ThoughtWorks®

Theory in Practice

GO BEYOND PARADIGMS

Is Go object-oriented? Is that a good question?
Understand language features, use the right patterns.

Luciano Ramalho
@ramalhoorg

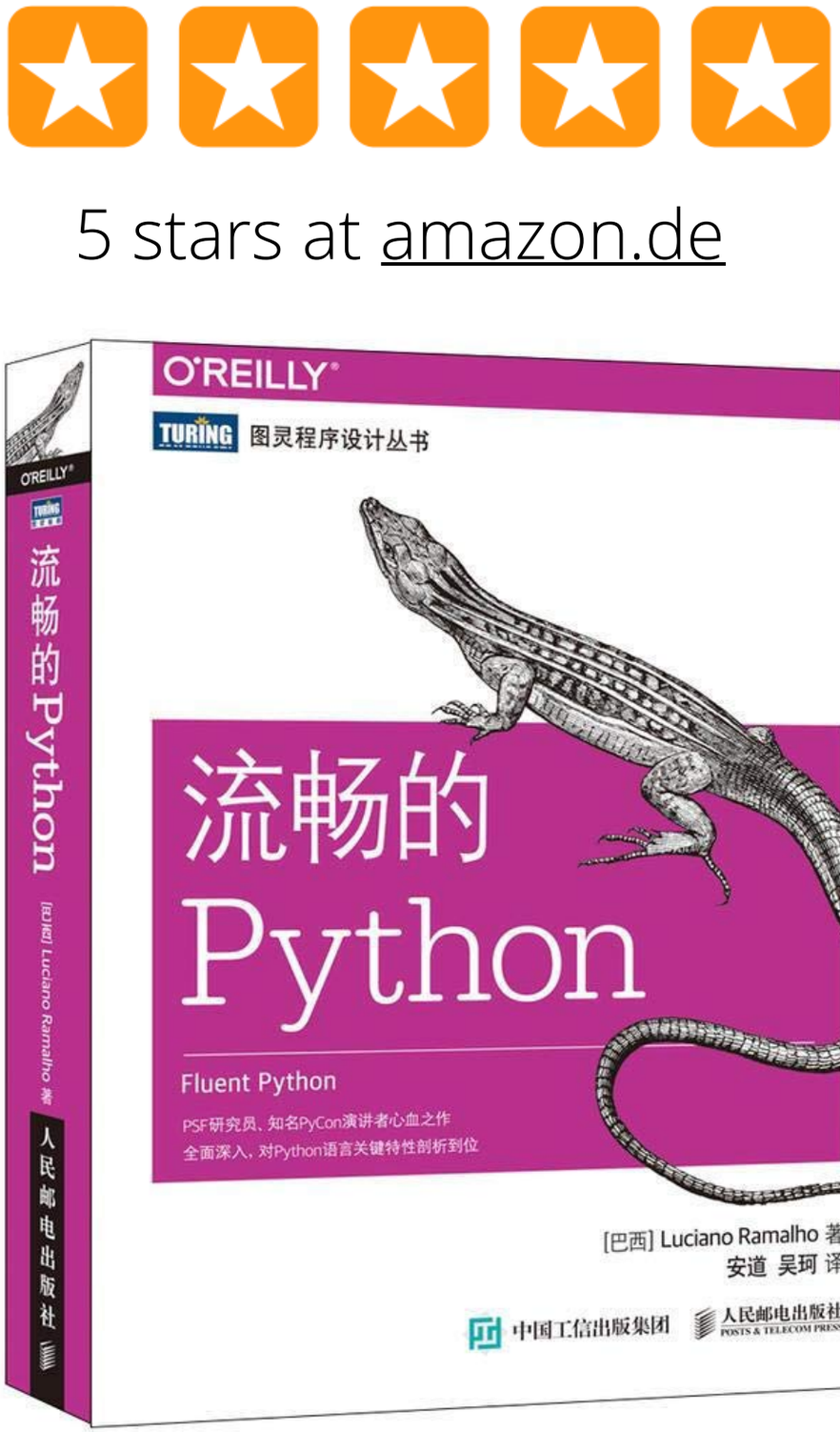
ThoughtWorks®

LUCIANO RAMALHO

Technical Principal

@ramalhoorg
luciano.ramalho@thoughtworks.com

FLUENT PYTHON



5 stars at amazon.de

Published in 9 languages:

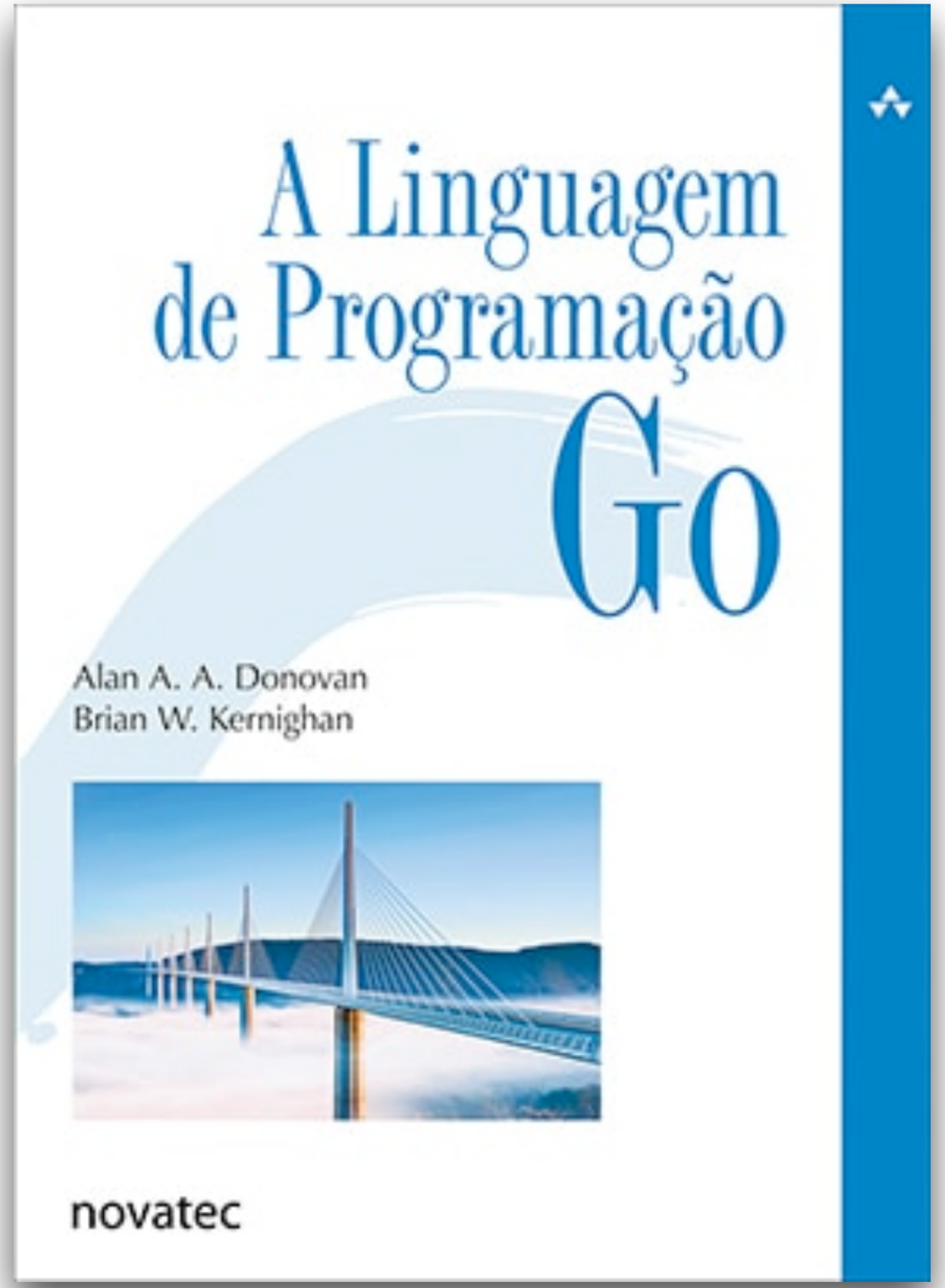
- Chinese (simplified)
- Chinese (traditional)
- English
- French
- Japanese
- Korean
- Polish
- Portuguese
- Russian

2nd edition: Q4 2020

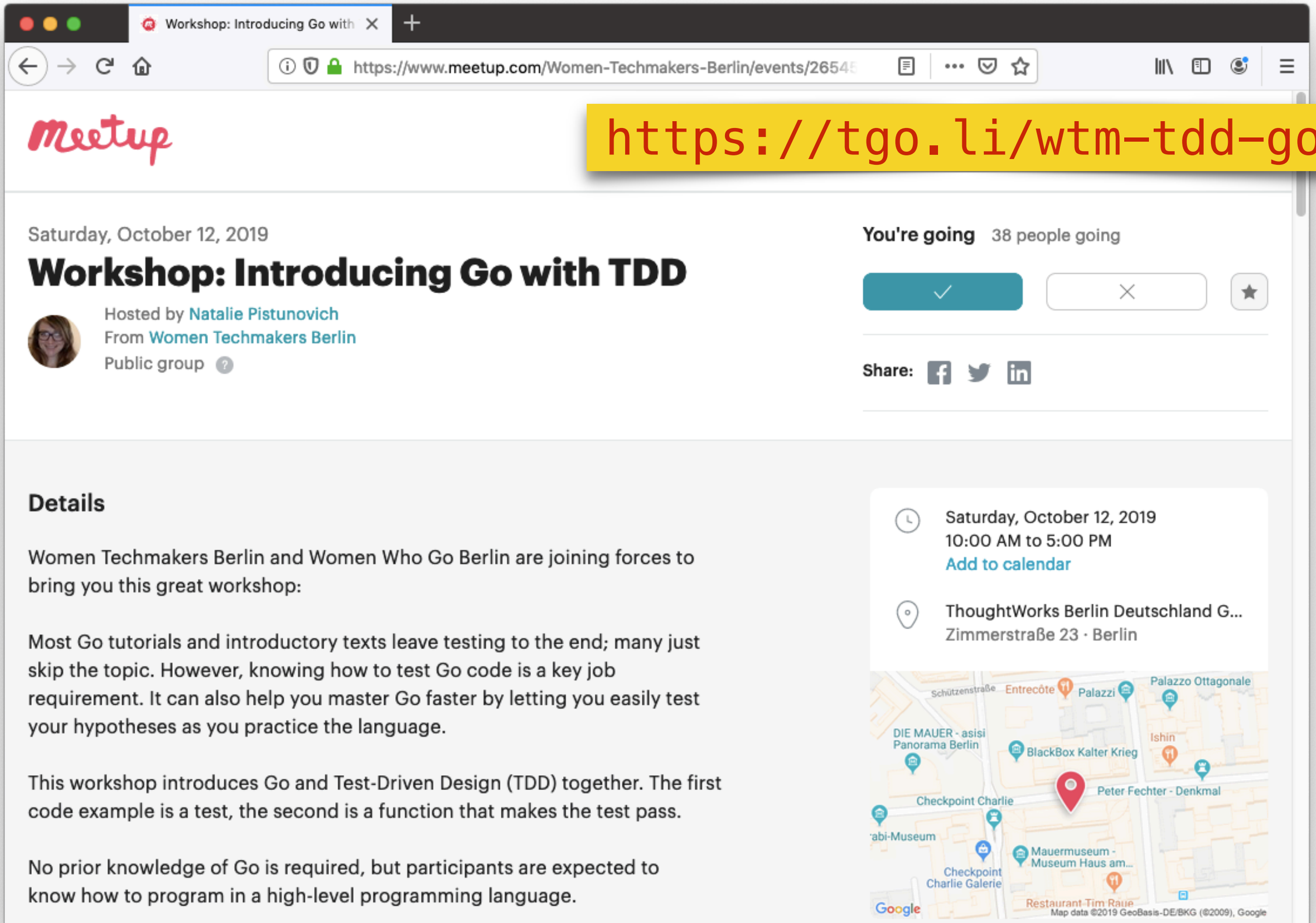
TECH REVIEWER OF THE BRAZILIAN EDITION OF GOPL

GOPL:
The Go Programming Language

Donovan & Kernighan



WORKSHOP: INTRODUCING GO WITH TDD



ThoughtWorks®

EMPORIO CELESTIAL

By Jorge Luis Borges

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Español
Português
Русский
Українська
Edit links

Celestial Emporium of Benevolent Knowledge

From Wikipedia, the free encyclopedia

Celestial Emporium of Benevolent Knowledge (Spanish: *Emporio celestial de conocimientos benévolos*) is a fictitious taxonomy of animals described by the writer Jorge Luis Borges in his 1942 essay "The Analytical Language of John Wilkins" (*El idioma analítico de John Wilkins*).[1][2]

Wilkins, a 17th-century philosopher, had proposed a universal language based on a classification system that would encode a description of the thing a word describes into the word itself—for example, *Zi* identifies the genus *beasts*; *Zit* denotes the "difference" *rapacious beasts of the dog kind*; and finally *Zita* specifies *dog*.

