ThoughtWorks®

Theory for practice

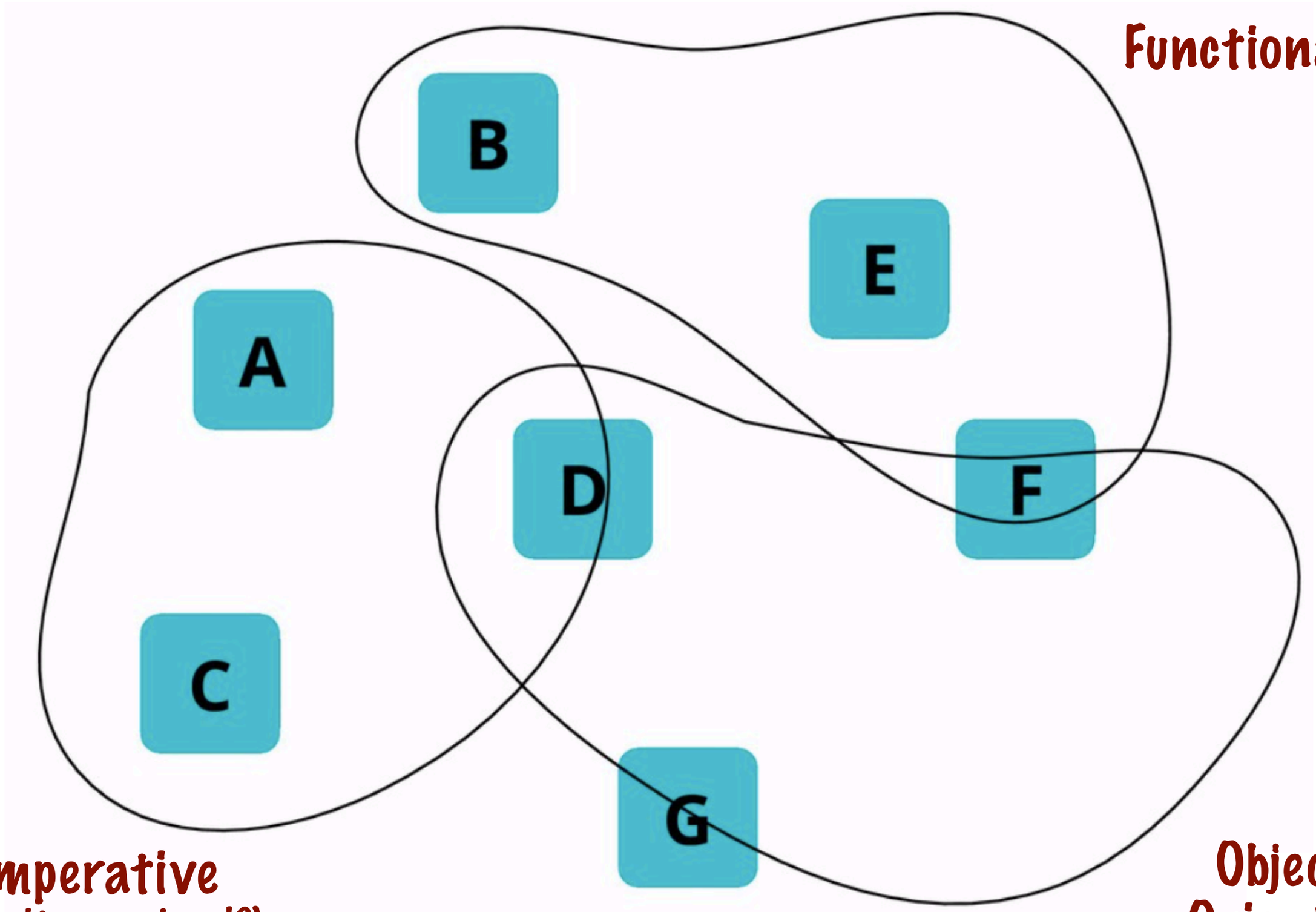
BEYOND PARADIGMS

Understand language features,
use the right patterns.

North Bay Python
July 29th, 2023

Luciano Ramalho
@ramalhoorg@fosstodon.org

LANGUAGES AND PROGRAMMING PARADIGMS



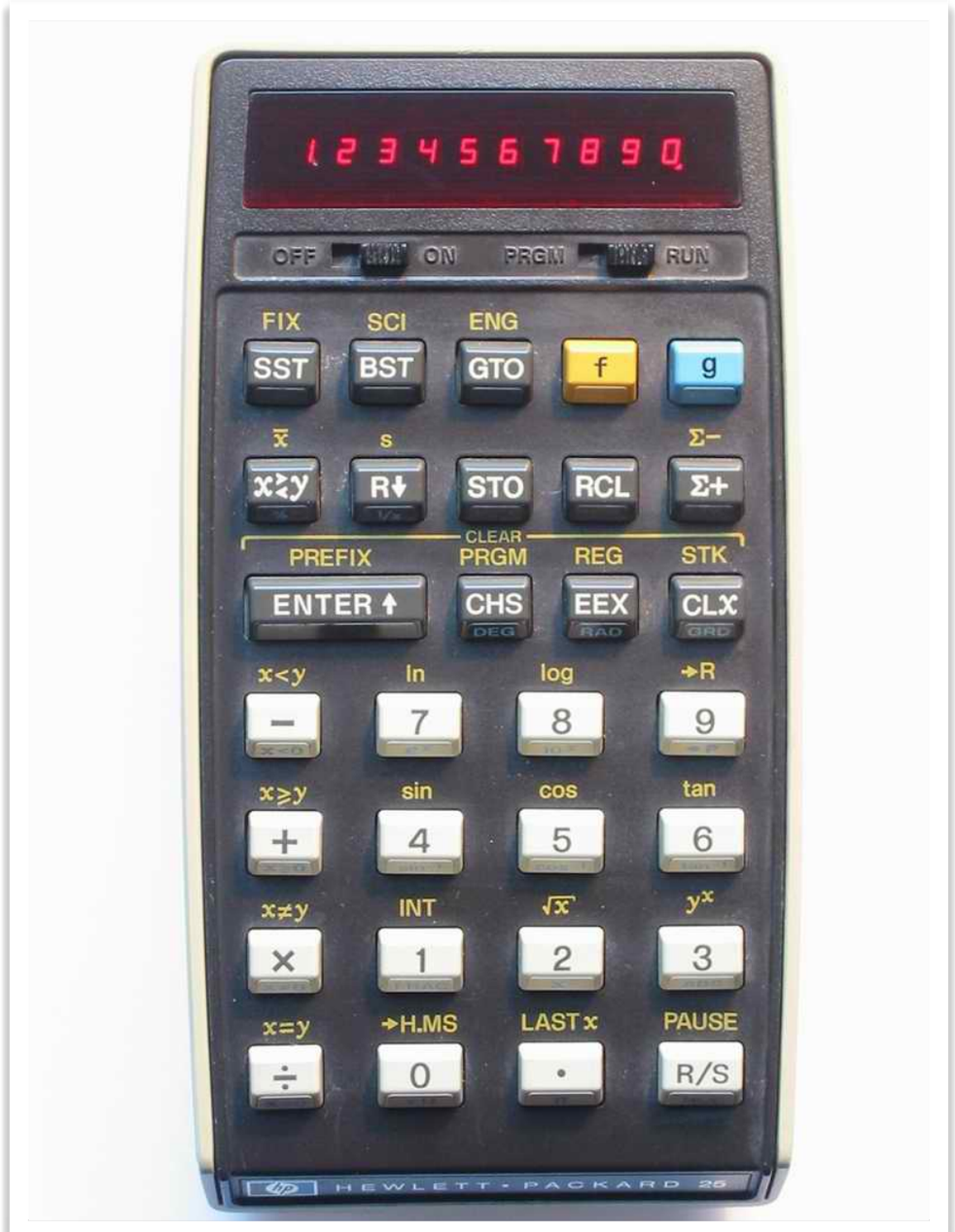
ThoughtWorks®

PARADIGMS

Programming language categories

[redacted]

CALCULATOR PROGRAMMING WAS IMPERATIVE (1976)



