Engineering, a Path To Science

Richard P. Gabriel
IBM Research

"I don't want to die
in a language I can't understand"

Only mystery enables us to live

–Federico Garcia Lorca

Only mystery enables us to live

Only mystery

–Federico Garcia Lorca

Engineering, Science, and Paradigms

- for us, engineering precedes science
  - ...as it does for many sciences
- a small paradigm shift in programming languages happened around 1990
  - ...we can notice it using Kuhn's notion of incommensurability

At My Age...

- interested in larger, more philosophical questions
- lived through many science and technology changes, not all for the good
- computing and software present difficult ontological problems

Wednesday, March 21, 2012

Warning!

- I use the paper "Mixin-Based Inheritance" by Gilad Bracha and William Cook to investigate the concept of ***incommensurability*** in computer science
- This is not a criticism of this paper nor of these authors
- This is a superb and very important paper
- I am using it to show how engineering and science are related

Engineering, a Path To Science

many believe science always precedes engineering

I don't

...the practical application of science to commerce or industry...the discipline dealing with the art or science of applying scientific knowledge to practical problems....

–http://wordnetweb.princeton.edu/perl/webwn

Engineering is the discipline, art, and profession of acquiring and applying technical, scientific, and mathematical knowledge to design and implement materials, structures, machines, devices, systems, and processes that safely realize a desired objective or invention.

–http://en.wikipedia.org/wiki/Engineering

...engineer: a person who uses scientific knowledge to solve practical problems....

–http://en.wikipedia.org/wiki/Engineering

Engineers work to develop economic and safe solutions to practical problems, by applying mathematics, scientific knowledge, and ingenuity while considering technical constraints....

–http://en.wikipedia.org/wiki/Engineering

Engineering is the practical application of science and math to solve problems....

–http://en.wikipedia.org/wiki/Engineering

Engineering Precedes Science

Skilled manual labor entails a systematic encounter with the material world, precisely the kind of encounter that gives rise to natural science. From its earliest practice, craft knowledge has entailed knowledge of the "ways" of one's materials—that is, knowledge of their nature, acquired through disciplined perception.

–Matthew B. Crawford, *Shop Class as Soulcraft*

Engineering Precedes Science

...in areas of well-developed craft practices, technological developments typically preceded and gave rise to advances in scientific understanding, not vice versa.

–Matthew B. Crawford, *Shop Class as Soulcraft*

Art & Engineering Precede Science

The steam engine is a good example. It was developed by mechanics who observed the relations between volume, pressure, and temperature. This was at a time when theoretical scientists were tied to the caloric theory of heat, which later turned out to be a conceptual dead end.

–Matthew B. Crawford, *Shop Class as Soulcraft*

Engineering is quite different from science. Scientists try to understand nature. Engineers try to make things that do not exist in nature. Engineers stress invention.

[T]he creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes....

–http://en.wikipedia.org/wiki/Engineering

Wednesday, March 21, 2012

Case Study
Mixins & Method Combination

- 1966: Teitelman invents *Advice* (simple mixins)
- 1979 / 1980: Howard I. Cannon invents *Flavors* (full-blown mixins, method combination)
- 1986: Symbolics revises Flavors to *New Flavors* (elaborated mixins, method combination)
- 1989: CLOS added to Common Lisp (refined mixins, method combination)

Case Study

Mixins & Method Combination

- 1990: (Computer) Science discovers mixins
- 1996 / 1997: (Computer) Science discovers aspects (but that's a longer story)

What is a Mixin?

- a class that supplies behavior but not identity
- a mechanism for combining component behavior without mucking with the source-code internals of the components

What is a Mixin?

In object-oriented programming languages, a mixin is a class that provides a certain functionality to be inherited or just reused by a subclass, while not meant for instantiation (the generation of objects of that class). Mixins are synonymous with abstract base classes. Inheriting from a mixin is not a form of specialization but is rather a means of collecting functionality. A class or object may "inherit" most or all of its functionality from one or more mixins, therefore mixins can be thought of as a mechanism of multiple inheritance.

–http://en.wikipedia.org/wiki/Mixin

What is a Mixin?

- Examples:
  - Flavors / CLOS with multiple inheritance and method combination
  - Aspects with pointcuts and join points
  - any abstract class with a method

Ordinary Class Primary Method

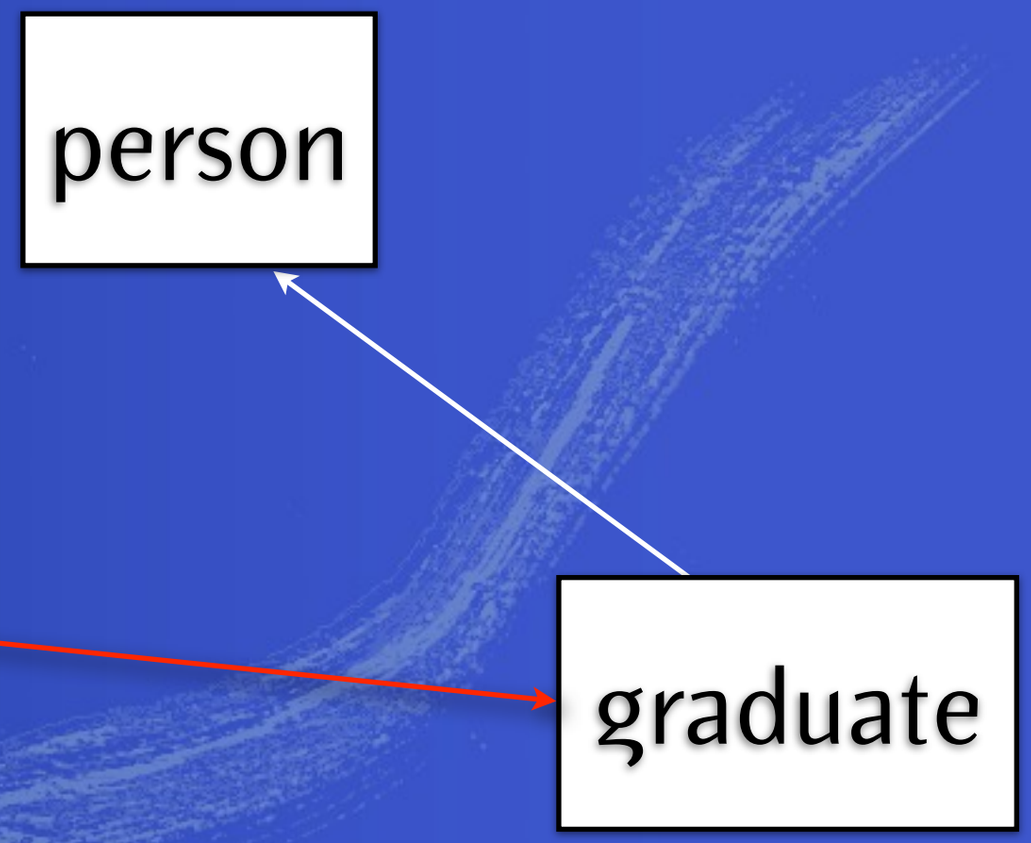
```
(defclass person () ((name :accessor name :initarg :name :initform "")))
(defmethod print-object ((person person) stream)
  (format stream (name person)))
```

```
CL-USER 31 > (setq rpg (make-instance 'person :name "Richard P. Gabriel"))
Richard P. Gabriel
```

```
(defclass graduate (person) ((degree :accessor degree :initarg :degree :initform "")))

(defmethod print-object :after ((person graduate) stream)
  (format stream " ~A" (degree person)))
```