In response to this proposal and in order to illustrate the arbitrariness and cultural specificity of any attempt to categorize the world, Borges describes this example of an alternate taxonomy, supposedly taken from an ancient Chinese encyclopædia entitled *Celestial Emporium of Benevolent Knowledge*.

The list divides all animals into 14 categories:

- Those that belong to the emperor
- Embalmed ones
- Those that are trained
- Suckling pigs
- Mermaids (or Sirens)
- Fabulous ones
- Stray dogs
- Those that are included in this classification
- Those that tremble as if they were mad
- Innumerable ones
- Those drawn with a very fine camel hair brush
- *Et cetera*
- Those that have just broken the flower vase
- Those that, at a distance, resemble flies

Borges claims that the list was discovered in its Chinese source by the translator Franz Kuhn.[3][4][5]

Influences of the list [edit]

This list has stirred considerable philosophical and literary commentary.

Michel Foucault begins his preface to *The Order of Things*,[6]

> This book first arose out of a passage in Borges, out of the laughter that shattered, as I read the passage, all the familiar landmarks of thought—*our* thought, the thought that bears the stamp of our age and our geography—breaking up all the ordered surfaces and all the planes with which we are accustomed to tame the wild profusion of existing things and continuing long afterwards to disturb and threaten with collapse our age-old definitions between the Same and the Other.

Foucault then quotes Borges' passage.

Louis Sass has suggested, in response to Borges' list, that such "Chinese" thinking shows signs of typical schizophrenic thought processes.[7] By contrast, the linguist George Lakoff has pointed out that while Borges' list is not possibly a human categorization, many categorizations of objects found in nonwestern cultures have a similar feeling to Westerners.[8]

Keith Windschuttle, an Australian historian, cited alleged acceptance of the authenticity of the list by many academics as a sign of the degeneration of the Western academy[9] and a terminal lack of humor.

Attribution [edit]

Scholars have questioned whether the attribution of the list to Franz Kuhn is genuine. While Kuhn did indeed translate Chinese literature, Borges' works often feature many learned pseudo-references resulting in a mix of facts and fiction. To date, no evidence for the existence of such a list has been found.[10]

Borges himself questions the veracity of the quote in his essay, referring to "the unknown (or false) Chinese encyclopaedia writer".[4]

See also [edit]

- *An Essay towards a Real Character and a Philosophical Language*, the 1668 work of philosopher John Wilkins that was the subject of Borges' essay
- *Book of Imaginary Beings*, Borges' bestiary, a catalog of fantastic animals
- *Leishu* – a genre of reference books historically compiled in China and other countries of the Sinosphere

References [edit]

1. ^ Borges, Jorge Luis (1999), "John Wilkins' Analytical Language", in Weinberger, Eliot (ed.), *Selected nonfictions*, Eliot Weinberger, transl., Penguin Books, p. 231, ISBN 0-14-029011-7. The essay was originally published as "El idioma analítico de John Wilkins", *La Nación* (in Spanish), Argentina, 8 February 1942, and republished in *Otras inquisiciones*
2. ^ Mantovani, Giuseppe (2000), *Exploring borders: understanding culture and psychology*, Routledge, ISBN 041523400X, retrieved 26 April 2011
3. ^ A slightly different English translation is at: Luis Borges, Jorge (April 8, 2006), *The Analytical Language of John Wilkins*, Lilia Graciela Vázquez, transl.
4. ^ a b Borges, Jorge Luis (April 8, 2006), *El idioma analítico de John Wilkins* (in Spanish and English), Crockford
5. ^ "Borges", *Darwin-L*, (mailing list archive), RJ Ohara, 1996
6. ^ Foucault, Michel (1994) [1966]. *The Order of Things : An Archaeology of Human Sciences*. Vintage. p. XVI. ISBN 0-679-75335-4.
7. ^ Sass, Louis (1994) [1992], *Madness and Modernism: Insanity in the Light of Modern Art, Literature and Thought*, Harvard University Press, ISBN 0-674-54137-5
8. ^ Lakoff, George (1987), *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind*, University of Chicago Press, ISBN 0-226-46804-6
9. ^ Windschuttle, Keith (September 15, 1997), "Academic Questions", *Absolutely Relative*, National Review, archived from the original on March 8, 2005
10. ^ "LINGUIST List 7.1446: Borgesian joke". Linguistlist.org. Retrieved 2013-01-25.

v • t • e	**Jorge Luis Borges**	[hide]
Bibliography		
Original collections	***A Universal History of Infamy***	"On Exactitude in Science"
	Ficciones	"Tlön, Uqbar, Orbis Tertius" · "The Approach to Al-Mu'tasim" · "Pierre Menard, Author of the Quixote" · "The Circular Ruins" · "The Lottery in Babylon" · "An Examination of the Work of Herbert Quain" · "The Library of Babel" · "The Garden of Forking Paths" · "Funes the Memorious" · "The Form of the Sword" · "Theme of the Traitor and the Hero" · "Death and the Compass" · "The Secret Miracle" · "Three Versions of Judas" · "The End" · "The Sect of the Phoenix" · "The South"
	The Aleph	"The Immortal" · "The Dead Man" · "The Theologians" · "Emma Zunz" · "The House of Asterion" · "Deutsches Requiem" · "Averroes's Search" · "The Zahir" · "The Writing of the God" · "The Two Kings and the Two Labyrinths" · "The Wait" · "The Aleph"
	Otras Inquisiciones (1937–1952)	"The Analytical Language of John Wilkins"
	Dreamtigers	"Borges and I"
	Dr. Brodie's Report	"The Encounter" · "The Gospel According to Mark"
	The Book of Sand	"The Other" · "Ulrikke" · "The Congress" · "There Are More Things" · "The Disk" · "The Book of Sand"
	Shakespeare's Memory	"Blue Tigers" · "Shakespeare's Memory"
Other works	*Historia de la eternidad* · "A New Refutation of Time" · *Borges on Martín Fierro* · "El Golem" · *Book of Imaginary Beings* · *Labyrinths* · *Adrogue, con ilustraciones de Norah Borges*	
Related	Leonor Acevedo Suarez (mother) · Jorge Guillermo Borges (father) · Norah Borges (sister) · ***Celestial Emporium of Benevolent Knowledge*** · H. Bustos Domecq · Pedro Mata · Uqbar · Borges and mathematics	

Categories: Classification systems | Jorge Luis Borges | Michel Foucault | Sociology of knowledge | Taxonomy | Taxonomy (biology) | Fictional elements introduced in 1942

Celestial Emporium of Benevolent Knowledge

From Wikipedia, the free encyclopedia

Celestial Emporium of Benevolent Knowledge (Spanish: *Emporio celestial de conocimientos benévolos*) is a fictitious taxonomy of animals described by the writer Jorge Luis Borges in his 1942 essay "The Analytical Language of John Wilkins" (*El idioma analítico de John Wilkins*).[1][2]

Wilkins, a 17th-century philosopher, had proposed a universal language based on a classification system that would encode a description of the thing a word describes into the word itself—for example, *Zi* identifies the genus *beasts*; *Zit* denotes the "difference" *rapacious beasts of the dog kind*; and finally *Zita* specifies *dog*.

In response to this proposal and in order to illustrate the arbitrariness and cultural specificity of any attempt to categorize the world, Borges describes this example of an alternate taxonomy, supposedly taken from an ancient Chinese encyclopædia entitled *Celestial Emporium of Benevolent Knowledge*.

The list divides all animals into 14 categories:

- Those that belong to the emperor
- Embalmed ones
- Those that are trained
- Suckling pigs
- Mermaids (or Sirens)
- Fabulous ones
- Stray dogs
- Those that are included in this classification
- Those that tremble as if they were mad
- Innumerable ones
- Those drawn with a very fine camel hair brush
- *Et cetera*
- Those that have just broken the flower vase
- Those that, at a distance, resemble flies

Borges claims that the list was discovered in its Chinese source by the translator Franz Kuhn.[3][4][5]

Influences of the list [edit]

this example of an alternate taxonomy, supposedly taken from an ancient Chinese encyclopædia entitled *Celestial Emporium of Benevolent Knowledge*.

The list divides all animals into 14 categories:

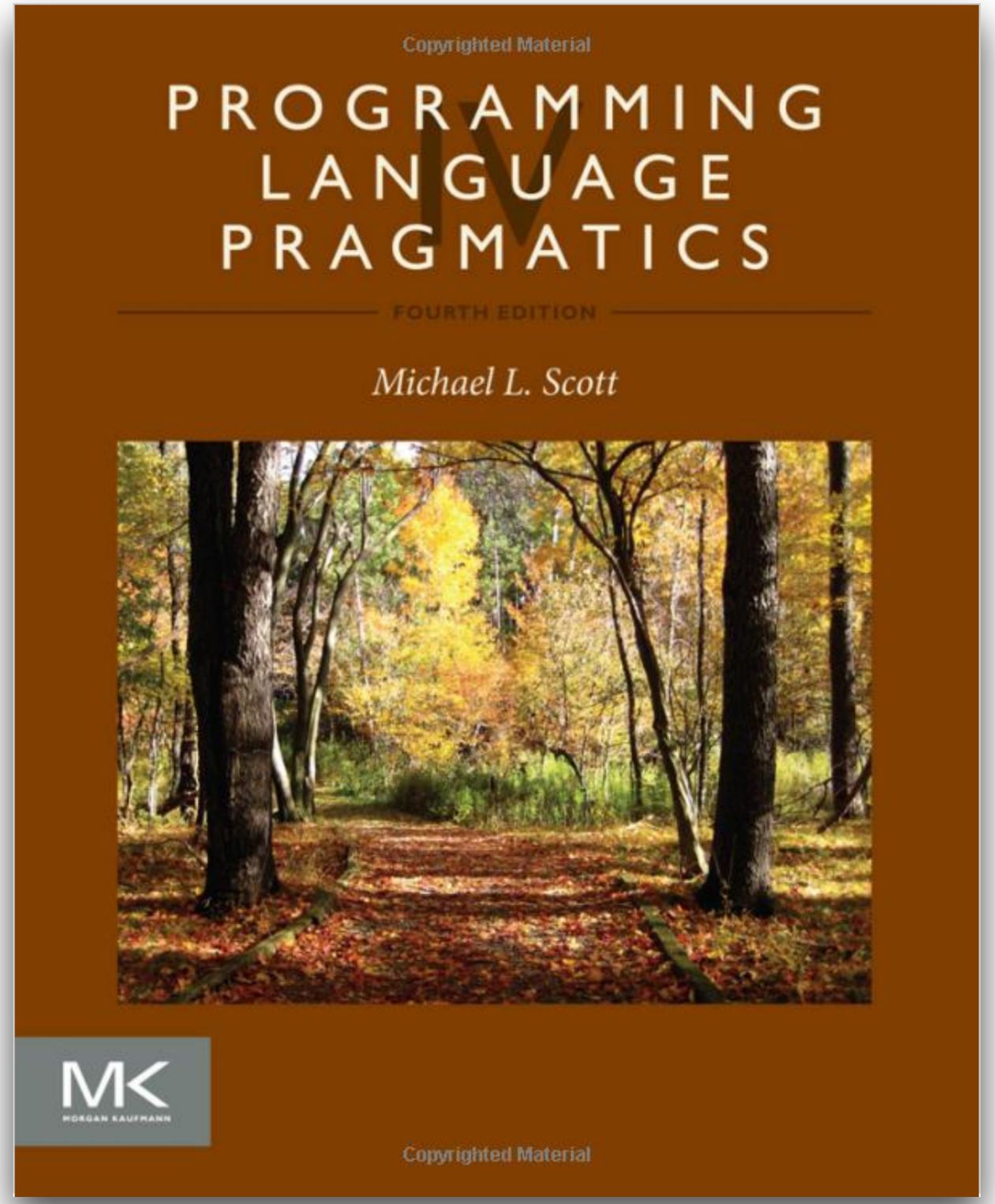
- Those that belong to the emperor
- Embalmed ones
- Those that are trained
- Suckling pigs
- Mermaids (or Sirens)
- Fabulous ones
- Stray dogs
- Those that are included in this classification
- Those that tremble as if they were mad
- Innumerable ones
- Those drawn with a very fine camel hair brush
- *Et cetera*
- Those that have just broken the flower vase
- Those that, at a distance, resemble flies

ThoughtWorks®

PARADIGMS

Programming language categories

UM LIVRO ESTILO “SURVEY” (LEVANTAMENTO)



Programming Language Pragmatics, 4th edition (2015)

Michael L. Scott

GCD ASM X86

Máximo divisor comum em assembly x86

(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 EM C, OCAML E 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 GO

```go
func Gcd(a, b int) int {
    for a != b {
        if a > b {
            a -= b
        } else {
            b -= a
        }
    }
    return a
}
```

Imperative style

```go
func Gcd(a, b int) int {
    if a == b {
        return a
    }
    if a > b {
        return Gcd(b, a-b)
    }
    return Gcd(a, b-a)
}
```

Functional style

GCD IN GO

Bad fit for Go:
no tail-call
optimisation

```
func Gcd(a, b int) int {
    for a != b {
        if a > b {
            a -= b
        } else {
            b -= a
        }
    }
    return a
}
```

Imperative style

```
func Gcd(a, b int) int {
    if a == b {
        return a
    }
    if a > b {
        return Gcd(b, a-b)
    }
    return Gcd(a, b-a)
}
```

Functional style

SCOTT'S 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

SCOTT'S 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, …

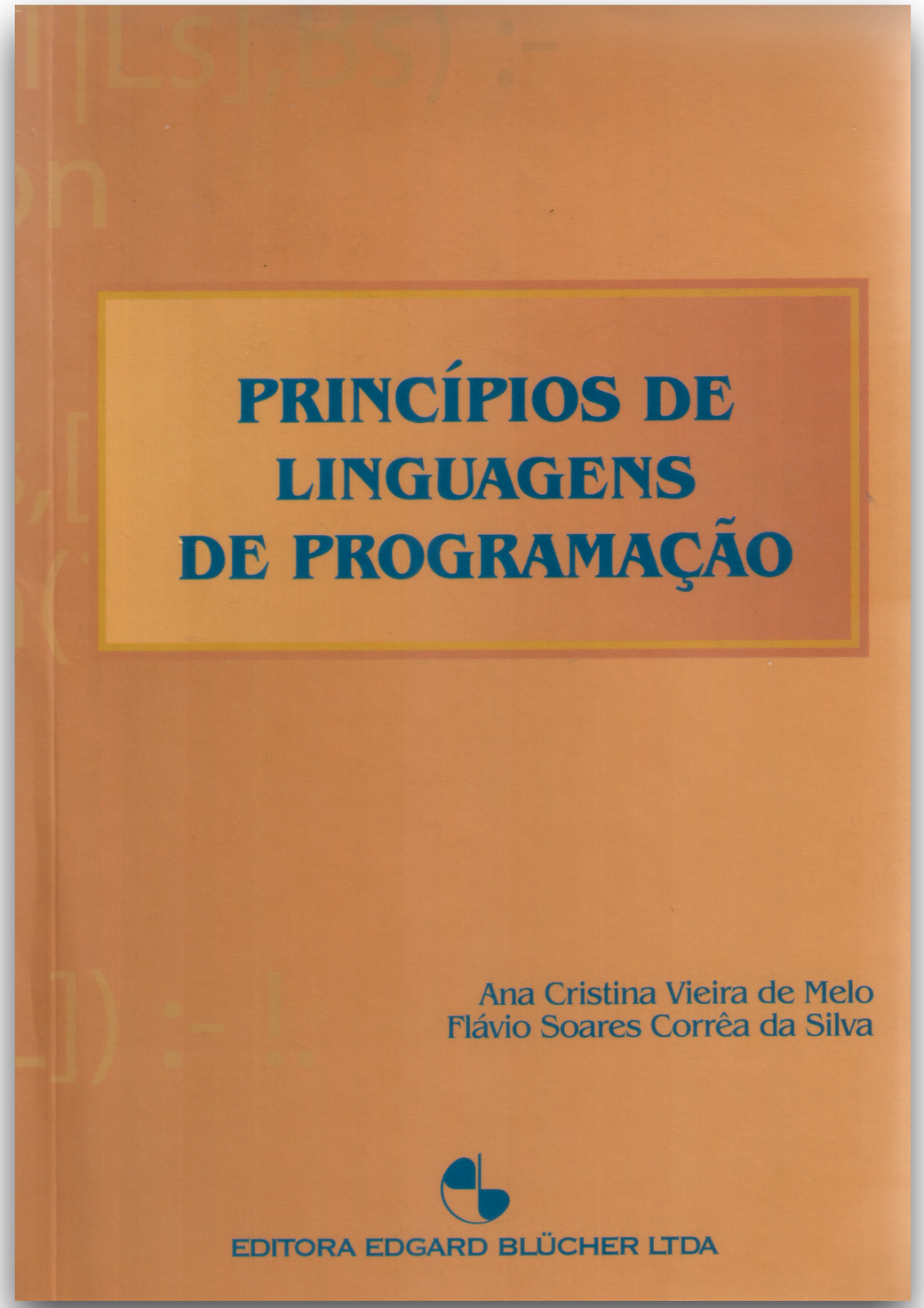
FIGURE 1.1 Classification of programming languages. Note that the categories are fuzzy and open to debate. In particular, it is possible for a functional language to be object-oriented, and many authors do not consider functional programming to be declarative.

SCOTT'S 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

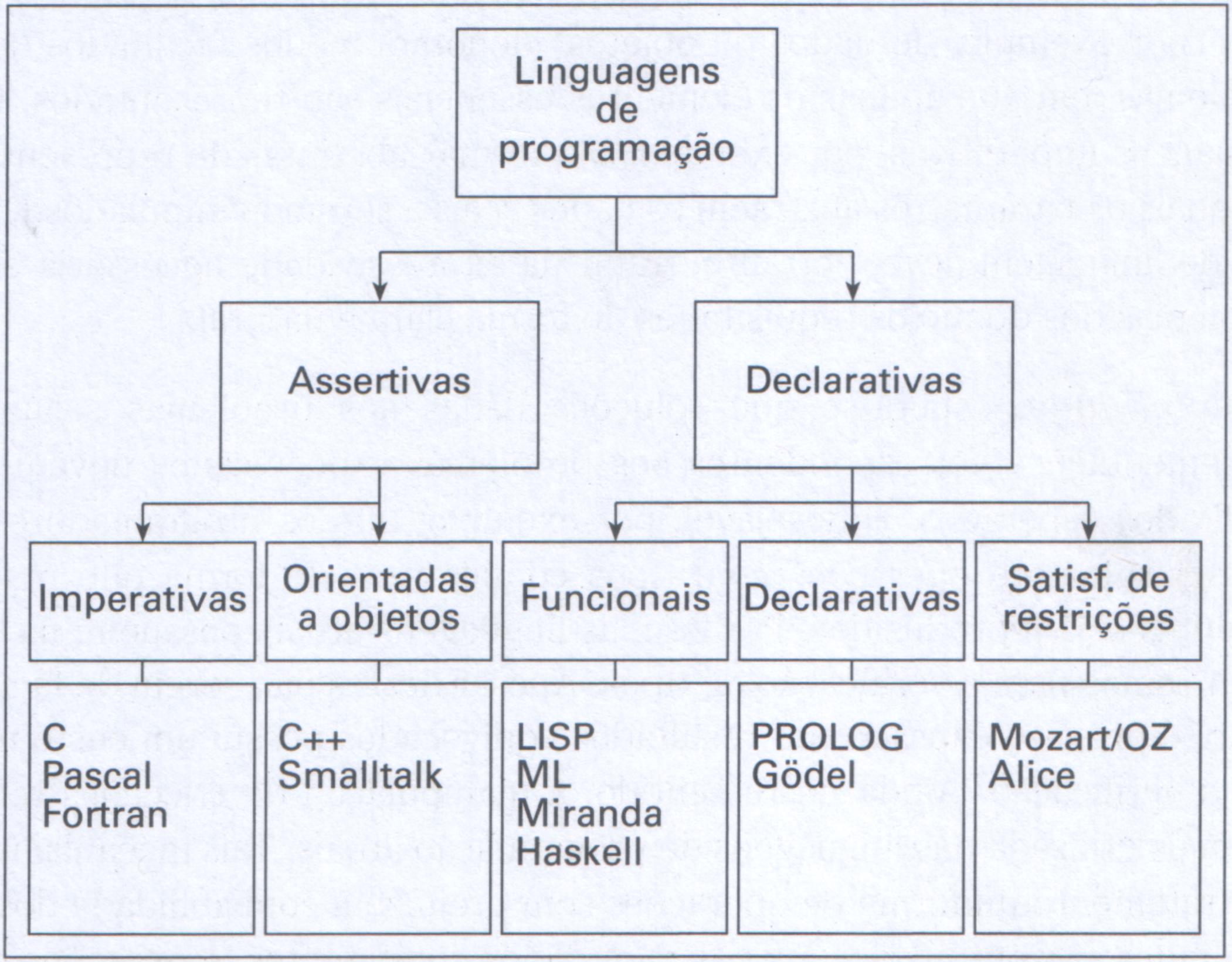


Figura 1 — Tipologia de linguagens de programação.