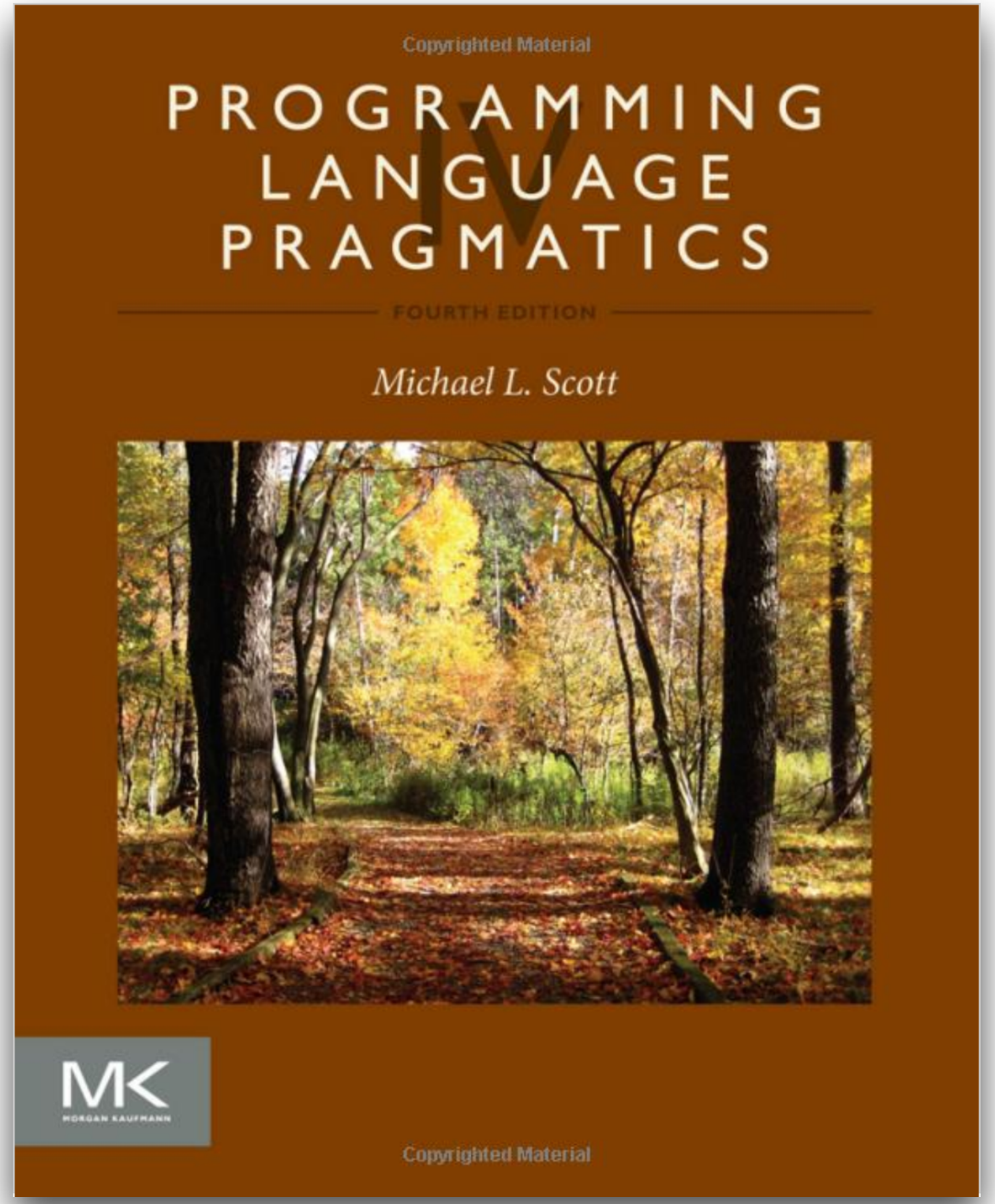
HP-25

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

```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

ThoughtWorks®

THE PROBLEM WITH CATEGORIES

Ontologies are so 1900's

ThoughtWorks®

DEWEY DECIMAL CLASSIFICATION

DDC 23 (2013 edition)

DEWEY'S CLASSIFICATION: A TIDY HIERARCHY

```
000 Computer science, information and general works
100 Philosophy and psychology
200 Religion
300 Social sciences
400 Language
500 Pure science
600 Technology
700 Arts and recreation
800 Literature
900 History and geography
```

A TIDY HIERARCHY

500 Natural sciences and mathematics

510 Mathematics

516 Geometry

516.3 Analytic geometries

516.37 Metric differential geometries

516.375 Finsler geometry

A TIDY HIERARCHY?

---

200 Religion

    210 Philosophy and theory of religion

    220 The Bible

    230 Christianity

    240 Christian practice and observance

    250 Christian orders and local church

    260 Social and ecclesiastical theology

    270 History of Christianity

    280 Christian denominations

    290 Other religions

A TIDY HIERARCHY?

```
200 Religion

    ...

