Book Review

Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow, *The Art of the Metaobject Protocol**

Richard P. Gabriel

Lucid Inc., 707 Laurel Street, Menlo Park, CA 94025, USA

Revised September 1992

MIT AI Lab playroom, 1973: We sit on the floor, legs crossed. Some of us are in the well along the back wall; someone is curled in the quarter-circle cutout next to the well. Beanbags form pillows. Then we here his voice in the hall, talking loud, fast. His short legs work fast, an Olympic-style—and -speed—shuffle.

He strides into the playroom, stops just at the overhead projector and says curtly, "Hi." To those in the room he has a smile as wide as his head; to ordinary people he wears a slight smirk. His shirt pocket bulges. And he swivels from side to side, checking the audience. Behind his wire rims his eyes are mirth-squinted. He reaches into his shirt pocket, fishes around, looks down pushing aside the glasses case, and pulls out a pen. Which he holds up for view in the tips of his left-hand fingers while he pulls the pen tip with his right-hand fingers, producing a pointer.

He continues, "I'm here to tell you all about Conniver, and why it is the completely and only winning alternative to microPlanner, which, as you know, is virtually impossible to program in."

He spins rapidly to just catch each person, "and rightly so, I might add."

He turns to the screen and walks towards it, but stops and twirls around on

his heels, "and it doesn't even live up to its own claims, as we'll see in a minute."

Gerry Sussman can only speak fast. And for the next two hours he explains with glee and childish delight the ideas in Conniver. He explains the failings of microPlanner. And he explains the way that Conniver can be used for writing problem-solving systems, parsers, language generators, and world modelers. We all sit entranced as we see the language ideas unfold and bloom from his descriptions and examples. We can imagine our own Conniver programs as he speaks of his own. We can feel the power of the new programming paradigm to liberate our thoughts about AI, and about thinking.

Sussman captivates us by his clear passion for the language he developed and the ideas behind them. We know he is smart and works the same problems we do, but he is able to abstract his concrete thoughts on programming strategy and problem-solving strategy to produce a language for us. And he is able to implement the language for us.

But, how many days like this have there been since the late 1970s? How many times since then have you sat on the floor, looking up at the speaker in a dream-like, child-like trance, forgetting to swallow, as new programming language ideas flowed into your head, spoken by a pied piper like Gerry? When was the last time you felt like a kid again when entertainment and learning collided in an adult-sized playroom?

AI used to be about programming. And it was no wonder that, during those years, some of the best and most memorable ideas in programming languages came from AI researchers and AI labs. It seems obvious that the brain or mind could be thought of as having been programmed—by evolution, by design, by learning, by experience—just as we program computers, and so it also seemed obvious that the important step was to find the language in which to write our computer-hosted mind-programs.

Think of some of the people in AI who are known for their language work either in addition to or instead of their pure AI work: John McCarthy (Lisp), Guy L. Steele Jr (Scheme), Gerry Sussman (microPlanner, Conniver, Scheme), Carl Hewitt (Actors, Planner, Plasma, microPlanner), Terry Winograd (microPlanner, Programmar, KRL), and Danny Bobrow (Lisp, KRL, LOOPS, CLOS).

But no more. There has been little or no significant programming language development for AI use since the late 1970s. And I just don't exactly know why—but I have some hypotheses. Today we see AI researchers working with logics, with pure theories, with microscopic advances in small—but, I suppose, important—technical areas. There just isn't any need for sophisticated programming languages any more, because there isn't a need for sophisticated programs. Or perhaps there is no desire for them outside the commercial realm.

Whatever the cause for the missing interest in new AI programming language concepts, the lack of interest in programming languages has spread, and

there have not been any—or at least not very many—new programming language concepts, new approaches, or new conceptual frameworks of any sort since 1980. Nor has there been the captivating presentation, the living driving force that nails our attention.

Until this book: *The Art of the Metaobject Protocol*.

1. History of CLOS and MOP

The metaobject protocol—what a dull name for such an intriguing concept— the metaobject protocol is a way to produce a specialized object-oriented language dialect by modifying an existing one, in a principled manner.

