ThoughtWorks®

Theory for practice

BEYOND PARADIGMS

Understand language features,
use the right patterns.

Luciano Ramalho
@ramalhoorg

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:

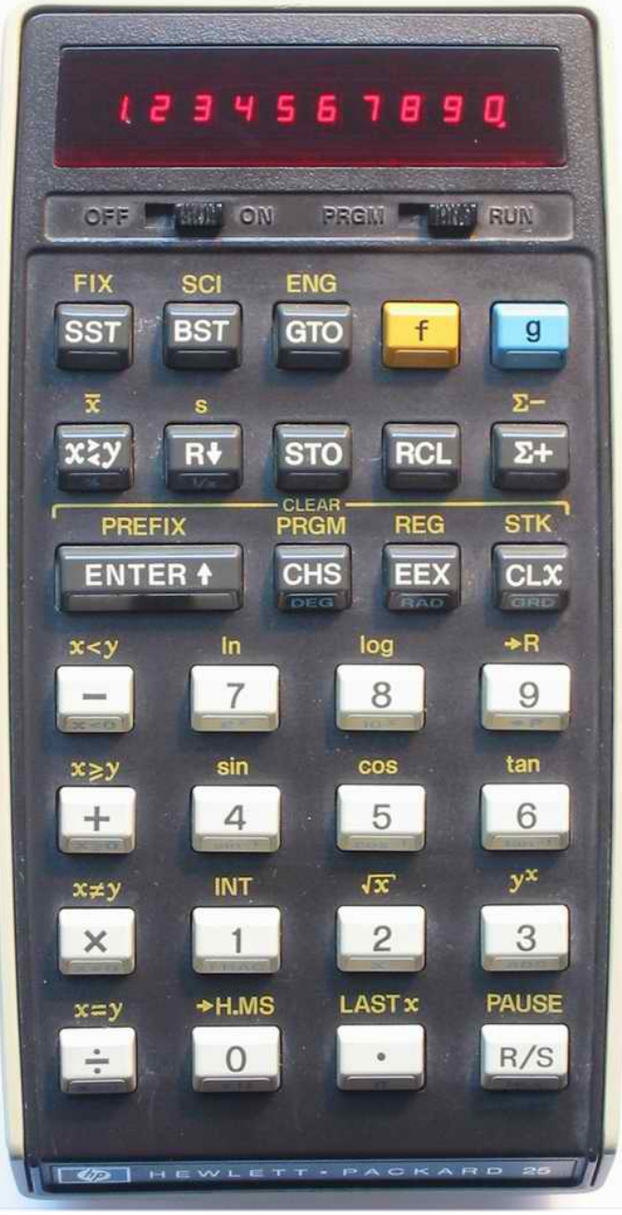
- Those that belong to the emperor
- Embalmed ones
- Those that are trained
- Suckling pigs
- Mermaids (or Sirens)
- Fabulous ones
- Stray dogs
- Those that are included in this classification
- Those that tremble as if they were mad
- Innumerable ones
- Those drawn with a very fine camel hair brush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

ThoughtWorks®

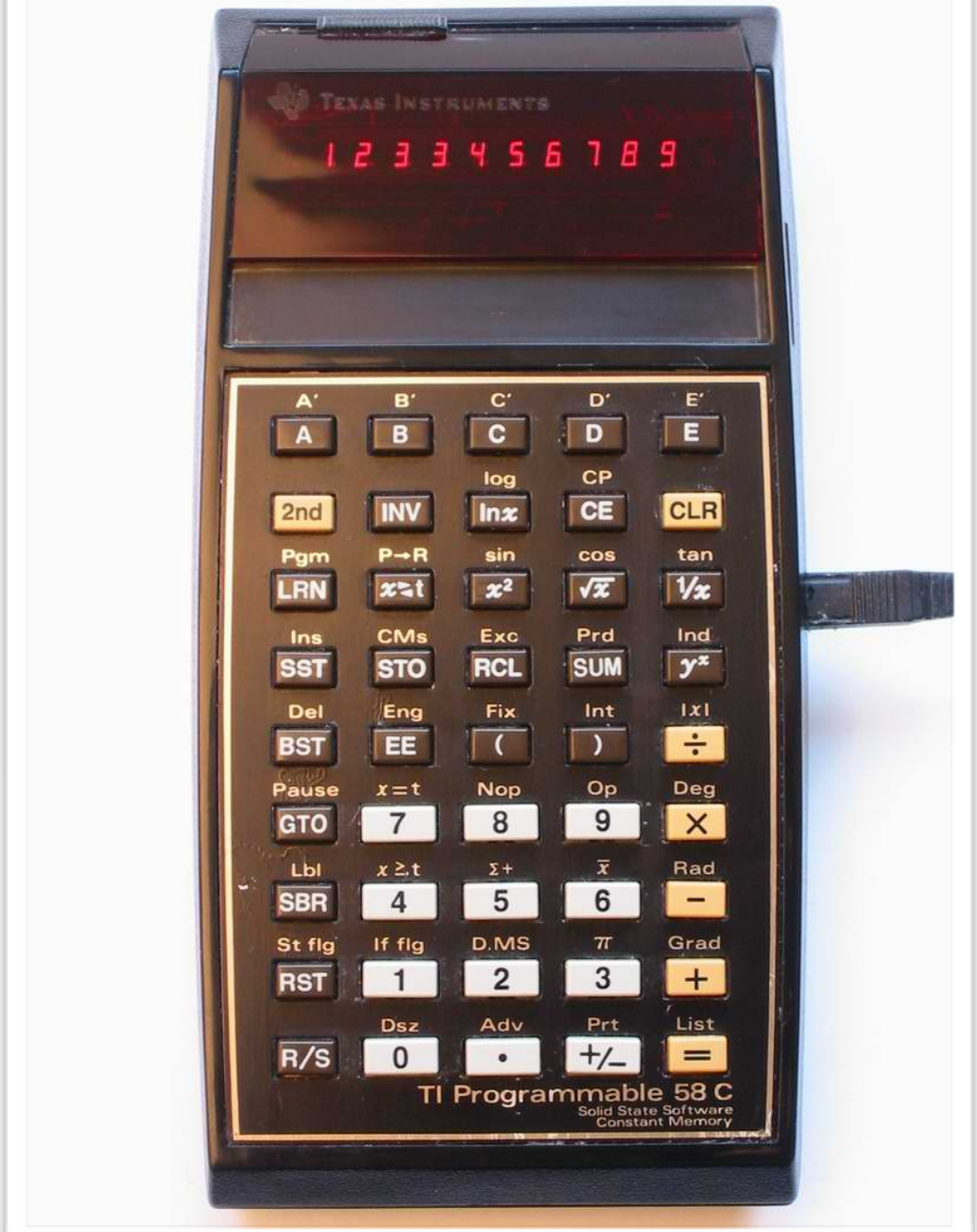
PARADIGMS

Programming language categories

CALCULATOR PROGRAMMING IS IMPERATIVE



HP-25

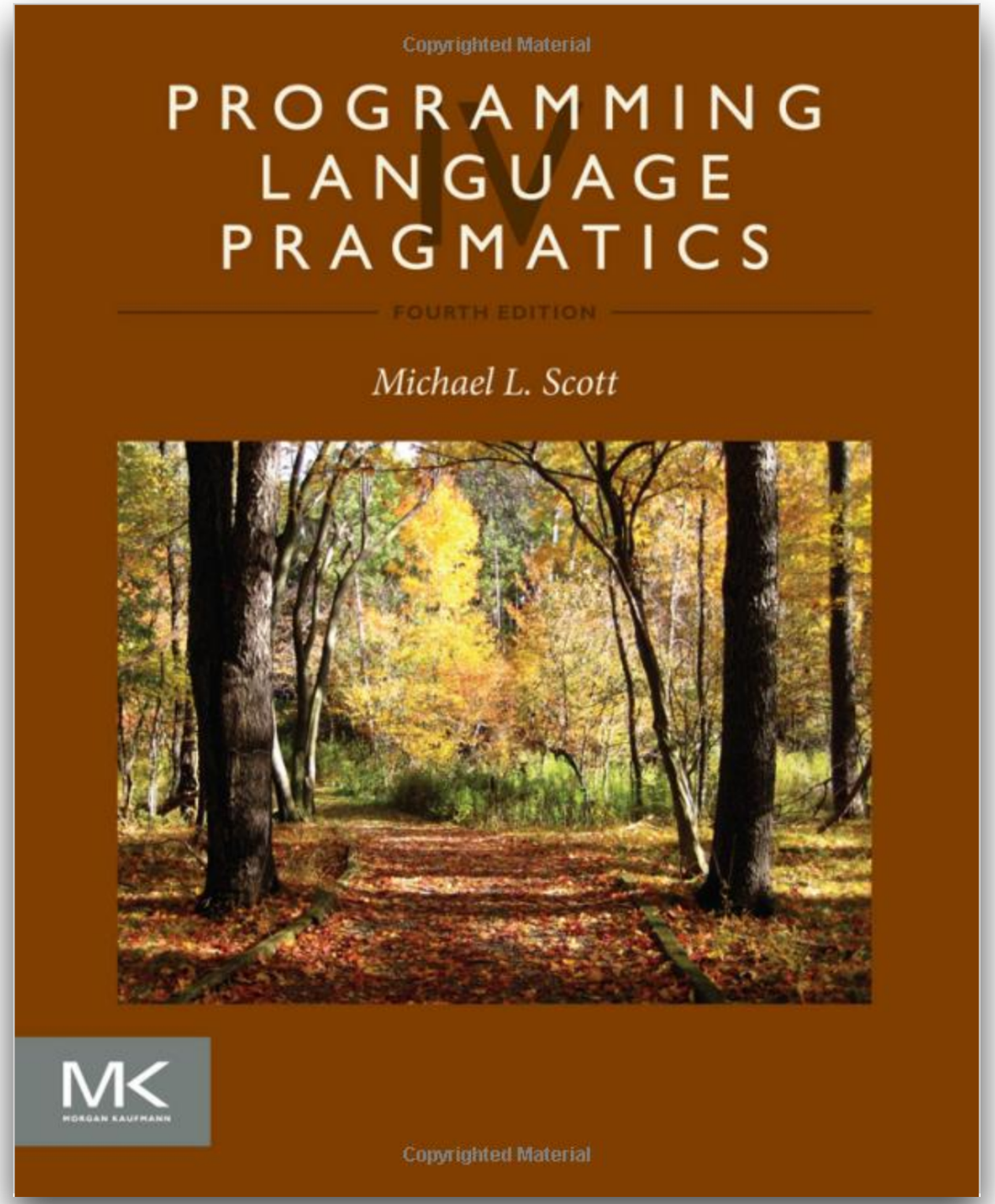


TI 58C

HP-25 CALCULATOR PROGRAMMING LANGUAGE

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

A SURVEY-STYLE PROGRAMMING LANGUAGES BOOK



Programming Language Pragmatics, 4th edition (2015)

Michael L. Scott

GCD ASM X86

Greatest common divisor in x86 Assembly

(Scott, 2015)

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

Figure 1.7 Naive x86 assembly language for the GCD program.