    290 Other religions

        291 No longer used – formerly "Comparative religion"

        292 Classical religion (Greek and Roman religion)

        293 Germanic religion

        294 Religions of Indic origin

        295 Zoroastrianism (Mazdaism, Parseeism)

        296 Judaism

        297 Islam, Bábism and Bahá’í Faith

        298 No longer used – formerly "Mormonism"

        299 Religions not provided for elsewhere

300 Social Sciences
```

ThoughtWorks®

CELESTIAL EMPORIUM

By Jorge Luis Borges (1899-1986)

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*.

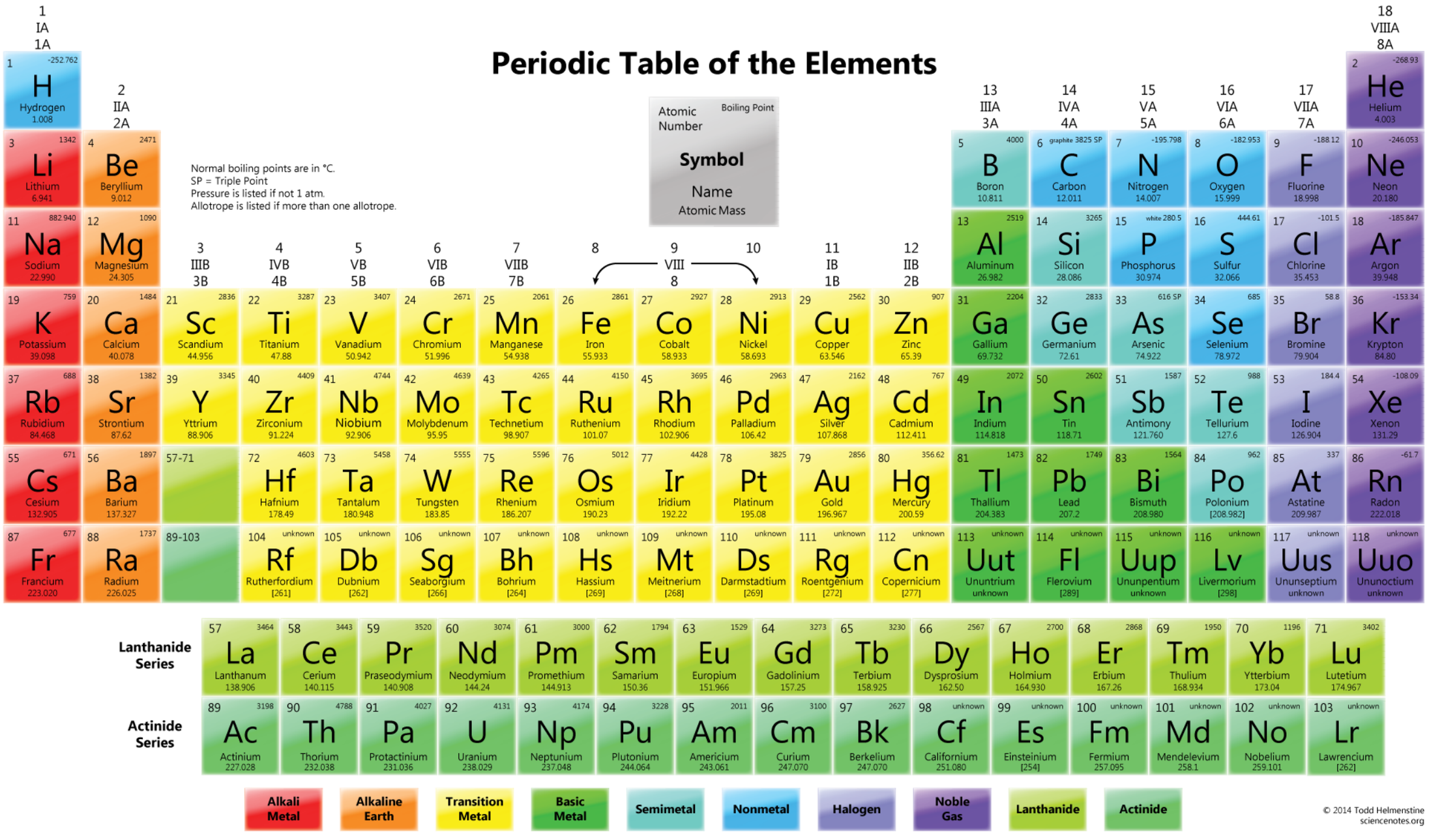
The list divides all animals into 14 categories:

- 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®

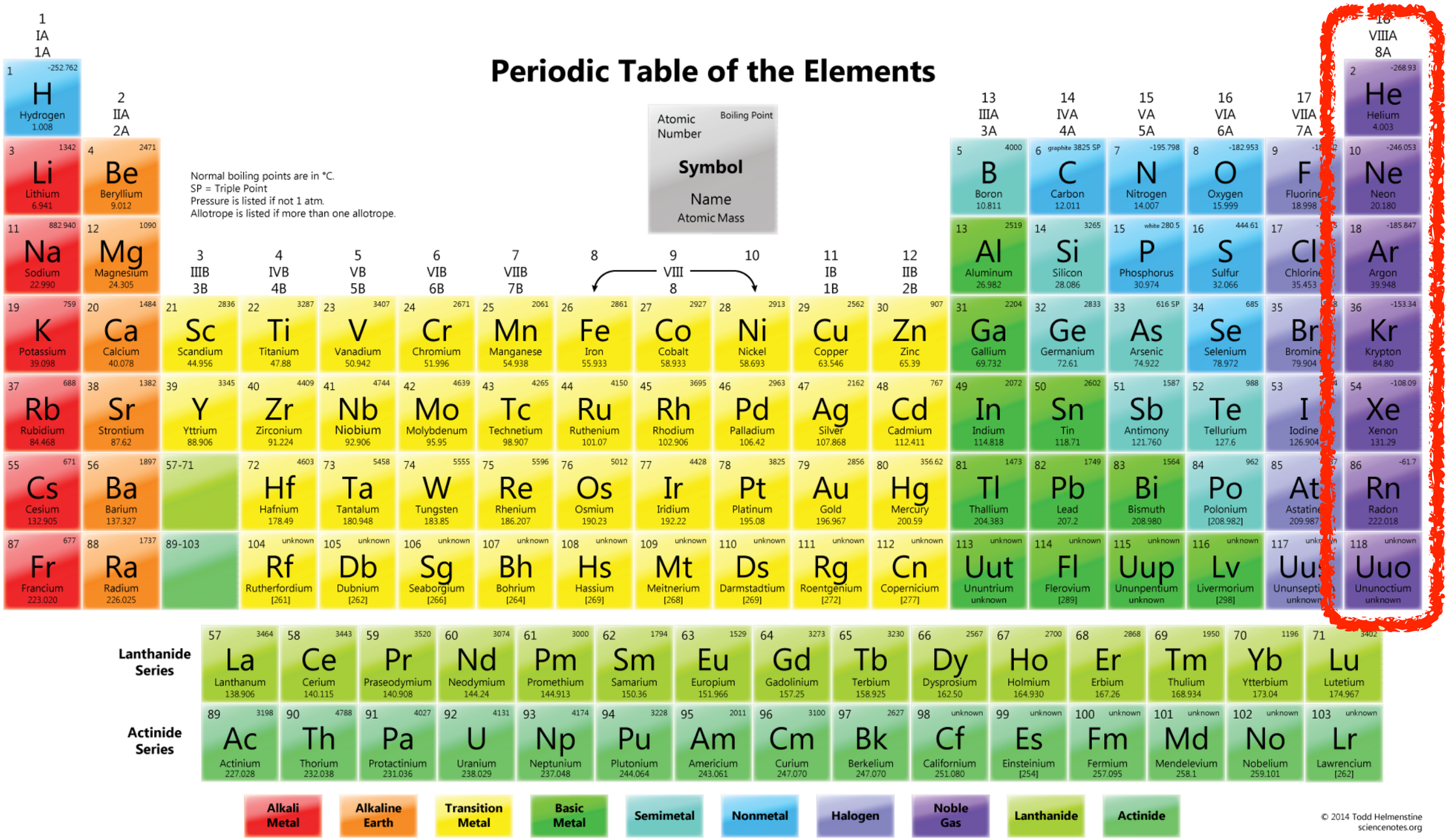
BETTER CATEGORIES?

A CLASSIFICATION BASED ON HARD FACTS



A CLASSIFICATION BASED ON HARD FACTS?

"Noble" gases!?



LANGUAGE CATEGORIES (1)

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.

