



*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*

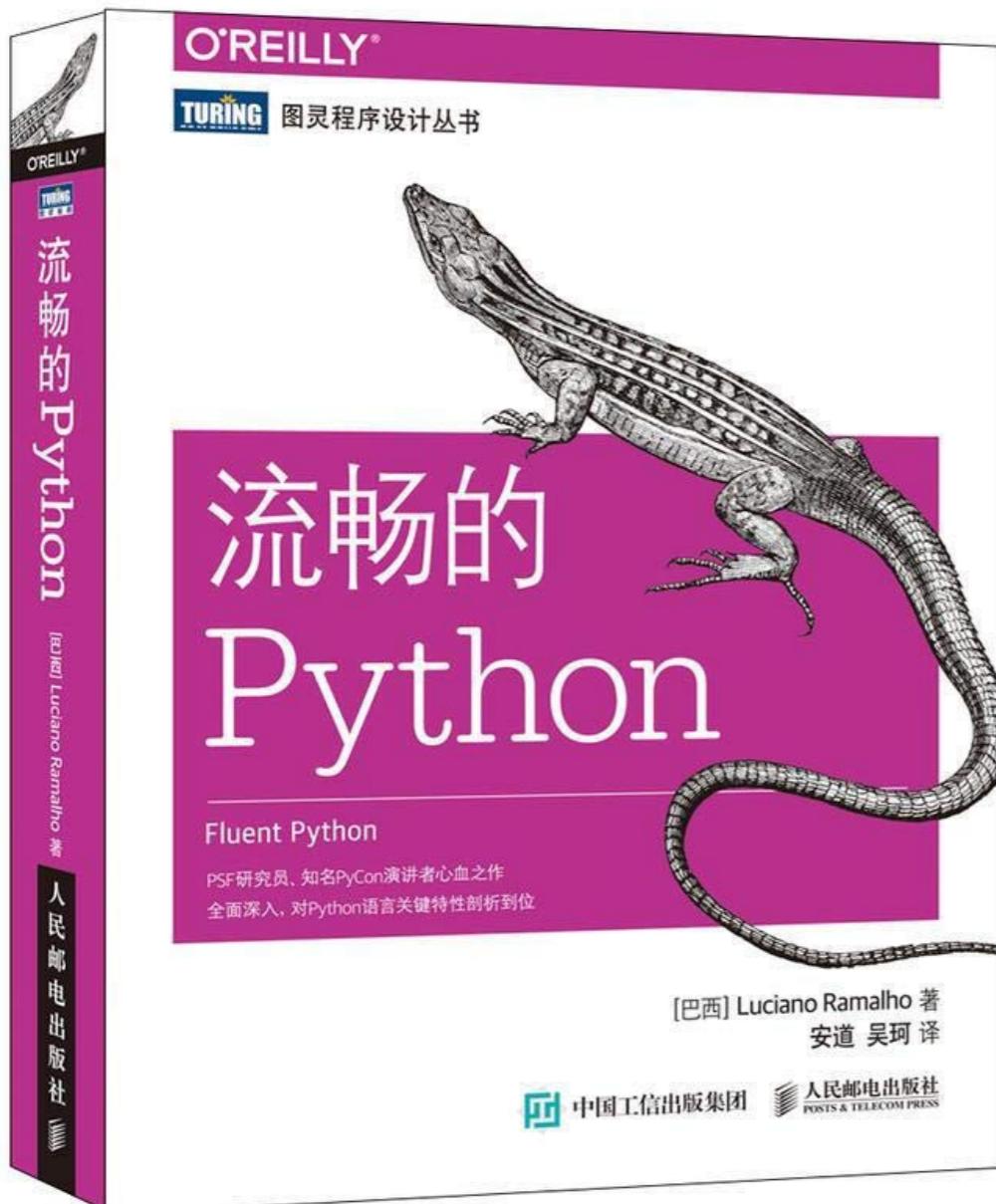
---

**@ramalhoorg**  
*luciano.ramalho@thoughtworks.com*

# FLUENT PYTHON



5 stars at [amazon.de](#)



Published in 9 languages:

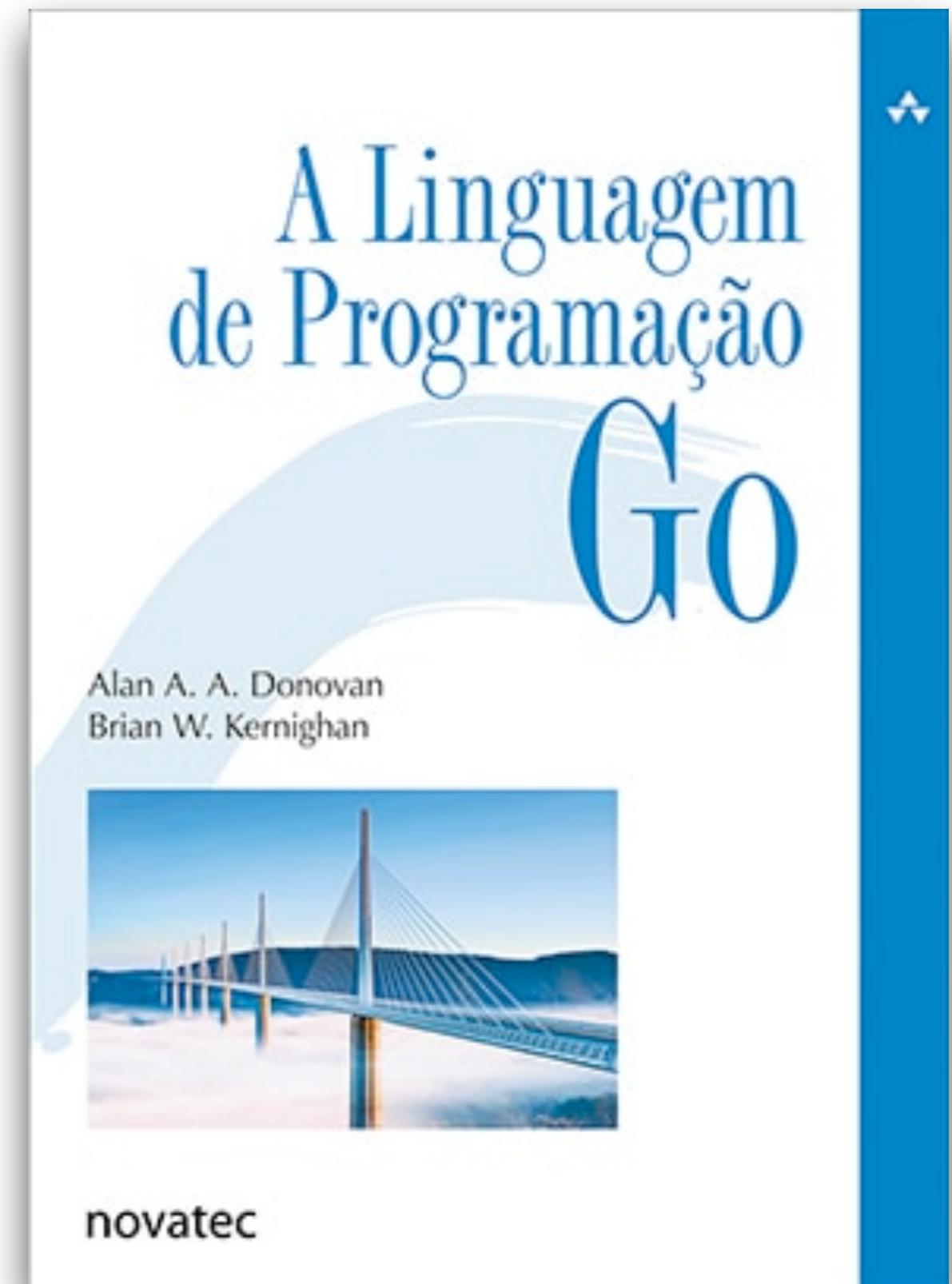
- Chinese (simplified)
- Chinese (traditional)
- English
- French
- Japanese
- Korean
- Polish
- Brazilian Portuguese
- Russian

2<sup>nd</sup> edition: late 2020

# TECH REVIEWER OF THE BRAZILIAN EDITION OF GOPL

---

GOPL:  
The Go Programming  
Language  
Donovan & Kernighan



# WORKSHOP: INTRODUCING GO WITH TDD

Workshop: Introducing Go with TDD

https://www.meetup.com/Women-Techmakers-Berlin/events/26545

Meetup https://tgo.li/wtm-tdd-go

Saturday, October 12, 2019

**Workshop: Introducing Go with TDD**

Hosted by Natalie Pistunovich  
From Women Techmakers Berlin  
Public group

You're going 38 people going

✓ X ★

Share: [f](#) [t](#) [in](#)

**Details**

Women Techmakers Berlin and Women Who Go Berlin are joining forces to bring you this great workshop:

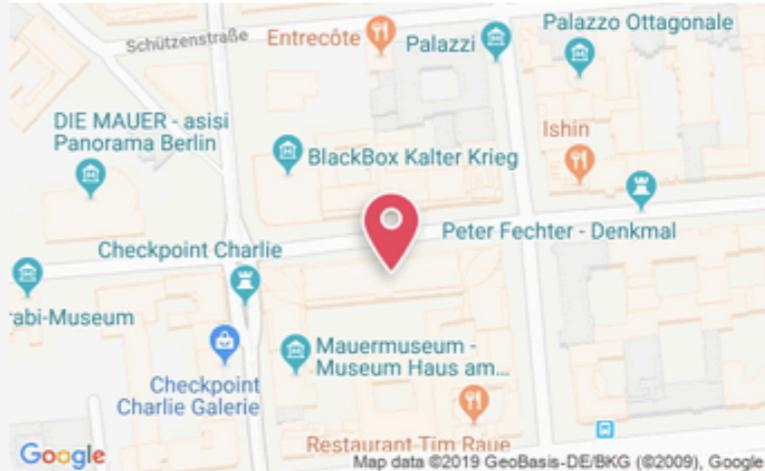
Most Go tutorials and introductory texts leave testing to the end; many just skip the topic. However, knowing how to test Go code is a key job requirement. It can also help you master Go faster by letting you easily test your hypotheses as you practice the language.

This workshop introduces Go and Test-Driven Design (TDD) together. The first code example is a test, the second is a function that makes the test pass.

No prior knowledge of Go is required, but participants are expected to know how to program in a high-level programming language.

Saturday, October 12, 2019  
10:00 AM to 5:00 PM  
[Add to calendar](#)

ThoughtWorks Berlin Deutschland G...  
Zimmerstraße 23 · Berlin

A map of Berlin, Germany, showing the location of ThoughtWorks Berlin. The map highlights several landmarks and streets, including Schützenstraße, Entrecôte, Palazzi, Palazzo Ottagonale, Ishin, Peter Fechter - Denkmal, Checkpoint Charlie, and the Mauermuseum. A red location pin marks the exact spot of the workshop.

**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*).<sup>[1][2]</sup>

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.<sup>[3][4][5]</sup>

### Influences of the list [edit]

This list has stirred considerable philosophical and literary commentary.