GCD IN C, OCAML AND PROLOG

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

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

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

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

GCD IN PYTHON

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

Imperative style

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

Functional style

GCD IN PYTHON

Bad fit for Python: no tail-call optimisation

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

Imperative style

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

Functional style

ONE CLASSIFICATION

1.2 The Programming Language Spectrum

Example 1.3

Classification of programming languages

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

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

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

Programming Language Pragmatics, 4th edition (2015)

Michael L. Scott

ONE CLASSIFICATION (ZOOM)

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

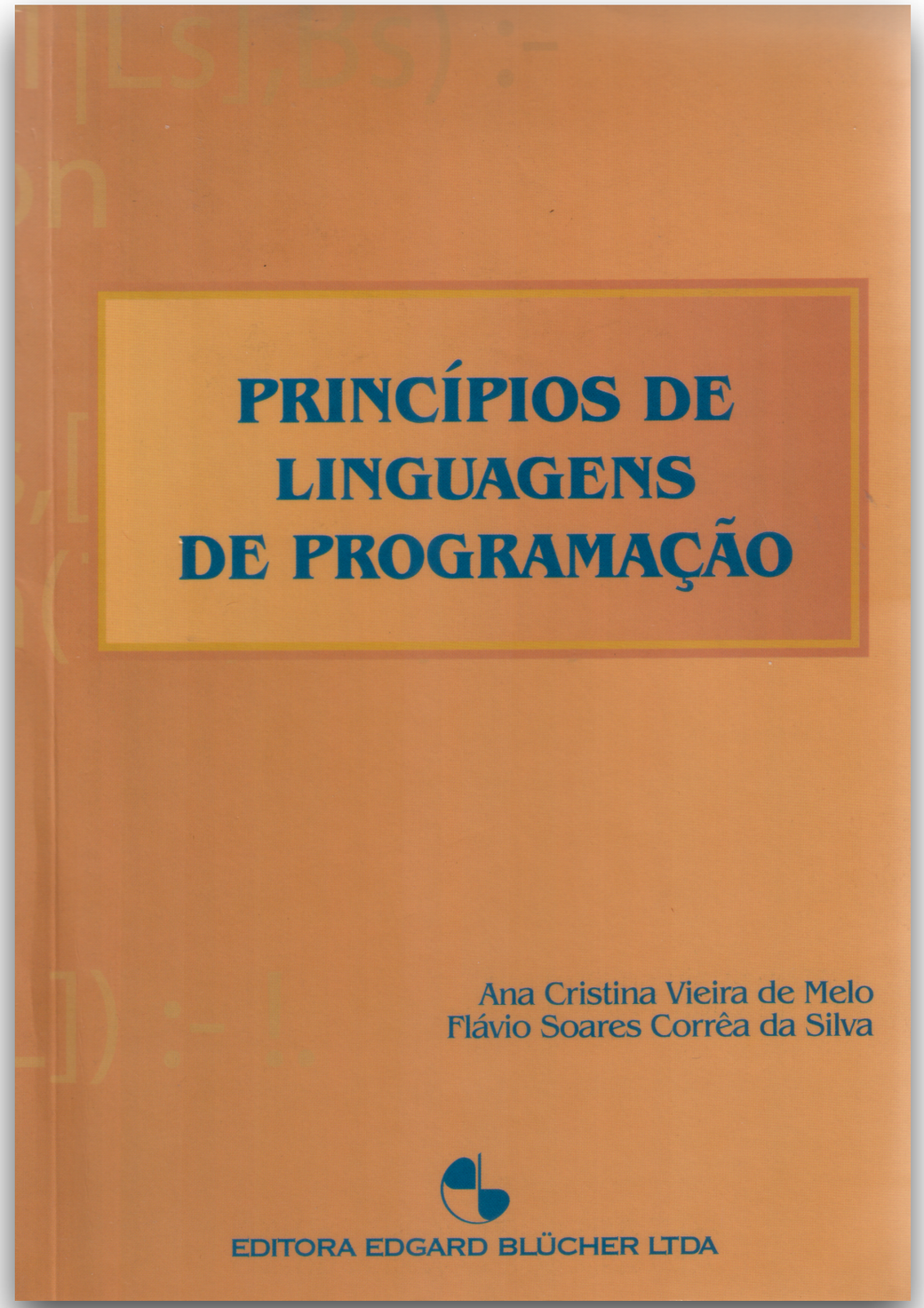
FIGURE 1.1 Classification of programming languages. Note that the categories are fuzzy and open to debate. In particular, it is possible for a functional language to be object-oriented, and many authors do not consider functional programming to be declarative.

ONE CLASSIFICATION (ZOOM)

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

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

ANOTHER BOOK



Princípios de Linguagens de Programação
(2003)

Ana Cristina Vieira de Melo

Flávio Soares Corrêa da Silva

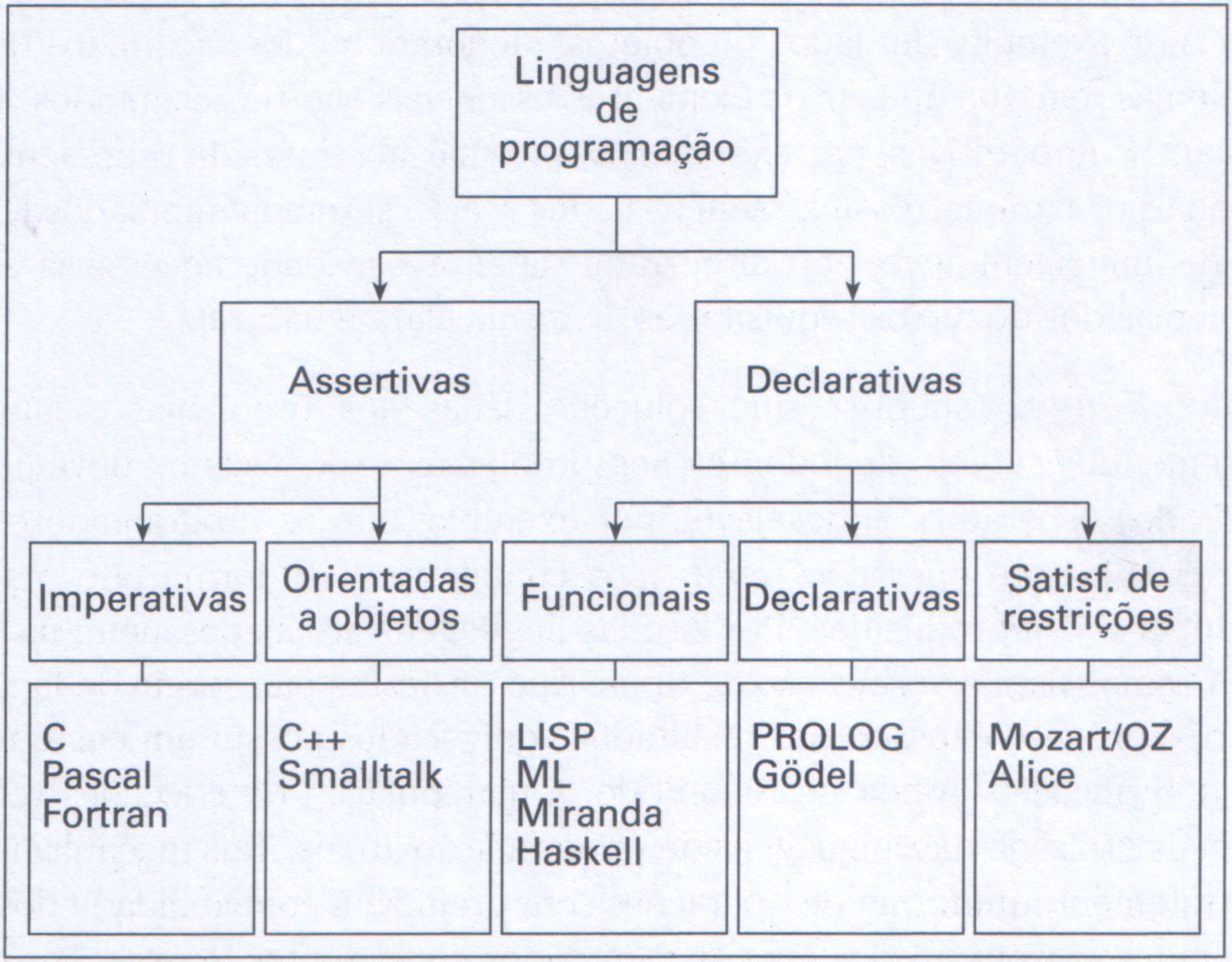


Figura 1 — Tipologia de linguagens de programação.

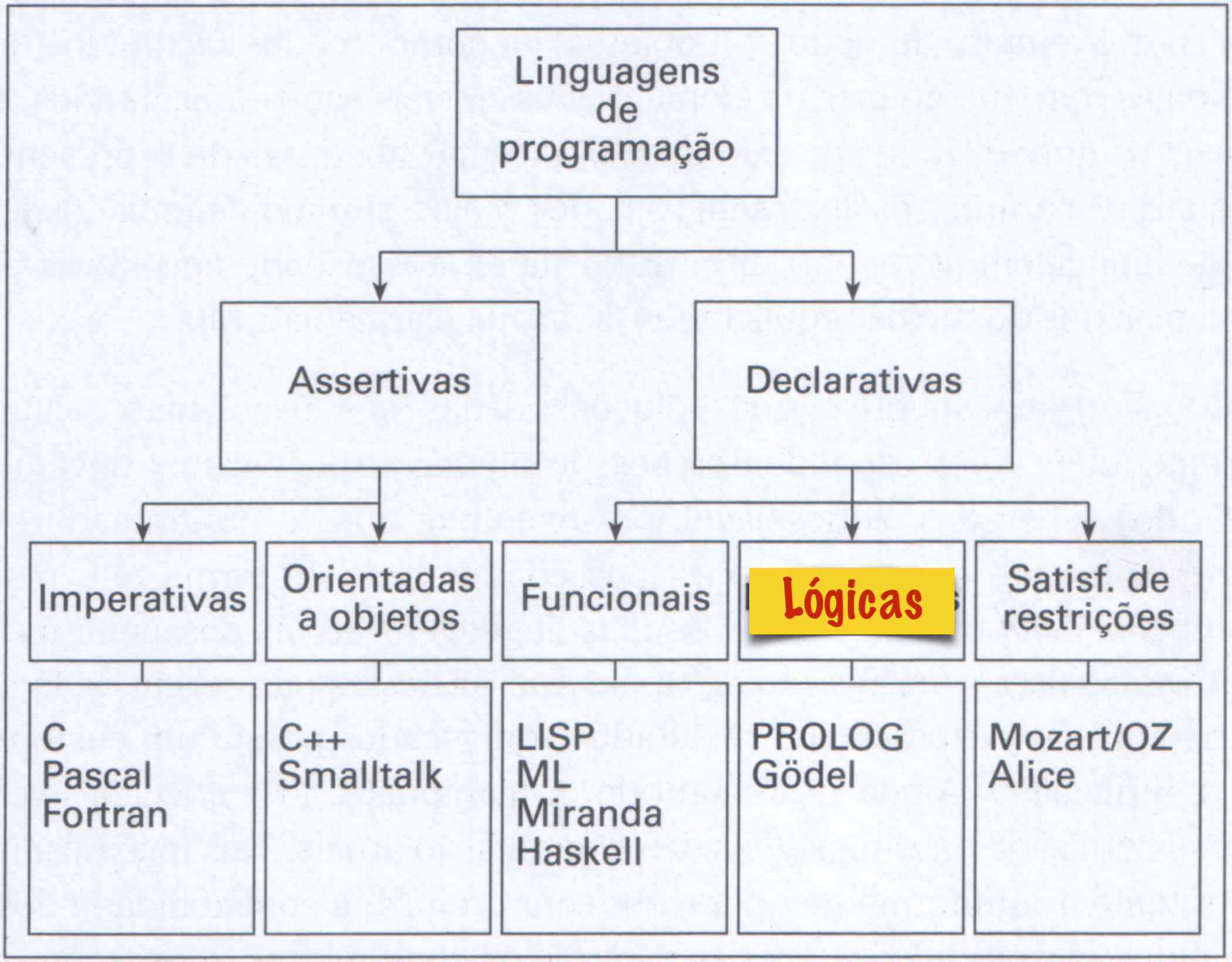


Figura 1 — Tipologia de linguagens de programação.