Let's look at the context in which the metaobject protocol arose, and thereby understand what it is all about.

Object-oriented programming has been on the scene since the mid-1960s, and Lisp and Lisp-like languages have been influenced by object-oriented languages—Smalltalk and Simula, primarily—for quite a while. Actors was one of the first object-oriented Lisp dialects influenced by earlier object-oriented languages, and later came LOOPS and Flavors. Though object-oriented programming was developed outside the world of AI, Lisp and AI-influenced researchers added some of the most interesting recent concepts to object-oriented programming: multiple inheritance, method combination, metaclasses, and now the metaobject protocol.

In the early 1980s, AI became a business, perhaps to its detriment. And with this came pressure for standardization, a process that is still underway. Unthinkable in the 1970s, Lisp standardization began with an informal agreement among a group of developers working on similar Maclisp-derived dialects—the agreement is called Common Lisp—and standardization has now progressed to ANSI and ISO standardization for various Lisp dialects.

Along the way the need arose to incorporate an object-oriented extension into Common Lisp. Several dialects waited in the wings—New Flavors (Symbolics), CommonLoops (Xerox PARC), Object Lisp (Gary Drescher, MIT), and Common Objects (Alan Snyder, HP). After some initial sparring, Object Lisp and Common Objects fell aside, and a group was formed whose charge was to blend the best from New Flavors and CommonLoops. This group worked for about two years and produced the Common Lisp Object System (CLOS), which was adopted into what would become ANSI Common Lisp.

The CLOS group which formed in 1987 worked well together and was partitioned into three main subgroups: the Symbolics technical group, the Xerox technical group, and the Lucid writing group. The Symbolics group brought the background understanding of multiple inheritance and method combination. The Xerox group brought the background understanding of the generic function model and an appreciation for—and the deepest understanding of—the strengths of first-class metaclasses and the metaobject protocol

direction. The Lucid group brought object-oriented naïveté and writing skills. The way the group worked was for the technical groups to argue out the details and for the writing group to produce drafts. Usually if the technical details were not well-worked-out, the writing group would fail or produce a convoluted draft. Also, the technical groups had to explain their ideas in significant depth for the separate writing group to get it.

The result, I think, is a specification that is of unrivaled quality, precision, and unambiguousness. In a fit of pride, the group even today holds the rights to the specification.

The Xerox group also produced a reference implementation of CLOS, called Portable CommonLoops (PCL), which started out as an implementation of CommonLoops—the Xerox root for CLOS—and mutated into CLOS. Written in PCL (and later CLOS), the PCL code looked like the model for the metaobject protocol—just describe the obvious points of extension and customization in PCL and you have it. You can visualize this by imagining that PCL is the mountainous surface of the ocean floor, and the obvious points of extension and customization are the islands that stick up above the surface of the water.

But the Xerox group broke up the tripartite team by expressing a strong desire to write Chapter 3 themselves. The result was a metaobject splinter group. And because the original group had the model of Chapters 1 and 2 for comparison, the initial efforts of the metaobject splinter group were not taken seriously, and the entire effort stopped at two chapters.

In retrospect, this was good. It took several years for the Xerox group to get the metaobject protocol right, and partly this was because at the outset they were unable to see that describing the islands of PCL was too limiting on real implementations. It took seeing alternative implementations to refine where the islands should be and how they should be shaped.

And partly it took them a while to get right the means of describing the metaobject protocol. Enlisting the aid of Jim des Rivières—speaker to and explainer of Brian "3-Lisp" Smith—certainly helped.

2. The layers of CLOS

CLOS is partitioned into three layers: the programmer interface, the programmatic interface, and the metaobject protocol. But only the first two of these are formally specified in CLOS.

The programmer interface generally contains macros and functions like `defclass`, `defmethod`, `slot-value`, and `make-instance`, which are used in most programs and by most programmers. These macros and functions define classes, methods, generic functions, method combinations, access slots, and make instances. These are for programmers who wish to program in vanilla CLOS.

The programmatic interface contains functions that the programmer interface uses. These include functions like `ensure-generic-function` which are used by programmers who wish to provide a different programmer interface or who wish to use a simple variant on standard behavior. These functions also define classes, methods, generic functions, method combinations, and they make and initialize instances.