Michel Foucault begins his preface to *The Order of Things*.<sup>[6]</sup>

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.<sup>[7]</sup> 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.<sup>[8]</sup>

Keith Windschitl, 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<sup>[9]</sup> 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.<sup>[10]</sup>

Borges himself questions the veracity of the quote in his essay, referring to "the unknown (or false) Chinese encyclopaedia writer".<sup>[11]</sup>

### 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. ^ A ^ B Borges, Darwin-L (mailing list archive), RJ Obara, 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* (in English), University of Chicago Press, ISBN 0-226-46804-6
9. ^ Windschitl, 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	<p><i>A Universal History of Infamy</i> "On Exactitude in Science"  <i>Ficciones</i> "Tón, Ugar, Orbis Teritus" · "The Approach to Al-Mutashim" · "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 Immortal" · "The Dead Man" · "The Theologians" · "Emma Zunz" · "The House of Asterion" · "Deutsche Requiem" · "Averroes's Search" · "The Zahir" · "The Writing of the God" · "The Two Kings and the Two Labyrinths" · "The Walt" · "The Aleph"</p> <p><i>The Aleph</i></p> <p><i>Otras Inquisiciones (1937–1952)</i> "The Analytical Language of John Wilkins"</p> <p><i>Dreamtigers</i> "Borges and I"</p> <p><i>Dr. Brodie's Report</i> "The Encounter" · "The Gospel According to Mark"</p> <p><i>The Book of Sand</i> "The Other" · "Ulrike" · "The Congress" · "There Are More Things" · "The Disk" · "The Book of Sand" · "Shakespeare's Memory" · "Blue Tigers" · "Shakespeare's Memory"</p>	
Other works	História de la eternidad" · "A New Refutation of Time" · Borges en Martín Fierro · "El Golem" · Book of Imaginary Beings · Labyrinths · Adrogue, con ilustraciones de Norah Borges	
Related	Leonor Acevedo Suárez (mother) · Jorge Guillermo Borges (father) · Norah Borges (sister) · <i>Celestial Emporium of Benevolent Knowledge</i> · H. Bustos Domecq · Pedro Mata · Ubac - Borges and mathematics	

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

This page was last edited on 30 September 2019, at 03:02 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

Privacy policy About Wikipedia Disclaimers Contact Wikipedia Developers Cookie statement Mobile view

Wikimedia project

Powered by MediaWiki

[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](#)[Article](#) [Talk](#)[Read](#) [Edit](#) [View history](#)[Search Wikipedia](#)

# 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*).<sup>[1][2]</sup>

[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](#).<sup>[3][4][5]</sup>

## 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



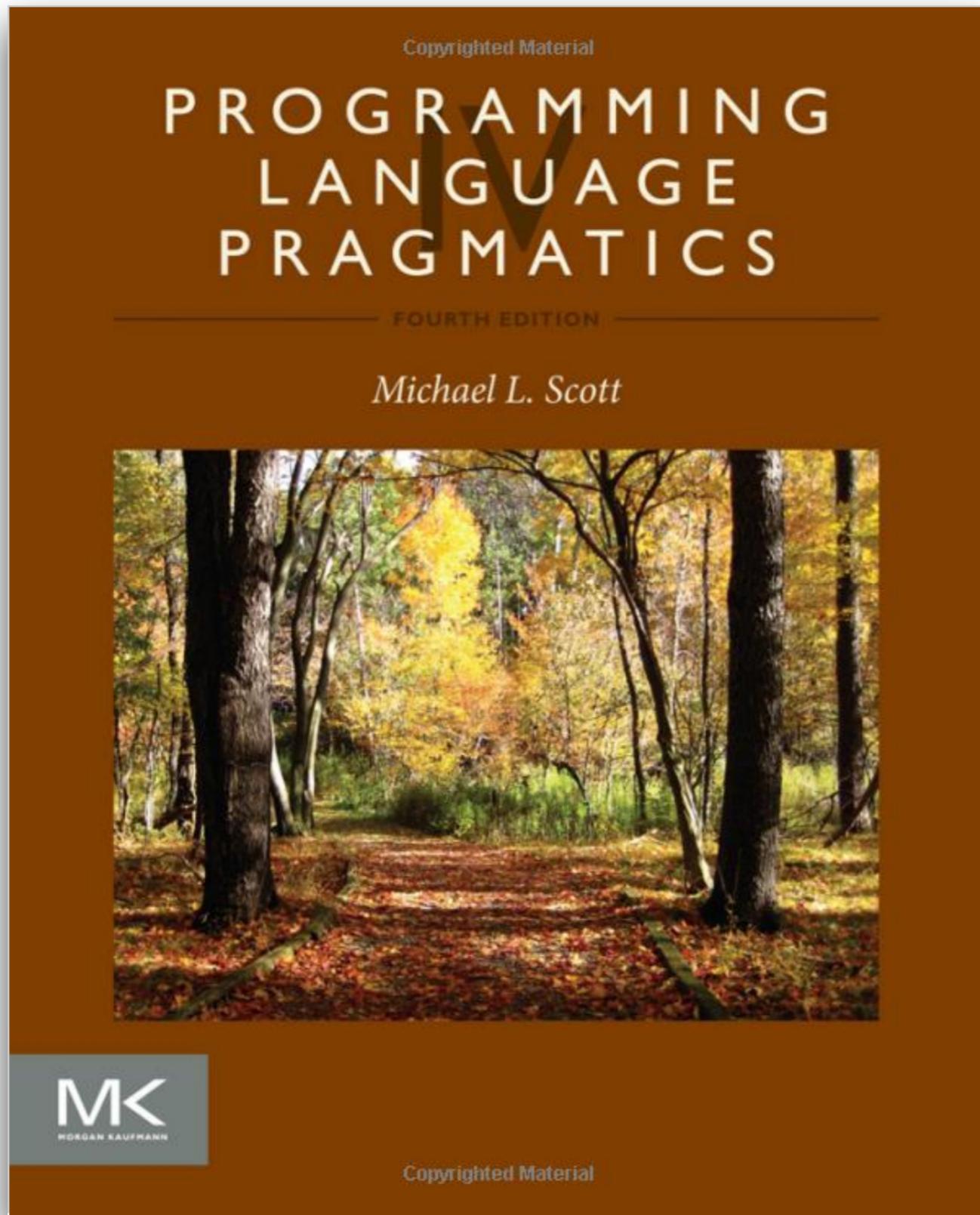
# PARADIGMS

---

Programming language categories

# A GOOD SURVEY-STYLE TEXTBOOK

---



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 IN C, OCAML, PROLOG

---

```
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 I.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

---

```
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

# 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)

---

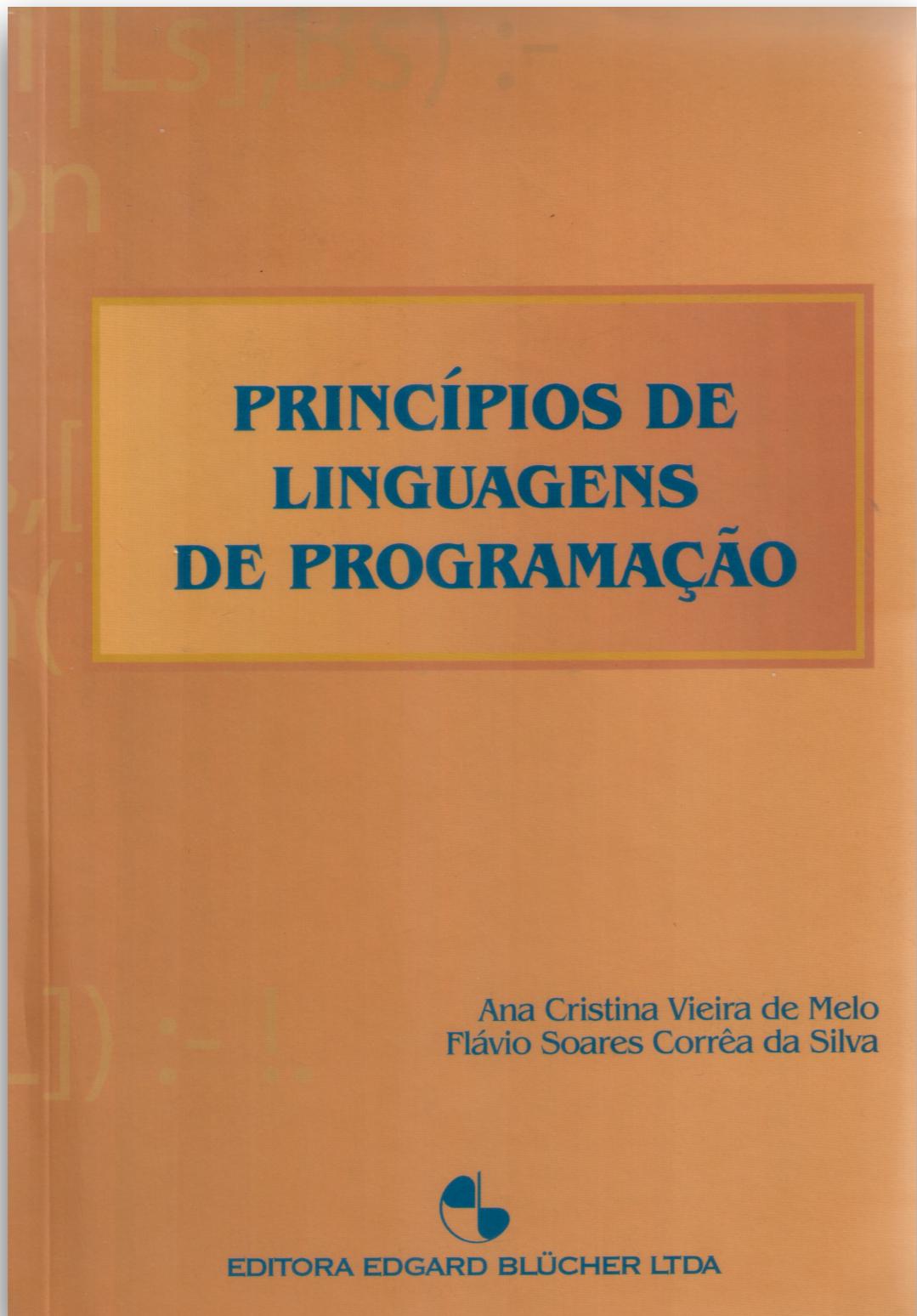
|                         |                              |
|-------------------------|------------------------------|
| declarative             |                              |
| functional              | Lisp/Scheme, ML, Haskell     |
| dataflow                | Id, Val                      |
| logic, 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.

