The Continuing Quest for Abstraction (ECOOP 20th Anniversary Panel)

Henry Lieberman

Media Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract. The history of Object-Oriented Programming can be interpreted as a continuing quest to capture the notion of *abstraction* – to create computational artifacts that represent the essential nature of a situation, and to ignore irrelevant details. Objects are defined by their essential behavior, not by their physical representation as data. The basic Object-Oriented paradigm of organizing programs as active objects and message passing has now been accepted by the mainstream, for which ECOOP can be justifiably proud. Future developments in the field will focus on capturing computational *ideas* that can't be expressed well simply by functional abstraction. Programming will evolve from textual programming languages to using natural language, graphics, demonstrated actions, and other techniques.

1 The revolution is dead. Long live the revolution!

ECOOP has plenty to be proud of in its 20-year history of promoting Object-Oriented Programming. Object-Oriented Programming is a revolution, that, largely, we've won. At the start of ECOOP, and its sister conference, OOPSLA, in the eighties, OOP was a fringe movement exemplified only by a few research programming languages, and which saw little use outside the research community. Now, OOP is mainstream, and popular programming languages such as Java, C++, C#, Python, etc. have at least some form of the object-oriented concept. We all can be proud of that success.

Of course, like all revolutions, widespread acceptance of the concept didn't quite happen in exactly the way the original revolutionaries (including myself) envisioned. And the ideas didn't get accepted in their pure form, as they originally appeared in Actors and Smalltalk; along the way they were only incompletely understood by the mainstream, diluted, and got mixed with more conventional approaches. You can't win them all. But they did take hold.

Overall, the result was positive. While battles still remain to be won on many fronts, the basic idea of organizing programs as objects, methods, and message passing is accepted widely both in academia and industry. The success of object-oriented programming was achieved because of its ability to facilitate modularity, separation of concerns, reuse, extensibility, transparency, and parallelism in

programming. Object-oriented programming, in its original form, is a done deal. Where do we go from here?

2 The search for abstraction

I think that the history, and the success, of object-oriented programming is**Error! Bookmark not defined.** attributable, in large part, to the ability of objects to facilitate *abstraction*. By abstraction, I mean the ability to disregard inessential aspects of a situation, and to focus on the essential aspects. The key idea of object-oriented programming, in my view, is that objects are defined by their *behavior* (responses to messages) and not by their physical representation, i.e., particular patterns of bits inside a computer. Behavior is essential, bit patterns are not.

Object-oriented programming succeeded by taking an ontological position on the basic "stuff" of which computer programs and their data are made. In the early history of computing, computers were seen as manipulators of bit patterns. Programs are bits. Data are bits. This was fine when programming was done in assembly language. No pretension was made that a programmer was doing anything except manipulating bits. Problem is, people don't think very well about patterns of bits.

High-level programming languages were introduced as an ontological shift in what computers were about. Different languages took different stances on what that shift should be. FORTRAN said that computers were about manipulating numbers and the mathematical formulae that describe relationships between numbers. COBOL said that computers were about manipulating English-like descriptions of databases (actually, maybe not such a bad idea, as we'll see later on in this paper). Simula said that computers were about manipulating simulations. LISP said that computers were about manipulating symbols and lists.

Those last two are significant. Because it isn't very far, once you think about computers as manipulating simulations, or manipulating symbols, to arrive at the point where you realize that computers are really about manipulating *ideas*. Humans think in ideas. Why not computers? That's why Simula and LISP were able to give birth to object-oriented programming in a way that descendents of FORTRAN, COBOL, ALGOL and other languages were not.

It is encouraging to me that, while I have been away from working directly in the field in recent years, much work in the field of Object-Oriented Programming still seems focused on the goal of abstraction and capturing ideas in programming. While I think there was a slowdown in innovation in the field in the nineties, as people's energies were focused on facilitating OOP's entrance to the mainstream, now I think that research in the area is going in some healthy directions. Particularly, work in two areas seems like it has a well-motivated concern with how to capture kinds of abstraction that aren't well served by the original conception of object-oriented programming, that relies purely on functional abstraction to capture ideas.

First, there is the Patterns movement. Patterns are an attempt to capture some high-level design rationale in systems that go beyond simply abstracting them into a function. Because the only way a function (or conventional object) can abstract something is to create a variable for it, systems of objects and patterns of message

passing between objects, such as protocols, were not well captured by previous abstraction mechanisms. These ideas can be easily expressed by people in natural language (a signal that programming languages *should* be able to handle them), but are hard for conventional programming languages to capture. Thus pattern languages represent a true advance. However, work continues in making them more capable and precise, and developing tools to help programmers work with them.

