

paradigms lost, paradigms regained

*programming with objects and
functions and more*

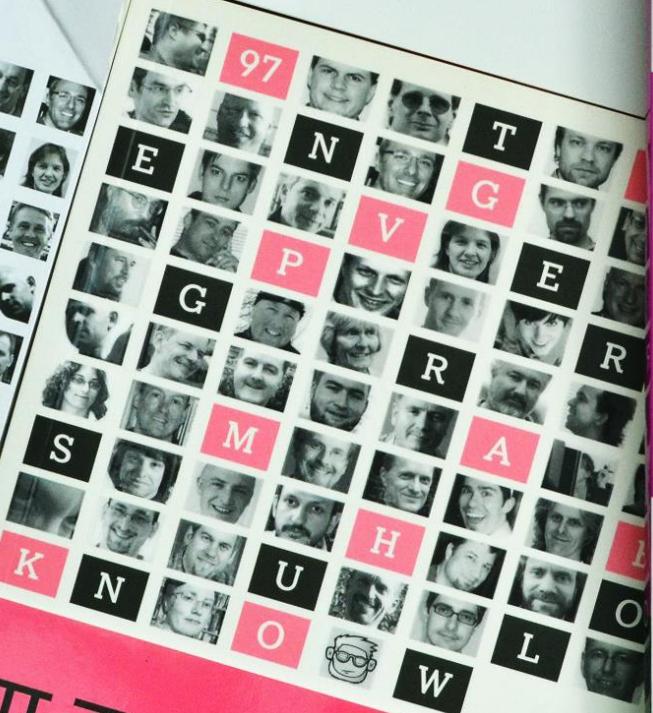
@KevlinHenney

知道的
事

Kevlin Henney 编
李军泽 吕骏 审校
电子工业出版社。
http://www.phei.com.cn

ノログ
97 Things Every Prog
知るべき

O'REILLY®
オライリー・ジャパン



프로그래머가
알아야
하는
97가지

97



Collective Wisdom
from the Experts

97 Things Every Programmer Should Know

O'REILLY®

Edited by Kevlin Henney

zkušenosti
expertů z praxe



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

A Pattern Language for
Distributed Computing



Volume 4

Frank Buschmann
Kevlin Henney
Douglas C. Schmidt



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

On Patterns and Pattern Languages



Volume 5

Frank Buschmann
Kevlin Henney
Douglas C. Schmidt

The
Timeless Way of
Building

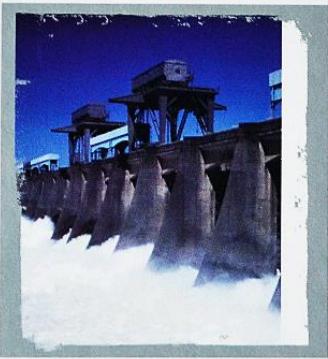


Christopher Alexander

We know that every pattern is an instruction of the general form:
context → conflicting forces → configuration

So we say that a pattern is good, whenever we can show that it meets the following two empirical conditions:

1. *The problem is real.* This means that we can express the problem as a conflict among forces which really do occur within the stated context, and cannot normally be resolved within that context. This is an empirical question.
2. *The configuration solves the problem.* This means that when the stated arrangement of parts is present in the stated context, the conflict can be resolved, without any side effects. This is an empirical question.



ARCHITECTING ENTERPRISE SOLUTIONS

**Patterns for High-Capability
Internet-Based Systems**

Paul Dyson
Andy Longshaw



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

Our position is that an architectural definition is something that answers three questions:

- **What are the structural elements of the system?**
- **How are they related to each other?**
- **What are the underlying principles and rationale that guide the answers to the previous two questions?**

Paul Dyson

Andy Longshaw

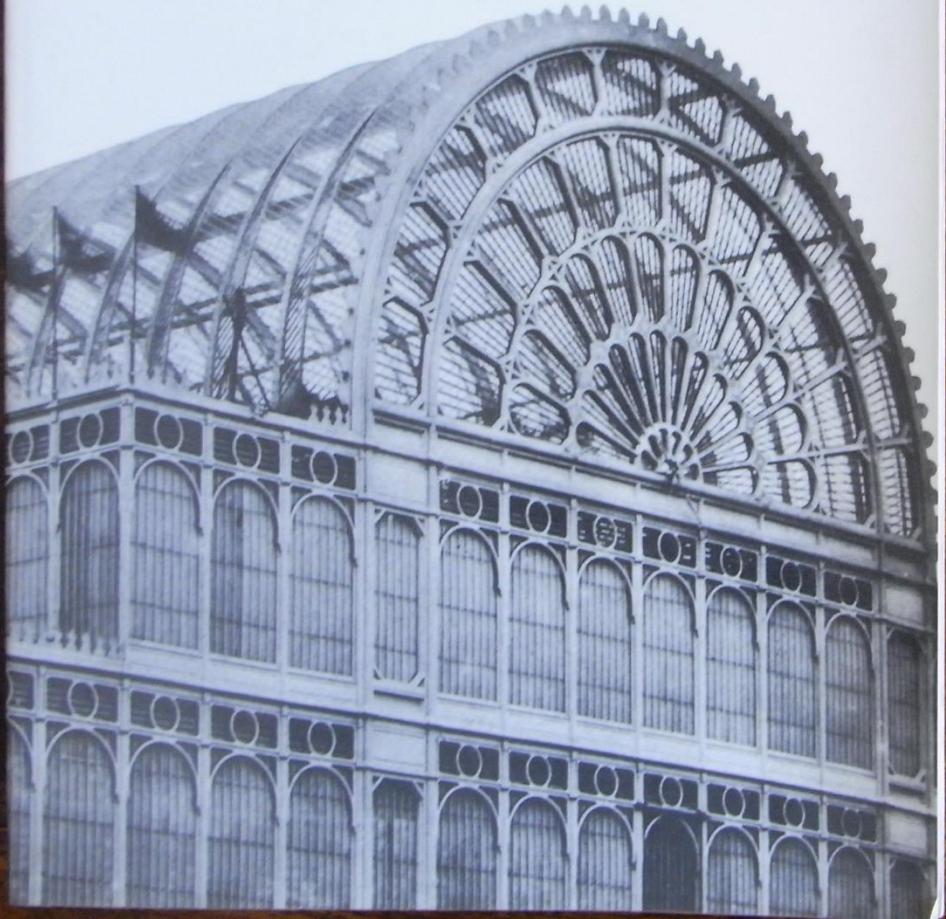


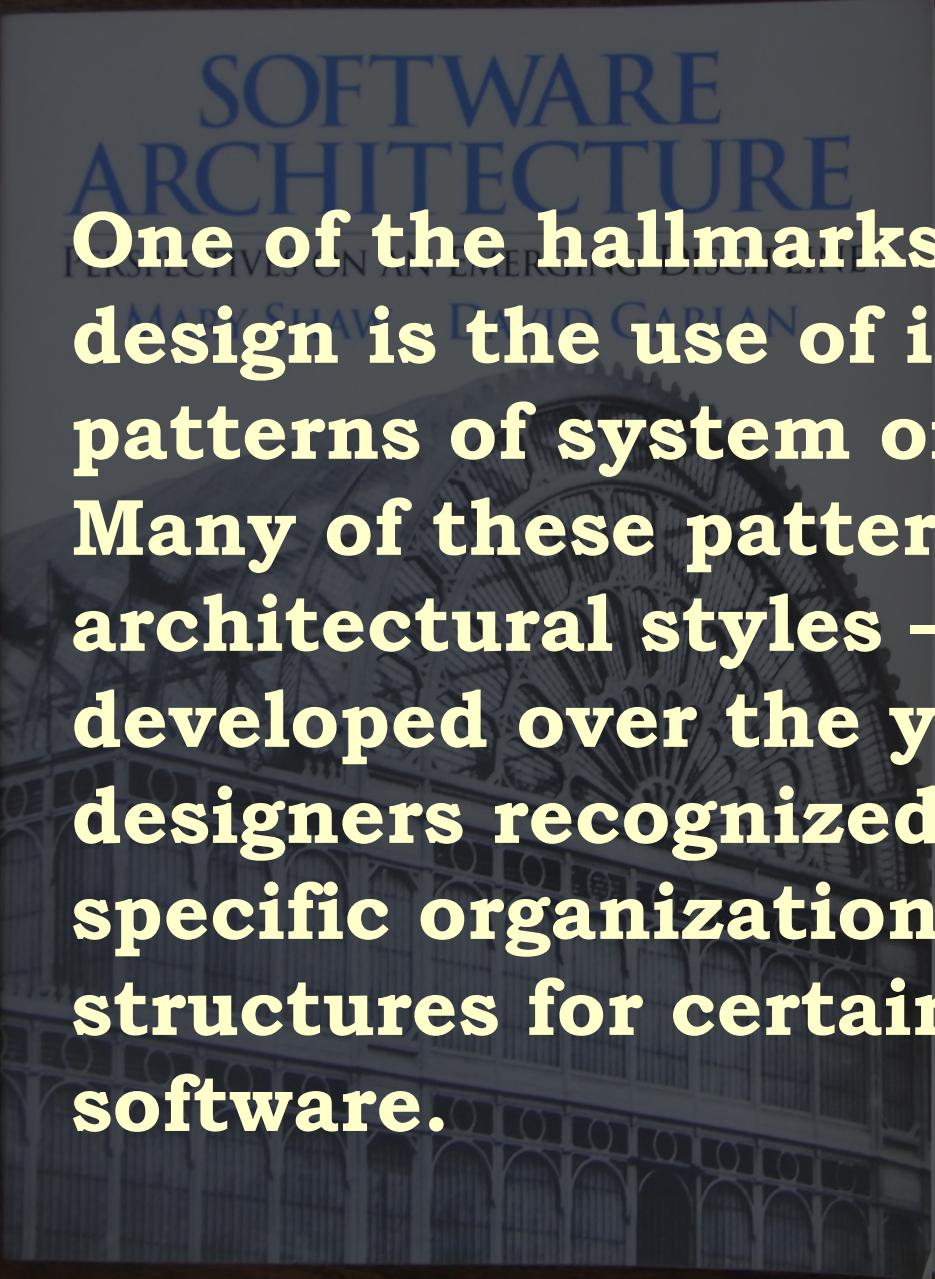
WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

SOFTWARE ARCHITECTURE

PERSPECTIVES ON AN EMERGING DISCIPLINE

MARY SHAW DAVID GARLAN





SOFTWARE ARCHITECTURE

One of the hallmarks of architectural design is the use of idiomatic patterns of system organization. Many of these patterns — or architectural styles — have been developed over the years as system designers recognized the value of specific organizational principles and structures for certain classes of software.

The Paradigms of Programming

Robert W. Floyd
Stanford University



Paradigm(pæ·radim, -dēim) . . . [a. F. *paradigme*, ad. L. *paradigma*, a. Gr. παραδειγμα pattern, example, f. παραδεικνυ·ναι to exhibit beside, show side by side. . .]
1. A pattern, exemplar, example.

1752 J. Gill *Trinity* v. 91

The archetype, paradigm, exemplar, and idea, according to which all things were made.

From the Oxford English Dictionary.

Today I want to talk about the paradigms of programming, how they affect our success as designers of computer programs, how they should be taught, and how they should be embodied in our programming languages.

A familiar example of a paradigm of programming is the technique of *structured programming*, which appears to be the dominant paradigm in most current treatments of programming methodology. Structured programming, as formulated by Dijkstra [6], Wirth [27, 29], and Parnas [21], among others, consists of two phases.

In the first phase, that of top-down design, or stepwise refinement, the problem is decomposed into a very small number of simpler subproblems. In programming the solution of simultaneous linear equations, say, the first level of decomposition would be into a stage of triangularizing the equations and a following stage of back-substitution in the triangularized system. This gradual decomposition is continued until the subproblems that arise are simple enough to cope with directly. In the simultaneous equation example, the back substitution process would be further decomposed as a backwards iteration of a process which finds and stores the value of the i th variable from the i th equation. Yet further decomposition would yield a fully detailed algorithm.

I believe that the current state of the art of computer programming reflects inadequacies in our stock of paradigms, in our knowledge of existing paradigms, in the way we teach programming paradigms, and in the way our programming languages support, or fail to support, the paradigms of their user communities.

**Rien n'est plus
dangereux qu'une
idée, quand on n'a
qu'une idée.**

Émile-Auguste Chartier

**Nothing is more
dangerous than an
idea, when you have
only one idea.**

Émile-Auguste Chartier

**Nothing is more
dangerous than an
IDE, when you have
only one IDE.**

**Nothing is more
dangerous than OO,
when you have only
one object.**



Share a Coke® with

Kevlin

FROME VALLEY
HEREFORDSHIRE

• HENNEY'S •
DRY CIDER

MADE FROM 100% FRESH PRESSED JUICE

THE SINGLETON
AGED 12 YEARS
OF DUFFTOWN

THE SINGLETON

Single Malt Scotch Whisky
of Dufftown

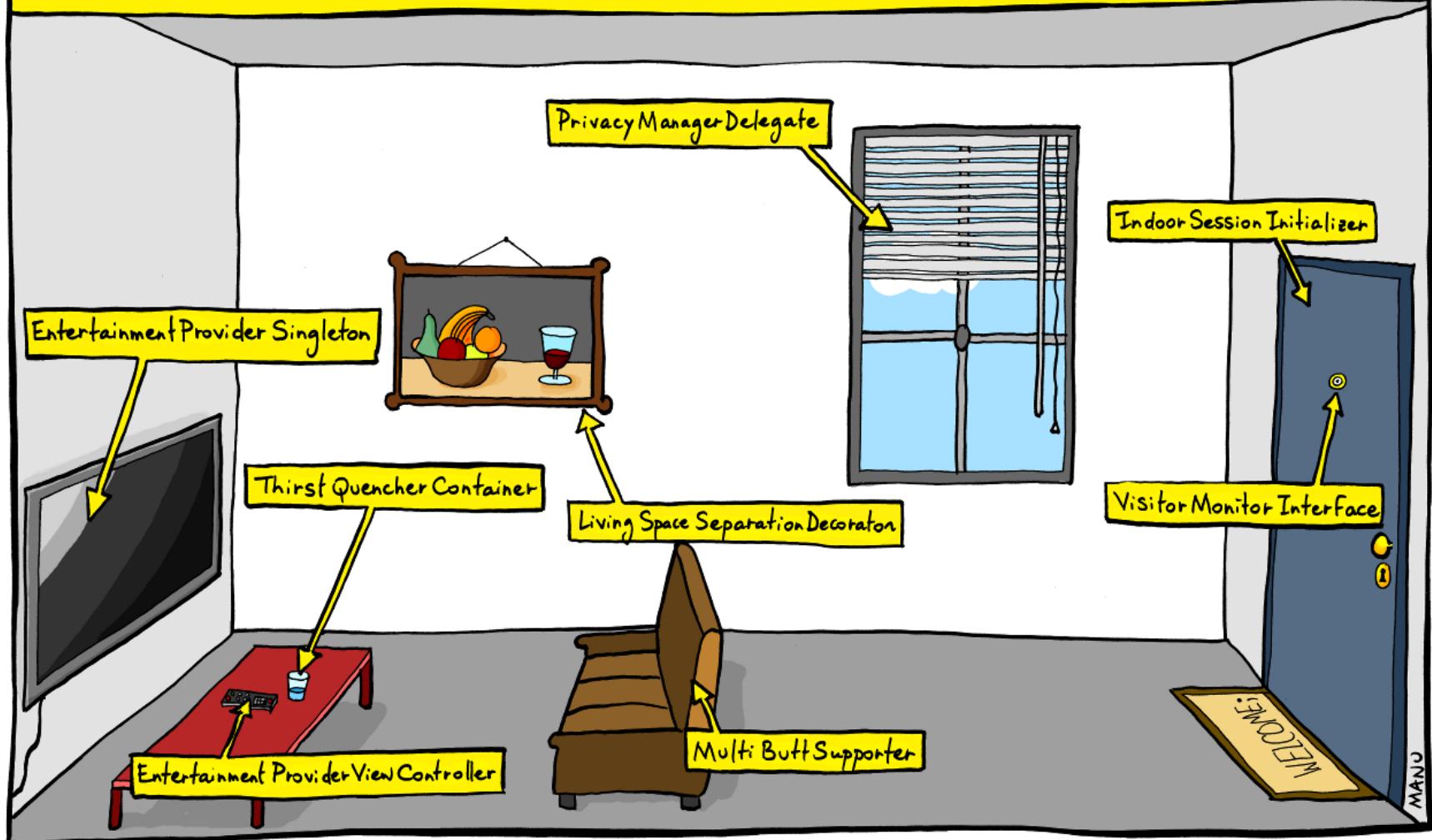
DUFFTOWN
STD 1896

AGED FOR
12 YEARS

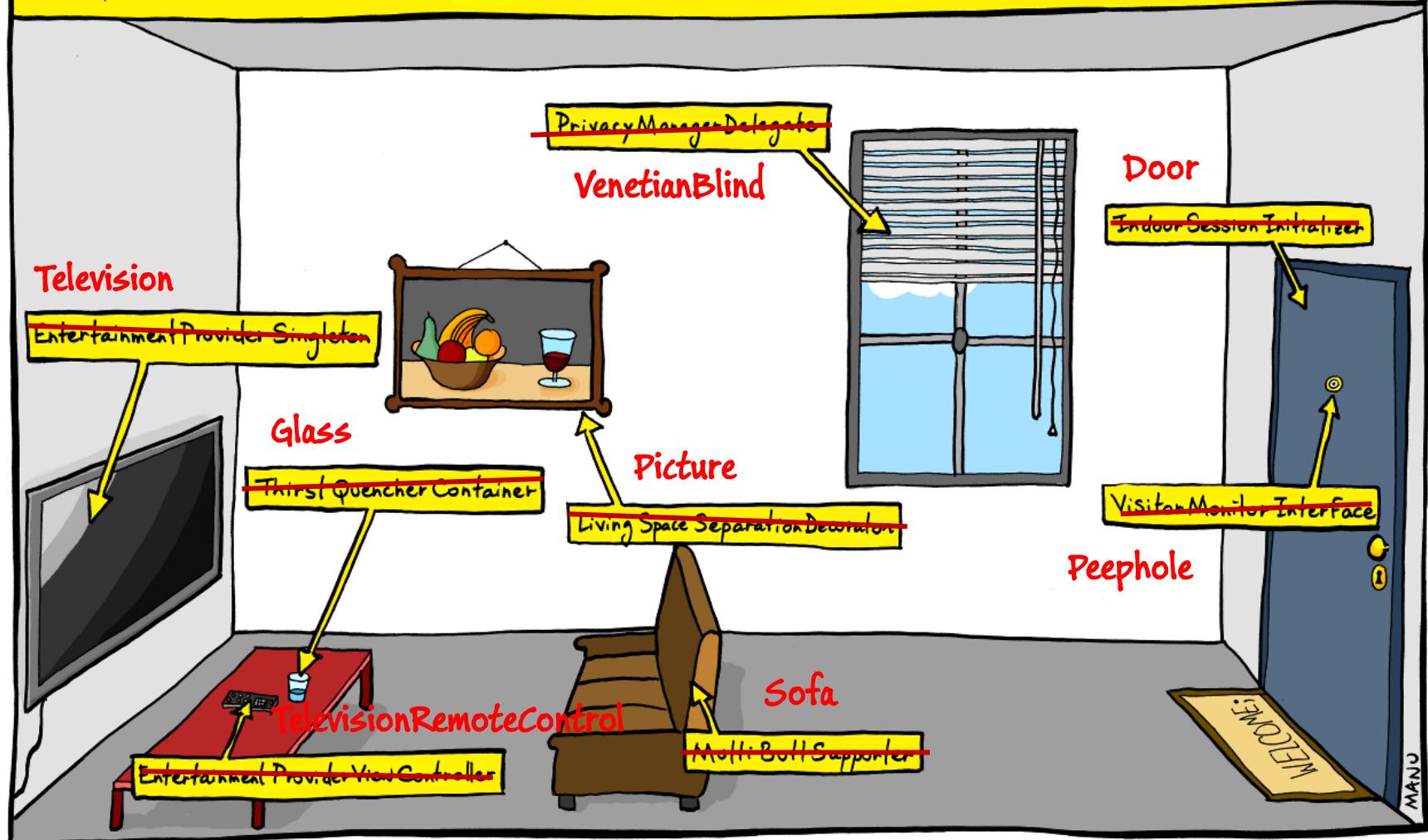
12

PRODUCT OF
SCOTLAND

THE WORLD SEEN BY AN "OBJECT-ORIENTED" PROGRAMMER.



THE WORLD SEEN BY AN "~~OBJECT ORIENTED~~" PROGRAMMER.





Kevlin Henney

@KevlinHenney



Follow

Having experienced Enterprisation of OO — large classes named as Manager, Controller, Object, etc. — now dreading Enterprisation of FP.

6:42 AM - 12 Jun 2014

9 RETWEETS 6 FAVORITES



<https://twitter.com/KevlinHenney/status/476962681636020224>



Kevlin Henney

@KevlinHenney

Follow

Having experienced Enterprisation of OO — large classes named as Manager, Controller, Object, etc. — now dreading Enterprisation of FP.

6:42 AM - 12 Jun 2014

9 RETWEETS 6 FAVORITES



Kevlin Henney

@KevlinHenney

Follow

Enterprise FP: large **functions** with Function, Transform, Process, Calculate, etc. in their names... and lots of dependencies and side effects.

6:43 AM - 12 Jun 2014

7 RETWEETS 3 FAVORITES



Agglutination is a process in linguistic morphology derivation in which complex words are formed by stringing together morphemes, each with a single grammatical or semantic meaning. Languages that use agglutination widely are called agglutinative languages.

hippopotomonstrosesquipedaliophobia

pneumonoultramicroscopicsilicovolcanoconiosis

fylkestrafikk sikkerhetsutvalgs sekretariatsleder funksjonene

Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz

muvaaffakiyetsizleştiricileştiriveremeyebileceklerimizdenmissinizcesine

Classnamer

Can't think of a good class name? Try this:

TemporaryLolcatSelector

generators: [generic](#), [Spring](#)
[text only](#)
[about](#)

Method Namer

Can't think of a good method name? Try this:

abstractCustomerSingleton

My First Method Naming style ▾

My First Method Naming style

Spring Framework style

JDK style

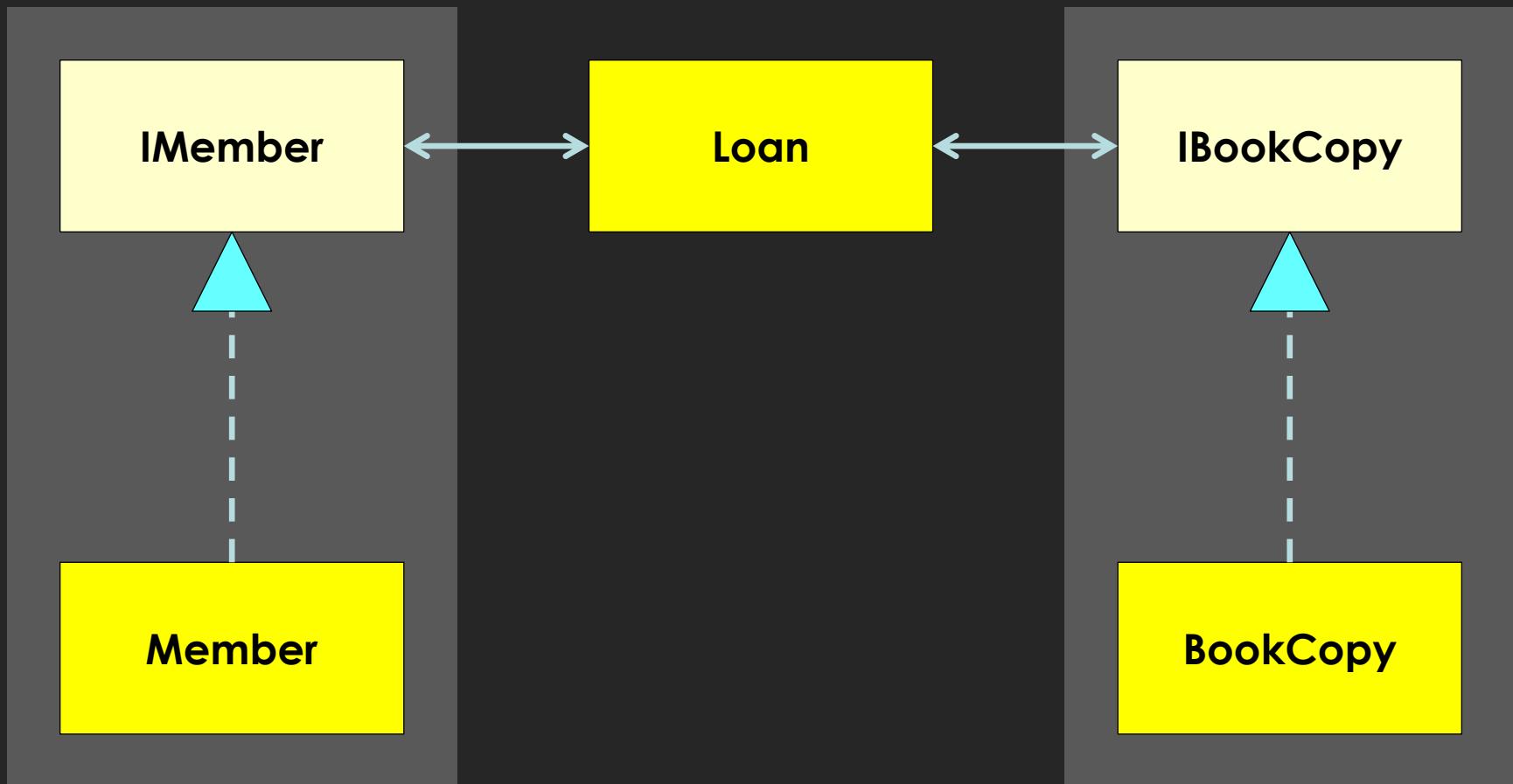
Generate Name

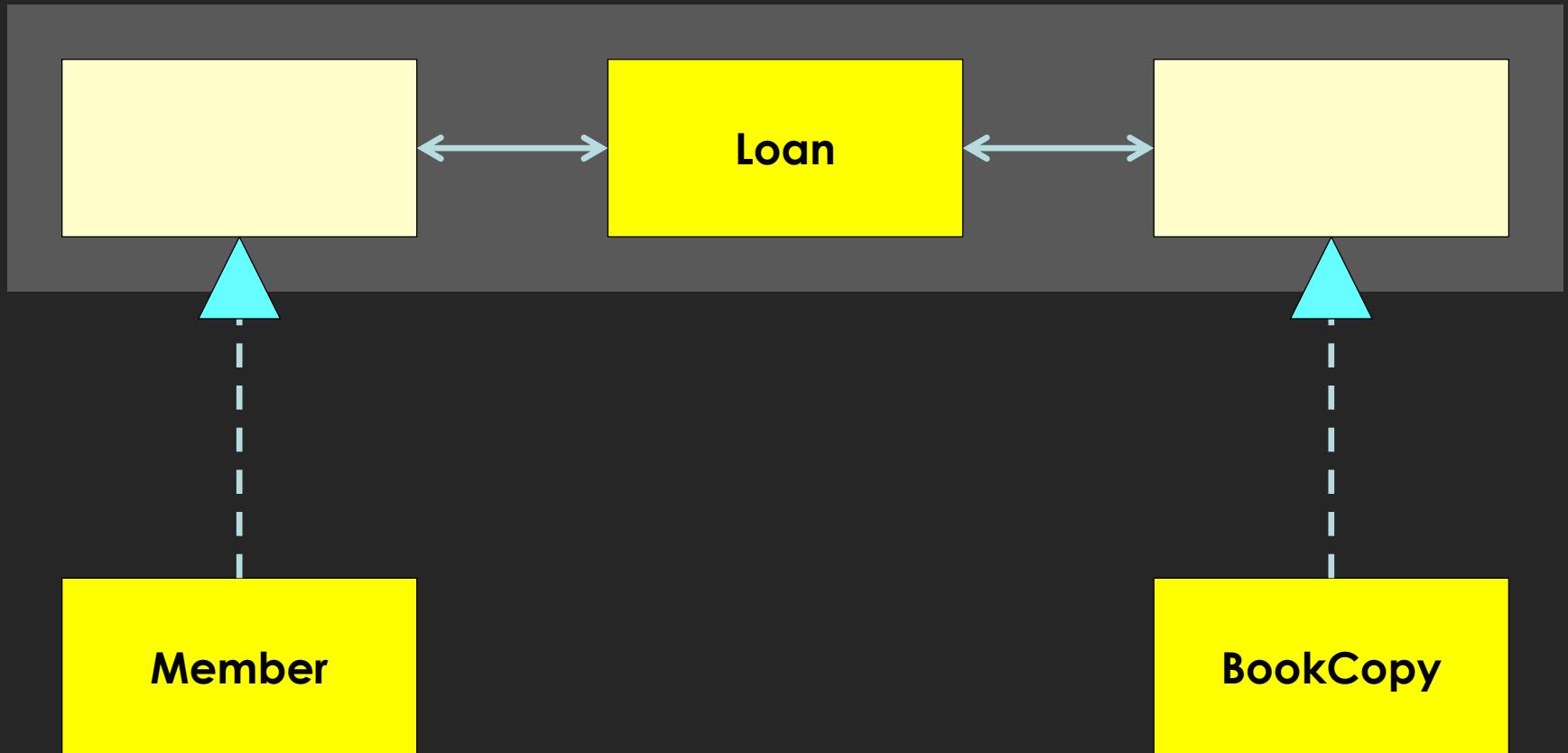
Idea inspired by [classnameser.com](#) and a tweet from [Michael Feathers](#). Contact me at [my blog](#)

managerialism, noun

- belief in or reliance on the use of professional managers in administering or planning an activity
- application of managerial techniques of businesses to the running of other organisations, such as the civil service or local authorities
- belief in the importance of tightly managed organisations, as opposed to individuals, or groups that do not resemble an organisation