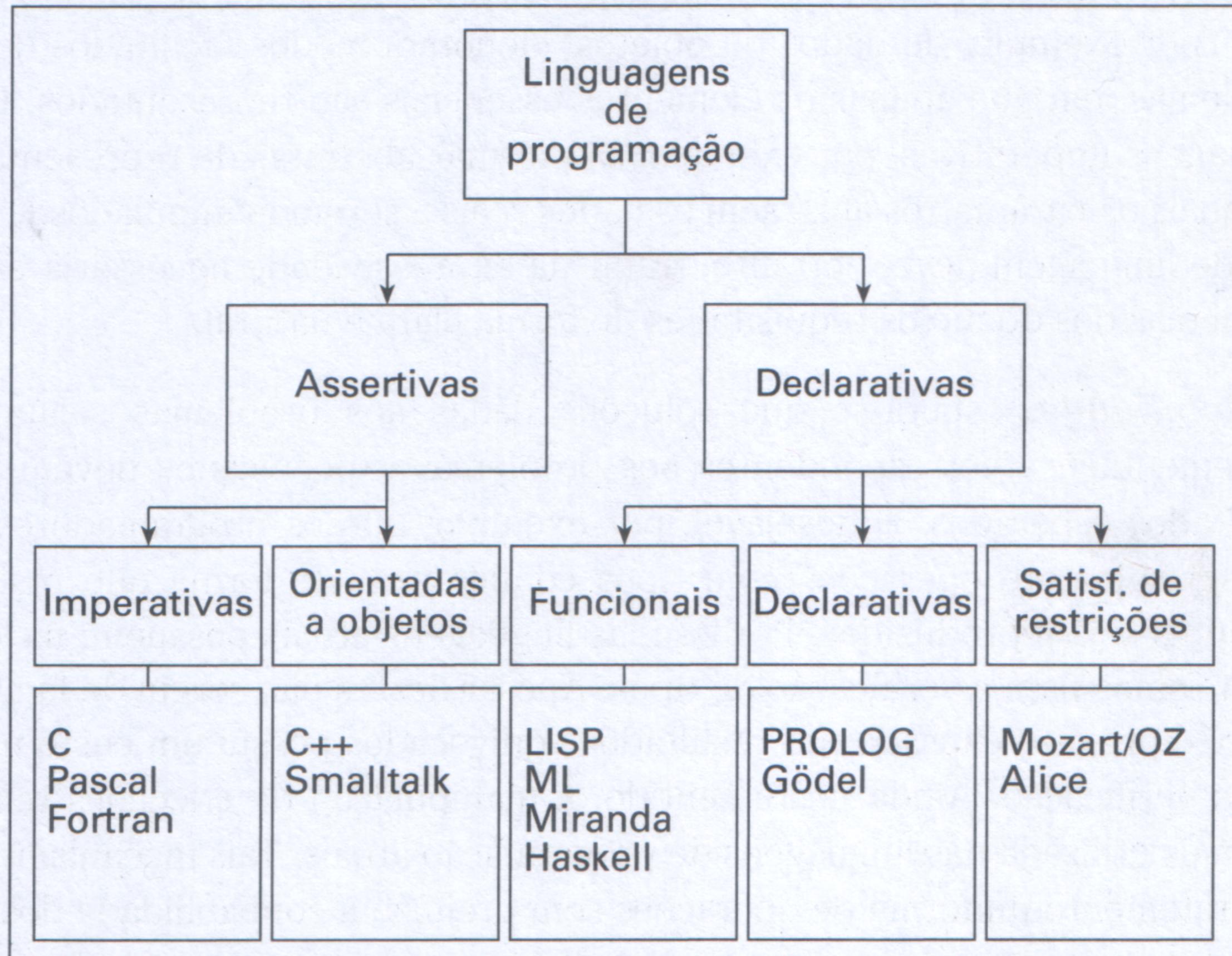
# ANOTHER BOOK

---

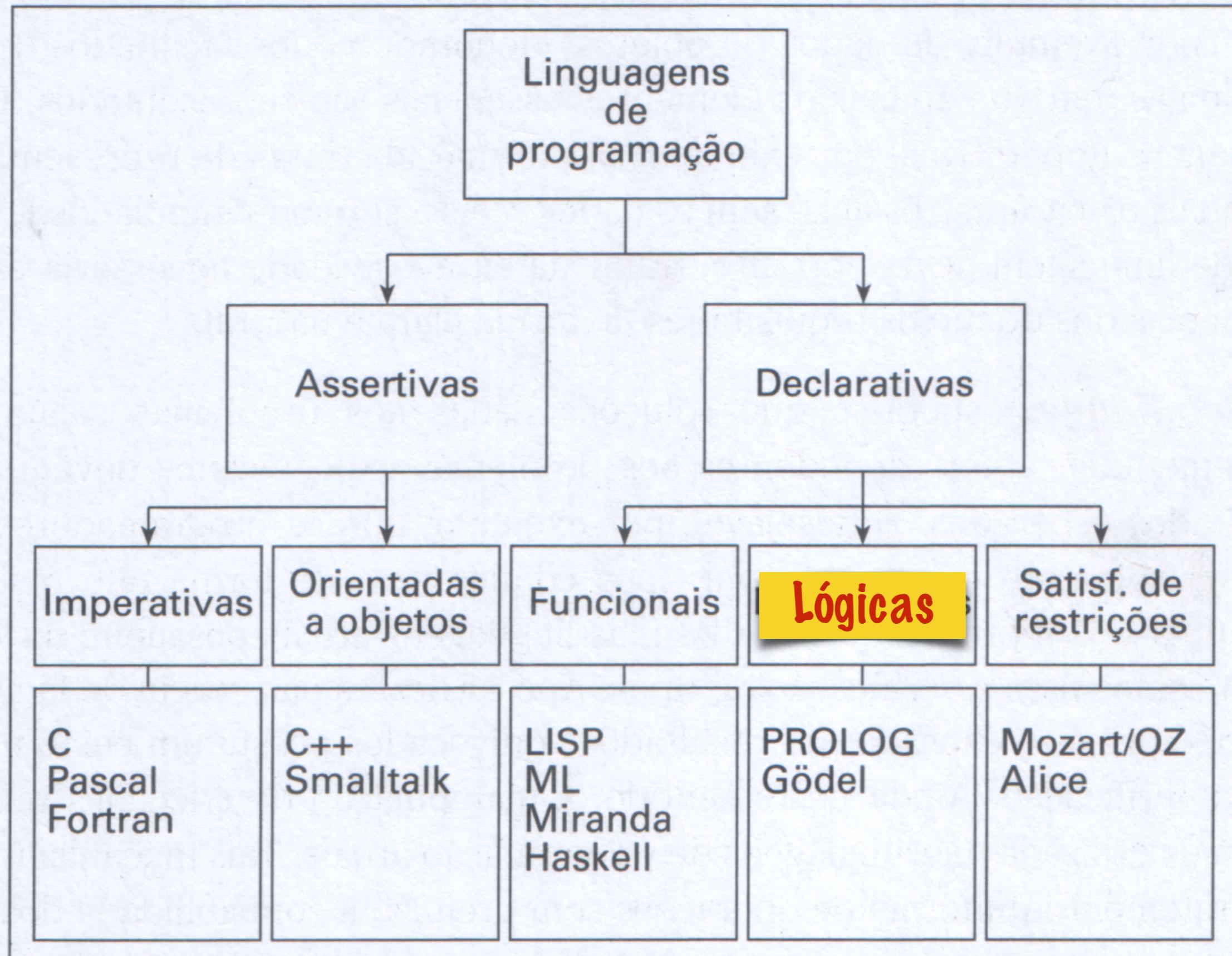


Princípios de Linguagens de  
Programação  
(2003)

Ana Cristina Vieira de Melo  
Flávio Soares Corrêa da Silva



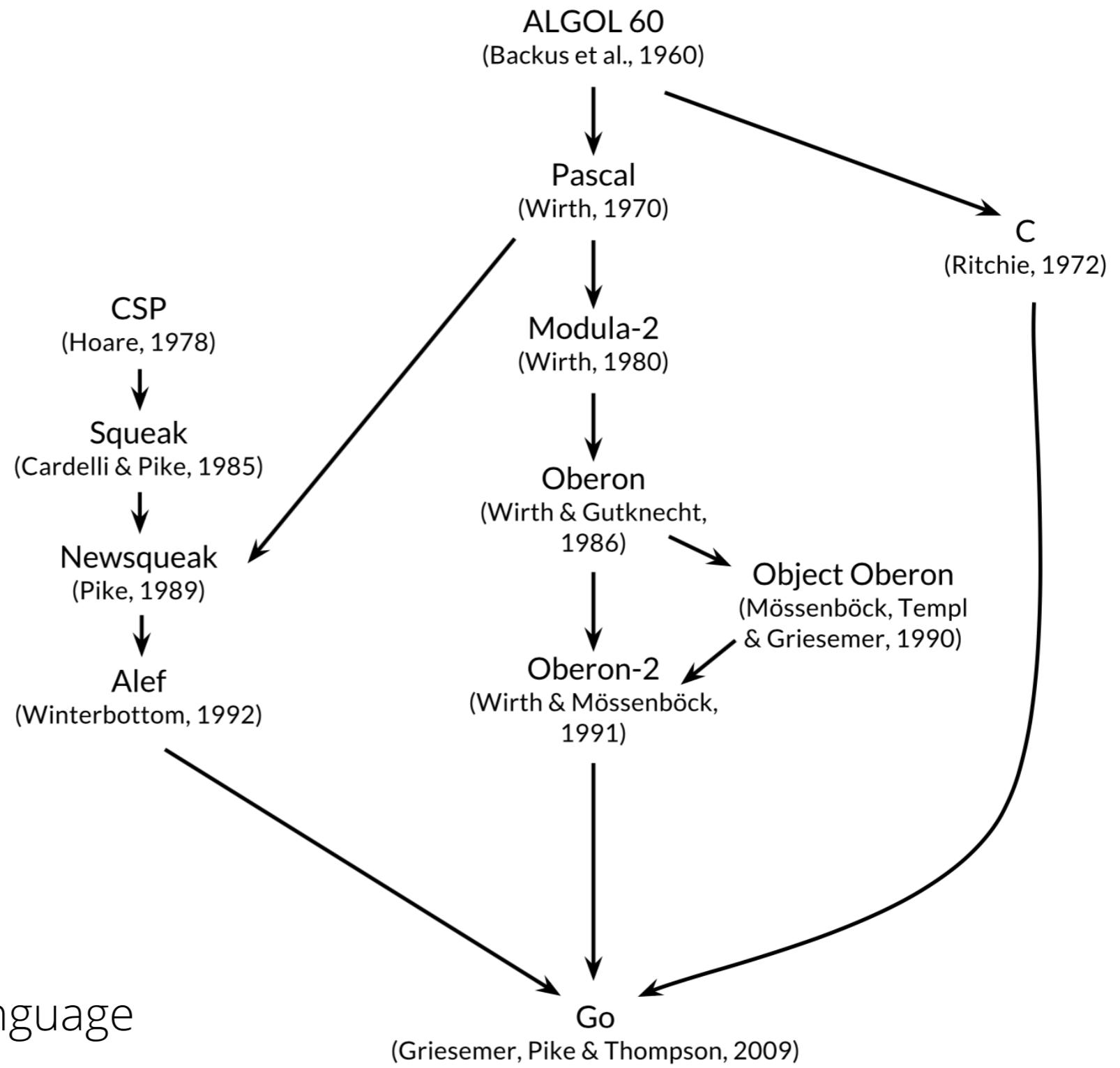
**Figura 1 — Tipologia de linguagens de programação.**



**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

**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             | Hydrogen 1.008          |
| 2             | Be Beryllium 9.012      |
| 3             | Lithium 6.941           |
| 4             | Magnesium 24.305        |
| 11            | Sodium 22.990           |
| 12            | Mg Magnesium 24.305     |
| 19            | K Potassium 39.098      |
| 20            | Ca Calcium 40.078       |
| 21            | Sc Scandium 44.956      |
| 22            | Ti Titanium 47.88       |
| 23            | V Vanadium 50.942       |
| 24            | Cr Chromium 51.996      |
| 25            | Mn Manganese 54.938     |
| 26            | Fe Iron 55.933          |
| 27            | Co Cobalt 58.933        |
| 28            | Ni Nickel 58.693        |
| 29            | Cu Copper 63.546        |
| 30            | Zn Zinc 65.39           |
| 31            | Ga Gallium 69.732       |
| 32            | Ge Germanium 72.61      |
| 33            | As Arsenic 74.922       |
| 34            | Se Selenium 78.972      |
| 35            | Br Bromine 79.904       |
| 36            | Kr Krypton 84.80        |
| 37            | Rb Rubidium 84.468      |
| 38            | Sr Strontium 87.62      |
| 39            | Y Yttrium 88.906        |
| 40            | Zr Zirconium 91.224     |
| 41            | Nb Niobium 92.906       |
| 42            | Mo Molybdenum 95.95     |
| 43            | Tc Technetium 98.907    |
| 44            | Ru Ruthenium 101.07     |
| 45            | Rh Rhodium 102.906      |
| 46            | Pd Palladium 106.42     |
| 47            | Ag Silver 107.868       |
| 48            | Cd Cadmium 112.411      |
| 49            | In Indium 114.818       |
| 50            | Sn Tin 118.71           |
| 51            | Sb Antimony 121.760     |
| 52            | Te Tellurium 127.6      |
| 53            | I Iodine 126.904        |
| 54            | Xe Xenon 131.29         |
| 55            | Cs Cesium 132.905       |
| 56            | Ba Barium 137.327       |
| 57            | Hf Hafnium 178.49       |
| 58            | Ta Tantalum 180.948     |
| 59            | W Tungsten 183.85       |
| 60            | Re Rhenium 186.207      |
| 61            | Os Osmium 190.23        |
| 62            | Ir Iridium 192.22       |
| 63            | Pt Platinum 195.08      |
| 64            | Au Gold 196.967         |
| 65            | Hg Mercury 200.59       |
| 66            | Tl Thallium 204.383     |
| 67            | Pb Lead 207.2           |
| 68            | Bi Bismuth 208.980      |
| 69            | Po Polonium [208.982]   |
| 70            | At Astatine 209.987     |
| 71            | Rn Radon 222.018        |
| 87            | Fr Francium 223.020     |
| 88            | Ra Radium 226.025       |
| 89            | Rf Rutherfordium [261]  |
| 90            | Db Dubnium [262]        |
| 91            | Sg Seaborgium [266]     |
| 92            | Bh Bohrium [264]        |
| 93            | Hs Hassium [269]        |
| 94            | Mt Meitnerium [268]     |
| 95            | Ds Darmstadtium [269]   |
| 96            | Rg Roentgenium [272]    |
| 97            | Cn Copernicium [277]    |
| 98            | Uut Ununtrium unknown   |
| 99            | F1 Flerovium [289]      |
| 100           | Uup Ununpentium unknown |
| 101           | Lv Livermorium [298]    |
| 102           | Uus Ununseptium unknown |
| 103           | Uuo Ununoctium unknown  |

