The Structure of a Programming Language Revolution

> *"I don't want to die in a language I can't understand."*
> – Jorge Luis Borges

Richard P. Gabriel
IBM Research
Redwood City, California USA
rpg@{us.ibm.com, dreamsongs.com}

Abstract

Engineering often precedes science. Incommensurability is real.

Categories and Subject Descriptors A.0 [General]

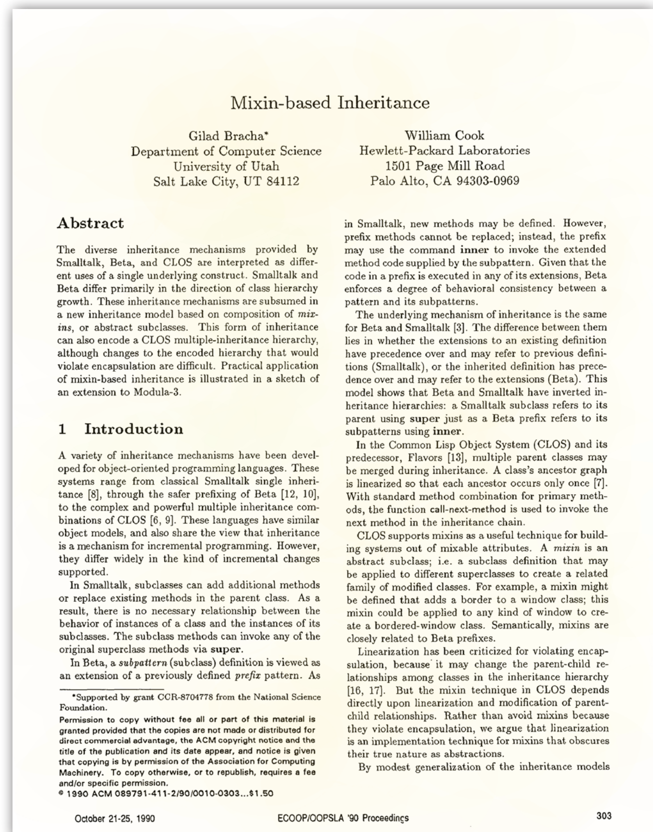
General Terms Design

Keywords Engineering, science, paradigms, incommensurability

In 1990, two young and very smart computer scientists—Gilad Bracha and William Cook—wrote a pivotal paper called "Mixin-based Inheritance" [1], which immediately laid claim to being the first scientific paper on mixins. In that paper they described looking at Beta, Smalltalk, Flavors, and CLOS, and discovering a mechanism that could account for the three different sorts of inheritance found in these languages—including mixins from Flavors and CLOS. They named their new mechanism "mixins."

My attention was directed to this paper by Gilad Bracha himself when he told me in Brazil at AOSD in the spring of 2011 that most Lisp people who read the paper had strong objections to what he and William Cook had written about Lisp and CLOS.

That night I pulled the paper down from the ACM server and read it while outside enormous puffed clouds dwelled overhead, lit from beneath by the town of Porto de Galinhas on the Brazilian coast; the smells of burning sugarcane and bitter ocean pushed into my room.



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 object models, and also share the view that inheritance is a mechanism for incremental programming. However, they differ widely in the kind of incremental changes supported.

In Smalltalk, subclasses can add additional methods or replace existing methods in the parent class. As a result, there is no necessary relationship between the behavior of instances of a class and the instances of its subclasses. The subclass methods can invoke any of the original superclass methods via **super**.

In Beta, a *subpattern* (subclass) definition is viewed as an extension of a previously defined *prefix* pattern. As 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.

CLOS supports mixins as a useful technique for building systems out of mixable attributes. A *mixin* is an abstract subclass; i.e. a subclass definition that may be applied to different superclasses to create a related family of modified classes. For example, a mixin might be defined that adds a border to a window class; this mixin could be applied to any kind of window to create a bordered-window class. Semantically, mixins are closely related to Beta prefixes.

Linearization has been criticized for violating encapsulation, because it may change the parent-child relationships among classes in the inheritance hierarchy [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

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1990 ACM 089791-411-2/90/0010-0303...$1.50

Engineering$_{(,)}$ A Path To Science

Engineers build things; scientists describe reality; philosophers get lost in broad daylight.

What I read in Brazil reminded me of my quest to demonstrate that in the pursuit of knowledge, at least in software and programming languages, engineering typically precedes science—that is, even if science ultimately produces the most reliable facts, the process often begins with engineering.

I believe it's a common belief that engineers only follow paths laid down by scientists, adding creativity and practical problem solving. Philip Kitcher, a philosopher of science at Columbia University, in an essay for the New York Times

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Onward! 2012, October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1562-3/12/10…$15.00.

Book Review on March 25, 2012 ("Seeing is Unbelieving") [2], wrote it this way:

> *The natural sciences command admiration through the striking successes of the interventions into nature they enable: satellites are sent into space, new tools forged to combat disease. Those achievements rest on the ability to develop rigorous chains of inferences that take us from readily detectable aspects of the world to reliable conclusions about more remote matters. As the conclusions become established, they often yield new methods of detection: a novel theory inspires instruments like microscopes and spectrometers that expand the range of the senses.*
>
> –Philip Kitcher, *Seeing is Unbelieving*, 2012 [2]

Sheldon Cooper, a character on *The Big Bang Theory* said it this way:

> *Engineering, where the noble, semi-skilled laborers execute the vision of those who think and dream. Hello Oompa Loompas of science.*
>
> –Sheldon Cooper (character), "The Jerusalem Duality," *The Big Bang Theory*, 2008 [3]

I believe that, in general, this view of engineering and science is false: I believe engineering and science are intertwined, and for programming languages and software creation techniques, it's often the case that engineering precedes science—and it's very easy to see it. Let's look at some definitions of engineering.

> *…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, 2012 [4]

> *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, 2012 [5]

> *Engineering is the practical application of science and math to solve problems….*
>
> –http://en.wikipedia.org/wiki/Engineering, 2012 [5]

(Scientific) knowledge, though, comes from interacting with the world or the subject of investigation. Go to an auto mechanic, describe your problem, and the mechanic almost immediately says "let's take a look," followed by peering, moving cables aside, accessing innards, running tests, starting the engine and listening, taking measurements (sometimes using sophisticated sensors), and otherwise interacting directly with the thing itself. Isn't this like science?—a mystery in front of us, and we poke and prod in search of a theory, a guess that explains it?

> *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* [6]

One good example is the steam engine. Engineers began its development while scientists were making their way from the ***phlogiston*** theory of combustion to the ***caloric*** theory of heat, both today considered hilarious.

Phlogiston theory was an attempt to explain oxidation—fire and rust. The phlogiston theory held that combustible resources contain phlogiston, a colorless, odorless, tasteless, massless substance, which is liberated by burning. A *phlogisticated* substance contains phlogiston and is *dephlogisticated* when burned, leaving behind its "true" form: *calx*.

> *…[S]ubstances that burned in air were said to be rich in phlogiston; the fact that combustion soon ceased in an enclosed space was taken as clear-cut evidence that air had the capacity to absorb only a definite amount of phlogiston. When air had become completely phlogisticated it would no longer serve to support combustion of any material, nor would a metal heated in it yield a calx; nor could phlogisticated air support life, for the role of air in respiration was to remove the phlogiston from the body.*
>
> –James Bryan Conant, ed.
> *The Overthrow of Phlogiston Theory: The Chemical Revolution of 1775–1789.*
> Cambridge: Harvard University Press, 1950 [7]

Phlogiston seems like the opposite of oxygen as far as combustion is concerned. In the 18th century, scientists noticed that when metals became oxidized (rusted), they got heavier, throwing the theory into question. For a time after that, phlogiston was taken to be a principle and not a material substance. Later this theory was more or less replaced by the caloric theory, which held that heat consisted of a fluid called *caloric* that flows from hotter to colder bodies. Caloric is a weightless gas that can pass in and out of pores in solids and liquids.

Funny as these theories sound today, steam engines became practical during their dominance in science, and im-

portant other scientific discoveries were enabled by them: the speed of sound was better estimated as the theory of caloric was refined, and the idea that metabolism was related to combustion was a direct tenet of the phlogiston theory.

Steam engines, though, kept huffing and puffing along through the tail end of phlogiston, the entirety of caloric, and now into thermodynamics.

> *…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* [6]

The relative roles of science and engineering are complex. Scientists try to understand while engineers try to build. Engineering is the creative application of scientific principles to design or develop new stuff.

> *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.*
> –http://en.wikipedia.org/wiki/Engineering, 2012 [5]

What's the effect of these differing opinions? What happens when engineering is reduced to science's handmaiden? In the fields of programming languages and software, and in computer science in general, the effect has been to separate engineers from scientists and put them into a little hierarchy—engineers are for the most part left out of the (lofty) scientific academy.