Given the small size and relative simplicity of CLOS, two years was a long time to spend on it, which reflects both the degree of difference in point of view brought by the New Flavors and CommonLoops groups and the difficulty of writing a specification at the right level of detail. There was little hope of agreeing on the most controversial part of the breakdown: the metaobject protocol.

The metaobject protocol is used by programmers who wish to produce a variant of CLOS. CLOS itself can be seen as implemented by an object-oriented program which is itself in CLOS. As with any object-oriented program, the strength of a CLOS-in-CLOS is that it can be customized and specialized—by subclassing, adding methods, shadowing methods, etc.

Reflect now for a minute on what this can mean: The implementation of the language is opened up, so you can make a variation, perhaps an arbitrary variation; you can tweak the implementation for different performance profiles; you can study its architecture and improve it.

But hold on, you really cannot do all of that. First, if the CLOS you are using to run the CLOS that implements CLOS uses the CLOS you are running, you cannot change that CLOS too much, or maybe it will stop working, or maybe you will stop understanding what it is doing and will be unable to continue. So, you really cannot wander too far afield in your modifications. And, in fact, part of the reason the metaobject protocol was not standardized—not even completed—by the CLOS group was the difficulty in understanding just what could be allowed and what must be restricted to maintain a consistent base.

Another part of the reason it wasn't standardized was that a real CLOS implementation must run fast, which means that the implementors must be able to depend on certain things in the base document. For example—though this example isn't part of the metaobject protocol—`slot-value` is a function that calls `slot-value-using-class`, which can be defined differently for different classes. But, an implementation of CLOS will almost never call `slot-value-using-class` unless there is a user-defined method on it. Similarly, an implementation may never have a slot accessor call `slot-value` unless there is a method on `slot-value-using-class`. To use this optimization, implementors must be sure that user-defined methods on particular generic functions are the only way to customize slot access.

The designers of a metaobject protocol must be aware of all or most similar tricks in the implementation trade, so that the protocol does not eliminate optimizations or tricks that have significant performance implications.

3. Metaclasses and metaobjects

In *The Art of the Metaobject Protocol*, we are presented with the basic approach to metaobject protocols used by the authors, and then we are given a particular metaobject protocol for CLOS.

CLOS is different from many other object-oriented languages in specifying that every Lisp object be an instance of some class. This includes such mundane things as numbers and lists, but also such interesting things as functions, generic functions, methods, classes, and method combinations.

If each class is an instance of some class, then either there are a lot of classes ("turtles all the way down") or else some classes are instances of themselves or something circular like that. In fact, in CLOS standard-class is a subclass of class, and both class and standard-class are instances of standard-class. Classes whose instances are classes are called *metaclasses*.

A metaclass controls the representation of instances of its instances. A class specifies the names of the slots in its instances and determines the behavior of its instances. This is because you write methods for a class, and methods are applied to instances. The method that runs, then, depends on the classes of its arguments, and behavior is defined as the actions of generic functions (which are made up of all their methods).

The generic function that makes instances is called make-instance, and it is invoked on classes. Therefore, it is defined on metaclasses. Because the action of make-instance is to allocate storage for the instance and set up slots, the metaclass controls representation. Similarly, slot-value calls slot-value-using-class, which is defined on metaclasses, and slot-value must access the representation of instances.

In most other object-oriented languages, C++ for example, there is no real concept of metaclass because the storage layout is dictated by the complier writer, not by the user. In CLOS, even, implementors have optimized layout for standard-class, the default metaclass.

So the use of first-class metaclasses—every language has second-class metaclasses, which is just another way of saying that something controls representation, it's often just not visible—is the first step in the direction of metaobject protocols.

If CLOS is implemented in CLOS, there are objects that control the behavior of CLOS—objects like classes, instances, and generic functions—and these objects are called *metaobjects*. The way that objects behave is called a *protocol*, so there is a protocol for making instances, initializing them, for the way to evoke behavior, and, most importantly, for how to customize those objects and that behavior. So a *metaobject protocol* is the way metaobjects behave and how to customize them.