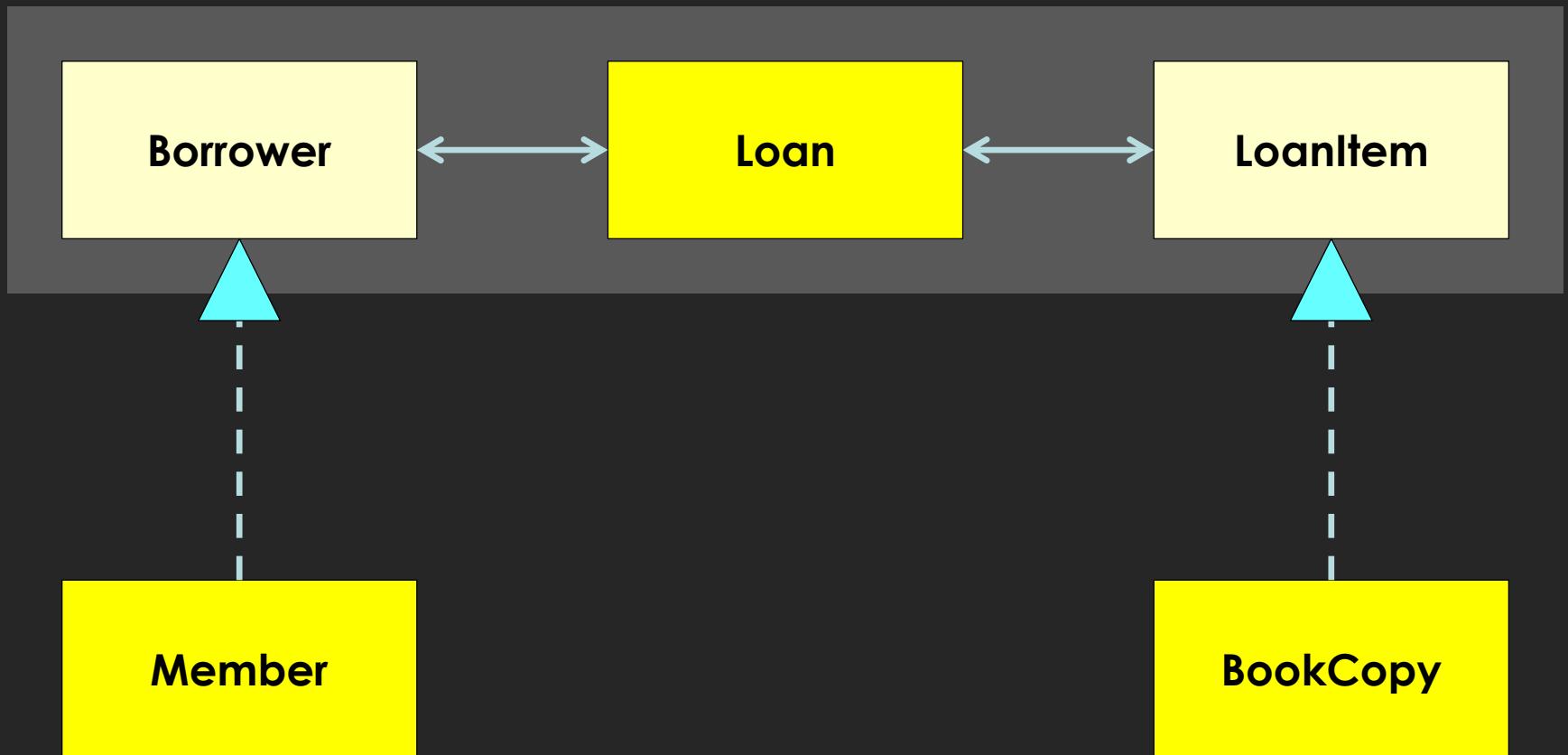






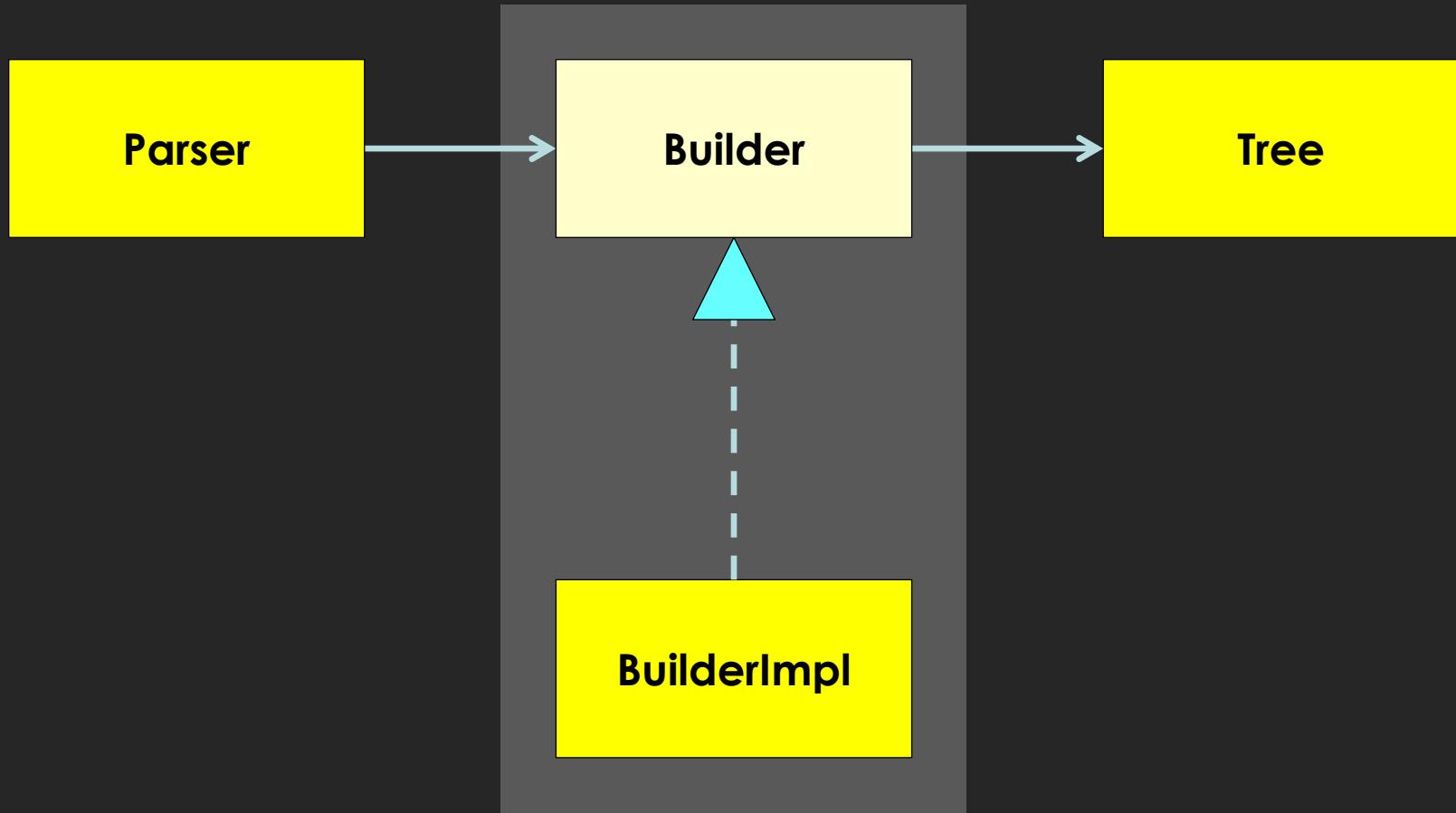
role, noun

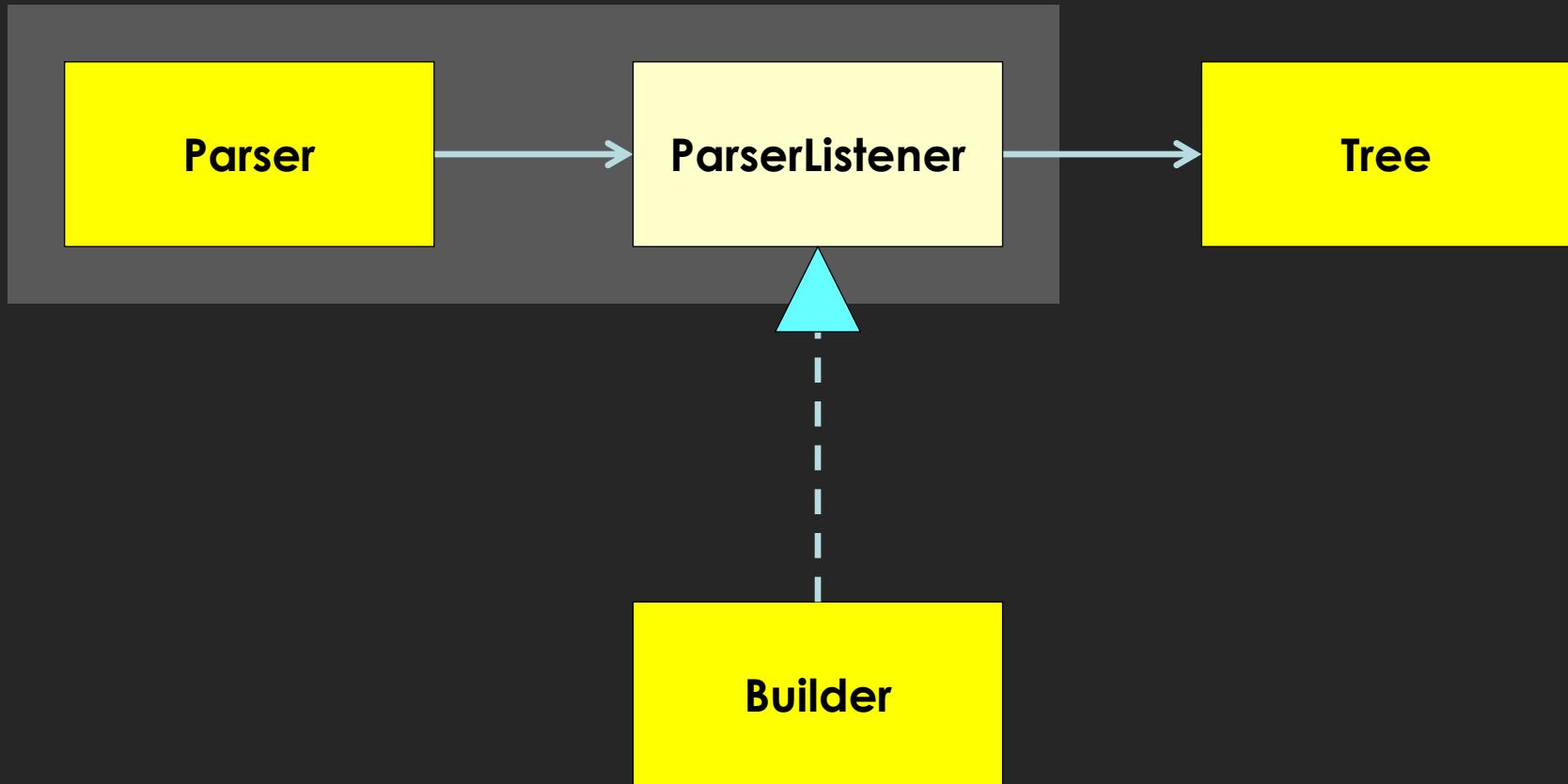
- an actor's part in a play, film, etc.
- a person or thing's function in a particular situation
- a function, part or expected behaviour performed in a particular operation or process

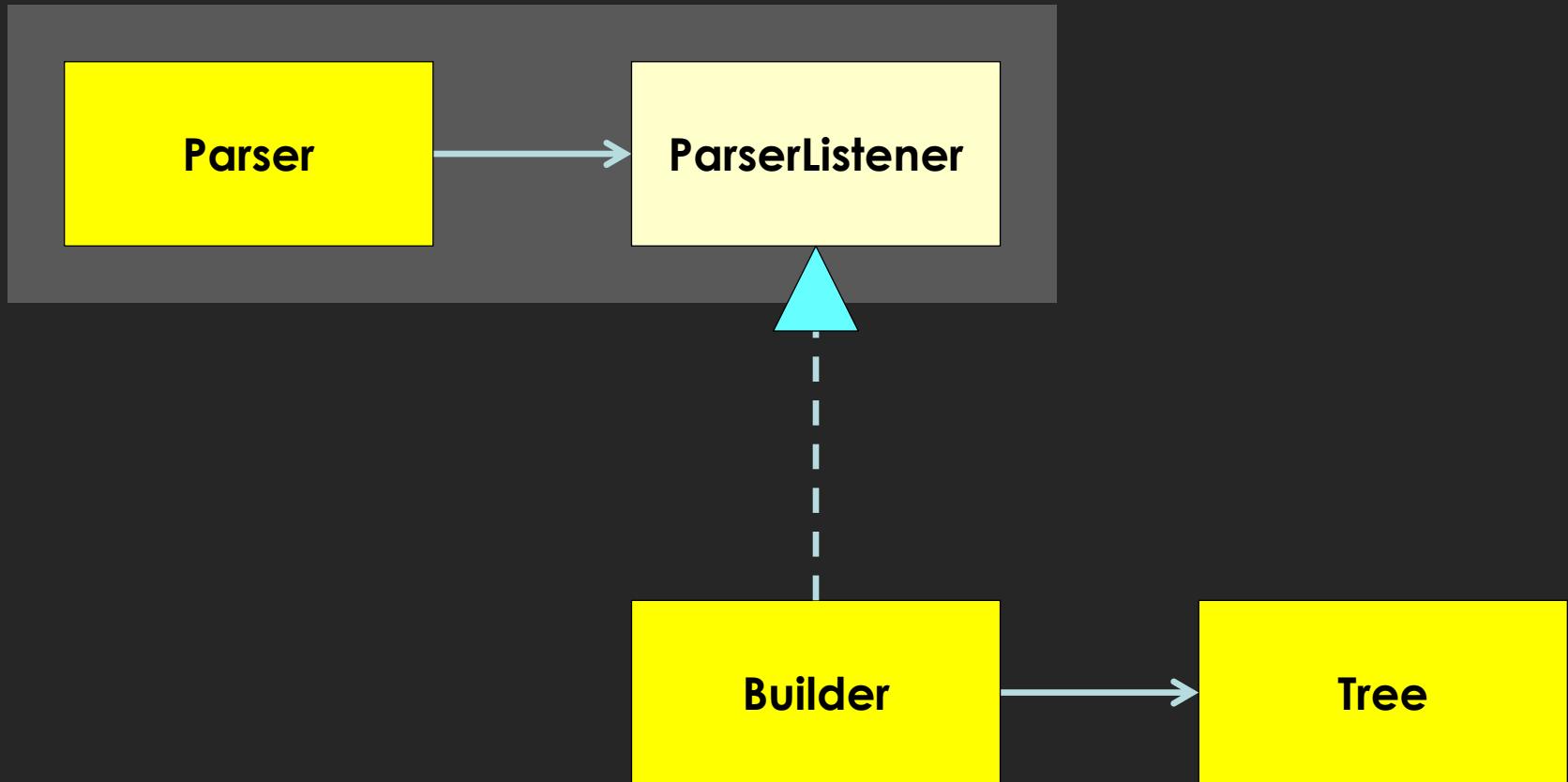


interface









Encapsulation

Inheritance

Polymorphism

Encapsulation

Polymorphism

Inheritance

Encapsulation

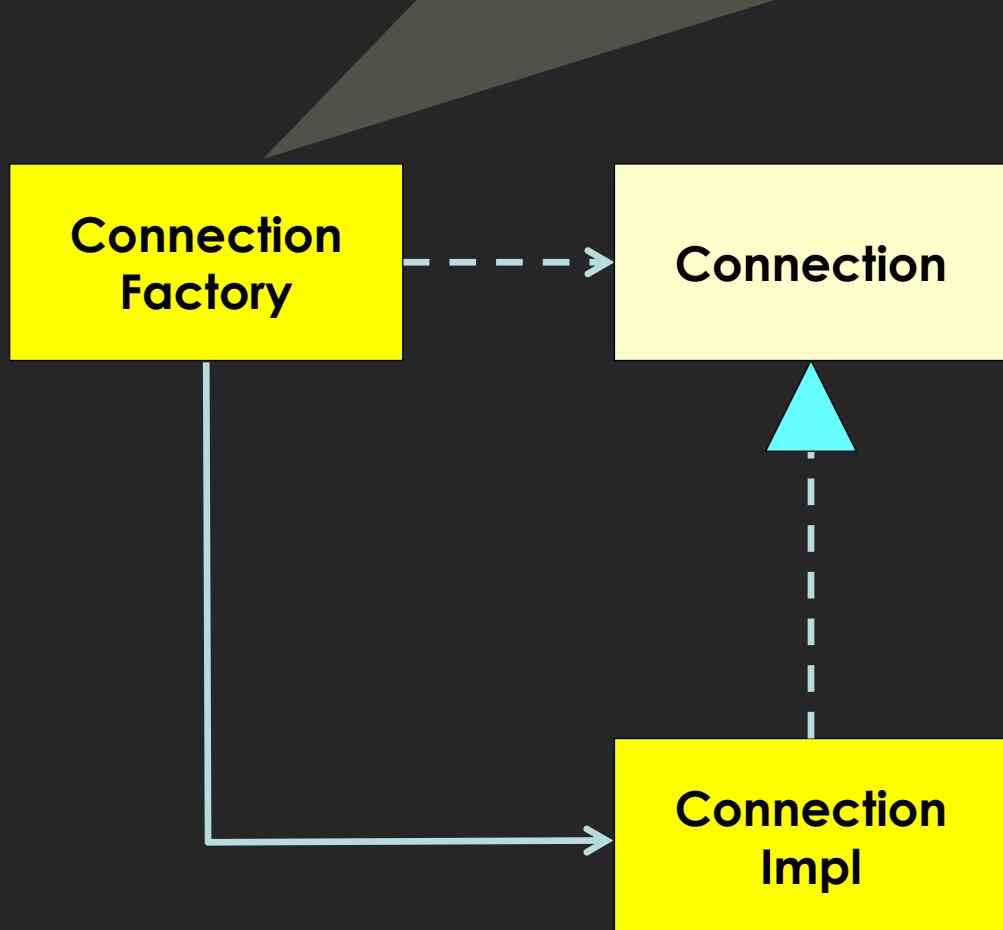
Polymorphism

Inheritance

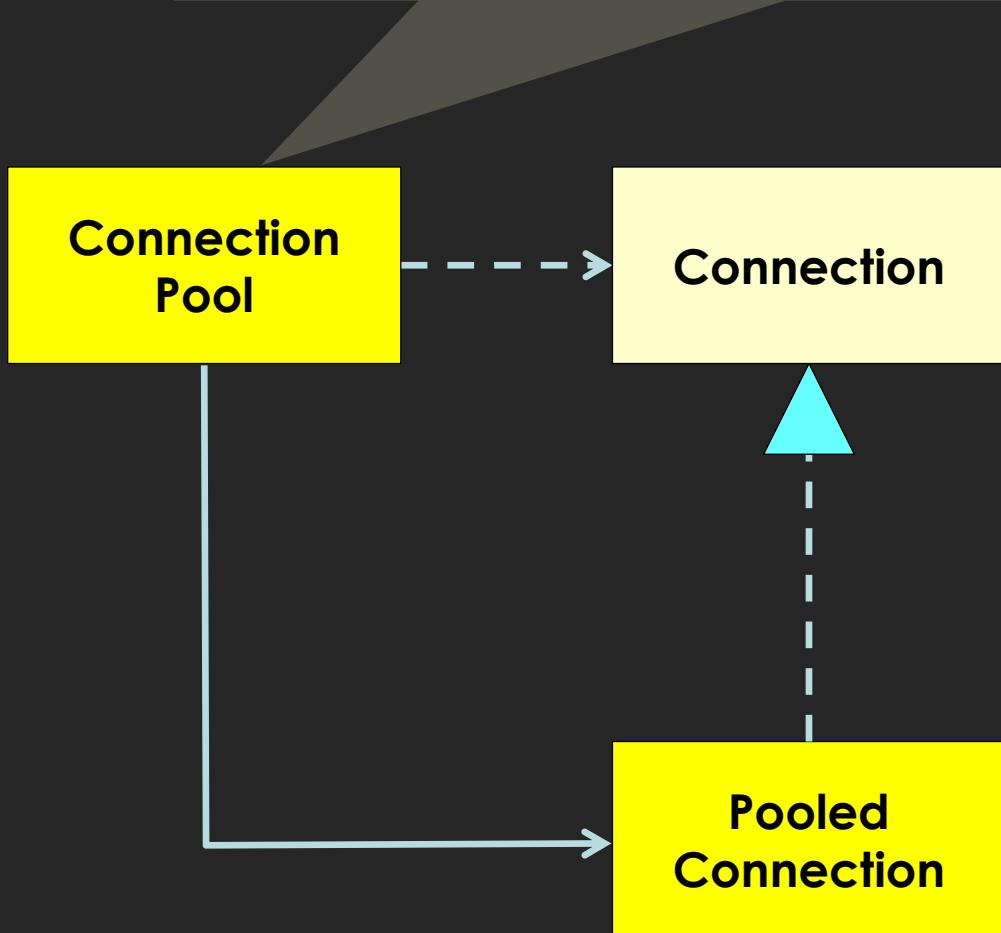
It is possible to
do Object-Oriented
programming in
Java

Object-Oriented
subset of Java:
class name is
only after “new”

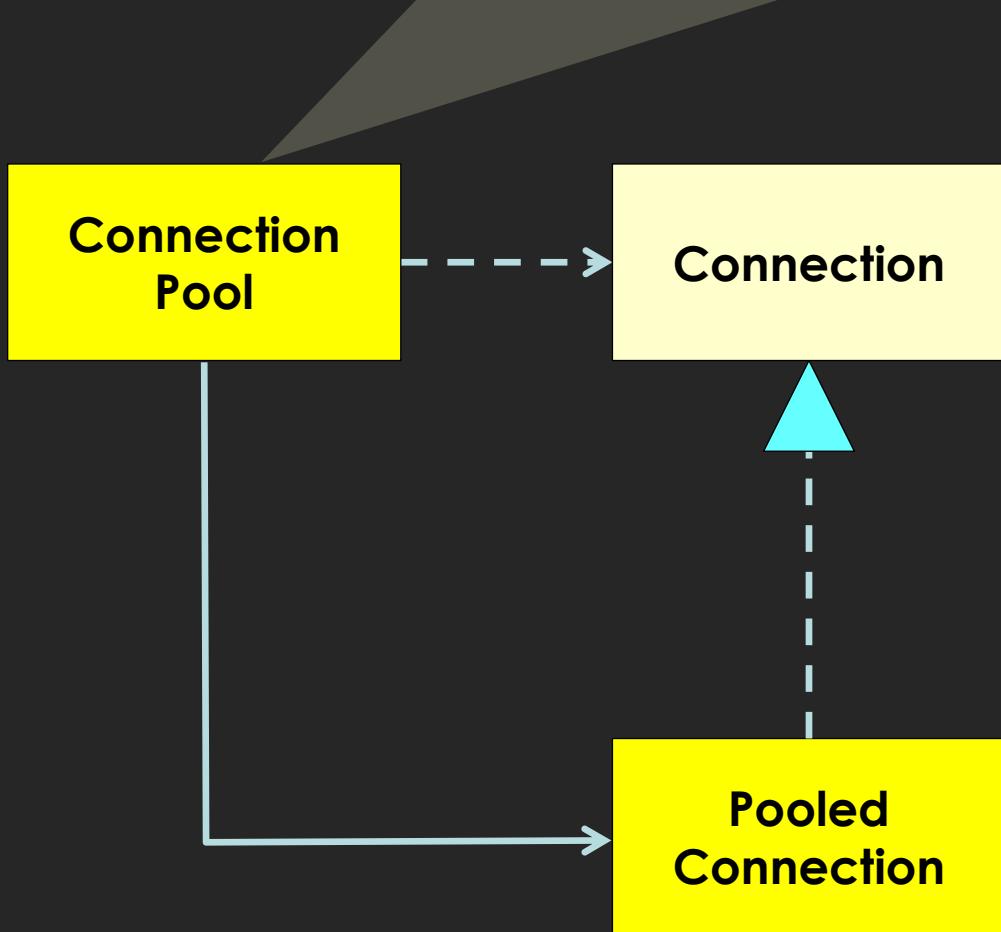
```
public Connection createConnection(Provider provider)  
throws ConnectionFailureException
```



```
public Connection connectTo(Provider ofUpdates)  
throws ConnectionFailure
```



```
public Connection connectTo(Provider ofUpdates)
```



name

choose the right

specific

get

eliminate

words

Naomi Epel

The Observation Deck

STRUCTURED PROGRAMMING

O.-J. DAHL, E. W. DIJKSTRA
and C. A. R. HOARE

One of the most powerful mechanisms for program structuring [...] is the block and procedure concept.

A procedure which is capable of giving rise to block instances which survive its call will be known as a class; and the instances will be known as objects of that class.