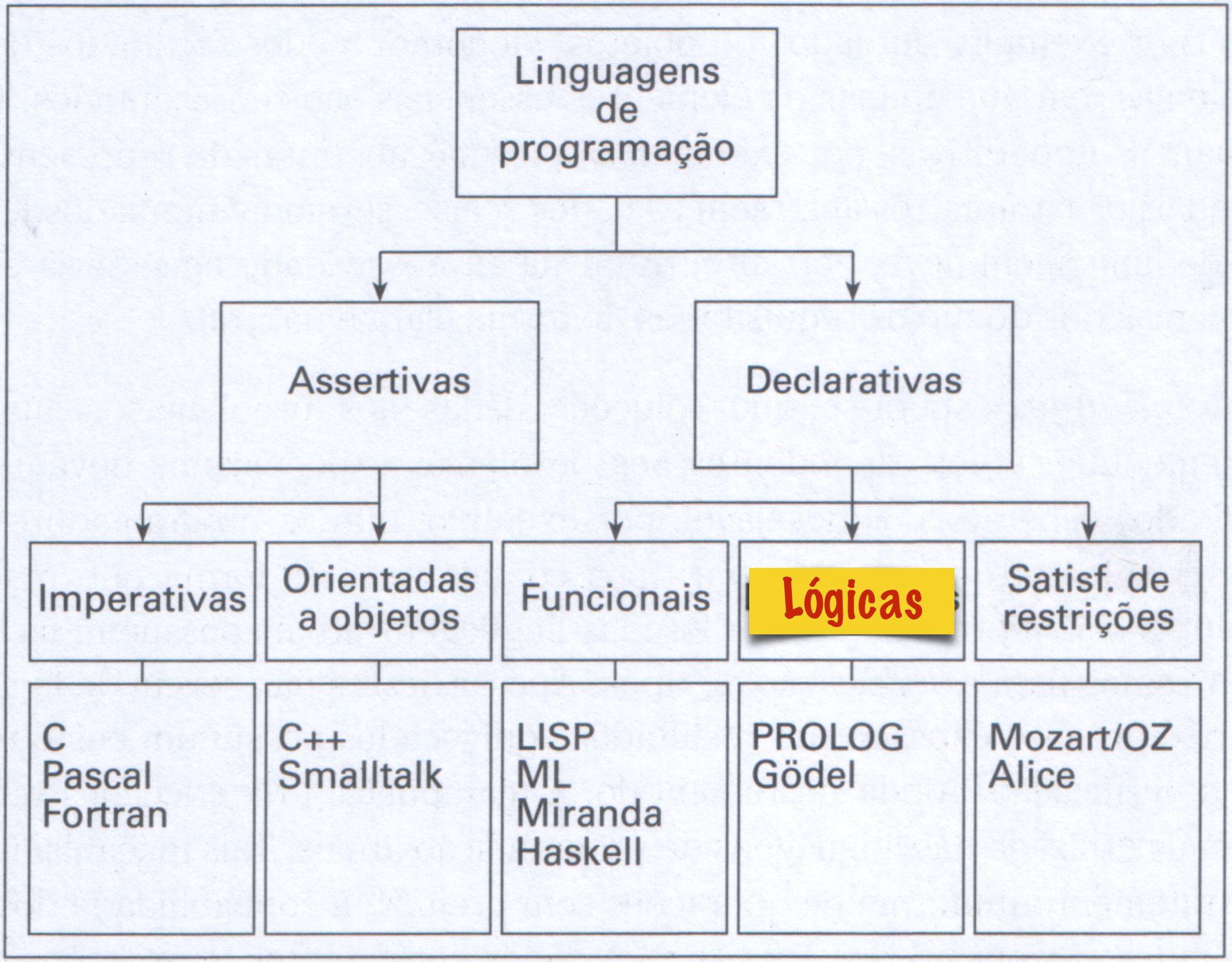
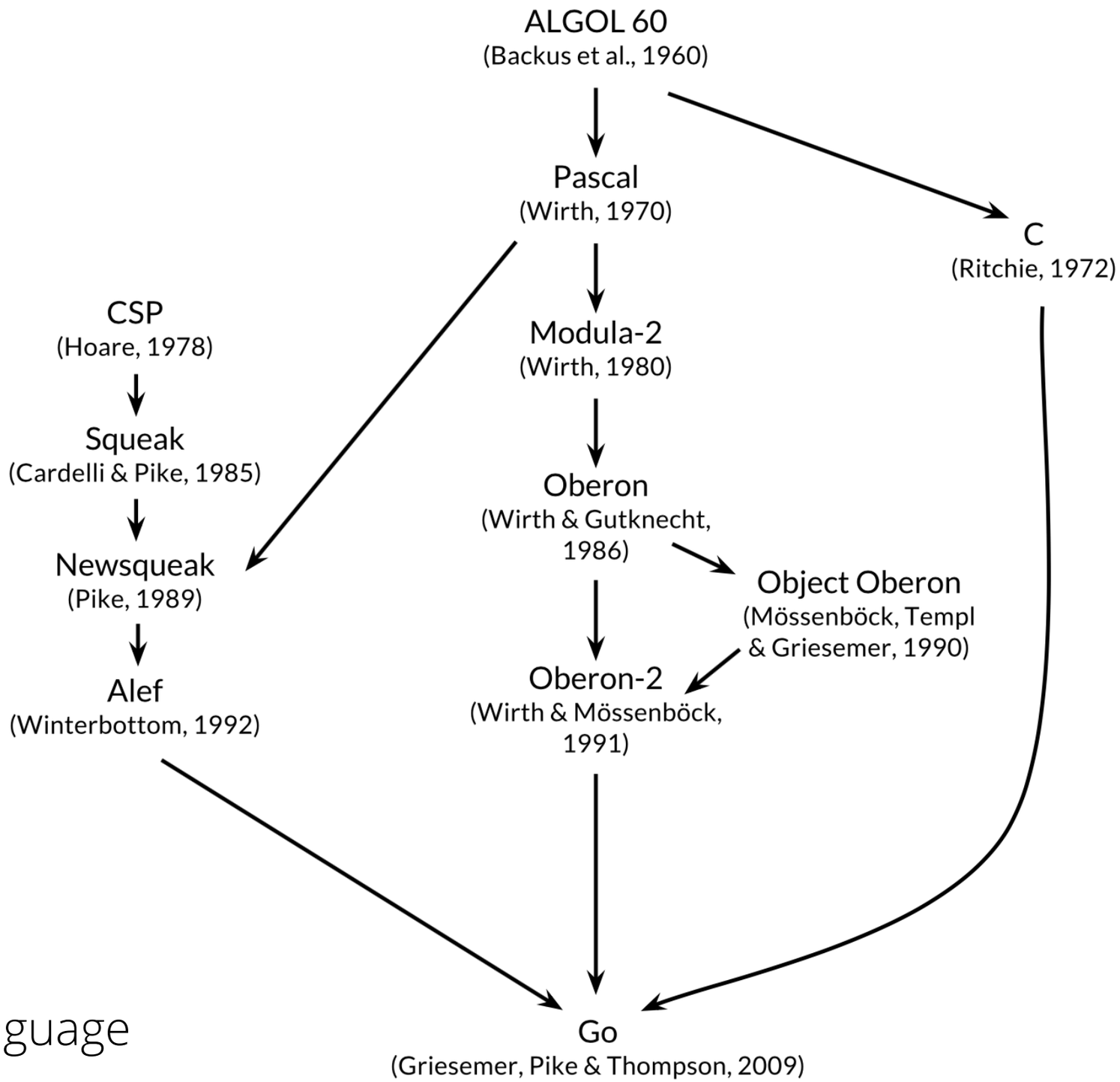


Figura 1 — Tipologia de linguagens de programação.

THE GO FAMILY TREE



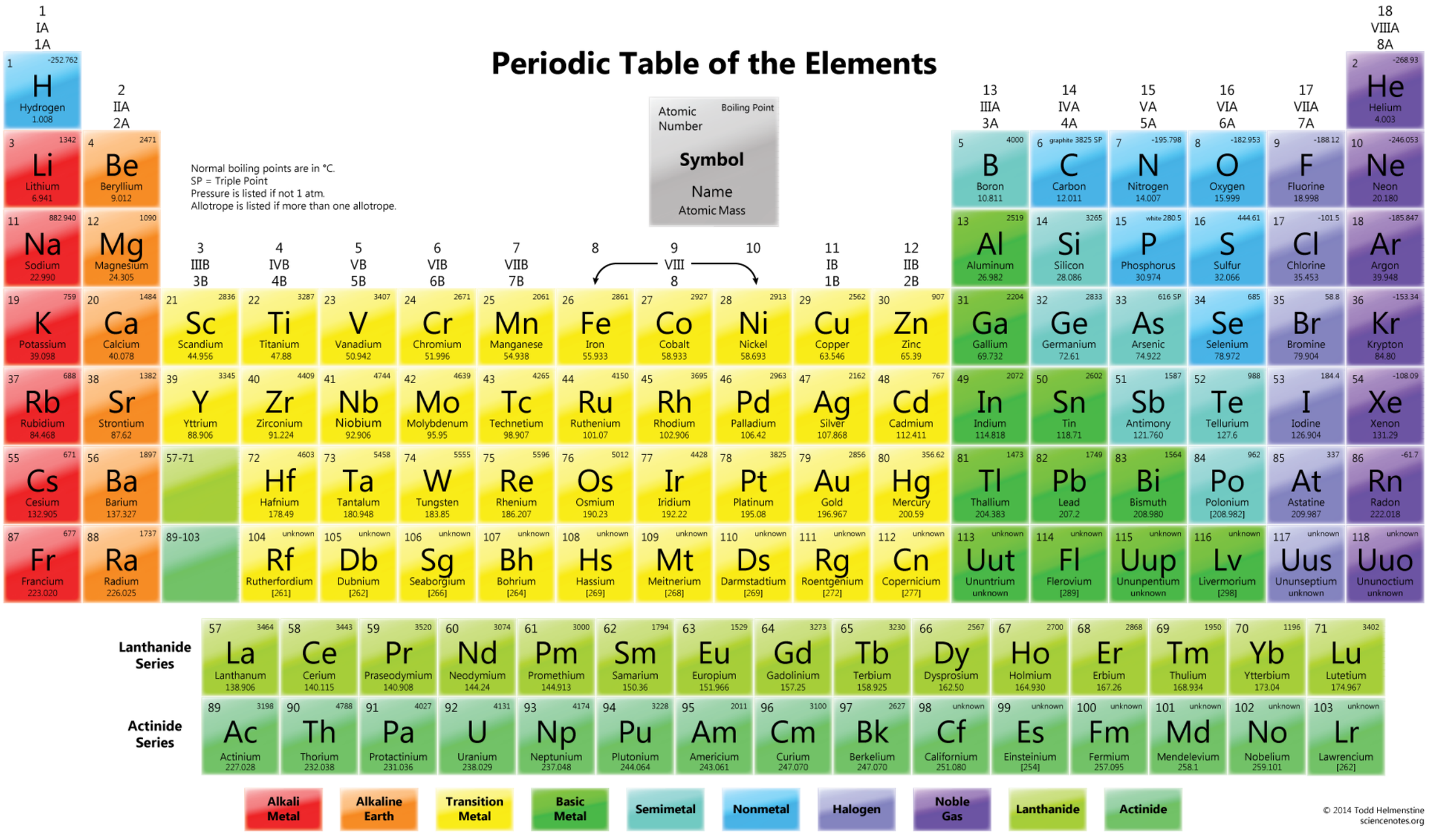
source:
The Go Programming Language
Donovan & Kernighan

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 -188.12 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 -34.04 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 58.8 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	53 184.4 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]	85 337 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 unknown 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

https://tgo.li/onto-over

ThoughtWorks®

A BETTER APPROACH

Fundamental Features of Programming Languages

HANDS-ON 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.

