Object-Oriented Programming: Regaining the Excitement

Andrew P. Black

Oregon Graduate Institute of Science & Technology
Portland, Oregon, USA
black@cse.ogi.edu

Abstract. This paper is based on a speech delivered at the ECOOP'98 Conference Banquet. It is not a literal transcription of my talk, since no recording was made, but has been reconstructed *ex post facto* based upon my speaker's notes and my memory. I have also taken the opportunity to add some headings and references.

Distinguished Chairmen, Members of the Conference Committee, Representatives of the sponsoring organizations, distinguished Professors, conference participants, and friends: Good evening.

It's customary to start this kind of talk with a joke, at least in part to give the audience a chance to become accustomed to my strange accent before I start to say anything interesting, but I'm going to skip that tonight because I think that we have already heard quite enough jokes for one evening.

To set the record straight from the very first, I should make it clear that I do not work for IBM, nor have I worked for IBM in the past. Although I did once spend a very enjoyable year at IBM's Yorktown Heights Research Laboratory, IBM was very clear that I didn't work for them, even though I did turn up every day and they did pay me: I believe that the distinction had something to do with social security tax or health insurance.

I should also point out, particularly for the benefit of those of you around the corner who cannot see me, that yes, I am wearing a tie, but no, I don't use a mainframe. I used to use a mainframe, but one day the mainframe broke and all of the little beads came off. [At this point the reader will have to imagine a large broken abacus frame with bent wires and missing beads.]

1 Inventing the Future of Object Technology

The right thing to do in a talk of this nature is to predict the future of object technology, but prediction is hard, and predicting the future is especially hard! Alan Kay once said: "The best way to predict the future is to invent it". And he was remarkably accurate. Let me read you a quote from over 20 years ago.

> In the 1990's there will be millions of personal computers. They will be the size of notebooks of today, have high resolution flat screen reflective displays, weigh less than 10 pounds, have 10 to 20 times the computing and storage capacity of an Alto. Let's call them Dynabooks.
>
> The purchase price will be about that of a color television set of the era...
>
> Though the Dynabook will have considerable local storage and do most computing locally, it will spend a large percentage of its time hooked to various large, global information utilities which will permit communication with others of ideas, data, working models as well as the daily chit-chat that organizations need in order to function.

This is from Alan's paper "The Early History of Smalltalk"[5], in which Alan quotes an internal Xerox PARC memo from around 1976. I recommend that paper most highly: if you find any wisdom in my remarks this evening, please attribute it to Alan Kay and not to me.

My goal tonight is to challenge you to put the excitement back into object-orientation, and to recapture some of the dynamism of those early Smalltalk days.

Is this possible? Or has object-orientation become like structured programming: the right idea, but no longer the focus of innovation, exactly because everyone is already doing it. For example, we don't have a European Conference on Structured Programming every year.

I don't think that object-orientation is yet at that point. There are still many hard problems to solve: scale and encapsulation are two that I will mention briefly tonight.

2 Programming Language Contributions Relevant to Object-orientation

The programming language research community has historically been a great source of innovation. I believe that we should challenge ourselves to recapture that atmosphere of innovation; to help us on our way, I will take a brief look at some of the ideas that have been "brought to market" by influential programming languages over the last forty years. I'm not claiming that the listed languages invented the concepts in every case, but rather that these languages were among the first to popularize them.

2.1 Lisp (late 1950s)

Lisp was a startling language for its time: I believe that Lisp is one of Alan Kay's "almost new things". It contributed:

- heap allocation and garbage collection,
- interpreted execution, and
- conditional expressions.

2.2 Algol 60

Another "almost new thing", in contrast to the "better old things" of COBOL and FORTRAN. Its contributions include:

– a semi-formal, concise definition,
– grammars as a descriptive technique,
– block structure,
– recursion—but not the word "recursion";
    - recursion introduced the distinction between program text (or "static instance") and its execution (or "dynamic instance"). This distinction is the kernel of the idea that became Simula Objects [7]. Algol 60 also introduced
– declarations,
– call-by-name,
– "own" variables,
– the extension of the concept of expression to non-numeric values,
    - Boolean expressions, and
    - "designational expressions", and
– the idea of "Security", which we would probably today call semantic integrity.

Security is the principle that a program must either be rejected as incorrect by compile-time or run-time checks, or *its behavior must be understandable by reasoning based entirely on the language semantics*, independent of the implementation [2,7].