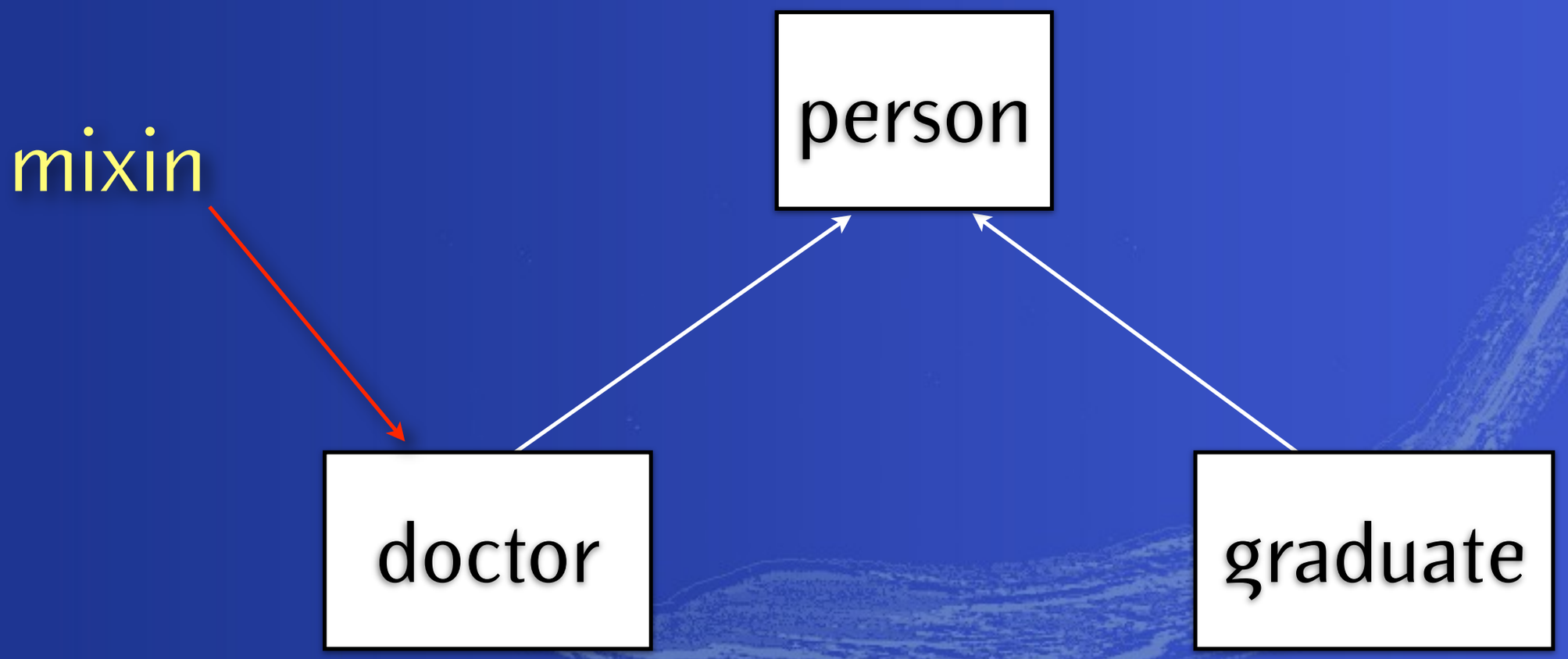
method combination

mixin



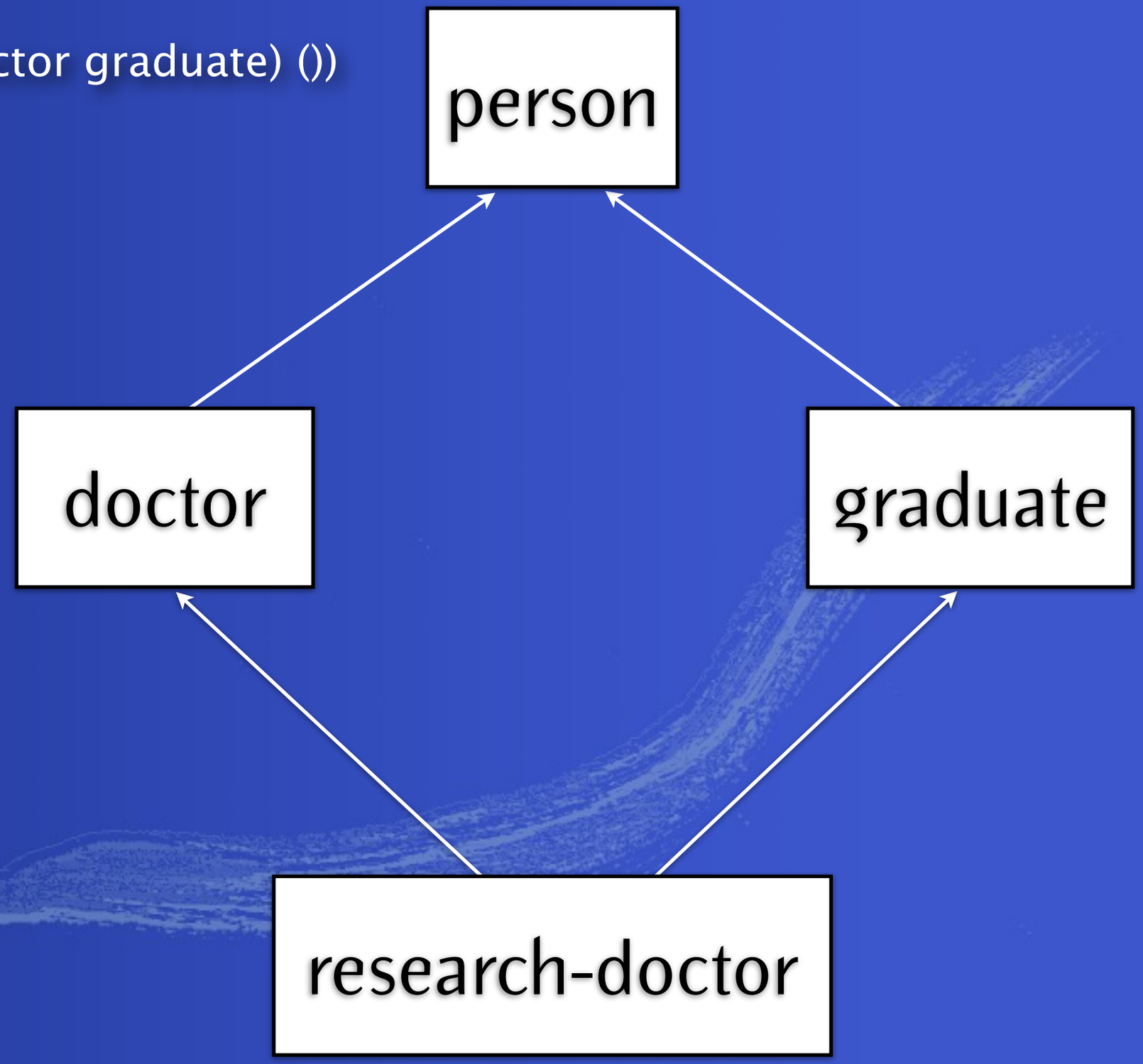

```
CL-USER 32 > (setq rpg (make-instance 'graduate :name "Richard P. Gabriel"
                                                :degree "PhD"))

Richard P. Gabriel PhD
```

```
(defclass doctor (person) ())

(defmethod print-object :before ((person doctor) stream) (format stream "Dr. "))
```




```
CL-USER 35 > (setq rpg (make-instance 'doctor :name "Richard P. Gabriel"))
Dr. Richard P. Gabriel
```

(defclass research-doctor (doctor graduate) ())




```
CL-USER 37 > (setq rpg (make-instance 'research-doctor
                                      :name "Richard P. Gabriel"
                                      :degree "PhD"))

Dr. Richard P. Gabriel PhD
```

```
CL-USER 38 > (setq rpg (make-instance 'research-doctor
                                      :name "Richard P. Gabriel"
                                      :degree "PhD"))

Dr. Richard P. GabrielRichard P. Gabriel PhD
```

Mixin-based Inheritance

Gilad Bracha*
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

William Cook
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303-0969

Abstract

The diverse inheritance mechanisms provided by Smalltalk, Beta, and CLOS are interpreted as different uses of a single underlying construct. Smalltalk and Beta differ primarily in the direction of class hierarchy growth. These inheritance mechanisms are subsumed in a new inheritance model based on composition of *mixins*, or abstract subclasses. This form of inheritance can also encode a CLOS multiple-inheritance hierarchy, although changes to the encoded hierarchy that would violate encapsulation are difficult. Practical application of mixin-based inheritance is illustrated in a sketch of an extension to Modula-3.

1 Introduction

A variety of inheritance mechanisms have been developed for object-oriented programming languages. These systems range from classical Smalltalk single inheritance [8], through the safer prefixing of Beta [12, 10], to the complex and powerful multiple inheritance combinations of CLOS [6, 9]. These languages have similar

*Supported by grant CCR-8704778 from the National Science Foundation.

in Smalltalk, new methods may be defined. However, prefix methods cannot be replaced; instead, the prefix may use the command **inner** to invoke the extended method code supplied by the subpattern. Given that the code in a prefix is executed in any of its extensions, Beta enforces a degree of behavioral consistency between a pattern and its subpatterns.

The underlying mechanism of inheritance is the same for Beta and Smalltalk [3]. The difference between them lies in whether the extensions to an existing definition have precedence over and may refer to previous definitions (Smalltalk), or the inherited definition has precedence over and may refer to the extensions (Beta). This model shows that Beta and Smalltalk have inverted inheritance hierarchies: a Smalltalk subclass refers to its parent using **super** just as a Beta prefix refers to its subpatterns using **inner**.

In the Common Lisp Object System (CLOS) and its predecessor, Flavors [13], multiple parent classes may be merged during inheritance. A class's ancestor graph is linearized so that each ancestor occurs only once [7]. With standard method combination for primary methods, the function call-next-method is used to invoke the next method in the inheritance chain.

[16, 17]. But the mixin technique in CLOS depends directly upon linearization and modification of parent-child relationships. Rather than avoid mixins because they violate encapsulation, we argue that linearization is an implementation technique for mixins that obscures their true nature as abstractions.