Nevertheless, people like Matthew Crawford see things differently, and I do too.

I am going to use the Bracha & Cook paper as a lens through which to see what it means in computer science for engineering to precede science in the production of scientific facts—a case study. We will learn that it's absolutely natural, and has worked wonderfully.

The case study centers on the discovery or invention of *mixins* and *method combination*, spearheaded by Lisp programmers, researchers, and engineers beginning in the mid-1960s and progressing through the 1980s. Mixins were introduced in 1979 as the key idea behind Flavors. In 1989 CLOS—the Common Lisp Object System, a descendant of Flavors—was officially added to the Common Lisp standard. CLOS had a version of method combination along with a(n unnamed) notion of mixins.

Let's dig in.

What is a mixin? This will be our first encounter with the language difficulties this case study raised for me. There are at least two understandings of the term "mixin" at work. The case study, thus, began bumpy and stayed that way. I came to realize that the Bracha & Cook paper is a dividing point, or perhaps a bridge, between two different paradigms—or perhaps "micro-paradigms." This observation further raised the question of *incommensurability*, a key part of Kuhn's ideas about scientific revolutions [8]. Beginning with the word "mixin," I was stumbling onto this idea too—that the notion of incommensurability can explain some of the mysteries in the Bracha & Cook paper. I'll tell you about that later, but first I need to explain (old-style) mixins. Along the way we'll see that engineering knowledge and scientific knowledge are not always of a kind.

In the earlier micro-paradigm, the term "mixin" was part of a constellation of concepts related to combining behavior without requiring source code to be written. That is, you just declared that a component to be a mixin, and a mechanism behind the scenes combined the behavior of applicable methods for you. In the later micro-paradigm, the term mixin refers to any abstract (uninstantiatable) class along with explicit invocation of other "mixed-in" behavior. The following is the current definition:

> *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, 2012 [9]

Because we're going to be looking at the engineering predecessors of the modern idea of mixins, we'll start with the earlier definition. Pre-1990 mixins have to do with two other mechanisms: *multiple inheritance* and *method combination*.

Code. Check out Figure 1 on the next page.

CLOS is part of Common Lisp, and Common Lisp has a Read-Eval-Print Loop (REPL). Definition 1 defines a class named `person` which has a slot accessed by a method named `name`. This class has no declared superclasses. But behind the scenes it does, including one that defines a generic function called `print-object` which is what all print functions eventually call. It simply outputs the person's name.

At `CL-USER 31` we see how the REPL works and the result of printing the instance that represents me.

Now let's define another class called `graduate`. This class inherits from `person` because every graduate we care about is human. And as it turns out, `graduate` is a mixin. The way we can tell is that the method we've defined for `print-object` is an `:after` method. An `:after` method is one that is called

```
Definition 1: (defclass person () ((name :accessor name :initarg :name :initform “”)))

Definition 2: (defmethod print-object ((person person) stream)
               (format stream “~A” (name person)))

CL-USER 31 > (setq rpg (make-instance ‘person :name “Richard P. Gabriel”))
Richard P. Gabriel

Definition 3: (defclass graduate (person) ((degree :accessor degree :initarg :degree :initform “”)))

Definition 4: (defmethod print-object :after ((person graduate) stream)
               (format stream “ ~A” (degree person)))

CL-USER 32 > (setq rpg (make-instance ‘graduate :name “Richard P. Gabriel”
                                                :degree “PhD”))
Richard P. Gabriel PhD

Definition 5: (defclass doctor (person) ())

Definition 6: (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

Definition 7: (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
```

Figure 1

after all the applicable and invoked primary methods for the generic function have been executed. It runs *after*.

At CL-USER 32 we see how this works. I’ve made an instance of `graduate` with my name and having a PhD, and without any source code apparently being written by me, my name and degree get printed out together in the right order.

Now a similar new mixin, this for people who have a doctorate of some sort. Again it’s a mixin—the `:before` method runs before any other applicable methods are invoked.

Finally the money shot. Definition 7 creates a class named `research-doctor` with two parents, and because of method combination, printing my name in all its inherited glory happens correctly without my having to say anything at all.

See Figure 2 for the class diagram for this example (arrows point from superclass to subclass).

This example typifies but doesn’t exemplify how mixins, multiple inheritance, and method combination work in Flavors and CLOS. It shows how the mechanism works, but it exemplifies poor design.

The beginning of this essay contains a picture of the first page of the Bracha & Cook paper, considered to be the first scientific paper about mixins.

The primary engineering papers that preceded it are by David Moon (“Object-Oriented Programming with Flavors” [10]) and Howard Cannon (“Flavors: A Non-Hierarchical Approach to Object-Oriented Programming” [11]).

Citation counts are a standard measure of scientific relevance in computer science. The table on the next page shows citation counts for these papers. Notice that the Bracha & Cook paper has twice the number of Google Scholar citations as the Moon paper; and the Cannon paper, which first defined multiple inheritance, method combination, and mixins, is hardly cited at all—in fact, CiteSeer doesn’t even notice the Moon and Cannon papers.

Bracha & Cook are careful about their claims, stating principally that they have defined a new primitive called “mixins”

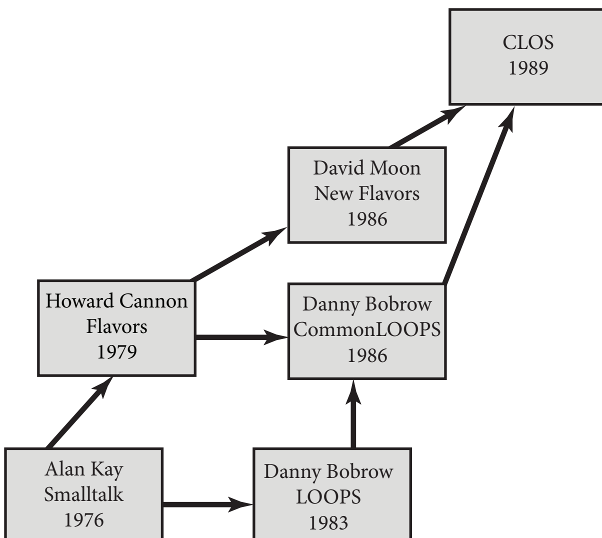
Bracha & Cook	CiteSeer	375
	ACM DL	203
	Google Scholar	842
Moon	CiteSeer	<not found>
	ACM DL	102
	Google Scholar	402
Cannon	CiteSeer	<not found>
	Google Scholar	55

which can be used to define the inheritance mechanisms of Smalltalk, Beta, and CLOS, as they understand them.

> *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 [1]

Historically, Flavors, Smalltalk, and CLOS are related as shown in the diagram below. The Bracha & Cook paper was published in 1990.

Scientists study and try to explain nature. Even if we assume that software and programming language engineers use then-current science to invent things, those new things become part of the nature that subsequent computer scientists study. Bracha & Cook were studying the reality created by Birger Møller-Pedersen, Kristen Nygaard, Howard Cannon, David Moon, Danny Bobrow, and the designers of CLOS.



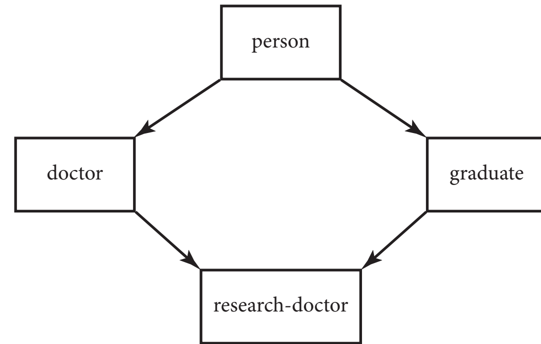


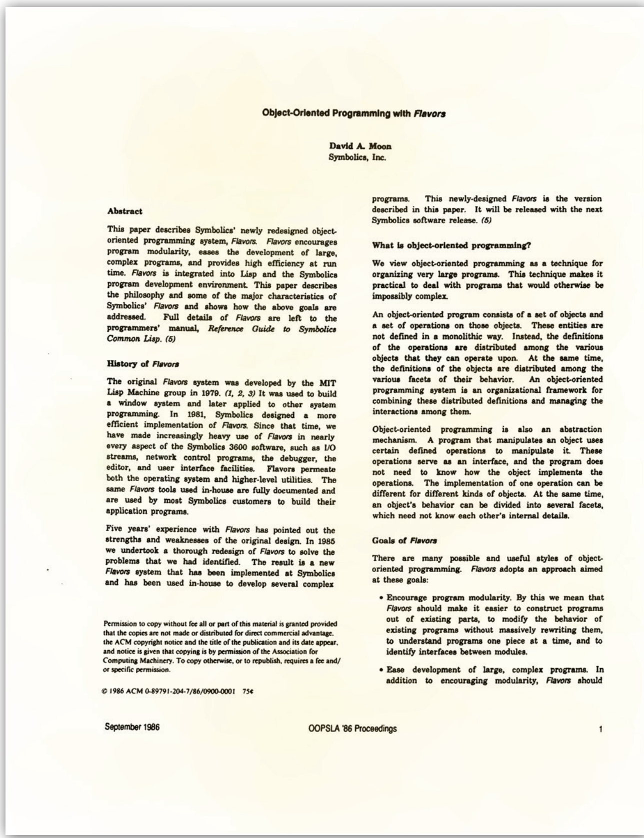
Figure 2

From this (engineering) reality, Bracha & Cook came up with a theory of mixin-based inheritance, creating a new (scientific) reality.