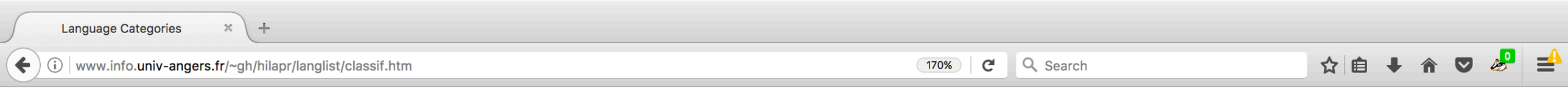
Applicative Language

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)



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

LANGUAGE CATEGORIES (3)

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 (4)

- 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.

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.

“Ontology is overrated.”

Clay Shirky

ThoughtWorks®

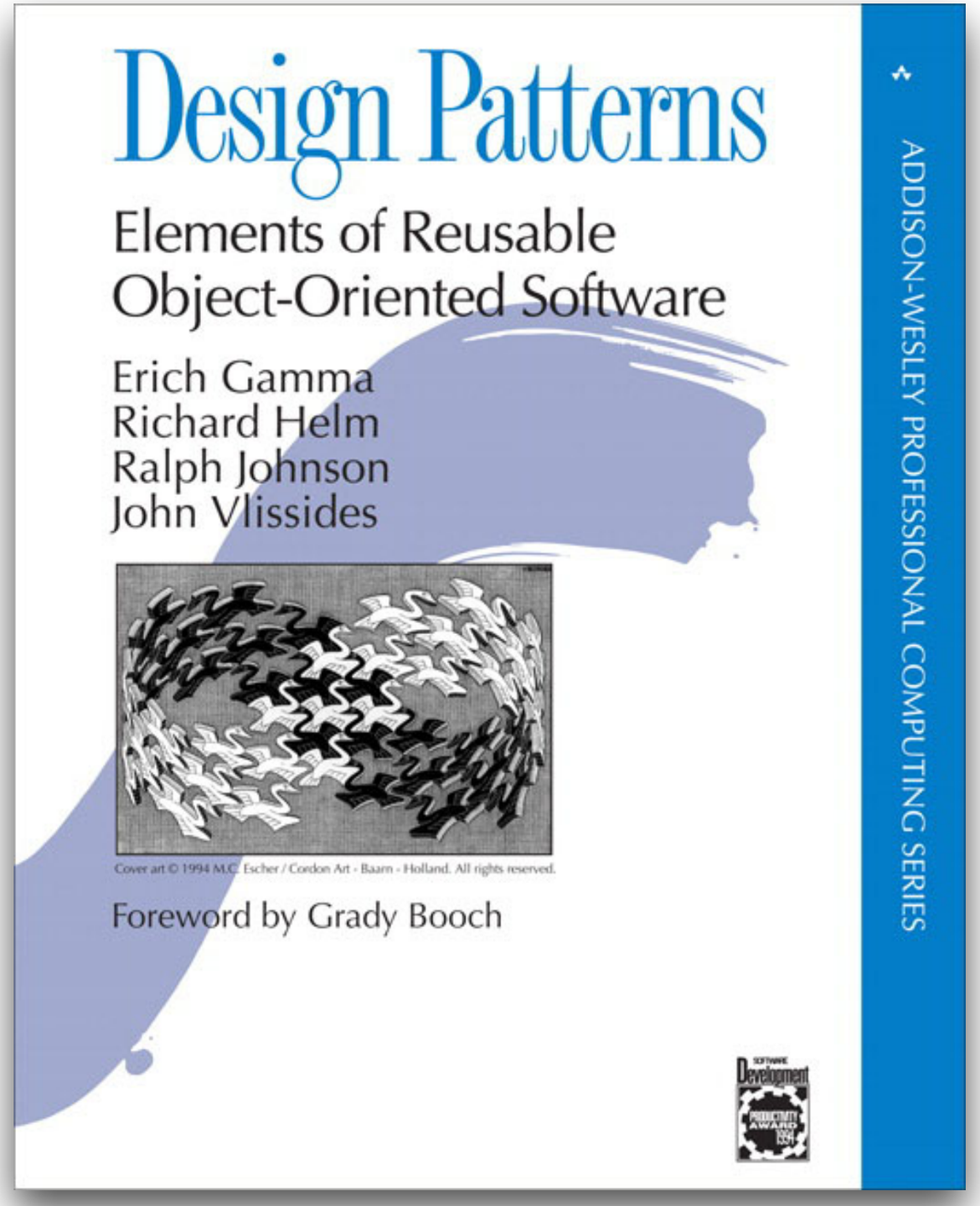
DESIGN PATTERNS

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®

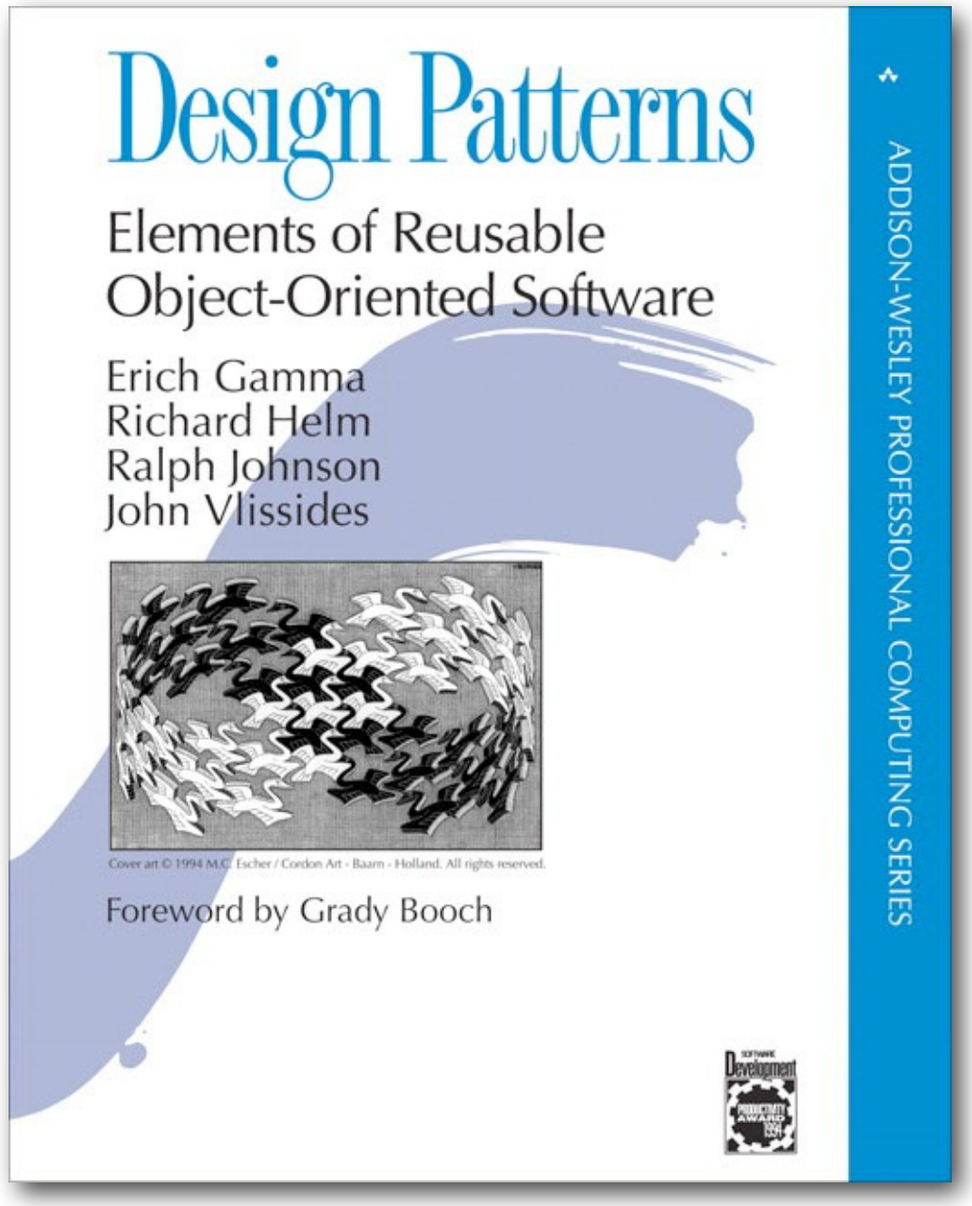
THE ITERATOR PATTERN

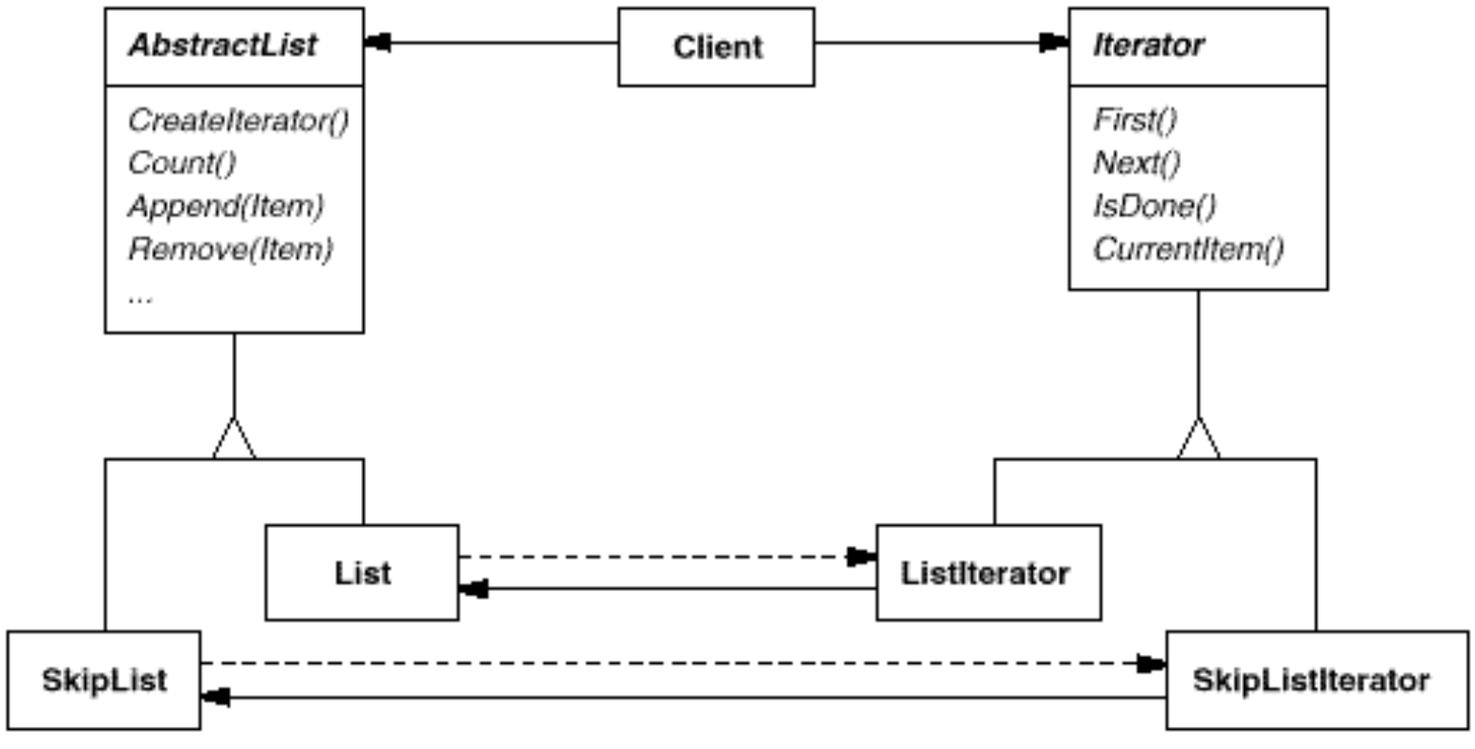
The classic recipe

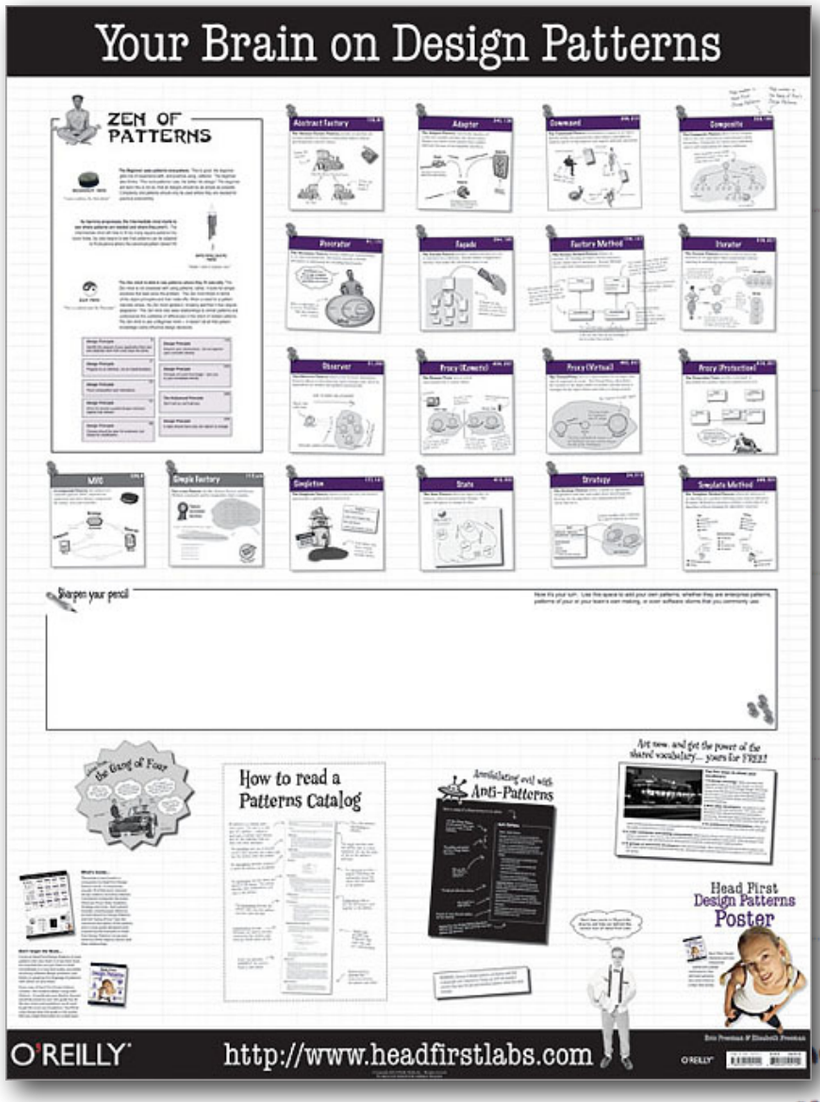
THE ITERATOR FROM THE GANG OF FOUR

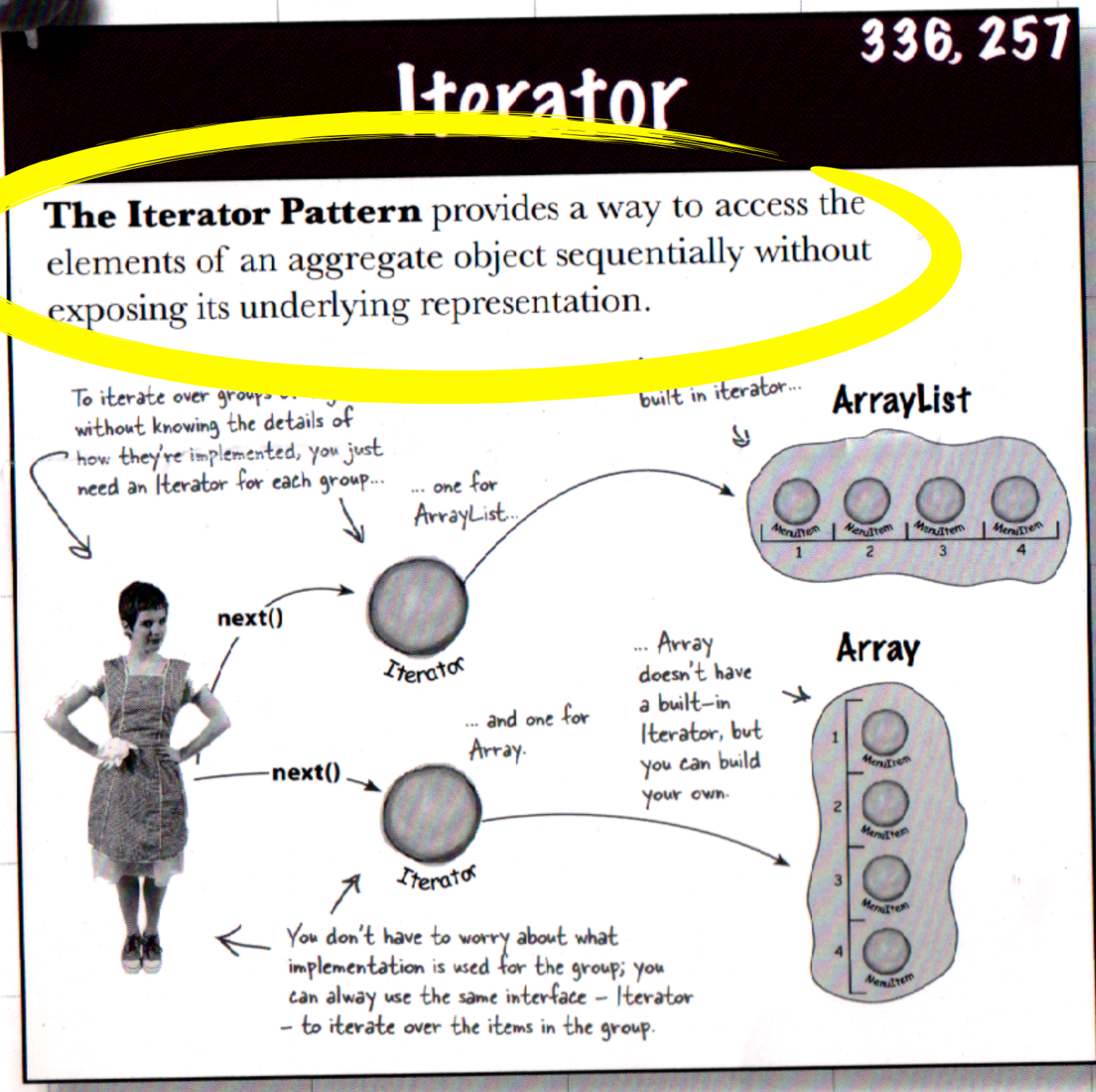
Design Patterns

Gamma, Helm, Johnson & Vlissides









Head First Design Patterns Poster

O'Reilly

ISBN 0-596-10214-3

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)
```