A call of a class generates a new object of that class.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"

Lambda-calculus
was the first
object-oriented
language (1941)

```
newRecentlyUsedList =  
λ • (let items = ref(⟨⟩) •  
{  
    isEmpty = λ • #items = 0,  
    size = λ • #items,  
    get = λ i • itemsi,  
    add = λ x •  
        items := ⟨x⟩^⟨get(i) | i ∈ 0...size() ∧ get(i) ≠ x⟩  
})
```

```
var newRecentlyUsedList = function() {
    var items = []
    return {
        isEmpty: function() {
            return items.length === 0
        },
        size: function() {
            return items.length
        },
        get: function(index) {
            return items[index]
        },
        add: function(newItem) {
            (items = items.filter(function(item) {
                return item !== newItem
            })).unshift(newItem)
        }
    }
}
```

```
var newRecentlyUsedList = () => {
  var items = []
  return {
    isEmpty: () => items.length === 0,
    size: () => items.length,
    get: index => items[index],
    add: newItem =>
      (items = items.filter(item => item !== newItem) )
      .unshift(newItem)
  }
}
```

```
public interface RecentlyUsedList
{
    boolean isEmpty();
    int size();
    String get(int index);
    void add(String toAdd);
}
```

```
Supplier<RecentlyUsedList> newRecentlyUsedList = () ->
{
    List<String> items = new ArrayList<String>();
    return new RecentlyUsedList()
    {
        public boolean isEmpty()
        {
            return items.isEmpty();
        }
        public int size()
        {
            return items.size();
        }
        public String get(int index)
        {
            return items.get(size() - index - 1);
        }
        public void add(String toAdd)
        {
            items.remove(toAdd);
            items.add(toAdd);
        }
    };
};
```

Package java.util.function

Functional interfaces provide target types for lambda expressions and method references.

See: [Description](#)

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type.
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	Represents a supplier of <code>boolean</code> -valued results.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.
DoubleBinaryOperator	Represents an operation upon two <code>double</code> -valued operands and producing a <code>double</code> -valued result.
DoubleConsumer	Represents an operation that accepts a single <code>double</code> -valued argument and returns no result.
DoubleFunction<R>	Represents a function that accepts a <code>double</code> -valued argument and produces a result.
DoublePredicate	Represents a predicate (boolean-valued function) of one <code>double</code> -valued argument.
DoubleSupplier	Represents a supplier of <code>double</code> -valued results.
DoubleToIntFunction	Represents a function that accepts a <code>double</code> -valued argument and produces an <code>int</code> -valued result.
DoubleToLongFunction	Represents a function that accepts a <code>double</code> -valued argument and produces a <code>long</code> -valued result.
DoubleUnaryOperator	Represents an operation on a single <code>double</code> -valued operand that produces a <code>double</code> -valued result.
Function<T,R>	Represents a function that accepts one argument and produces a result.
IntBinaryOperator	Represents an operation upon two <code>int</code> -valued operands and producing an <code>int</code> -valued result.
IntConsumer	Represents an operation that accepts a single <code>int</code> -valued argument and returns no result.

Package Java.util.function

Functional interfaces provide target types for lambda expressions and method references.

See: Description

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BinaryOperator<T>	Represents an operation that takes two operands of the same type, producing a result of the same type as the operands.
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	Represents a supplier of boolean-valued results.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.
DoubleBinaryOperator	Represents an operation upon double-valued operands and producing a double-valued result.
DoubleConsumer	Represents an operation that accepts a single double-valued argument and returns no result.
DoubleFunction<R>	Represents a function that accepts a double-valued argument and produces a result.
DoublePredicate	Represents a predicate (boolean-valued function) of one double-valued argument.
DoubleSupplier	Represents a supplier of double-valued results.
DoubleToIntFunction	Represents a function that accepts a double-valued argument and produces an int-valued result.
DoubleToLongFunction	Represents a function that accepts a double-valued argument and produces a long-valued result.
DoubleUnaryOperator	Represents an operation that takes a double-valued operand that produces a double-valued result.
Function<T,R>	Represents a function that accepts one argument and produces a result.
IntBinaryOperator	Represents an operation on two int-valued operands and producing an int-valued result.
IntConsumer	Represents an operation that accepts a single int-valued argument and returns no result.
IntFunction<R>	Represents a function that accepts an int-valued argument and produces a result.
IntPredicate	Represents a predicate (boolean-valued function) of one int-valued argument.
IntSupplier	Represents a supplier of int-valued results.
IntToDoubleFunction	Represents a function that accepts an int-valued argument and produces a double-valued result.
IntToLongFunction	Represents a function that accepts an int-valued argument and produces a long-valued result.
IntUnaryOperator	Represents an operation on a single int-valued operand that produces an int-valued result.
LongBinaryOperator	Represents an operation on two long-valued operands that produces a long-valued result.
LongConsumer	Represents an operation that accepts a single long-valued argument and returns no result.
LongFunction<R>	Represents a function that accepts a long-valued argument and produces a result.
LongPredicate	Represents a predicate (boolean-valued function) of one long-valued argument.
LongSupplier	Represents a supplier of long-valued results.
LongToDoubleFunction	Represents a function that accepts a long-valued argument and produces a double-valued result.
LongToIntFunction	Represents a function that accepts a long-valued argument and produces an int-valued result.
LongUnaryOperator	Represents an operation on a single long-valued operand that produces a long-valued result.
ObjDoubleConsumer<T>	Represents an operation that accepts an object-valued and a double-valued argument, and returns no result.
ObjIntConsumer<T>	Represents an operation that accepts an object-valued and a int-valued argument, and returns no result.
ObjLongConsumer<T>	Represents an operation that accepts an object-valued and a long-valued argument, and returns no result.
Predicate<T>	Represents a predicate (boolean-valued function) of one argument.
Supplier<T>	Represents a supplier of results.
ToDoubleBiFunction<T,U>	Represents a function that accepts two arguments and produces a double-valued result.
ToDoubleFunction<T>	Represents a function that produces a double-valued result.
ToIntBiFunction<T,U>	Represents a function that accepts two arguments and produces an int-valued result.
ToIntFunction<T>	Represents a function that produces an int-valued result.
ToLongBiFunction<T,U>	Represents a function that accepts two arguments and produces a long-valued result.
ToLongFunction<T>	Represents a function that produces a long-valued result.
UnaryOperator<T>	Represents an operation on a single operand that produces a result of the same type as its operand.

apply
test
accept
get

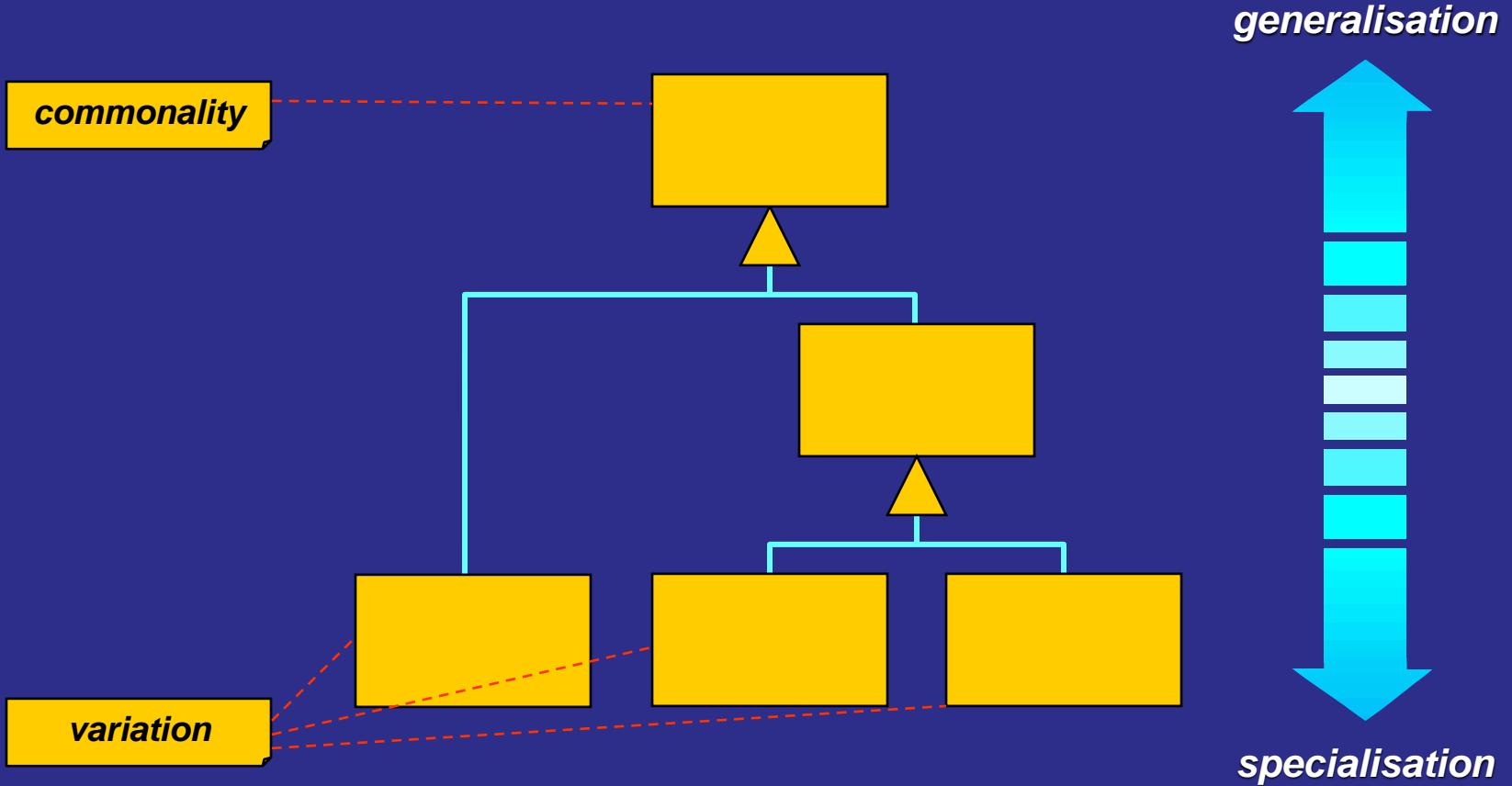
[] () { }

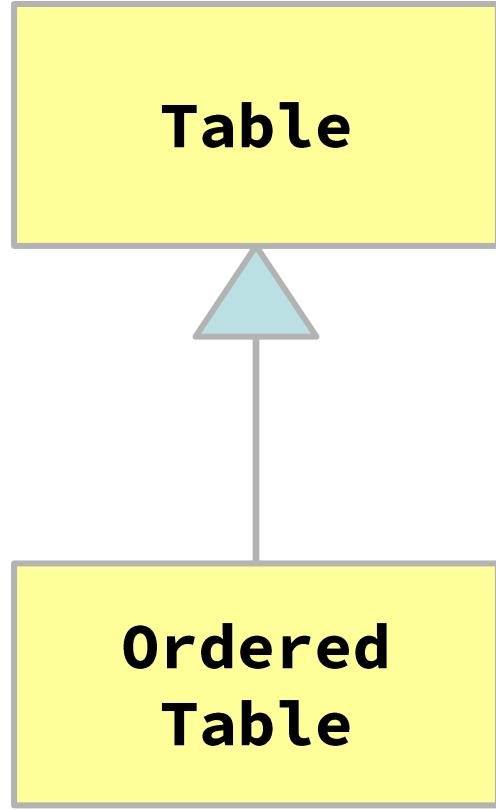
[] () { } ()

Concept Hierarchies

The construction principle involved is best called *abstraction*; we concentrate on features common to many phenomena, and we abstract *away* features too far removed from the conceptual level at which we are working.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"





E.g., Python's *dict* class, which is a
hashed mapping container

E.g., Python's *OrderedDict* class,
which preserves order of insertion

**Unordered
Table**



**Ordered
Table**

Table



**Unordered
Table**

**Ordered
Table**

**Sorted
Table**

lgr

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .

Barbara Liskov
"Data Abstraction and Hierarchy"

```
public class Ellipse
{
    private double semiMajor, semiMinor;
    public Ellipse(double a, double b) ...
    public double semiMajorAxis() ...
    public double semiMinorAxis() ...
    public void semiMajorAxis(double a) ...
    public void semiMinorAxis(double b) ...
    ...
}

public class Circle extends Ellipse
{
    public Circle(double r) ...
    public double radius() ...
    public void radius(double r) ...
    ...
}
```

```
public class Ellipse
{
    ...
    public void semiMajorAxis(double a) ...
    public void semiMinorAxis(double b) ...
    ...
}

public class Circle extends Ellipse
{
    ...
    @Override
    public void semiMajorAxis(double a)
    {
        throw new UnsupportedOperationException();
    }
    @Override
    public void semiMinorAxis(double b) ...
    ...
}
```

The reason a solution is so hard to come by is because the problem is poorly stated: mathematics tells us that a circle is an ellipse, so I can substitute a circle wherever an ellipse is required, suggesting that a circle is a subtype of an ellipse.

Kevlin Henney

"Vicious Circles", *Overload 8*, June 1995

<http://accu.org/var/uploads/journals/Overload08.pdf>

The reason a solution is so hard to come by is because the problem is poorly stated: mathematics tells us that a circle is an ellipse, so I can substitute a circle wherever an ellipse is required, suggesting that a circle is a subtype of an ellipse.

The troubles start when we introduce any state modifying functions, such as assignment or the ability to change the major and minor axes independently.

Kevlin Henney

"Vicious Circles", *Overload 8*, June 1995
<http://accu.org/var/uploads/journals/Overload08.pdf>

The reason a solution is so hard to come by is because the problem is poorly stated: mathematics tells us that a circle is an ellipse, so I can substitute a circle wherever an ellipse is required, suggesting that a circle is a subtype of an ellipse.

The troubles start when we introduce any state modifying functions, such as assignment or the ability to change the major and minor axes independently.

We are so confident that we understand the mathematical concepts behind circles and ellipses that we have not bothered to ask any more questions of that domain.

Kevlin Henney

"Vicious Circles", *Overload 8*, June 1995

<http://accu.org/var/uploads/journals/Overload08.pdf>

The first observation is that there is no way to change circles and ellipses once you have created them.

Kevlin Henney

"Vicious Circles", *Overload 8*, June 1995

<http://accu.org/var/uploads/journals/Overload08.pdf>

The first observation is that there is no way to change circles and ellipses once you have created them.

This is the correct mathematical model: there are no side effects in maths, conic sections do not undergo state changes, and there are no variables in the programming sense of the word.

Kevlin Henney

"Vicious Circles", *Overload 8*, June 1995

<http://accu.org/var/uploads/journals/Overload08.pdf>

The first observation is that there is no way to change circles and ellipses once you have created them.

This is the correct mathematical model: there are no side effects in maths, conic sections do not undergo state changes, and there are no variables in the programming sense of the word.

Readers who are comfortable and familiar with functional programming and data flow models will recognise the approach.

Kevlin Henney

"Vicious Circles", *Overload 8*, June 1995

<http://accu.org/var/uploads/journals/Overload08.pdf>

The first observation is that there is no way to change circles and ellipses once you have created them.

This is the correct mathematical model: there are no side effects in maths, conic sections do not undergo state changes, and there are no variables in the programming sense of the word.

Readers who are comfortable and familiar with functional programming and data flow models will recognise the approach.

In the case of circles and ellipses, the circle is simply an ellipse with specialised invariants. There is no additional state and none of the members of an ellipse need overriding as they apply equally well to a circle.

Kevlin Henney

"Vicious Circles", *Overload 8*, June 1995

<http://accu.org/var/uploads/journals/Overload08.pdf>

```
public class Ellipse
{
    private double semiMajor, semiMinor;
    public Ellipse(double a, double b) ...
    public double semiMajorAxis() ...
    public double semiMinorAxis() ...
    ...
}
```

```
public class Circle extends Ellipse
{
    public Circle(double r) ...
    public double radius() ...
    ...
}
```

Pure Interface Layer

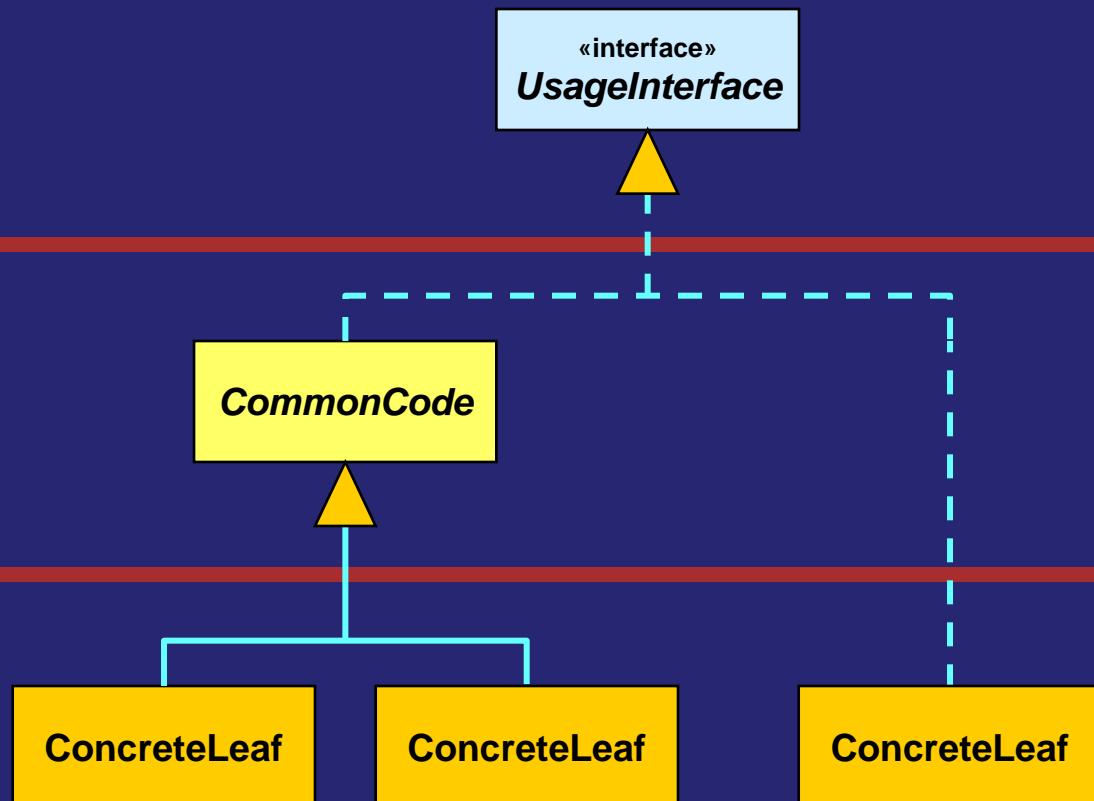
Interfaces may extend interfaces, but there is no implementation defined in this layer.

Common Code Layer

Only abstract classes are defined in this layer, possibly with inheritance, factoring out any common implementation.

Concrete Class Layer

Only concrete classes are defined, and they do not inherit from one another.



```
public interface Ellipse
{
    double semiMajorAxis();
    double semiMinorAxis();
    ...
}
```

```
public interface Circle extends Ellipse
{
    double radius();
    ...
}
```

```
public class ??? implements Ellipse
{
    private double semiMajorAxis, semiMinorAxis;
    ...
}
```

```
public class ??? implements Circle
{
    private double radius;
    ...
}
```

```
public class ??? implements Ellipse  
{  
    ...  
}
```

*The Naming of Cats is a difficult matter,
It isn't just one of your holiday games;
You may think at first I'm as mad as a hatter
When I tell you, a cat must have THREE DIFFERENT NAMES.
[...]
But above and beyond there's still one name left over,
And that is the name that you never will guess;
The name that no human research can discover—
But THE CAT HIMSELF KNOWS, and will never confess.*

```
pu  
{  
    ...  
}
```

T S Eliot

```
public class Ellipse
{
    private double semiMajor, semiMinor;
    public Ellipse(double a, double b) ...
    public double semiMajorAxis() ...
    public double semiMinorAxis() ...
    ...
}
```

```
public class Circle
{
    private double radius;
    public Circle(double r) ...
    public double radius() ...
    public Ellipse toEllipse() ...
    ...
}
```

```
public class Ellipse
{
    private double semiMajor, semiMinor;
    public Ellipse(double a, double b) ...
    public double semiMajorAxis() ...
    public double semiMinorAxis() ...
    public boolean isCircle() ...
    ...
}
```

When it is not
necessary to
change, it is
necessary not to
change.

Lucius Cary

In many object-oriented programming languages the concept of *inheritance* is present, which provides a mechanism for sharing code among several classes of objects. Many people even regard inheritance as the hallmark of object-orientedness in programming languages. We do not agree with this view, and argue that the essence of object-oriented programming is the encapsulation of data and operations in objects and the protection of individual objects against each other.

Pierre America

"A Behavioural Approach to Subtyping in Object-Oriented Programming Languages"

In many object-oriented programming languages the concept of *inheritance* is present, which provides a mechanism for sharing code among several classes of objects. Many people even regard inheritance as the hallmark of object-orientedness in programming languages. We do not agree with this view, and argue that the essence of object-oriented programming is the encapsulation of data and operations in objects and the protection of individual objects against each other.

The author considers this principle of protection of objects against each other as the basic and essential characteristic of object-oriented programming. It is a refinement of the technique of abstract data types, because it does not only protect one type of objects against all other types, but one object against all other ones. As a programmer we can consider ourselves at any moment to be sitting in exactly one object and looking at all the other objects from outside.

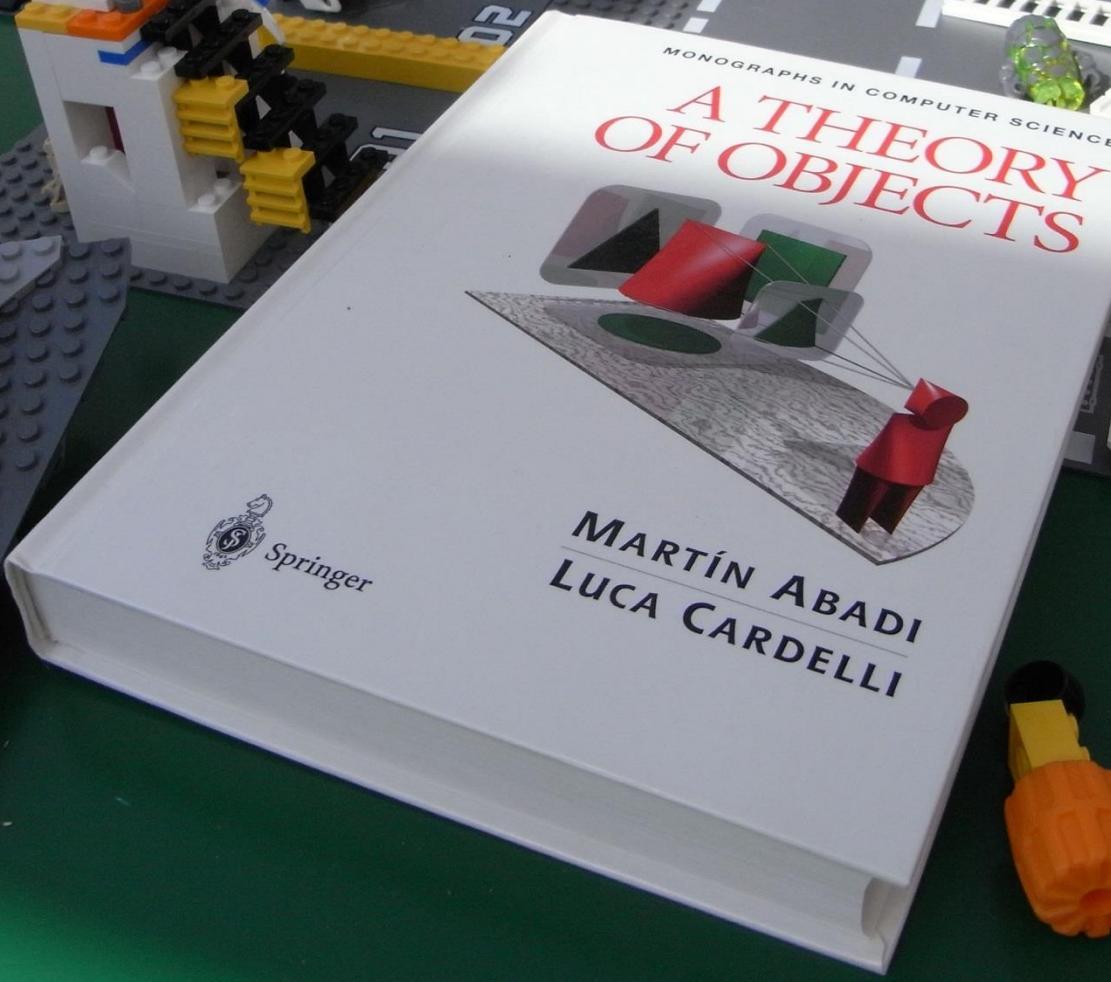
Pierre America

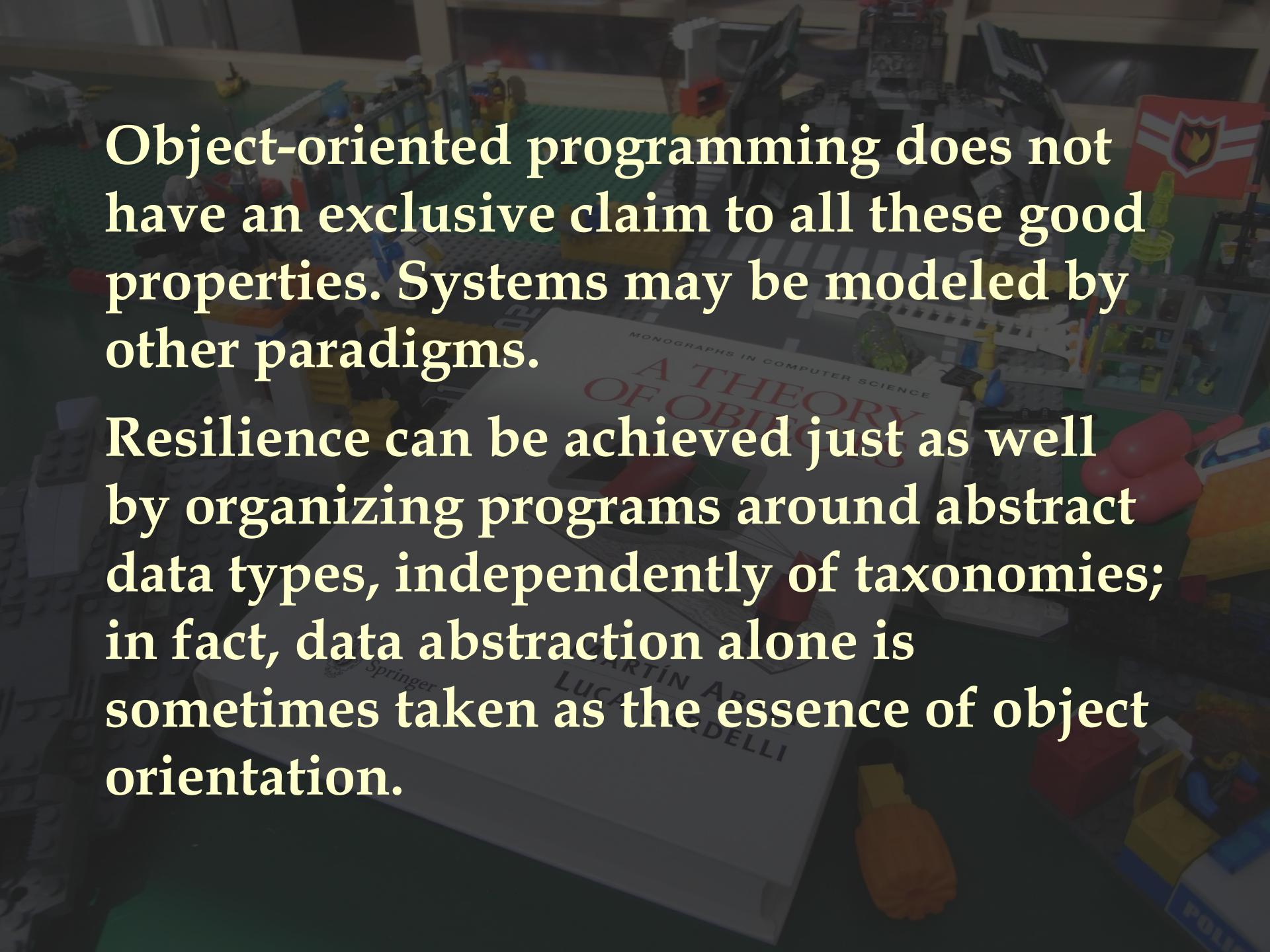
"A Behavioural Approach to Subtyping in Object-Oriented Programming Languages"

encapsulate, verb

- *enclose (something) in or as if in a capsule.*
- *express the essential feature of (someone or something) succinctly.*
- *provide an interface for (a piece of software or hardware) to allow or simplify access for the user.*

The New Oxford Dictionary of English





Object-oriented programming does not have an exclusive claim to all these good properties. Systems may be modeled by other paradigms.