By modest generalization of the inheritance models

First Scientific Publication about Mixins

Bracha & Cook is arguably the primary reference for mixins in the (computer) scientific literature

Some Statistics: Citations

Bracha & Cook	CiteSeer	375
	ACM DL	203
	Google Scholar	842
Moon	CiteSeer	<not found>
	ACM DL	102
	Google Scholar	402
Cannon	CiteSeer	<not found>
	Google Scholar	55

What They Claim:

The diverse inheritance mechanisms provided by Smalltalk, Beta, and CLOS are interpreted as different uses of a single underlying construct. Smalltalk and Beta differ primarily in the direction of class hierarchy growth. These inheritance mechanisms are subsumed in a new inheritance model based on composition of mixins, or abstract subclasses. This form of inheritance can also encode a CLOS multiple-inheritance hierarchy, although changes to the encoded hierarchy that would violate encapsulation are difficult.

–Bracha & Cook, *Mixin-Based Inheritance, 1990*

CLOS
1989

David Moon
New Flavors
1986

Howard Cannon
Flavors
1979

Danny Bobrow
Common Loops
1986

Alan Kay
Smalltalk
1980

Danny Bobrow
Loops
1983

For us...

- Our engineers (computer science / software people) build things that become part of the nature our scientists study
- That is, our engineers create the nature our scientists study

- Because they create nature, our engineers can understand things differently from our scientists
- Our engineers and our scientists don't necessarily speak the same language
- Both publish / report their understandings of the nature they create / study
- Those engineers and scientists don't talk to each other much, because...
- They don't publish in the same places (anymore)

Object-Oriented Programming with *Flavors*

David A. Moon
Symbolics, Inc.

Abstract

This paper describes Symbolics' newly redesigned object-oriented programming system, *Flavors*. *Flavors* encourages program modularity, eases the development of large, complex programs, and provides high efficiency at run time. *Flavors* is integrated into Lisp and the Symbolics program development environment. This paper describes the philosophy and some of the major characteristics of Symbolics' *Flavors* and shows how the above goals are addressed. Full details of *Flavors* are left to the programmers' manual, *Reference Guide to Symbolics Common Lisp. (5)*

History of *Flavors*

The original *Flavors* system was developed by the MIT Lisp Machine group in 1979. *(1, 2, 3)* It was used to build a window system and later applied to other system programming. In 1981, Symbolics designed a more efficient implementation of *Flavors*. Since that time, we have made increasingly heavy use of *Flavors* in nearly every aspect of the Symbolics 3600 software, such as I/O streams, network control programs, the debugger, the editor, and user interface facilities. Flavors permeate both the operating system and higher-level utilities. The same *Flavors* tools used in-house are fully documented and are used by most Symbolics customers to build their application programs.

Five years' experience with *Flavors* has pointed out the strengths and weaknesses of the original design. In 1985 we undertook a thorough redesign of *Flavors* to solve the problems that we had identified. The result is a new *Flavors* system that has been implemented at Symbolics and has been used in-house to develop several complex programs. This newly-designed *Flavors* is the version described in this paper. It will be released with the next Symbolics software release. *(5)*

What is object-oriented programming?

We view object-oriented programming as a technique for organizing very large programs. This technique makes it practical to deal with programs that would otherwise be impossibly complex.

An object-oriented program consists of a set of objects and a set of operations on those objects. These entities are not defined in a monolithic way. Instead, the definitions of the operations are distributed among the various objects that they can operate upon. At the same time, the definitions of the objects are distributed among the various facets of their behavior. An object-oriented programming system is an organizational framework for combining these distributed definitions and managing the interactions among them.

Object-oriented programming is also an abstraction mechanism. A program that manipulates an object uses certain defined operations to manipulate it. These operations serve as an interface, and the program does not need to know how the object implements the operations. The implementation of one operation can be different for different kinds of objects. At the same time, an object's behavior can be divided into several facets, which need not know each other's internal details.

Goals of *Flavors*

There are many possible and useful styles of object-oriented programming. *Flavors* adopts an approach aimed at these goals:

- Encourage program modularity. By this we mean that *Flavors* should make it easier to construct programs out of existing parts, to modify the behavior of existing programs without massively rewriting them,

First Engineering Publication about Mixins

Flavors

A non-hierarchical approach to object-oriented programming

by

Howard I. Cannon

Object-oriented programming systems have typically been organized around hierarchical inheritance. Unfortunately, this organization restricts the usefulness of the object-oriented programming paradigm. This paper presents a non-hierarchically organized object-oriented system, an implementation of which has been in practical use on the MIT Lisp Machine since late 1979.

First Engineering Paper / Report about Mixins

Flavors

A non-hierarchical approach to object-oriented programming

by

Howard I. Cannon

Object-oriented programming systems have typically been organized around hierarchical inheritance. Unfortunately, this organization restricts the usefulness of the object-oriented programming paradigm. This paper presents a non-hierarchically organized object-oriented system, an implementation of which has been in practical use on the MIT Lisp Machine since late 1979.

First Engineering Paper / Report about Mixins

PILOT: A Step Toward Man-Computer Symbiosis

by

Warren Teitelman

Submitted to the Department of Mathematics on June 14, 1966, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

PILOT is a programming system constructed in LISP. It is designed to facilitate the development of programs by easing the familiar sequence: write some code, run the program, make some changes, write some more code, run the program again, etc. As a program becomes more complex, making these changes becomes harder and harder because the implications of changes are harder to anticipate.

In the PILOT system, the computer plays an active role in this evolutionary process by providing the means whereby changes can be effected immediately, and in ways that seem natural to the user. The user of PILOT feels that he is giving advice, or making suggestions, to the computer about the operation of his programs, and that the system then performs the work necessary. The PILOT system is thus an interface between the user and his program, monitoring both the requests of the user and the operation of his program.

The user may easily modify the PILOT system itself by giving

First Engineering Paper / Report about the Precursor to Mixins

Thesis Supervisor: Marvin L. Minsky

Title: Professor of Electrical Engineering

PILOT: A Step Toward Man-Computer Symbiosis

by

Warren Teitelman

Submitted to the Department of Mathematics on June 14, 1966, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

PILOT is a programming system constructed in LISP. It is designed to facilitate the development of programs by easing the familiar sequence: write some code, run the program, make some changes, write some more code, run the program again, etc. As a program becomes more complex, making these changes becomes harder and harder because the implications of changes are harder to anticipate.

In the PILOT system, the computer plays an active role in this evolutionary process by providing the means whereby changes can be effected immediately, and in ways that seem natural to the user. The user of PILOT feels that he is giving advice, or making suggestions, to the computer about the operation of his programs, and that the system then performs the work necessary. The PILOT system is thus an interface between the user and his program, monitoring both the requests of the user and the operation of his program.

The user may easily modify the PILOT system itself by giving

First Engineering Paper / Report about the Precursor to Mixins

Thesis Supervisor: Marvin L. Minsky

Title: Professor of Electrical Engineering

Wednesday, March 21, 2012

before Advice after

Advising is the basic innovation in the model.... Advising consists of inserting new procedures at any or all of the entry or exit points to a particular procedure (or class of procedures).

Since each piece of advice is itself a procedure, it has its own entries and exits. In particular, this means that the execution of advice can cause the procedure that it modifies to be bypassed completely, e.g., by specifying as an exit from the advice one of the exits from the original procedure; or the advice may change essential variables and continue with the computation so that the original procedure is executed, but with modified variables. Finally, the advice may not alter the execution or affect the original procedure at all, e.g., it may merely perform some additional computation such as printing a message or recording history. Since advice can be conditional, the decision as to what is to be done can depend on the results of the computation up to that point.

–Warren Teitelman, *PhD Dissertation, MIT, 1966*

(tell solution1, (before number advice),
If (countf history ((solution1 -))) is greater than 2,
then quit)

Non-Hierarchical Object Systems

To restate the fundamental problem: there are several separate (orthogonal) attributes that an object wants to have; various facets of behavior (features) that want to be independently specified for an object.... It is very easy to combine completely non-interacting behaviors. Each would have its own set of messages, its own instance variables, and would never need to know about other objects with which they would be combined....The problem arises when it is necessary to have modular interactions between the orthogonal issues. Though the label does not interact strongly with either the window or the border, it does have some minor interactions. For example, it wants to get redrawn when the window gets refreshed. Handling these sorts of interactions is the Flavor system's main goal.

–Howard I. Cannon, *Flavors, MIT, 1979*

```
(defflavor WINDOW
   (OBJECT)
   (X-POSITION Y-POSITION WIDTH HEIGHT))
(defmethod (WINDOW :REFRESH)
   (send SELF ':CLEAR))
(defflavor BORDER
   ()
   (BORDER-WIDTH))
(defmethod (BORDER :AFTER :REFRESH) ()
   (send SELF ':DRAW-BORDER))
(defflavor LABEL
   ()
   (LABEL))
(defmethod (LABEL :AFTER :REFRESH) ()
   (send SELF ':DRAW-LABEL))
 (defflavor WINDOW-WITH-LABEL-AND-BORDER
   (LABEL BORDER WINDOW)
   ())
```

Flavors-Example.lisp


```
(defflavor WINDOW
   (OBJECT)
   (X-POSITION Y-POSITION WIDTH HEIGHT))

(defmethod (WINDOW :REFRESH)
   (send SELF ':CLEAR))


(defflavor BORDER
   ()
   (BORDER-WIDTH))

(defmethod (BORDER :AFTER :REFRESH) ()
   (send SELF ':DRAW-BORDER))


(defflavor LABEL
   ()
   (LABEL))

(defmethod (LABEL :AFTER :REFRESH) ()
   (send SELF ':DRAW-LABEL))

(defflavor WINDOW-WITH-LABEL-AND-BORDER
   (LABEL BORDER WINDOW)
   ())
```

Method Combination

In...class systems, which method(s) to run is largely determined at run time. This is possible as only local knowledge is necessary to make the decision. This is not true of Flavors: ...determining the methods to be run requires inspecting all of the component flavors and generating a combined method. It is from this use of global knowledge that Flavors gain the ability to modularly integrate essentially orthogonal issues. At instantiation time, as the component flavors are inspected, combined methods are generated....