Engineers and scientists understand these two realities differently, using different vocabularies and more than that, different language. Before 1990, programming language engineers and programming language scientists published their work in the same conferences and publications, and so they had some basis to understand each other. In the early 1990s this changed and engineering papers were effectively banned from scientific conferences and publications.

"Object-Oriented Programming with Flavors," by David Moon was the first engineering publication about mixins, being published in the first OOPSLA conference in 1986. Moon's paper described an evolution of earlier work done by Howard Cannon who informally published his paper on Flavors at the MIT AI Lab. Cannon's paper was the first to describe the concepts of mixins, multiple inheritance, and method combination as we know them today. Notice that Cannon's paper was written in 1979.

The MIT AI Lab at that time was in the middle of a temporary paradigm in which *heterarchy* was preferred to hierarchy. This was the milieu in which Minsky's "Society of Mind" emerged. It was the idea that a population of agents and critics could more effectively demonstrate and give rise to intelligence than a top-down, centralized approach. It was a small revolution against hierarchy, and it's reflected in the title of Cannon's paper: "Flavors: A Non-Hierarchical Approach to Object-Oriented Programming." At that time at the MIT AI Lab, Smalltalk was well known, as was the Actor model developed by MIT's Carl Hewitt. The most famous exploration of what message-passing means in actor-like object systems was done by Gerry Sussman and Guy Steele; it was called "Scheme."



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*

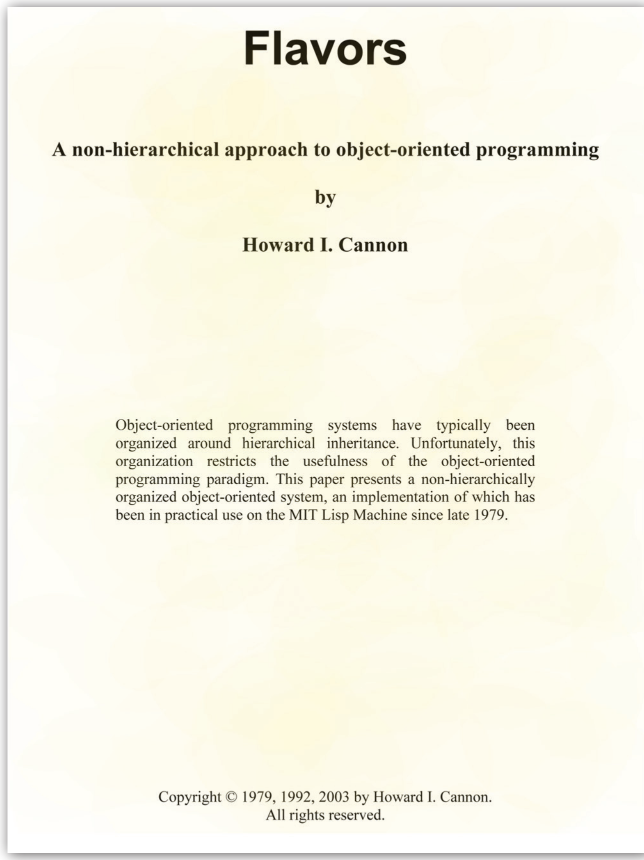
There are many possible and useful styles of object-oriented programming. *Flavors* adopts an approach aimed at these goals:

- 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, to understand programs one piece at a time, and to identify interfaces between modules.
- Ease development of large, complex programs. In addition to encouraging modularity, *Flavors* should

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1986 ACM 0-89791-204-7/86/0900-0001 75¢

September 1986 OOPSLA '86 Proceedings 1



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.

Copyright © 1979, 1992, 2003 by Howard I. Cannon.
All rights reserved.

But Cannon's paper was not the first to talk about, essentially, mixins. Warren Teitelman's dissertation did ("Pilot: A Step Toward Man-Computer Symbiosis" [12]), and it was published in 1966, 24 years before the Bracha & Cook paper.

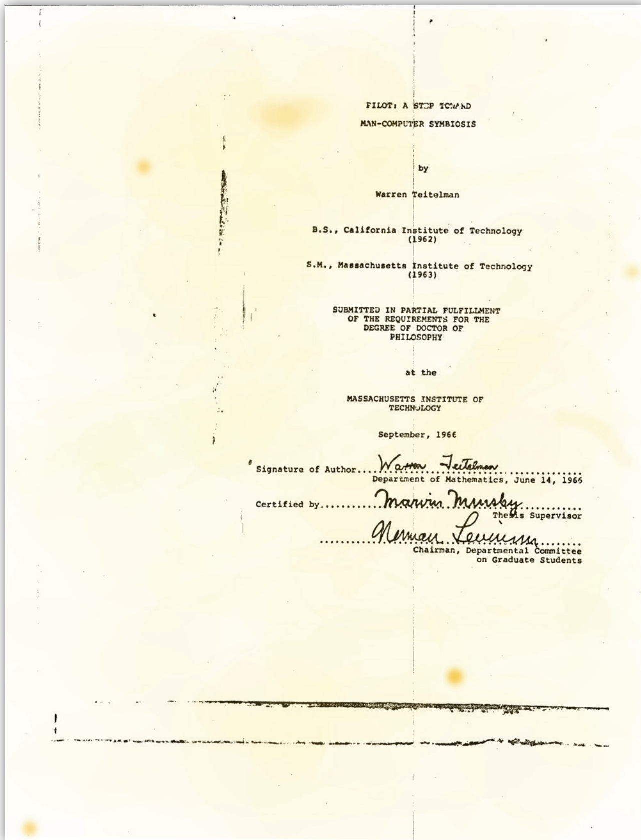
This demonstrates that at least in the case of mixins, engineering can precede science by quite a margin. We've looked at some of the concepts through examples, and now we'll look at some of the concepts and principles more abstractly. —Because the scientific view that's emerged is that these early, engineering efforts were not based on solid scientific principles, and perhaps not even on good engineering or design principles. These guys were, after all, just hackers hacking things together, so what could you expect?

Teitelman's dissertation was about a programming system called *Pilot* which assisted human programmers. His early version of mixins was called "advice," and even today in aspect-oriented programming we hear that term. The idea is that a procedure can be inserted at any or all entry or exit points of any other particular procedure or class of procedure, hence combining behavior. Advice not only provides *before* and *after* functionality, but also what would become known in CLOS as *around* methods.

> *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

Here is the syntax of advice as envisioned in 1966.

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



PILOT: A STEP TOWARD

MAN-COMPUTER SYMBIOSIS

by

Warren Teitelman

B.S., California Institute of Technology
(1962)

S.M., Massachusetts Institute of Technology
(1963)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF
PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF
TECHNOLOGY

September, 1966

Signature of Author........................
Department of Mathematics, June 14, 1966

Certified by........................
Thesis Supervisor

....
Chairman, Departmental Committee
on Graduate Students

Cannon's original Flavors system was aimed at non-hierarchical object systems. Although he was interested in independently specified behavior associated with a single flavor, he observed that when behaviors don't interact it's easy to combine them—by using disjunct messages and instance variables.

The interesting complication comes when there are interactions between mostly orthogonal issues and the desire is to have modular interactions. The example he used was a window with a border and a label. These mostly independent but modular aspects interact by having to coordinate their sizes and positions, and to respond to the same message around the same time.

> *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 [11]

Check out the 1979 code for this example (Figure 3).

First the window is defined with a `:refresh` method (Definitions 1 & 2). Then a border with an `:after` method for `:refresh` (Definitions 3 & 4). Next the label, again with an `:after` method for `:refresh` (Definitions 5 & 6). Finally the mixed together thing: a window with border and label (Definition 7).

Here is how the thing acts when a `:refresh` message is sent to the mixed-together contraption:

First the window clears itself, and presumably the text is redrawn (though the code doesn't show that).

Then the border clears and redraws itself.

Finally the label does too.