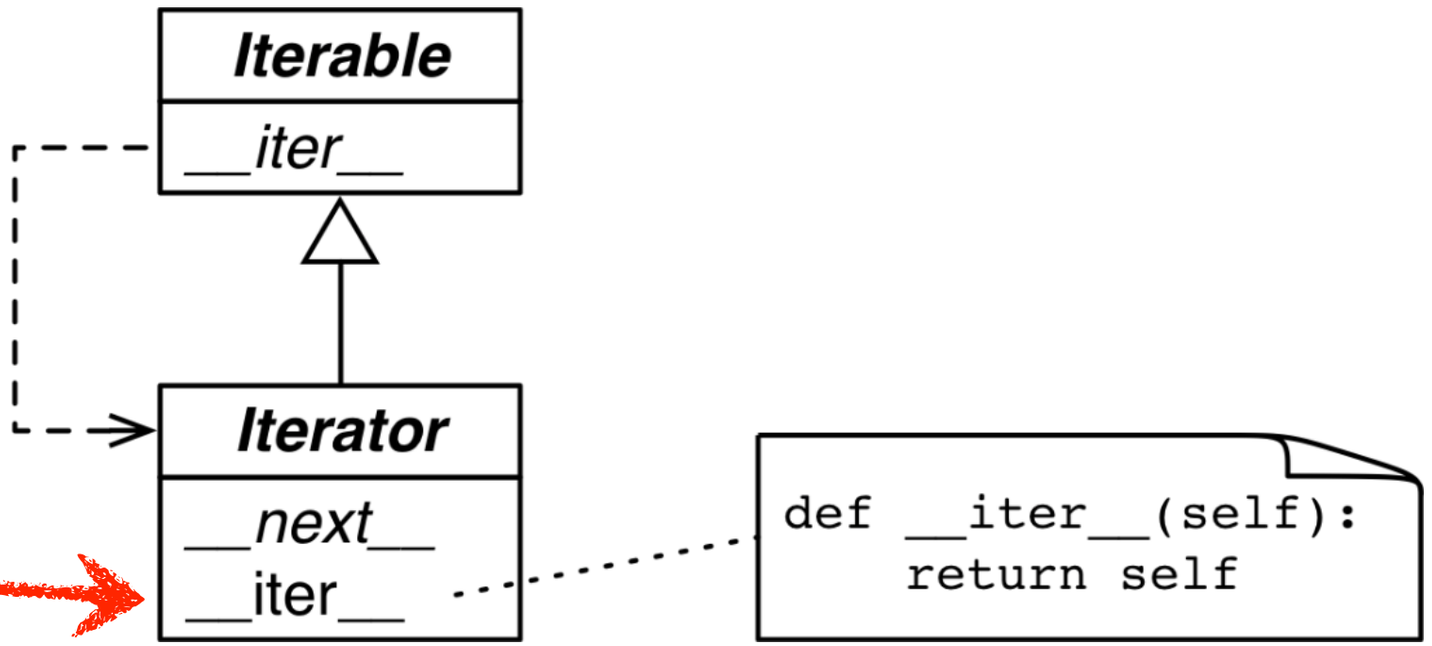
- Terminates when a call to **next()** raises **StopIteration**.