**Lanthanide Series**

|    |                         |
|----|-------------------------|
| 57 | La Lanthanum 138.906    |
| 58 | Ce Cerium 140.115       |
| 59 | Pr Praseodymium 140.908 |
| 60 | Nd Neodymium 144.24     |
| 61 | Pm Promethium 144.913   |
| 62 | Sm Samarium 150.36      |
| 63 | Eu Europium 151.966     |
| 64 | Gd Gadolinium 157.25    |
| 65 | Tb Terbium 158.925      |
| 66 | Dy Dysprosium 162.50    |
| 67 | Ho Holmium 164.930      |
| 68 | Er Erbium 167.26        |
| 69 | Tm Thulium 168.934      |
| 70 | Yb Ytterbium 173.04     |
| 71 | Lu Lutetium 174.967     |

**Actinide Series**

|     |                         |
|-----|-------------------------|
| 89  | Ac Actinium 227.028     |
| 90  | Th Thorium 232.038      |
| 91  | Pa Protactinium 231.036 |
| 92  | U Uranium 238.029       |
| 93  | Np Neptunium 237.048    |
| 94  | Pu Plutonium 244.064    |
| 95  | Am Americium 243.061    |
| 96  | Cm Curium 247.070       |
| 97  | Bk Berkelium 247.070    |
| 98  | Cf Californium 251.080  |
| 99  | Es Einsteinium [254]    |
| 100 | Fm Fermium 257.095      |
| 101 | Md Mendelevium 258.1    |
| 102 | No Nobelium 259.101     |
| 103 | Lr Lawrencium [262]     |

**Classification Legend:**

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

© 2014 Todd Helmenstine sciencenotes.org

# A CLASSIFICATION BASED ON HARD FACTS

**"Noble" gases!?**

**Periodic Table of the Elements**

| 1<br>IA<br>1A                                                                                                                                | H<br>Hydrogen<br>1.008   | 2<br>IIA<br>2A                   | Be<br>Beryllium<br>9.012       | 3<br>IIIB<br>3B                     | Li<br>Lithium<br>6.941          | 4<br>IVB<br>4B                    | Mg<br>Magnesium<br>24.305        | 5<br>VB<br>5B                    | V<br>Vanadium<br>50.942          | 6<br>VIB<br>6B                   | Cr<br>Chromium<br>51.996           | 7<br>VIIB<br>7B                  | Mn<br>Manganese<br>54.938       | 8                                 | Fe<br>Iron<br>55.933             | 9                                | Co<br>Cobalt<br>58.933 | 10                           | Ni<br>Nickel<br>58.693  | 11<br>IB<br>1B            | Cu<br>Copper<br>63.546 | 12<br>IIB<br>2B         | Zn<br>Zinc<br>65.39       | 13<br>IIIA<br>3A            | B<br>Boron<br>10.811       | 14<br>IVA<br>4A            | C<br>Carbon<br>12.011       | 15<br>VA<br>5A           | N<br>Nitrogen<br>14.007       | 16<br>VIA<br>6A             | O<br>Oxygen<br>15.999        | 17<br>VIIA<br>7A             | F<br>Fluorine<br>18.998     | 18<br>VIIIA<br>8A           | He<br>Helium<br>4.003     |                        |
|----------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|----------------------------------|--------------------------------|-------------------------------------|---------------------------------|-----------------------------------|----------------------------------|----------------------------------|----------------------------------|----------------------------------|------------------------------------|----------------------------------|---------------------------------|-----------------------------------|----------------------------------|----------------------------------|------------------------|------------------------------|-------------------------|---------------------------|------------------------|-------------------------|---------------------------|-----------------------------|----------------------------|----------------------------|-----------------------------|--------------------------|-------------------------------|-----------------------------|------------------------------|------------------------------|-----------------------------|-----------------------------|---------------------------|------------------------|
| 1<br>-252.762                                                                                                                                | 3<br>1342                | 4<br>2471                        | 11<br>882.940                  | 12<br>1090                          | 19<br>759                       | 20<br>1484                        | 21<br>2836                       | 22<br>3287                       | 23<br>3407                       | 24<br>2671                       | 25<br>2061                         | 26<br>2861                       | 27<br>2927                      | 28<br>2913                        | 29<br>2562                       | 30<br>907                        | 31<br>2204             | 32<br>2833                   | 33<br>616.5P            | 34<br>685                 | 35<br>1587             | 36<br>-153.34           |                           |                             |                            |                            |                             |                          |                               |                             |                              |                              |                             |                             |                           |                        |
| Normal boiling points are in °C.<br>SP = Triple Point<br>Pressure is listed if not 1 atm.<br>Allotrope is listed if more than one allotrope. |                          |                                  |                                |                                     |                                 |                                   |                                  |                                  |                                  |                                  |                                    |                                  |                                 |                                   |                                  |                                  |                        |                              |                         |                           |                        |                         |                           |                             |                            |                            |                             |                          |                               |                             |                              |                              |                             |                             |                           |                        |
| 38<br>688                                                                                                                                    | 39<br>1382               | 40<br>3345                       | 41<br>4409                     | 42<br>4744                          | 43<br>4639                      | 44<br>4265                        | 45<br>4150                       | 46<br>3695                       | 47<br>2963                       | 48<br>2162                       | 49<br>767                          | 50<br>2072                       | 51<br>1587                      | 52<br>988                         | 53<br>1587                       | 54<br>-108.09                    | 55<br>671              | 56<br>1897                   | 57-71                   | 72<br>4603                | 73<br>5458             | 74<br>5555              | 75<br>5596                | 76<br>5012                  | 77<br>4428                 | 78<br>3825                 | 79<br>2856                  | 80<br>356.62             | 81<br>1473                    | 82<br>1749                  | 83<br>1564                   | 84<br>962                    | 85<br>1564                  | 86<br>-61.7                 |                           |                        |
| Fr<br>Francium<br>223.020                                                                                                                    | Rb<br>Rubidium<br>84.468 | Sr<br>Strontium<br>87.62         | Y<br>Yttrium<br>88.906         | Zr<br>Zirconium<br>91.224           | Nb<br>Niobium<br>92.906         | Mo<br>Molybdenum<br>95.95         | Tc<br>Technetium<br>98.907       | Ru<br>Ruthenium<br>101.07        | Rh<br>Rhodium<br>102.906         | Pd<br>Palladium<br>106.42        | Ag<br>Silver<br>107.868            | Cd<br>Cadmium<br>112.411         | In<br>Indium<br>114.818         | Sn<br>Tin<br>118.71               | Sb<br>Antimony<br>121.760        | Te<br>Tellurium<br>127.6         | I<br>Iodine<br>126.904 | Xe<br>Xenon<br>131.29        | Cs<br>Cesium<br>132.905 | Ba<br>Barium<br>137.327   |                        | Hf<br>Hafnium<br>178.49 | Ta<br>Tantalum<br>180.948 | W<br>Tungsten<br>183.85     | Re<br>Rhenium<br>186.207   | Os<br>Osmium<br>190.23     | Ir<br>Iridium<br>192.22     | Pt<br>Platinum<br>195.08 | Au<br>Gold<br>196.967         | Hg<br>Mercury<br>200.59     | Tl<br>Thallium<br>204.383    | Pb<br>Lead<br>207.2          | Bi<br>Bismuth<br>208.980    | Po<br>Polonium<br>[208.982] | At<br>Astatine<br>209.987 | Rn<br>Radon<br>222.018 |
| 87<br>677                                                                                                                                    | 88<br>1737               | 89-103                           | 104<br>unknown                 | 105<br>unknown                      | 106<br>unknown                  | 107<br>unknown                    | 108<br>unknown                   | 109<br>unknown                   | 110<br>unknown                   | 111<br>unknown                   | 112<br>unknown                     | 113<br>unknown                   | 114<br>unknown                  | 115<br>unknown                    | 116<br>unknown                   | 117<br>unknown                   | 118<br>unknown         | Rf<br>Rutherfordium<br>[261] | Db<br>Dubnium<br>[262]  | Sg<br>Seaborgium<br>[266] | Bh<br>Bohrium<br>[264] | Hs<br>Hassium<br>[269]  | Mt<br>Meitnerium<br>[268] | Ds<br>Darmstadtium<br>[269] | Rg<br>Roentgenium<br>[272] | Cn<br>Copernicium<br>[277] | Uut<br>Ununtrium<br>unknown | Fl<br>Flerovium<br>[289] | Uup<br>Ununpentium<br>unknown | Uup<br>Livermorium<br>[298] | Lv<br>Ununseptium<br>unknown | Uuo<br>Ununoctium<br>unknown | Lu<br>Ununoctium<br>unknown |                             |                           |                        |
| Lanthanide Series                                                                                                                            |                          | 57<br>La<br>Lanthanum<br>138.906 | 58<br>Ce<br>Cerium<br>140.115  | 59<br>Pr<br>Praseodymium<br>140.908 | 60<br>Nd<br>Neodymium<br>144.24 | 61<br>Pm<br>Promethium<br>144.913 | 62<br>Sm<br>Samarium<br>150.36   | 63<br>Eu<br>Europium<br>151.966  | 64<br>Gd<br>Gadolinium<br>157.25 | 65<br>Tb<br>Terbium<br>158.925   | 66<br>Dy<br>Dysprosium<br>162.50   | 67<br>Ho<br>Holmium<br>164.930   | 68<br>Er<br>Erbium<br>167.26    | 69<br>Tm<br>Thulium<br>168.934    | 70<br>Yb<br>Ytterbium<br>173.04  | 71<br>Lu<br>Lutetium<br>174.967  |                        |                              |                         |                           |                        |                         |                           |                             |                            |                            |                             |                          |                               |                             |                              |                              |                             |                             |                           |                        |
| Actinide Series                                                                                                                              |                          | 89<br>Ac<br>Actinium<br>227.028  | 90<br>Th<br>Thorium<br>232.038 | 91<br>Pa<br>Protactinium<br>231.036 | 92<br>U<br>Uranium<br>238.029   | 93<br>Np<br>Neptunium<br>237.048  | 94<br>Pu<br>Plutonium<br>244.064 | 95<br>Am<br>Americium<br>243.061 | 96<br>Cm<br>Curium<br>247.070    | 97<br>Bk<br>Berkelium<br>247.070 | 98<br>Cf<br>Californium<br>251.080 | 99<br>Es<br>Einsteinium<br>[254] | 100<br>Fm<br>Fermium<br>257.095 | 101<br>Md<br>Mendelevium<br>258.1 | 102<br>No<br>Nobelium<br>259.101 | 103<br>Lr<br>Lawrencium<br>[262] |                        |                              |                         |                           |                        |                         |                           |                             |                            |                            |                             |                          |                               |                             |                              |                              |                             |                             |                           |                        |
| Alkali Metal                                                                                                                                 |                          |                                  | Alkaline Earth                 |                                     |                                 | Transition Metal                  |                                  |                                  | Basic Metal                      |                                  |                                    | Semimetal                        |                                 |                                   | Nonmetal                         |                                  |                        | Halogen                      |                         |                           | Noble Gas              |                         |                           | Lanthanide                  |                            |                            | Actinide                    |                          |                               |                             |                              |                              |                             |                             |                           |                        |

© 2014 Todd Helmenstine  
sciencenotes.org

“Ontology is overrated.”

*Clay Shirky*

<https://tgo.li/onto-over>



# A BETTER APPROACH

---

Fundamental Features of Programming Languages

# HANDS-ON PROGRAMMING LANGUAGE THEORY

---

The screenshot shows a web browser window with the following details:

- Title Bar:** Teaching Programming Langua x
- Address Bar:** cs.brown.edu/~sk/Publications/Papers/Published/sk-teach-pl-post-linnaean/
- Content Area:**
  - Section Header:** Teaching Programming Languages in a Post-Linnaean Age
  - Author:** Shriram Krishnamurthi
  - Conference:** 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:**

**Left Sidebar:** A vertical bar on the left side of the page has the text "Shriram Krishnamurthi" repeated vertically in white on a brown background, with a blue gradient at the bottom.