THE LANGUAGE LIST

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

The Language List

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

The Language List

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

Maintained by Bill Kinnersley

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

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

Start a Search

LANGUAGES ARE MISSING…

Search the Language List

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

Search for a particular language entry:

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

Home

Not Found

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

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

LANGUAGE CATEGORIES

Language Categories

Procedural Language

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

Imperative Language

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

Declarative Language

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

Applicative Language

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

Functional Language

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

LANGUAGE CATEGORIES (2)

Constraint Language

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

Object-Oriented Language

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

Concurrent Language

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

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

Concurrent languages differ in the way that processes are created:

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

and the ways in which processes interact:

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

LANGUAGE CATEGORIES (3)

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

Fourth Generation Language (4GL)

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

Query Language

An interface to a database.

Specification Language

A formalism for expressing a hardware or software design.

Assembly Language

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

Intermediate Language

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

Metalanguage

A language used for the formal description of another language.

LANGUAGE CATEGORIES (4)

Definitional Language

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

Single Assignment Language

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

Dataflow Language

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

Logic Language

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

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

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

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

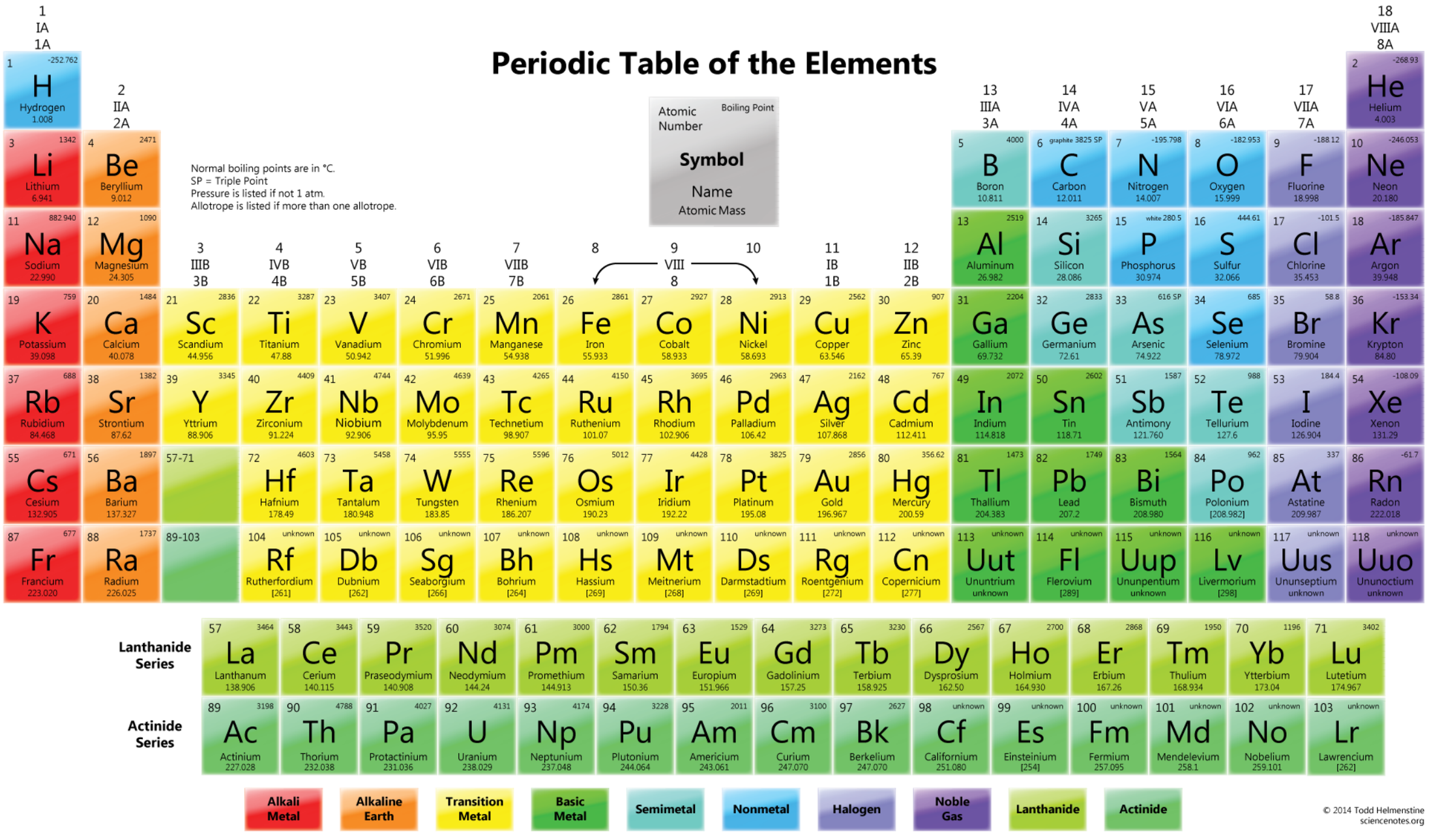
Constraint Language

ThoughtWorks®

CATEGORIES?

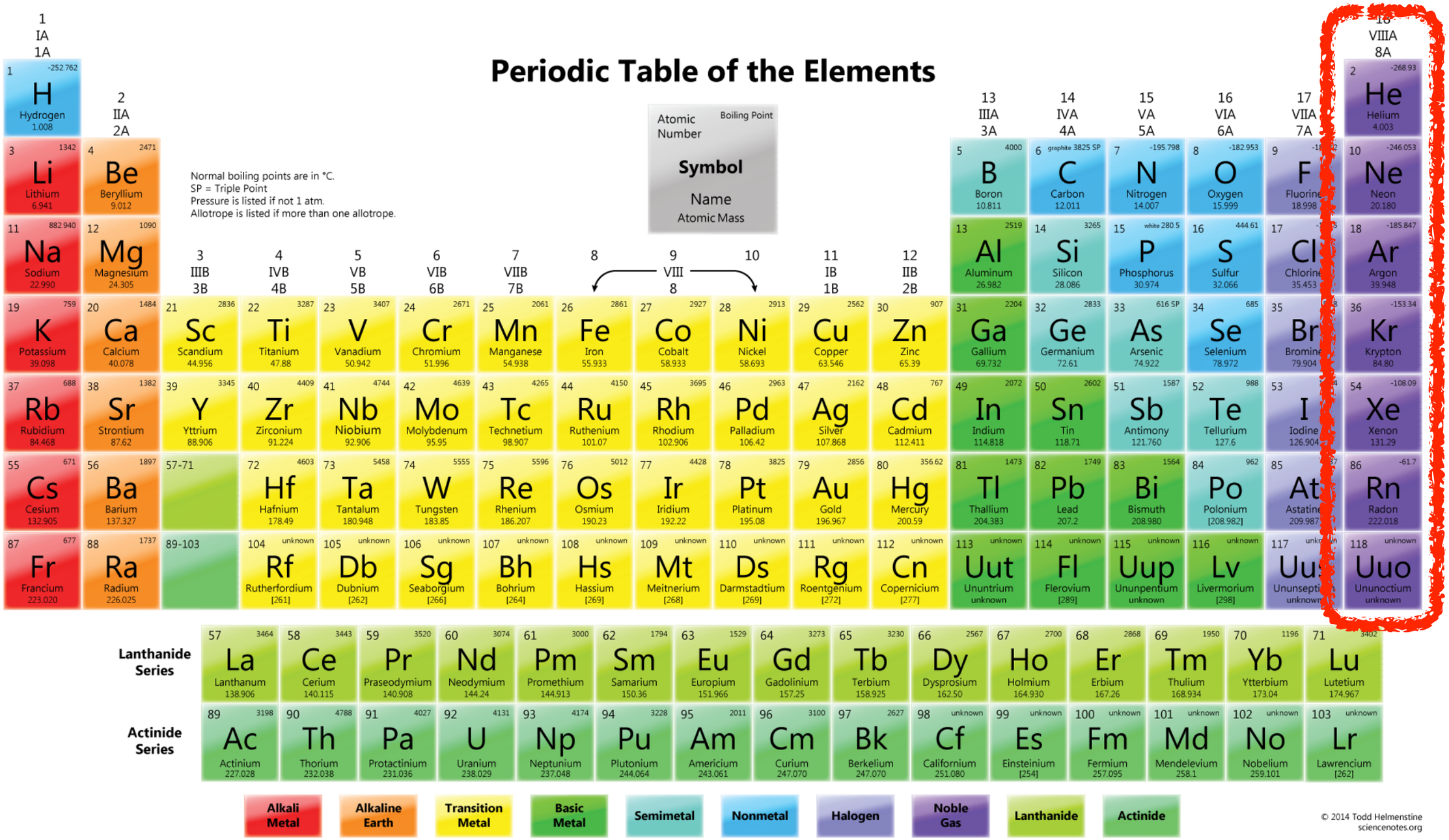
Ontologies are so 1900’s

A CLASSIFICATION BASED ON HARD FACTS



A CLASSIFICATION BASED ON HARD FACTS?

"Noble" gases!?



“Ontology is overrated.”

Clay Shirky

ThoughtWorks®

A BETTER APPROACH

Fundamental Features of Programming Languages

TEACHING PROGRAMMING LANGUAGE THEORY

Teaching Programming Langua ×

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

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

Abstract

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

Comment

The book discussed in this paper is available here.

Paper

PDF

A PAPER PRESENTING THE APPROACH

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

Abstract

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

A PAPER PRESENTING THE APPROACH

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

Abstract

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

A PAPER PRESENTING THE APPROACH

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

Abstract

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

A PAPER PRESENTING THE APPROACH

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

Abstract

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

THEORY IN PRACTICE WITH RACKET (A SCHEME DIALECT)

Programming Languages: Appl ×

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

On this page:

Version Second Edition

Programming Languages: Application and Interpretation

by Shriram Krishnamurthi

THEORY IN PRACTICE WITH PASCAL (1990)

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

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

goodreads Home My Books Browse Community Search books

Read

My rating:

Programming Languages: An Interpreter-Based Approach

by Samuel N. Kamin

5.0 · Rating details · 1 Rating · 0 Reviews

GET A COPY

Amazon BR | Online Stores | Book Links

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

Recommend It | Stats | Recent Status Updates

FRIEND REVIEWS

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

READER Q&A

Ask the Goodreads community a question about Programming Languages

Ask anything about the book

Be the first to ask a question about Programming Languages

LISTS WITH THIS BOOK

GENRES

Science > Computer Science 1 user

See top shelves...

BOOKS BY SAMUEL N. KAMIN

More...