This is an example of method combination and decent design of that era. Cannon tells us that in ordinary class-based systems, the method to run can be determined at method-invocation time because that requires only local knowledge. But with Flavors, all relevant component Flavors need to be examined to create a combined method, and the earliest this examination can take place is when the Flavor is instantiated—because that's when the applicable methods are known.

> *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 [11]

Note that Cannon is careful to talk about the need to "modularly integrate essentially orthogonal issues." This doesn't sound like a cowboy hacker.

The example shows three types of method: `:before`, `:after`, and **primary**. Method combination puts them together. A mainline flavor combines with orthogonal side-flavors that add behavior to the main flavor and enough state for minimal coordination. Good design and modularity demand something like this. The poor design in the `research-doctor` example—according to the Flavors notion of good design—is that the traits of being a `doctor`, a `graduate`, and a `person` are not essentially orthogonal as defined.

> *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 appropri-*

Definition 1: `(defflavor WINDOW (OBJECT) (X-POSITION Y-POSITION WIDTH HEIGHT))`

Definition 2: `(defmethod (WINDOW :REFRESH) (send SELF ':CLEAR))`

Definition 3: `(defflavor BORDER () (BORDER-WIDTH))`

Definition 4: `(defmethod (BORDER :AFTER :REFRESH) () (send SELF ':DRAW-BORDER))`

Definition 5: `(defflavor LABEL () (LABEL))`

Definition 6:
```
(defmethod (LABEL :AFTER :REFRESH) () (send SELF ':DRAW-LABEL))
                                      (send SELF ':DRAW-LABEL))
```

Definition 7: `(defflavor WINDOW-WITH-LABEL-AND-BORDER (LABEL BORDER WINDOW) ())`

```
(defflavor WINDOW (OBJECT) (X-POSITION Y-POSITION WID
(defmethod (WINDOW :REFRESH) (send SELF ':CLEAR))
(defflavor BORDER () (BORDER-WIDTH))
(defmethod (BORDER :AFTER :REFRESH) () (send SELF ':D
(defflavor LABEL () (LABEL))
(defmethod (LABEL :AFTER :REFRESH) () (send SELF ':DR
                                      (send SELF ':DR
(defflavor WINDOW-WITH-LABEL-AND-BORDER (LABEL BORDER
```
`Window.lisp`

Figure 3

> *ate 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[s] 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 [11]

Note that there can be several types of method combination. Note also that methods are divided into types which a method combination type uses to determine the behavioral roles of the different methods.

Cannon Conventions

> *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 [11]

Here Cannon is talking about programming in a way that illuminates the paradigm he was working in. In Cannon's community of practice, good design was important, and the set of available tools and underlying mechanisms needed to be flexible enough to express such a design when it came along. Moreover, a good design can live on many programming substrates using conventions if they are adaptable enough—"`structs` with an attitude" was usually enough to support OOP. Modularity was near the top of the list of traits of good design for that community, and Flavors was aimed right at it.

I knew Howard back then. He was very young—about 20 years old. And I remember when someone emailed me his paper—Howard was at MIT and I at Stanford. I remembered the excitement, and I remembered how hard it was for me to understand the paper. He was an undergraduate and I a graduate student of seven years at that time.

Youthful geniuses are not unknown. Here is the poet Arthur Rimbaud [13]…

> 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.

…writing at sixteen years old.

Around this time a number of companies began producing Lisp machines. In the United States in the 1980s, there were five such companies: Symbolics, Lisp Machines Incorporated, Three Rivers, Texas Instruments, and Xerox. Symbolics, which grabbed most of the Lisp talent from MIT, pushed Flavors forward with improvements and extensions, bringing it more in line with Lisp. David Moon was one of the chief designers and implementers of New Flavors, which was the topic of the 1986 Moon Flavors paper at OOPSLA.

Flavors Well-Organized Programs

> *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 [10]

The Flavors system doesn't seem too big on static concerns. There is no hard compiler control of what the language means, just local checking of syntax at read time and simple compiler checks. Contrary to some popular beliefs, the underlying Lisp is not untyped. Objects in memory have runtime types associated with them. At the time, the terms for typing were as follows: a *statically typed language* has its type correctness checked at compile time; a *dynamically typed language* has its type correctness checked at run time; and in a *strongly typed language* it is not possible for an operator to be applied to arguments of inappropriate types. Lisp is a strongly typed, dynamic language.

Themes shared by Cannon and Moon—and prevalent throughout the Flavors / CLOS literature—are good design, modularity, and orthogonal concerns.

Constraints On Component Flavors

> *Each flavor defines certain constraints on the ordering of itself and its direct 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 [10]

When each flavor contributes a distinct chunk of behavior—ideally orthogonal chunks of behavior—it makes sense to linearize a multiple inheritance graph, because there will be no unexpected interactions between behaviors: each contributing flavor will be as if in its own single inheritance chain. And having a single chain to reason about simplifies method-combination computations—it represents an engineering tradeoff. See Appendix: **Fine Print**.

In defining a flavor, the super-flavors mentioned don't describe direct inheritance but constraints on the order of shadowing.

Here are the rules for linearization used in New Flavors. (These changed slightly in CLOS to fix a weird and delightfully convoluted corner case.)

- *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 [10]

Modularity was a major concern, and in fact was the driving force behind the design of flavors. Those designers—and I was friends with each of them—were as hardcore in their faith in modularity as anyone. But for them, modularity was the result of careful design, not magic enforced by a compiler or a system. Any underlying programming system can, at best, ease creating good designs.

Flavors System Guarantees

> *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*, OOPSLA 1986 [10]

Those were my observations after reading the Bracha & Cook paper in Brazil that night—a clear path beginning in engineering inventiveness driven by a keen desire to support good design and modularity. The vocabulary of those early engineering papers seemed a bit dated, but I lived in that world for decades at the beginning of my career. What I read supported the idea that engineering could and often *did* precede science in the realm of programming languages.

My own experiences suggested something else, too. In the 1990s it seemed to me that scientists in the programming community pulled back the welcome mat from engineers. I noticed that the kinds of papers that could be published then were different from when I was publishing at a good clip in the 1980s. By the mid-1990s it was clear that a paper like Moon's from 1986 would never be published at OOPSLA or any other programming language conference.

In the mid-1990s I went back to school and got my MFA in Creative Writing, believing that when I finished I would take up my research career again. When I returned in 1998 I was startled to learn that my field—advances in Lisp-like languages approached in an engineering manner—had been deleted, literally. The conference I routinely published in deleted "Lisp" from its name in 1996 (*Lisp and Function Programming* became the *International Conference on Functional Programming*); the journal I founded with Guy Steele deleted "Lisp" from its name in 1997 (*Lisp and Symbolic Computation* became *Higher Order and Symbolic Computation*). And when I attended OOPSLA, I found I couldn't understand any of the papers. While in the 1980s, engineering-centric researchers could publish implementation or new-language papers at OOPSLA, in the late 1990s such papers went into "Practitioner Reports." In the 2000s, some universities introduced a new faculty level, *Professor of (the) Practice*, extended to those who had "developed a high level of expertise in fields of particular importance" to the university in question. But some of those schools enforced a stipulation that reflects the hesitancy with which the title is granted: "appointees will hold the rank of Professor but, while having the stature, will not have rights that are limited to tenured faculty" [14].

It seemed like some kind of paradigm shift in the Kuhnian sense happened. But what was it?

"I Don't Want to Die in a Language I Can't Understand"

Let's start again.

…That night I pulled the paper down from the ACM server and read it while outside enormous puffed clouds dwelled overhead, lit from beneath by the town of Porto de Galinhas on the Brazilian coast; the smells of burning sugarcane and bitter ocean pushed into my room. I fell asleep after two or three *attempts to understand the apparently nonsensical passages* I *encountered in* that *old scientific text*.[1]

1. Italic—see quote *Kuhn Discovers Incommensurability* below.

Later, back from Brazil, I went back and looked more closely at the Bracha & Cook paper: the passages in their paper on mixins in CLOS and Flavors were nonsensical, as if the authors were *confused* or held *mistaken beliefs*.[1]

But Bracha & Cook are smart guys. They were young when they wrote the paper, but they graduated from good schools and good CS programs. I on the other hand was clearly starting to head down the drain. It all reminded me of Feyerabend talking about using voodoo as a starting place for a new take on a scientific field—"anything goes," as he wrote [15].

Later that spring, the New York Times published a long essay in five parts called "The Ashtray" by Errol Morris [16]. It was about incommensurability—Kuhnian incommensurability—the idea that paradigms exist in different, mutually unfathomable linguistic domains. In the end, Errol Morris rejected incommensurability. He wrote: "The past may be a foreign country, but I do not believe that people there speak a language that we can not understand."