ITERABLE VERSUS ITERATOR

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

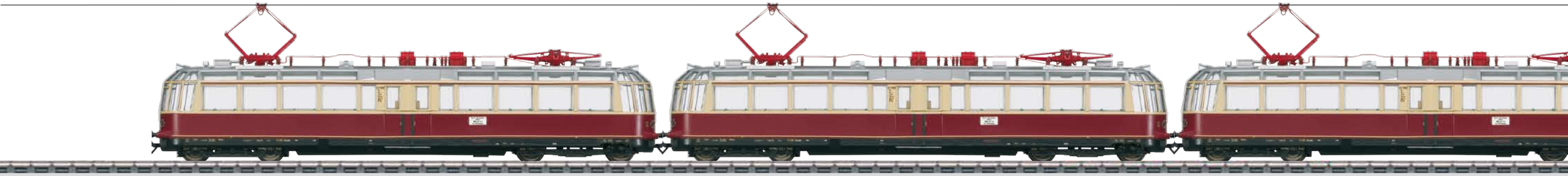
- **iterator**: implements **Iterator** interface (**__next__** method)
  - **__next__** method returns next item in series and
  - raises **StopIteration** to signal end of the series

Python iterators are also iterable!



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
>>>
```

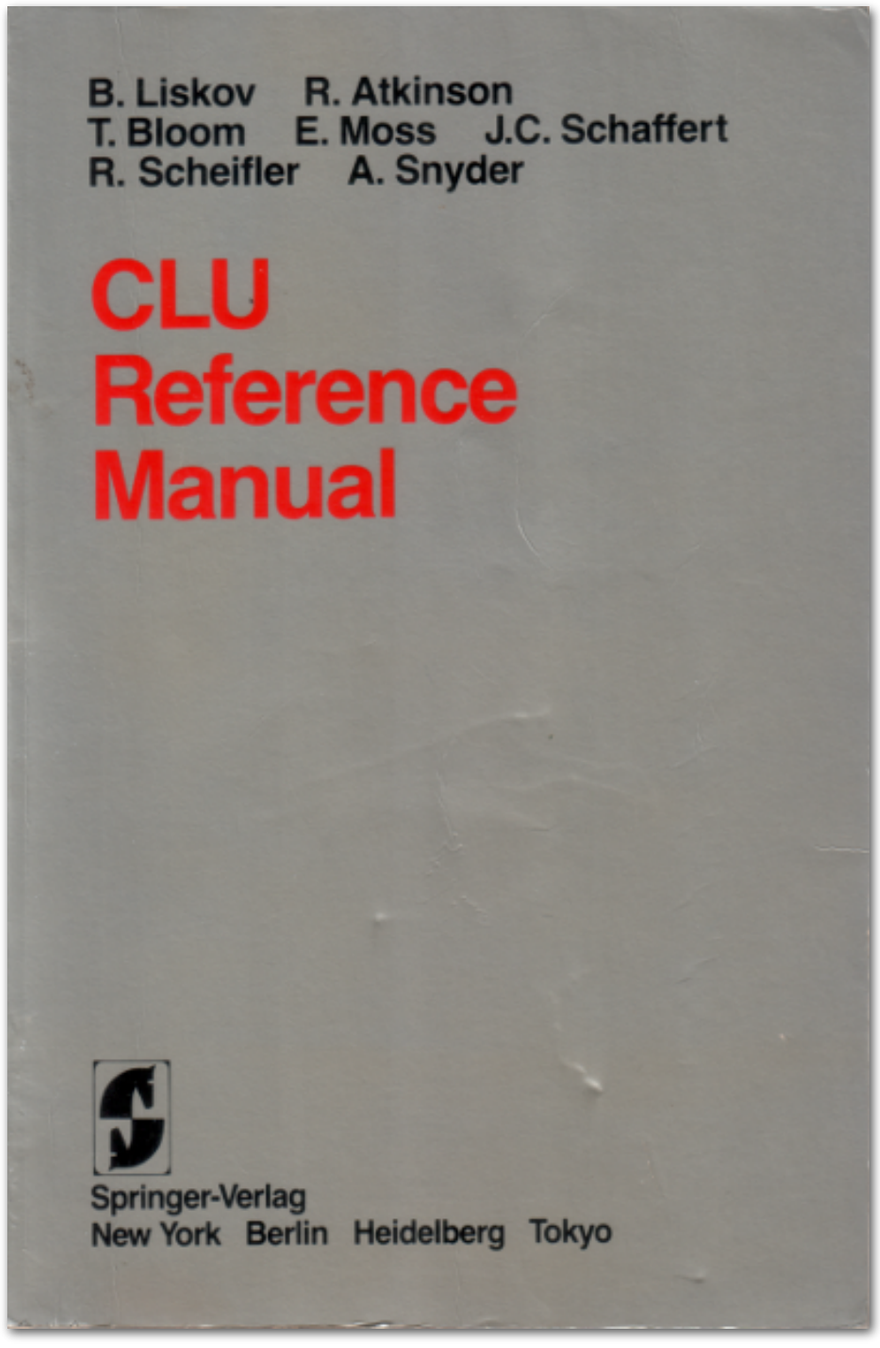
CLASSIC ITERATOR IMPLEMENTATION

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

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

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

- 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®

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

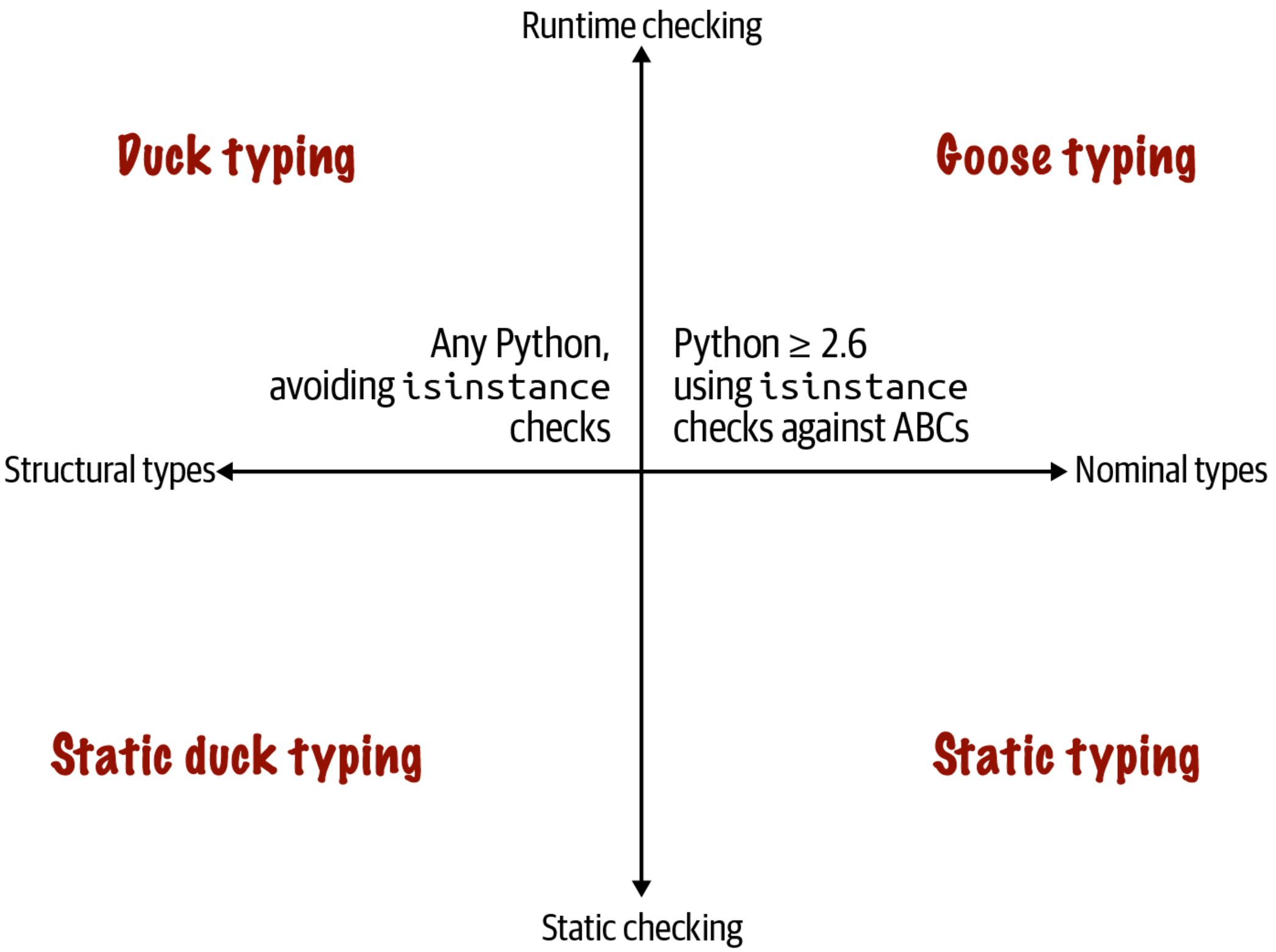
SAMPLE FEATURES ✖ LANGUAGES

	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

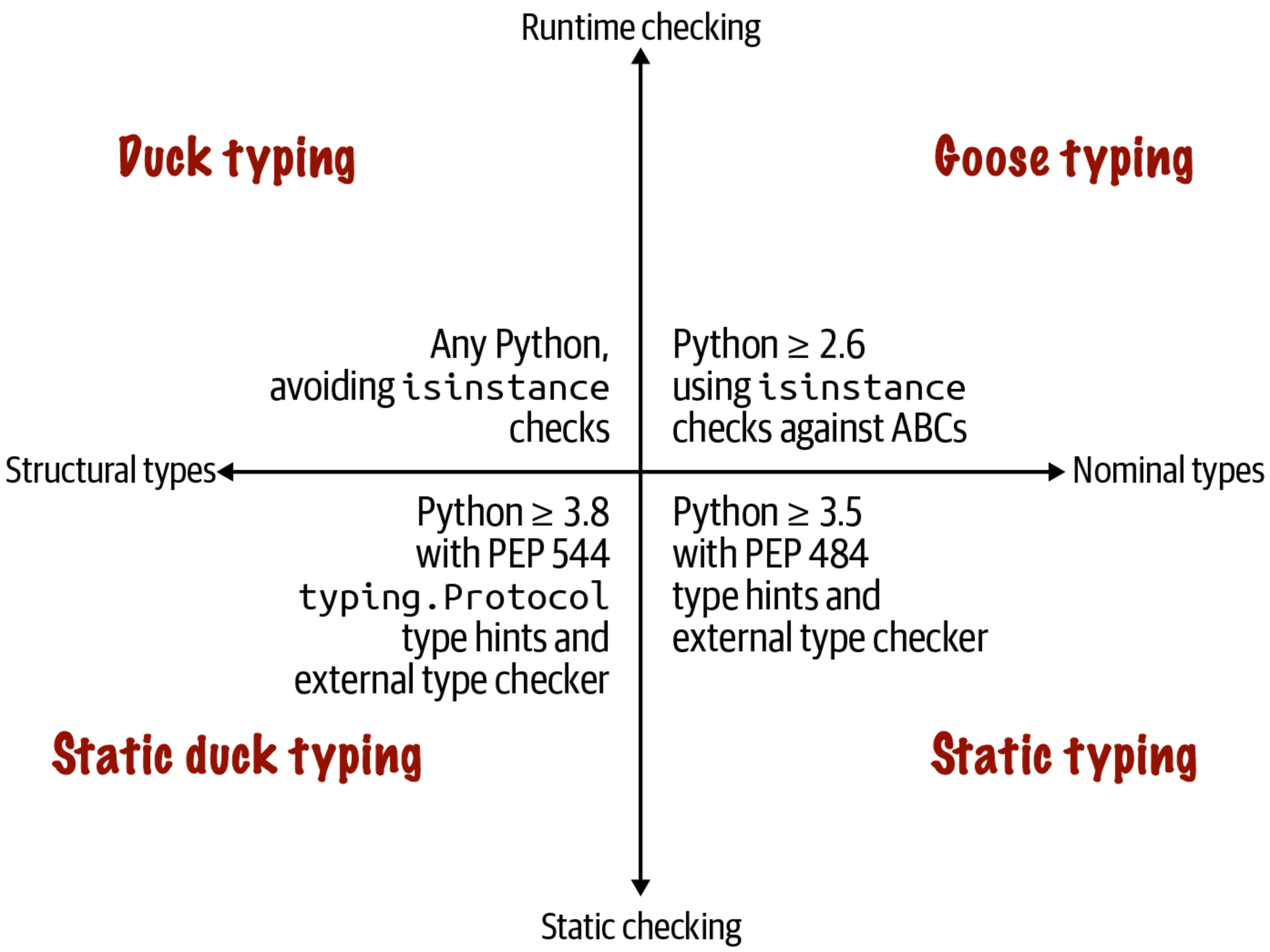
SAMPLE FEATURES ✖ LANGUAGES

	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

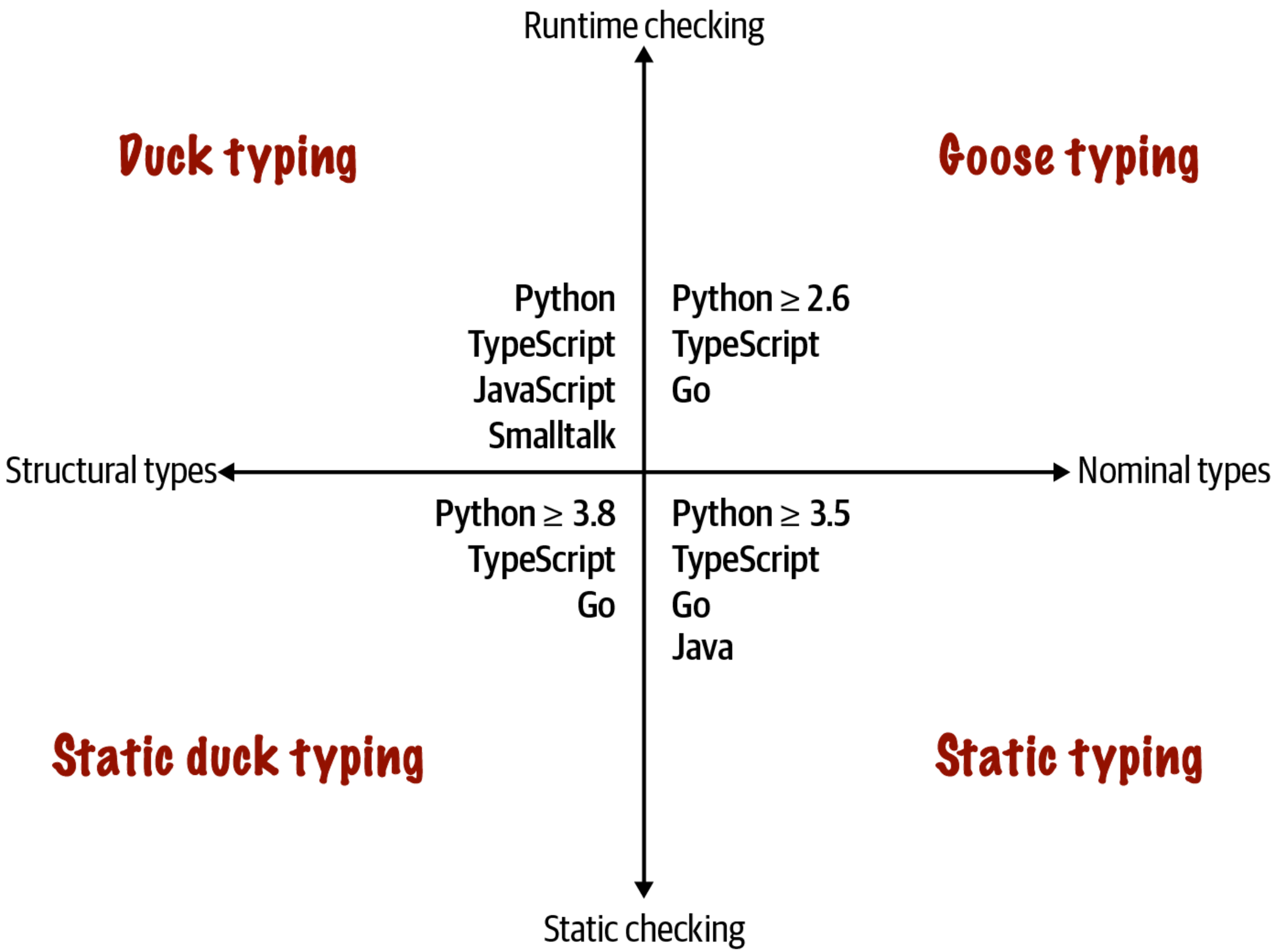
TYPE CHECKING & TYPE EXPRESSION



TYPE CHECKING & TYPE EXPRESSION IN PYTHON



TYPE CHECKING & TYPE EXPRESSION

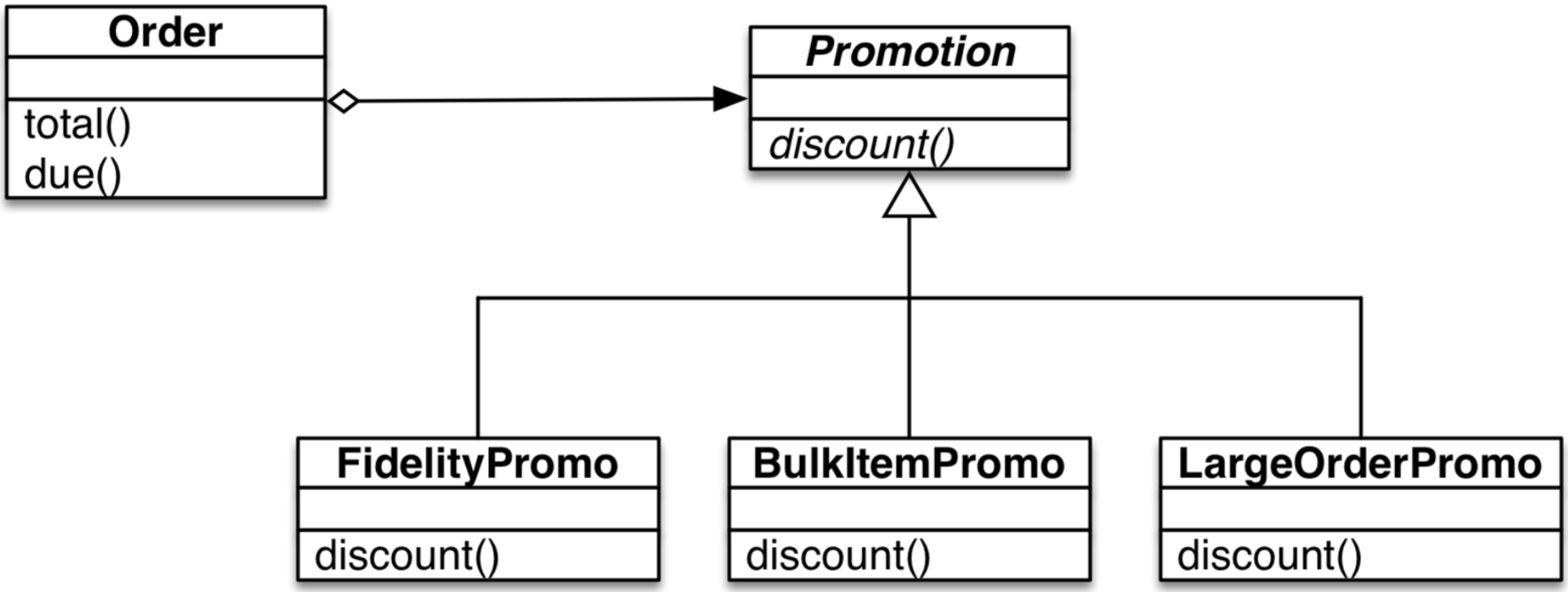


ThoughtWorks®

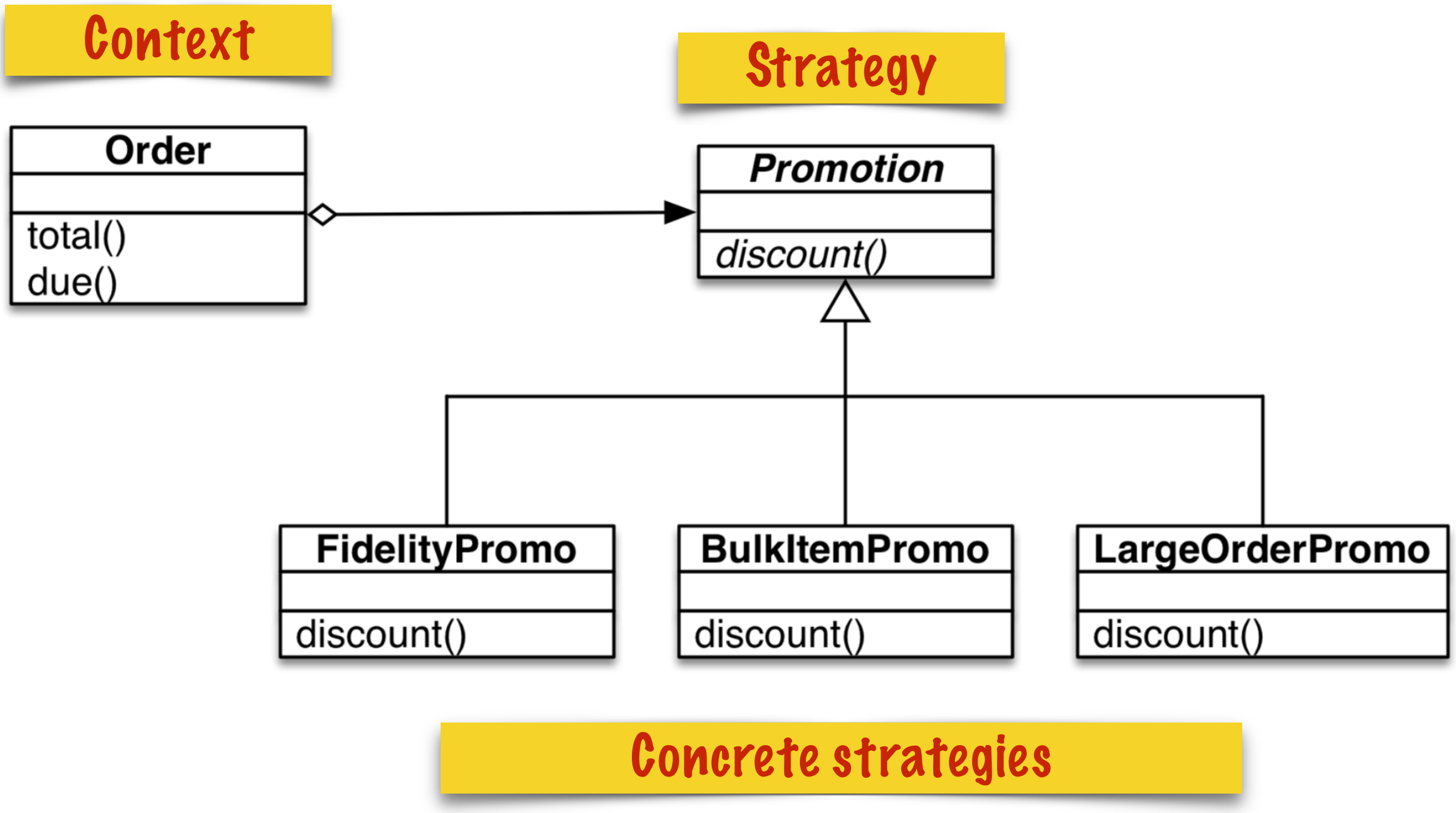
STRATEGY IN PYTHON

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
```