ThoughtWorks®

FEATURES

Core features, not mere syntax

SAMPLE FEATURES ✖ LANGUAGES

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

SAMPLE FEATURES ✖ LANGUAGES

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

SAMPLE FEATURES ✖ LANGUAGES

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

SAMPLE FEATURES ✖ LANGUAGES

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

SAMPLE FEATURES ✖ LANGUAGES

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

SAMPLE FEATURES ✖ LANGUAGES

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

ThoughtWorks®

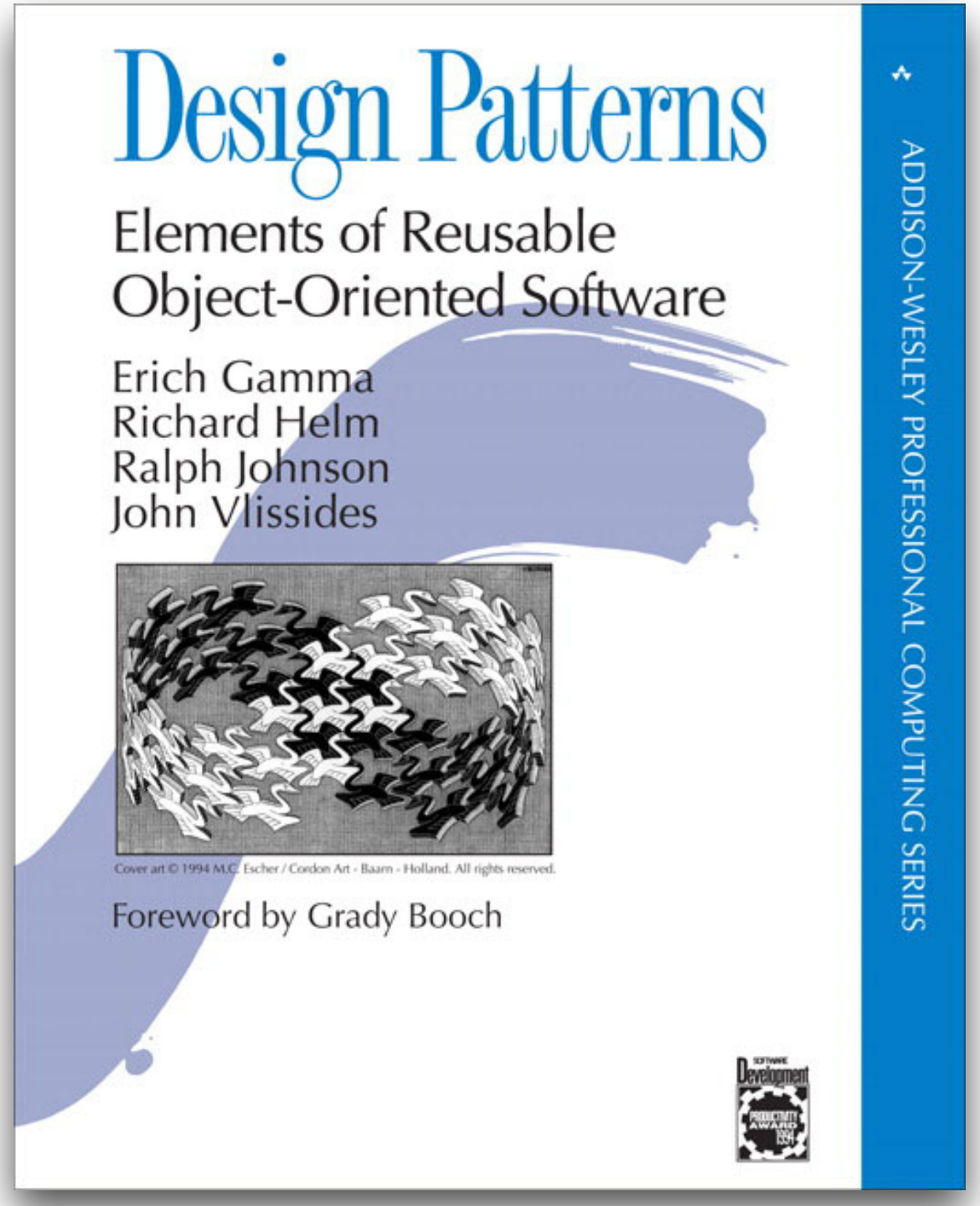
DESIGN PATTERNS

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)