–Howard I. Cannon, *Flavors, MIT, 1979*

A method combination can be viewed as a template for converting a list of methods into a piece of code that calls the appropriate methods in the appropriate order and returns the appropriate values. This code is the combined method. The default...method combination is called ***daemon combination****. There are three types of methods...: primary, before, and after. Untyped methods default to primary type. The combined method first calls all of the before methods in order and throws away the value they return, then the first primary method is called and its value is saved, then the after methods are called in reverse order, and then the combined method returns the value returned by the primary method.*

–Howard I. Cannon, *Flavors, MIT, 1979*

Convention

A Choice, not a Cop-out

From one point of view, object oriented programming is a set of conventions that help the programmer organize his program. When the conventions are supported by a set of tools that make them easier to follow, then an object oriented programming system is born. It is neither feasible nor desirable to have the system enforce all of the conventions, however. Since the Flavor system provides more flexibility than other object oriented programming systems, programmer enforced conventions become correspondingly more important. Therefore, the Flavor system is as much conventions as it is code.

–Howard I. Cannon, *Flavors, MIT, 1979*

Howard I. Cannon was ~20 years old when he wrote that paper / did that research / implemented the first Flavors system.

I recall the excitement it spawned as it was passed from hacker to hacker over the ARPANet.

I could barely understand the paper.

If there is one water in Europe I want, it is the black
Cold pool where into the scented twilight
A child squatting full of sadness, launches
A boat as fragile as a butterfly in May.

–Arthur Rimbaud, *The Drunken Boat, 1871 (age 17)*
translated by Rebecca Seiferle

Wednesday, March 21, 2012

Flavors Redesigned

A typical flavor is defined by combining several other flavors, called its components. The new flavor inherits...from its components. In a well-organized program, each component flavor is a module that defines a single facet of behavior. When two types of objects have some behavior in common, they each inherit it from the same flavor, rather than duplicating the code. When flavors are mixed together, Flavors organizes and manages the interactions between them. This multiple inheritance is a key aspect of the design of Flavors....

–David A. Moon, *OOP with Flavors, OOPSLA 1986*

Each flavor defines certain constraints on the ordering of itself and its direct[ly specified] components. Taken together, these constraints determine a partial ordering of all of the components of a flavor. Flavors computes a total ordering that is consistent with the partial ordering. Three rules control the ordering of flavor components:

–David A. Moon, *OOP with Flavors, OOPSLA 1986*

- *A flavor always precedes its own components*
- *The local ordering of components of a flavor is preserved. This is the order of components given in the* defflavor *form*
- *Duplicate flavors are eliminated from the ordering. If a flavor appears more than once, it is placed as close to the beginning of the ordering as possible, while still obeying the other rules*

–David A. Moon, *OOP with Flavors, OOPSLA 1986*

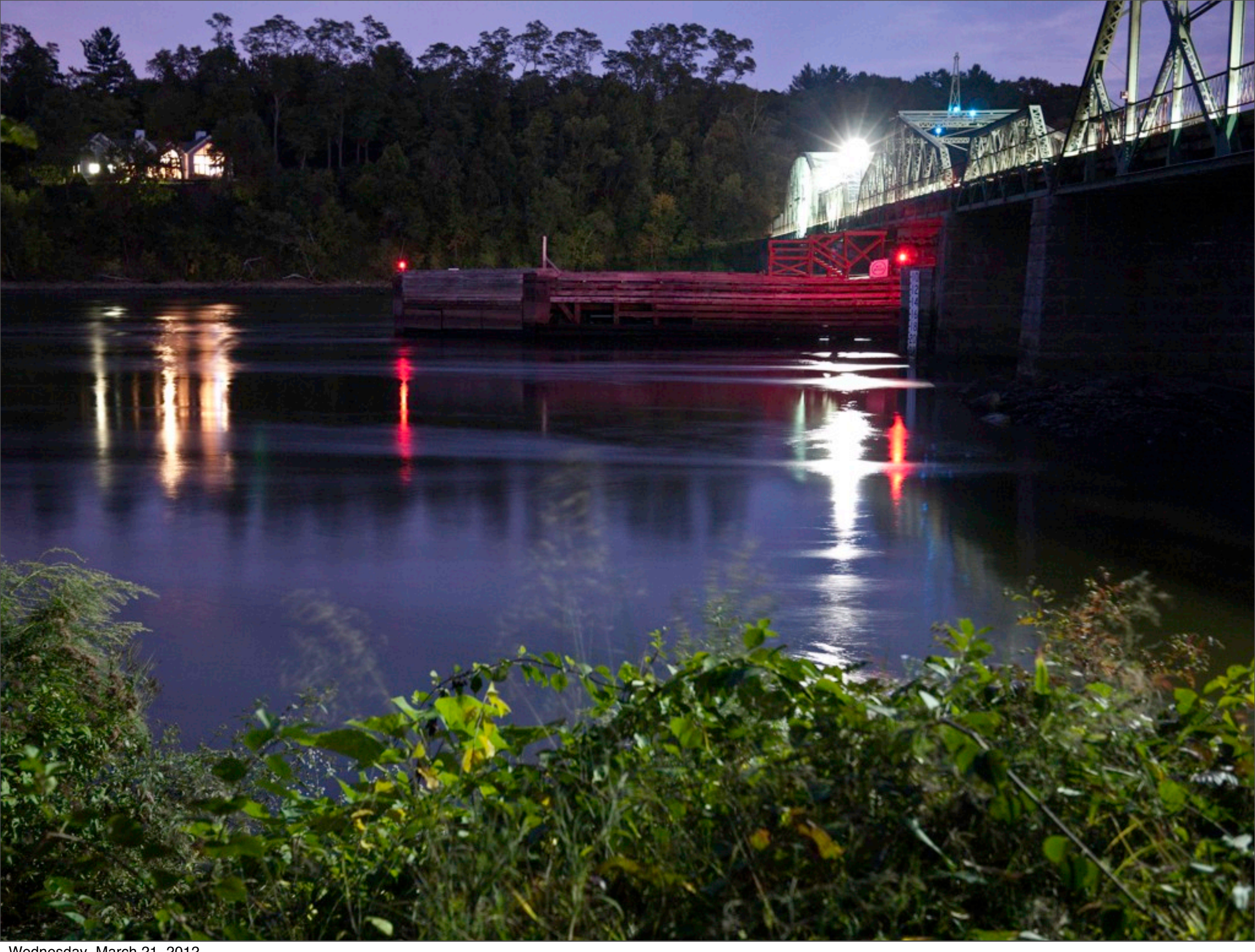
No programming system can guarantee program modularity or eliminate the need for careful design of a program's structure. However, a programming system can make it easier to build modular programs.

–David A. Moon, *OOP with Flavors, OOPSLA 1986*

L&FP, OOPSLA, etc Go All-In Science

- 1996: ACM L&FP becomes ICFP(IL)
- 1997: the journal LASC becomes HO&SC(IL)
- 1990s: Computer Science effectively abandons a substantial source of big ideas—its other voice—inadvertently classifying them as practitioners
- 2000s: Professor of Practice: "appointees will hold the rank of Professor but, while having the stature, will not have rights...." –University of Maryland, 2012

There was something funny about the Bracha & Cook paper...