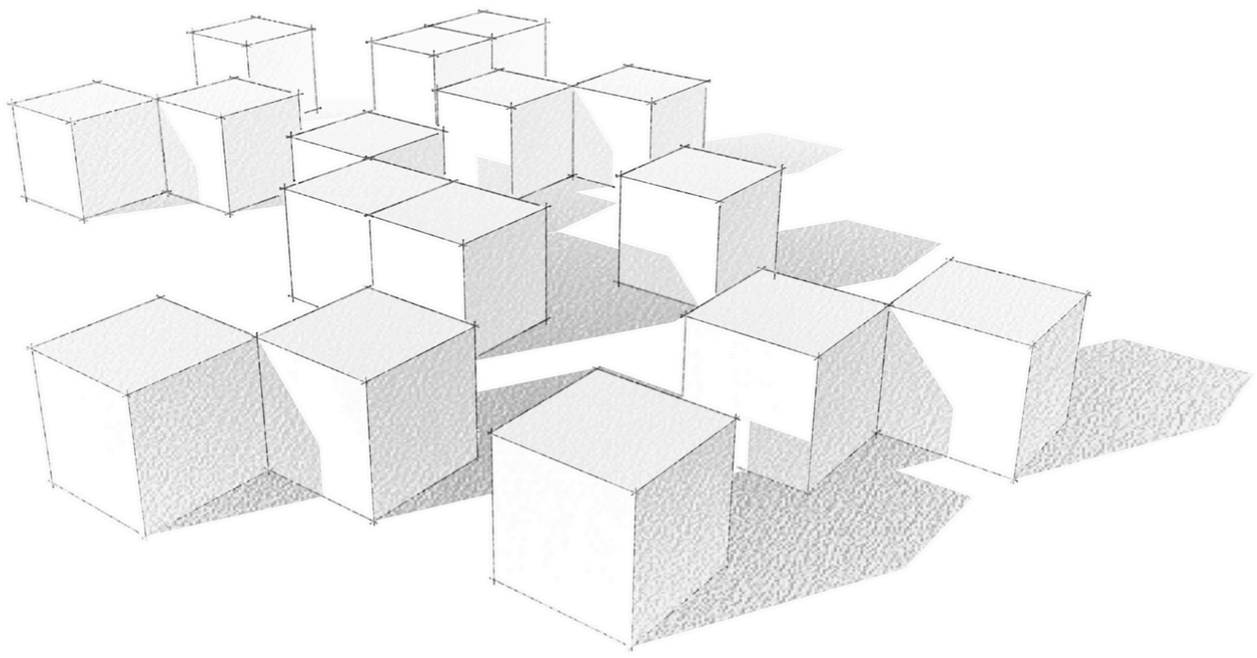
ThoughtWorks®

VECTORIZING

One aspect of functional programming

From Python to Numpy

Latest version - May 2017
DOI: 10.5281/zenodo.225783



File Edit View Run Kernel Settings Help Not Trusted

Code JupyterLab Python 3 (ipykernel)

From objects to vectors

Examples adapted from the the book From Python to Numpy by Nicolas P. Rougier.

Compute the final position of a random walk with `n` steps.

Object oriented approach

[1]:
```
STEPS = 20
```

[2]:
```
import random

class RandomWalker:
    def __init__(self):
        self.position = 0

    def walk(self, n):
        self.position = 0
        for i in range(n):
            yield self.position
            self.position += 2 * random.randint(0, 1) - 1  # ?
```

[3]:
```
%%timeit -n 1000

walker = RandomWalker()
walk = list(walker.walk(STEPS))
```

```
14.9 µs ± 3.81 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

File Edit View Run Kernel Settings Help Not Trusted

Code JupyterLab Python 3 (ipykernel)

Object oriented approach

[1]:
```
STEPS = 20
```

[2]:
```
import random

class RandomWalker:
    def __init__(self):
        self.position = 0

    def walk(self, n):
        self.position = 0
        for i in range(n):
            yield self.position
            self.position += 2 * random.randint(0, 1) - 1  # ?
```

[3]:
```
%%timeit -n 1000

walker = RandomWalker()
walk = list(walker.walk(STEPS))
```

```
14.9 µs ± 3.81 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[4]:
```
walker = RandomWalker()
list(walker.walk(STEPS))
```

[4]:
```
[0, -1, 0, 1, 0, -1, -2, -1, -2, -1, -2, -1, -2, -1, 0, 1, 0, -1, -2, -3]
```

File Edit View Run Kernel Settings Help Not Trusted

Code JupyterLab Python 3 (ipykernel)

Procedural approach

[5]:
```
def random_walk(n):
    position = 0
    walk = [position]
    for i in range(n):
        position += 2*random.randint(0, 1)-1
        walk.append(position)
    return walk
```