(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`

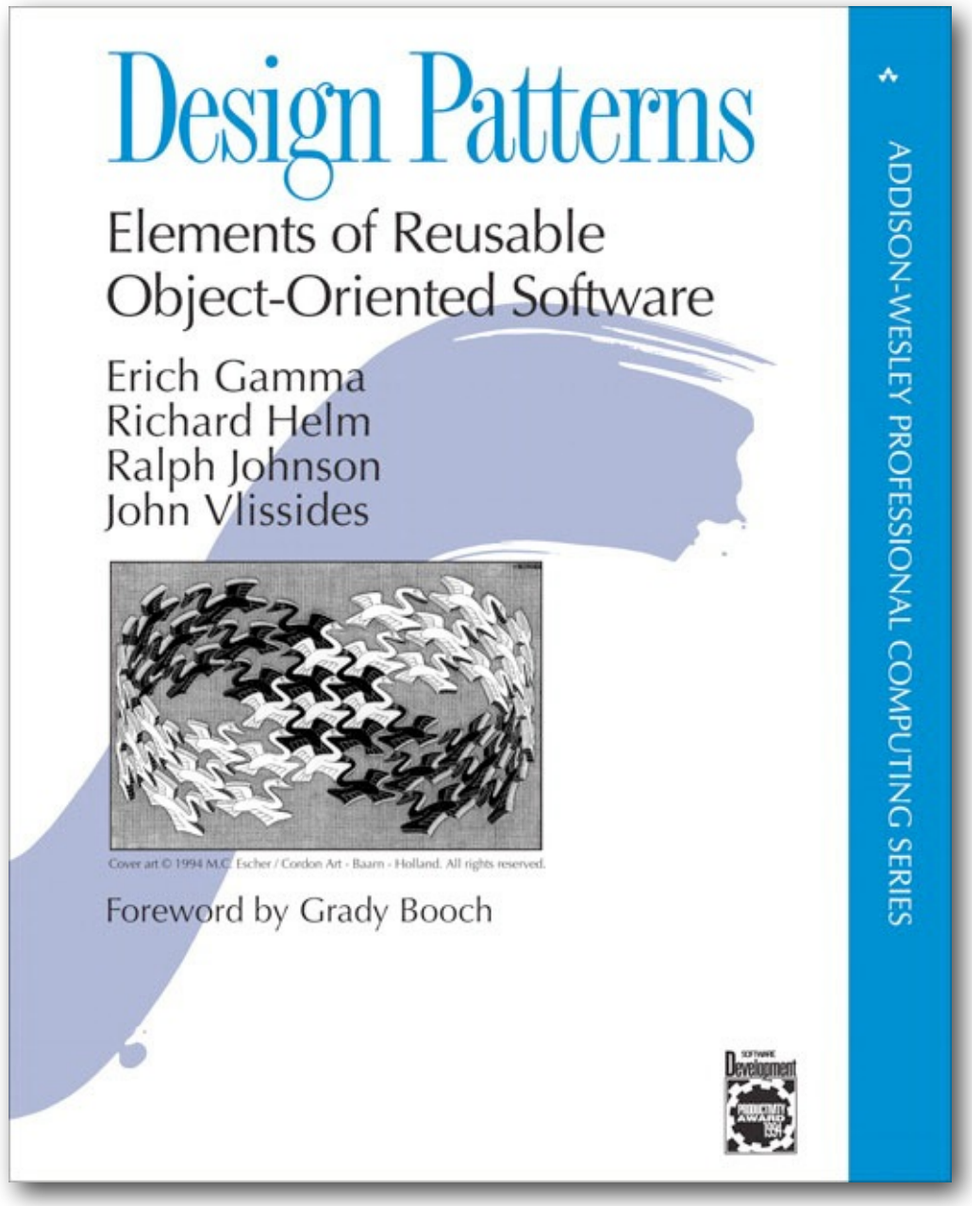
ThoughtWorks®

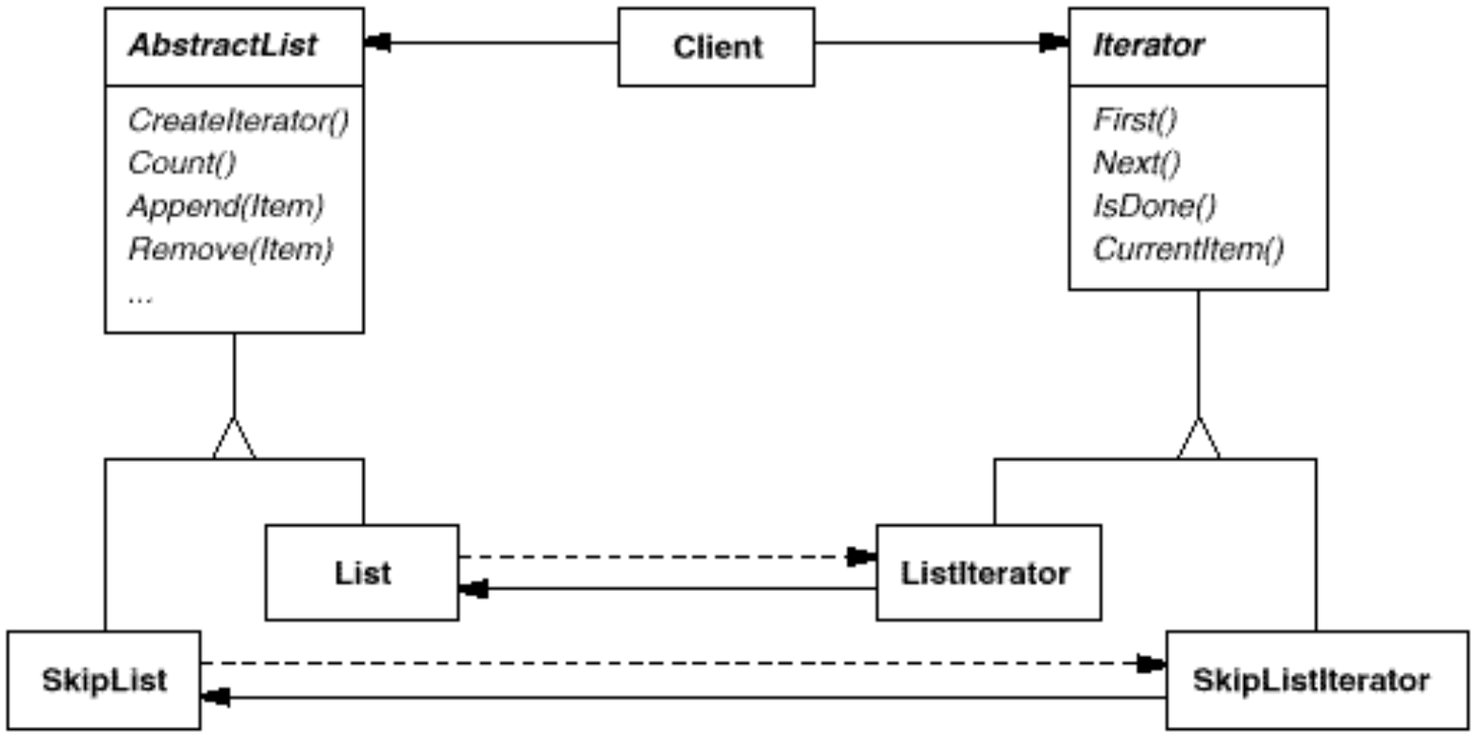
THE ITERATOR PATTERN

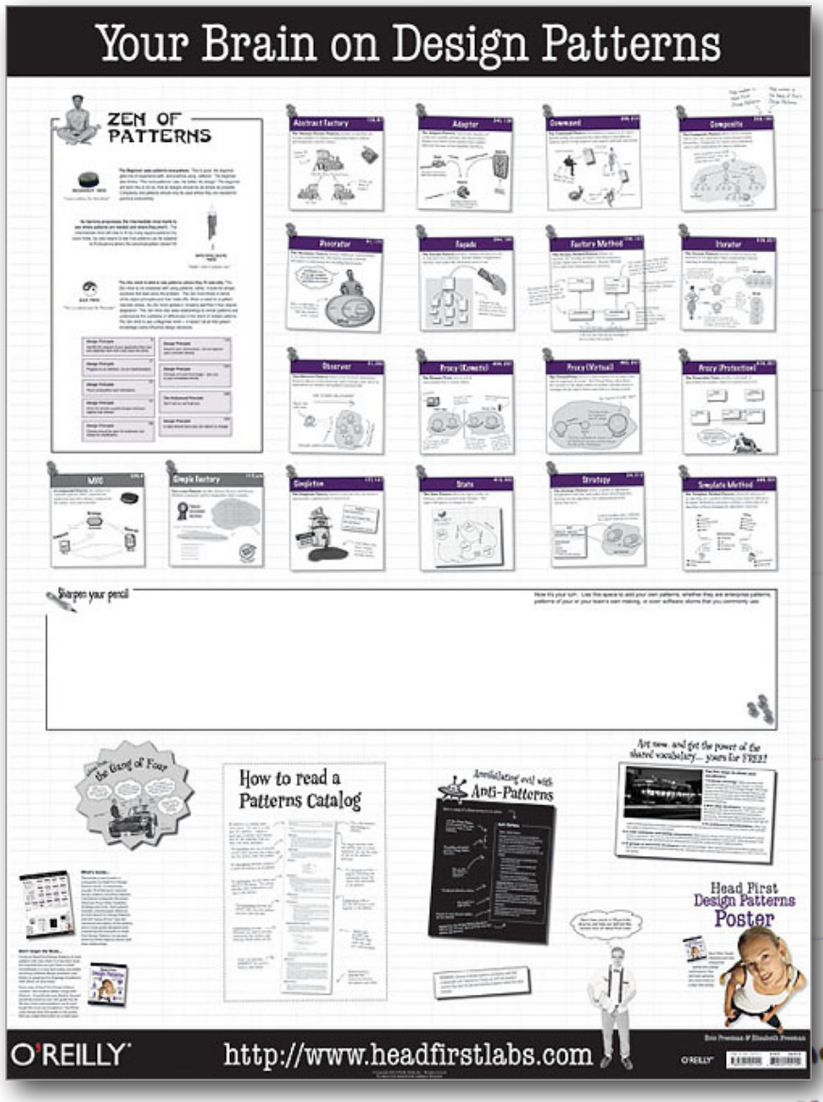
The classic recipe

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.

THE FOR LOOP MACHINERY

- In Python, the for loop, automatically:
  - Obtains an iterator from the iterable
  - Repeatedly invokes next() on the iterator, retrieving one item at a time
  - Assigns the item to the loop variable(s)

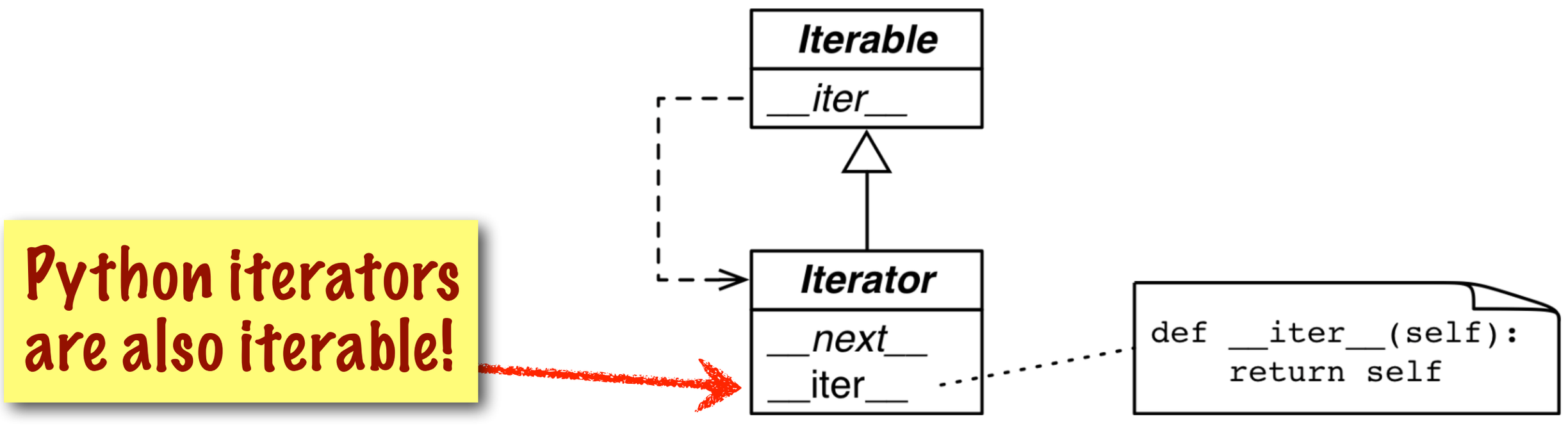
```
for item in an_iterable:
    process(item)
```

- Terminates when a call to next() raises StopIteration.

ITERABLE VERSUS ITERATOR

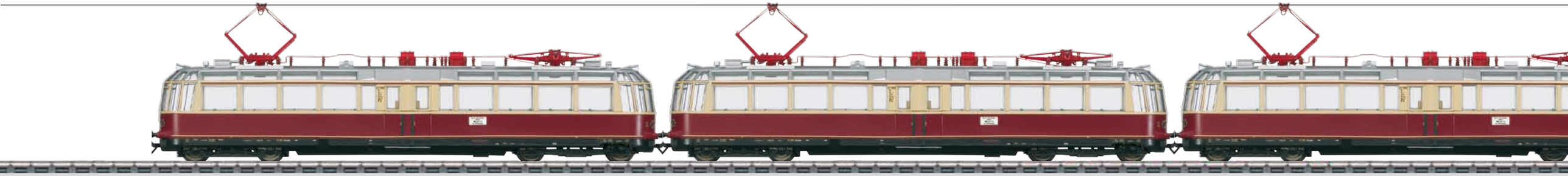
- iterable: implements Iterable interface (__iter__ method)
  - __iter__ method returns an Iterator

- iterator: implements Iterator interface (__next__ method)
  - __next__ method returns next item in series and
  - raises StopIteration to signal end of the series



AN ITERABLE TRAIN

An instance of Train can be iterated, car by car



```
>>> t = Train(3)
>>> for car in t:
...     print(car)
car #1
car #2
car #3
>>>
```

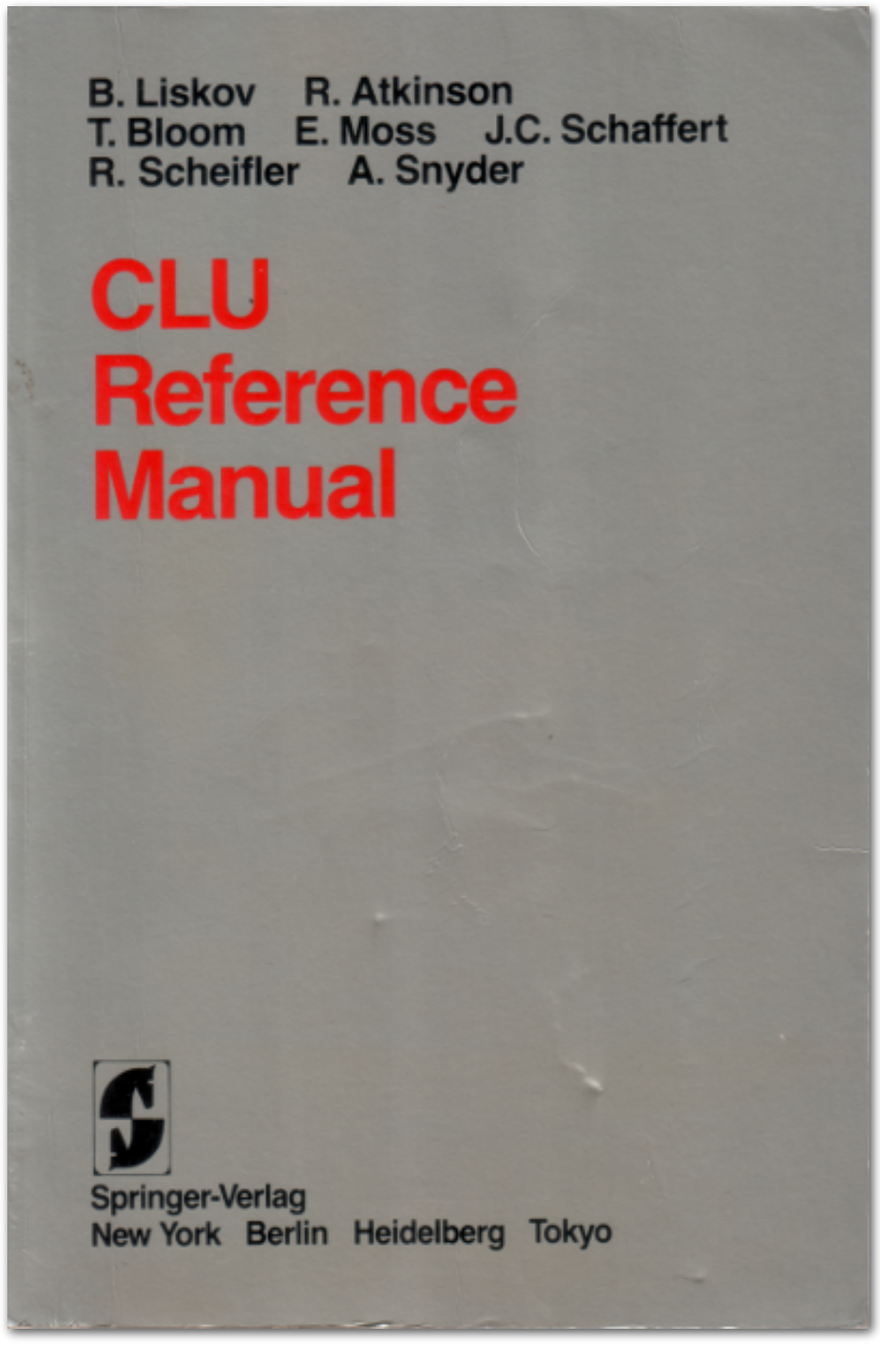
CLASSIC ITERATOR IMPLEMENTATION

The pattern as described by Gamma et. al.

```
class Train:

    def __init__(self, cars):
        self.cars = cars

    def __iter__(self):
        return TrainIterator(self.cars)

class TrainIterator:

    def __init__(self, cars):
        self.next = 0
        self.last = cars - 1

    def __next__(self):
        if self.next <= self.last:
            self.next += 1
            return 'car #%s' % (self.next)
        else:
            raise StopIteration()
```

```
>>> t = Train(4)
>>> for car in t:
...     print(car)
car #1
car #2
car #3
car #4
```

BARBARA LISKOV'S CLU LANGUAGE



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

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
العربية
Deutsch
Español
Français
Italiano
Polski
Português
Русский
中文
8 more
Edit links

CLU (programming language)

From Wikipedia, the free encyclopedia

CLU is a programming language created at the Massachusetts Institute of Technology (MIT) by Barbara Liskov and her students between 1974 and 1975. While it did not find extensive use, it introduced many features that are used widely now, and is seen as a step in the development of object-oriented programming (OOP).

