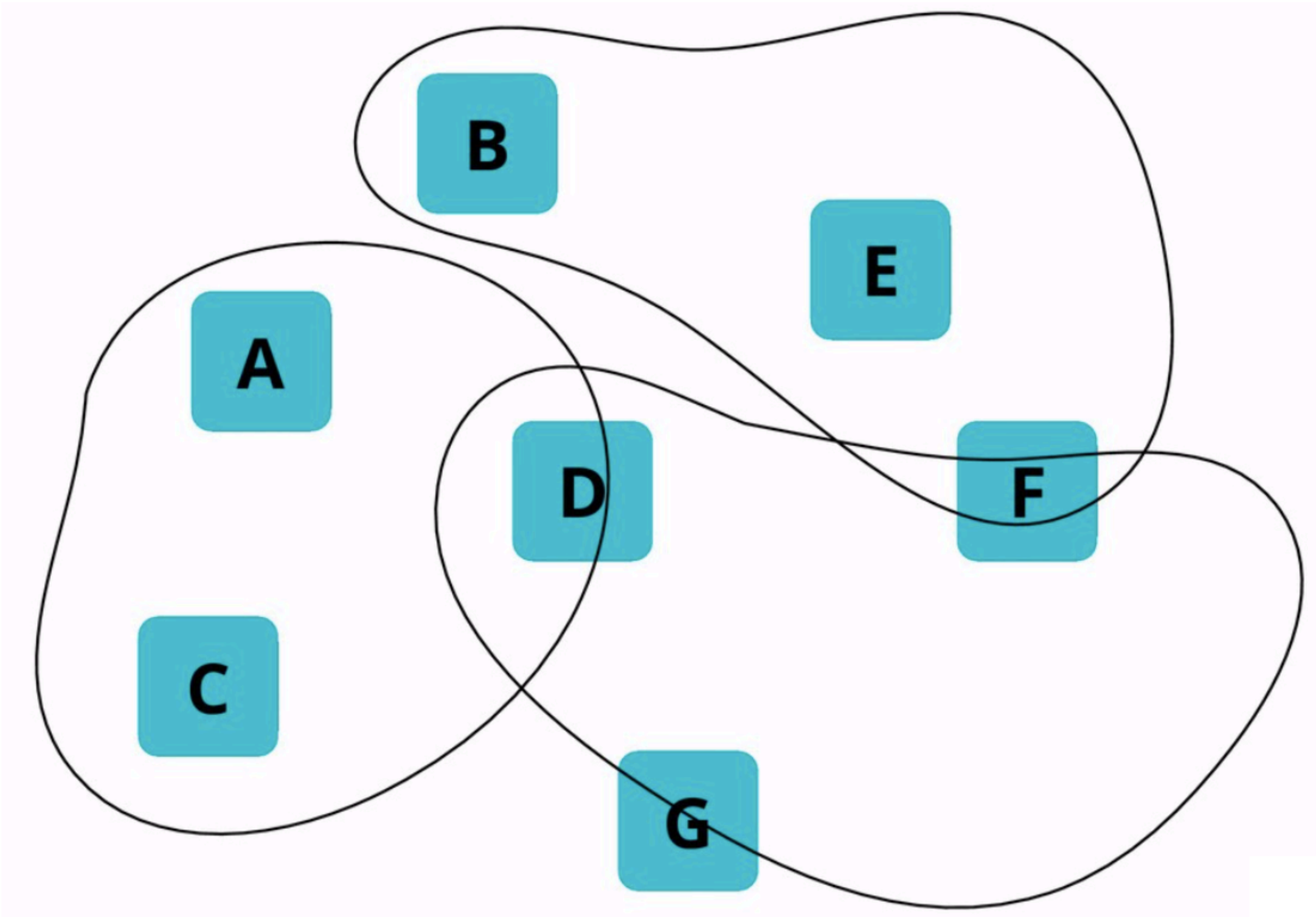
- Graham Dumpleton recommends callable classes as the best way to code decorators