Wednesday, March 21, 2012

“I don’t want to die in a language I can’t understand”

–Jorge Luis Borges

Something is Odd about Bracha & Cook

- B&C refers to CLOS, but gets several terms wrong; seems to ignore others
- B&C uses a simplification of method combination that throws away the very aspect they want to explore
- B&C relies on a paper about Flavors, a brief overview of CLOS from ECOOP, and a programmer's guide written by a technical writer at Symbolics
- A lot of "Lisp people" have found fault with B&C's understanding of CLOS and Lisp

1990

Engineering Paradigm

for programming languages

1995

Scientific Paradigm

for programming languages

1990 1995

Engineering Paradigm
for programming languages

Scientific Paradigm
for programming languages

1900 1974	1975 1979	1980 1984	1985 1989	1990 1994	1995 1999	2000 2004	2005 2009	2010 2011
39	20	26	60	150	236	379	829	400

All citations for OOPSLA 2011

1990 1995

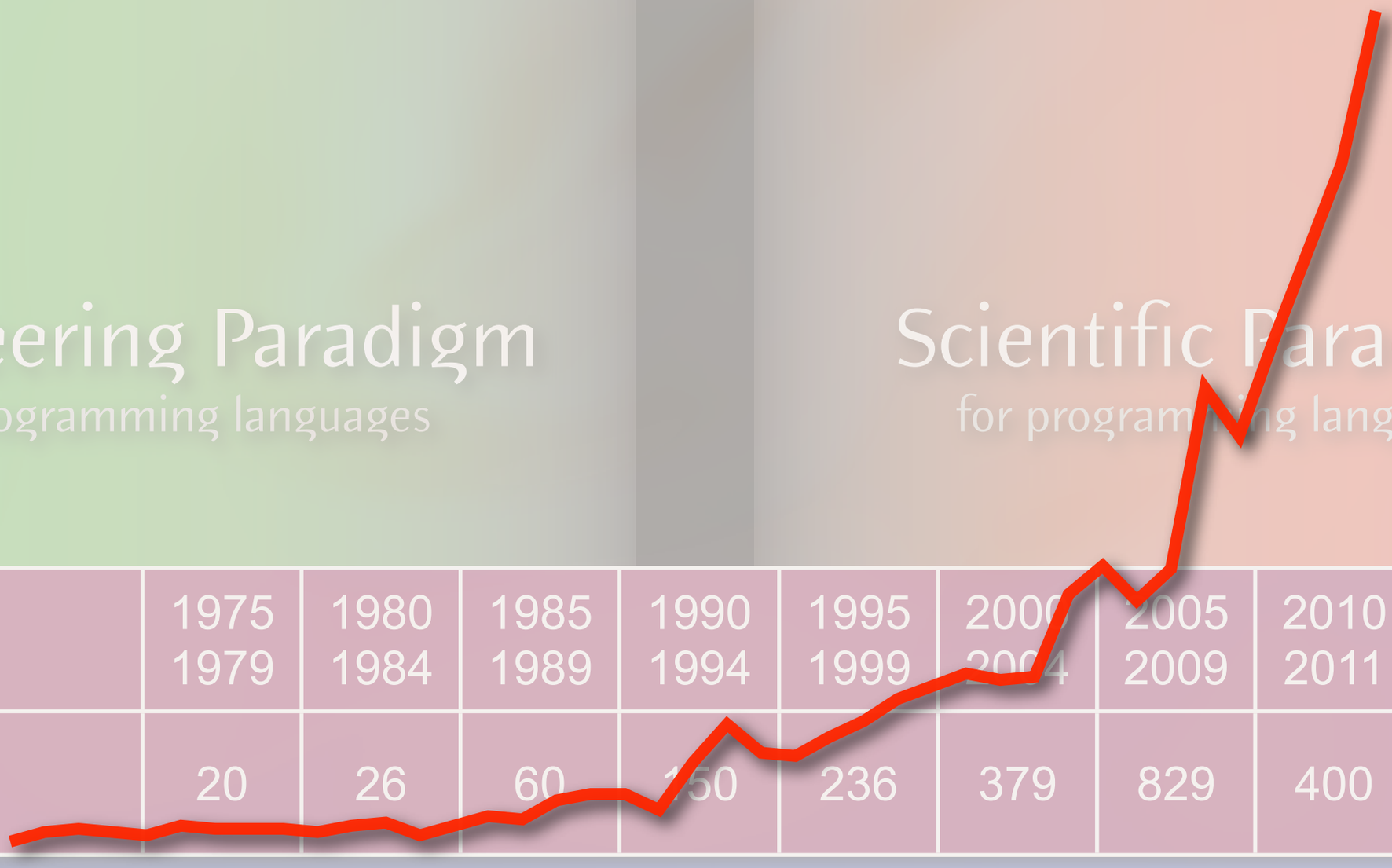
Engineering Paradigm

for programming languages

Scientific Paradigm

for programming languages

1900 1974	1975 1979	1980 1984	1985 1989	1990 1994	1995 1999	2000 2004	2005 2009	2010 2011
39	20	26	60	150	236	379	829	400



All citations for OOPSLA 2011

The Argument

- B&C was published in the Scientific Paradigm
- The papers it refers to about CLOS were published in the Engineering Paradigm
- These paradigms are incommensurable—are the authors even talking about the same things???

Wednesday, March 21, 2012

Language vs System

A variety of inheritance mechanisms have been developed for object-oriented programming languages. These systems range from classical Smalltalk single inheritance, through the safer prefixing of Beta, to the complex and powerful multiple inheritance combinations of CLOS. These languages have similar object models, and also....

–Bracha & Cook, *Mixin-Based Inheritance, 1990*

Language vs System

- Bracha & Cook talk about ***languages***
- Cannon, Moon, & CLOS talk about ***systems***

> *[Language:] a formal system of signs governed by grammatical rules of combination to communicate meaning*
>
> *[A] System is a set of interacting or interdependent components forming an integrated whole*

–http://en.wikipedia.org/

Language vs System

- In a system, a good designer, following good design principles, ensures good design and beneficial characteristics
- In a language, language designers endeavor to make some examples of bad or poor design ungrammatical

...object oriented programming is a set of conventions that helps the programmer organize his program. When the conventions are supported by a set of tools that make them easier to follow, then an object oriented programming system is born. It is neither feasible nor desirable to have the system enforce all of the conventions, however. Since the Flavor system provides more flexibility than other object oriented programming systems, programmer enforced conventions become correspondingly more important. Therefore, the Flavor system is as much conventions as it is code.

–Howard I. Cannon, *Flavors, 1979*

This paper describes Symbolics' newly redesigned object-oriented programming system, Flavors.

Flavors encourages program modularity,....

–David A. Moon, *OOP with Flavors, 1986*

An object-oriented program consists of a set of objects and a set of operations on those objects. These entities are not defined in a monolithic way. Instead, the definitions of the operations are distributed among the various objects that they can operate upon. At the same time, the definitions of the objects are distributed among the various facets of their behavior. An object-oriented programming system is an organizational framework for combining these distributed definitions and managing the interactions among them.

–David A. Moon, *OOP with Flavors, 1986*

The Common Lisp Object System is an object-oriented extension to Common Lisp as defined in "Common Lisp: The Language, by Guy L. Steele Jr." It is based on generic functions, multiple inheritance, declarative method combination, and a meta-object protocol.

–DeMichiel et al, *The Common Lisp Object System Specification, 1988*

1990

System Paradigm
for programming languages

1995

Language Paradigm
for programming languages

Lisp Requires System Thinking

This is one of the great advantage of Lisp-like languages: They have very few ways of forming compound expressions, and almost no syntactic structure.... After a short time we forget about syntactic details of the language (because there are none) and get on with the real issues.

–Abelson & Sussman

What I like about Lisp is that you can feel the bits between your toes.

–Drew McDermott

Lisp has such a simple syntax and semantics that parsing can be treated as an elementary task. Thus parsing technology plays almost no role in Lisp programs, and the construction of language processors is rarely an impediment to the rate of growth and change of large Lisp systems.

–Alan Perlis

Pascal is for building pyramids—imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms....

–Alan Perlis

Pascal is for building pyramids—
imposing, breathtaking, static structures
built by armies pushing heavy blocks into
place. Lisp is for building organisms....

Pascal is for building pyramids—imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms....

–Alan Perlis

In Lisp
You Build Systems

- Careful design
- Good organization
- Modular thinking
- The underlying system is designed ***to help*** you

By altering the living, running one right in front of you

In Case You're Not Sure CLOS is a True System

A class that is an instance of standard-class can be redefined if the new class will also be an instance of standard-class. Redefining a class modifies the existing class object to reflect the new class definition; it does not create a new class object for the class.

When the class C is redefined, changes are propagated to its instances and to instances of any of its subclasses. Updating such an instance occurs at an implementation-dependent time, but no later than the next time a slot of that instance is read or written. Updating an instance does not change its identity as defined by the eq *function. The updating process may change the slots of that particular instance, but it does not create a new instance.*

–DeMichiel et al, *The Common Lisp Object System Specification, 1988*

In Case You're Not Sure CLOS is a True System

The function change-class *can be used to change the class of an instance from its current class,* C_{from}, *to a different class,* C_{to}; *it changes the structure of the instance to conform to the definition of the class* C_{to}.

–DeMichiel et al, *The Common Lisp Object System Specification, 1988*

(Behavioral) Inheritance

A CLOS class may inherit from more than one parent. As a result, a given ancestor may be inherited more than once.... To remedy this situation, CLOS linearizes the ancestor graph of a class to produce an inheritance list in which each ancestor occurs only once.