Resilience can be achieved just as well by organizing programs around abstract data types, independently of taxonomies; in fact, data abstraction alone is sometimes taken as the essence of object orientation.

An abstract data type (ADT) is a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

https://en.wikipedia.org/wiki/Abstract_data_type

SANDLER INTERNAL OBJECTS REVISITED

KARNAC
BOOKS

The Self and the Object World

Edith Jacobson M.D.

1962
JAC M.D.

The shadow of the object

Christopher Bollas

FA

Greenberg and Mitchell

Harvard

Object Relations in Psychoanalytic Theory

Montgomery

Stack

SANDLER

INTERNAL OBJECTS REVISITED

KARNAC
BOOKS

The Self and the Object World

Edith Jacobson M.D.

The shadow of the object

Christopher Bollas

FAB

Greenberg and Mitchell

Harvard

Object Relations in Psychoanalytic Theory

Stack

{push, pop, depth, top}

The Self and the Object World

Edith Jacobson M.D.

The shadow of the object

Christopher Bollas

Greenberg and Mitchell

Harvard

Object Relations in Psychoanalytic Theory

$\forall T \bullet \exists \text{Stack}$

{

empty: Stack[T],

push: Stack[T] \times T \rightarrow Stack[T],

pop: Stack[T] \rightarrow Stack[T],

depth: Stack[T] \rightarrow Integer,

top: Stack[T] \rightarrow T

}

Stack[T]

{

SANDLER INTERNAL OBJECTS REVISITED

KARNAC
BOOKS

The Self and the Object World

push(T),

Edith Jacobson M.D.

The shadow of the object

pop(),

Christopher Bollas

FAB

Greenberg and Mitchell

Object Relations in Psychoanalytic Theory

Harvard

}

We propose [...] that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.

David L Parnas

"On the Criteria to Be Used in Decomposing Systems into Modules"

An interface is a contract to deliver a certain amount of service.

Clients of the interface depend on the contract, which is usually documented in the interface specification.

Butler W Lampson
"Hints for Computer System Design"



Bertrand Meyer
**Object-oriented
Software
Construction**

PRENTICE HALL
INTERNATIONAL
SERIES IN
COMPUTER
SCIENCE

C.A.R. HOARE SERIES EDITOR

Stack[T]

{

push(item : T),

pop(),

depth() : Integer,

top() : T

}

postcondition:
 $\text{depth}() = 0$

given:

$\text{before} = \text{depth}()$

postcondition:

$\text{depth}() = \text{before} + 1 \wedge \text{top}() = \text{item}$

given:

$\text{before} = \text{depth}()$

precondition:

$\text{before} > 0$

postcondition:

$\text{depth}() = \text{before} - 1$

precondition:
 $\text{depth}() > 0$

given:

$\text{result} = \text{depth}()$

postcondition:

$\text{result} \geq 0$

C.A.R. Hoare
**Communicating
Sequential
Processes**

C.A.R. HOARE SERIES EDITOR

alphabet(Stack) =

{push, pop, depth, top}

The Self and the Object World

Edith Jacobson M.D.

The shadow of the object

Christopher Bollas

Greenberg and Mitchell

Harvard

Object Relations in Psychoanalytic Theory

trace(Stack) =

{<,>}

SANDLER

INTERNAL OBJECTS REVISITED

KARNAC
BOOKS

<push>,

<depth>,

<push, pop>,

<push, top>,

<push, depth>,

<push, push>, Psychoanalytic Theory

<depth, push>,

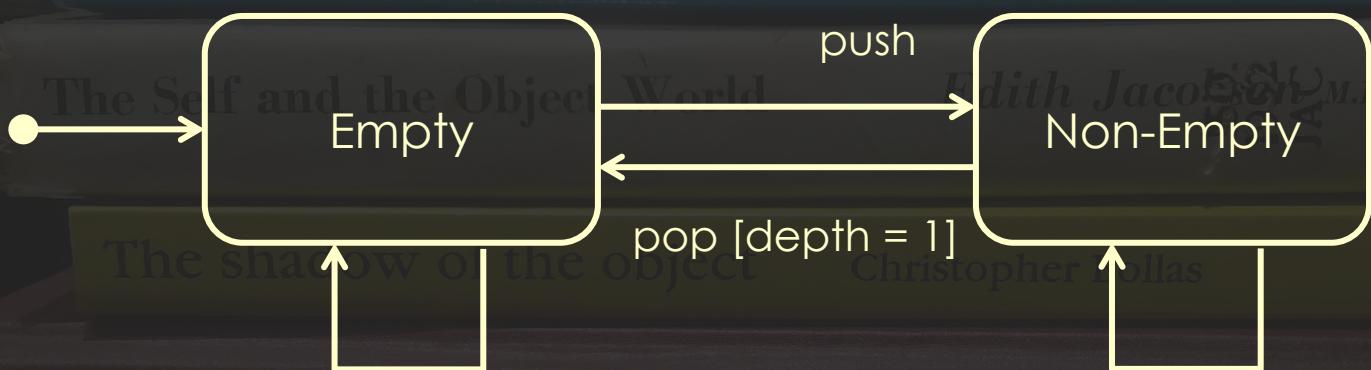
<depth, depth>,

<push, push, pop>,

...}

SANDLER INTERNAL OBJECTS REVISITED

KARNAC
BOOKS



Greenberg and M depth

Object Relations in Psychoanalytic

depth Harvard
top push
pop [depth > 1]

LOGIC

An introductory course

W.H. Newton-Smith

LOGIC

An introductory course
W.H.Newton-Smith

**Propositions
are vehicles
for stating
how things are
or might be.**

LOGIC

An introductory course
W.H.Newton-Smith

Thus only indicative sentences which it makes sense to think of as being true or as being false are capable of expressing propositions.



jasongorman

@jasongorman

Fun Fact: "Given... when... then..." is what we call a Hoare Triple [en.wikipedia.org/wiki/Hoare_log...](https://en.wikipedia.org/wiki/Hoare_logic)

7:42 PM - 3 Mar 2015



17



16

{P} S {Q}

```
public class Stack_spec
{
    public static class A_new_stack
    {
        @Test
        public void is_empty() ...
    }

    public static class An_empty_stack
    {
        @Test(...)
        public void throws_when_queried_for_its_top_item() ...
        @Test(...)
        public void throws_when_popped() ...
        @Test
        public void acquires_depth_by_retaining_a_pushed_item_as_its_top() ...
    }

    public static class A_non_empty_stack
    {
        @Test
        public void becomes_deeper_by_retaining_a_pushed_item_as_its_top() ...
        @Test
        public void on_popping_reveals_tops_in_reverse_order_of_pushing() ...
    }
}
```

```
public class
  Stack_spec
{
  public static class
    A_new_stack
  {
    @Test
    public void is_empty() ...
  }

  public static class
    An_empty_stack
  {
    @Test(...)
    public void throws_when_queried_for_its_top_item() ...
    @Test(...)
    public void throws_when_popped() ...
    @Test
    public void acquires_depth_by_retaining_a_pushed_item_as_its_top() ...
  }

  public static class
    A_non_empty_stack
  {
    @Test
    public void becomes_deeper_by_retaining_a_pushed_item_as_its_top() ...
    @Test
    public void on_popping_reveals_tops_in_reverse_order_of_pushing() ...
  }
}
```

```
public class
  Stack_spec
{
  public static class
    A_new_stack
  {
    @Test
    public void is_empty()
    }

    public static class
      An_empty_stack
    {
      @Test(...)
      public void throws_when_queried_for_its_top_item()
      @Test(...)
      public void throws_when_popped()
      @Test
      public void acquires_depth_by_retaining_a_pushed_item_as_its_top()
    }

    public static class
      A_non_empty_stack
    {
      @Test
      public void becomes_deeper_by_retaining_a_pushed_item_as_its_top()
      @Test
      public void on_popping_reveals_tops_in_reverse_order_of_pushing()
    }
}
```

```
public class Stack<T>
{
    private List<T> items = new ArrayList<>();
    public int depth()
    {
        return items.size();
    }
    public T top()
    {
        if (depth() == 0)
            throw new IllegalStateException();
        return items.get(depth() - 1);
    }
    public void pop()
    {
        if (depth() == 0)
            throw new IllegalStateException();
        items.remove(depth() - 1);
    }
    public void push(T newTop)
    {
        items.add(newTop);
    }
}
```

```
public interface Stack<T>
{
    int depth();
    T top();
    Stack<T> pop();
    Stack<T> push(T newTop);
    ...
}
```

The Self and the Object World

Edith Jacobson M.D.

The shadow of the object

Christopher Bollas

Greenberg and Mitchell

Harvard

Object Relations in Psychoanalytic Theory

```
public interface Stack<T>
{
    ...
    class Empty<T> implements Stack<T>
    {
        public int depth()
        {
            return 0;
        }
        public T top()
        {
            throw new IllegalStateException();
        }
        public Stack<T> pop()
        {
            throw new IllegalStateException();
        }
        public Stack<T> push(T newTop)
        {
            return new NonEmpty(newTop, this);
        }
    }
    ...
}
```

SANDLER INTERNAL OBJECTS REVISITED

KARNAC
BOOKS

The Self and the Object World

Edith Jacobson, M.D.

The shadow of the object

Christopher Bollas

Greenberg and Mitchell

Harvard

Object-Oriented Python Analytic Theory

```
public interface Stack<T>
{
    ...
    default Stack<T> push(T newTop)
    {
        return new NonEmpty<>(newTop, this);
    }
    class Empty<T> implements Stack<T>
    {
        public int depth()
        {
            return 0;
        }
        public T top()
        {
            throw new IllegalStateException();
        }
        public Stack<T> pop()
        {
            throw new IllegalStateException();
        }
    }
    ...
}
```

```
public interface Stack<T>
{
    ...
    class NonEmpty<T> implements Stack<T>
    {
        private T top;
        private Stack<T> tail;
        public NonEmpty(T newTop, Stack<T> previous)
        {
            top = newTop;
            tail = previous;
        }
        public int depth()
        {
            return 1 + tail.depth();
        }
        public T top()
        {
            return top;
        }
        public Stack<T> pop()
        {
            return tail;
        }
    }
}
```

```
Stack<String> stack = new Stack<>();  
stack.push("WAW");  
stack.push("VNO");
```

SANDLER INTERNAL OBJECTS REVISITED

KARNAC
BOOKS

The Self and the Object World

Edith Jacobson M.D.

JAC

The shadow of the object

Christopher Bollas

FA

Greenberg and Mitchell

Harvard

Object Relations in Psychoanalytic Theory

```
Stack<String> stack = new Stack.Empty<>();  
stack = stack.push("WAW").push("VNO");
```

SANDLER INTERNAL OBJECTS REVISITED

KARNAC
BOOKS

The Self and the Object World

Edith Jacobson M.D.

The shadow of the object

Christopher Bollas

FAB

Greenberg and Mitchell

Harvard

Object Relations in Psychoanalytic Theory

```
public interface Stack<T>
{
```

```
    int depth();
```

```
    T top();
```

```
    Stack<T> pop();
```

```
    Stack<T> push(T newTop);
```

```
    ...
```

```
}
```

Greenberg and Mitchell

Object Relations in Psychoanalytic Theory

Either T must be a type whose instances are immutable, or only instances that will remain immutable may be *pushed*.

Edith Jacobson M.D.

FA

Harvard

practice

Article contributed by **Elmer**
Software Corp.

00010.1148.3044112

We need it, we can afford it,
and the time is now.

BY PAT HILLARD

Immutability Changes Everything

latches has become harder. Latency losses lots of opportunities. Keeping copies of lots of data is now easier, and one payoff is reduced coding challenges.

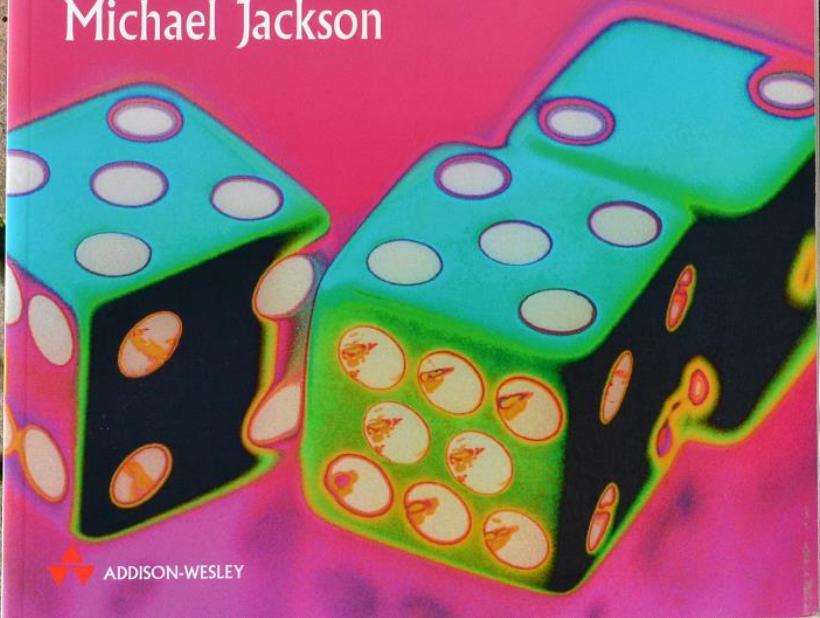
Storage is increasing as the cost per terabyte of disk keeps dropping. This means a lot of data can be kept for a long time. Distribution is becoming as more and more data and services are spread across a great distance. Data within a data center seems "far away." Data within a many-node cluster may seem "far away." Ambiguity increases when trying to coordinate with systems that are far away—new stuff has happened since you last heard the news. Can you take action with incomplete knowledge? Can you wait for enough knowledge?

Turtles all the way down. As various technological areas have rolled, they have responded to them and of increasing storage, abstraction, and automation.

PROBLEM FRAMES

Michael Jackson

Analyzing
and structuring
software
development
problems



ADDISON-WESLEY

Phenomenon: An element of what we can observe in the world. Phenomena may be individuals or relations. Individuals are entities, events, or values. Relations are roles, states, or truths.

Individual: An individual is a phenomenon that can be named and is distinct from every other individual: for example, the number 17, George III, or Deep Blue's first move against Kasparov.

Value: A value is an intangible individual that exists outside time and space, and is not subject to change.

```
public class Date implements ...  
{  
    ...  
    public int getYear() ...  
    public int getMonth() ...  
    public int getDayInMonth() ...  
    public void setYear(int newYear) ...  
    public void setMonth(int newMonth) ...  
    public void setDayInMonth(int newDayInMonth) ...  
    ...  
}
```

```
public class Date implements ...  
{  
    ...  
    public int getYear() ...  
    public int getMonth() ...  
    public int getWeekInYear() ...  
    public int getDayInYear() ...  
    public int getDayInMonth() ...  
    public int getDayInWeek() ...  
    public void setYear(int newYear) ...  
    public void setMonth(int newMonth) ...  
    public void setWeekInYear(int newWeek) ...  
    public void setDayInYear(int newDayInYear) ...  
    public void setDayInMonth(int newDayInMonth) ...  
    public void setDayInWeek(int newDayInWeek) ...  
    ...  
}
```

Just because you
have a getter,
doesn't mean you
should have a
matching setter.



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

A Pattern Language for
Distributed Computing



Volume 4

Frank Buschmann
Kevlin Henney
Douglas C. Schmidt



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

Immutable Value

Define a value object type whose instances are immutable. The internal state of a value object is set at construction and no subsequent modifications are allowed.

Kevlin Henney
Douglas C. Schmidt

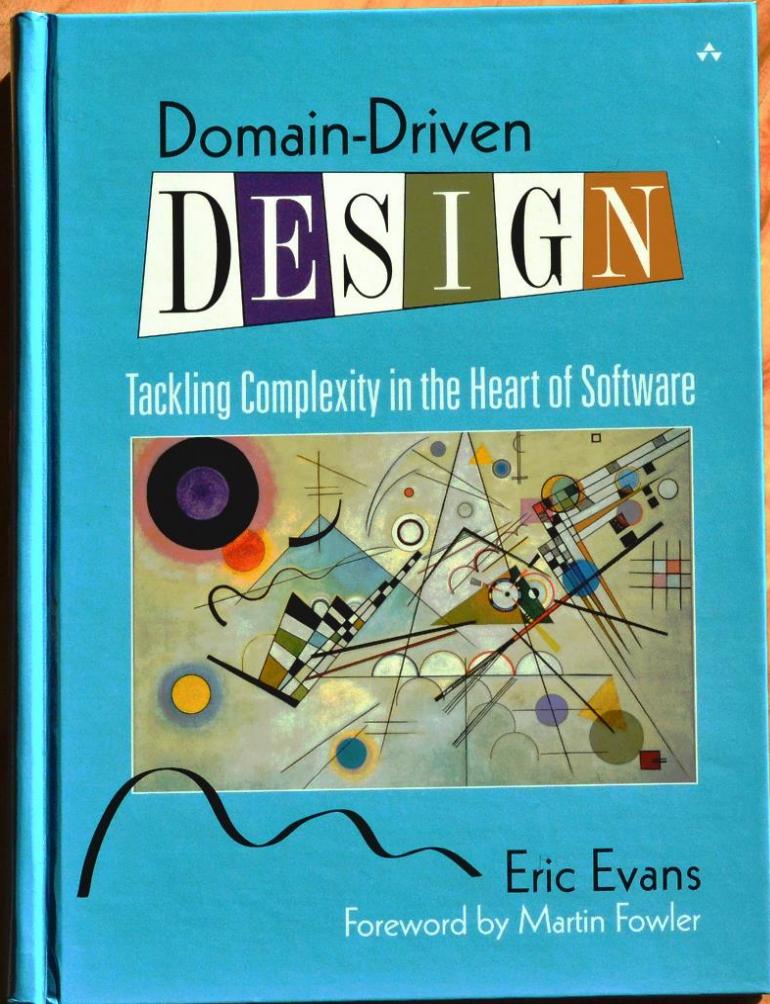
```
public final class Date implements ...  
{  
    ...  
    public int getYear() ...  
    public int getMonth() ...  
    public int getWeekInYear() ...  
    public int getDayInYear() ...  
    public int getDayInMonth() ...  
    public int getDayInWeek() ...  
    ...  
}
```

Domain-Driven DESIGN

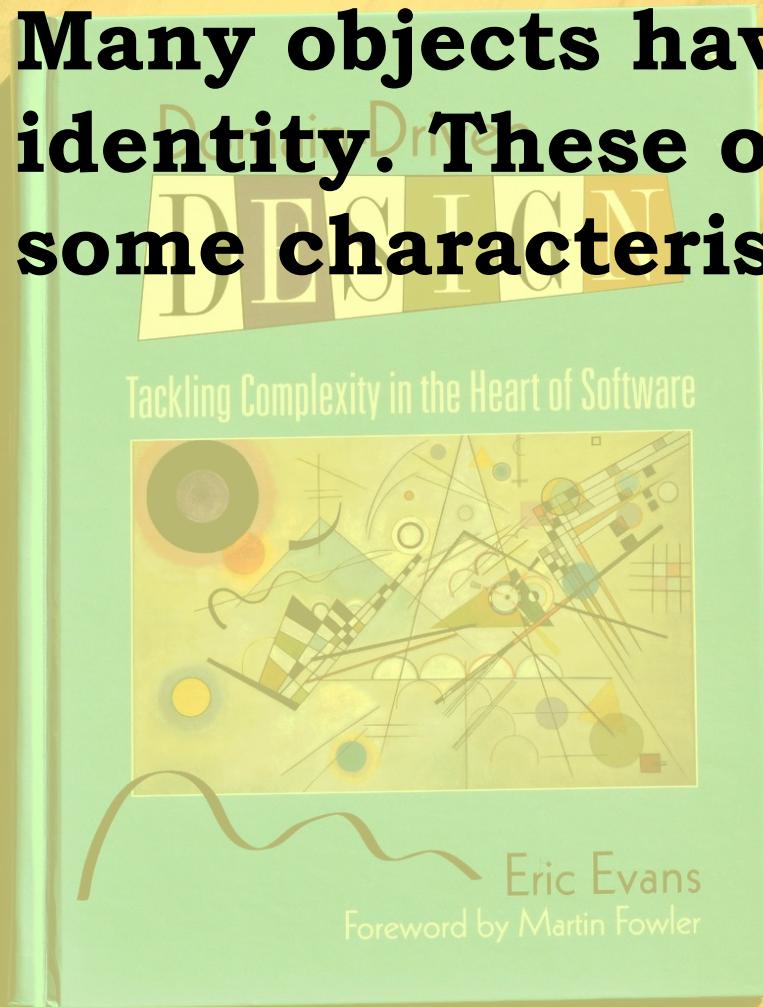
Tackling Complexity in the Heart of Software



Eric Evans
Foreword by Martin Fowler



Many objects have no conceptual identity. These objects describe some characteristic of a thing.



When you care only about the attributes of an element of the model, classify it as a *value object*. Make it express the meaning of the attributes it conveys and give it related functionality. Treat the value object as immutable. Don't give it any identity and avoid the design complexities necessary to maintain entities.

Identity

State

Behaviour

Identity

State

Behaviour

```
public final class Date implements ...  
{  
    ...  
    public int getYear() ...  
    public int getMonth() ...  
    public int getWeekInYear() ...  
    public int getDayInYear() ...  
    public int getDayInMonth() ...  
    public int getDayInWeek() ...  
    ...  
}
```

```
public final class Date implements ...  
{  
    ...  
    public int year() ...  
    public int month() ...  
    public int weekInYear() ...  
    public int dayInYear() ...  
    public int dayInMonth() ...  
    public int dayInWeek() ...  
    ...  
}
```

get

 PRONUNCIATION SPELLINGS ETYMOLOGY QUOTATIONS

Find

get, n.¹get, n.²get, n.³

get, v.

geta, n. pl.

Getan, a. (and n.)

get-at-able, a.

get-away, getaw

gete, n.

gete, v.

gete

getee

geten

getenly, adv.

geterne

get-go, n.

gethe

gether, adv.

gethicall

Gethsemane

get, v.

(gɛt)

Pa. tense **got** (arch. **gat**). Pa. pple. **got** (**gotten**). Pres. pple. **getting**. Forms: inf. 3-4 **geten**, (5 **getyn**), 3-6 **gete**, (4 **geit**, **geyt**, **gite**, Sc. **gat(e)**, 4-5 **gyte**, 6 Sc. **gait**), 3-7 **gett**, (4-6 **gette**, 4 **gitte**, 5 **gytt**, 9 dial. **git**), 3-**get**. pa. tense 3-7 **gate**, (3 **gait**, 4 **get**, pl. **gaten**, **geton**, -**yn**, **geetun**, **getton**, 5 **geten**), 3-6 **gatt**, (4-6 **gatte**), 3-**gat**, 6-**got**, (6 **got(t)e**). pa. pple. α. 3-5 **geten**, (3 **ȝeten**, **getun**, 4 **getin**, **geteyn**, **giten**, -**in**, **gyten**, -**in**, 4-6 **getyn**, 5 **geton**), 3-5 **getten**, (4-5 **gettyn**, 5 **getton**, 6 **gitten**), 4-6 **gete**, (4 **i-gete**, 5 **y-gete**, **gyte**), 4-6 **gette**, (5 **y-gette**), 5-6 **gett**, (5 **get**). β. 3-4 **gotin**, 3-6 **goten**, (4 **gotyn**, **gote**, 5 **y-goten**, **goton**, **gothen**), 4-6 Sc. **gottin**, -**yn**, 5-7 **gotton**, 6-**gotten**, **got**, (6 **y-got**).

[a. ON. *geta* (*gat*, *gátum*, *getenn*) to get, obtain, to beget, also, to guess (Sw. *gitta*, Da. *gide* to be able or willing, MSw. *gäta*, Da. *gjette* to guess) = OE. -*ȝietan* (only in the compounds *a-*, *be-*, *for-*, *ofer-*, *on-*, *under-*-*ȝietan*: see **BEGET**, **FORGET**), OFris. (*ur-*, *for-*-*jeta*, OS. (*bi-*, *far-*)-*getan* (MDu. *ver-gheten*, Du. *ver-geten*), OHG. *ge<zced><zced>an*, *ke<zced><zced>an* (once in pple.

set

 PRONUNCIATION SPELLINGS ETYMOLOGY QUOTATIONS

Find

set, n.¹set, n.²set, v.¹set, v.²

set, ppl. a.

set, conj.

set-

seta

setace

setaceo-

setaceous, a.

setaceous

setal, a.

setar

setarious, a.

set-aside, n. and

set-back

setchal, setchel(

set-down

sete, n.

set, v.¹(s^ɛt)Forms: see below. Pa. tense and pple. **set**.

[Com. Teut.: OE. *sættan* = OFris. *setta* (mod.Fris. *sette*), OS. *settian* (MDu., MLG. *setten*, Du. *zetten*), OHG. *sezzan* beside *sazzan* (MHG. *sezzen*, G. *setzen*), ON. *setja* (Sw. *satta*, Da. *sætte*), Goth. *satjan*; causal of **setjan* (*sitjan*) to **SIT**.]

Confusion between *set* and *sit* arose as early as the beginning of the 14th c., owing partly to the identity or close similarity of the forms of their past tenses and pa. pples., and partly to the identity of meaning in some uses, as between *to be set* (= seated) and *to sit*; cf. **SIT V.** (etym. note and A. 5 a α note). For cases of mere substitution of forms of *sit* for forms of *set*, see A. 1 γ, 2 ζ below. The spelling *sett* is still sometimes found in technical senses; cf. **SET n.¹**]

A. Inflexional Forms.

1. a. inf. and pres. stem. (α) 1 **settan** (Northumb. **setta**), 2-5 (6 arch.) **setten**, 3-6 **sette** (2 **setton**, **seotte**, 3 Orm. **settenn**, Lay. **sætten**, 4 Kent. **zettēn**, 5 **settyn**, **cettyn**, **satte**, 6 **seatt-**), 4-9

reset

 PRONUNCIATION SPELLINGS ETYMOLOGY QUOTATIONS

Find

reset, *n.*¹reset, *n.*²reset, *v.*¹reset, *v.*²

resetment

resettable, *a.*

resetter

resettle, *v.*

resettlement

reseve

resew, *v.*

resew, reseyve

reseyt

resgat

resh

reshape, *v.*

reshaper

reshape, *v.*resharpen, *v.*resheathe, *v.***reset**, *v.*²

(ri:'set)

Also **re-set**.