ThoughtWorks®

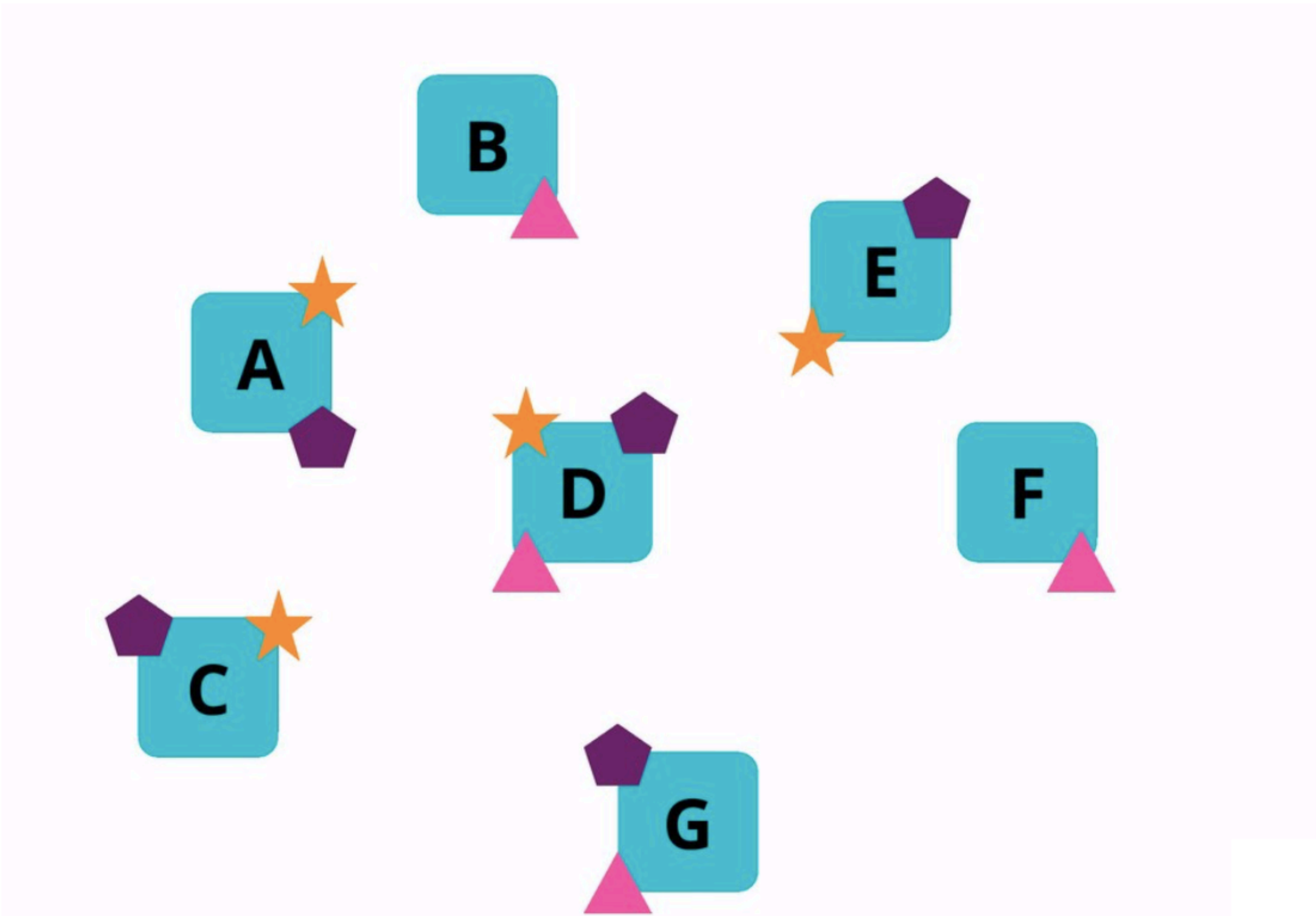
WRAPPING UP

Learn the fundamentals

INSTEAD OF PARADIGMS...



FOCUS ON FUNDAMENTAL FEATURES



FEATURES ARE THE BEST GUIDE...

...to decide whether a particular pattern or implementation makes the most of the language.

	A	B	C	D	E	F	G
		✔		✔		✔	✔
	✔		✔	✔	✔		
	✔		✔	✔	✔		✔

WHY LEARN THE FUNDAMENTALS*

Learn new languages faster

Leverage language features

Choose among alternative implementations

Make sensible use of design patterns

Debug hard problems

Emulate missing features when they are helpful

* Inspired by Programming Language Pragmatics

  Michael L. Scott

¡GRACIAS!

Luciano Ramalho
luciano.ramalho@thoughtworks.com

Twitter: @ramalhoorg / @standupdev
Repo: github.com/standupdev/beyond-paradigms
Slides: speakerdeck.com/ramalho

ThoughtWorks®