**Page Footer:** These papers may differ in formatting from the versions that appear in print. They are made available only to support the rapid dissemination of results; the printed versions, not these, should be considered

## 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 x

← → C ⌂ cs.brown.edu/courses/cs173/2012/book/

Version Second Edition

◀ prev up next ▶

**Programming Languages: Application and Interpretation**

1 Introduction  
2 Everything (We Will Say) About Parsing  
3 A First Look at Interpretation  
4 A First Taste of Desugaring  
5 Adding Functions to the Language  
6 From Substitution to Environments  
7 Functions Anywhere  
8 Mutation: Structures and Variables  
9 Recursion and Cycles: Procedures and Data  
10 Objects  
11 Memory Management  
12 Representation Decisions  
13 Desugaring as a Language Feature  
14 Control Operations  
15 Checking Program Invariants Statically: Types  
16 Checking Program Invariants Dynamically: Contracts  
17 Alternate Application Semantics

by Shriram Krishnamurthi

# Programming Languages: Application and Interpretation

## 1 Introduction

- 1.1 Our Philosophy
- 1.2 The Structure of This Book
- 1.3 The Language of This Book

## 2 Everything (We Will Say) About Parsing

- 2.1 A Lightweight, Built-In First Half of a Parser
- 2.2 A Convenient Shortcut
- 2.3 Types for Parsing
- 2.4 Completing the Parser
- 2.5 Coda

## 3 A First Look at Interpretation

- 3.1 Representing Arithmetic
- 3.2 Writing an Interpreter
- 3.3 Did You Notice?
- 3.4 Growing the Language

## 4 A First Taste of Desugaring

- 4.1 Extension: Binary Subtraction
- 4.2 Extension: Unary Negation

## 5 Adding Functions to the Language

- 5.1 Defining Data Representations
- 5.2 Growing the Interpreter
- 5.3 Substitution
- 5.4 The Interpreter, Resumed
- 5.5 Oh Wait, There's More!

## 6 From Substitution to Environments

- 6.1 Introducing the Environment

On this page:

**ThoughtWorks®**

# FEATURES

---

Understand core features, not just syntax and libraries

# SAMPLE FEATURES × LANGUAGES

---

|                       | Common<br>Lisp |
|-----------------------|----------------|
| First-class functions | ✓              |
| First-class types     | ✓              |
| Iterators             | *              |
| Variable model        | reference      |
| Type checking         | dynamic        |
| Type expression       | structural     |

# SAMPLE FEATURES × LANGUAGES

---

| Common<br>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<br>Lisp | C       | Java                   |
|-----------------------|----------------|---------|------------------------|
| First-class functions | ✓              | *       | ✓                      |
| First-class types     | ✓              |         |                        |
| Iterators             | *              |         | ✓                      |
| Variable model        | reference      | value*  | value and<br>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

---

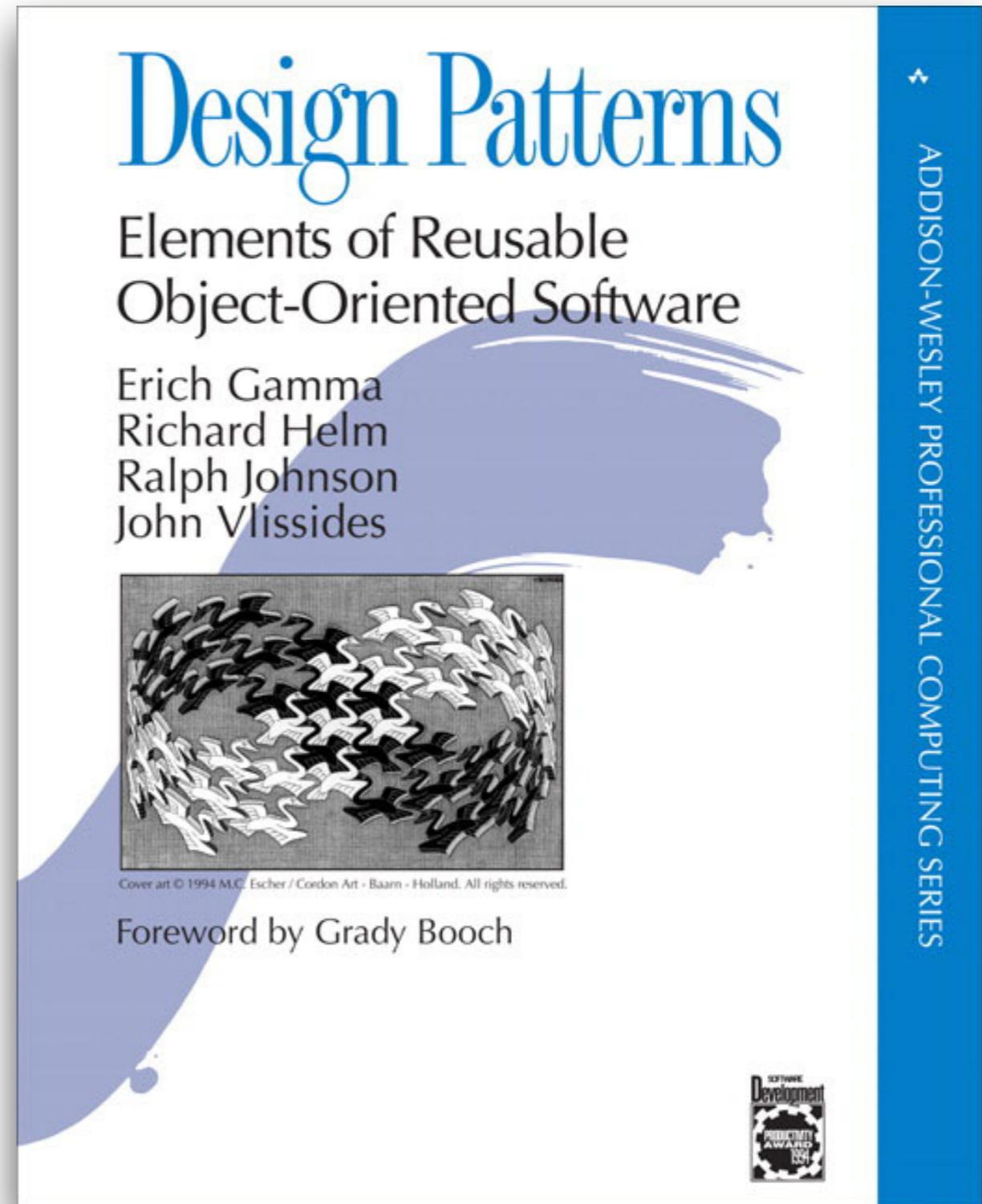
When languages fall short

# 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

---

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

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.<sup>2</sup>

# NOT EVERY PATTERN IS UNIVERSAL

---

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

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.<sup>2</sup>

# NOT EVERY PATTERN IS UNIVERSAL

---

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

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.<sup>2</sup>

# **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

## ITERABLES & ITERATORS

---

An excellent abstraction

# ITERATION IN 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

---

```
package main

import (
    "fmt"
    "os"
)

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

# ITERATION IN C AND GO

---

```
#include <stdio.h>

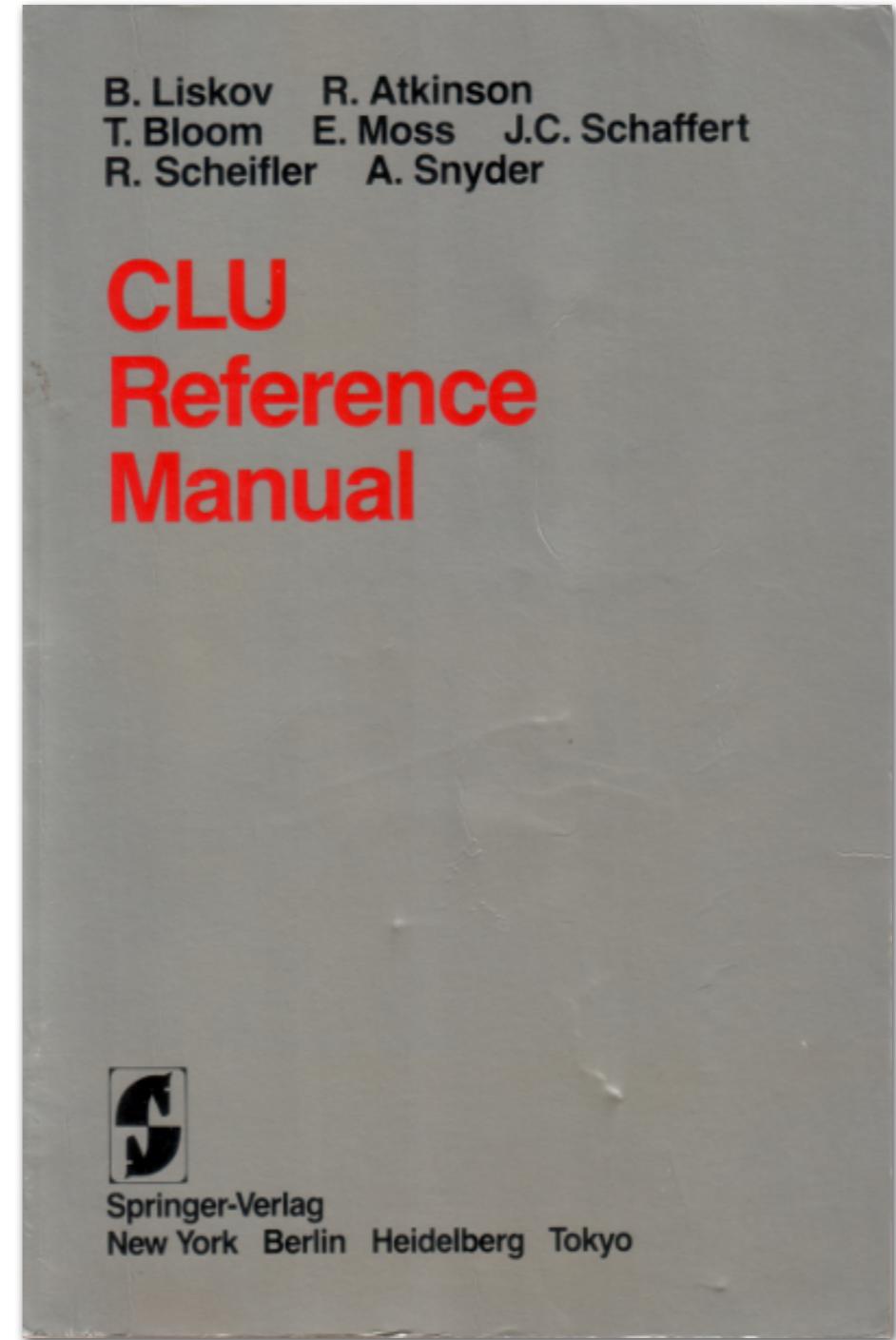
int main(int argc, char *argv[]) {
    for(int i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

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

# FOREACH IN BARBARA LISKOV'S CLU

---

© 2010 Kenneth C. Zirkel — CC-BY-SA



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>

2

Iterators

§1.2

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 *data type* with associated *operations* to create and manipulate those objects. The cluster also provides *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**