HANDS-ON WITH RACKET (A MODERN 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

ThoughtWorks®

FEATURES

Understand core features, not just syntax and libraries

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®

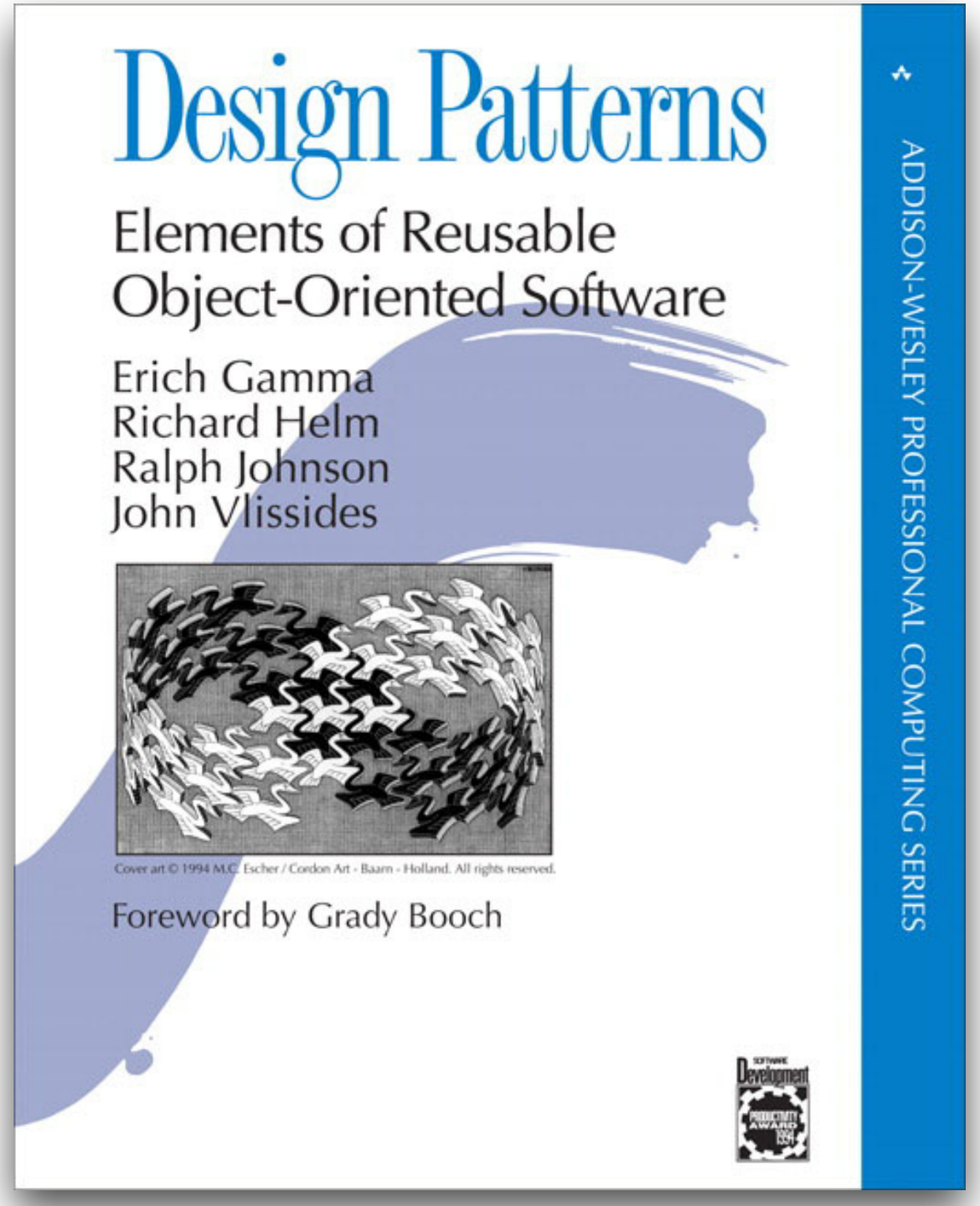
DESIGN PATTERNS

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



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)

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®

ITERABLES & ITERATORS

An excellent abstraction

ITERATION IN C

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for(int i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

ITERATION IN GO

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    for _, arg := range os.Args {
        fmt.Println(arg)
    }
}
```

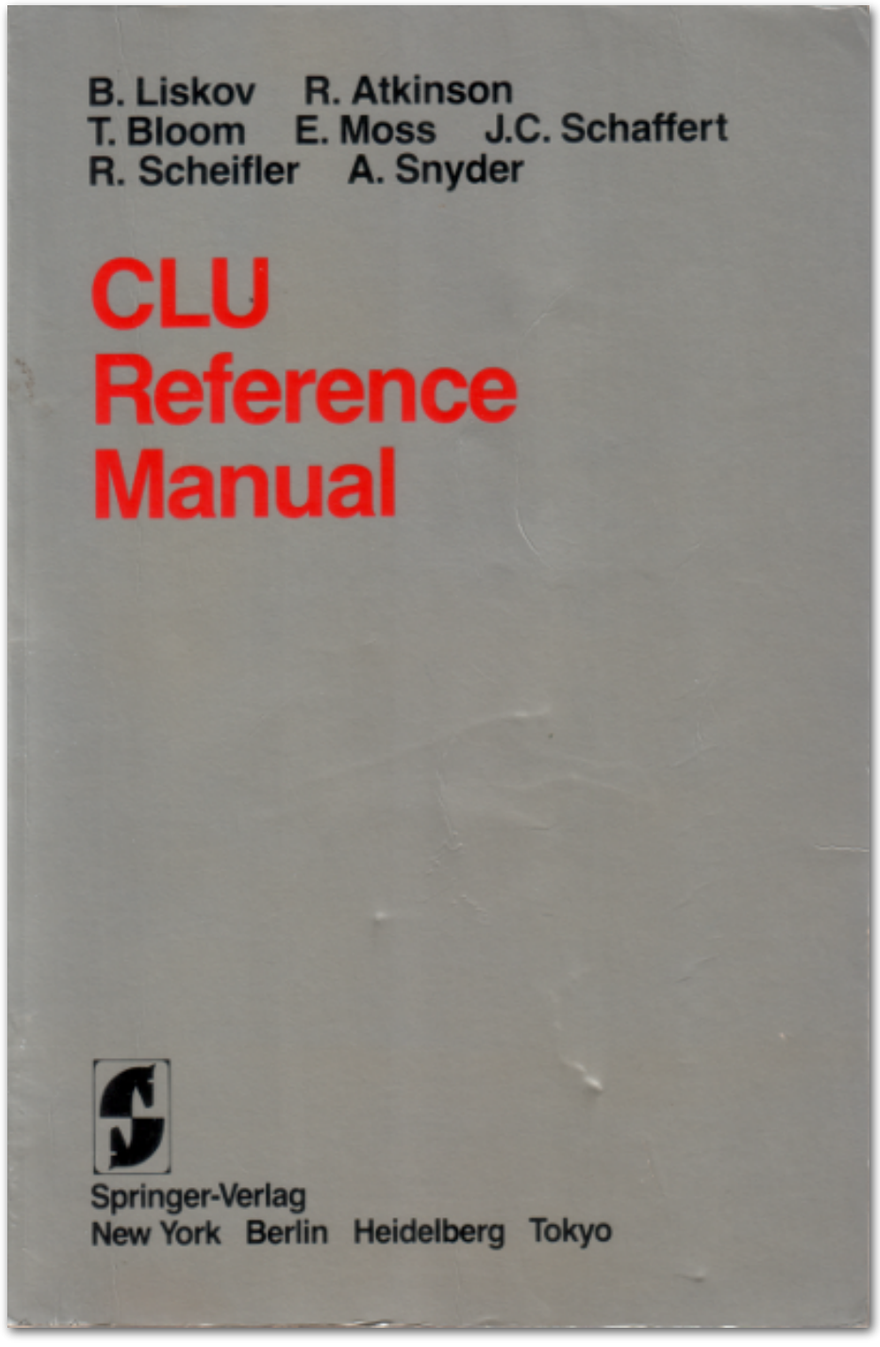
ITERATION IN C AND GO

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for(int i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

```go
func main() {
    for _, arg := range os.Args {
        fmt.Println(arg)
    }
}
```

FOREACH IN BARBARA LISKOV'S CLU



CLU Reference Manual — B. Liskov et. al. — © 1981 Springer-Verlag — also available online from MIT: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-225.pdf

FOREACH IN CLU

year: 1975

types of results can be returned in the exceptional conditions. All information about the names of conditions, and the number and types of arguments and results is described in the *iterator heading*. For example,

```
leaves = iter (t: tree) yields (node)
```

is the heading for an iterator that produces all leaf nodes of a tree object. This iterator might be used in a **for** statement as follows:

```
for leaf: node in leaves(x) do
    ... examine(leaf) ...
end
```

1.3 Clusters

A *cluster* implements a data abstraction, which is a set of objects and a set of primitive *operations* to create and manipulate those objects. The [illegible] or control abstractions. The *cluster heading* states what operations are available, e.g.,

CLU Reference Manual, p. 2
B. Liskov et. al. — © 1981 Springer-Verlag

ITERABLE OBJECTS: THE KEY TO FOREACH

- Python, Java, CLU etc. let programmers define iterable objects

