Parenthetically Speaking with Kent M. Pitman

What's in a Name?

Uses and Abuses of Lispy Terminology

Every now and then, when leafing through the trade rags or rummaging the shelves at a nearby computer store, I notice that some concept that I've known from the research community has crossed the magic threshold into commodity producthood and joined the ranks of other ubiquitous, meaningless, must-have product buzz phrases.

Until recently, it seemed to be "Graphical User Interface" (GUI). No modern program would dare be without one, but does it actually mean anything useful to say you have one? I doubt it. From what I've seen, adding a GUI to your application can imply anything from the mere replacement of words in a familiar natural language with icons having all the intuitive appeal of international road signs to what I think of as a *real* graphical interface with displayed drawings and images that permit commands to be issued by point-and-click gestures at semantically interesting points on the display. As programs have improved and terminological requirements have weakened so that now every program can claim to have a GUI, I've been watching for the new trend in marketing hype. And now I think I've found it.

"Object-Oriented." My initial reaction was one of relief. At last something that emphasizes semantic/structural considerations in programs and not just presentational glitz.

And there was some initial excitement, too: Lisp would finally have its chance to compete on its own terms. Well, I *thought* they were Lisp's own terms.

"**Object-oriented.** A characteristic of a graphical user-interface. It means that you command the computer by working with on-screen objects rather than issuing written commands."
—*Consumer Reports*, September 1993

Well, OK, let's be generous. The folks at *Consumer Reports* are only just barely getting started in the computer arena and they obviously have a lot to learn, so perhaps we can forgive them for being so far afield in their definition. But it does show pretty plainly what can happen to marketing buzz phrases once they take on a life of their own, and it points to the need to be watchful about how such terms get used, lest traditional terms for describing Lisp's strengths be redefined out from underneath us and we find ourselves scrambling for new terms to distinguish Lisp from its latter-day pretenders.

Recently, in a discussion with a non-Lisper about object-oriented design, I was outright stunned by one of his questions. "Lisp? Is that an object-oriented language? I didn't know it had encapsulation."

Encapsulation? Since when had the presence of encapsulation become the defining characteristic of an object-oriented system? The basic concept has been around for a respectable while, going at least as far back as work by Barbara Liskov's group at MIT in the late 1970's on the language CLU [Liskov 79].

It's not a bad idea at all. But it is *not* what makes an object-oriented language.

"A module is encapsulated if clients are restricted by the definition of the programming language to access the module only via its defined external interface. ... Objects in most object-oriented programming languages are encapsulated modules whose external interface consists of a set of operations."
—A. Snyder, *Encapsulation and Inheritance in Object-Oriented Programming Languages*

"Lisp *is* object-oriented, but encapsulation has nothing to do with it," I responded in an indignant tone.

"But how can you say you have an object if it doesn't control its own behavior?" he replied.

I pointed to an ashtray on the table. "Not all objects in the world have control over their own behavior. This has no intrinsic behavior of its own. If I tap it with my hand, the resulting effect is as much a property of my hand as it is of the ashtray," I said, plainly edging the discussion toward multi-methods.

Object systems in Lisp have evolved away from the idea of associating all methods with a single class. Lisp code once tended toward a "classical method" style that dispatched off of one argument, as in:

```
(define-class foo () ((a 3)))
(define-class bar () ((a 4)))
(define-method (:frob foo) (bar) (+ (foo-a self) (* 2 (bar-a bar))))
(send (make-instance 'foo) :frob (make-instance 'bar)) => 11
```

In modern Lisp systems, such as Common Lisp, the trend has been to permit the dispatch to be symmetric across as many operations as are necessary:

```
(defclass foo () ((a :accessor foo-a :initform 3)))
(defclass bar () ((a :accessor bar-a :initform 4)))
(defmethod frob ((x foo) (y bar)) (+ (foo-a x) (* 2 (bar-a y))))
(frob (make-instance 'foo) (make-instance 'bar)) => 11
```

"But if an object cannot control its own behavior, how can one make guarantees about the integrity and security of the data it represents?"