However, no language is perfect. Two "non-contributions" of Algol 60 were:

– that the non-numeric expressions that Algol 60 chose to include were labels, not procedures; and that
– input/output was relegated to a library.

2.3 Algol W, Pascal and Simula 67

Contributions:

– quasi-parallel execution (previously introduced in Knuth's SOL);
– user-definable types;
– static inheritance (Simula's prefix classes);
– use of prefix classes to provide "language dialects"
– Sum types, also known as variant types, or discriminated unions;
– the combination of data and operations in a single language construct; and
– the idea that binding of a value to a name requires only type *compatibility*, not type identity.

Non-contributions:

– Explicit reference variables, and
– Pascal's sacrifice of the security of Algol 60 by the variant record construct.

This last innovation is perhaps one of the reasons that Tony Hoare said of Algol 60: "here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors"[2].

2.4 Algol 68

Contributions:

- unification of statement and expression, that is, making the distinction between them one of type, rather than one of context-free syntax;
- the void type;
- coercion;
- environment enquiries;
- conceptual minimality, for example,
  - the use of references to eliminate the notion of variable, and
  - the use of procedures to eliminate call-by-name; and
- obtaining an extensible syntax—then an important goal—by means of the definition of new operators, and by that means alone.

Non-contributions:

- 2-level grammars as a descriptive technique, and
- once again, breaching security, this time by requiring that the programmer not export a stack-allocated variable outside of its scope, which is not something for which implementations can easily check.

2.5 CLU, Alphard, Modula (1974-78)

Contributions:

- encapsulation,
- named scopes (*i.e.*, the module concept),
- parametric polymorphism, and
- F-bounded polymorphism (CLU's where clauses)
  - but it's not clear that the CLU designers realized at the time what they had invented.

2.6 Emerald (1984-6)

Contributions:

- safe, static subtyping,
- F-bounded polymorphism (in the interpretation of conformity in where clauses)[4]
  - but we also did not yet know that it would be given this name,
- mobile objects,
- location-independent invocation,
- object constructors (which I believe are relevant to Ole Lehrmann Madsen's quest to integrate prototype and class-based programming styles),
- separation of type from implementation, and
- the "open world" assumption

This last is the idea that new objects and new classes (superclasses as well as subclasses) can be added to a system at any time, so that we don't have to shut down and recompile the internet.

2.7 Later Developments

What has happened since then?

- multi-paradigm languages (LIFE, Leda, *etc.*)
- higher-order functional languages with object-oriented concepts (Objective CAML, Haskel with its type classes, *etc.*)

These are interesting, but not earth-shaking.

And then, of course, we have Java. What are Java's advances over Emerald? The major one is of course replacing word pair brackets (like **do** ... **end**) with curly braces {}, but we should not overlook typed byte-code and byte-code verification.

The latter means that one can obtain a class that someone else has compiled and be sure that it is type-safe. This is a real advance, although proof-carrying code [6] may well be a more elegant and more widely applicable way of achieving the same effect.

3 A Research Agenda

To summarize this historical review, I believe that Programming Language Research has made *major* contributions to the state of computing today, but that recently the focus of research has moved away from pure OO towards hybrid languages and applications. These are interesting topics, but I do not believe that we have completed the development of object-orientation itself.

3.1 Objects Demonstrate Recursive Structure

My prescription is that we should re-examine the core of object-orientation. Going back to Smalltalk, we find that there are two key ideas:

- that objects localize *data structures* and the *code* that operates on them in the same place; and
- that objects, using the words of Alan Kay again: are *a recursion on the idea of computer itself.*

Kay writes: "The basic principle of recursive design is to make the parts have the same power as the whole". Rather than dividing the computer into "lesser stuffs", like data structures and procedures, we should divide it into lots of little computers that communicate together. This enables us to postpone representation decisions almost indefinitely (and those are the decisions that are almost always the cause of our troubles).

This sounds like a model for distributed computing to me.

The World Wide Web is the largest and most successful distributed object-oriented system ever built—but what a poor system it is! For example, where is location independent naming? And how does one introduce a new class? The answer to the latter question is that we persuade the W3 consortium to call an international meeting to get agreement on a new protocol!

I propose that we take a long hard look at Kay's three basic laws of object-oriented programming.

1. Everything is an object.
2. Objects communicate by sending and receiving messages (in term of objects).
3. Objects have their own memory (in terms of objects).