Let's look some more at the structure of CLOS. The surface layer is generally a set of macros that expand to functions in the glue layer, and those

functions use metaobjects to determine their operation. Metaobjects are not metaclasses, though some metaobjects are metaclasses—some functions and ordinary classes are metaobjects.

The first voyage into metaobject protocols can be taken by using instance creation and access. The surface layer of CLOS here is defclass. It expands into a call to ensure-class, which checks whether there is a class with the same name and, if not, calls make-instance on the appropriate metaclass. Here we see the three layers in action.

With the metaobject protocol one can do things like add persistence to CLOS. Persistence is the property of CLOS instances surviving from Lisp session to Lisp session. This is accomplished by saving them in files, but the idea is to make file residency transparent. With the metaobject protocol, one can create a persistent metaclass that will create objects that can live in files but which are loaded into a Lisp when needed and saved out when a checkpointing operation is invoked. Instances, before they are paged in, are represented by stubs that are of the right class. Accessing the innards (slots) of an instance causes it to be paged in. Recall that slot-value calls slot-value-using-class, which would be customized for the special metaclass.

Other examples of things you can do with a metaobject protocol are dynamic slots—slots that are not actually allocated until they are used because in any instance they are sparse—and slot facets, which store additional information for each slot. Dynamic slots require a new metaclass and methods on slot access and initialization. Facets require a subclass of slot definitions, which are metaobjects.

4. The book's pedagogical approach

The Art of the Metaobject Protocol tries to accomplish three goals: to teach the concept of a metaobject protocol, to teach the CLOS metaobject protocol, and to propose something to be *the* metaobject protocol for CLOS.

The teaching strategy is first to define a subset of CLOS, called Closette, which leaves out some of the more problematic parts of CLOS—class redefinition, eql specializers, and shared slots, for example—and then to present the source of Closette as a Closette program to illustrate how a metaobject protocol works and can be designed.

In the first part of the book, the structure of Closette is gradually revealed in a striptease to the beat of posed questions and proposed extensions. This part works very well, and the clarity of the pedantic part of the book teaches us something about what was lost when AI and programming languages split.

This first part achieves the first two goals, and very well—readers at almost all levels of CLOS sophistication can learn something here, and all the while enjoying the read.

The second part of the book proposes a metaobject protocol for CLOS. This is supposed to be the missing third chapter of the CLOS specification. Writing *The Art of the Metaobject Protocol* including this proposal was a daring move because of the odd group dynamics that led to Chapter 3 being missing from the original specification.

Because of these maturing influences and the passage of time, later drafts of the metaobject protocol were taken more seriously, and the resulting proposed metaobject protocol has been thoroughly and repeatedly reviewed not only by the original CLOS designers but by every CLOS implementation group. And it shows. I would expect it to be adopted by every implementation group as soon as possible—that is, I would expect it if there were a viable Lisp market to create the incentive to take on the workload. Since there isn't, I suspect some will adopt and implement it more or less than others.

Nevertheless, some parts of the metaobject protocol are still controversial—for example, should some of the protocol be aimed at compile-time rather than run-time customizations—and the jury is still out on whether these controversial parts represent an insurmountable.

The book concludes with a listing of a working Closette implementation. The sources can be FTPed from Xerox Parc.

5. AI and programming

But what about the relationship between AI and programming?

Do you remember when AI papers routinely contained the source code for the programs they described? Perhaps those were the naïve years, but they were the years when AI was still about programming, and some AI researchers were far better language designers than, well, AI researchers. But because there was little distinction between the two activities—doing AI research and designing programming languages—both types of person were considered AI researchers.

The programming language guys in every AI research team brought something valuable to their team. The language guys refined the ideas and distilled them into the most coherent possible form. They acted like the naïve writing team for the CLOS specification—they had to get it explained over and over (usually just in their own heads, by themselves), and the result was a clear document that explained the programming ideas embodied in the AI research.

But more than just clarifying some of the ideas in AI research, the programming language people took the best of those ideas and embodied them in programming languages that were used not only by other AI researchers but by other people as well.