–Bracha & Cook, *Mixin-Based Inheritance, 1990*

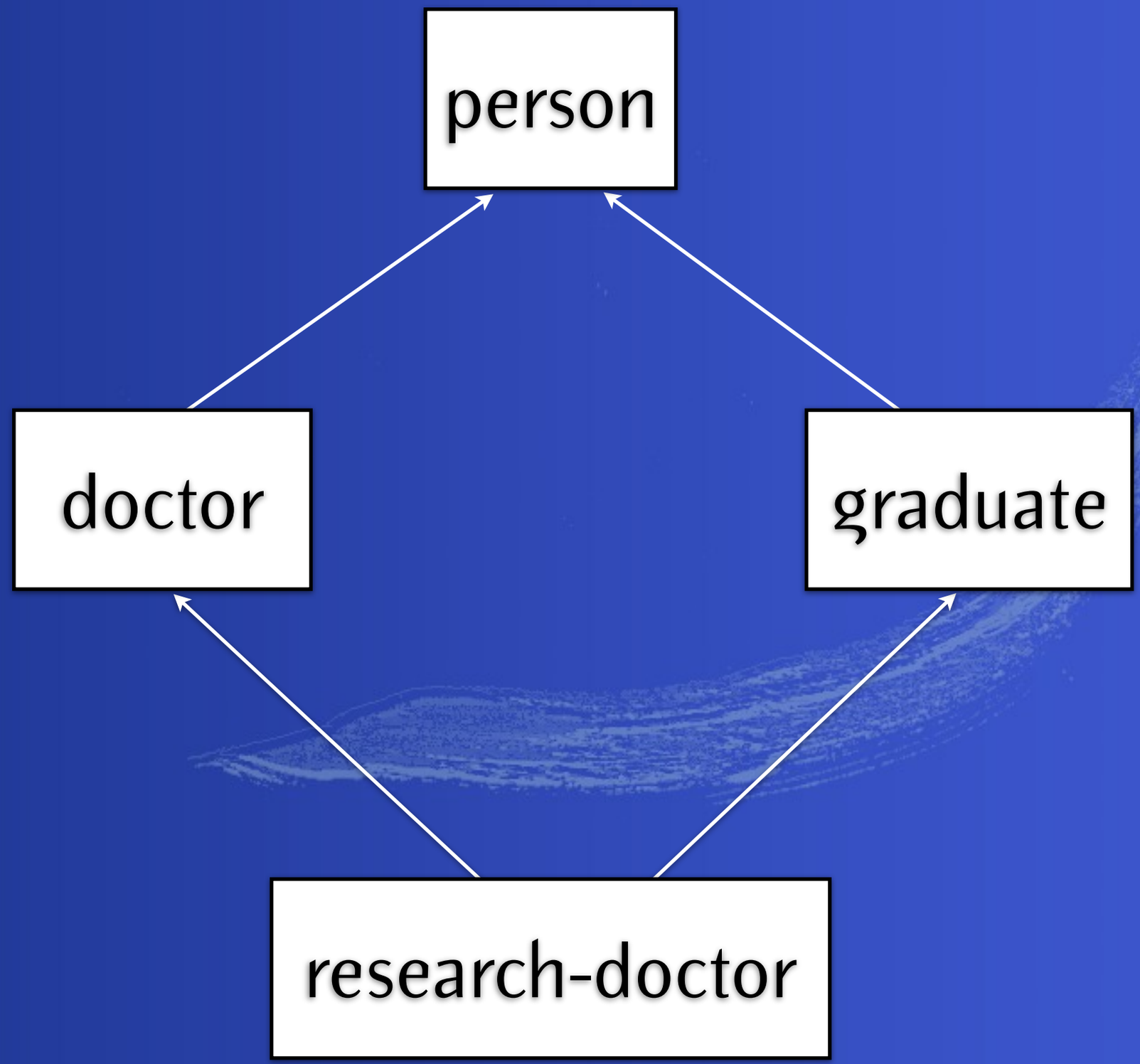
The process of linearization has been criticized for violating encapsulation. One argument is that the relationship between a class and its declared parents may be modified during linearization. This is demonstrated by the example above, where in the linearization the class Graduate *is placed between* Doctor *and* Person, *in contradiction of the explicit declaration of* Doctor *that it inherits directly from Person.*

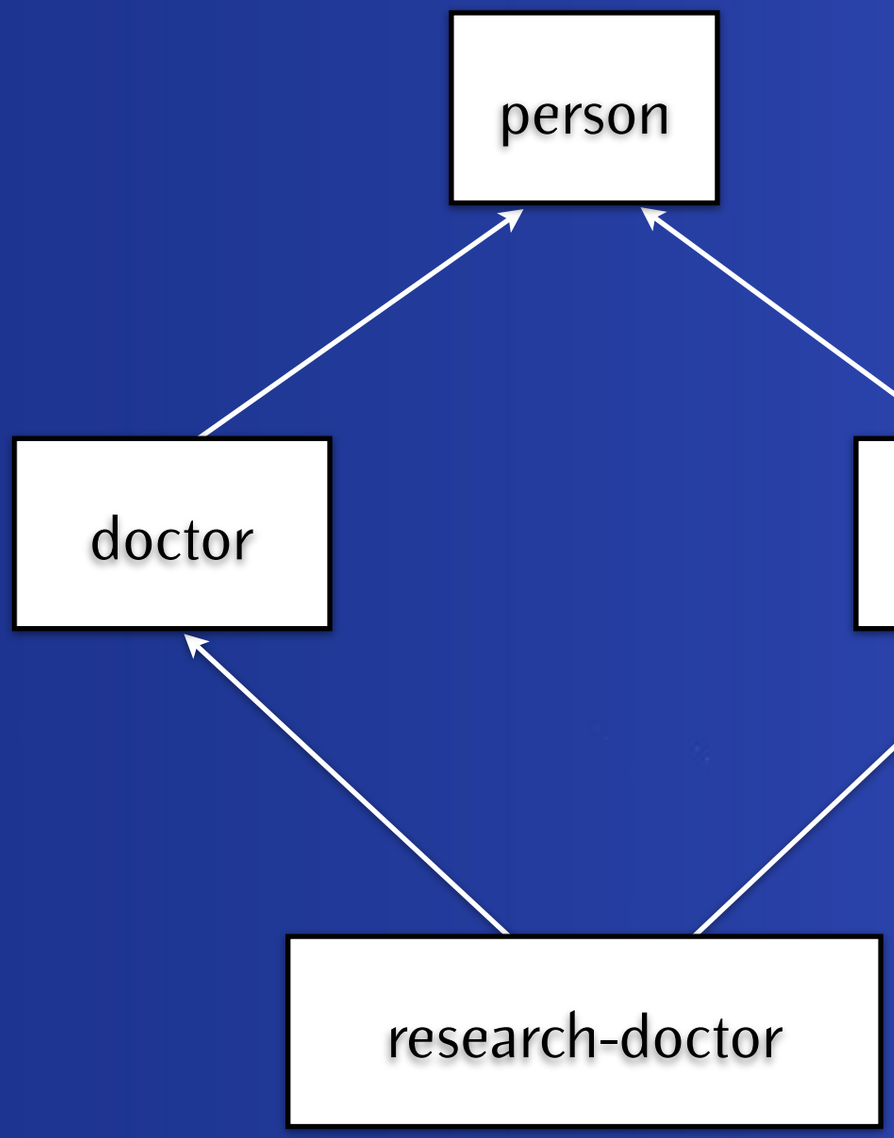
–Bracha & Cook, *Mixin-Based Inheritance, 1990*

```
(defclass person () (name))

(defclass graduate (person) (degree))

(defclass doctor (person) ())

(defclass research-doctor (doctor graduate) ())
```

in the linearization, the class Graduate
is placed between Doctor *and* Person

–Bracha & Cook, *Mixin-Based Inheritance, 1990*


person
doctor
graduate
research-doctor



...in contradiction of the explicit declaration of Doctor *that it inherits directly from* Person.

–Bracha & Cook, *Mixin-Based Inheritance, 1990*

(RESEARCH-DOCTOR DOCTOR GRADUATE PERSON STANDARD-OBJECT T)

Why B&C Might Believe That

A class C_1 is a direct superclass of a class C_2 if C_2 explicitly designates C_1 as a superclass in its definition. In this case C_2 is a direct subclass of C_1. A class C_n is a superclass of a class C_1 if there exists a series of classes $C_2,\ldots,C_{n-1}$ such that C_{i+1} is a direct superclass of C_i for $1 \leq i < n$. In this case, C_1 is a subclass of C_n. A class is considered neither a superclass nor a subclass of itself. That is, if C_1 is a superclass of C_2, then $C_1 \neq C_2$. The set of classes consisting of some given class C along with all of its superclasses is called "C and its superclasses."

–DeMichiel et al, *The Common Lisp Object System Specification, 1988*

But...

Each class has a class precedence list, *which is a total ordering on the set of the given class and its superclasses. The total ordering is expressed as a list ordered from most specific to least specific. The class precedence list is used in several ways. In general, more specific classes can shadow, or override, features that would otherwise be inherited from less specific classes. The method selection and combination process uses the class precedence list to order methods from most specific to least specific.*

When a class is defined, the order in which its direct superclasses are mentioned in the defining form is important. Each class has a local precedence order, *which is a list consisting of the class followed by its direct superclasses in the order mentioned in the defining form.*

A class precedence list is always consistent with the local precedence order of each class in the list.