Key contributions include abstract data types,[6] call-by-sharing, iterators, multiple return values (a form of parallel assignment), type-safe parameterized types, and type-safe variant types. It is also notable for its use of classes with constructors and methods, but without inheritance.

CLU	
Paradigm	multi-paradigm: object-oriented, procedural
Designed by	Barbara Liskov and her students
Developer	Massachusetts Institute of Technology
First appeared	1975; 44 years ago
Stable release	Native CLU 1.5 (SPARC, VAX) / May 26, 1989; 30 years ago[1] Portable CLU / November 6, 2009; 9 years ago[2]
Typing discipline	strong
Website	www.pmg.lcs.mit.edu/CLU.html
Major implementations	
Native CLU,[1] Portable CLU,[2] *clu2c*[3]	
Influenced by	
ALGOL 60, Lisp, Simula	
Influenced	
Ada, Argus, C++, Lua, Python,[4] Ruby, Sather, Swift[5]	

Clusters [edit]

The syntax of CLU was based on ALGOL, then the starting point for most new language designs. The key addition was the concept of a *cluster*, CLU's type extension system and the root of the language's name (CLUster).[7] Clusters correspond generally to the concept of a "class" in an OO language, and have similar syntax. For instance, here is the CLU syntax for a cluster that implements complex numbers:

```
complex_number = cluster is add, subtract, multiply, ...
    rep = record [ real_part: real, imag_part: real ]
    add = proc ... end add;
    subtract = proc ... end subtract;
    multiply = proc ... end multiply;
    ...
end complex_number;
```

A cluster is a module that encapsulates all of its components except for those explicitly named in the "is" clause. These correspond to the public components of a class in recent OO languages. A cluster also defines a type that can be named outside the cluster (in this case, "complex_number"), but its representation type (rep) is hidden from external clients.

Cluster names are global, and no namespace mechanism was provided to group clusters or allow them to be created "locally" inside other clusters.

CLU does not perform implicit type conversions. In a cluster, the explicit type conversions *up* and *down* change between the abstract type and the representation. There is a universal type *any*, and a procedure force[] to check that an object is a certain type. Objects may be mutable or immutable, the latter being *base types* such as integers, booleans, characters and strings.[7]

Other features [edit]

Another key feature of the CLU type system are *iterators*, which return objects from a collection serially, one after another.[7] Iterators offer an identical application programming interface (API) no matter what data they are being used with. Thus the iterator for a collection of `complex_number`s can be used interchangeably with that for an array of `integer`s. A distinctive feature of CLU iterators is that they are implemented as coroutines, with each value being provided to the caller via a *yield* statement. Iterators like those in CLU are now a common feature of many modern languages, such as C#, Ruby, and Python, though recently they are often referred to as generators.

CLU also includes exception handling, based on various attempts in other languages; exceptions are raised using `signal` and handled with `except`. Unlike most other languages with exception handling, exceptions are not implicitly resignaled up the calling chain. Also unlike most other languages that provide exception handling, exceptions in CLU are considered part of ordinary execution flow and are considered a "normal" and efficient typesafe way to break out of loops or return from functions; this allows for direct assignment of return values "except when" other conditions apply. Exceptions that are neither caught nor resignaled explicitly are immediately converted into a special failure exception that typically terminates the program.

CLU is often credited as being the first language with type-safe variant types, called *oneofs*, before the language ML had them.

A final distinctive feature in CLU is parallel assignment (multiple assignment), where more than one variable can appear on the left hand side of an assignment operator. For instance, writing `x,y := y,x` would exchange values of `x` and `y`. In the same way, functions could return several values, like `x,y,z := f(t)`. Parallel assignment (though not multiple return values) predates CLU, appearing in CPL (1963), named *simultaneous assignment*,[8] but CLU popularized it and is often credited as the direct influence leading to parallel assignment in later languages.

All objects in a CLU program live in the heap, and memory management is automatic.

CLU supports type parameterized user-defined data abstractions. It was the first language to offer type-safe bounded parameterized types, using structure *where clauses* to express constraints on actual type arguments.

Influence on other programming languages [edit]

CLU has influenced many other languages in many ways. In approximate chronological order, these include:

CLU and Ada were major inspirations for C++ templates.

CLU's exception handling mechanisms influenced later languages like C++ and Java.

Sather, Python, and C# include iterators, which first appeared in CLU.[*citation needed*]

Perl and Lua took multiple assignment and multiple returns from function calls from CLU.[9]

Python and Ruby borrowed several concepts from CLU, such as call by sharing, the *yield* statement, and multiple assignment[*citation needed*]

References [edit]

1. ^ a b Curtis, Dorothy (2009-11-06). "CLU home page". *Programming Methodology Group, Computer Science and Artificial Intelligence Laboratory*. Massachusetts Institute of Technology. Retrieved 2016-05-26.
2. ^ a b Curtis, Dorothy (2009-11-06). "Index of /pub/pclu". *Programming Methodology Group, Computer Science and Artificial Intelligence Laboratory*. Massachusetts Institute of Technology. Retrieved 2016-05-26.
3. ^ Ushijima, Tetsu. "clu2c". *clu2c*. woodsheep.jp. Retrieved 2016-05-26.
4. ^ Lundh, Fredrik. "Call By Object". *effbot.org*. Retrieved 21 November 2017. "replace "CLU" with "Python", "record" with "instance", and "procedure" with "function or method", and you get a pretty accurate description of Python's object model."
5. ^ Lattner, Chris (2014-06-03). "Chris Lattner's Homepage". Chris Lattner. Retrieved 2014-06-03. "The Swift language is the product of tireless effort from a team of language experts, documentation gurus, compiler optimization ninjas, and an incredibly important internal dogfooding group who provided feedback to help refine and battle-test ideas. Of course, it also greatly benefited from the experiences hard-won by many other languages in the field, drawing ideas from Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, and far too many others to list."
6. ^ Liskov, Barbara; Zilles, Stephen (1974). "Programming with abstract data types". *Proceedings of the ACM SIGPLAN symposium on Very high level languages*. pp. 50–59. CiteSeerX 10.1.1.136.3043. doi:10.1145/800233.807045.
7. ^ a b c Liskov, B.; Snyder, A.; Atkinson, R.; Schaffert, C. (August 1977). "Abstraction mechanisms in CLU". *Communications of the ACM*. **20** (8): 564–576. CiteSeerX 10.1.1.112.656. doi:10.1145/359763.359789.
8. ^ Barron, D. W.; Buxton, J. N.; Hartley, D. F.; Nixon, E.; Strachey, C. (1963). "The main features of CPL". *Computer Journal*. **6** (2): 134–143. doi:10.1093/comjnl/6.2.134.
9. ^ Ierusalimschy, R.; De Figueiredo, L. H.; Celes, W. (2007). "The evolution of Lua". *Proceedings of the third ACM SIGPLAN conference on History of programming languages – HOPL III* (PDF). pp. 2–1–2–26. doi:10.1145/1238844.1238846. ISBN 978-1-59593-766-7.

External links [edit]

- Official website
- A History of CLU (pdf)
- clu2c: a program to compile CLU code to C
- Dictionary of Programming Languages
- CLU comparison at '99 bottles of beer' multi-language demo algorithm site

Categories: Academic programming languages | Class-based programming languages | Massachusetts Institute of Technology software | Procedural programming languages | Programming languages created in 1975 | Programming languages created by women

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
العربية
Deutsch
Español

CLU (programming language)

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.

Find sources: "CLU" programming language – news · newspapers · books · scholar · JSTOR *(February 2013) (Learn how and when to remove this template message)*

CLU is a programming language created at the Massachusetts Institute of Technology (MIT) by Barbara Liskov and her students between 1974 and 1975. While it did not find extensive use, it introduced many features that are used widely now, and is seen as a step in the development of object-oriented programming (OOP).

Key contributions include abstract data types,[6] call-by-sharing, iterators, multiple return values (a form of parallel assignment), type-safe parameterized types, and type-safe variant types. It is also notable for its use of classes with constructors and methods, but without inheritance.

Contents [hide]

Clusters [edit]

The syntax of CLU was based on ALGOL, then the starting point for most new language designs. The key addition was the concept of a *cluster*, CLU's type extension system and the root of the language's name (CLUster).[7] Clusters correspond generally to the concept of a "class" in an OO language, and have similar syntax. For instance, here is the CLU syntax for a cluster that implements complex numbers:

```
complex_number = cluster is add, subtract, multiply, ...
    rep = record [ real_part: real, imag_part: real ]
```

CLU

Paradigm	multi-paradigm: object-oriented, procedural
Designed by	Barbara Liskov and her students
Developer	Massachusetts Institute of Technology
First appeared	1975; 44 years ago
Stable release	Native CLU 1.5 (SPARC, VAX) / May 26, 1989; 30 years ago[1] Portable CLU / November 6, 2009; 9 years ago[2]
Typing discipline	strong
Website	www.pmg.lcs.mit.edu/CLU.html
Major implementations	
Native CLU,[1] Portable CLU,[2] clu2c[3]	
Influenced by	
ALGOL 60, Lisp, Simula	
Influenced	
Ada, Argus, C++, Lua, Python,[4] Ruby, Sather, Swift[5]	

CLU is a programming language created at the Massachusetts Institute of Technology (MIT) by Barbara Liskov and her students between 1974 and 1975. While it did not find extensive use, it introduced many features that are used widely now, and is seen as a step in the development of object-oriented programming (OOP).

Key contributions include abstract data types,[6] call-by-sharing, iterators, multiple return values (a form of parallel assignment), type-safe parameterized types, and type-safe variant types. It is also notable for its use of classes with constructors and methods, but without inheritance.

Contents [hide]

Clusters [edit]

The syntax of CLU was based on ALGOL, then the starting point for most new language designs. The key addition was the concept of a *cluster*. CLU's type extension system and the root of the

"complex_number"), but its representation type (rep) is hidden from external clients.

Cluster names are global, and no namespace mechanism was provided to group clusters or allow t
clusters.

CLU does not perform implicit type conversions. In a cluster, the explicit type conversions *up* and *do*
the representation. There is a universal type *any*, and a procedure force[] to check that an object is
immutable, the latter being *base types* such as integers, booleans, characters and strings.[7]

Other features [edit]

Another key feature of the CLU type system are *iterators*, which return objects from a collection ser
identical application programming interface (API) no matter what data they are being used with. Th
`complex_number`s can be used interchangeably with that for an array of `integer`s. A distinctiv
implemented as coroutines, with each value being provided to the caller via a *yield* statement. Iterat
feature of many modern languages, such as C#, Ruby, and Python, though recently they are often r

CLU also includes exception handling, based on various attempts in other languages; exceptions ar
`except`. Unlike most other languages with exception handling, exceptions are not implicitly resign
other languages that provide exception handling, exceptions in CLU are considered part of ordinary
"normal" and efficient typesafe way to break out of loops or return from functions; this allows for dire
when" other conditions apply. Exceptions that are neither caught nor resignaled explicitly are immed
exception that typically terminates the program.