I paused for a moment as I contemplated how anyone could in good conscience say that any data was ever secure. Surely that was as much a statement about the operating system as the program itself. A sketch of a multi-method explaining the problem came into my head:

```
(defmethod secure-data-p ((p programming-language) (os operating-system)) ...)
```

Somehow I didn't think this kind of meta-circular attack on the problem was going to prove fruitful. So I opted for making my appeal in English instead. "If I write a program in the world's most secure language and run it on a processor where another process can open my program's memory and read it as ordinary binary data, what does that say about security? Please don't say that to be object-oriented means I have to be secure, because then I can prove that nothing is object-oriented!"

What is Object-Oriented, Really?

Lisp has been object-oriented since its beginnings—when there were no user-defined data types at all, and when there were consequently no distinctions between internal and external interfaces (other than those that programmers contrived for their own use).

I think "object-oriented" is a metaphor, philosophy, or methodology that guides how you think about programs. The original source of the metaphor seems rooted in the concept of object identity. Even the earliest Lisps exhibit this concept in at least two powerful ways: First, symbols are interned, so when you name a symbol you are not just asking for any old object of type symbol with the indicated name, but rather for some very specific symbol. Second, some objects that are not interned can still be modified by side-effect, just as real-world objects can be, leading to important phenomena such as structural sharing (and even circular reference) and informational locality.

To me, the essence of object-oriented programming is captured by the idea that objects are things with an identity that extends uniformly throughout a program. The programming system is not free to reclaim my object and substitute a structurally similar but detectably different object on the mere hope that it will be equally satisfactory. There is a contract between the programming system and me about what the salient aspects of my objects are, and they are preserved throughout those objects' observable lifetime within my program.

Drawing the Battle Lines

While it may be hard to describe object-oriented programming, it is often easy to recognize it in practice. Consider the board game Battleship: Each of two players gets a 10x10 grid representing a patch of ocean where a battle will take place. On that grid, he places markers representing battleships which remain fixed in place but hidden to his opponent during the game. The players take turns calling out coordinates where they have dropped a bomb, asking on each occasion whether they have gotten a "hit" or a "miss."

Consider now a typical FORTRAN implementation of this game. We imagine that for each player this is a 10x10 integer array in which the possible values of the array cells are likely to be 1 for "empty and not yet bombed," 2 for "containing a part of a battleship, and not yet bombed," 3 for "empty and bombed" (*i.e.,* a "miss"), and 4 for "containing a part of a battleship and bombed" (*i.e.*, a "hit"). This is not object-oriented because the objects in the array (mere integers) represent ships, but they are not recognizable out of context as ships, they are not subject to inspection as ships, and they provide no notion of object identity that would distinguish one ship from another. The sum total of these things, in my opinion, might or might not make this particular program more difficult to develop and debug initially (depending on how well-understood the problem situation was at the outset), but certainly makes the program more difficult to maintain, modify, and extend because the chosen representation contains only exactly the information needed for the application task and makes no overt attempt to focus on the identity and nature of the ships in a way that might be later extended or reused.

By contrast, consider a typical implementation of this problem in Lisp. Probably the programmer would attempt to construct some representation of a ship general enough that it could be recognized and used not only in this application, but perhaps others as well. The ships themselves might be constructed of sub-objects to which the grid array might point. It might be possible to determine things about the ships not required by the game—for example, which direction the ships were pointing. These kinds of situations—object identity, inspectable representation, organization with intent to reuse and extend—are characteristic of what that I think of as object-oriented.

Object-Oriented Programming vs Object-Oriented Languages

Object-oriented programming is best characterized as a philosophy, a methodology, a style, or an attitude that emphasizes not just the mere representation of information, but also various aspects of its organization. In general, when issues of object identity (rather than mere structural equality) come into play, you know you're moving into the object-oriented realm.

It is common, but not necessary, for object-oriented programming to be associated with dynamic typing, with the ability to define opaque types (disjoint from the built-in types initially provided by a language or system), and with various interactive (sometimes visual) debugging facilities. It is important that these terms not become synonyms, though, since sometimes it's useful to describe a system as having only one or the other of these properties.

Object-oriented programming is fostered by certain programming languages more than others, but it is neither intrinsically present nor intrinsically absent in any given language. In our Battleship example, I could imagine a clever FORTRAN programmer laboriously creating data structures that were object-oriented in spirit. I could imagine a Lisp programmer choosing a FORTRAN-like data representation even though Lisp provides better ways to do it. But overall, Lisp's facilities tend to lead more naturally to an object-oriented approach, so that's why I call the language object-oriented.