Second, there is the Aspects movement. Aspects were motivated, again, by the limits of functional abstraction. Again, researchers noted that there were significant ideas about how programs should be organized that crossed functional boundaries. Again, also, these ideas were easily expressed by programmers in natural language, but difficult to capture in programming languages. While Patterns try to capture notions that are bigger, in some sense, than a single object, Aspects try to model notions that cut across small pieces of multiple objects, and would otherwise necessitate multiple unsynchronized edits and traces.

Finally, I think a future direction that will be important is to understand the *dynamic* nature of abstraction. By and large, current ideas of abstraction in programming languages are about *static* abstraction; at the time of programming, the programmer decides what abstractions need to be made and how they are expressed. But increasingly, abstraction will have to be done on-the-fly, by applications as they are running and interacting with users. The reason for this is the increasing prevalence of *context-sensitive* applications, sparked by interest in mobile, distributed and personalized applications.

In some sense, abstraction and context-sensitivity are in opposition. Abstraction gains its power from *ignoring* details of a particular situation, which is the last thing you want to do if you want to make an application context-sensitive! But the key to resolving this dilemma is to introduce abstraction dynamically. If the program can decide, on the fly, what aspects to ignore or to take into account according to the particular situation, it can be both general and adaptable at the same time. This will bring object-oriented programming much closer to machine learning.

3 Computers are about manipulating ideas

Computers are about manipulating *ideas*. Not numbers, strings, arrays, lists, or code per se. Ideas. They are about taking ideas that people have in their heads, and, when they are expressible computationally, translating them into a form where computers can manipulate them. The future of Object-Oriented Programming, then, will be in understanding new computational ideas and how they can be expressed. Maybe the word "object" is really just another word for "idea".

I think we should consider radically new ways of representing ideas that are not just limited to conventional textual programming languages, though they may play a part in the process. People have a wide range of representing and communicating ideas in a variety of media, and our programming systems ought to as well. We should work towards ways of letting people express ideas directly, minimizing the amount of specialized knowledge and arcane procedures necessary to express ideas in

computational form. That will benefit the usability of our programming systems, and extend the possibility of their use to new audiences.

Perhaps it is obvious, but people naturally express ideas with words. So one direction that we ought to be going in, is to allow people to express programming concepts in words. Not in keywords, reserved words, or identifiers. Simply in words.

The next section will present some work I have done with my colleague Hugo Liu in exploring the, perhaps crazy, idea that people could program computers directly in natural language. Other interesting approaches to expressing computational ideas in natural language are also being explored, such as Lotfi Zadeh's Computing with Words [9].

There are, of course, other ways to express ideas than talking about them with words. Ideas can also be expressed visually, and the idea of visual programming has been explored, notably in research reported in IEEE Human-Centric Computing (formerly Visual Languages). I think the field of Object-Oriented Programming should take the idea of visual programming more seriously. Some of the objections often raised against visual programming, such as visual programs "taking up more screen space" are obsolete and/or easily overcome. Visual programming could empower a whole new generation of people to do programming, who are "visual thinkers" by nature, who are currently disenfranchised by the overly verbal and symbolic nature of today's programming languages. Approaches which use graphical manipulation to introduce new ways of working with programs, such as the remarkable SubText of Jonathan Edwards [1] represent innovations that deserve more attention.

Finally, ideas are expressed not only in words or pictures, but in actions. There is the possibility that the computer could use a sequence of actions performed by or observed from the user as a representation. I am, of course, a big proponent of this approach, under the name of Programming by Example. I won't go further into it here, but refer the reader to my 2001 book [2] and 2006 collection on End-User Programming [3].

4 Object-Oriented Programming in Natural Language

In this section, I want to present, from my current work, an example of what I think will be an important, but perhaps controversial, direction for Object-Oriented Programming. I don't mean to say that all programming should be done this way, but I think it is an example of new kinds of approaches that ought to be considered.

In the Metafor project [4, 5, 6, 7, 8] we are exploring the idea of using descriptions in a natural language like English as a representation for programs. While we cannot yet convert arbitrary English descriptions to fully specified code, we can use a reasonably expressive subset of English as a conceptualization, visualization, editing and debugging tool. Simple descriptions of program objects and their behavior are converted to scaffolding (underspecified) code fragments, that can be used as feedback for the designer, and which can later be elaborated.

Roughly speaking, noun phrases can be interpreted as program objects; verbs can be functions, adjectives can be properties. It is our contention that today's natural language technology has improved to the point that reasonable, simple, descriptions of simple procedural and object oriented programming ideas can be understood (providing, of course, the user is trying to cooperate with the system, not fool it). There's no need to impose a rigid, unforgiving syntax on a user. Interactive dialogues can provide disambiguation when necessary, or if the system is completely stuck, it falls back on the user.