[RE- 5 a.]

trans. To set again, in various senses of the verb.**I. 1. a.** To replace (esp. gems) in a (former or new) setting.

1655 FULLER *Ch. Hist.* v. iv. §7 Elizabeth,..finding so fair a flower..fallen out of her Crown, was careful quickly to gather it up again, and get it re-sett therein. **1684** R. WALLER *Nat. Exper.* Pref., For a time they fall out of their Collets.., and [are] worth nothing till..they are again reset in their proper places. **1830** LYTTON P. *Clifford* xix, A stray trinket or two—not of sufficient worth to be reset. **1883** HALDANE *Worksh. Rec.* Ser. ii. 371/2 The hair can be again reset as firmly as it was before [etc.].

b. *Surg.* To set (a broken limb) again.

unset

 PRONUNCIATION SPELLINGS ETYMOLOGY QUOTATIONS

Find

unset, v.

unset, *ppl. a.*unse**e**t**e**, *a.*unsett**e**unsetting, *ppl. a.*unsett**e**le, *v.*unsett**e**able, *a.*unsettled, *ppl. a.*

unsettledness

unsettlement

unsettling, *vbl. n.*unsety, *a.*unseven, *v.*unsever, *v.*unseverable, *a.*unseverably, *adv.*unsevere, *a.*unsevered, *ppl. a.*unsew, *v.*unsewed, *ppl. a.***un'set**, *v.*[**UN⁻² 3, 7.** Cf. OE. *unsettan* (once), to take down.]**1.** *trans.* To put out of place or position; to undo the setting of.

1602 MARSTON *Ant. & Mel.* III. Wks. 1856 I. 37 O, you spoyle my ruffe, unset my haire. **1611** COTGR., *Desplanter*,..to vnplant, vnset, remoue. **1761** GRAY *Lett.* (1900) II. 204 The man was sent for: he unset it; it was a paste not worth 40 shillings. **1775** MRS. DELANY in *Life & Corr.* Ser. II. (1862) II. 105 There is some hazard in unsettling enamel for fear of chipping the edges. **1836** MARRYAT *Midsh. Easy* xxxii, How could he put the young men to fresh tortures by removing splints and unsettling limbs? **1884** *Law Times* 1 Nov. 8/1 On the morning in question Dawson had unset the gun.

2. *intr.* To get out of place or position.

1703 THORESBY *Let. to Ray, Spelk*, a wooden splinter tied on, to keep a broken bone from bending or unsettling again.

"Get something"
is an imperative
with an expected
side effect.

```
public final class Date implements ...  
{  
    ...  
    public int year() ...  
    public int month() ...  
    public int weekInYear() ...  
    public int dayInYear() ...  
    public int dayInMonth() ...  
    public int dayInWeek() ...  
    ...  
}
```

```
public final class Date implements ...  
{  
    ...  
    public int year() ...  
    public int month() ...  
    public int weekInYear() ...  
    public int dayInYear() ...  
    public int dayInMonth() ...  
    public int dayInWeek() ...  
    public Date withYear(int newYear) ...  
    ...  
}
```



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

Builder

DESIGN-PATTERN

Introduce a builder that provides separate methods for constructing and disposing of each different part of a complex object, or for combining cumulative changes in the construction of whole objects.

```
public final class Date implements ...  
{  
    ...  
    public int year() ...  
    public int month() ...  
    public int weekInYear() ...  
    public int dayInYear() ...  
    public int dayInMonth() ...  
    public int dayInWeek() ...  
    public Date withYear(int newYear)  
    {  
        return new Date(newYear, month(), dayInMonth());  
    }  
    ...  
}
```

```
public final class Date implements ...
{
    ...
    public int year() ...
    public int month() ...
    public int weekInYear() ...
    public int dayInYear() ...
    public int dayInMonth() ...
    public int dayInWeek() ...
    public Date withYear(int newYear)
    {
        return
            newYear == year()
            ? this
            : new Date(newYear, month(), dayInMonth());
    }
    ...
}
```

```
public final class Date implements ...  
{  
    ...  
    public int year() ...  
    public int month() ...  
    public int weekInYear() ...  
    public int dayInYear() ...  
    public int dayInMonth() ...  
    public int dayInWeek() ...  
    public Date withYear(int newYear) ...  
    public Date withMonth(int newMonth) ...  
    public Date withWeekInYear(int newWeekInYear) ...  
    public Date withDayInYear(int newDayInYear) ...  
    public Date withDayInMonth(int newDayInMonth) ...  
    public Date withDayInWeek(int newDayInWeek) ...  
    ...  
}
```



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

Copied Value

Define a value object type whose instances are copyable. When a value is used in communication with another thread, ensure that the value is copied.

Kevlin Henney

Douglas C. Schmidt

```
class date
{
public:
    date(int year, int month, int day_in_month);
    date(const date &);
    date & operator=(const date &);
    ...
    int year() const;
    int month() const;
    int day_in_month() const;
    ...
    void year(int);
    void month(int);
    void day_in_month(int);
    ...
};
```

```
class date
{
public:
    date(int year, int month, int day_in_month);
    date(const date &);
    date & operator=(const date &);
    ...
    int year() const;
    int month() const;
    int day_in_month() const;
    ...
    void set(int year, int month, int day_in_month);
    ...
};
```

```
today.set(2016, 11, 19);
```

```
class date
{
public:
    date(int year, int month, int day_in_month);
    date(const date &);
    date & operator=(const date &);
    ...
    int year() const;
    int month() const;
    int day_in_month() const;
    ...
};
```

```
today = date(2016, 11, 19);
```

```
class date
{
public:
    date(int year, int month, int day_in_month);
    date(const date &);
    date & operator=(const date &);
    ...
    int year() const;
    int month() const;
    int day_in_month() const;
    ...
};
```

```
today = { 2016, 11, 19 };
```



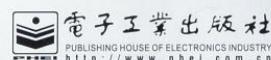
Collective Wisdom
from the Experts

程序员 应该知道的 97件事

O'REILLY®

Kevlin Henney 编

李军 译 吕骏 审校



Referential transparency is a very desirable property: it implies that functions consistently yield the same results given the same input, irrespective of where and when they are invoked. That is, function evaluation depends less—ideally, not at all—on the side effects of mutable state.

程序设计
应该知道的
97件事

O'REILLY®

Kevlin Henney 编
李军 译 吕骏 审校



电子工业出版社
"Apply Functional Programming Principles"

Edward Garson
"Apply Functional Programming Principles"

Idempotence is the property of certain operations in mathematics and computer science, that they can be applied multiple times without changing the result beyond the initial application.

The concept of idempotence arises in a number of places in abstract algebra [...] and functional programming (in which it is connected to the property of referential transparency).

<http://en.wikipedia.org/wiki/Idempotent>

Asking a question
should not change
the answer.

Bertrand Meyer

Asking a question
should not change
the answer, and
nor should asking
it twice!

In computing, a persistent data structure is a data structure that always preserves the previous version of itself when it is modified. Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure.

(A persistent data structure is not a data structure committed to persistent storage, such as a disk; this is a different and unrelated sense of the word "persistent.")

http://en.wikipedia.org/wiki/Persistent_data_structure

a



b = a



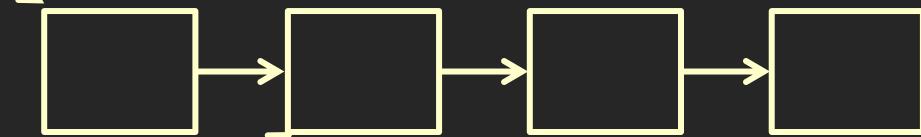
a



c = pop(a)



a



c = pop(a)

a



d = push(a, i)

LISP 1.5 Programmer's Manual

The Computation Center
and Research Laboratory of Electronics

Massachusetts Institute of Technology

I still have a deep fondness for the Lisp model. It is simple, elegant, and something with which all developers should have an infatuation at least once in their programming life.

Kevlin Henney

"A Fair Share (Part I)", *CUJ C++ Experts Forum*, October 2002

In functional programming, programs are executed by evaluating expressions, in contrast with imperative programming where programs are composed of statements which change global state when executed. Functional programming typically avoids using mutable state.

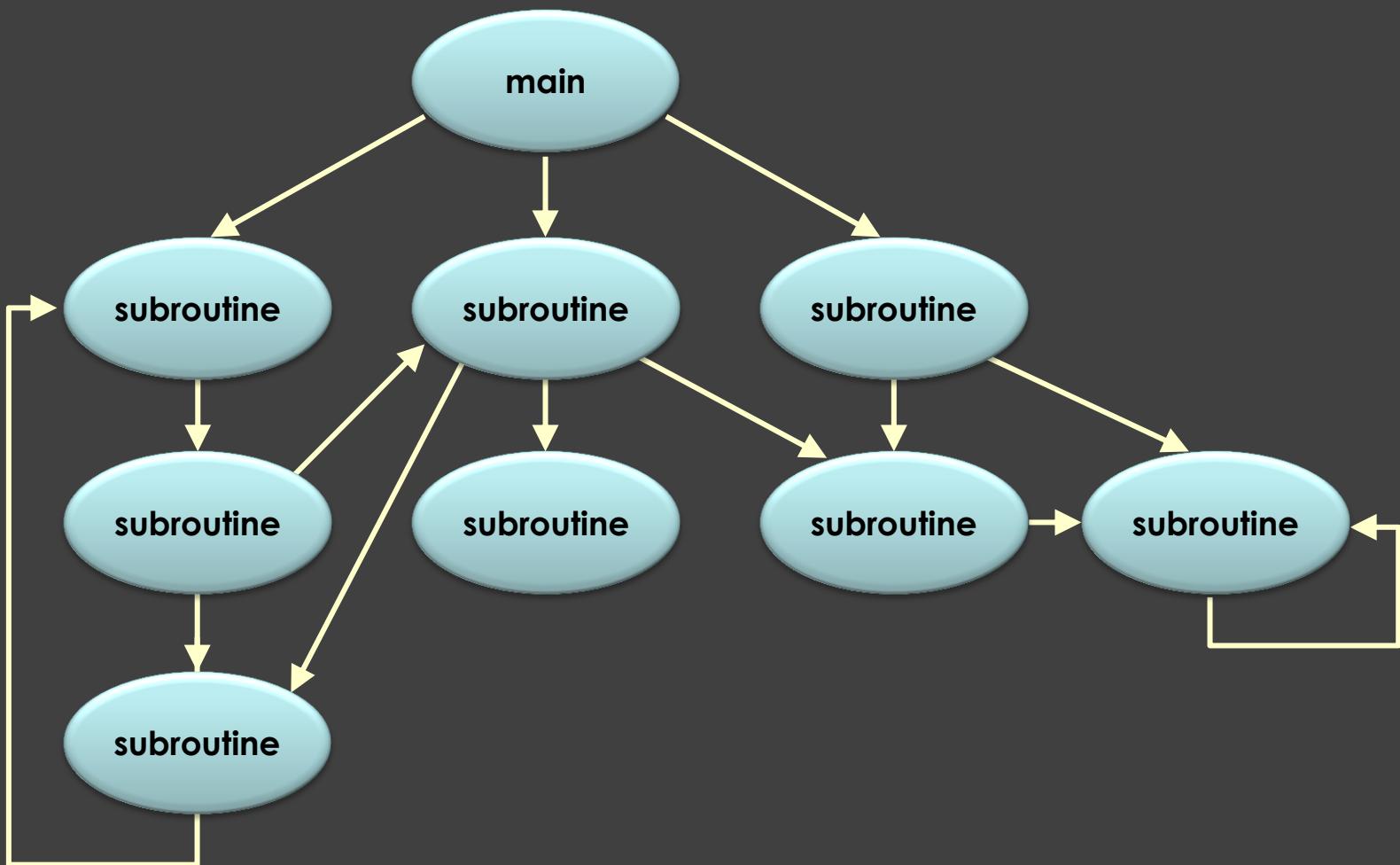
https://wiki.haskell.org/Functional_programming

STRUCTURED PROGRAMMING

O.-J. DAHL, E. W. DIJKSTRA
and C. A. R. HOARE

Main Program and Subroutine

The goal is to decompose a program into smaller pieces to help achieve modifiability.
A program is decomposed hierarchically.



Main Program and Subroutine

The goal is to decompose a program into smaller pieces to help achieve modifiability. A program is decomposed hierarchically. There is typically a single thread of control and each component in the hierarchy gets this control (optionally along with some data) from its parent and passes it along to its children.

Len Bass, Paul Clements & Rick Kazman
Software Architecture in Practice

**A *goto* completely
invalidates the high-level
structure of the code.**

Taligent's Guide to Designing Programs

Letters to the Editor

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

Editor:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if B then A**), alternative clauses (**if B then A1 else A2**), choice clauses as introduced by C. A. R. Hoare (case[i] of (A_1, A_2, \dots, A_n)), or conditional expressions as introduced by J. McCarthy ($B_1 \rightarrow E_1, B_2 \rightarrow E_2, \dots, B_n \rightarrow E_n$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while B repeat A** or **repeat A until B**). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n , its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommendation to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Giuseppe Jacopini seems to have proved the (logical) superfluousness of the **go to** statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

REFERENCES:

1. WIRTH, NIKLAUS, AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM* 9 (June 1966), 413-432.
2. BÖHM, CORRADO, AND JACOPINI, GIUSEPPE. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM* 9 (May 1966), 366-371.

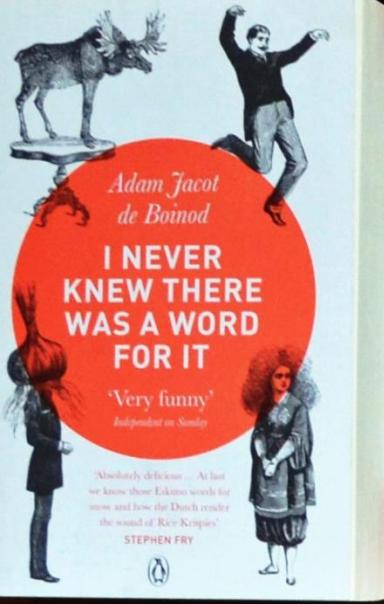
EDSGER W. DIJKSTRA
Technological University
Eindhoven, The Netherlands

2. **Rhetoric**: Repetition for vehemence or fullness.

lindrome (n) 'running back again' Words, phrases, sentences, etc., which read the same forwards and backwards. Richard A. Lanham *Adam, I'm Adam, I'm Adam*. A palindrome would seem to represent the compressed name of a Chiasmus.

epigram – Laudatio.

formal and ornate praise of person or deed. See *Rhetoric*: The branches in chapter 2.

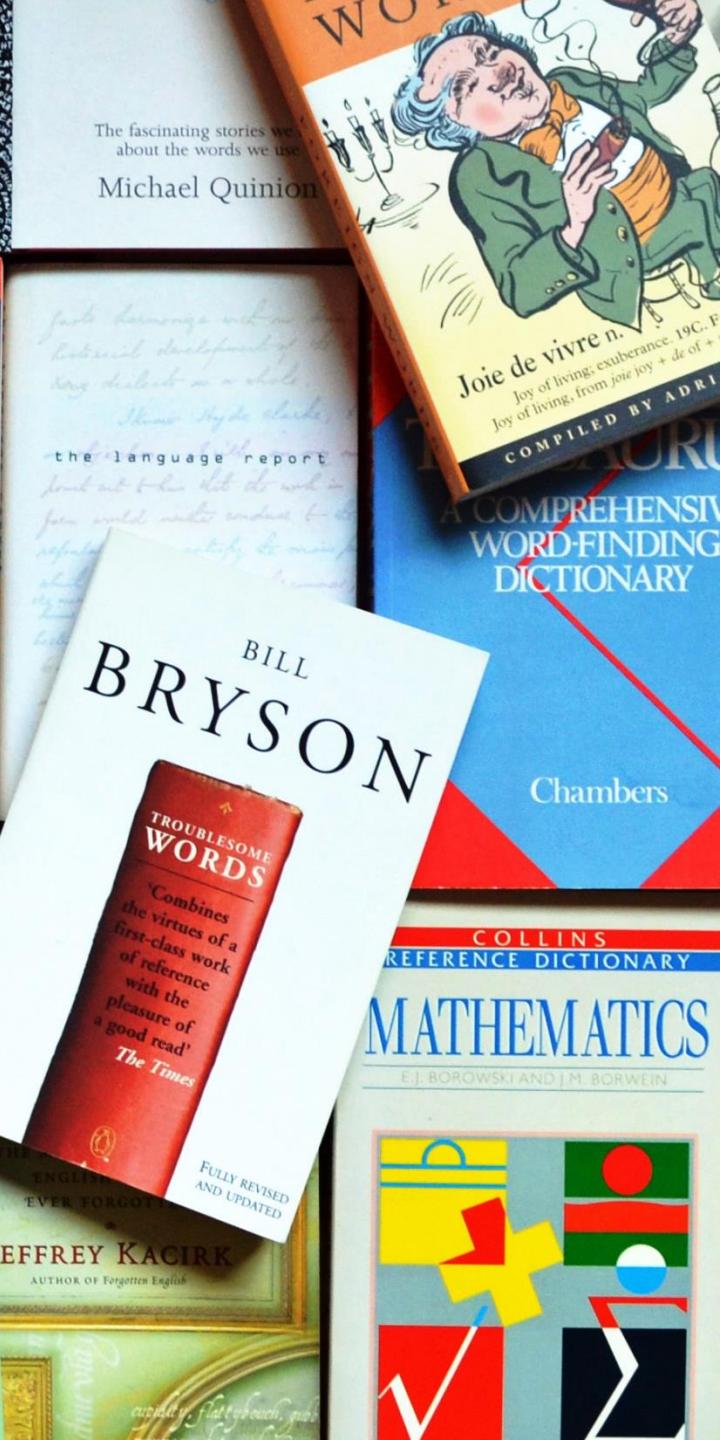


/ WordFriday



The fascinating stories we tell
about the words we use

Michael Quinion



snowclone, noun

- clichéd wording used as a template, typically originating in a single quote
- e.g., "X considered harmful", "These aren't the Xs you're looking for", "X is the new Y", "It's X, but not as we know it", "No X left behind", "It's Xs all the way down", "All your X are belong to us"

A Case against the GO TO Statement.

by Edsger W. Dijkstra
Technological University
Eindhoven, The Netherlands

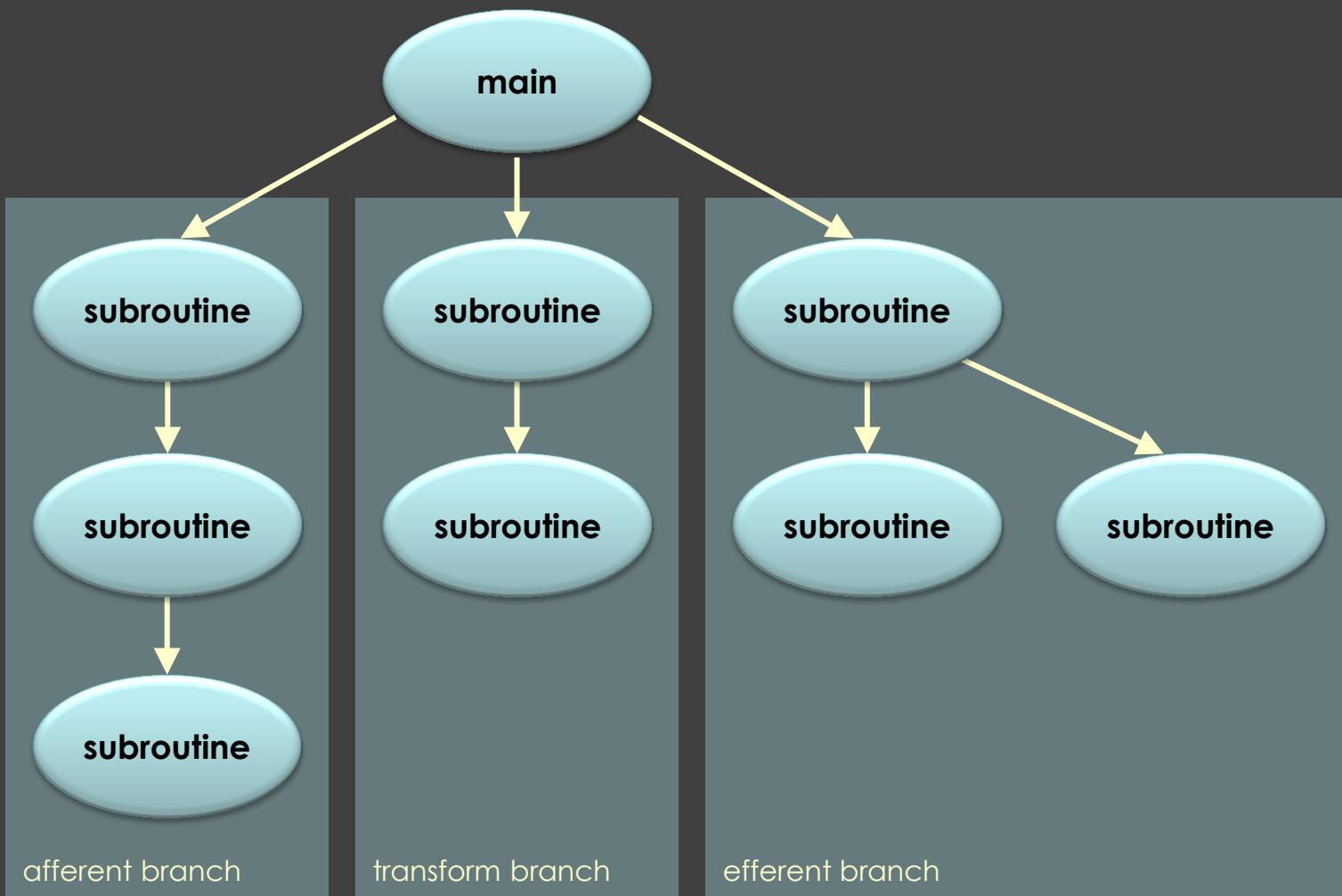
Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

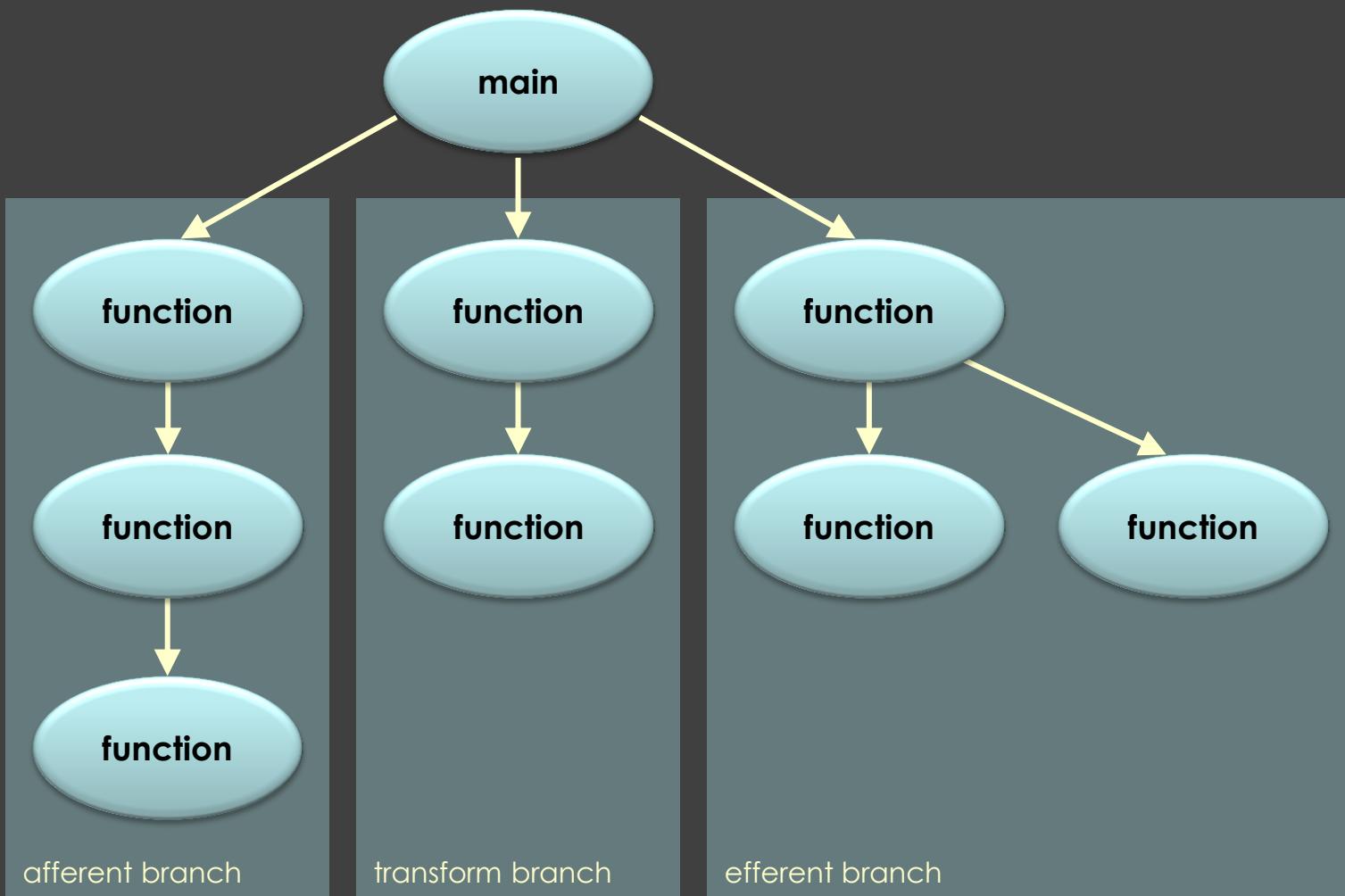
```
send(to, from, count)
register short *to, *from;
register count;
{
    register n=(count+7)/8;
    switch(count%8) {
        case 0: do{      *to = *from++;
        case 7:          *to = *from++;
        case 6:          *to = *from++;
        case 5:          *to = *from++;
        case 4:          *to = *from++;
        case 3:          *to = *from++;
        case 2:          *to = *from++;
        case 1:          *to = *from++;
                        }while(--n>0);
    }
}
```

Many people have said that the worst feature of C is that switches don't break automatically before each case label. This code forms some sort of argument in that debate, but I'm not sure whether it's for or against.

```
main() {  
    register short *to, *from;  
    register unsigned count;  
    {  
        switch(count%8) {  
            case 0: *to = *from++;  
            case 7: *to = *from++;  
            case 6: *to = *from++;  
            case 5: *to = *from++;  
            case 4: *to = *from++;  
            case 3: *to = *from++;  
            case 2: *to = *from++;  
            case 1: *to = *from++;  
        } while(--n>0);  
    }  
}
```

Tom Duff





Input

Process

Output

To iterate is human,
to recurse divine.