```
for item in an_iterable:
    process(item)
```

- In Go, only **array**, **slice**, **string**, **map**, **channel** are supported by **for/range**

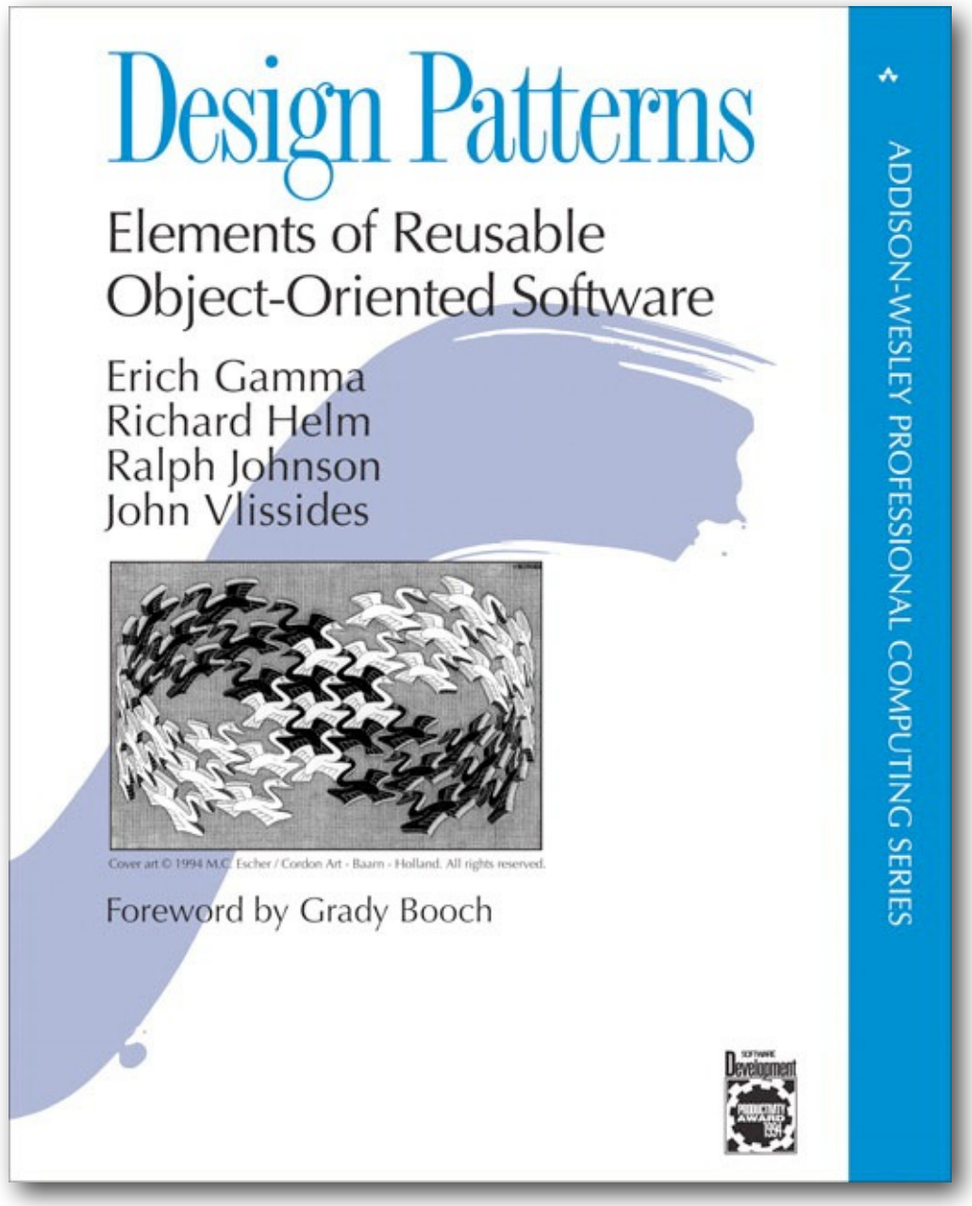
ThoughtWorks®

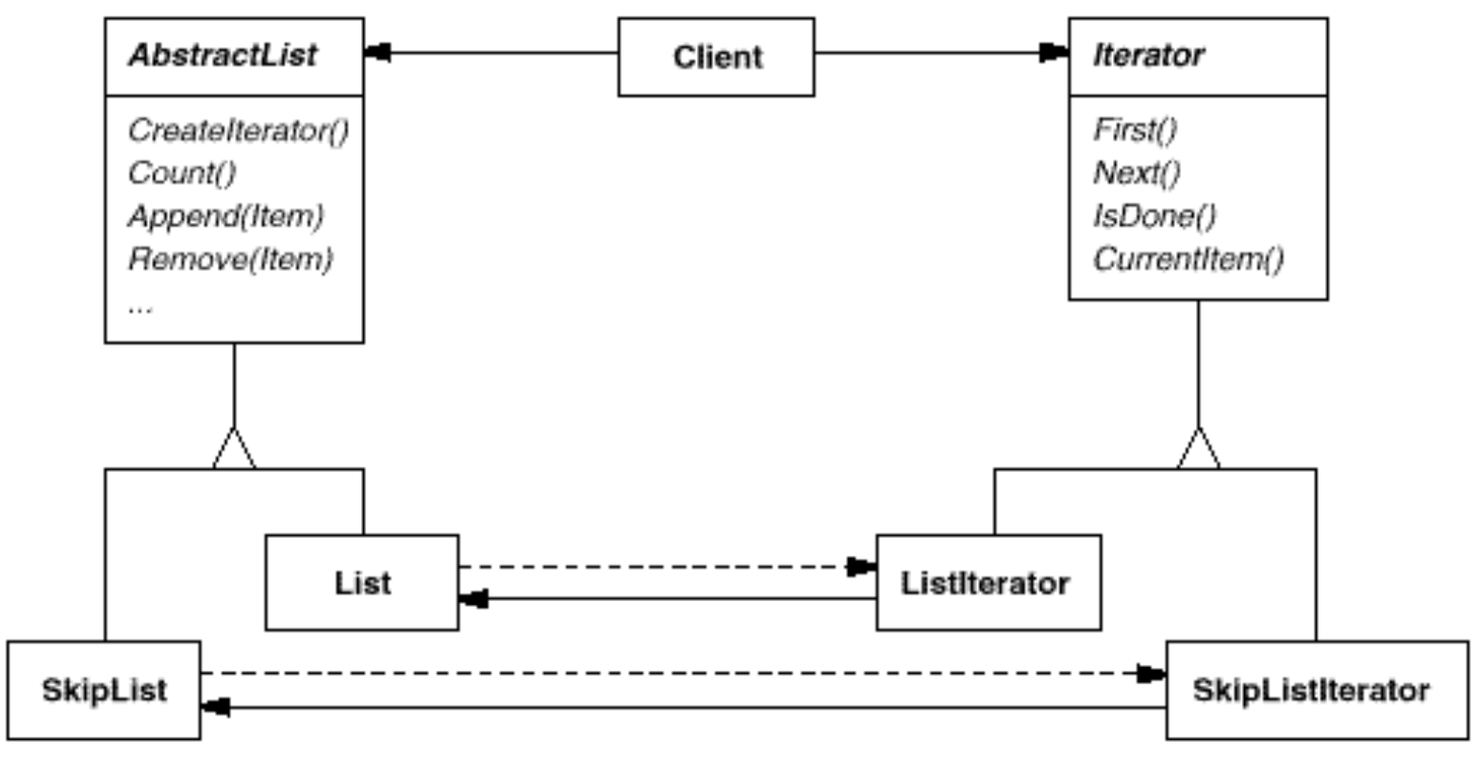
ITERATOR PATTERN

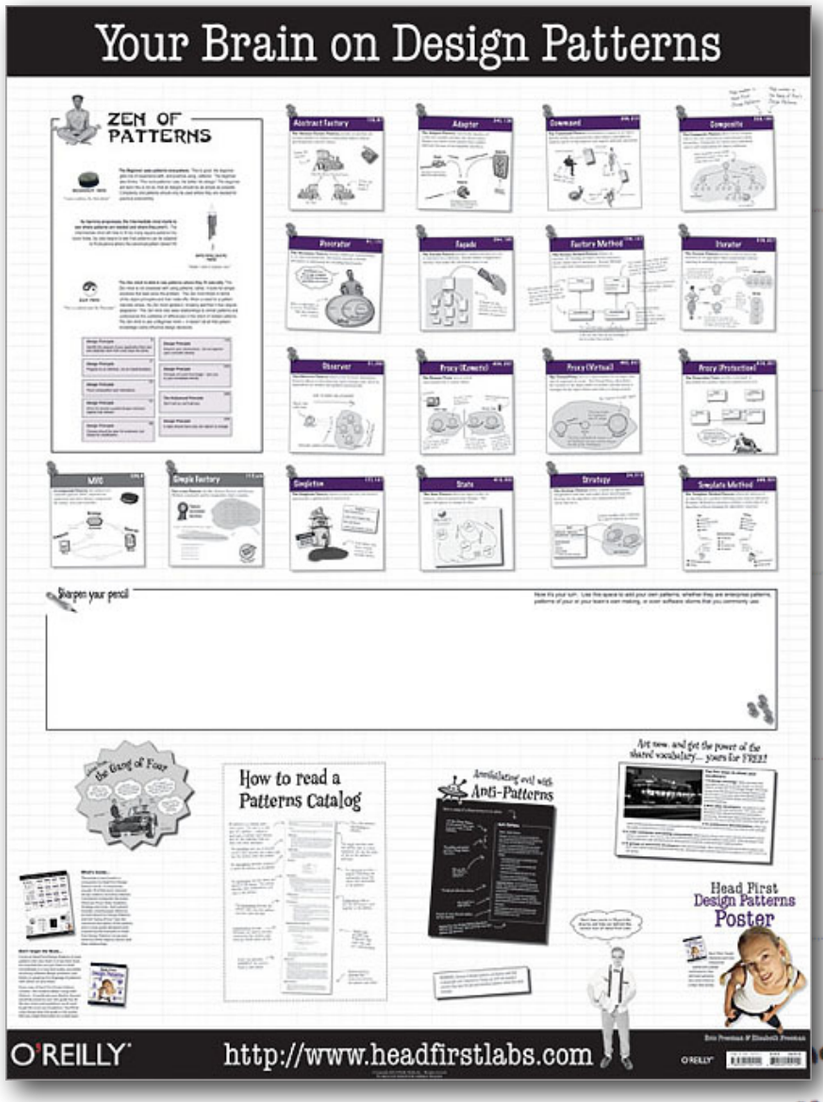
The classic recipe

THE ITERATOR FROM THE GANG OF FOUR

Design Patterns
Gamma, Helm, Johnson & Vlissides
©1994 Addison-Wesley







Head First Design Patterns Poster
O'Reilly
ISBN 0-596-10214-3

336.257

Iterator

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

To iterate over groups ... without knowing the details of how they're implemented, you just need an Iterator for each group...

... one for ArrayList...

built in iterator...

ArrayList

MenuItem 1 | MenuItem 2 | MenuItem 3 | MenuItem 4

next()

Iterator

... and one for Array.

next()

Iterator

... Array doesn't have a built-in Iterator, but you can build your own.

Array

1 MenuItem
2 MenuItem
3 MenuItem
4 MenuItem

You don't have to worry about what implementation is used for the group; you can alway use the same interface – Iterator – to iterate over the items in the group.

ThoughtWorks®

CLASSIC ITERATOR IN GO

The classic recipe in Go

USING A CLASSIC ITERATOR IN GO

for/range not supported 

```go
func Example() {
    sample := []float64{0, 1, -2, 3, -4, 5}
    iterator := NewPositiveIterator(sample)
    for iterator.Advance() {
        fmt.Printf("%0.1f ", iterator.Next())
    }
    // Output: 1.0 3.0 5.0
}
```

IMPLEMENTING A CLASSIC ITERATOR IN GO

Not too bad
(if you like Java)

```go
type PositiveIterator struct {
    target       []float64
    index        int
    currentValue float64
}

func NewPositiveIterator(s []float64) *PositiveIterator {
    return &PositiveIterator{target: s}
}
```

```go
func (p *PositiveIterator) Advance() bool {
    for p.index < len(p.target) {
        if v := p.target[p.index]; v > 0 {
            p.currentValue = v
            return true
        }
        p.index++
    }
    return false
}

func (p *PositiveIterator) Next() float64 {
    p.index++
    return p.currentValue
}
```

ThoughtWorks®

CHANNEL-BASED ITERATOR

Leveraging Go features

USING A CHANNEL-BASED ITERATOR

```
func Example() {
    sample := []float64{0, 1, -2, 3, -4, 5}
    for x := range PositiveIterator(sample) {
        fmt.Printf("%0.1f ", x)
    }
    // Output: 1.0 3.0 5.0
}
```

for/range supported! 

USAGE COMPARISON

```go
func Example() {
    sample := []float64{0, 1, -2, 3, -4, 5}
    for x := range PositiveIterator(sample) {
        fmt.Printf("%0.1f ", x)
    }
    // Output: 1.0 3.0 5.0
}
```

```go
func Example() {
    sample := []float64{0, 1, -2, 3, -4, 5}
    iterator := NewPositiveIterator(sample)
    for iterator.Advance() {
        fmt.Printf("%0.1f ", iterator.Next())
    }
    // Output: 1.0 3.0 5.0
}
```

IMPLEMENTING A CHANNEL-BASED ITERATOR