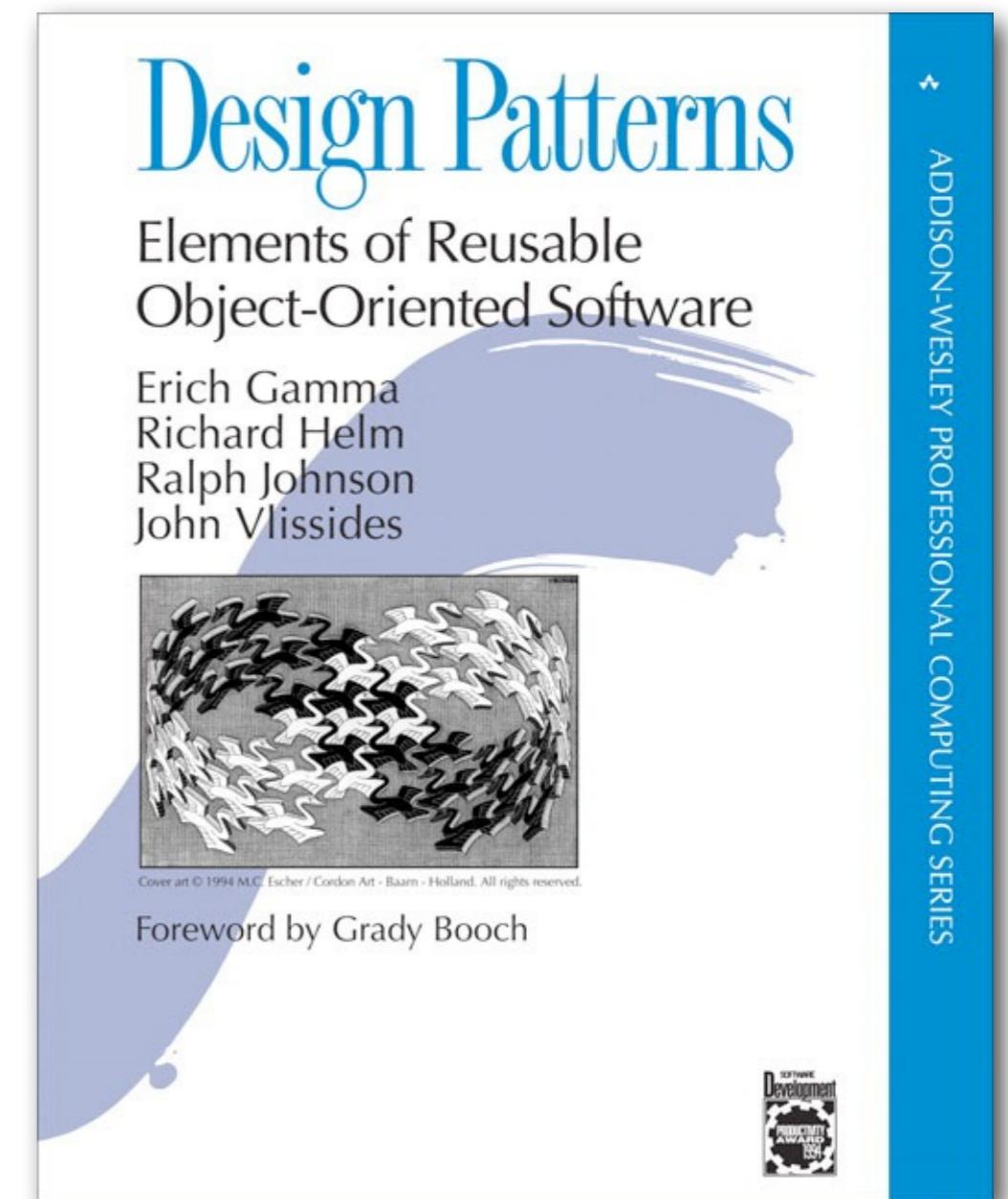
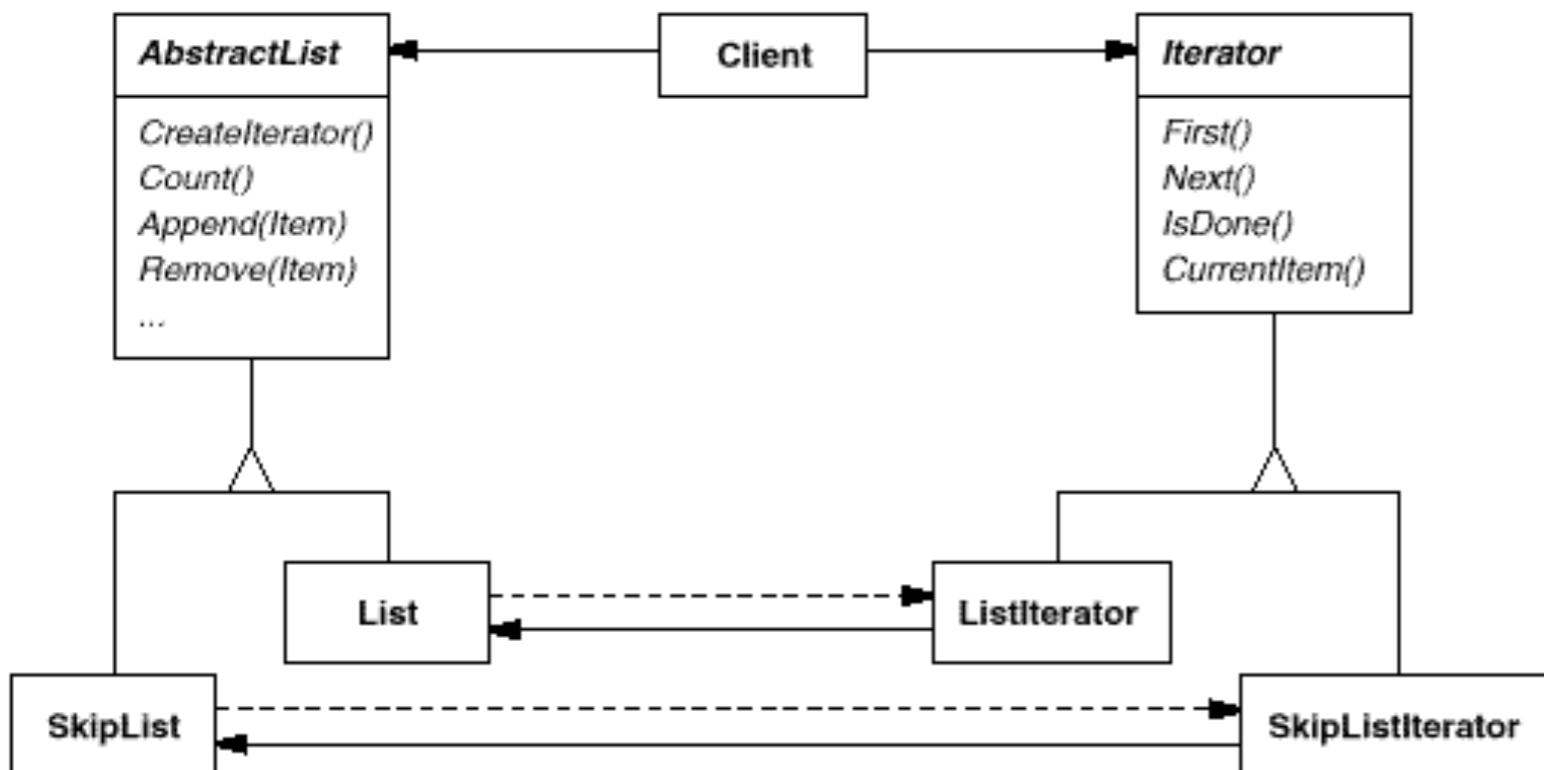
# ITERATOR PATTERN

---

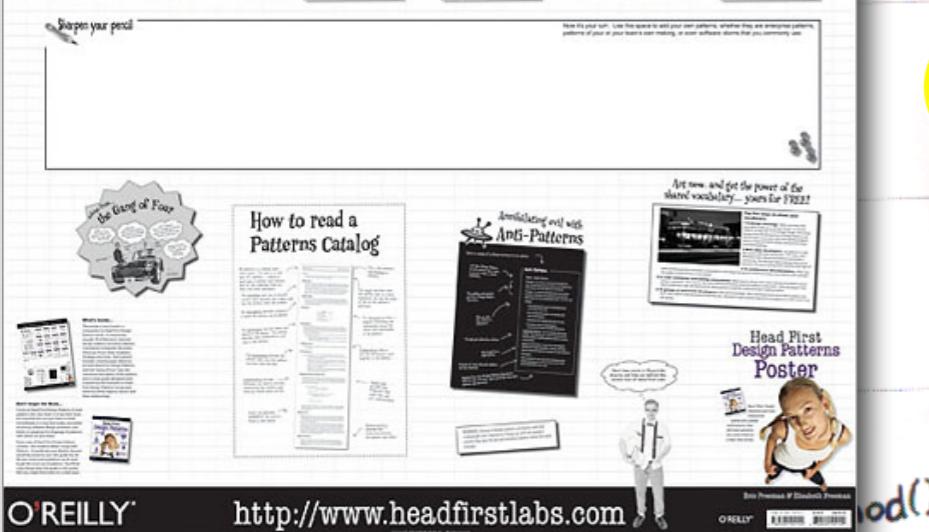
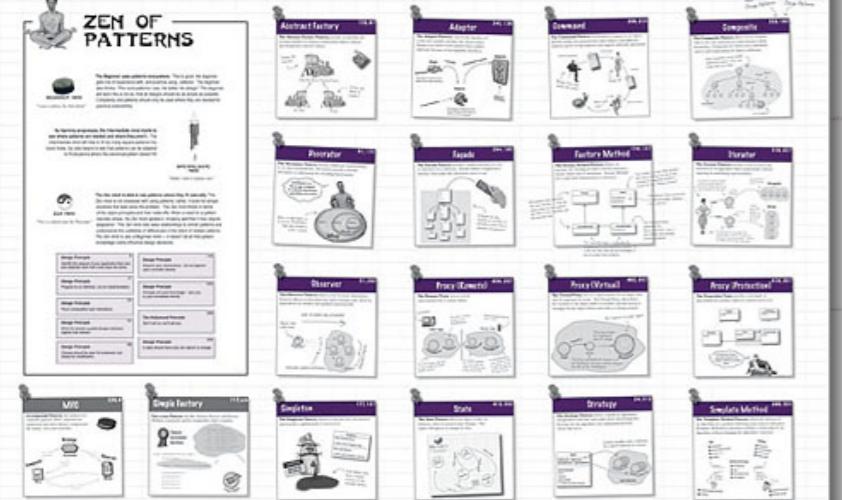
The classic recipe

# THE ITERATOR FROM THE GANG OF FOUR

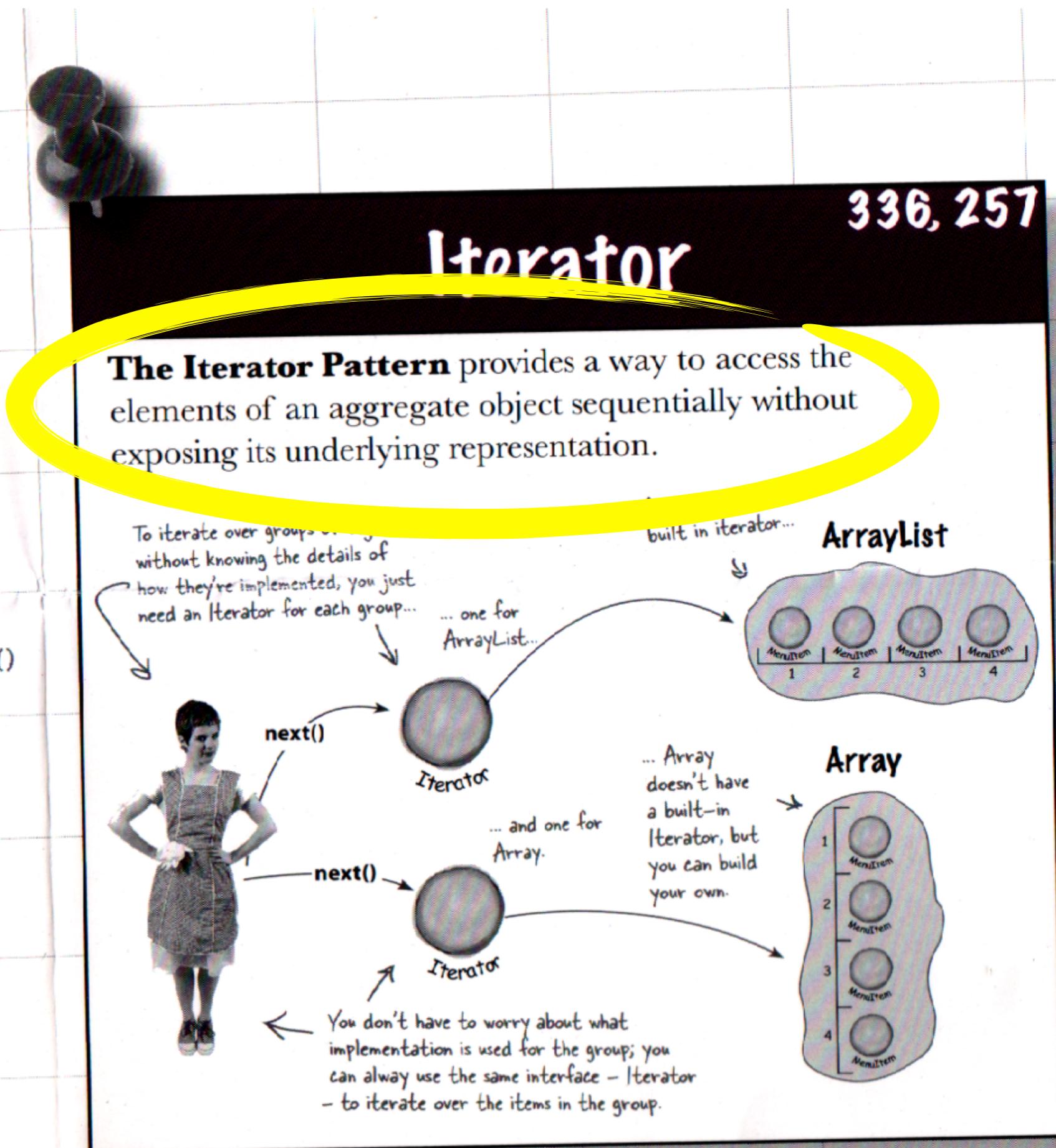
Design Patterns  
Gamma, Helm, Johnson & Vlissides  
©1994 Addison-Wesley



# Your Brain on Design Patterns



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



# CLASSIC ITERATOR IN GO

---

The classic recipe in Go

# USING A CLASSIC ITERATOR IN GO

---

**for/range** not supported



```
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)



```
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
}
```



# 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

---

```
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  
}
```



```
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

---

```
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

---

```
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
}
```



```
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

---

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

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

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

A *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.