L Peter Deutsch

```
function factorial(n) {  
    var result = 1  
    while (n > 1)  
        result *= n--  
    return result  
}
```

```
function factorial(n) {  
    if (n > 1)  
        return n * factorial(n - 1)  
    else  
        return 1  
}
```

```
function factorial(n) {  
    return (  
        n > 1  
        ? n * factorial(n - 1)  
        : 1)  
}
```

Tail-call optimization is where you are able to avoid allocating a new stack frame for a function because the calling function will simply return the value that it gets from the called function.

The most common use is tail-recursion, where a recursive function written to take advantage of tail-call optimization can use constant stack space.

<http://stackoverflow.com/questions/310974/what-is-tail-call-optimization>

```
function factorial(n) {  
    function loop(n, result) {  
        return (  
            n > 1  
            ? loop(n - 1, n * result)  
            : result)  
    }  
    return loop(n, 1)  
}
```

```
factorial(N) ->
if
    N == 0 -> 1;
    true -> N * factorial(N - 1)
end.
```

```
factorial(N) when N == 0 -> 1;  
factorial(N) -> N * factorial(N - 1).
```

factorial(0) -> 1;

factorial(N) -> N * factorial(N - 1).

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n - 1)! \times n & \text{if } n > 0. \end{cases}$$

$$n!=\prod_{k=1}^n k$$

factorial n = product [1..n]

```
factorial n = foldl (*) 1 [1..n]
```

A higher-order function is a function that takes other functions as arguments or returns a function as result.

https://wiki.haskell.org/Higher_order_function

Use procedure arguments to provide flexibility in an interface.

This technique can greatly simplify an interface, eliminating a jumble of parameters that amount to a small programming language.

Butler W Lampson
"Hints for Computer System Design"

**If you have a procedure with
ten parameters, you probably
missed some.**

Alan Perlis

A simple example is an enumeration procedure that returns all the elements of a set satisfying some property. The cleanest interface allows the client to pass a filter procedure that tests for the property, rather than defining a special language of patterns or whatever.

Butler W Lampson
"Hints for Computer System Design"



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

Enumeration Method

Bring the iteration inside the aggregate and encapsulate it in a single enumeration method that is responsible for complete traversal. Pass the task of the loop—the action to be executed on each element of the aggregate—as an argument to the enumeration method, and apply it to each element in turn.



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

Lifecycle Callback

Define key lifecycle events as callbacks in an interface that is supported by framework objects. The framework uses the callbacks to control the objects' lifecycle explicitly.

Kevin Henney
Douglas C. Schmidt

```
public interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

public interface IObserver<in T>
{
    void OnCompleted();
    void OnError(Exception error);
    void OnNext(T value);
}

IDisposable subscription =
    source.Subscribe(
        value => handle element,
        error => handle exception,
        () => handle completion);
```



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

Observer

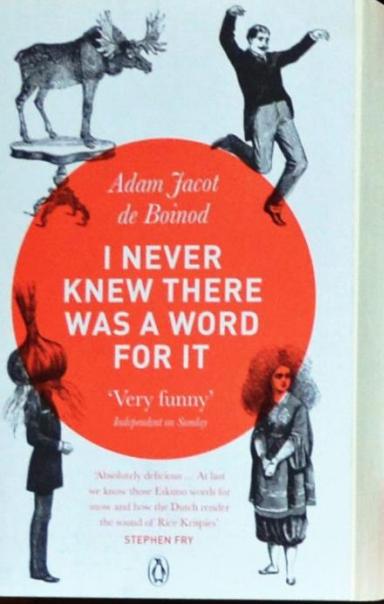
Define a change-propagation mechanism in which the provider—known as the ‘subject’—notifies registered consumers—known as the ‘observers’—whenever its state changes, so that the notified observers can perform whatever actions they deem necessary.

2. **Rhetoric**: Repetition for vehemence or fullness.

lindrome (n) 'running back again' Words, phrases, sentences, etc., which read the same forwards and backwards. Richard A. Lanham *Adam, I'm Adam*. I'm Adam.' A palindrome would seem to represent the compressed name of a Chiasmus.

epigram – Laudatio.

formal and ornate praise of person or deed. See *Rhetoric*: The branches in chapter 2.

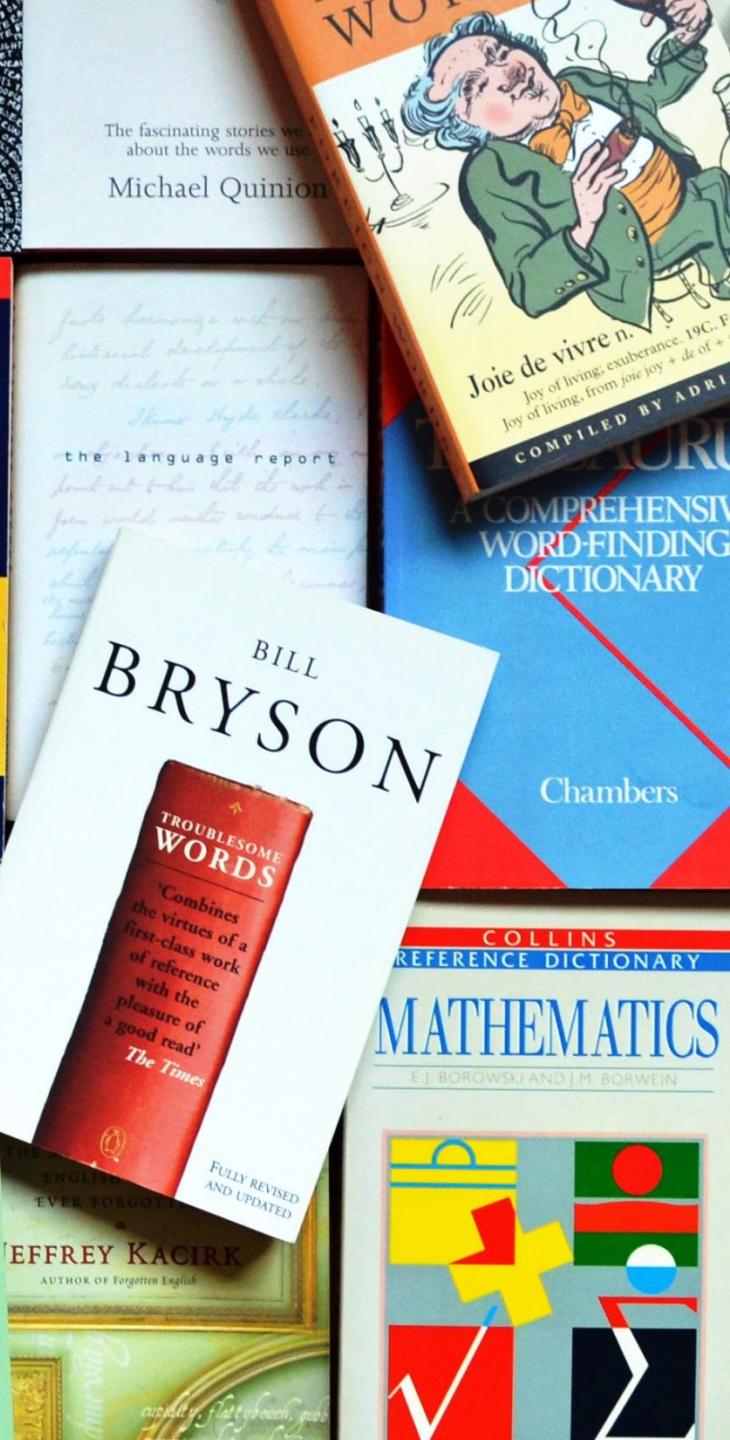


/ WordFriday



The fascinating stories we tell
about the words we use

Michael Quinion



bi-quinary coded decimal, noun

- A system of representing numbers based on counting in fives, with an additional indicator to show whether the count is in the first or second half of the decimal range, i.e., whether the number represented is in the range 0-4 or 5-9.
- This system is found in many abacus systems, with paired columns of counters (normally aligned) representing each bi-quinary range.
- The Roman numeral system is also a form of bi-quinary coded decimal.

```
def roman(number)
{
    result = ''
    while (number >= 1000)
    {
        result += 'M'
        number -= 1000
    }
    if (number >= 900)
    {
        result += 'CM'
        number -= 900
    }
    if (number >= 500)
    {
        result += 'D'
        number -= 500
    }
    if (number >= 400)
    {
        result += 'CD'
        number -= 400
    }
    if (number >= 100)
    {
        result += 'C'
        number -= 100
    }
    if (number >= 90)
    {
        result += 'XC'
        number -= 90
    }
    if (number >= 50)
    {
        result += 'L'
        number -= 50
    }
    if (number >= 40)
    {
        result += 'XL'
        number -= 40
    }
    if (number >= 10)
    {
        result += 'X'
        number -= 10
    }
    if (number == 9)
    {
        result += 'IX'
        number -= 9
    }
    if (number == 5)
    {
        result += 'V'
        number -= 5
    }
    if (number == 4)
    {
        result += 'IV'
        number -= 4
    }
    if (number == 1)
    {
        result += 'I'
        number -= 1
    }
}
```

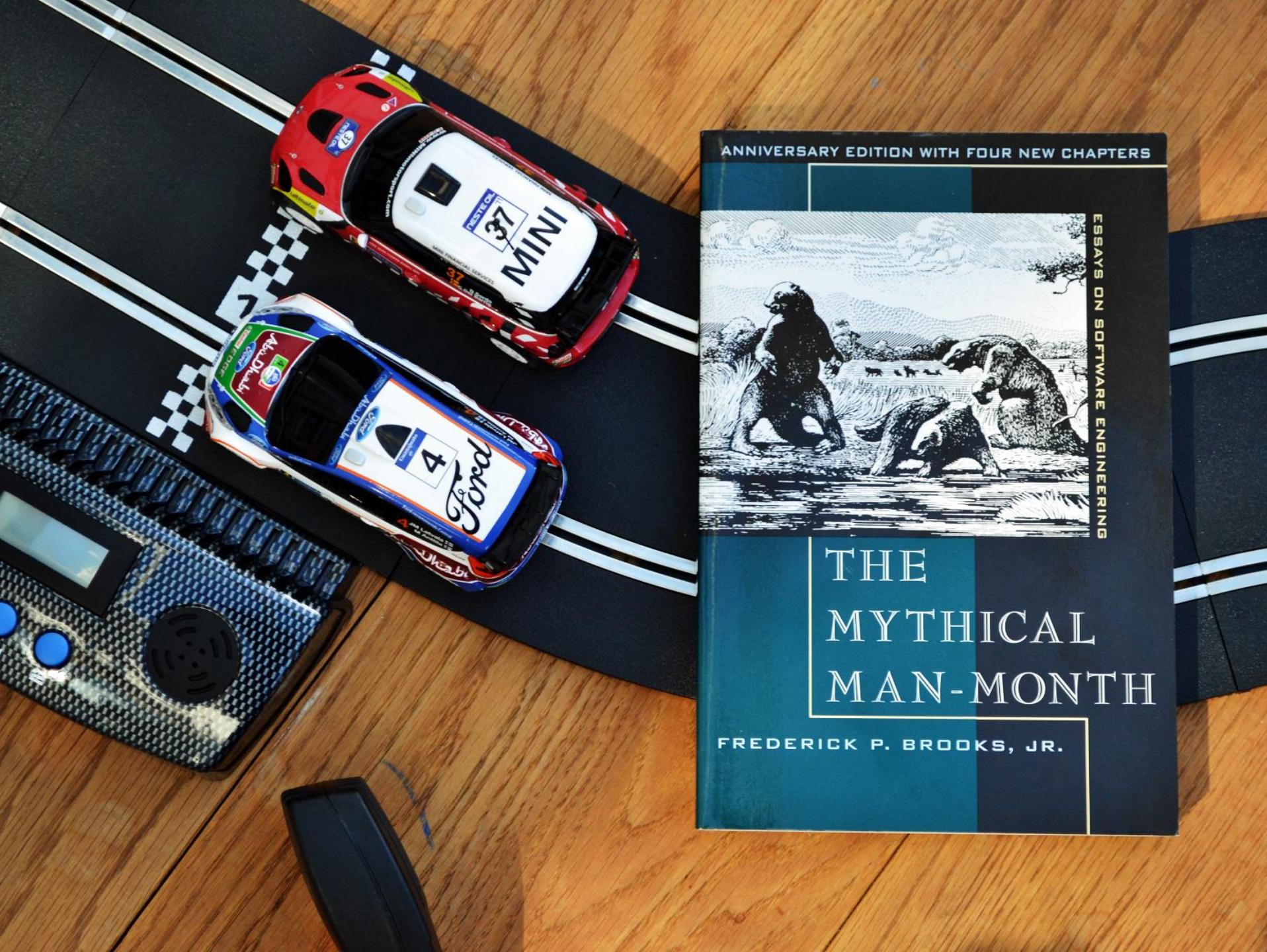
```
def roman(number)
{
    result = ''
    while (number >= 1000)
    {
        result += 'M'
        number -= 1000
    }
    if (number >= 900)
    {
        result += 'CM'
        number -= 900
    }
    if (number >= 500)
    {
        result += 'D'
        number -= 500
    }
    if (number >= 400)
    {
        result += 'CD'
        number -= 400
    }
    while (number >= 100)
    {
        result += 'C'
        number -= 100
    }
    if (number >= 90)
    {
        result += 'XC'
        number -= 90
    }
    if (number >= 50)
    {
        result += 'L'
        number -= 50
    }
    if (number >= 40)
    {
        result += 'XL'
        number -= 40
    }
    while (number >= 10)
    {
        result += 'X'
        number -= 10
    }
    if (number >= 9)
    {
        result += 'IX'
        number -= 9
    }
    if (number >= 5)
    {
        result += 'V'
        number -= 5
    }
    if (number >= 4)
    {
        result += 'IV'
        number -= 4
    }
    while (number >= 1)
    {
        result += 'I'
        number -= 1
    }
    result
}
```

```
def roman(number)
{
    result = ''
    multiples =
    [
        [1000, 'M'], [900, 'CM'],
        [500, 'D'], [400, 'CD'],
        [100, 'C'], [90, 'XC'],
        [50, 'L'], [40, 'XL'],
        [10, 'X'], [9, 'IX'],
        [5, 'V'], [4, 'IV'],
        [1, 'I']
    ]
    for (multiple in multiples)
    {
        (value, letters) = multiple
        result += letters * (number / value)
        number %= value
    }
    result
}
```

Algorithms +
Data Structures =
Programs

Niklaus Wirth

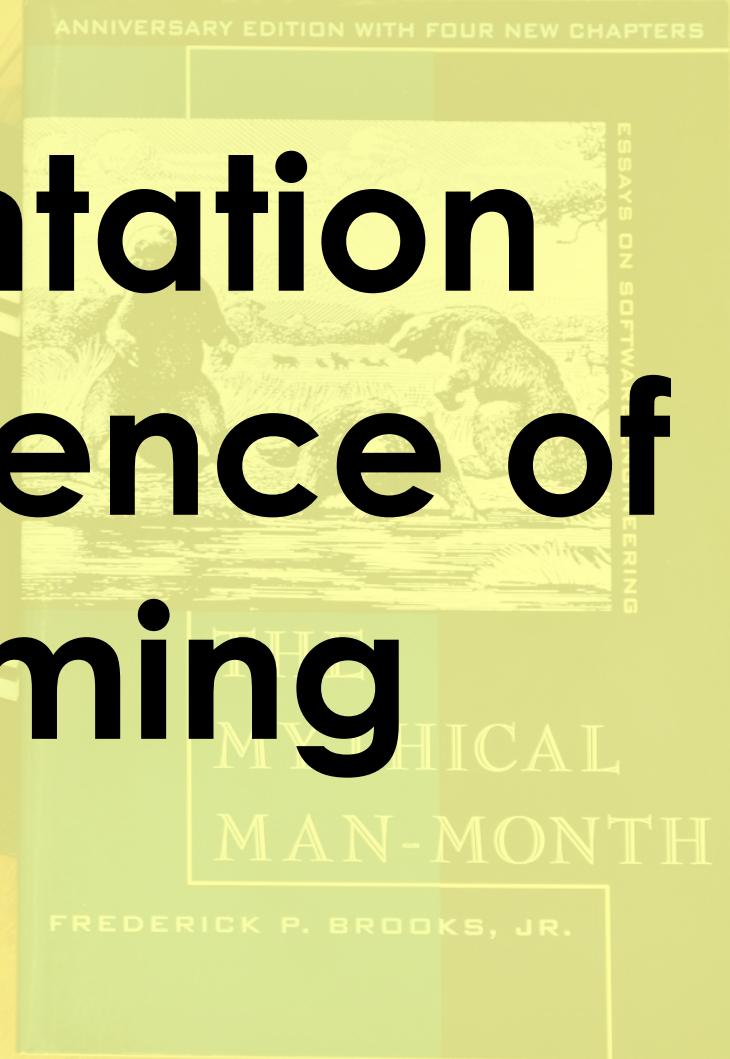
Data Structures =
Programs

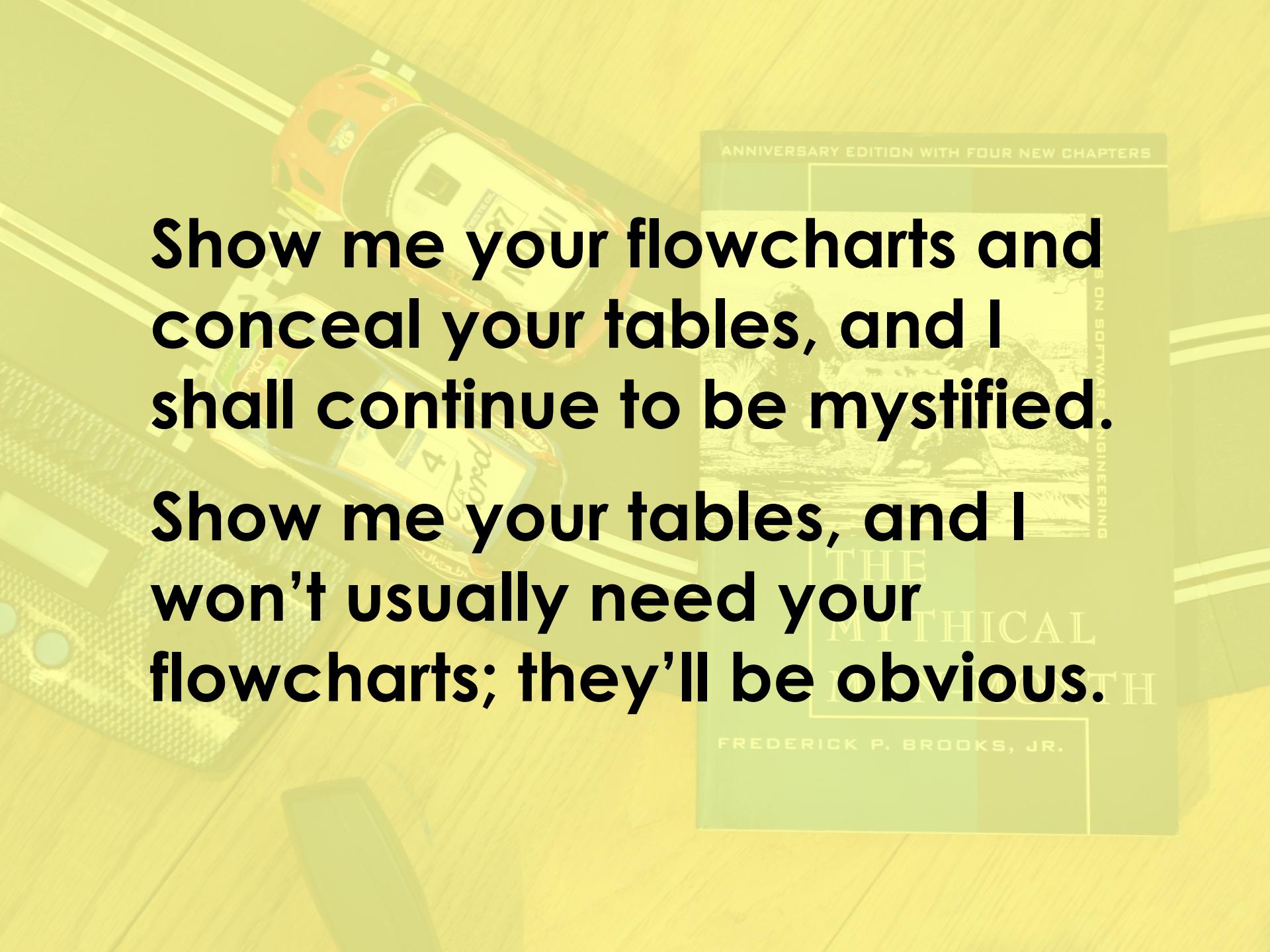


THE MYTHICAL MAN-MONTH

FREDERICK P. BROOKS, JR.

Representation Is the Essence of Programming

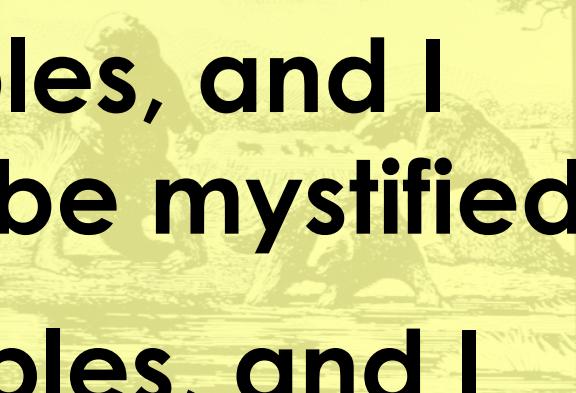




Show me your flowcharts and conceal your tables, and I shall continue to be mystified.

Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS



THE
MYTHICAL
MAN-MONTH

FREDERICK P. BROOKS, JR.

**Excel is the world's
most popular
functional language.**

Simon Peyton-Jones

```
def roman(number)
{
    result = ''
    multiples =
    [
        [1000, 'M'], [900, 'CM'],
        [500, 'D'], [400, 'CD'],
        [100, 'C'], [90, 'XC'],
        [50, 'L'], [40, 'XL'],
        [10, 'X'], [9, 'IX'],
        [5, 'V'], [4, 'IV'],
        [1, 'I']
    ]
    for (multiple in multiples)
    {
        (value, letters) = multiple
        result += letters * (number / value)
        number %= value
    }
    result
}
```

```
def roman(number)
{
    result = ''
    [
        [1000, 'M'], [900, 'CM'],
        [500, 'D'], [400, 'CD'],
        [100, 'C'], [90, 'XC'],
        [50, 'L'], [40, 'XL'],
        [10, 'X'], [9, 'IX'],
        [5, 'V'], [4, 'IV'],
        [1, 'I']
    ]
    .each
    {
        value, letters ->
        result += letters * (number / value)
        number %= value
    }
    result
}
```

```
def roman(number)
{
    [
        [1000, 'M'], [900, 'CM'],
        [500, 'D'], [400, 'CD'],
        [100, 'C'], [90, 'XC'],
        [50, 'L'], [40, 'XL'],
        [10, 'X'], [9, 'IX'],
        [5, 'V'], [4, 'IV'],
        [1, 'I']
    ]
    .inject(['', number])
    {
        injected, multiple ->
        (result, input) = injected
        (value, letters) = multiple
        [result + letters * (input / value), input % value]
    }[0]
}
```

```
def roman(number)
{
    ('I' * number)
    .replace('IIII', 'V')
    .replace('IIII', 'IV')
    .replace('VV', 'X')
    .replace('VIV', 'IX')
    .replace('XXXXX', 'L')
    .replace('XXXX', 'XL')
    .replace('LL', 'C')
    .replace('LXL', 'XC')
    .replace('CCCCC', 'D')
    .replace('CCCC', 'CD')
    .replace('DD', 'M')
    .replace('DCD', 'CM')
}
```

```
private static String roman(int n)
{
    return
        join("", nCopies(n, "I"))
        .replace("IIII", "V")
        .replace("IIII", "IV")
        .replace("VV", "X")
        .replace("VIV", "IX")
        .replace("XXXXX", "L")
        .replace("XXXX", "XL")
        .replace("LL", "C")
        .replace("LXL", "XC")
        .replace("CCCCC", "D")
        .replace("CCCC", "CD")
        .replace("DD", "M")
        .replace("DCD", "CM");
}
```

```
(defn roman [number]
  (replace
    (replace
      (replace
        (replace
          (replace
            (replace
              (replace
                (replace
                  (replace
                    (replace
                      (replace
                        (replace
                          (join (repeat number
                            "IIII" "V")
                            "IIII" "IV")
                            "VV" "X")
                            "VIV" "IX")))))))))
```

```
(defn roman [number]
  (replace
    (replace
      (replace
        (replace
          (replace
            (replace
              (replace
                (replace
                  (replace
                    (replace
                      (replace
                        (join (repeat number "I")))
                      "IIII" "V")
                      "IIII" "IV")
                      "VV" "X")
                      "VIV" "IX")
                      "XXXXX" "L")
                      "XXXX" "XL")
                      "LL" "C")
                      "LXL" "XC")
                      "CCCCC" "D")
                      "CCCC" "CD"))
                      "DD" "M"))
                      "DCD" "CM"))
```

```
(defn roman [number]
  ((comp
    (fn [numerals] (replace numerals "DCD" "CM"))
    (fn [numerals] (replace numerals "DD" "M"))
    (fn [numerals] (replace numerals "CCCC" "CD"))
    (fn [numerals] (replace numerals "CCCCC" "D"))
    (fn [numerals] (replace numerals "LXL" "XC"))
    (fn [numerals] (replace numerals "LL" "C"))
    (fn [numerals] (replace numerals "XXXX" "XL"))
    (fn [numerals] (replace numerals "XXXXX" "L"))
    (fn [numerals] (replace numerals "VIV" "IX"))
    (fn [numerals] (replace numerals "VV" "X"))
    (fn [numerals] (replace numerals "IIII" "IV"))
    (fn [numerals] (replace numerals "IIIII" "V"))))
  (join (repeat number "I")))))
```

```
(defn roman [number]
  ((comp
    #(replace % "DCD" "CM")
    #(replace % "DD" "M")
    #(replace % "CCCC" "CD")
    #(replace % "CCCCC" "D")
    #(replace % "LXL" "XC")
    #(replace % "LL" "C")
    #(replace % "XXXX" "XL")
    #(replace % "XXXXX" "L")
    #(replace % "VIV" "IX")
    #(replace % "VV" "X")
    #(replace % "IIII" "IV")
    #(replace % "IIIII" "V"))
   (join (repeat number "I"))))
```

```
(defn roman [number]
  (->
    (join (repeat number "I")))
    (.replace "IIII" "V")
    (.replace "IIII" "IV")
    (.replace "VV" "X")
    (.replace "VIV" "IX")
    (.replace "XXXXX" "L")
    (.replace "XXXX" "XL")
    (.replace "LL" "C")
    (.replace "LXL" "XC")
    (.replace "CCCCC" "D")
    (.replace "CCCC" "CD")
    (.replace "DD" "M")
    (.replace "DCD" "CM"))))
```

Concatenative programming is so called because it uses function *composition* instead of function *application*—a non-concatenative language is thus called *applicative*.

Jon Purdy

[http://evincarofautumn.blogspot.com/2012/02/
why-concatenative-programming-matters.html](http://evincarofautumn.blogspot.com/2012/02/why-concatenative-programming-matters.html)

$$f(g(h(x))))$$

$$(f\circ g\circ h)(x)$$

x h gf

h | *g* | *f*

Concatenative programming is so called because it uses function *composition* instead of function *application*—a non-concatenative language is thus called *applicative*.