Looking at the Bracha & Cook paper, I felt I was staring into the unsmiling face of modern day incommensurabilty. And examining the paper through the lens of incommensurability, I was struck by a couple of things. When discussing CLOS, Bracha & Cook get several terms wrong and seem to ignore others. When talking about method combination, they throw out the very mechanisms that support what they want to focus on: mixins. Their choice of papers to reference seems odd: a paper about Flavors [10], a brief overview of CLOS from an ECOOP paper [17], and a programmer's guide written by a technical writer at Symbolics [18]. This was after the CLOS specification had been widely publicized—it was published in full in September 1988 in SIGPLAN Notices [19].

My first thought was that this paper tagged the pivot point of a Kuhnian paradigm shift.[2] This was bolstered by my belief that engineering papers had become rather suddenly unwelcome at programming language conferences around the same time.

My initial hypothesis was that the paradigms in question were *engineering* on one side and *science* on the other. But the paradigms in question turn out to be more technically focussed.

But: how fascinating! —That incommensurability could be real. I had lived through this micro-paradigm shift, and my realization came as a surprise because it explained so much while remaining hidden from me all these years.

The real paradigm shift? *Systems* versus *languages*. Before 1990, a person interested in programming could work comfortably both in programming languages and in programming systems, but not so easily after. To tell the truth,

2. A paradigm shift is not a clean demarcation between past and future—paradigms co-exist. The Newtonian paradigm is still used for many common calculations.

I never even noticed the different words—*language* versus *system*—never attached any significance to the word choice until this exploration. I used the phrases interchangeably for many years: Programming System / Programming Language. Bracha & Cook wrote the following (underlines are mine):

> *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 [1]

Bracha & Cook seem confused about the terms in this quote. Perhaps the use of "systems" is an oversight. Maybe the word "mechanisms" should have been repeated? Or maybe languages and systems *at that time* were unclear or confused concepts.

But mostly Bracha & Cook write of languages, while Cannon, Moon, and the CLOS guys write of systems. The following are definitions of language and system:

> *[**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/, 2012 [20]

A system is a set of interacting components, though sometimes the interaction is in the realm of ideas—and thus a language can also be a system. But the usual case requires a set of actual, observable behavior. A real set of things doing real stuff. —Even if in a computer.

A language is a system of signs but for the purpose of conveying meaning. A language is words on the page. Grammatical correctness is important to a language.

You can see the overlap in meaning. But the difference is clear: systems are about things happening, and languages are about conveying meaning.

As we've seen in quotes from Moon and Cannon, for systems, good design by a human designer is essential, and though the system can go only so far to help you, it should go some distance. Today, one of the goals of *programming language* designers is to make some kinds of bad or poor design ungrammatical, thereby cutting them off.

Recall that Howard Cannon wrote that when conventions—that is, good design principles—are supported by tools which make them easier to follow, you have a programming *system* (see quote: *Cannon Conventions*).

The Moon paper speaks of a programming system: here…

> *This paper describes Symbolics' newly redesigned object-oriented programming system, Flavors. Flavors encourages program modularity,….*
>
> –David A. Moon, *OOP with Flavors*, OOPSLA 1986 [10]

…and here:

> *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*, OOPSLA 1986 [10]

But note the slight ambiguity: "an organizational framework for combining these distributed definitions and managing the interactions among them" can refer to a system of thought, but nevertheless, "managing interactions" sounds very physical.

The "S" in CLOS stands for "System." The Common Lisp Object *System*.

The Bracha & Cook paper is one of the papers that marks the demise of the System Paradigm and the rise of the Language Paradigm. Of course, both paradigms existed since the early 1960s, and both still exist, but while the System and Language paradigms were of essentially equal prominence *before* 1990, *after*, the System Paradigm disappeared almost completely until the mid-2000s.

The Lisp world from which Pilot, Flavors, and CLOS emerge naturally falls into the System Paradigm—to do well with Lisp requires system thinking. Syntax—a hallmark in the Language paradigm—is relatively unimportant for Lisp. Lisp is about execution because you can "feel the bits between your toes" [21]. Alan Perlis said it most eloquently, I believe, when he wrote the following:

> ***Pascal** is for building pyramids—imposing, breathtaking, static structures built by armies pushing heavy blocks into place. **Lisp** is for building organisms….*
>
> –Alan Perlis [22]

When working with a system one must explicitly attend to careful design, good organization, and modular thinking. In Lisp, the underlying system is designed to help you. And your design thinking is effected by altering the living, running system right in front of you.

CLOS is decidedly a system designed to create and manipulate executing systems. Here's how you know: CLOS

defines a protocol for updating class objects and instances, while retaining identity, when a class is redefined. Were CLOS a programming language, no protocol would be needed because the programmer would simply recompile and rebuild the program—such a change is a text editing chore, not a system update: CLOS describes how the affected objects in the running system should be updated and how the programmer can determine how best to do this in the context of the system's domain.

> *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 an instance does not change its identity.…*
>
> –Daniel Bobrow et al, *The Common Lisp Object System Specification*, 1988 [19]

Similarly, CLOS supports a related protocol for changing the class of an object.

> *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}.
>
> –Daniel Bobrow et al, *The Common Lisp Object System Specification*, 1988 [19]

In a programming language there would be no such thing. CLOS is a system, and the meaning of "system" is a running, executing thing, not a "system of thought."

Bracha & Cook discuss some of the shortcomings of CLOS, as understood by them based on a paper by Linda Demichiel and myself published at ECOOP [17] and a user manual written by a technical writer at Symbolics [18]. Linda and I wrote the CLOS specification [19] which was adopted by X3J13, the Common Lisp standardization group.[3] One of the major criticisms of CLOS is that it uses a class precedence list to determine 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 [1]

Specifically, the criticism is that linearization of multiple inheritance chains violates encapsulation by ignoring directly and explicitly specified inheritance relationships. This is a good example of where the language and terminology in the two paradigms are at odds. The example Bracha & Cook are talking about is in Figure 1. Recall the classes used in the example: `person`, which directly inherits from nothing; `graduate`, which is a `person`; `doctor`, which is also a `person`; and `research-doctor`, which is both a `graduate` and a `doctor`.

The directed acyclic graph that these class definitions specify is in Figure 2.

And here is the class precedence list:

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

The complaint is that the definition of `Doctor` specifies that it directly inherits from `Person`, and the class precedence list interposes `Graduate`.

> *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 [1]

But notice something else: the definition of `Person` specifies that it inherits from nothing, but here it inherits from `Standard-Object` and also from the class named `T`. `Standard-Object` provides default behavior for a variety of methods on objects that share the structure and behavior defined by `Standard-Class`. This hints that "direct inheritance" and "direct superclass" mean different things in the two paradigms—or at least are misdirections.

Language from the CLOS specification (not referenced in the Bracha & Cook paper) probably gave rise to this confusion.

> *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.*
>
> –Daniel Bobrow et al, *The Common Lisp Object System Specification*, 1988 [19]

But, read that sentence carefully. It speaks of the class C_2 *explicitly designating* C_1 in C_2's definition—that is, in the `defclass` form that defines C_2. Earlier, the specification states the following using odd wording:

3. Linda and I did the actual writing; the author list is the list of CLOS designers.

A class whose definition refers to other classes for the purpose of inheriting from them is said to be a subclass of each of those classes.

–Daniel Bobrow et al, *The Common Lisp Object System Specification*, 1988 [19]

Again this defines terms that refer to what can be observed in expressions found in CLOS source text, and later the specification describes what the CLOS system does in response to those expressions—they are taken to be instructions on how to alter the running CLOS system.

Each class's class precedence list (and not text found in `defclass` expressions) is the basis for inheritance in CLOS—to the extent that inheritance exists in CLOS.

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.

–Daniel Bobrow et al, The Common Lisp Object System Specification, 1988 [19]

When talking about the source-code text used to modify systems, some terms refer to the syntactic / textual (defining) forms ("direct superclass") while others refer to the reality in the running system ("class precedence list"). This is a characteristic of programming *systems*. When talking about text that conveys meaning as in a programming *language*, most if not all terms refer to relationships between textual items.

This is one of the confusions I found while reading the Bracha & Cook paper. It is an example of incommensurability in which those authors were befuddled by the technical details of an earlier paradigm.

As we recall (see quote *Constraints on Component Flavors*), Flavors shares the same stance regarding the flavors mentioned in `defflavor` forms. Notice that these flavors are called "components," and the defining form is specifying constraints not direct relationships.

Here is the crux: in a programming system, features in the language that are used to define the system and changes to it typically require the system to do something in order to deliver the intention expressed in the language, while in a programming language, the program text printed out on paper is enough[4]—is all there is and is all that need be.

Moon is careful to point out that in well-organized programs each flavor defines a single facet of behavior (see quote *Flavors Well-Organized Programs*). And rather than violating encapsulation, the purpose of linearization is to preserve modularity.