ITERATION IN CLU

CLU also introduced *true iterators* with **yield** and a generic **for/in** statement.

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:

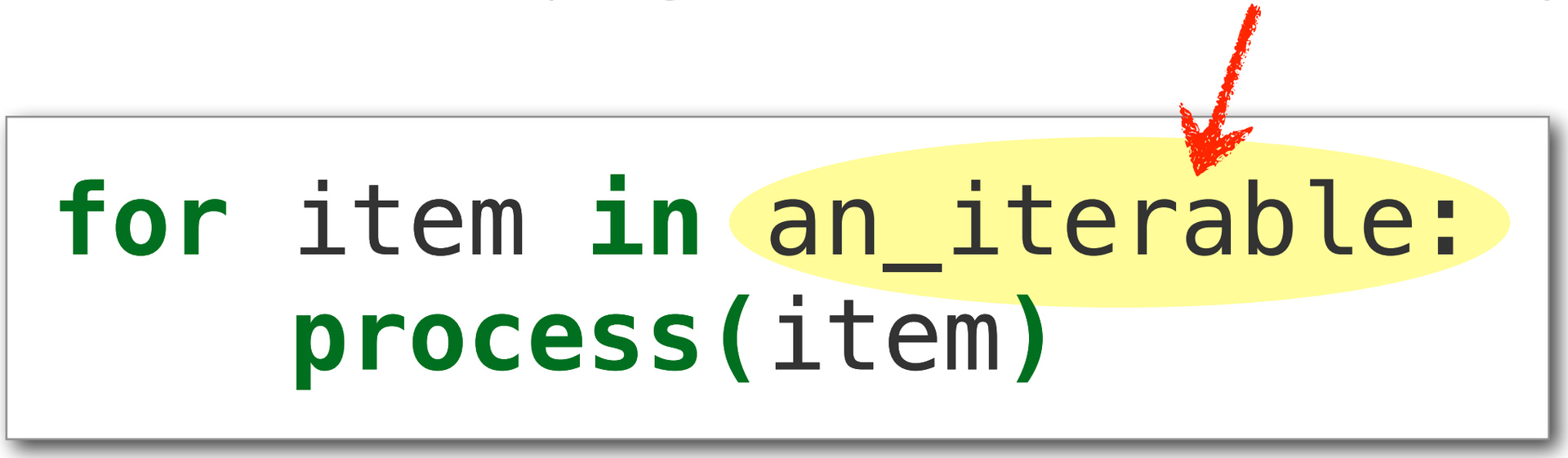
```
for leaf: node in leaves(x) do
    ... examine(leaf) ...
end
```

1.3 Clusters

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

ITERABLE OBJECTS: THE KEY TO FOREACH

- Python, Java & CLU let programmers define iterable objects



- Some languages don't offer this flexibility
  - C has no concept of iterables
  - In Go, only some built-in types are iterable and can be used with foreach (written as the for ... range special form)

ITERABLE TRAIN WITH A GENERATOR METHOD

The Iterator pattern as a language feature:

```
class Train:

    def __init__(self, cars):
        self.cars = cars

    def __iter__(self):
        for i in range(self.cars):
            yield 'car #%s' % (i+1)
```

Train is now iterable because __iter__ returns a generator!

```
>>> t = Train(3)
>>> it = iter(t)
>>> it
<generator object __iter__ at 0x…>
>>> next(it), next(it), next(it)
('car #1', 'car #2', 'car #3')
```

COMPARE: CLASSIC ITERATOR × GENERATOR METHOD

The classic Iterator recipe is obsolete in Python since v.2.2 (2001)

```
class Train:

    def __init__(self, cars):
        self.cars = cars

    def __iter__(self):
        return IteratorTrem(self.cars)

class TrainIterator:

    def __init__(self, cars):
        self.next = 0
        self.last = cars - 1

    def __next__(self):
        if self.next <= self.last:
            self.next += 1
            return 'car #%s' % (self.next)
        else:
            raise StopIteration()
```

```
class Train:

    def __init__(self, cars):
        self.cars = cars

    def __iter__(self):
        for i in range(self.cars):
            yield 'car #%s' % (i+1)
```

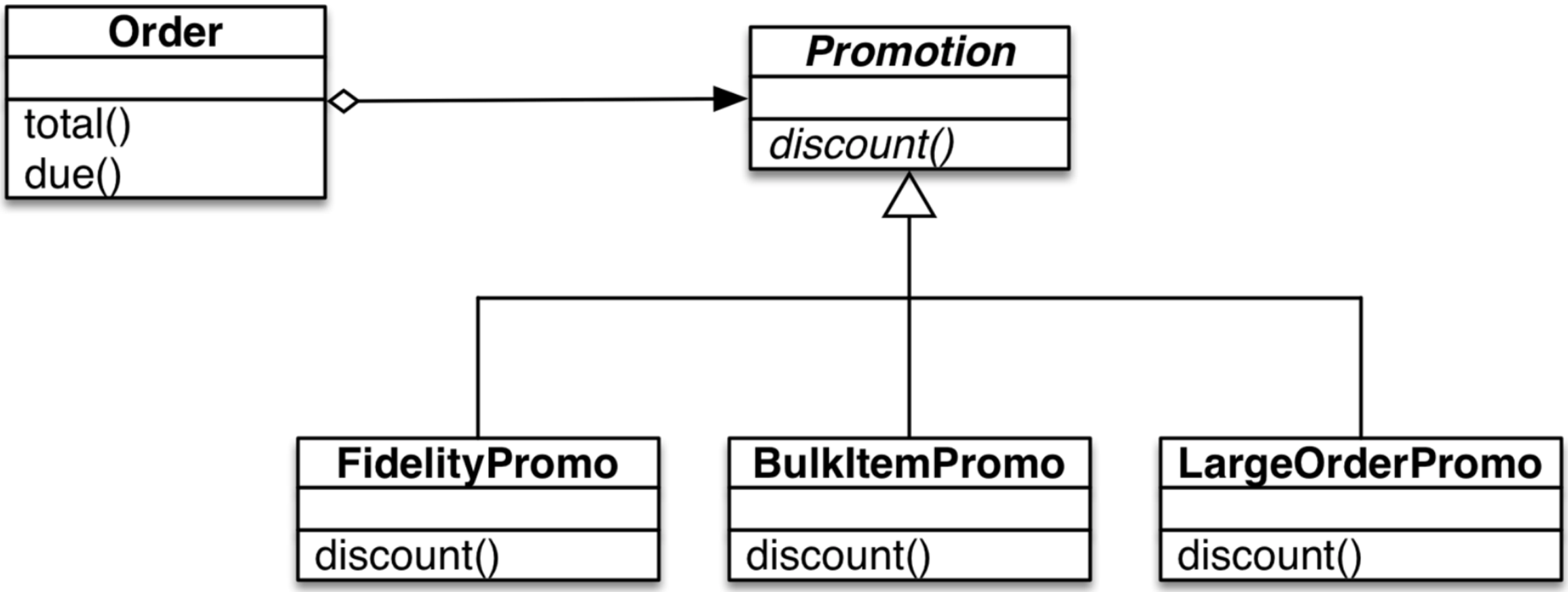
Generator function handles the state of the iteration

ThoughtWorks®

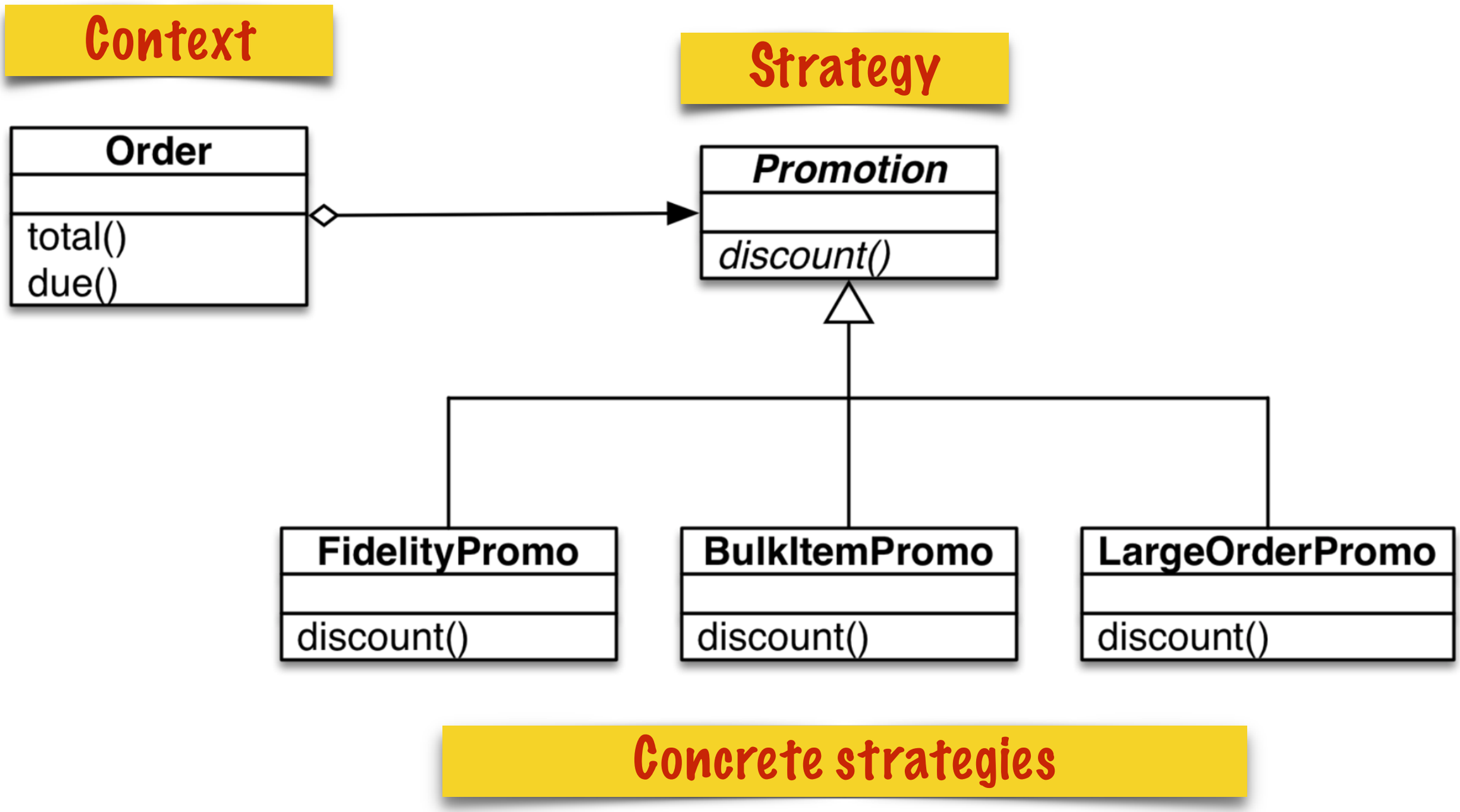
STRATEGY IN PYTHON

Leveraging Python's fundamental features

CHOOSE AN ALGORITHM AT RUN TIME



CHOOSE AN ALGORITHM AT RUN TIME



DOCTESTS FOR CONTEXT AND ONE CONCRETE STRATEGY

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

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

Create a shopping cart with some fruits:

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

The `FidelityPromo` only gives a discount to Ann:

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

Instance of Strategy is given to Order constructor

DOCTESTS FOR TWO ADDITIONAL CONCRETE STRATEGIES

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

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

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

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

VARIATIONS OF STRATEGY IN PYTHON

Classic implementation using ABC

First-class function implementation

Parameterised closure implementation

Parameterised callable implementation

CLASSIC STRATEGY: THE CONTEXT CLASS

Strategy is given to constructor

```
class Order:  # the Context

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

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

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

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

Strategy is used here

STRATEGY ABC AND A CONCRETE STRATEGY

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

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

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

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

TWO CONCRETE STRATEGIES

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

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


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

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

ThoughtWorks®

FIRST-CLASS FUNCTION STRATEGY

CONTEXT: STRATEGY FUNCTION AS ARGUMENT

Strategy function is passed to Order constructor

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

CONTEXT: INVOKING THE STRATEGY FUNCTION

classic_strategy.py ↔ strategy.py

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

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

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

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


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

 @abstractmethod
```

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

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

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

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

fidelity_promo(order):
```

CONCRETE STRATEGIES AS FUNCTIONS

classic_strategy.py ↔ strategy.py

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


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

CONCRETE STRATEGIES AS FUNCTIONS (2)

classic_strategy.py ↔ strategy.py

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


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

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


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

ThoughtWorks®

PARAMETERISED STRATEGY WITH CLOSURE

PARAMETERISED STRATEGY IMPLEMENTED AS CLOSURE

Promo is called with discount percent value

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

PARAMETERISED STRATEGY IMPLEMENTED AS CLOSURE

Outer function gets percent argument

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

Inner function carries percent binding in its closure

LAMBDA: SHORTCUT TO DECLARE INNER FUNCTION

strategy.py ↔ strategy_param.py

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

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

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

ThoughtWorks®

PARAMETERISED STRATEGY WITH CALLABLE

PARAMETERISED STRATEGY IMPLEMENTED AS CLOSURE

Promo is instantiated with discount percent value

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

PROMOTION AS A CALLABLE CLASS

strategy_param.py ↔ strategy_param2.py

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

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

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


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

CONCRETE STRATEGIES AS CALLABLE CONCRETE CLASSES

strategy_param.py ↔ strategy_param2.py

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

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

ThoughtWorks®

WHICH IS MORE IDIOMATIC?

Classes x functions

WHICH IS MORE IDIOMATIC?

Classic strategy *feels* too verbose for Python*

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

WHICH IS MORE IDIOMATIC?

Classic strategy *feels* too verbose for Python*

First-class functions are very common in the standard library

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

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

WHICH IS MORE IDIOMATIC — WITH PARAMETER?

Use of closure is common in Python

- nonlocal was added in Python 3 to support it better

WHICH IS MORE IDIOMATIC — WITH PARAMETER?

Use of closure is common in Python

- nonlocal was added in Python 3 to support it better

Callable objects are uniquely Pythonic

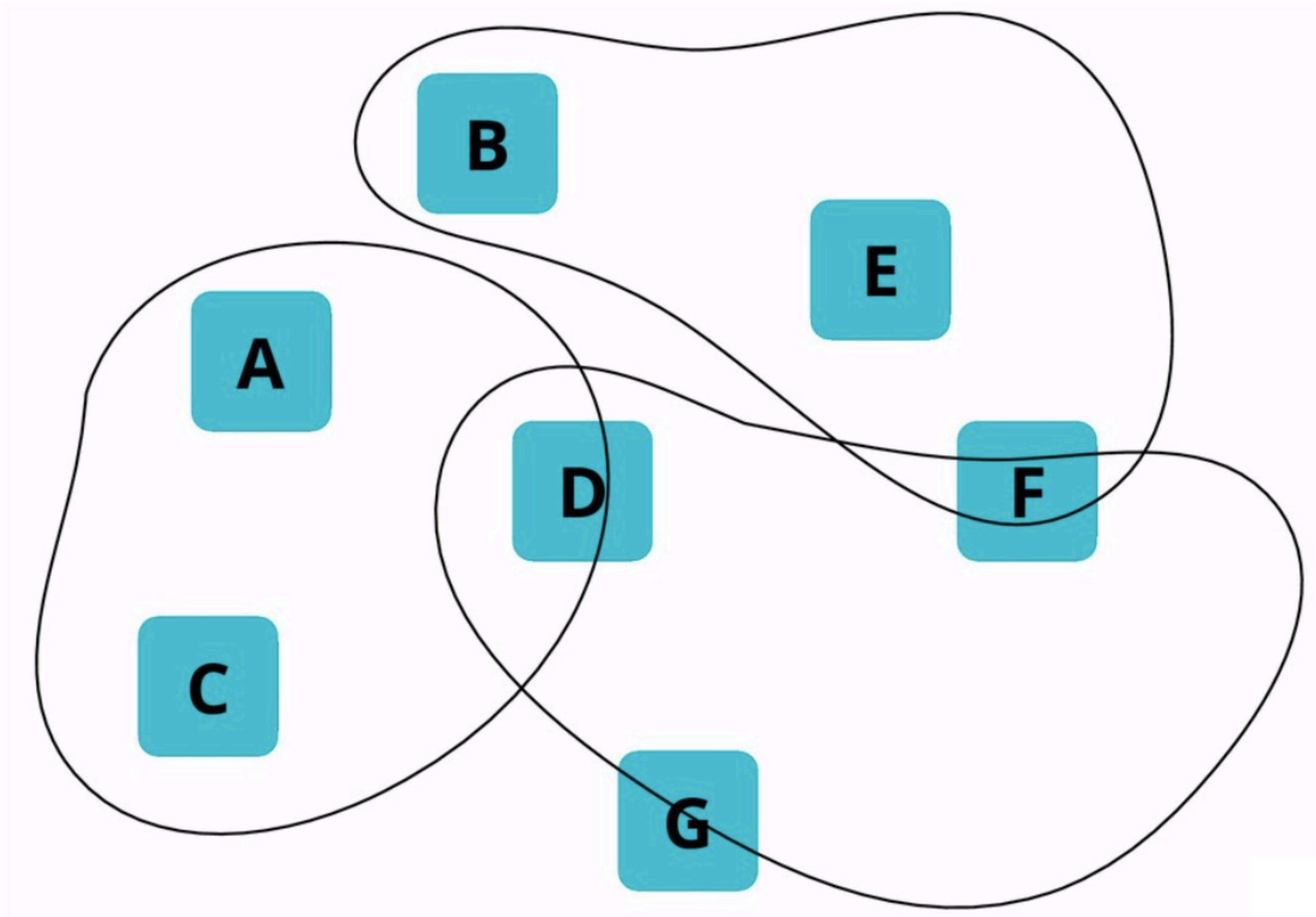
- Graham Dumpleton recommends callable classes as the best way to code decorators

ThoughtWorks®

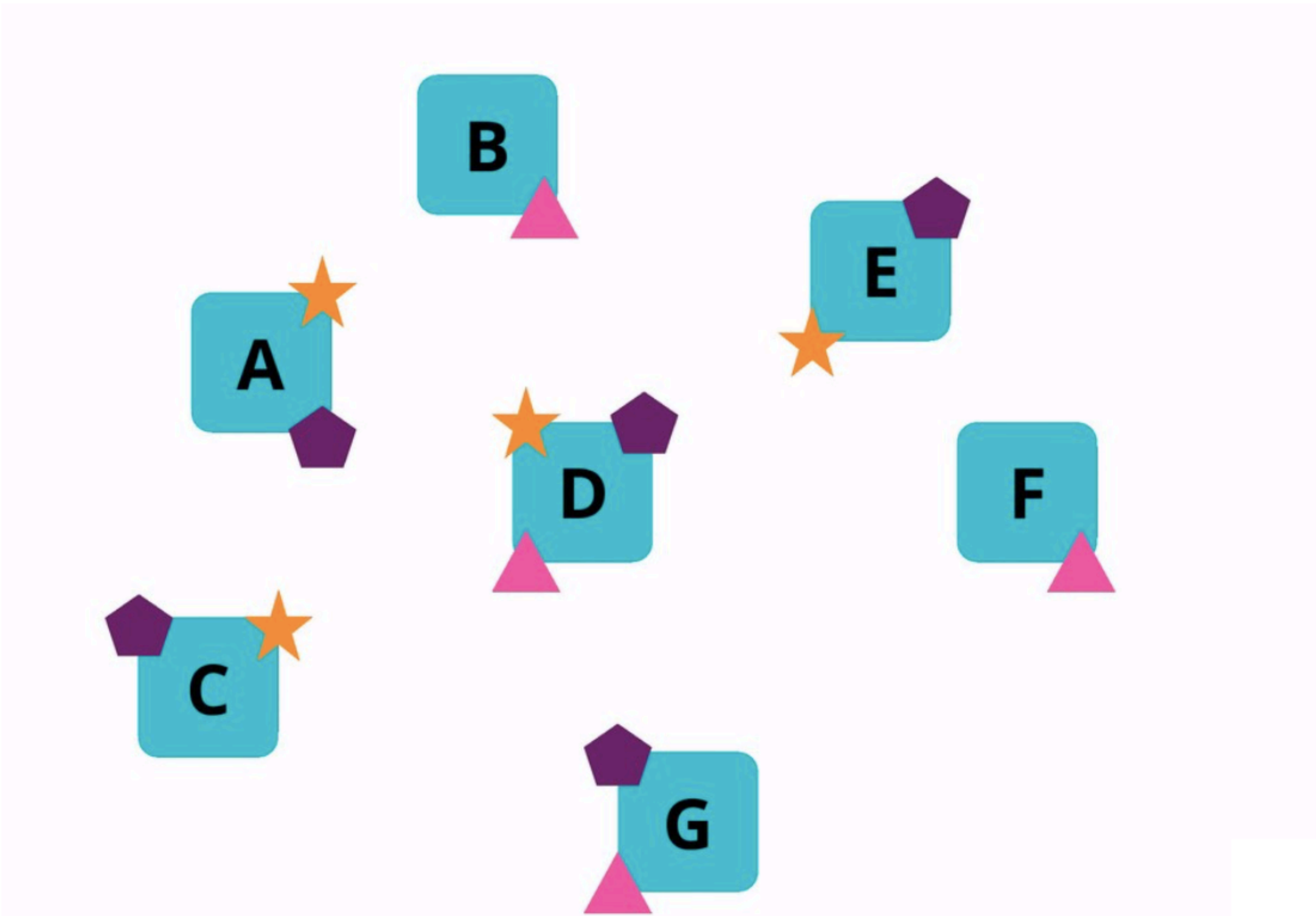
WRAPPING UP

Learn the fundamentals

INSTEAD OF PARADIGMS...



FOCUS ON FUNDAMENTAL FEATURES



FEATURES ARE THE BEST GUIDE...

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

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

WHY LEARN THE FUNDAMENTALS*

Learn new languages faster

Leverage language features

Choose among alternative implementations

Make sensible use of design patterns

Debug hard problems

Emulate missing features when they are helpful

* Inspired by Programming Language Pragmatics

  Michael L. Scott

VIELEN DANK !

Luciano Ramalho
luciano.ramalho@thoughtworks.com

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

ThoughtWorks®