This is the basic reason Unix pipes are so powerful: they form a rudimentary string-based concatenative programming language.

Jon Purdy

[http://evincarofautumn.blogspot.com/2012/02/
why-concatenative-programming-matters.html](http://evincarofautumn.blogspot.com/2012/02/why-concatenative-programming-matters.html)

Pipes and Filters

Divide the application's task into several self-contained data processing steps and connect these steps to a data processing pipeline via intermediate data buffers.

HOLY GOD
WILL BRING
JUDGMENT
DAY ON
MAY 21, 2011

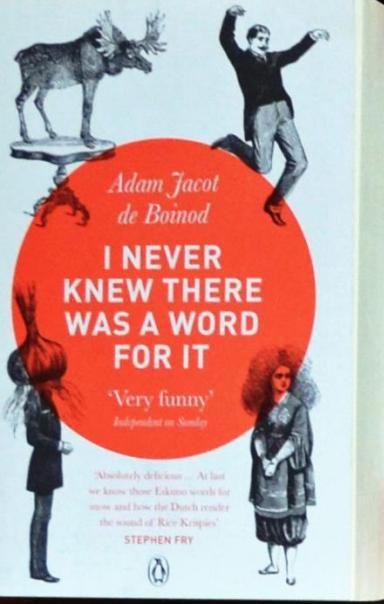
CRY MIGHTILY UNTO
GOD FOR MERCY SEE
PSALMS - 51:
JONAH - 3:

2. **Rhetoric**: Repetition for vehemence or fullness.

lindrome (n) 'running back again' Words, phrases, sentences, etc., which read the same forwards and backwards. Richard A. Lanham *Adam, I'm Adam*. I'm Adam.' A palindrome would seem to represent the compressed name of a Chiasmus.

epigram – Laudatio.

formal and ornate praise of person or deed. See *Rhetoric*: The branches in chapter 2.

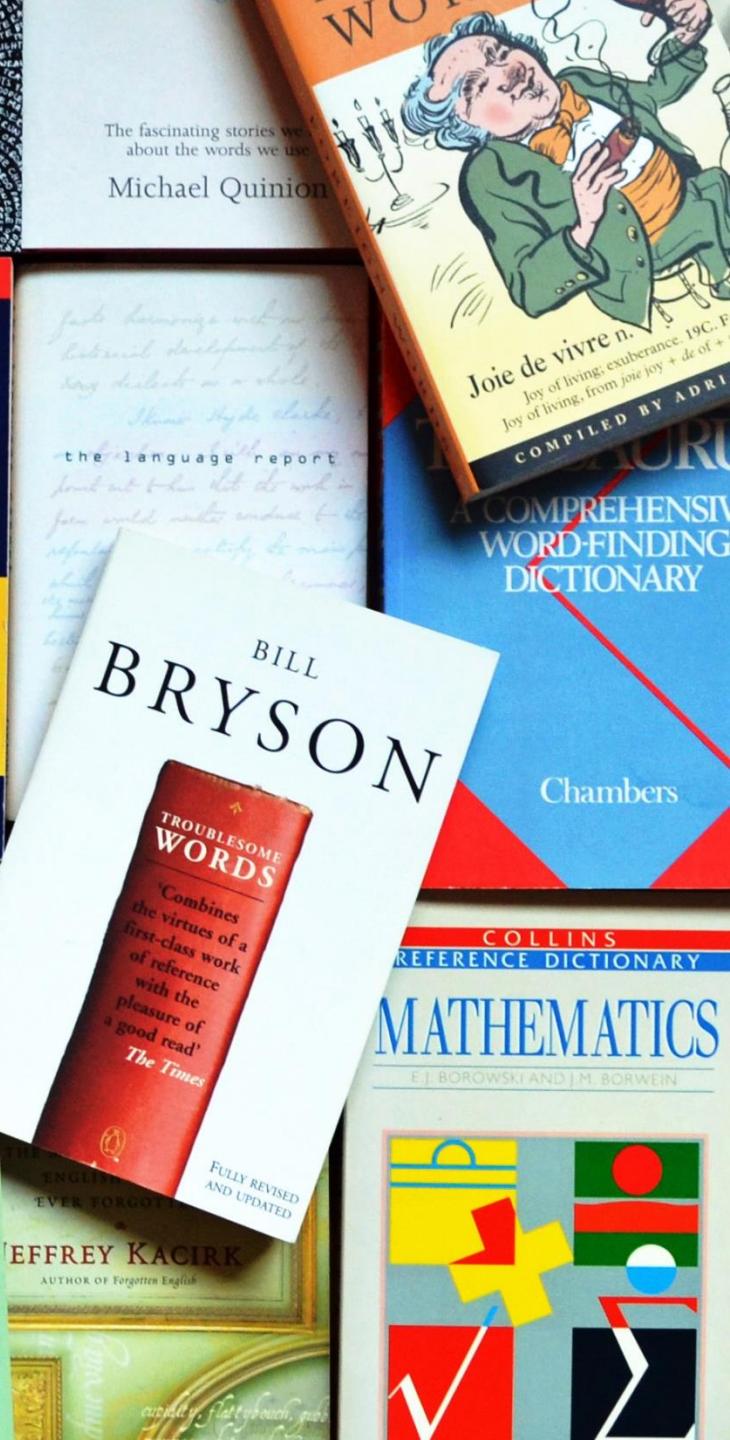


/ WordFriday



The fascinating stories we tell
about the words we use

Michael Quinion



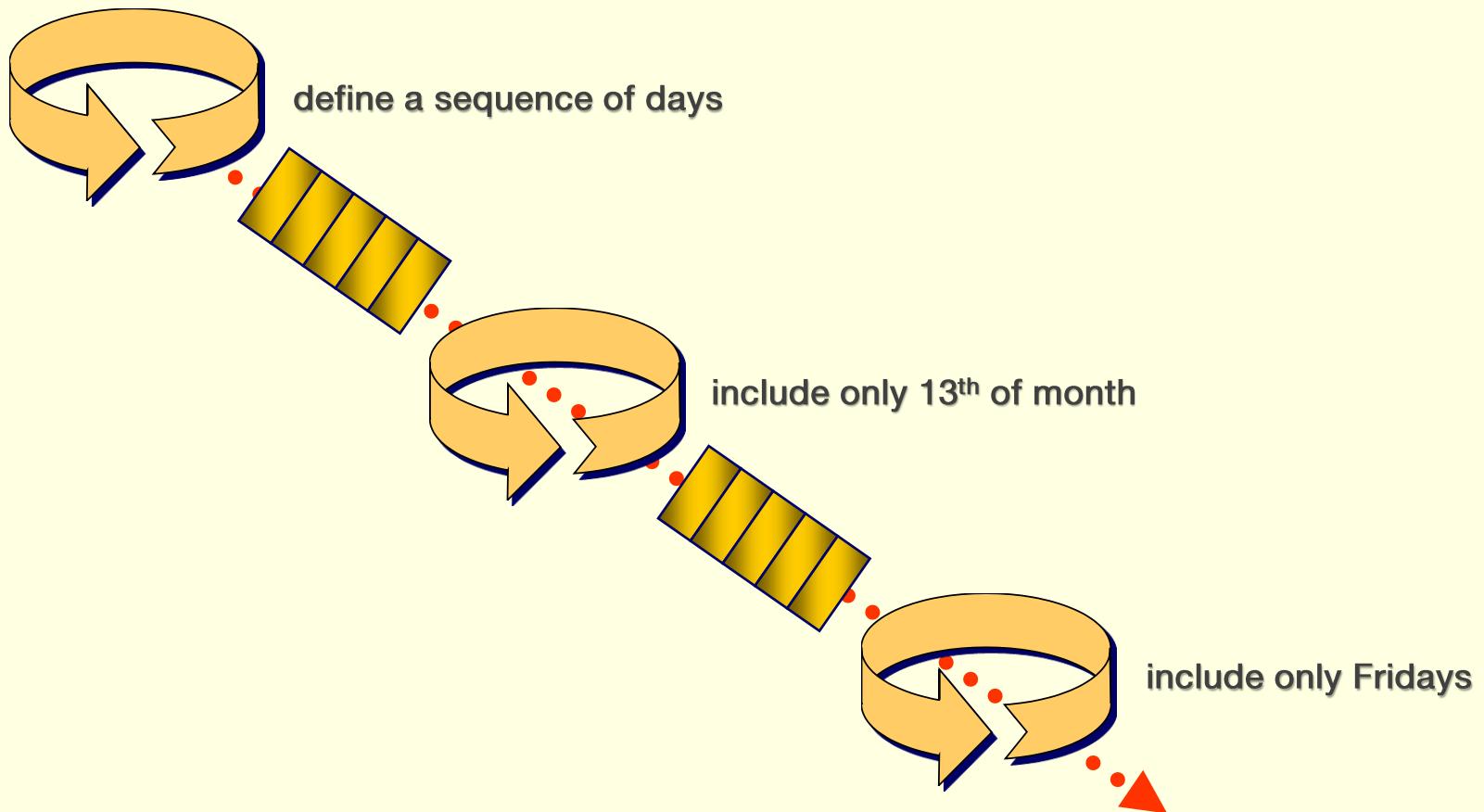
paraskevidekatriaphobia, noun

- The superstitious fear of Friday 13th.
- Contrary to popular myth, this superstition is relatively recent (19th century) and did not originate during or before the medieval times.
- Paraskevidekatriaphobia also reflects a particularly egocentric attributional bias: the universe is prepared to rearrange causality and probability around the believer based on an arbitrary and changeable calendar system, in a way that is sensitive to geography, culture and time zone.

```
struct tm next_friday_13th(struct tm when)
{
    enum { daily_secs = 24 * 60 * 60 };
    time_t seconds = mktime(&when);
    do
    {
        seconds += daily_secs;
        when = *localtime(&seconds);
    }
    while (when.tm_mday != 13 || when.tm_wday != 5);
    return when;
}
```

```
std::find_if(
    ++begin, day_iterator(),
[] (const std::tm & day)
{
    return day.tm_mday == 13 && day.tm_wday == 5;
});
```

```
var friday13ths =  
    from day in Days.After(start)  
    where day.Day == 13  
    where day.DayOfWeek == DayOfWeek.Friday  
    select day;  
  
foreach(var irrationalBelief in friday13ths)  
{  
    ...  
}
```



```
function NextFriday13thAfter($from) {  
    (1..500) |  
    %{$from.AddDays($_)} |  
    ?{ $_.Day -eq 13} |  
    ?{ $_.DayOfWeek -eq [DayOfWeek]::Friday } |  
    select -first 1  
}
```

Go with
the flow.

Queens of the Stone Age

Java 8 Streams Cheat Sheet

Definitions

- A stream **is** a pipeline of functions that can be evaluated.
- Streams **can** transform data.
- A stream **is not** a data structure.
- Streams **cannot** mutate data.

Intermediate operations

- Always return streams.
- Lazily executed.

Common examples include:

Function	Preserves count	Preserves type	Preserves order
map	✓	✗	✓
filter	✗	✓	✓
distinct	✗	✓	✓
sorted	✓	✓	✗
peek	✓	✓	✓

Stream examples

Get the unique surnames in uppercase of the first 15 book authors that are 50 years old or over.

```
library.stream()  
    .map(book -> book.getAuthor())  
    .filter(author -> author.getAge() >= 50)  
    .limit(15)  
    .map(Author::getSurname)  
    .map(String::toUpperCase)  
    .distinct()  
    .collect(toList());
```

Compute the sum of ages of all female authors younger than 25.

```
library.stream()  
    .map(Book::getAuthor)  
    .filter(a -> a.getGender() == Gender.FEMALE)  
    .map(Author::getAge)  
    .filter(age -> age < 25)  
    .reduce(0, Integer::sum);
```

Terminal operations

- Return concrete types or produce a side effect.
- Eagerly executed.

Common examples include:

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach	side effect	to perform a side effect on elements

Parallel streams

Parallel streams use the common ForkJoinPool for threading.

```
library.parallelStream()...
```

or intermediate operation:

```
IntStream.range(1, 10).parallel()...
```

Useful operations

Grouping:

```
library.stream().collect(  
    groupingBy(Book::getGenre));
```

Stream ranges:

```
IntStream.range(0, 20)...
```

Infinite streams:

```
IntStream.iterate(0, e -> e + 1)...
```

Max/Min:

```
IntStream.range(1, 10).max();
```

FlatMap:

```
twitterList.stream()  
    .map(member -> member.getFollowers())  
    .flatMap(followers -> followers.stream())  
    .collect(toList());
```

Pitfalls

- Don't update shared mutable variables i.e.

```
List<Book> myList = new ArrayList<>();  
library.stream().forEach  
(e -> myList.add(e));
```
- Avoid blocking operations when using parallel streams.

Java 8 Streams Cheat Sheet

Definitions

- A stream **is** a pipeline of functions that can be evaluated.
- Streams **can** transform data.
- A stream **is not** a data structure.
- Streams **cannot** mutate data.

Intermediate operations

- Always return streams.
- Lazily executed.

Common examples include:

Function	Preserves count	Preserves type	Preserves order
map	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
filter	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
distinct	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
sorted	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
peek	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Stream examples

Get the unique surnames in uppercase of the first 15 book authors that are 50 years old or over.

```
library.stream()  
    .map(book -> book.getAuthor())  
    .filter(author -> author.getAge() >= 50)  
    .map(Author::getSurname)  
    .map(String::toUpperCase)  
    .distinct()  
    .limit(15)  
    .collect(toList());
```

Compute the sum of ages of all female authors younger than 25.

```
library.stream()  
    .map(Book::getAuthor)  
    .filter(a -> a.getGender() == Gender.FEMALE)  
    .map(Author::getAge)  
    .filter(age -> age < 25)  
    .reduce(0, Integer::sum);
```

Terminal operations

- Return concrete types or produce a side effect.
- Eagerly executed.

Common examples include:

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach	side effect	to perform a side effect on elements

Parallel streams

Parallel streams use the common ForkJoinPool for threading.

```
library.parallelStream()...
```

or intermediate operation:

```
IntStream.range(1, 10).parallel()...
```

Useful operations

Grouping:

```
library.stream().collect(  
    groupingBy(Book::getGenre));
```

Stream ranges:

```
IntStream.range(0, 20)...
```

Infinite streams:

```
IntStream.iterate(0, e -> e + 1)...
```

Max/Min:

```
IntStream.range(1, 10).max();
```

FlatMap:

```
twitterList.stream()  
    .map(member -> member.getFollowers())  
    .flatMap(followers -> followers.stream())  
    .collect(toList());
```

Pitfalls

- Don't update shared mutable variables i.e.

```
List<Book> myList = new ArrayList<>();  
library.stream().forEach  
    (e -> myList.add(e));
```

- Avoid blocking operations when using parallel streams.

```
// Get the unique surnames in uppercase of the  
// first 15 book authors that are 50 years old  
// or older?
```

```
library.stream()  
    .map(book -> book.getAuthor())  
    .filter(author -> author.getAge() >= 50)  
    .limit(15)  
    .map(Author::getSurname)  
    .map(String::toUpperCase)  
    .distinct()  
    .collect(toList()))
```

```
// Get the first 15 unique surnames in  
// uppercase of the book authors that are 50  
// years old or older.
```

```
library.stream()  
    .map(book -> book.getAuthor())  
    .filter(author -> author.getAge() >= 50)  
    .map(Author::getSurname)  
    .map(String::toUpperCase)  
    .distinct()  
    .limit(15)  
    .collect(toList()))
```

```
// Get the unique surnames in uppercase of the  
// first 15 book authors that are 50 years old  
// or older.
```

```
library.stream()  
    .map(book -> book.getAuthor())  
    .filter(author -> author.getAge() >= 50)  
    .distinct()  
    .limit(15)  
    .map(Author::getSurname)  
    .map(String::toUpperCase)  
    .distinct()  
    .collect(toList()))
```

**Simple filters that can be arbitrarily
chained are more easily re-used,
and more robust, than almost any
other kind of code.**

Brandon Rhodes

<http://rhodesmill.org/brandon/slides/2012-11-pyconca/>

```
// Get the unique surnames in uppercase of the  
// first 15 book authors that are 50 years old  
// or older?
```

```
List<Author> authors = new ArrayList<Author>();  
for (Book book : library)  
{  
    Author author = book.getAuthor();  
    if (author.getAge() >= 50)  
    {  
        authors.add(author);  
        if (authors.size() == 15)  
            break;  
    }  
}  
List<String> result = new ArrayList<String>();
```

```
// Get the unique surnames in uppercase of the
// first 15 book authors that are 50 years old
// or older?

List<Author> authors = new ArrayList<Author>();
for (Book book : library)
{
    Author author = book.getAuthor();
    if (author.getAge() >= 50)
    {
        authors.add(author);
        if (authors.size() == 15)
            break;
    }
}
List<String> result = new ArrayList<String>();
for(Author author : authors)
{
    String name = author.getSurname().toUpperCase();
    if (!result.contains(name))
        result.add(name);
}
```

```
// Get the first 15 unique surnames in
// uppercase of the book authors that are 50
// years old or older.

List<String> result = new ArrayList<String>();
for (Book book : library)
{
    Author author = book.getAuthor();
    if (author.getAge() >= 50)
    {
        String name = author.getSurname().toUpperCase();
        if (!result.contains(name))
        {
            result.add(name);
            if (result.size() == 15)
                break;
        }
    }
}
```

```
// Get the unique surnames in uppercase of the
// first 15 book authors that are 50 years old
// or older.

List<Author> authors = new ArrayList<Author>();
for (Book book : library)
{
    Author author = book.getAuthor();
    if (author.getAge() >= 50 && !authors.contains(author))
    {
        authors.add(author);
        if (authors.size() == 15)
            break;
    }
}
List<String> result = new ArrayList<String>();
for(Author author : authors)
{
    String name = author.getSurname().toUpperCase();
    if (!result.contains(name))
        result.add(name);
}
```

Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes when it becomes necessary to massage data in another way.
This is the way of IO also.
2. Our loader should be able to do link-loading and controlled establishment.
3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.
4. It should be possible to get private system components (all routines are system components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

The program Bentley asked Knuth to write is one that's become familiar to people who use languages with serious text-handling capabilities: Read a file of text, determine the n most frequently used words, and print out a sorted list of those words along with their frequencies.

Knuth wrote his program in WEB, a literate programming system of his own devising that used Pascal as its programming language. His program used a clever, purpose-built data structure for keeping track of the words and frequency counts; and the article interleaved with it presented the program lucidly.

```
tr -cs A-Za-z '\n' |  
tr A-Z a-z |  
sort |  
uniq -c |  
sort -rn |  
sed ${1}q
```

If you are not a UNIX adept, you may need a little explanation, but not much, to understand this pipeline of processes. The plan is easy:

1. Make one-word lines by transliterating the complement (-c) of the alphabet into newlines (note the quoted newline), and squeezing out (-s) multiple newlines.
2. Transliterate upper case to lower case.
3. Sort to bring identical words together.
4. Replace each run of duplicate words with a single representative and include a count (-c).
5. Sort in reverse (-r) numeric (-n) order.
6. Pass through a stream editor; quit (q) after printing the number of lines designated by the script's first parameter (\${1}).

McIlroy's review is both an explanation and an exemplar of the Unix Way: small programs that do elementary tasks, but which are written so they can be combined in complex ways.

<http://www.leancrew.com/all-this/2011/12/more-shell-less-egg/>

```
$ ./roman 42
```

```
XLII
```

```
$ cat roman
```

```
printf %$1s |
tr ' ' 'I' |
sed s/AAAAA/V/g |
sed s/AAA/IV/ |
sed s/VV/X/g |
sed s/VV/IX/ |
sed s/AAAAA/L/g |
sed s/AAA/XL/ |
sed s/LL/C/g |
sed s/LXL/XC/ |
sed s/CCCC/D/g |
sed s/CCC/CD/ |
sed s/DD/M/g |
sed s/DCD/CM/
echo
```

```
$ ./roman 42
```

```
XLII
```

```
$ cat roman
```

```
printf %$1s |
```

```
tr ' ' 'I' |
```

```
sed '
```

```
  s/IIIII/V/g
```

```
  s/IIII/IV/
```

```
  s/VV/X/g
```

```
  s/VIV/IX/
```

```
  s/XXXXXX/L/g
```

```
  s/XXXX/XL/
```

```
  s/LL/C/g
```

```
  s/LXL/XC/
```

```
  s/CCCCC/D/g
```

```
  s/CCCC/CD/
```

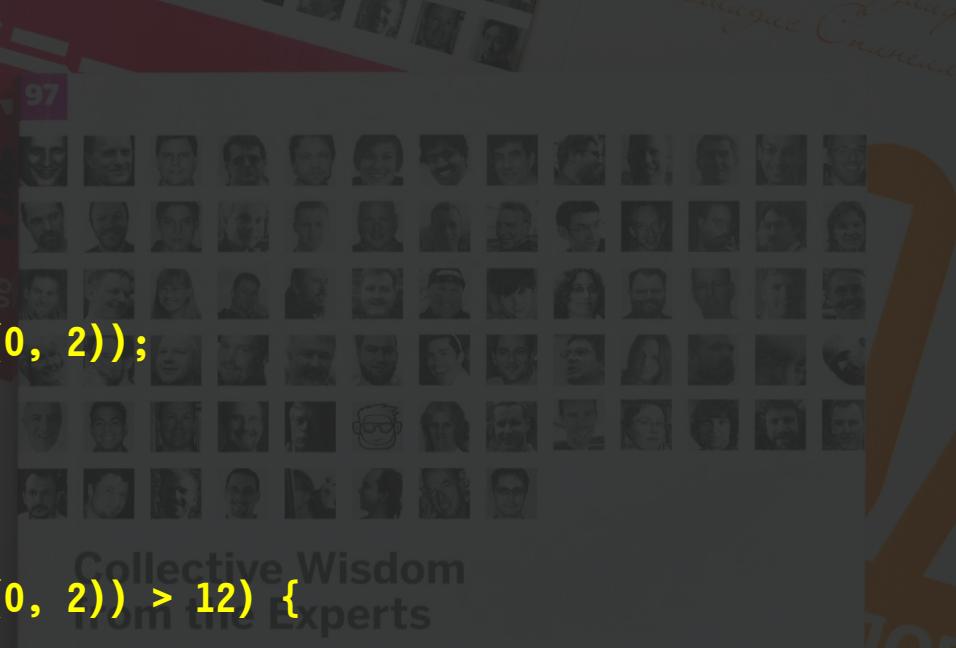
```
  s/DD/M/g
```

```
  s/DCD/CM/
```

```
'
```

```
echo
```

```
try {  
    Integer.parseInt(time.substring(0, 2));  
}  
} catch (Exception x) {  
    return false;  
}  
if (Integer.parseInt(time.substring(0, 2)) > 12) {  
    return false;  
}  
...  
if (!time.substring(9, 11).equals("AM") &  
    !time.substring(9, 11).equals("PM")) {  
    return false;  
}
```



97 Things Every Programmer Should Know

Burk Hufnagel

"Put the Mouse Down and Step Away from the Keyboard"

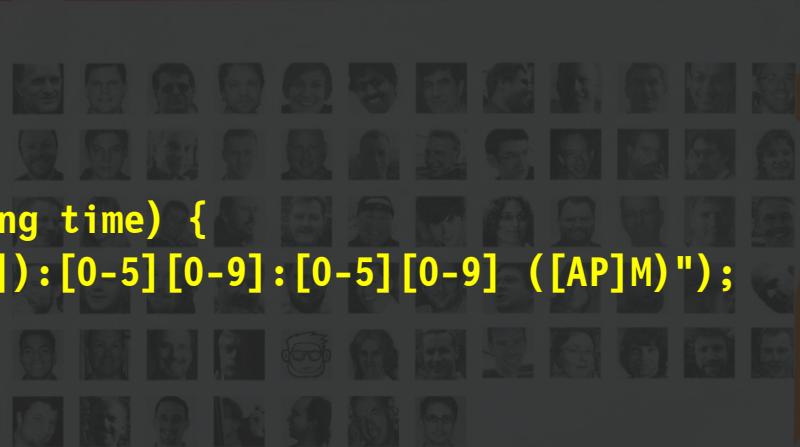
O'REILLY®

Edited by Kevlin Henney

```
public static boolean validateTime(String time) {  
    return time.matches("(0[1-9]|1[0-2]):[0-5][0-9]:[0-5][0-9] ([AP]M)");  
}
```



O'REILLY®
オライリー・ジャパン



Collective Wisdom
from the Experts

97 Things Every Programmer Should Know

Burk Hufnagel

"Put the Mouse Down and Step Away from the Keyboard"

O'REILLY®

Edited by Kevlin Henney

Computer Science in the 1960s to 80s
spent a lot of effort making languages
which were as powerful as possible.
Nowadays we have to appreciate the
reasons for picking not the most
powerful solution but the least powerful.

Tim Berners-Lee

<https://www.w3.org/DesignIssues/Principles.html>

The reason for this is that the less powerful the language, the more you can do with the data stored in that language. If you write it in a simple declarative form, anyone can write a program to analyze it in many ways.

Tim Berners-Lee

<https://www.w3.org/DesignIssues/Principles.html>

object
 {}
 { *members* }

members
 pair
 pair , members

pair
 string : value

array
 []
 [*elements*]

elements
 value
 value , elements

value
 string
 number
 object
 array
 true
 false
 null

string
 ""
 " *chars* "

chars
 char
 char chars

char
 any-non-control-char
 \\"
 \\\\
 \\V
 \\b
 \\f
 \\n
 \\r
 \\t
 \ufe0f*four-hex-digits*

number
 integer
 integer fraction
 integer exponent
 integer fraction exponent

integer
 digit
 non-zero-digit digits
 - *digit*
 - *non-zero-digit digits*

fraction
 . *digits*

exponent
 e *digits*

e
 e
 e+
 e-
 E
 E+
 E-

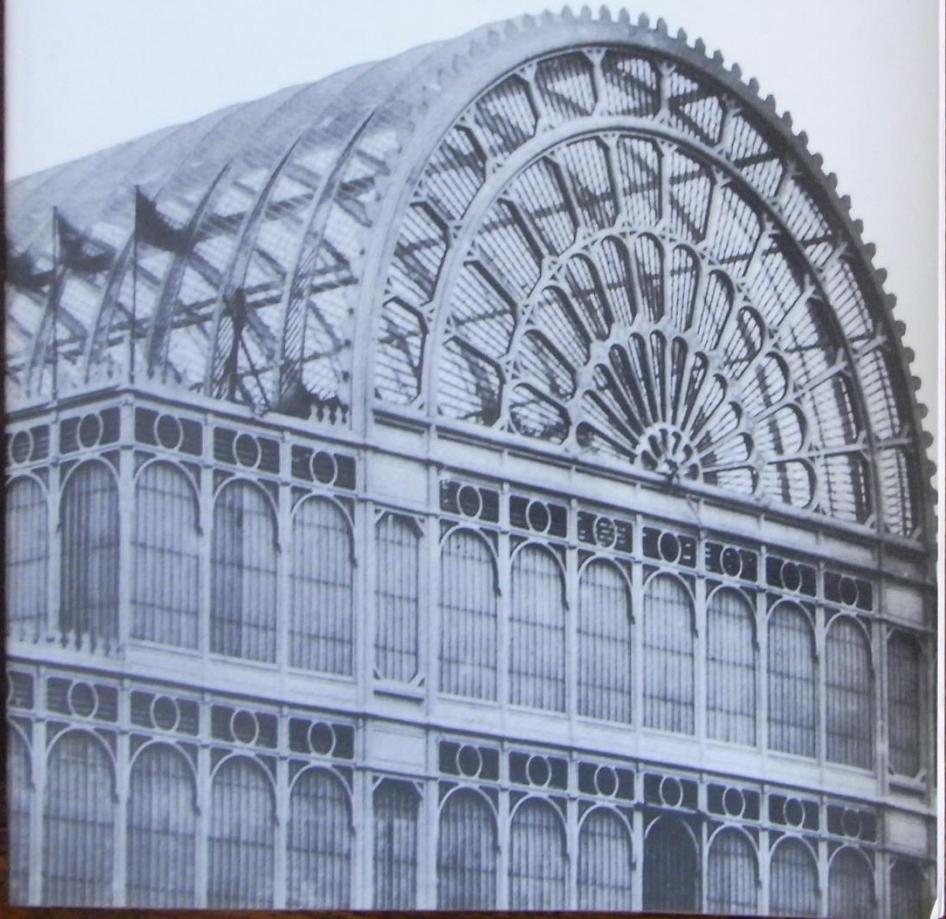
```
group ::=  
  '(' expression ')' '  
factor ::=  
  integer | group  
term ::=  
  factor (('*' factor) | ('/' factor))*  
expression ::=  
  term (('+' term) | ('-' term))*
```

```
group =
  '(' >> expression >> ')';
factor =
  integer | group;
term =
  factor >> *(('*' >> factor) | ('/' >> factor));
expression =
  term >> *(('+' >> term) | ('-' >> term));
```

SOFTWARE ARCHITECTURE

PERSPECTIVES ON AN EMERGING DISCIPLINE

MARY SHAW DAVID GARLAN



Event-Based, Implicit Invocation

The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events.