Encapsulation is about information hiding, and the claim Bracha & Cook make is that the information about direct superclasses is modified by an outside process. But in CLOS and Flavors, this "private information" is just a set of constraints, to be strictly observed. If they cannot be, an error is signaled—both in Flavors and in CLOS.

Moon admits that a programming system cannot guarantee program modularity (see quote *Flavors System Guarantees*), and in fact, programming languages cannot either. Parnas himself observed this in 2002 when he observed a failure in being able to achieve perfect modularity in the very piece of code he used in his famous 1972 paper to exemplify his concept of information hiding and its principles.

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 [23]

Flavors defines two ways to support modularity: *Inheritance* with which similar behavior can be specialized, along with *mixins* with which object types can be built from a toolkit of (essentially orthogonal) component parts.

As Cannon points out, the purpose of method combination is to support modularity, and the purpose of linearization is to make method combination *practical*—an engineer-

4. Along with a formal semantics for the underlying language.

ing concern:

> *In essence, the lattice structure is flattened into a linear one. This is important [because] 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*, MIT, 1979 [11]

And a question remains: are Bracha & Cook using the same notion of inheritance as CLOS? In Smalltalk, inheritance is straightforward: if a message is not handled directly by the class of its receiver, the message is passed to the superclass of that class, and so on. In CLOS it's not quite as simple, though the concept CLOS uses is strongly related to simple inheritance. The language used in the CLOS specification speaks of "applicability."

> *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.*
>
> –Daniel Bobrow et al, *The Common Lisp Object System Specification*, 1988 [19]

The Smalltalk notion of explicitly invoking a superclass using message passing and the Beta notion of invoking a subpattern using a call to `inner` (which smells like message passing) form the basis of Bracha & Cook mixins. Hence the paradigm disconnect with Flavors and CLOS. See Apendix: **Fine Print**.

So far we've seen some examples of where Bracha & Cook's understanding of Flavors and CLOS seem to be slightly off base, but this can be attributed to a lack of fluency or perhaps the slant they are after in their paper. Kuhn's notion of incommensurability, though, goes far beyond that. I believe the Bracha & Cook paper rises to the level of demonstrating true incommensurability. Let's explore.

> *The past is a foreign country.*
> *They do things differently there.*
>
> –L. P. Hartley, *The Go-Between*, 1953 [24]

Incommensurability is an extreme symptom of one scientific paradigm being different from another. The word refers to things that do not share a common standard of measurement, and its origin stems from such facts as that $\sqrt{2}$ is irrational (not a rational number), and so there is no way to measure it using only rationals—all one can do is approximate it.

Kuhn's own testimony of how he stumbled onto the idea of incommensurability is similar to mine. Parts of the Bracha & Cook paper seemed confused or mistaken, and some passages were pure nonsense.

Kuhn Discovers Incommensurability

> *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* [25]

Kuhn worked through his perceptions and discovered that earlier nonsensical passages made sense once he understood the older terms and their narrative, but without such work at understanding, given two paradigms (an original and its replacement), statements in one can look like craziness to adherents of the other.

> *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 [16]

The difficulty is the narrative. Kuhn realized that it wasn't simply a matter of defining technical words from one paradigm into terms familiar in another—a good deal of the entire theory surrounding the technical terms needs to be explained in order to understand how the terms interact and play out. Recall the description of the phlogiston theory of combustion I gave before. I couldn't have simply defined *phlogiston* in isolation. If I had said phlogiston was…

> *…a substance contained in all combustible bodies, released during combustion…*

…you would think me mad because you would be interpreting this statement in the paradigm of thermodynamics. Instead, I wove a story about how this "substance" did its thing, and thereby you came to understand—I hope—that it was a sensible theory that happened to be wrong or perhaps not as accurate as the caloric theory and, later, thermodynamics, but when it was in force, it was used to do useful and accurate computations about physical phenomena.

Sometimes the world after a paradigm shift is radically different. Scientists in one paradigm are responding to a

different world from scientists in the other precisely because their stories are different.

> *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 [16]

> *We may want to say that after a revolution scientists are responding to a different world.*
>
> –Thomas Kuhn, *The Structure of Scientific Revolutions* [8]

When Copernicus proclaimed that the earth moved around the Sun, scientists in the prevalent paradigm believed him mad because the narrative surrounding the thing called "earth" was that it had a fixed position—when you stood on the earth you were standing on solid, unmoving ground.

Let's return to the Bracha & Cook paper. I was a little confused reading the following sentence…

> *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 [1]

…because my understanding was that "mixin" as a concept inherited from Flavors referred primarily to classes that defined only auxiliary methods. Therefore, this sentence says that the authors of a paper reflecting on CLOS mixins have chosen to ignore…mixins!:

> *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 [1]

And though Howard Cannon writes specifically of object-oriented programming being—from one point of view—a set of coding conventions, mixins are quite recognizable in CLOS code and marked quite clearly.

I felt exasperated and confused. Did these guys just not get it?

Some (possibly interesting) facts. The CLOS spec indeed does not mention mixins by name. Unlike Smalltalk and Beta, CLOS uses generic functions and not message passing—to better integrate with Common Lisp—so there is only a kinship to message-passing-based inheritance to begin with. But CLOS supports mechanisms that make it formally clear when and how mixins are being used despite the concept being informal. (Standard) method combination defines method types which play specific roles; in general, classes with only methods that declare specialized roles are considered mixins—such methods are called "auxiliary methods."

Methods have a clearly visible type in the Lispy sense of the word. In source text the type is explicit—you can see it here…

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

…and as represented in the runtime you can see it in the object inspector at the bottom of the next page.

Perhaps not formally defined but apparent—and formal. Cannon discusses the notion of flavors this way:

> *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 [11]

A base flavor is the foundation; mixins can be added to supply orthogonal behavior; mixins are not instantiated by themselves; and mixins are named specially. In the way of the programming systems paradigm, his characterization is as formal as it can be. The formality Bracha & Cook seek is more suitable to grammatical rules.

And the contribution of their paper is to provide a formal definition of mixins suitable for a programming *language*.

Flavors though focuses on generic functions which aggregate methods into a combined method.

> *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 [10]

Flavors has a few built-in method combination types, and programmers can define their own. Each type, though, has access to all applicable methods for an object, and constructs a combined method that manages how the constituent methods are called and what is done with the values they return.

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 [10]

This approach affords an additional pair of dimensions to modularity. A programmer who is designing well will think only about what a method must do while designing that method, and will think only about how methods interact while designing or specifying a method-combination type to use. These separate concerns are thereby separated.

CLOS adopts this same approach. And the method types are explicit.

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.
–Daniel Bobrow et al, *The Common Lisp Object System Specification*, 1988 [19]

Bracha & Cook relied on Sonya Keene's book [18] to understand CLOS, and though the CLOS specification does not define or even mention mixins, her book does. This is her glossary entry for it.

***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 [18]

Bracha & Cook seem to ignore this and restrict themselves to primary methods, and they note that the best way to determine whether a class is a mixin is by observing whether it has a (primary) method with a free use of `call-next-method`.

In CLOS…locally unbound uses of `call-next-method` *are a clear indication that a class is a mixin….*
–Bracha & Cook, *Mixin-Based Inheritance*, 1990 [1]

But the real way to tell whether a class is a mixin in CLOS is by observing whether it has no primary methods and only `:before` and `:after` methods.

By the way, `call-next-method` is specified to be illegal in `:before` and `:after` methods, thereby rendering Bracha & Cook's "clear indication" that a class is a mixin a tad murky, and raising the possibility that they really wanted to talk about their ideas and not those from the past.

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.*
–Daniel Bobrow et al, *The Common Lisp Object System Specification*, 1988 [19]

A number of scientists and philosophers have concluded that incommensurability is nonsense. The central reason for this conclusion is the belief that reality is real, the truth is the truth, and scientists are moving toward perfect understanding slowly but surely. In this they are claiming implicitly that unlike biological evolution, the evolution of scientific theories has a fitness function directing it toward a definite goal: the truth that underlies the real universe.

Part and parcel of their objections is that the idea of incommensurability implies there can be no notion of "better" or "improvement," or even "truth," because the inability to

```
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>)
```

comprehend one paradigm from the viewpoint of another means that there is neither a way to compare them nor a solid notion of what truth is. And therefore Kuhn's view implies strict relativism. Errol Morris, the director of "The Fog of War" who was once a student of Kuhn's wrote the following:

> *If everything is incommensurable, then everything is seen through the lens of the present, the lens of now.… There is no history. There is no truth, just truth for the moment, contingent truth, relative truth. And who is to say which version of the truth is better than any other, if we can't look beyond the paradigm in which we find ourselves.*
>
> –Errol Morris, *The Ashtray*, NYT, 2011 [16]

This is the path to *antirealism*—the idea that science can never figure out what the world is like because we live in a sort-of manufactured conceptual or representational world that dictates reality to us. The physicist turned philosopher—a common transition, it seems—Andrew Pickering proposes a way out of this that retains incommensurability as we've seen it manifest in Bracha & Cook but avoids antirealism [26].