The Trend Continues

The term "incremental compilation" is another one that I've been sad to see recycled in the marketplace. Here again is another key feature of Lisp not duplicated in most of its competitors: The ability to compile and load new code without exiting your running application. Here, too, those who haven't duplicated the functionality have nevertheless figured out how to borrow its name for their own uses.

In Common Lisp, the definition of a compiled function `F` is represented as an object of type `COMPILED-`

FUNCTION that resides in F's function cell. Although in some languages, recompilation is a mysterious process that requires extralingual operations such as relinking and reloading compiled files, the Lisp model is so conceptually simple that recompilation can occur in a way that is perfectly explainable in terms familiar to the programmer. The COMPILE function creates an identifiably distinct object of type COMPILED-FUNCTION, so that the old and new function objects can meaningfully reside together. Then, finally, F's function cell is given a new value—that of the freshly created object—just as any other cell value in Lisp might acquire a new object as its value. Most importantly, this is all done within a running Lisp application with no need to exit, relink, or restart.

But now the term "incremental compilation" is sometimes used more simplistically in some circles—to mean merely that the full set of source files is not compiled. Instead, only a code fragment is compiled and the resulting definition is, for example, appended to an existing binary file with an annotation that says "use this definition instead of one you might have seen earlier." But the important part—at least to me—has been lost: In this scenario, you must exit the running application, run the compiler as a separate job, relink and reload the application, and start a fresh application.

To those who have only had full file compilation before, even this stripped down kind of service may seem like a real step up. But to me it's a step down from what I expect from "incremental compilation" since it forces me to exit my running application. So I think it's an abuse of long-standing terminology, perhaps even in some cases with a deliberate intent to mislead.

There are others abuses as well, but I will leave finding them as an exercise to the reader. Pick up a brochure on some hot new programming system and watch for the familiar terms leap out at you. Then try the system itself to see if it meets your expectations or if they're just playing games with the naming.

Conclusion (and Call to Arms)

The Lisp community has been around for a long time, and I think has every right to continued use of the terms that have traditionally expressed Lisp's merits. But we need to remember to assert that right or we'll find ourselves pushed out of the way by others with their own agenda for our words.

There are new and interesting systems emerging all the time that are aptly described by and worthy of our venerable terms—we should be supportive of such uses. Likewise there is other emerging systems which are interesting but not usefully described by our terms—make them pick their own. And then there are the charlatans and the hype-makers, who just need to be exposed.

Why does it matter? Well, the next time you tell your boss that you can't work without a system that's object-oriented, supports incremental compilation, and so on, maybe it will be more clear. You may find you get just such a system, but that the terms have been so devalued that you haven't adequately expressed your real needs. Or you may find that Lisp wasn't even considered because someone thought it wasn't object-oriented!

So if you see your favorite term getting abused, don't just stand idly by—get in there and ask the hard questions, and make them defend their use. Either you'll learn something, or they will.

Acknowledgments

Gary Roberts helped me to focus in on the problem, and to realize its full scope. Roy Badami, Richard Gabriel, Gregor Kiczales, Jason Trenouth, and JonL White provided useful background information. Judy "yduJ" Anderson, Andy Latto, and Ken Olum reviewed early drafts of this article, providing useful feedback. Ken Olum also let me borrow his Battleship game; examining it, I learned that 1967 was apparently before the time when it first became fashionable to trademark every name in sight—so I probably wouldn't have to say the name of its owner, Milton Bradley®, but since its name appears in this sentence, I do have to tell you that even that long ago Milton Bradley itself was a registered trademark—presumably owned by itself.

References

[CR 93] "The Basic Choice in Computers," *Consumer Reports*, September 1993, p572.

[Liskov 79] B. Liskov, et al, *CLU Reference Manual*, Technical Report 225, MIT Laboratory for Computer Science, Cambridge, MA, October 1979.

[Snyder 86] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages", *OOPSLA '86 Conference Proceedings* (Norman Meyrowitz, editor), available as *SIGPLAN Notices* (special issue), Volume 21, Number 11, November 1986.