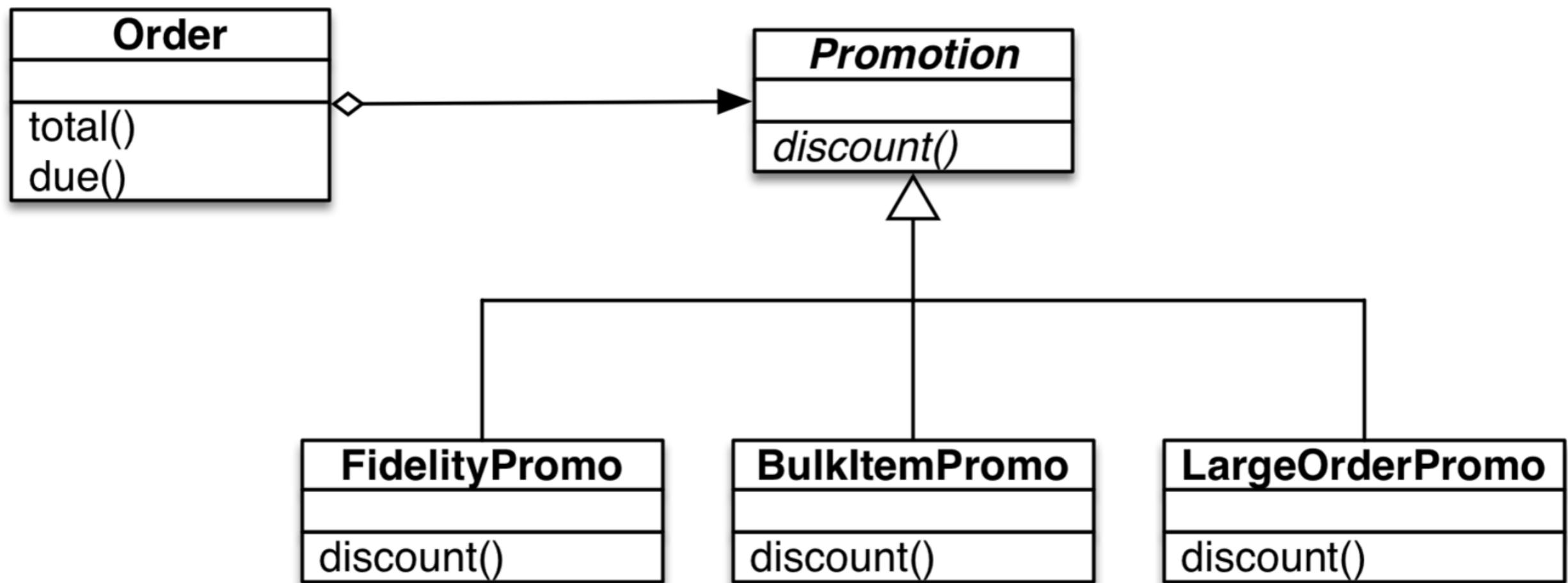
# STRATEGY IN GO

---

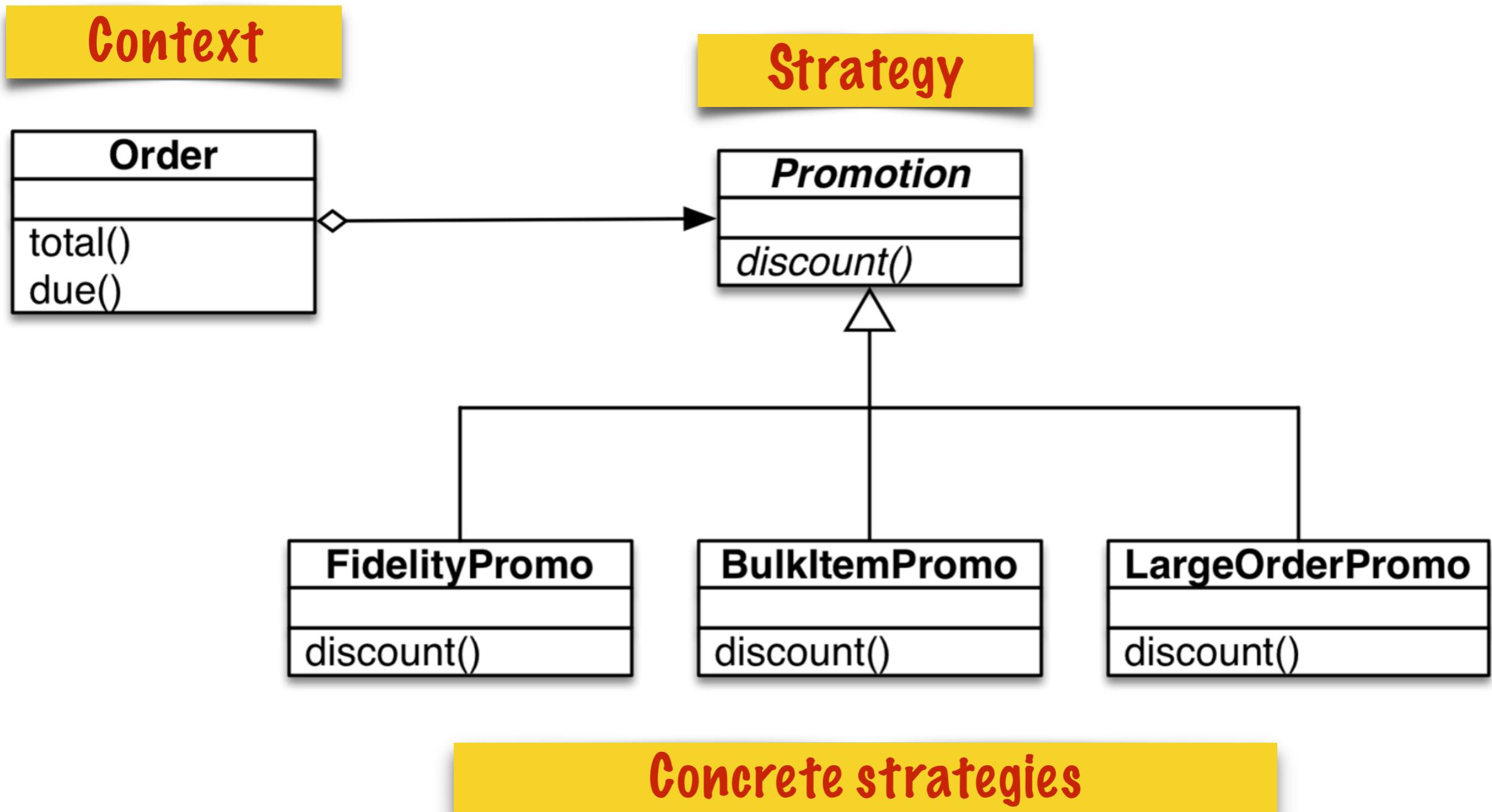
Leveraging core features of Go

# 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

```
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
```

# 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

```
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 }
```

```
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

```
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 }
```

# STRATEGY WITH FIRST CLASS FUNCTIONS

The diagram illustrates the evolution of a Go program from an interface-based design to one using first-class functions. It shows two tabs: 'interface | functions' (file A) and 'strategy.go (interface) | strategy.go (functions)' (file B).

**File A (interface.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
```

**File B (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
```

Annotations highlight specific changes:

- Line 3: The comment is updated to refer to a "promotion discount strategy".
- Line 7: The field name "Discounter" is replaced by "func(Order) int".
- Line 10: The function name "Total" is added.
- Line 18: The function name "Due" is added.
- Line 23: The call to "o.Promotion.Discount(o)" is replaced by "o.Promotion(o)".

Tool interface at the bottom:

- Blocks
- Fluid
- Unified
- Ignore whitespace
- Change 1 of 4
- Up/Down arrows

# TESTS WITH FUNCTIONS

The screenshot shows a code editor with two tabs open, A and B, side-by-side.

**Tab A: interface | functions**

```
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
```

**Tab B: strategy\_test.go (interface) | strategy\_test.go (functions)**

```
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
```

The code in both tabs is identical, demonstrating how to write tests for interfaces and functions. The code defines a `Customer` struct, a `LineItem` type, and three test functions: `TestOrderTotal`, `TestNoFidelityOrder`, and `TestFidelityOrder`. Each test function creates a `Customer` object, a `LineItem` slice, and an `Order` object using the `FidelityPromo` implementation. It then compares the expected result (`want`) with the actual result (`got`) using `t.Errorf`.

# CONCRETE STRATEGIES AS FUNCTIONS

```
interface | functions 2 | store.go (interface) | store.go (functions) 2 | strategy_test.go (interface) | strategy_test.go (functions) 2 |
A [navigation] strategy > interface > store.go B [navigation] strategy > functions > store.go
30 type FidelityPromo struct{} 28 }
31 29
32 // 5% discount for customers with at least 1000 30 // 5% discount for customers with at least 1000
fidelity points 31 func FidelityPromo(o Order) int {
33 func (p FidelityPromo) Discount(o Order) int { 32     if o.Customer.Fidelity >= 1000 {
34         if o.Customer.Fidelity >= 1000 { 33         return o.Total() / 20 // %5
35             return o.Total() / 20 // %5 34     }
36     } 35     return 0
37     return 0 36 }
38 } 37
39 38 // 10% discount for each line item with at least 20
40 type BulkItemPromo struct{} 39 units
41 40 func BulkItemPromo(o Order) int {
42 // 10% discount for each line item with at least 20 41     discount := 0
units 42     for _, item := range o.Cart {
43 func (p BulkItemPromo) Discount(o Order) int { 43         if item.Quantity >= 20 {
44         discount := 0 44             discount += item.Total() / 10
45         for _, item := range o.Cart { 45     }
46             if item.Quantity >= 20 { 46         return discount
47                 discount += item.Total() / 10 47     }
48             } 48
49         } 49 // 7% discount for orders with 10 or more items
50         return discount 50 func LargeOrderPromo(o Order) int {
51     } 51     if len(o.Cart) >= 10 {
52 52         return o.Total() * 7 / 100
53 type LargeOrderPromo struct{} 53     }
54 54     return 0
55 // 7% discount for orders with 10 or more items 55 }
56 func (p LargeOrderPromo) Discount(o Order) int { 56 }
```

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\*:

The screenshot shows a web browser window displaying the Go documentation for the `http` package. The URL in the address bar is `https://golang.org/pkg/net/http/`. The browser interface includes standard controls like back, forward, and search, along with a zoom level of 133%.

**func Handle**

```
func Handle(pattern string, handler Handler)
```

Handle registers the handler for the given pattern in the DefaultServeMux. The documentation for ServeMux explains how patterns are matched.

**func HandleFunc**