He observes that the practice of science is one in which scientists attempt to find a stable point between a *conceptual framework* and the *material agency* of the world as revealed through *machines*—or instruments. Scientists dream up and build machines to explore reality. What the machine does or reports is interpreted according to a conceptual framework. When the machine reports something unexpected or contradictory (a *resistance* in Pickering's terminology), the conceptual framework is adjusted along with the machine (usually) (a pair of *accommodations*), and more observations are made. Pickering calls this (and a bit more) the *mangle*[5] *of practice*. Eventually, the conceptual framework, the machine, and the observations settle down, and—if the scientist is lucky—a fact is manufactured.

An example he uses is the story of two physicists trying to determine whether free quarks[6] exist. William Fairbank and Giacomo Morpurgo took similar approaches: each constructed a machine that would perform a variant on Millikan's oil drop experiment—the experiment that originally measured the fundamental electrical charge—but on much smaller materials. The experiment works by measuring how much a chunk of material is displaced by an electrical field while suspended by a magnetic levitation field [27][7]. By doing this they hoped to find globs of material with non-integral electrical charge thus showing free quarks exist (in those globs).

Morpurgo used graphite for the material and built a machine that operated at room temperature. In an early experiment he found a grain of graphite that demonstrated contradictory behavior in his machine—essentially doing one thing before lunch, and the opposite after lunch. This was his first experience with resistance in this experiment. He came up with an explanation for the anomaly by manipulating the conceptual model that informed how to configure the machine and interpret results—this was his first accommodation. In the end, he was never able to find evidence for free quarks.

William Fairbank was expert in superconductors, and so his machine operated at very cold temperatures and used superconducting niobium balls. Fairbank and his students found evidence of free quarks.

Pickering says this a clear-cut example of incommensurability even though there are no linguistic comprehension issues. He wrote:

> *…I remember that the punch-line…was incommensurability. In the quark study, for example, it appeared to me that the two central protagonists, William Fairbank and Giacomo Morpurgo, lived in different worlds. Those two physicists had found ways to exhibit, respectively, the existence or non-existence of free quarks; what counted as evidence for one was something that needed to be explained away for the other. And yet, this divergence did not quite fit the Kuhnian mould. Kuhn's basic idea was that incommensurability arises from differences in paradigms, which set people up to perceive the world and pay attention to it differently. I could not see any split between Fairbank and Morpurgo in that sense. The relevant difference was rather that they had arrived at different material set-ups, and that Fairbank's apparatus really did provide evidence for free quarks while Morpurgo's apparatus really did provide evidence against their existence. It was as simple as that.*
>
> –Andrew Pickering, *Reading the* Structure, 2001 [28]

This seems a lot like our situation with the Bracha & Cook paper versus the earlier mixin papers. I've already said that the difference between Bracha & Cook and Cannon[8] is embodied in programming languages versus programming systems—these terms each package both the machines or material set-ups as well as the conceptual frameworks that go with them.

A programming *system* consists of an executing software system, tools for examining and altering that system (typically but not necessarily executing as part of that same system), and a mechanism for people to express changes to that system.

5. He means "mangle" mostly in the British sense of a wringer on an old-fashioned washing machine, but also as in *mutilation* and *disfigurement*.

6. A quark is an elementary particle with fractional charge. Normally a number of quarks combine to make up a hadron of integral charge. A free quark is a single quark not part of a hadron.

7. Read especially Feynman's comment in the section labeled "Millikan's experiment as an example of psychological effects in scientific methodology."

8. Really it's Cannon, Moon, and the CLOS designers, but Howard Cannon originated the ideas, so I'll abbreviate the list to "Cannon."

That mechanism of expression typically looks like program source text. The machine or material set-up is the executing system, its tools, and program source text; and the conceptual framework is how the program source text describes or specifies acts of examination or mutation.

A programming *language* consists of a set of program source texts, an empty & idle computer, and a semantics that states what computation the computer would perform when executing the semantics specified by a syntactically legal program source text. The machine or material set-up is the computer and the program source text; and the conceptual framework is the semantics.

Just like the two quark experiments, these two material set-ups and conceptual frameworks are similar, and using them similarly leads to different conclusions. For one thing, programming systems exhibit behavior which can be observed or modified while programming languages are for specifying computations. Roughly speaking.

Bracha & Cook and Cannon are perfectly capable of understanding everything about each others' material set-ups and conceptual frameworks, but they don't want to, because they are in the mangle of their own practices—to use Pickering's terminology.

Our context is different from the one Kuhn talks about. First, the study of programming has little pure nature, little external reality to guide us—and mostly it's a reality we've created. Second, people can choose whether to view programming through the lens of programming systems or through the lens of programming languages. Each paradigm represents a community of practice, which takes study and endorsement to master and then enter. And the choice of how to extend reality is determined by the encompassing paradigm.

Third, I believe there is no question that Bracha & Cook were trying to invent a new concept—*mixin-based inheritance*—in the programming language paradigm, taking their approximate cues from what they gathered from their readings in the programming system paradigm. They plumbed that paradigm to just the depth they needed.

Fourth, my intuition that programming systems versus programming languages represents a micro-paradigm shift might just be a difference of material set-up or more precisely a difference between the kinds of "machines" the two camps use to observe and manipulate the realm of programming in order to manufacture scientific facts about it.

Fifth, you likely noticed that I interpreted the hell out of those old Flavors and CLOS materials for you; I did this to make their narratives clear so that it was apparent how far off the Bracha & Cook paper was. But my *close reading* [29] also demonstrates how hard it is for researchers in one paradigm to get to the heart of the work in another, particularly a paradigm from another era.

And sixth, computer scientists and practitioners benefited tremendously from the (temporary) shift from programming systems to programming languages, a move that focused the research.

Perhaps incommensurability then is the potential for approximate understanding between two paradigms, and effort expended determines how close is the approximation.

One thing that amazes me is that we could have noticed the paradigm shift from programming systems to programming languages right when it happened—using incommensurability as the theoretical basis and "this looks like nonsense" as the instrument. Gilad Bracha told me as much in Brazil that spring evening.

Not many of us stop to reflect. We're just programmers, after all. Or software developers or engineers. And as ants and termites do, we are content to follow where others have gone, and most of us go where most *have* gone. But when we *do* reflect, perhaps what we find is important—and surprising.

Scientists (in other fields) and some philosophers have rejected the idea of incommensurability because there is a real world out there that is directing our inquiries into it. Reality co-designs science. But in our field incommensurability seems real. Our relationship to reality is not quite as fixed as in the natural sciences—our engineers make the nature we study. But are we, anyway, so different?

References

[1] Gilad Bracha & William Cook, *Mixin-based Inheritance*. Proceedings of the Fifth ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. 1990.

[2] Philip Kitcher, *Seeing is Unbelieving*. New York Times Book Review. March 25, 2012.

[3] Chuck Lorre & Bill Prady, "The Jerusalem Duality," *The Big Bang Theory*. CBS. Aired April 14, 2008.

[4] http://wordnetweb.princeton.edu/perl/webwn. 2012.

[5] http://en.wikipedia.org/wiki/Engineering. 2012.

[6] Matthew B. Crawford, *Shop Class as Soulcraft: An Inquiry Into the Value of Work*. Penguin. New York. 2009.

[7] James Bryan Conant, ed. *The Overthrow of Phlogiston Theory: The Chemical Revolution of 1775–1789.* Cambridge: Harvard University Press. 1950.

[8] Thomas Kuhn, *The Structure of Scientific Revolutions*. University Of Chicago Press. 1996.

[9] http://en.wikipedia.org/wiki/Mixin. 2012.

[10] David A. Moon, *Object-Oriented Programming with Flavors*. The First ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. 1986.

[11] Howard Cannon, *Flavors: A Non-Hierarchical Approach to Object-Oriented Programming*. Unpublished. MIT AI Lab. 1979.

[12] Warren Teitelman, *Pilot: A Step Toward Man-Computer Symbiosis*. MIT PhD Dissertation. 1966.

[13] Arthur Rimbaud, *The Drunken Boat*. (Translated by Rebecca Seiferle). 1871.

[14] http://www.faculty.umd.edu/FacultyAppointment/titles/POP.htm. 2012.

[15] Paul Feyerabend, *Against Method: Outline of an Anarchistic Theory of Knowledge*, 3rd edition. Verso. 1993.