Thus, the most widely applicable results of a particular research team were spread by the programming language folks distilling, abstracting, and imple-

menting them. Of special interest in these efforts to distill and disseminate were the efforts made to disentangle the interactions among the features of the new or extended language. Coincidental or muddy interactions in the original research vehicle were clarified and brought to the fore. For example, one of the important contributions of Conniver was to explore and resolve questions about using the right environment for searching and continuing search, and part of the insights gained led to the current designs for lexical Lisps, like Scheme (which was a descendant—in some sense of the word—of Conniver) and Common Lisp (a descendant—in another sense—of Scheme).

Though it seems funny now to think that Scheme lexical environments were ever difficult to understand, you must take my word that in 1976, people with Ph.D.'s in computer science required careful explanation with lots of examples to get it. In fact, many people—myself included—had to write lexical interpreters to understand the interactions that led to the external behavior, and understanding the interactions was required to get it.

Though the beacon provided by the language designers was important for AI, the field of computer science has probably lost more by the drifting of AI away from programming than has AI by the drifting away of programming language research. When you face programming a large AI problem, you face the near impossible; and assembly-language-like languages such as C and C++ just don't provide much assistance. Sort of like getting a sturdy step ladder from Ace to begin the journey to Jupiter: Sure it gets you part way there, but the same part way that a good high jumper could do.

So the language designers in the AI lab set up a distiller containing AI problems and approaches, and the distillate was new ideas and programming concepts. Such as Actors, continuations (from Steele and Sussman trying to understand Actors), multiple inheritance, method combination, multimethods, backtracking, data-driven computation, logic programming, and futures, to name some.

There were benefits for each group: The AI guys got a distillation of their ideas into pure form and dissemination of language-related fraction of their work, and the programming language guys got a source of ideas. When groups split, it is often because there is an imbalance—even a small imbalance with enough rpms can shatter the strongest wheels.

Several possible imbalances come to mind: AI didn't get enough from the programming language ties—limited funding, lack of interest, or just plain irrelevance to AI's changed goals forced the group apart; or programming languages didn't get enough from the artificial intelligence ties—practical AI turning away from Lisp, part of AI turning away from programming, and the rest of AI honing such things as knowledge representations was the wedge.

These hypotheses share a common theme: each group views the other as having run out of inspiration. Which, if either, is the truth? Let's look.

Artificial intelligence has definitely moved its focus over the years in ways

irrelevant to programming languages. In the 1960s and during part of the 1970s, artificial intelligence gained a lot of power directly from the expressive power of the language it used. Examples are backtracking, frame-based representation, and agenda-based, blackboard-based, and resource-based control structures. But now power is derived from a variety of sources: large stores of knowledge in finely and extensively tuned knowledge bases, neural networks, connectionism, genetic algorithms, and programless systems—in all these cases the contributions of programming languages to the power of systems is less than in earlier research.

When programming is still done today in artificial intelligence, its focus and requirements are different from what they were in the past. Today there is somewhat less focus than before on large, single-person-written systems that "do something." Single-person programs demonstrate feasibility only, not usefulness. Large-group-written programs run into problems that programming languages rarely solve—group interactions, multiple-model designs, configuration management, and coordination.

Further, when AI programs are written today with usefulness in mind, sometimes the best approaches do not depend as much on the ability to solve difficult programming tasks as on other factors. For example, in some DARPA-supported natural language projects, success is at least partially measured by the ability of systems to summarize short newspaper articles. The best performing systems use statistical techniques. These systems are simpler than comparably performing ones based on more traditional natural language approaches, and the crux is getting the statistics right, not getting an extraordinarily tough program right.

Similarly, programming language work has moved its focus over the years in ways irrelevant to artificial intelligence. Object-oriented programming—once of interest to AI because of the frame-like qualities of object-oriented languages—has supplanted pure knowledge representation and problem-solving control structures as fertile fields for research. Bringing the benefits of AI language technology—derived from languages like Lisp, Smalltalk, and Prolog—to the real world of programming with its high premium on performance has replaced the challenge of programming the impossible that AI once presented. Programming environment work—once common in the AI world by the programming language contingent—now focuses in traditional languages like, C, C++, and Ada.