–DeMichiel et al,*The Common Lisp Object System Specification, 1988*

Flavors Redesigned

Each flavor defines certain constraints on the ordering of itself and its direct[ly specified] components. Taken together, these constraints determine a partial ordering of all of the components of a flavor. Flavors computes a total ordering that is consistent with the partial ordering. Three rules control the ordering of flavor components:

–David A. Moon, *OOP with Flavors, OOPSLA 1986*

- *A flavor always precedes its own components*
- *The local ordering of components of a flavor is preserved. This is the order of components given in the* defflavor *form*
- *Duplicate flavors are eliminated from the ordering. If a flavor appears more than once, it is placed as close to the beginning of the ordering as possible, while still obeying the other rules*

–David A. Moon, *OOP with Flavors, OOPSLA 1988*

In a well-organized program, each component flavor is a module that defines a single facet of behavior.

The goal of ordering flavor components is to preserve the modularity of programs. A flavor should be treated as an intact unit, with well-defined characteristics and behavior; it is essential that mixing flavors together does not alter the internal details of any of the component flavors.

–David A. Moon, *OOP with Flavors, 1988*

No programming system can guarantee program modularity or eliminate the need for careful design of a program's structure. However, a programming system can make it easier to build modular programs. Flavors provides organizational techniques for writing programs in a modular way and keeping them modular as they evolve. Inheritance of methods encourages modularity by allowing objects that have similar behavior to share code. Objects that have somewhat different behavior can combine the generalized behavior with code that specializes it. Multiple inheritance further encourages modularity by allowing object types to be built up from a toolkit of component parts.

–David A. Moon, OOP with Flavors, 1986

–David L. Parnas, *On the criteria to be used in decomposing systems into modules, 1972*

My original example of information hiding was the simple KWIC index program described in [his famous 1972 paper]. *This program is still used as an example of the principle. Only once has anyone noticed that it contains a flaw caused by a failure to hide an important assumption. Every module in the system was written with the knowledge that the data comprised strings of strings. This led to a very inefficient sorting algorithm because comparing two strings, an operation that would be repeated many times, is relatively slow. Considerable speed-up could be obtained if the words were sorted once and replaced by a set of integers with the property that if two words are alphabetically ordered, the integers representing them will have the same order. Sorting strings of integers can be done much more quickly than sorting strings of strings. The module interfaces described in* [a companion to that famous 1972 paper] *do not allow this simple improvement to be confined to one module.*

–David L. Parnas, *The Secret History of Information Hiding, 2002*

No programming system can guarantee program modularity or eliminate the need for careful design of a program's structure. However, a programming system can make it easier to build modular programs. Flavors provides organizational techniques for writing programs in a modular way and keeping them modular as they evolve. Inheritance of methods encourages modularity by allowing objects that have similar behavior to share code. Objects that have somewhat different behavior can combine the generalized behavior with code that specializes it. Multiple inheritance further encourages modularity by allowing object types to be built up from a toolkit of component parts.

–David A. Moon, OOP with Flavors, 1986

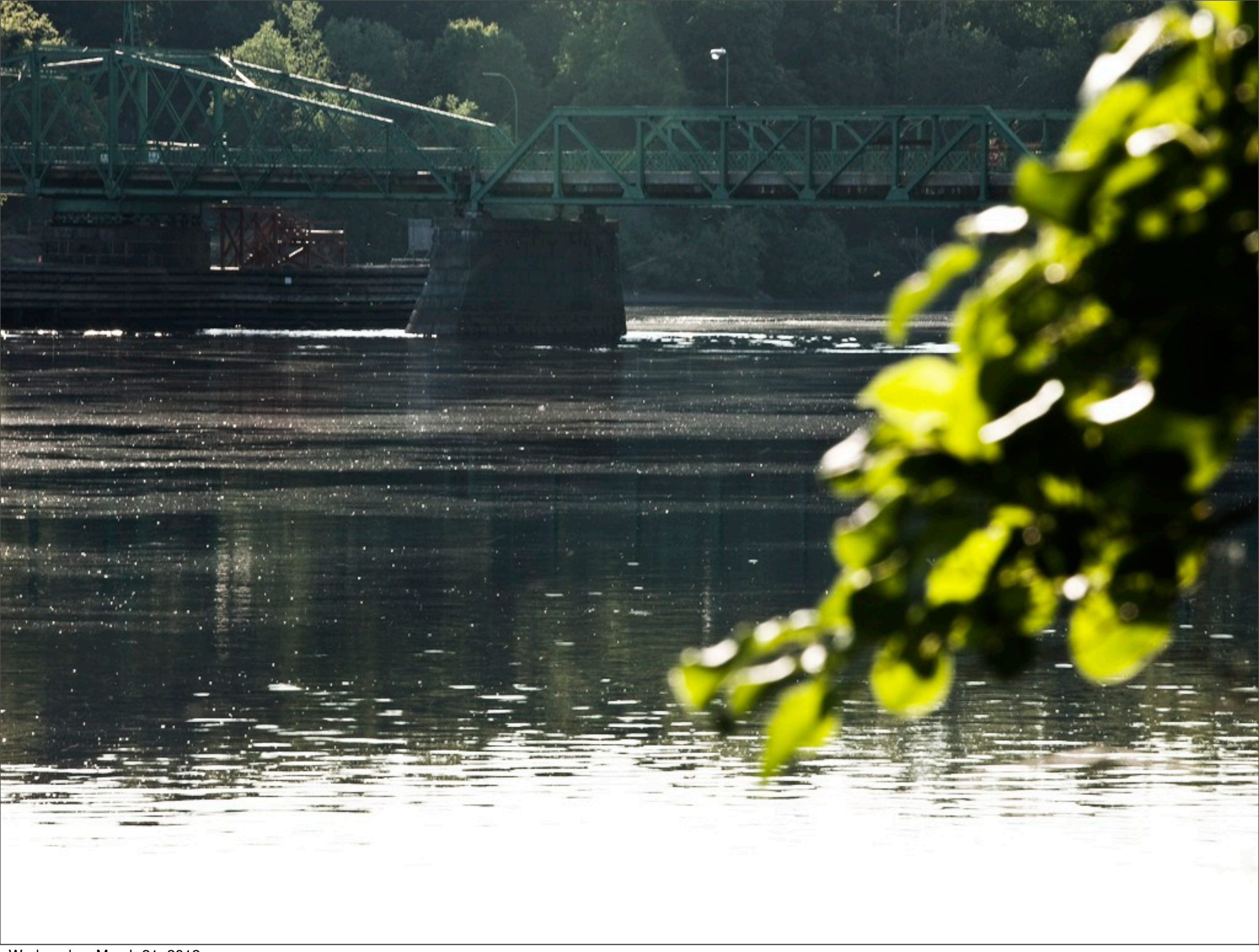
In...class systems, which methods to run is largely determined at run time. This is possible as only local knowledge is necessary to make the decision. This is not true of Flavors: in general, determining the methods to be run requires inspecting all of the component flavors and generating a combined method. It is from this use of global knowledge that Flavors gain[s] the ability to modularly integrate essentially orthogonal issues. At instantiation time, as the component flavors are inspected, combined methods are generated and an association list of message names and combined methods is constructed. In essence, the lattice structure is flattened into a linear one. This is important it makes the use of global knowledge practical, since in certain cases, the combined methods are not trivial, and could not easily be generated dynamically.

–Howard I. Cannon, *Flavors, 1979*

Is It Even Inheritance?

A subclass inherits methods in the sense that any method applicable to all instances of a class is also applicable to all instances of any subclass of that class.

–DeMichiel et al, *The Common Lisp Object System Specification, 1988*



Wednesday, March 21, 2012

Incommensurability?

The past is a foreign country.
They do things differently there.

–L. P. Hartley, *The Go-Between*, 1953

Thomas Kuhn

The Structure of Scientific Revolutions

Incommensurability is a notion that for me emerged from attempts to understand apparently nonsensical passages encountered in old scientific texts. Ordinarily they have been taken as evidence of the author's confused or mistaken beliefs. My experiences led me to suggest, instead, that those passages were being misread: the appearance of nonsense could be removed by recovering older meanings for some of the terms involved, meanings different from those subsequently current.

–Thomas Kuhn, *The Road Since Structure*

Thomas Kuhn

The Structure of Scientific Revolutions

The most important and most controversial aspect of Kuhn's theory involved his use of the terms "paradigm shift" and "incommensurability." That the scientific terms of one paradigm are incommensurable with the scientific terms of the paradigm that replaces it. A revolution occurs. One paradigm is replaced with another. And the new paradigm is incommensurable with the old one.

–Errol Morris, *The Ashtray*, NYT, 2011

Thomas Kuhn

The Structure of Scientific Revolutions

People in different paradigms speak different languages, and there is no way to translate the scientific language of one paradigm into the scientific language of another[—e]ven when they use the same words.

–Errol Morris, *The Ashtray*, NYT, 2011

Thomas Kuhn

The Structure of Scientific Revolutions

We may want to say that after a revolution scientists are responding to a different world.

–Thomas Kuhn, *The Structure of Scientific Revolutions*

Thomas Kuhn

The Structure of Scientific Revolutions

Consider...the men who called Copernicus mad because he proclaimed the earth moved....Part of what they meant by 'earth' was fixed position.... Copernicus' innovation was...a whole new way of regarding the problems of physics and astronomy. The proponents of competing paradigms practice their trades in different worlds....