[16] Errol Morris, *The Ashtray*. The New York Times. March 6, 2011. http://opinionator.blogs.nytimes.com/2011/03/06/the-ashtray-the-ultimatum-part-1/. 2011.

[17] Linda DeMichiel & Richard P. Gabriel, *The Common Lisp Object System: An Overview*. ECOOP. 1987.

[18] Sonya E. Keene, *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley. 1989.

[19] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, David A. Moon, *The Common Lisp Object System Specification*. Technical Document 88-002R of X3J13. LASC Volume 1, Numbers 3–4. SIGPLAN Notices. June 1988.

[20] http://en.wikipedia.org/. 2012.

[21] Guy L. Steele Jr. & Richard P. Gabriel, *The Evolution of Lisp*, quote from Drew McDermott. ACM Conference on the History of Programming Languages, II. ACM SIGPLAN Notices, Volume 28, Number 3. March 1993.

[22] Alan Perlis, from the Foreword to Harold Abelson & Gerald Jay Sussman, *Structure and Interpretation of Computer Programs*, 2nd Edition. MIT Press. 1996.

[23] David L. Parnas, "The Secret History of Information Hiding." *Software Pioneers: Contributions to Software Engineering*. Manfred Broy & Ernst Denert, eds, Springer-Verlag. 2002.

[24] L. P. Hartley, *The Go-Between*. 1953.

[25] Thomas Kuhn, *The Road Since Structure*. University of Chicago Press. 2000.

[26] Andrew Pickering, *The Mangle of Practice: Time, Agency, and Science*. The University of Chicago Press. 1995.

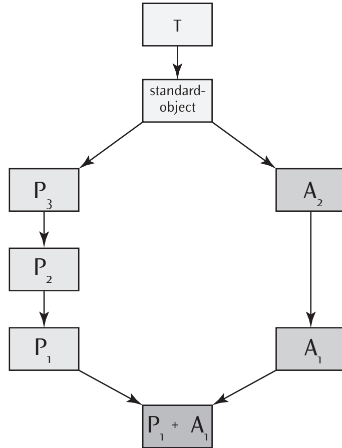
[27] http://en.wikipedia.org/wiki/Oil_drop_experiment. 2012

[28] Andrew Pickering, "Reading the *Structure*." *Perspectives on Science*, Volume 9, Number 4. 2001.

[29] http://en.wikipedia.org/wiki/Close_reading. 2012.

[30] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow, *CLOS: Integrating Object-Oriented and Functional Programming*. Communications of the ACM, Volume 34, Issue 9. September 1991.

Fine Print



In "well-organized" programs, as David Moon would say, a class that inherits from two others would have a class hierarchy that looks like the one to the right. Here, P_1 supplies the primary methods, and A_1 supplies the auxiliary methods. The join would be near the top of the hierarchy, typically at `standard-object`, which is part of the CLOS infrastructure.

This structure is what Cannon means by "*essentially orthogonal,*" and what Moon means by "*each component flavor is a module that defines a single facet of behavior.*" Hence linearization does not violate encapsulation because the interspersed classes are orthogonal. This is not guaranteed, because, as Moon says, "*no programming system can guarantee program modularity or eliminate the need for careful design of a program's structure.*"

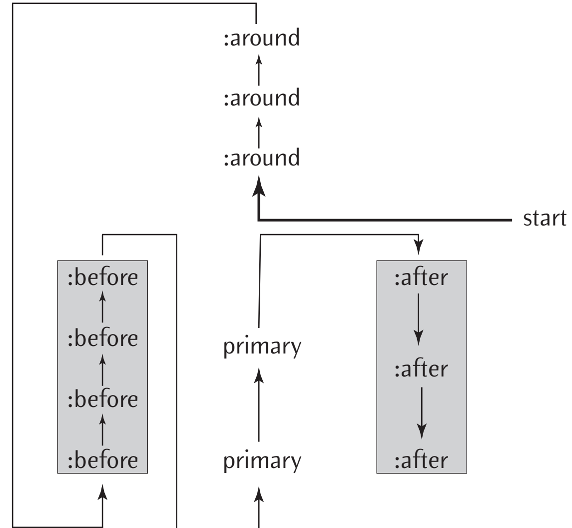
In CLOS, standard method combination recognizes primary, `:around`, `:before`, and `:after` methods, as shown at the top of the right-hand column. Flavors, as discussed in the Moon Paper [10], did not have `:around` methods. The diagram shows the order combined methods are invoked: the most specific `:around` method is invoked first; `call-next-method` invokes the next most specific `:around` method; if there are no `:around` methods or `call-next-method` finds no other, all the `:before` methods are invoked from most specific to least specific; then the most specific primary method is called; `call-next-method` invokes the next most specific primary method; finally, all the `:after` methods are invoked from least specific to most specific. In the diagram, methods with a grey background are invoked automatically without needing `call-next-method` (and in fact, it is illegal for `:before` and `:after` methods to mention it). It's called "standard" because it's the most commonly used one; there are other built-in method combination types (`+`, `and`, `append`, `list`, `max`, `min`, `nconc`, `or`, `progn`), and programmers can define others.

Bracha & Cook consider only primary methods, which means they consider only *procedural method combination* in Flavors and CLOS. They examined Smalltalk's use of `super` and Beta's use of `inner`, which are how those languages combine methods. In Smalltalk, the most specific class of an object responds to a method invocation, and if that method needs to use the method in its superclass that it overrides, it mentions `super`, which refers that superclass. In Beta, subpatterns (essentially subclasses) are not permitted to override methods defined in its superpatterns, so the superpattern that declares the method virtual controls execution, and if it needs to use the method's enhancement in a subpattern, it mentions `inner`, which refers to the current pattern's relevant subpattern.

Bracha & Cook's concept for mixins was to generalize `super` and `inner`; they noticed that Flavors and CLOS also support a version of procedural method combination when primary methods mention `call-next-method`, and this is part of what Cannon originally called *mixins*. But Cannon and later Moon, and, even later, CLOS considered mixins and method combination to be about the auxiliary methods—`:around`, `:before`, and `:after` methods—that is, about declarative method combination, which is an abstraction of procedural method combination.

Bracha & Cook mixins express inheritance by combining classes in a linear order. The rightmost class controls method execution, and a mention of `super` invokes methods in the class "to the left." Combined classes can model Smalltalk inheritance or Beta enhancement depending on whether the rightmore classes are taken to be more specific or more general—this is sort of a coding convention. This also covers CLOS's use of `call-next-method`, so the evocative name, "mixin," was used.

In 1991, Jon L. White, Daniel G. Bobrow, and I published a look at the design landscape CLOS fits into, and we talked about procedural versus de-



clarative method combination. Here is what we wrote:

> *In CLOS, a method can be composed from sub-pieces that are designated as playing different roles in the operation through a technique known as* declarative method combination. *The enhancement techniques of Beta and Smalltalk are simple cases of procedural method combination: Explicit calls are made to related behavior. Another technique for procedural method combination is for a basic class to have a method that calls other explicitly named methods, which will be defined in derived classes. For example, consider a simple stream-opening protocol, shown* <below>. *Notice that the method for* `open` *defined on* `base-stream` *provides a template for the operations on streams. The auxiliary methods* `pre-open` *and* `post-open` *have default definitions on* `base-stream`*, the base class, which do nothing.*
>
> Declarative method combination *is an abstraction based on this sort of procedural method combination. In this example code there are four methods—*`open`*,* `pre-open`*,* `basic-open`*, and* `post-open`*. The main sub-operation,* `basic-open`*, cannot be named* `open`*, since that name refers to the whole combined operation. The other two names—*`pre-open` *and* `post-open`*—are placeholders for actions before and after the main one, i.e., those preparatory steps taken before the main part can be executed, and those subsequent clean-up actions performed afterwards. There really is just one action—opening—and all other actions are auxiliary to it: They play particular roles. This constellation of actions should have just one name, and the auxiliary names need only distinguish their roles.*
>
> *In declarative method combination, the role markers act like an orthogonal naming dimension.…*
>
> –Gabriel et al, *CLOS: Integrating Object-Oriented and Functional Programming* [30]

This discussion is in the context of talking about partitioned operations, which are defined incrementally.

```
(defclass base-stream ...)

(defmethod open ((s base-stream))
 (pre-open s)
 (basic-open s)
 (post-open s))

(defmethod pre-open ((s base-stream)) nil)
(defmethod basic-open ((s base-stream)) (os-open ...))
(defmethod post-open ((s base-stream)) nil)

(defclass abstract-buffer ...)

(defmethod pre-open ((x abstract-buffer))
  (unless (has-buffer-p x) (install-new-buffer x)))

(defmethod post-open ((x abstract-buffer))
  (fill-buffer (buffer x) x))

(defclass buffered-stream (base-stream abstract-buffer) ...)
```