Further, programming languages have been developed to a point where they are adequate for the task AI demands of them today: the data abstraction mechanisms, the module systems, control structure, and object-oriented facilities provide enough general mechanism for almost all existing programming tasks, and for the non-programming tasks as well. The problems of the real world—multi-user environments, reuse, performance, and bringing the traditional world of programming into the modern (post-FORTRAN) era—are

more challenging to the programming language researcher; and more important than any in the AI world.

New areas of research unforeseen by the needs of AI have opened up. Such things as user interface design, end-user programming, object exchange and system sharing, application platforms, and multi-lingual programs. A lot of programming language work today is on the theoretical underpinnings of languages.

A less theoretically interesting hypothesis for the split between AI and programming language work is that funding dried up for work with both AI and programming language components.

Nevertheless, in 1980—to pick a round date—the AI folks and programming languages folks parted ways. And with their departure came programming language winter.

6. Using MOP to express interactions

What's complicated about CLOS and any object-oriented programming language is that the interactions are more complex among the language features than in non-object-oriented languages. Consider, for example, class definitions: ordinary data structure definitions don't interact except when they happen to be operated on by a single piece of code; but a class definition is the confluence of all its superclasses. The effective class definition is not something you can look at by local inspection—without a very fancy programming environment—it takes looking at a variety of pieces of code or reading very carefully crafted protocol definitions or documentation. Similarly with method combination, the effective method is a combination of applicable methods, which is a set determined, usually, at run-time, and the effective method cannot be viewed directly.

People generally agree that object-oriented languages provide considerably more power than traditional languages, especially for reuse. This power is a direct result of the high degree of interaction—and hence compression of expression—the language affords among the components of the system being put together. By *compression* I mean the phenomenon of being able to state compactly one's (computational) meaning because the context is rich and supplies information through interaction. For example, one can compactly add locks to a resource—if the resource is an object whose access is controlled by a generic function—by using before and after methods, which nicely partition the grabbing and releasing behavior just because the interactions that the programmer would normally have to express in explicit code are handled by the abstraction of method combination, which implicitly handles the interaction of different components of desired behavior.

As we saw earlier, the way that programming language folks distill and

clarify the concepts in a potential language is by understanding the implicit—
and often muddy—interactions of the language elements. Often this is done by
going through the process of designing, documenting, and implementing the
new language. When Guy Steele and Gerry Sussman were trying to understand
the meaning and interactions of the combination of a functional language
(Lisp) and an object-oriented language (Actor), they wrote a series of interpre-
ters. These interpreters became Scheme, and from them they learned that
function calling and message-passing were fundamentally the same, a lesson
still not well and thoroughly learned today.

With object-oriented languages, the degree of interaction is very much
higher than with traditional languages, and so understanding those interactions
is a lot more difficult. The difficult challenge—and the contribution, if suc-
cessful—of the metaobject protocol is to make these interactions explicit and
first-class by providing a model of CLOS as a CLOS program. To accomplish
this requires two things: to understand the interactions well enough to decide
what should be exposed, and to know the right place along the high-level-
abstraction/low-level-implementation dimension to make the exposure. For
example, you want to set the level high enough that accidental implementation
details are hidden, but low enough that some real customizations and changes
can be made.

Getting all this experience took the Xerox splinter group—the authors of the
book—4 or 5 years. And how very much like the experience of the program-
ming language guys in the AI lab and the naïve writing group it was.

The interesting part about the metaobject protocol is that it provides
reflection to CLOS. Reflection is the ability of a program to manipulate as data
something representing the state of the program during its own execution.
Reflection can be introspective or intercessory, or both. Introspection is the
ability to observe and analyze one's own state; intercession is the ability to
modify one's own execution state or to alter one's own meaning. The meta-
object protocol is both introspective and intercessory.

It is one thing to provide the ability to customize a program, but it is
extraordinary to be able to customize a program whose correct functioning
depends on that program.

It took all these years to see that programming language research is all about
understanding interactions. And isn't AI about interactions even now?