The principal objection to natural language programming rests in the supposed *ambiguity* of natural language. But ambiguity can be your friend. A surprising amount of information about program structure can be inferred by our parser from relations implicit in the linguistic structure. We refer to this phenomenon as *programmatic semantics*.

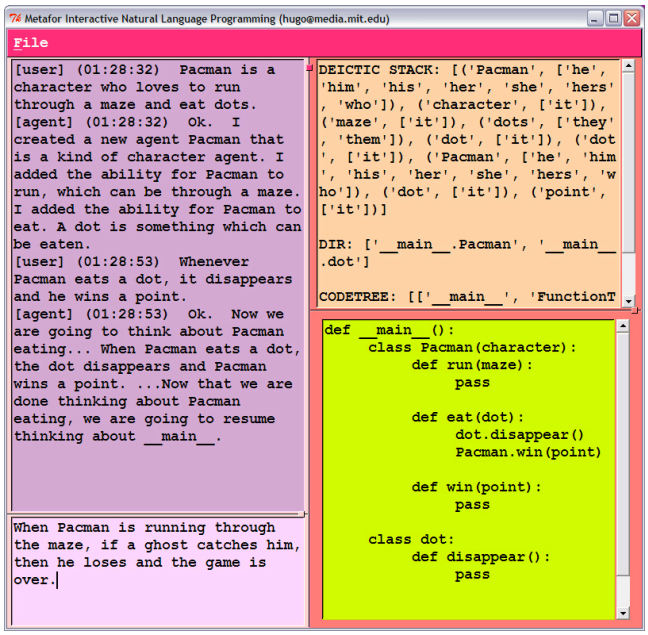

Fig. 1. The Metafor natural language programming system. At the lower left is the user's natural language input. At the lower right the automatically generated code. The two top windows trace the parser's actions and are not intended for the end user.

Metafor has some interesting capabilities for *refactoring* programs. Different ways of describing objects in natural language can give rise to different representation and implementation decisions as embodied in the details of the code. Conventional programming requires making up-front commitments to overspecified details, and saddles the user with having to perform distributed, error-prone edits in order to change design decisions. Metafor uses the inherent "ambiguity" of natural language as

an advantage, automatically performing refactoring as the system learns more about the user's intent. For example,

a) There is a bar. (Single object. But what kind of "bar"?)
b) The bar contains two customers. (unimorphic list. Now, a place serving alcohol)
c) It also contains a waiter. (unimorphic wrt. persons)
d) It also contains some stools. (polymorphic list)
e) The bar opens and closes. (class / agent)
f) The bar is a kind of store. (inheritance class)
g) Some bars close at 6pm. (subclass or instantiatable)

More details about Metafor and natural language programming appear in the references. We realize that this approach is controversial, and many details and problems still need to be worked out. But we present this as an example that radical new approaches to the programming problem need to be considered if Object-Oriented Programming is to advance in the future.

I look forward to the next 20 years of ECOOP!

5 References

1. Jonathan Edwards. Subtext: Uncovering the Simplicity of Programming. 20th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). October 2005, San Diego, California.
2. Henry Lieberman, ed., *Your Wish is My Command: Programming by Example*, Morgan Kaufmann, San Francisco, 2001.
3. Henry Lieberman, Fabio Paterno and Volker Wulf, eds., *End-User Development*, Springer, 2006.
4. Hugo Liu and Henry Lieberman (2004) Toward a Programmatic Semantics of Natural Language. Proceedings of VL/HCC'04: the 20th IEEE Symposium on Visual Languages and Human-Centric Computing. pp. 281-282. September 26-29, 2004, Rome. IEEE Computer Society Press.
5. Hugo Liu and Henry Lieberman (2005) Programmatic Semantics for Natural Language Interfaces. Proceedings of the ACM Conference on Human Factors in Computing Systems, CHI 2005, April 5-7, 2005, Portland, OR, USA. ACM Press.
6. Hugo Liu and Henry Lieberman (2005) Metafor: Visualizing Stories as Code. Proceedings of the ACM International Conference on Intelligent User Interfaces, IUI 2005, January 9-12, 2005, San Diego, CA, USA, to appear. ACM 2005.
7. Henry Lieberman and Hugo Liu. Feasibility Studies for Programming in Natural Language. H. Lieberman, F. Paterno, and V. Wulf (Eds.) Perspectives in End-User Development, to appear. Springer, 2006.
8. Rada Milhacea, Henry Lieberman and Hugo Liu. NLP for NLP: Natural Language Processing for Natural Language Programming, International Conference on Computational Linguistics and Intelligent Text Processing, Mexico City, Springer Lecture Notes in Computer Science, February 2006.
9. Lotfi Zadeh, Precisiated natural language (PNL), *AI Magazine*, Volume 25, Issue 3 Pages: 74 – 91, Fall 2004.