```go
func PositiveIterator(s []float64) <-chan float64 {
    ch := make(chan float64)
    go func() {
        for _, v := range s {
            if v > 0 {
                ch <- v
            }
        }
        close(ch) // terminate client loop
    }()
    return ch
}
```

IMPLEMENTATION COMPARISON

```go
type PositiveIterator struct {
    target       []float64
    index        int
    currentValue float64
}

func NewPositiveIterator(s []float64) *PositiveIterator {
    return &PositiveIterator{target: s}
}

func (p *PositiveIterator) Advance() bool {
    for p.index < len(p.target) {
        if v := p.target[p.index]; v > 0 {
            p.currentValue = v
            return true
        }
        p.index++
    }
    return false
}

func (p *PositiveIterator) Next() float64 {
    p.index++
    return p.currentValue
}
```

```go
func PositiveIterator(s []float64) <-chan float64 {
    ch := make(chan float64)
    go func() {
        for _, v := range s {
            if v > 0 {
                ch <- v
            }
        }
        close(ch) // terminate client loop
    }()
    return ch
}
```

CONSIDERING THE OPTIONS

The *classic two-method iterator* is more cumbersome to use and to implement.

It is used in the standard library. Ex: **buffo.Scanner** (with methods called **Scan** and **Text**).

Clients can quit at any time; iterator will be garbage-collected.

The *channel-based iterator* is simpler to use and to implement.

However, there is a context switch cost to run a coroutine, and the coroutine may leak if it does not run to completion.

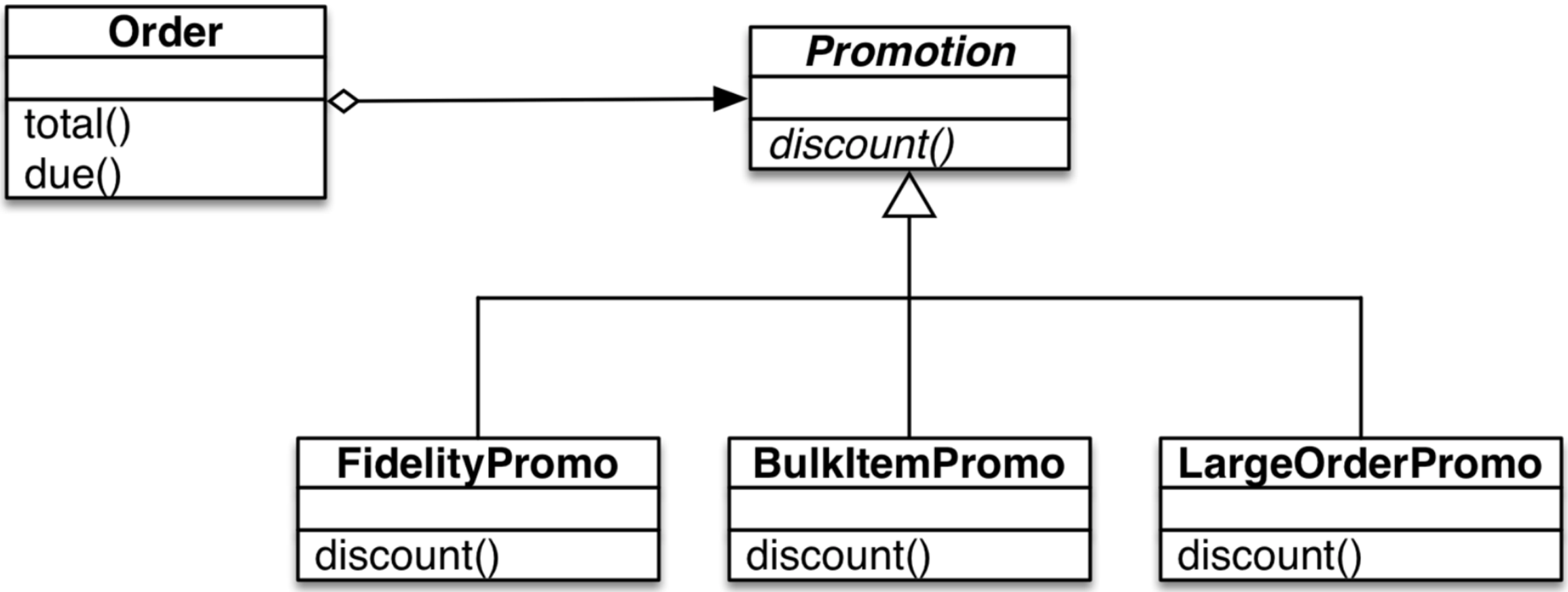
Allowing clients to cancel the iterator requires an additional channel for signaling.

ThoughtWorks®

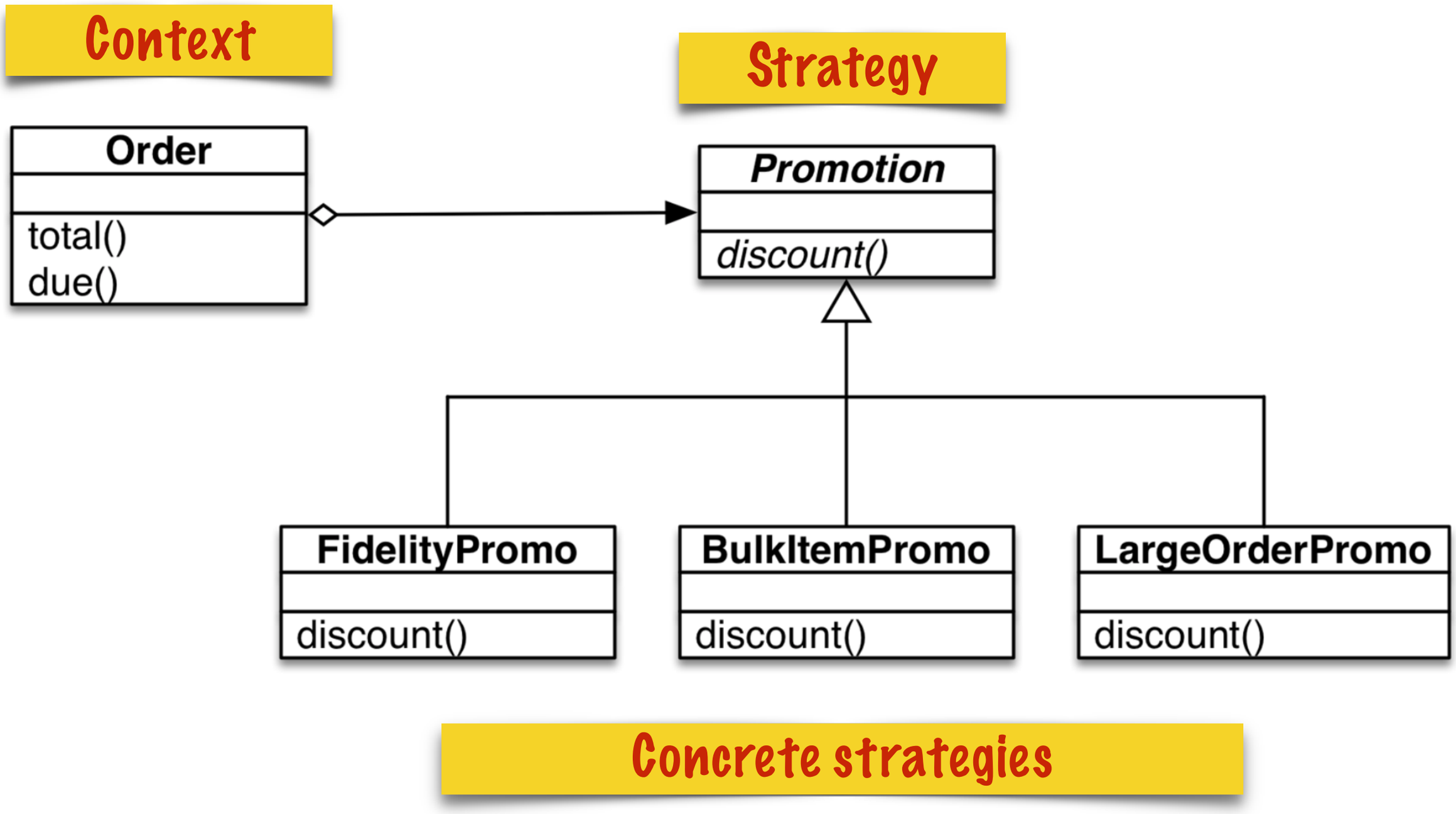
STRATEGY IN GO

Leveraging core features of Go

CHOOSE AN ALGORITHM AT RUN TIME



CHOOSE AN ALGORITHM AT RUN TIME



STRATEGY IN GO

"Classic" implementation with an interface

Implementation with first-class functions

TESTS

https://tgo.li/gobeyond2019

```
22  func TestOrderTotal(t *testing.T) {
23          cart := []LineItem{bananas, apples, mellons}
24          order := Order{joe, cart, FidelityPromo{}}
25          want := 4200
26          got := order.Total()
27          if want != got {
28                  t.Errorf("\nwant: %v;\n total: %v", want, got)
29          }
30  }
31
32  func TestNoFidelityOrder(t *testing.T) {
33          cart := []LineItem{bananas, apples, mellons}
34          order := Order{joe, cart, FidelityPromo{}}
35          want := 4200
36          got := order.Due()
37          if want != got {
38                  t.Errorf("\nwant: %v;\n got: %v", want, got)
39          }
40  }
41
```

interface/strategy_test.go

TESTS (CONT.)

https://tgo.li/gobeyond2019

```
42  func TestFidelityOrder(t *testing.T) {
43          customer := Customer{"Ann Smith", 1000}
44          cart := []LineItem{bananas, apples, mellons}
45          order := Order{customer, cart, FidelityPromo{}}
46          want := 3990
47          got := order.Due()
48          if want != got {
49                  t.Errorf("\nwant: %v;\n got: %v", want, got)
50          }
51  }
52
53  func TestBulkItemOrder(t *testing.T) {
54          cart := []LineItem{{"banana", 30, 50}, apples}
55          order := Order{joe, cart, BulkItemPromo{}}
56          want := 2850
57          got := order.Due()
58          if want != got {
59                  t.Errorf("\nwant: %v;\n got: %v", want, got)
60          }
61  }
62
```

interface/strategy_test.go

CUSTOMER AND LINE ITEM

https://tgo.li/gobeyond2019

```
1   package main
2
3   import (
4           "fmt"
5   )
6
7   // Customer has name and fidelity points
8   type Customer struct {
9           Name     string
10          Fidelity int
11  }
12
13  // LineItem represents one item in an Order
14  type LineItem struct {
15          Product  string
16          Quantity int
17          Price    int // cents
18  }
19
20  func (li LineItem) Total() int {
21          return li.Quantity * li.Price
22  }
```

interface/store.go

CONTEXT AND STRATEGY

https://tgo.li/gobeyond2019

```
3   // Order is the context where a Discounter strategy is used
4   type Order struct {
5           Customer  Customer
6           Cart      []LineItem
7           Promotion Discounter
8   }
9
10  type Discounter interface {
11          Discount(Order) int
12  }
13
14  func (o Order) Total() int {
15          total := 0
16          for _, item := range o.Cart {
17                  total += item.Total()
18          }
19          return total
20  }
21
22  func (o Order) Due() int {
23          return o.Total() - o.Promotion.Discount(o)
24  }
```

Context

Strategy

interface/strategy.go

CONCRETE STRATEGY

https://tgo.li/gobeyond2019