3.2 Information Hiding

Notice that there is nothing in the concept of little computers communicating with each other and protecting their own data that speaks of *how* to make information hiding decisions.

Recall that the first key idea behind objects is "localization", not "encapsulation". Parnas' idea of *information hiding* is a strong and powerful idea in program structuring: implementations of abstractions should hide as much as possible about themselves (but no more)!

Hence, there are two parts to each information hiding decision: we must specify not only *what* to hide, but also from *whom.* The weakness of most encapsulation mechanisms is that they typically give the programmer fairly good control on the *what*, but very little on the *whom.*

Java gives us three sets for whom: methods in this object, methods in objects that subclass from this object, and the rest of the world. But this is inadequate. I will give just two examples.

– Think about allowing accessor and updater methods access to a slot, but prohibiting that access to other methods in the same object.
– Think about a persistent system: how can programmers evolve the contents of an object in the store if they can't access its fields?

Don't be misled into thinking that this last class of access somehow takes place "outside" of the system; our goal should be to describe the whole world as objects, and to use the reflective properties of those objects to examine and change their structure and their protocol. To put it another way: I want objects to take over the world, so I'm not prepared to start by excluding our own programming environments from the set of things that I can describe with objects. To the contrary: we should be using the reflective properties of objects to *build* our environments.

Rather than the "all or nothing" encapsulation paradigm, what we need is a concept of encapsulation that is much closer to what operating system workers have thought of as "protection mechanisms". For example, one programmer might take on several different roles in relation to the *same* object: she may be a maintenance programmer, then a user, and maybe later an administrator, and under those roles, the objects "in the programmer's hand" should have different access rights to various parts of the target object.

3.3 Objects as an Integrative Paradigm

The third research direction that I would like to propose is that of objects as an integrative paradigm. The OO model is strong enough to capture functional and procedural styles of programming, but doing this naively is likely to result in "write only programs".

Let me explain. Many "idioms" can be written and not read. For example, I have used the functional style to build a Smalltalk Interval class in which the step and the limit slots hold blocks that are the step function and the limit predicate. Smalltalk programmers will be taken by surprise by such code.

The reverse is also true: I can build objects in ML, but ML programmers will not be able to read the resulting code.

Consequently, I do believe that multi-paradigm languages have a place, but I also believe that the non-OO features should be mapped onto objects. For example, functions should be objects with the convention that they have exactly one operation called $\circ$ (meaning "apply").

Nevertheless, it is important that the new features should be given an appropriate and readable syntax. The approach that I am advocating should enable us to retain semantic simplicity and avoid unpleasant interactions between features.

During Tuesday's Panel session, one of the questioners stated that his "old professor" (I do hate that phrase, now that I am one) said that:

> Programming language people think that they can solve a problem by designing a language to express it.

I believe that this professor was right, and that such "language people" are also right! The reason is that *language shapes thought:* first we shape the tool, and then the tool shapes us.

So it is vital that the tool be well-made. Category theory in mathematics is a good example: by exposing similarities between different branches of mathematics, some proofs that were obscure and difficult become self-evident. Lazy functional languages bring some of that clarity to programming certain kinds of problems.

Does clarity come from having to force your problem into the mold of the language? Only if it works! Sometimes, the struggle to force your problem into a notation that does not seem appropriate can yield great insight; sometimes it is a waste of time.

3.4 Scale

The final challenge is scaling, but we must ask: in what dimension? The answer is:

- in space, that is, wide area distribution;
- in time, that is, Persistent Object Systems;
- and, of course, in their combination.

Scaling in Space. Systems like Emerald that successfully handle distribution in the small do so by seeking to hide the distinction between local and remote objects. But the failure modes of local and remote objects are inherently different, and performance is radically reduced by distribution. This means that object placement is absolutely critical to application performance—and the tools that are currently available to automate or even understand placement are minimal.

Another problem is how to replicate an object and still preserve its "object nature".

The systems community has spent 20 years working on replication, and we now know how to implement reliable and resilient systems from replicated communicating objects. But once we have done so, how can we treat the resulting subsystem as a single object, in order to hide its internal structure from its clients?

A student (Mark Immel) and I presented one solution to this problem at ECOOP'93 [1], but I can't believe that it is the only solution, or even that it is necessarily a very good one. I encourage you to work on this problem.