[6]:
```
%%timeit -n 1000

walk = random_walk(STEPS)
```

```
11 µs ± 212 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[7]:
```
random_walk(STEPS)
```

[7]:
```
[0, -1, 0, 1, 2, 3, 4, 5, 4, 3, 4, 3, 2, 3, 4, 5, 6, 5, 4, 3, 4]
```

File Edit View Run Kernel Settings Help Not Trusted

Code JupyterLab Python 3 (ipykernel)

Vectorized approach

[8]:
```
from itertools import accumulate

def random_walk_faster(n):
    steps = random.choices([-1,+1], k=n)
    return [0]+list(accumulate(steps))
```

[17]:
```
%%timeit -n 1000

walk = random_walk_faster(STEPS)
```

```
4.14 µs ± 271 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[10]:
```
random_walk_faster(STEPS)
```

[10]:
```
[0, 1, 2, 3, 2, 1, 2, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 2, 3, 4]
```

File Edit View Run Kernel Settings Help Not Trusted

Code JupyterLab Python 3 (ipykernel)

[10]: `random_walk_faster(STEPS)`

[10]:
```
[0, 1, 2, 3, 2, 1, 2, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 2, 3, 4]
```

NumPy approach

[11]:
```
import numpy as np

def random_walk_fastest(n):
    # No 's' in numpy choice (Python offers choice & choices)
    steps = np.random.choice([-1,+1], n)
    return np.cumsum(steps)
```

[16]:
```
%%timeit -n 1000


walk = random_walk_fastest(STEPS)
```

```
24.8 µs ± 5.26 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[13]:
```
random_walk_fastest(STEPS)
```

[13]:
```
array([-1, -2, -1, -2, -3, -2, -3, -4, -5, -4, -3, -4, -5, -4, -5, -4, -5,
       -4, -3, -2])
```

15th Workshop on Hot Topics in Operating Systems

HotOS XV

MAY 18–20, 2015 • KARTAUSE ITTINGEN, SWITZERLAND

Sponsored by USENIX in cooperation with ACM SIGOPS

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Overview
Workshop Organizers
Workshop Program
Sponsorship
Instructions for Participants
Call for Papers
Past Workshops

Scalability! But at what COST?

Authors:

Frank McSherry; Michael Isard; Derek G. Murray

Abstract:

We offer a new metric for big data platforms, COST, or the Configuration that Outperforms a Single Thread. The COST of a given platform for a given problem is the hardware configuration required before the platform outperforms a competent single-threaded implementation. COST weighs a system's scalability against the overheads introduced by the system, and indicates the actual performance gains of the system, without rewarding systems that bring substantial but parallelizable overheads.

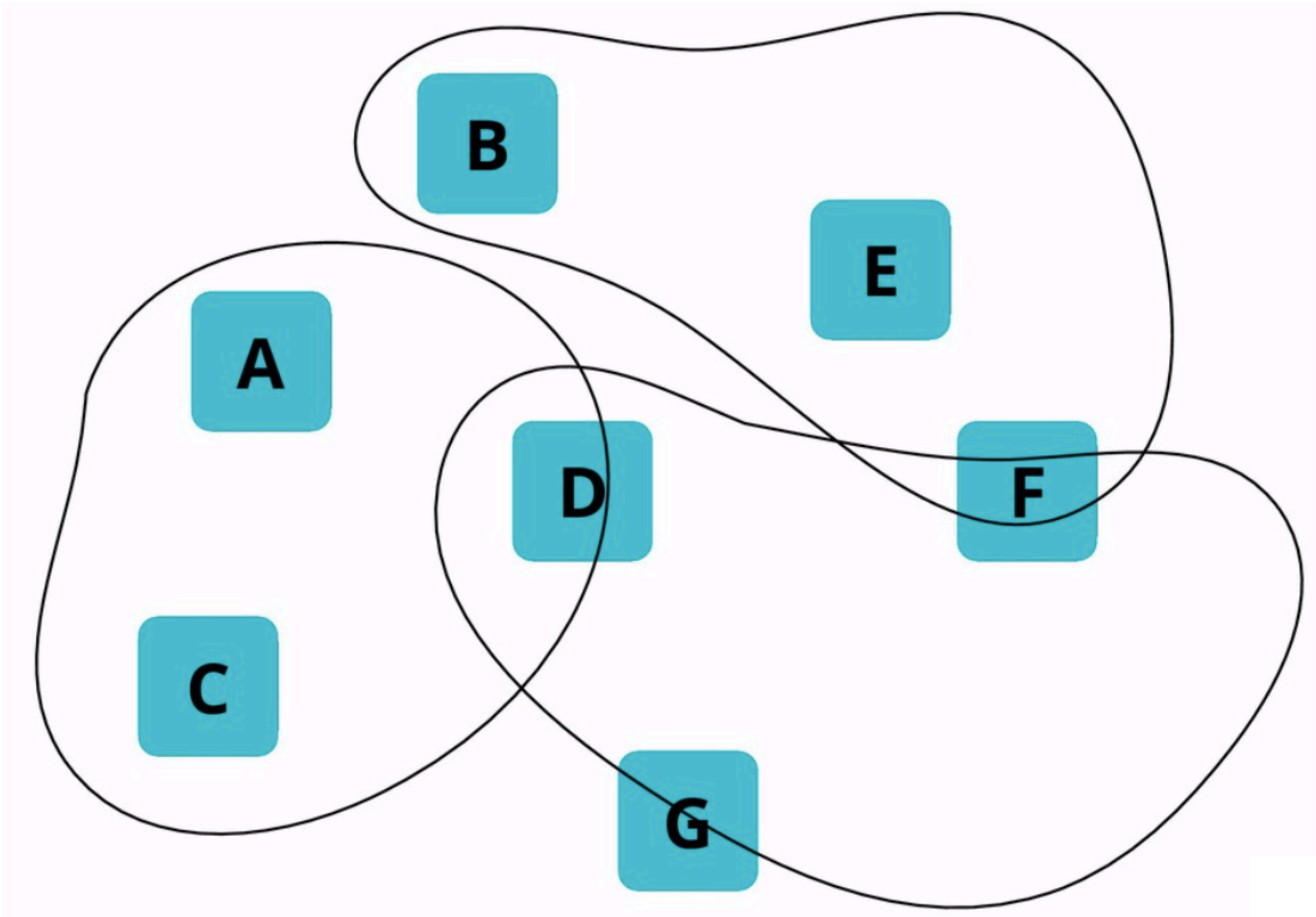
We survey measurements of data-parallel systems recently reported in SOSP and OSDI, and find that many systems have either a surprisingly large COST, often hundreds of cores, or simply underperform one thread for all of their reported configurations.

ThoughtWorks®

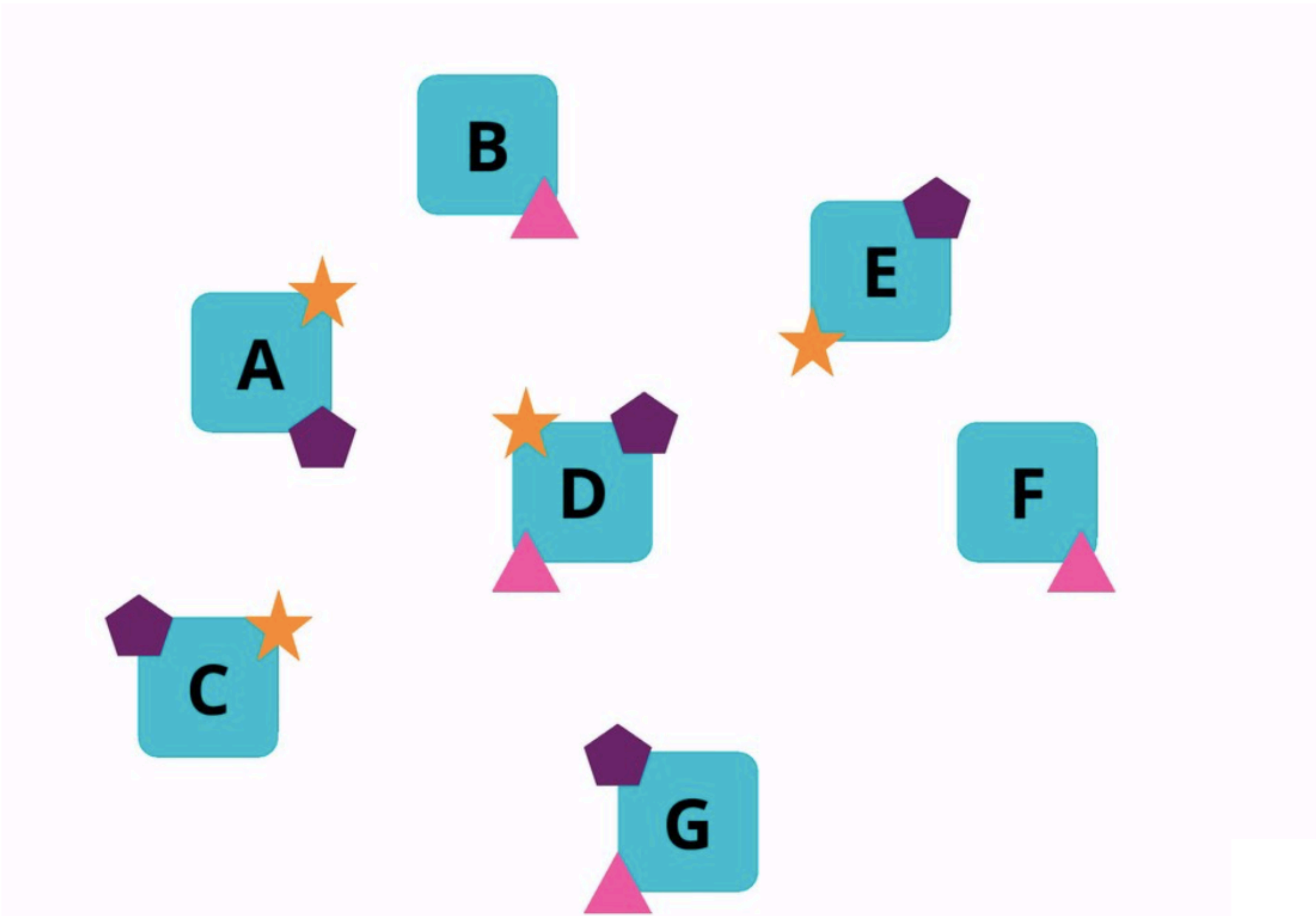
WRAPPING UP

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

THANKS !

Luciano Ramalho
lramalho@thoughtworks.com

Fediverse: *@ramgarlic@fosstodon.org*
PFKAT: *@ramalhoorg*
Repo: *github.com/ramalho/beyond-paradigms*
Slides: *speakerdeck.com/ramalho*

ThoughtWorks®