When the event is announced, the system itself invokes all of the procedures that have been registered for the event. Thus an announcement "implicitly" causes the invocation of procedures in other modules.

```
$0 % 3 == 0 { printf "Fizz" }
$0 % 5 == 0 { printf "Buzz" }
$0 % 3 != 0 && $0 % 5 != 0 { printf $0
                                { printf "\n" }
```

```
echo {1..100} | tr ' ' '\n' | awk '
$0 % 3 == 0           { printf "Fizz" }
$0 % 5 == 0           { printf "Buzz" }
$0 % 3 != 0 && $0 % 5 != 0 { printf $0
                                { printf "\n" }
'
```

```
echo {1..100} | tr ' ' '\n' | awk '
$0 % 3 == 0           { printf "Fizz" }
$0 % 5 == 0           { printf "Buzz" }
$0 % 3 != 0 && $0 % 5 != 0 { printf $0
                             { printf "\n"
'
| diff - expected && echo Pass
```



Richard Dalton
@richardadalton

FizzBuzz was invented to avoid the awkwardness of realising that nobody in the room can binary search an array.

11:29 AM - 24 Apr 2015



9



9

<https://twitter.com/richardadalton/status/591534529086693376>

Make — A Program for Maintaining Computer Programs

S. I. Feldman

ABSTRACT

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. Make provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell Make that any part of the program, the operations can be done. Whenever a change is made in correctly, and with a minimum amount of effort, Make will create the proper files simply.

The basic operation of Make is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; Make does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

August 15, 1978

The makefile language is similar to declarative programming. This class of language, in which necessary end conditions are described but the order in which actions are to be taken is not important, is sometimes confusing to programmers used to imperative programming.

August 15, 1978

[http://en.wikipedia.org/wiki/Make_\(software\)](http://en.wikipedia.org/wiki/Make_(software))

Many programming languages support programming in both functional and imperative style but the syntax and facilities of a language are typically optimised for only one of these styles, and social factors like coding conventions and libraries often force the programmer towards one of the styles.

https://wiki.haskell.org/Functional_programming

```
squares = []
i = 1
while i < 101:
    squares.append(i**2)
    i += 1
```

```
squares = []
for i in range(1, 101):
    squares.append(i**2)
```

```
squares = [i**2 for i in range(1, 101)]
```

intension, *n.* (*Logic*)

- the set of characteristics or properties by which the referent or referents of a given expression is determined; the sense of an expression that determines its reference in every possible world, as opposed to its actual reference. For example, the intension of *prime number* may be *having non-trivial integral factors*, whereas its **extension** would be the set {2, 3, 5, 7, ...}.

E J Borowski and J M Borwein
Dictionary of Mathematics

$$\{ x^2 \mid x \in N, x \geq 1 \wedge x \leq 100 \}$$



select

from

where

A list comprehension is a syntactic construct available in some programming languages for creating a list based on existing lists. It follows the form of the mathematical *set-builder notation* (*set comprehension*) as distinct from the use of map and filter functions.

http://en.wikipedia.org/wiki/List_comprehension

```
[x^2 | x <- [1..100]]
```

$$\{ x^2 \mid x \in N, x \geq 1 \}$$

```
[x^2 | x <- [1..]]
```

**Lazy
evaluation**

```
take 100 [x^2 | x <- [1..]]
```

fizzes

buzzes

words

numbers

choice

fizzbuzz

```
fizzes      = cycle [ "", "", "Fizz" ]  
buzzes      = cycle [ "", "", "", "", "Buzz" ]  
words       = zipWith (++) fizzes buzzes  
numbers     = map show [1..]  
choice      = max  
fizzbuzz   = zipWith choice words numbers
```

```
fizzes      = cycle [ "", "", "Fizz"]
buzzes      = cycle [ "", "", "", "", "Buzz"]
words       = zipWith (++) fizzes buzzes
numbers     = map show [1..]
choice       = max
fizzbuzz    = zipWith choice words numbers
take 100 fizzbuzz
```

```
(def fizzes
  (cycle ["" "" "Fizz"]))

(def buzzes
  (cycle ["" "" "" "" "Buzz"]))

(def fizzbuzzes
  (replace {"" nil}
    (map str fizzes buzzes)))

(def numbers
  (map str (rest (range)))))

(def fizzbuzz
  (map #(or %1 %2) fizzbuzzes numbers))

(take 100 fizzbuzz)
```



William Morgan
@wm

i love functional programming. it takes smart people who would otherwise be competing with me and turns them into unemployable crazies

7:53 PM - 30 Dec 2009

↪ 1,808 ❤ 1,765

Shared memory is like a canvas where threads collaborate in painting images, except that they stand on the opposite sides of the canvas and use guns rather than brushes.

The only way they can avoid killing each other is if they shout "duck!" before opening fire.

Bartosz Milewski

"Functional Data Structures and Concurrency in C++"

<http://bartoszmilewski.com/2013/12/10/functional-data-structures-and-concurrency-in-c/>

Some people, when confronted with a problem, think, "I know, I'll use threads," and then two they have problems.

Ned Batchelder

<https://twitter.com/#!/nedbat/status/194873829825327104>

Concurrency

Concurrency

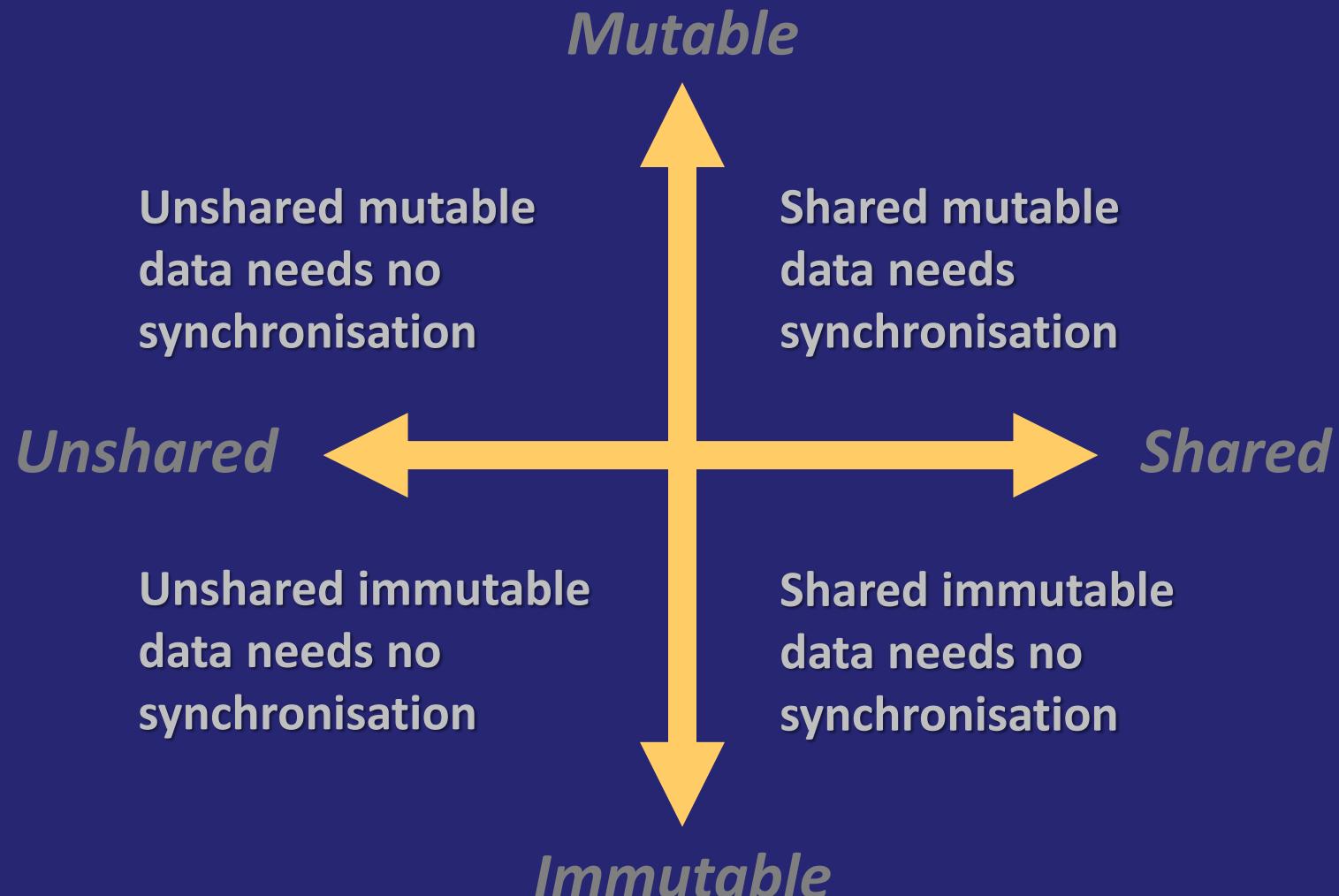
Threads

Concurrency

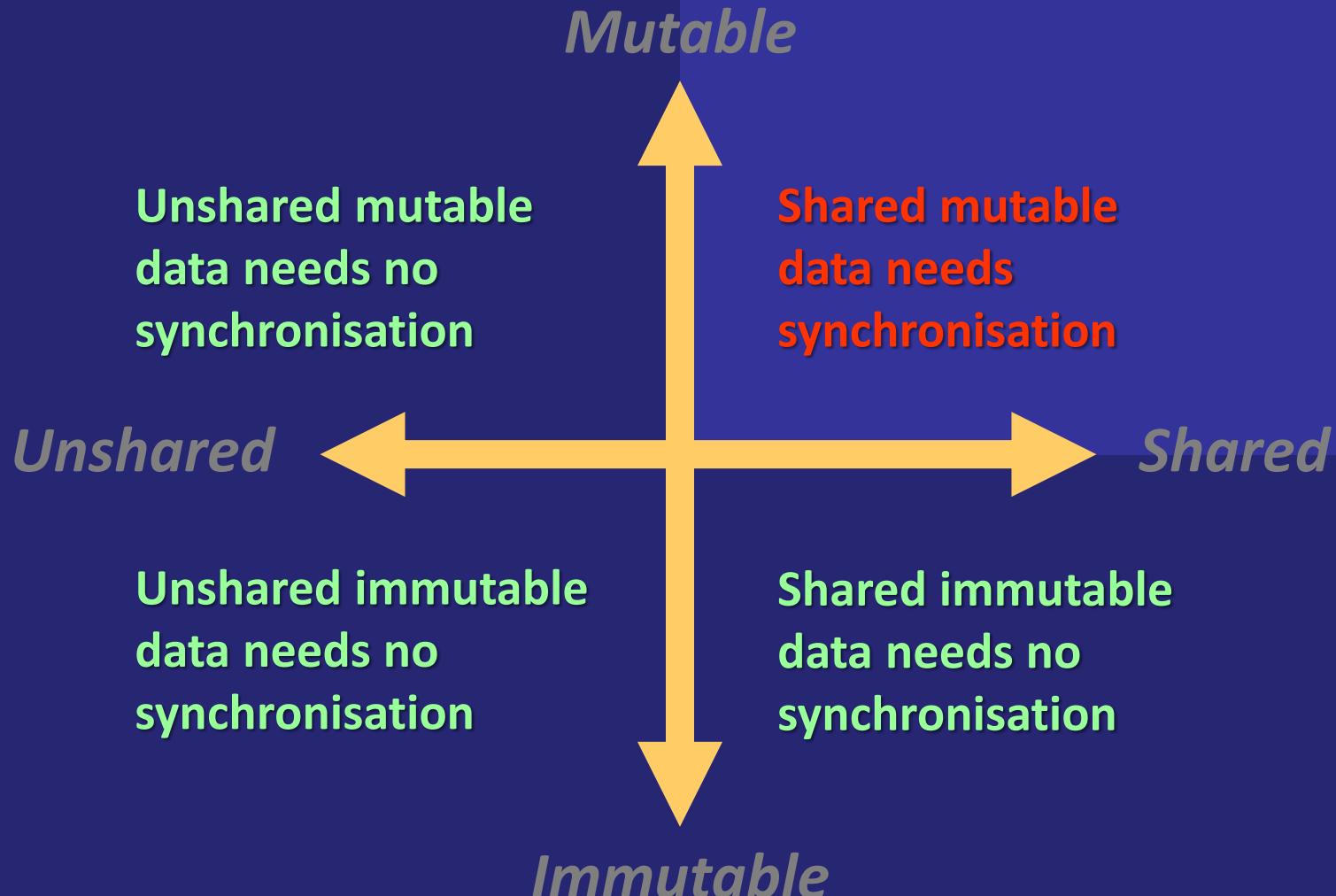
Threads

Locks

All computers
wait at the
same speed.



The Synchronisation Quadrant

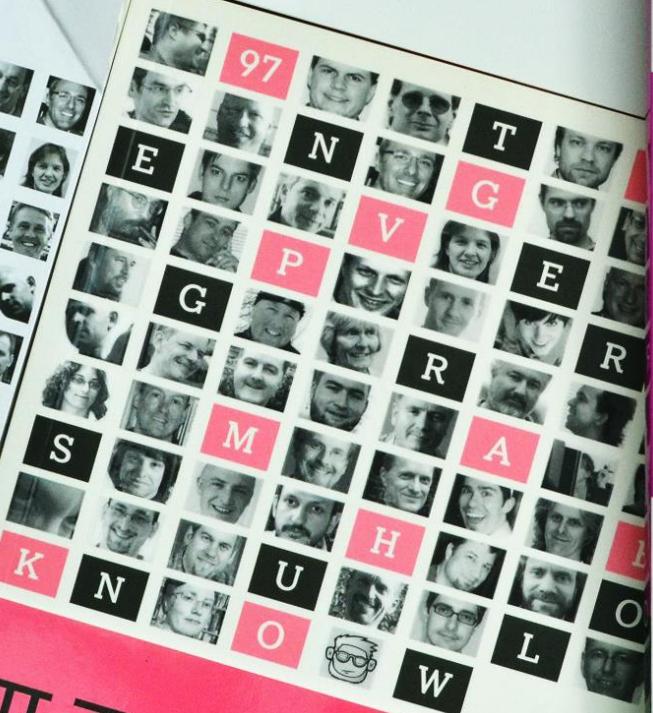


知道的
事

Kevlin Henney 编
李军泽 吕骏 审校
电子工业出版社。
http://www.phei.com.cn

ノログ
97 Things Every Prog
知るべき

O'REILLY®
オライリー・ジャパン



프로그래머가
알아야
하는
97가지

97



Collective Wisdom
from the Experts

97 Things Every Programmer Should Know

O'REILLY®

Edited by Kevlin Henney

zkušenosti
expertů z praxe

Instead of using threads and shared memory as our programming model, we can use processes and message passing. Process here just means a protected independent state with executing code, not necessarily an operating system process.

97 Things Every Programmer Should Know

Russel Winder

"Message Passing Leads to Better Scalability in Parallel Systems"

Languages such as Erlang (and occam before it) have shown that processes are a very successful mechanism for programming concurrent and parallel systems. Such systems do not have all the synchronization stresses that shared-memory, multithreaded systems have.

97 Things Every
Programmer
Should Know

Russel Winder

"Message Passing Leads to Better Scalability in Parallel Systems"

C.A.R. Hoare
**Communicating
Sequential
Processes**

C.A.R. HOARE SERIES EDITOR

```
func fizzbuzz(n int) string {
    result := ""
    if n % 3 == 0 {
        result += "Fizz"
    }
    if n % 5 == 0 {
        result += "Buzz"
    }
    if result == "" {
        result = strconv.Itoa(n)
    }
    return result
}
```

```
func fizzbuzzer(in <-chan int, out chan<- string) {  
    for n := range in {  
        out<-fizzbuzz(n)  
    }  
}
```

```
func main() {
    request := make(chan int)
    response := make(chan string)

    go fizzbuzzer(request, response)

    for i := 1; i <= 100; i++ {
        request<-i
        fmt.Println(<-response)
    }
}
```

```
(1..100) |  
%{  
    $fizzed = if($_ % 3 -eq 0) {"Fizz"}  
    $buzzed = if($_ % 5 -eq 0) {"Buzz"}  
    $fizzbuzzed = $fizzed + $buzzed  
    if($fizzbuzzed) {$fizzbuzzed} else {$_}  
}
```

To keep our C++ API boundary simple, we [...] adopted one-way data flow. The API consists of methods to perform fire-and-forget mutations and methods to compute view models required by specific views.

To keep the code understandable, we write functional style code converting raw data objects into immutable view models by default. As we identified performance bottlenecks through profiling, we added caches to avoid recomputing unchanged intermediate results.

The resulting functional code is easy to maintain, without sacrificing performance.



Michael Feathers
@mfeathers

OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.

3:27 PM - 3 Nov 2010



235

121

<https://twitter.com/mfeathers/status/29581296216>

*Computer Systems
Series*

ABCL

*An Object-Oriented Concurrent
System*

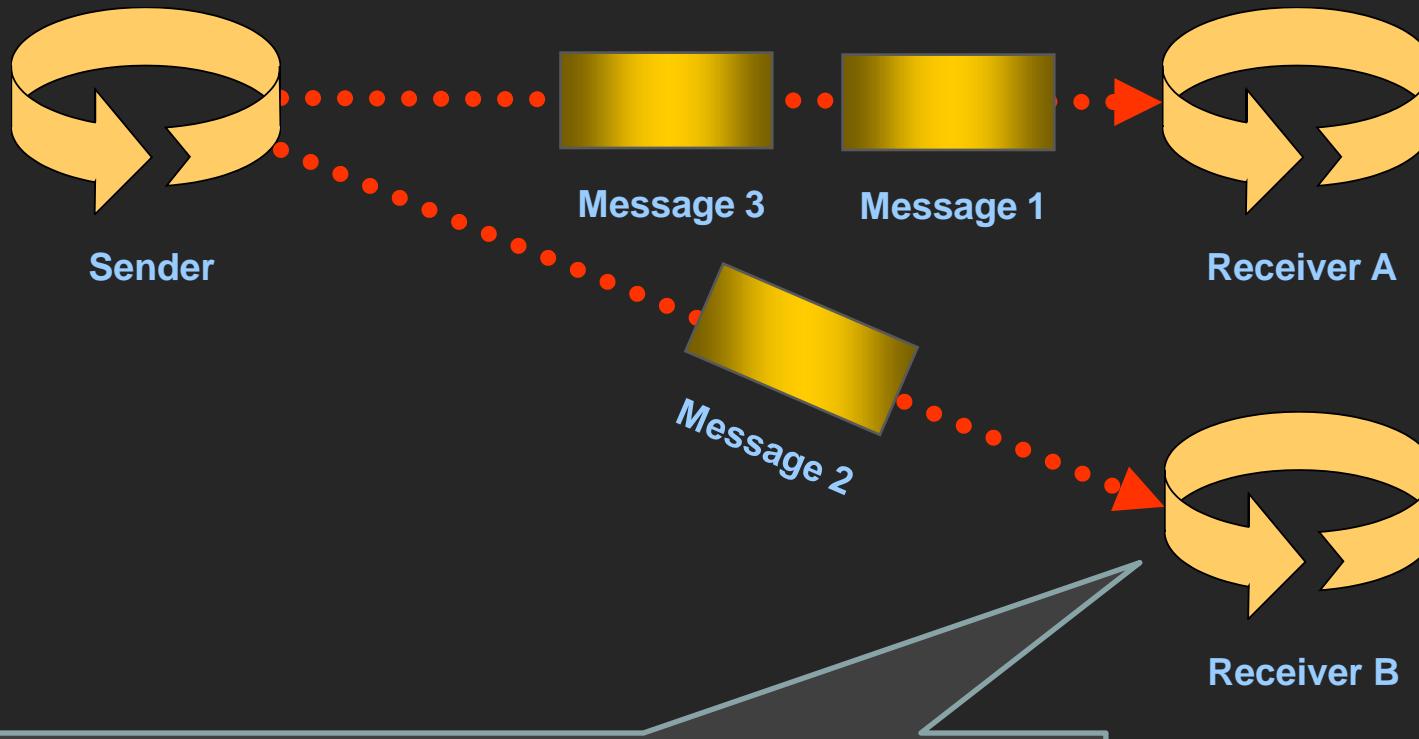
edited by Akinori Yonezawa

The MIT Press

Multithreading is just one
damn thing after, before, or
simultaneous with another.

Andrei Alexandrescu

Actor-based concurrency is
just one damn message after
another.



In response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received.

http://en.wikipedia.org/wiki/Actor_model

Stack

SANDLER

INTERNAL OBJECTS REVISITED

KARNAC
BOOKS

The Self and the Object World

Edith Jacobson

JAC M.D.

The shadow of the object

Christopher Bollas

FAB

Greenberg and Mitchell

Harvard

Object Relations in Psychoanalytic Theory

alphabet(Stack) =

{push, pop, popped, empty}

The Self and the Object World

Edith Jacobson M.D.

The shadow of the object

Christopher Bollas

Greenberg and Mitchell

Harvard

Object Relations in Psychoanalytic Theory

trace(Stack) =

{⟨ ⟩},

INTERNAL OBJECTS REVISITED

KARNAC
BOOKS

⟨push⟩,

The Self and the Object World

Edith Jacobson M.D.

⟨pop, empty⟩,

⟨push, push⟩,

⟨push, pop, popped⟩,

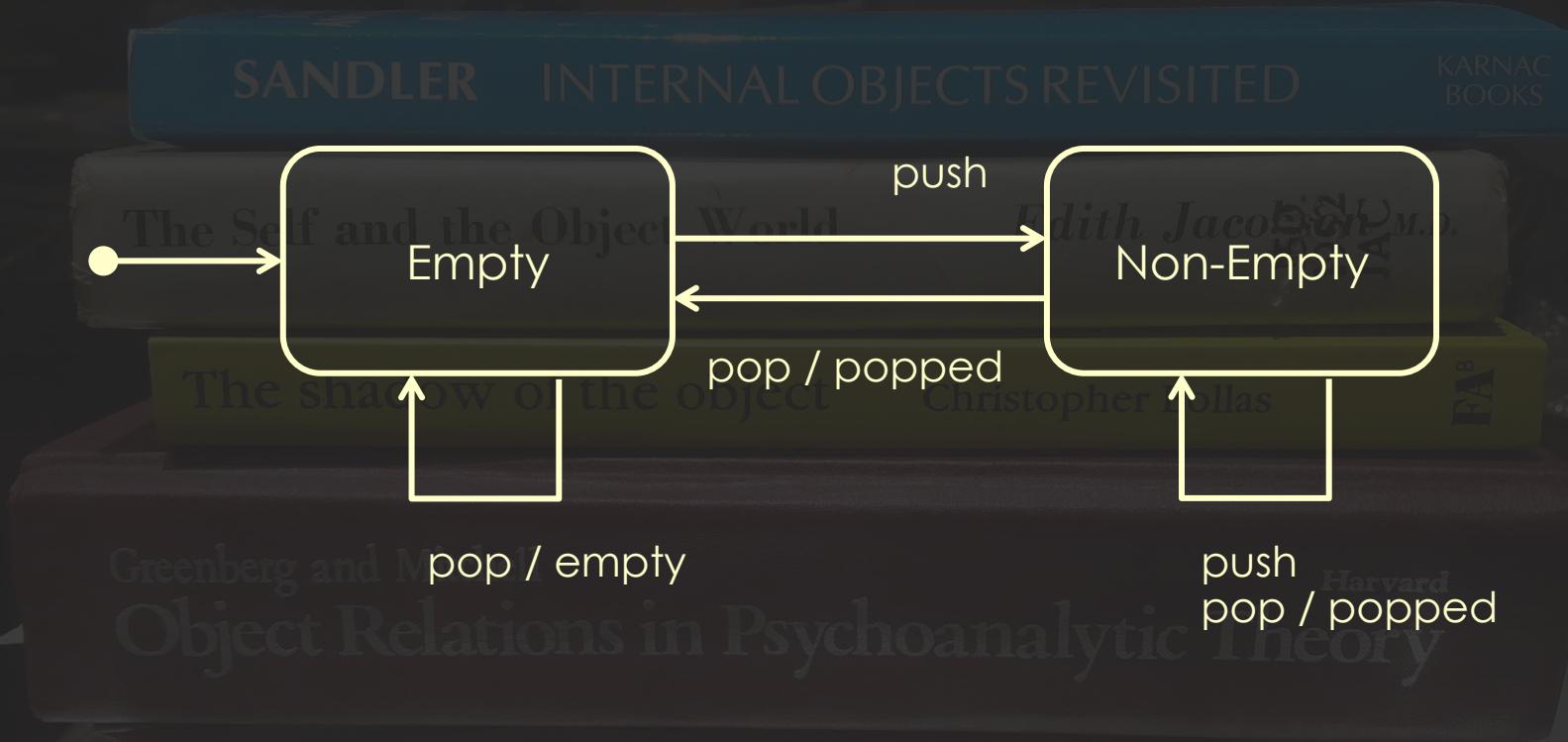
Object Relations Theory

Harvard

⟨push, push, pop, popped⟩,

⟨push, pop, popped, pop, empty⟩,

...}



empty() ->

receive

{push, Top} ->

non_empty(Top);

{pop, Return} ->

Return ! empty

end,

empty().

non_empty(Value) ->

receive

{push, Top} ->

non_empty(Top),

non_empty(Value);

{pop, Return} ->

Return ! {popped, Value}

end.

`Stack = spawn(stack, empty, []).`

`Stack ! {pop, self()}.`

`empty`

`Stack ! {push, 20}.`

`Stack ! {pop, self()}.` *Edith Jacobson M.D.*

`{popped, 20}`

`Stack ! {push, 20}.`

`Stack ! {push, 16}.`

`Stack ! {pop, self()}.` *Object Relations in Psychoanalytic Theory*

Harvard

`{popped, 16}`

`Stack ! {pop, self()}.`

`{popped, 20}`

OOP to me means only
messaging, local retention
and protection and hiding of
state-process, and extreme
late-binding of all things.

Alan Kay

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.

T S Eliot