```
func HandleFunc(pattern string, handler func(ResponseWriter, *Request))
```

HandleFunc registers the handler function for the given pattern in the DefaultServeMux. The documentation for ServeMux explains how patterns are matched.

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



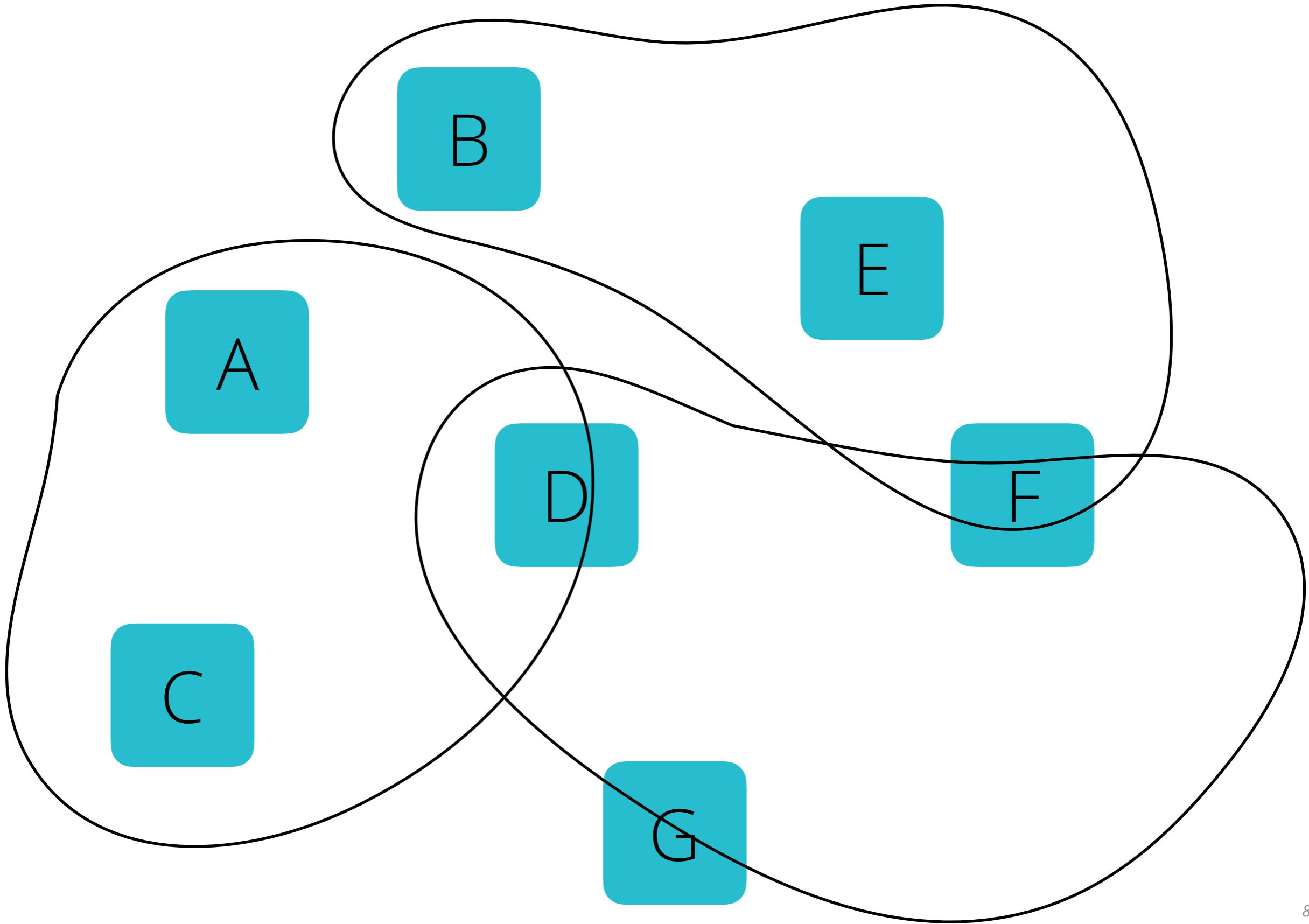
# WRAPPING UP

---

Why learn the fundamentals

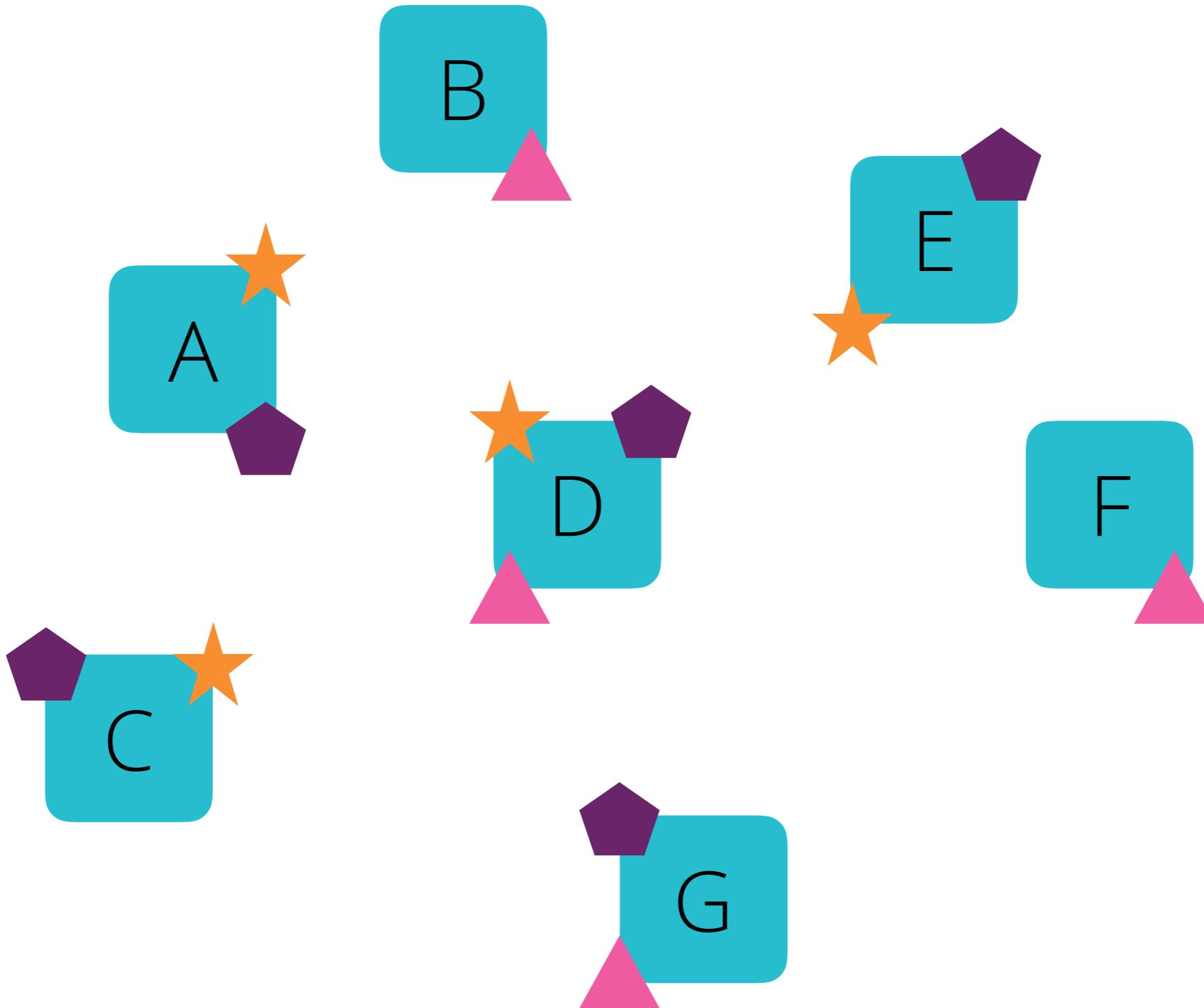
# INSTEAD OF PARADIGMS...

---



# FOCUS ON FUNDAMENTAL FEATURES

---

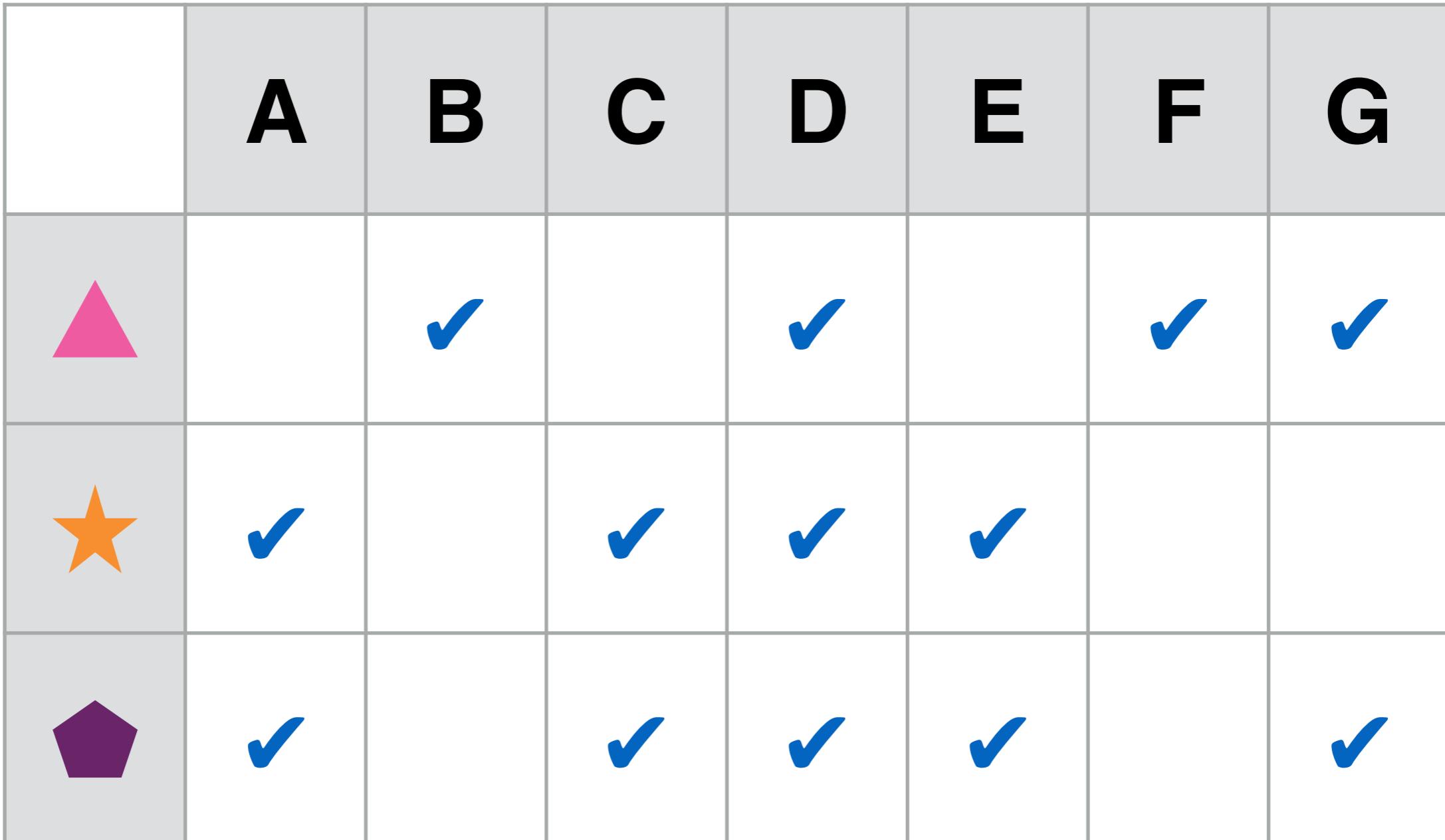


# FEATURES ARE THE BEST GUIDE...

---

...to decide whether a particular pattern or implementation makes sense.

|  | A | B | C | D | E | F | G |
|--|---|---|---|---|---|---|---|
|  |   |   |   |   |   |   |   |
|  |   |   |   |   |   |   |   |



|  | A | B | C | D | E | F | G |
|--|---|---|---|---|---|---|---|
|  |   |   |   |   |   |   |   |
|  |   |   |   |   |   |   |   |

# 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](http://speakerdeck.com/ramalho)*

**ThoughtWorks®**