Scaling in Time. I will defer to the experts who will be giving tomorrow morning's invited lecture on Persistence in Java—Malcolm Atkinson and Mick Jordan—by mentioning only briefly some of the issues that must be resolved:

- compilation distributed in time;
- explicit support for versioning;
- safe and unsafe evolution of types;
- new interfaces for old objects; and
- new objects for old interfaces.

4 Summary

The Programming Language Research community has contributed massively to our progress in object-orientation to date. But recently, the pace of innovation seems to have slowed: we are making better old things, but I don't see the almost new things.

My message is the following.

- Go back to the roots of object-orientation:
  - build little computational engines that communicate with one another to mimic the real world that we seek to model.
- Use objects as an integrative paradigm:
  - build a semantic model that will serve for concurrent, functional, object-oriented and, if possible, constraint-based and logic programming; but
  - remember that the purpose of a program is to be read; and
  - that idioms reduce readability whereas well-designed language constructs enhance it.
- Deal with the hard problems: scale, persistence, and their cross-product.

– Take encapsulation seriously, perhaps learning from the OS community.

But I have to caution you to be careful of what you learn from the Operating Systems Community.

At today's conference, I am reminded of the 1983 ACM Symposium on Operating Systems Principles, which took place at the Mt. Washington Hotel at Bretton Woods in New Hampshire. This is the same hotel that was the site of the famous Bretton Woods monetary conference, which was held there toward the close of the Second World War. Although the plumbing hadn't been touched since, it was a wonderfully atmospheric old hotel. I should add that it has since been completely renovated.

I was young Assistant Professor, presenting my first paper at a conference, and was somewhat overawed by all of the "grand old men" of operating systems; the conference has been called the Symposium for OS Princi*pals*, and that is only partially a joke.

I remember two things about the conference. The first is that it marked one of the early replicated failures of a replicated resilient system.

The Grapevine nameservice had recently been built at Xerox PARC, and was the subject of a paper [9, 8] to be presented at the conference that celebrated how, by supporting multiple servers that exchanged updates between them, the service had proved resilient to failure of one or even several of the individual servers.

Unfortunately, all of the servers ran the same code on the same hardware, so when a particular packet was sent to the first server, triggered a bug, and caused the server to crash, the client responded by re-sending the same packet to the second server, and crashing that, and so on up the replication chain. As I recall, this packet was sent just at the time that all of the principals who might have diagnosed the problem had left Palo Alto to travel to Bretton Woods.

The second thing that I recall about this conference is that it had shared "resources" in the form of bathrooms between adjacent pairs of rooms. It also had multiple processors, in the form of the occupants of the two rooms. Thus, to avoid collision, both processors had to correctly execute the bathroom door locking protocol.

At breakfast on the first morning of the conference, many of these notable OS wizards were missing. It turned out that they had deadlocked contending for the bathrooms and the hotel staff had to be called to execute a manual reset.

Thank you for listening so patiently; I hope that you enjoy the rest of your evening and the remainder of the conference.

References

1. Andrew P. Black and Mark P. Immel. Encapsulating plurality. In *Proceedings of European Conference on Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 57–79. Springer-Verlag, July 1993.
2. Charles Antony Richard Hoare. Hints on programming language design. Invited address at ACM SIGACT/SIGPLAN Symposium on Principles of Programming languages, (Reprinted in Horowitz [3], pages 35–40), October 1973.

3. Ellis Horowitz, editor. *Programming Languages: A Grand Tour*. Computer Science Press, third edition, 1987.
4. Norman Hutchinson. *Emerald: An Object-Oriented Language for Distributed Programming*. PhD thesis, University of Washington, Department of Computer Science, January 1987.
5. Alan C. Kay. The early history of Smalltalk. In *Preprints of the Second ACM SIGPLAN History of Programming Languages Conference (HOPL–II)*, pages 69–98, Cambridge, Massachusetts, March 1993. ACM SIGPLAN.
6. G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, 1998.
7. Kristen Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. In Richard L. Wexelblat, editor, *History of Programming Languages*, chapter IX, pages 439–493. Academic Press, 1981.
8. Michael D. Schroeder, Andrew D. Birell, and Roger M. Needham. Experience with Grapevine: The growth of a distributed system. *ACM Transactions on Computer Systems*, 2(1):3–23, February 1984.
9. Michael D. Schroeder, Andrew D.Birrell, and Roger M. Needham. Experience with Grapevine: The growth of a distributed system (Summary). In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 141–142. ACM SIGOPS, October 1983.