```
30  type FidelityPromo struct{}
31
32  // 5% discount for customers with at least 1000 fidelity points
33  func (p FidelityPromo) Discount(o Order) int {
34          if o.Customer.Fidelity >= 1000 {
35                  return o.Total() / 20 // %5
36          }
37          return 0
38  }
```

interface/store.go

MORE CONCRETE STRATEGIES

https://tgo.li/gobeyond2019

```
40  type BulkItemPromo struct{}
41
42  // 10% discount for each line item with at least 20 units
43  func (p BulkItemPromo) Discount(o Order) int {
44          discount := 0
45          for _, item := range o.Cart {
46                  if item.Quantity >= 20 {
47                          discount += item.Total() / 10
48                  }
49          }
50          return discount
51  }
52
53  type LargeOrderPromo struct{}
54
55  // 7% discount for orders with 10 or more items
56  func (p LargeOrderPromo) Discount(o Order) int {
57          if len(o.Cart) >= 10 {
58                  return o.Total() * 7 / 100
59          }
60          return 0
61  }
```

interface/store.go

STRATEGY WITH FIRST CLASS FUNCTIONS

interface | functions 2

A strategy > interface > strategy.go

```
1  package main
2
3  // Order is the context where a Discounter strategy is
   used
4  type Order struct {
5      Customer  Customer
6      Cart      []LineItem
7      Promotion Discounter
8  }
9
10 type Discounter interface {
11     Discount(Order) int
12 }
13
14 func (o Order) Total() int {
15     total := 0
16     for _, item := range o.Cart {
17         total += item.Total()
18     }
19     return total
20 }
21
22 func (o Order) Due() int {
23     return o.Total() - o.Promotion.Discount(o)
24 }
25
```

strategy.go (interface) | strategy.go (functions) 2

B strategy > functions > strategy.go

```
1  package main
2
3  // Order is the context where a promotion discount
   strategy is used
4  type Order struct {
5      Customer  Customer
6      Cart      []LineItem
7      Promotion func(Order) int
8  }
9
10 func (o Order) Total() int {
11     total := 0
12     for _, item := range o.Cart {
13         total += item.Total()
14     }
15     return total
16 }
17
18 func (o Order) Due() int {
19     return o.Total() - o.Promotion(o)
20 }
21
```

Blocks | Fluid | Unified Ignore whitespace Change 1 of 4

TESTS WITH FUNCTIONS

interface | functions 2

A strategy > interface > strategy_test.go

```
16          t.Errorf("\nwant: %q;\n got: %q", want, got)
17      }
18  }
19
20  var joe = Customer{"John Doe", 0}
21
22  func TestOrderTotal(t *testing.T) {
23      cart := []LineItem{bananas, apples, mellons}
24      order := Order{joe, cart, FidelityPromo{}}
25      want := 4200
26      got := order.Total()
27      if want != got {
28          t.Errorf("\nwant: %v;\n total: %v", want, got)
29      }
30  }
31
32  func TestNoFidelityOrder(t *testing.T) {
33      cart := []LineItem{bananas, apples, mellons}
34      order := Order{joe, cart, FidelityPromo{}}
35      want := 4200
36      got := order.Due()
37      if want != got {
38          t.Errorf("\nwant: %v;\n got: %v", want, got)
39      }
40  }
41
42  func TestFidelityOrder(t *testing.T) {
43      customer := Customer{"Ann Smith", 1000}
44      cart := []LineItem{bananas, apples, mellons}
45      order := Order{customer, cart, FidelityPromo{}}
46      want := 3990
```

strategy_test.go (interface) | strategy_test.go (functions) 2

B strategy > functions > strategy_test.go

```
16          t.Errorf("\nwant: %q;\n got: %q", want, got)
17      }
18  }
19
20  var joe = Customer{"John Doe", 0}
21
22  func TestOrderTotal(t *testing.T) {
23      cart := []LineItem{bananas, apples, mellons}
24      order := Order{joe, cart, FidelityPromo}
25      want := 4200
26      got := order.Total()
27      if want != got {
28          t.Errorf("\nwant: %v;\n total: %v", want, got)
29      }
30  }
31
32  func TestNoFidelityOrder(t *testing.T) {
33      cart := []LineItem{bananas, apples, mellons}
34      order := Order{joe, cart, FidelityPromo}
35      want := 4200
36      got := order.Due()
37      if want != got {
38          t.Errorf("\nwant: %v;\n got: %v", want, got)
39      }
40  }
41
42  func TestFidelityOrder(t *testing.T) {
43      customer := Customer{"Ann Smith", 1000}
44      cart := []LineItem{bananas, apples, mellons}
45      order := Order{customer, cart, FidelityPromo}
46      want := 3990
```

Blocks | Fluid | Unified | Ignore whitespace | Change 1 of 5

CONCRETE STRATEGIES AS FUNCTIONS

interface | functions 2 | store.go (interface) | store.go (functions) 2 | strategy_test.go (interface) | strategy_test.go (functions) 2

A: strategy / interface / store.go

```
30 type FidelityPromo struct{}
31
32 // 5% discount for customers with at least 1000
   fidelity points
33 func (p FidelityPromo) Discount(o Order) int {
34     if o.Customer.Fidelity >= 1000 {
35         return o.Total() / 20 // %5
36     }
37     return 0
38 }
39
40 type BulkItemPromo struct{}
41
42 // 10% discount for each line item with at least 20
   units
43 func (p BulkItemPromo) Discount(o Order) int {
44     discount := 0
45     for _, item := range o.Cart {
46         if item.Quantity >= 20 {
47             discount += item.Total() / 10
48         }
49     }
50     return discount
51 }
52
53 type LargeOrderPromo struct{}
54
55 // 7% discount for orders with 10 or more items
56 func (p LargeOrderPromo) Discount(o Order) int {
57     if len(o.Cart) >= 10 {
58         return o.Total() * 7 / 100
```

B: strategy / functions / store.go

```
28 }
29
30 // 5% discount for customers with at least 1000
   fidelity points
31 func FidelityPromo(o Order) int {
32     if o.Customer.Fidelity >= 1000 {
33         return o.Total() / 20 // %5
34     }
35     return 0
36 }
37
38 // 10% discount for each line item with at least 20
   units
39 func BulkItemPromo(o Order) int {
40     discount := 0
41     for _, item := range o.Cart {
42         if item.Quantity >= 20 {
43             discount += item.Total() / 10
44         }
45     }
46     return discount
47 }
48
49 // 7% discount for orders with 10 or more items
50 func LargeOrderPromo(o Order) int {
51     if len(o.Cart) >= 10 {
52         return o.Total() * 7 / 100
53     }
54     return 0
55 }
56
```

Blocks | Fluid | Unified — Ignore whitespace — Change 2 of 6

ThoughtWorks®

WHICH IS MORE IDIOMATIC GO?

Interfaces v. Functions

ThoughtWorks®

WHICH IS MORE GOISH?

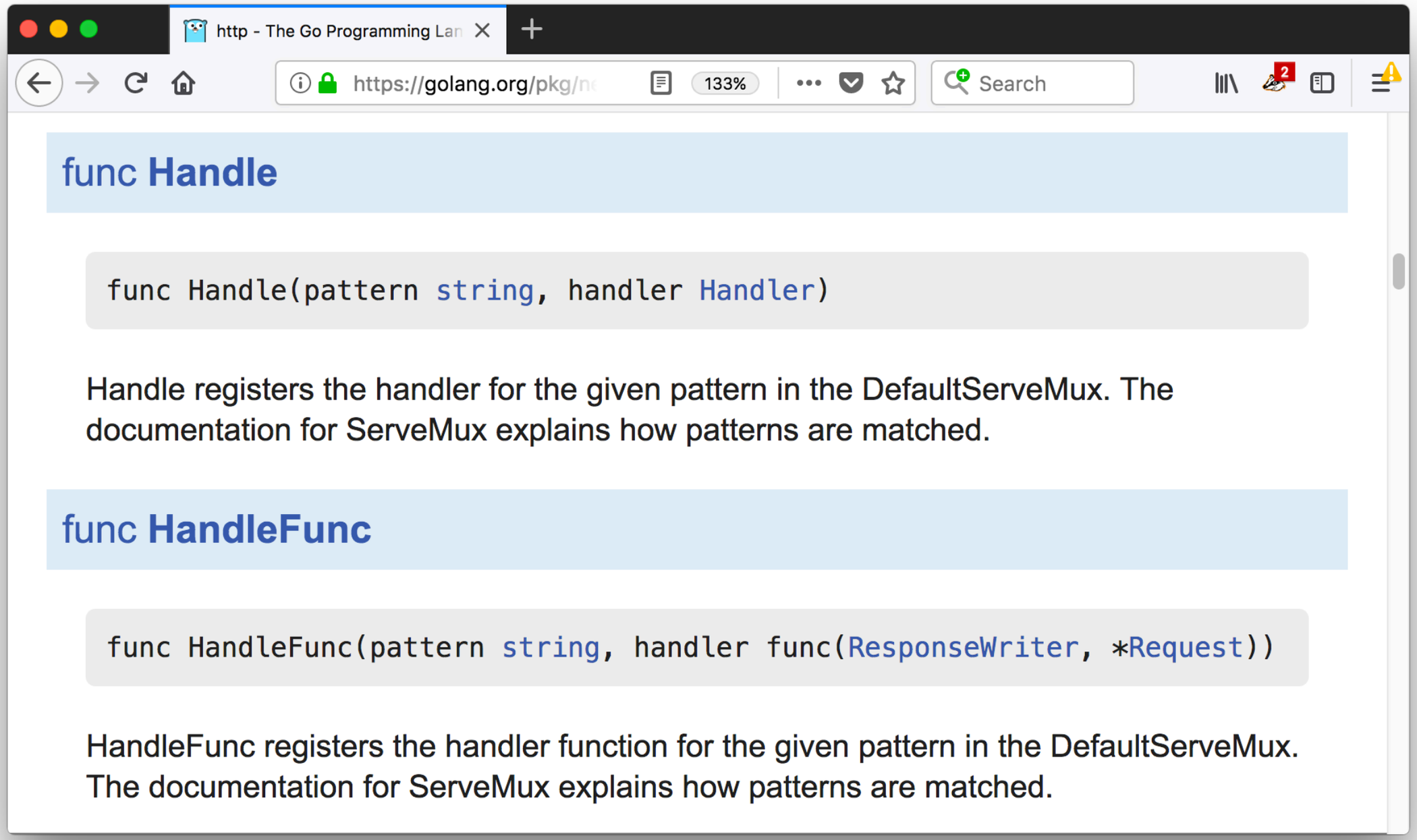
Interfaces v. Functions

INTERFACES V. FIRST CLASS FUNCTIONS

In the standard library, one-method interfaces are common.

INTERFACES V. FIRST CLASS FUNCTIONS

But the http package supports both ways*:



*Kumar Iyer (ThoughtWorks): Higher-order functions vs interfaces in golang http://bit.ly/2j39uKh

ThoughtWorks®

WRAPPING UP

Why learn the fundamentals

WHY LEARN THE FUNDAMENTALS*

- Learn new languages more easily
- Make the most of language features
- Know how to choose alternative ways of implementation
- Make good use of design patterns
- Debug complicated bugs
- Mimic useful features in languages where they are missing

* Inspired by Programming Language Pragmatics

  Michael L. Scott

VIELEN DANK !

Luciano Ramalho
luciano.ramalho@thoughtworks.com

Twitter: @ramalhoorg
Github: https://github.com/standupdev
Slides: speakerdeck.com/ramalho

ThoughtWorks®