–Thomas Kuhn, *The Structure of Scientific Revolutions*

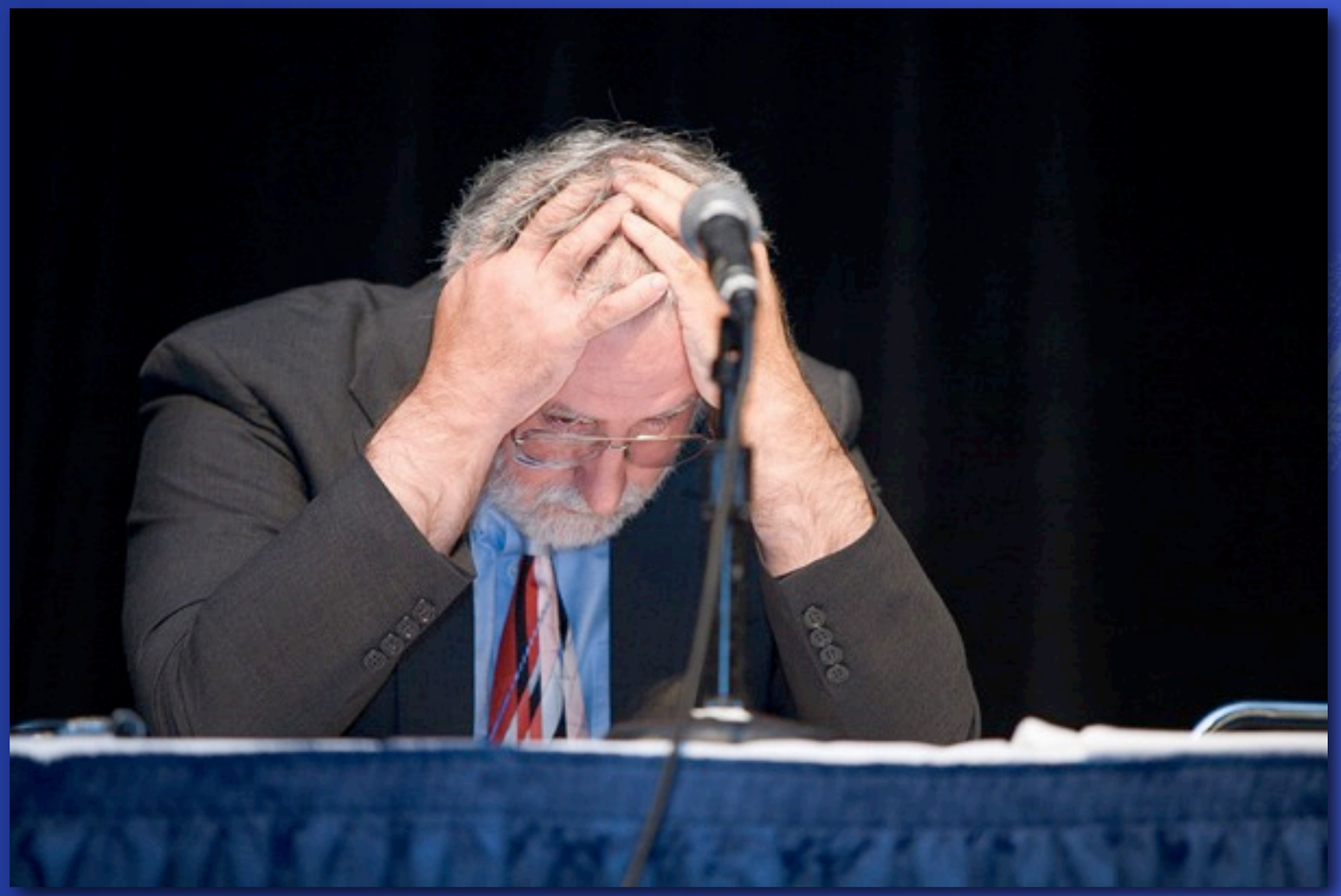
Method Combination

Although there are several significant aspects of CLOS inheritance, we focus only on standard method combination and primary methods.

–Bracha & Cook, *Mixin-Based Inheritance, 1990*

In CLOS, mixins are simply a coding convention and have no formal status. Although locally unbound uses of call-next-method *are a clear indication that a class is a mixin, the concept has no formal definition, and any class could be used as a mixin if it contributes partial behavior.*

–Bracha & Cook, *Mixin-Based Inheritance, 1990*



- CLOS doesn't mention the word "mixin"
- CLOS has generic functions, not methods with messages sent to them
- (standard) method combination defines method types which play specific roles; in general, classes with only methods that declare specialized roles are considered mixins

There are three types of methods for this combination [standard method combination]: primary, before, *and* after. *Untyped methods default to* primary *type.*

–Howard I. Cannon, *Flavors, 1979*

```
(defmethod display :before (message (w announcement-window))
                   (expose-window w))
```

CLOS::DOCUMENTATION-SLOT NIL
FUNCTION #<interpreted function (METHOD DISPLAY :BEFORE (T ANNOUNCEMENT-WINDOW)) 20711DD2>
GENERIC-FUNCTION #<STANDARD-GENERIC-FUNCTION DISPLAY 20711DBA>
LAMBDA-LIST (MESSAGE W)
CLOS::PLIST NIL
CLOS::QUALIFIERS (:BEFORE)
CLOS::SPECIALIZERS (#<BUILT-IN-CLASS T 207B1043> #<STANDARD-CLASS ANNOUNCEMENT-WINDOW 20713DD3>)

A *base flavor* serves as a foundation for building a family of flavors. It defines instance variables, sets up defaults, and is often not instantiable. A *mixin flavor* is one that implements a particular feature, which may or may not be included. Mixins are *almost never instantiable*, often have the word mixin in their name, and often define a handful of instance variables.

When an essentially orthogonal feature is to be implemented, a new mixin is defined, with no component flavors.

–Howard I. Cannon, *Flavors, 1979*

...These inheritance mechanisms are subsumed in a new inheritance model based on composition of mixins, or abstract subclasses....

–Bracha & Cook, *Mixin-Based Inheritance, 1990*

When a generic function is applied to an object of a particular flavor, methods for that generic function attached to that flavor or to one of its components are available. From this set of available methods, one or more are selected to be called. If more than one is selected, they must be called in some particular order and the values they return must be combined somehow.

–David A. Moon, *OOP with Flavors, OOPSLA 1986*

The method-combination type sorts the available methods according to the component ordering, thus identifying more specific and less specific methods. It then chooses a subset of the methods (possibly all of them). It *controls how the methods are called and what is done with the values they return* by constructing Lisp code that calls the methods. Any of the functions and special forms of the language may be used. The resulting function is called a *combined method*.

–David A. Moon, *OOP with Flavors, OOPSLA 1986*

Programmers control method combination separately from the definition of the methods themselves. They control it by declaring a method-combination type and constraints on the ordering of component flavors.

When defining a method, the programmer only thinks about what that method must do itself, and not about details of its interaction with other methods that aren't part of a defined interface. When specifying a method-combination type, the programmer only thinks about how the methods will interact, and not about the internal details of each method, nor about how the method-combination type is implemented. Programming an individual method and programming the combination of methods are two different levels of abstraction. Keeping them separate promotes modularity.

–David A. Moon, *OOP with Flavors, OOPSLA 1986*

Like an ordinary Lisp function, a generic function takes arguments, performs a series of operations, and perhaps returns useful values.... A generic function has a set of bodies of code of which a subset is selected for execution. The selected bodies of code and the manner of their combination are determined by the classes or identities of one or more of the arguments to the generic function and by its method combination type.

–DeMichiel et al, *The Common Lisp Object System Specification, 1988*

The role of each method in the effective method is determined by its method qualifiers and the specificity of the method.

Primary methods define the main action of the effective method, while auxiliary methods modify that action in one of three ways. A primary method has no method qualifiers. An auxiliary method is a method whose method qualifier is :before, :after, *or* :around.

–DeMichiel et al, *The Common Lisp Object System Specification, 1988*

Mixin class: *A descriptive term for a class intended to be a building block for other classes. It usually supports some aspect of behavior orthogonal to the behavior supported by other classes in the program; typically, this customization is supported in before- and after-methods. A mixin class is not intended to interfere with other behavior, so it usually does not override primary methods supplied by other classes.*

–Sonya E. Keene, *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS, 1989*

In CLOS a mixin is a class with only auxiliary (:around, :before, and :after) methods defined on it

In CLOS...locally unbound uses of call-next-method *are a clear indication that a class is a mixin....*

–Bracha & Cook

An error is signaled if call-next-method *is used in a* :before *method.... An error is signaled if* call-next-method *is used in an* :after *method.*

–DeMichiel et al, *The Common Lisp Object System Specification, 1988*

Bracha & Cook

- Stupid people? NO!
- Careless scholarship? NO!
- Barely observable but incontrovertible evidence of the reality of incommensurability?
- ...maybe...yeah, maybe...

Engineering(,) a Path to Science

“I Don’t Want to Die in a Language I Can’t Understand”

Engineering(,) a Path to Science

“I Don’t Want to Die in a Language I Can’t Understand”