# Objects Never? Well, Hardly Ever!

Mordechai (Moti) Ben-Ari

Department of Science Teaching

Weizmann Institute of Science

Rehovot 76100 Israel

http://stwww.weizmann.ac.il/g-cs/benari/

Copyright 2010 by Mordechai (Moti) Ben-Ari. This work is licensed under the Creative Commons Attribution Non-Commercial No Derivs 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/; or, (b) send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

This article are originally published in the Communications of the ACM 53, 9 (Sep. 2010), 32-35. DOI=http://doi.acm.org/10.1145/1810891.1810905.

## The great objects debate

At the 2005 SIGCSE (Special Interest Group in Computer Science Education) Symposium in St. Louis, a packed audience listened to the Great Objects Debate: should we teach “objects first" or “objects later”? [1] In the objects-first approach, novices are taught *object-oriented programming (OOP)* in their initial introduction to programming, as opposed to an objects-later approach, where novices are first introduced to procedural programming, leaving OOP to the end of the first semester or the end of the first year. Kim Bruce and Michael Kölling spoke in favor of objects-first, while their opponents were Stuart Reges and Eliot Koffman. One of Kim’s arguments was this: since OOP is dominant in the world of software development, it should be taught early. I later contacted Kim to ask for a warrant for the dominance of OOP, but he could not give me one, nor could any of several other experts to whom I posed the same question.

I claim that the use of object-oriented programming is not as prevalent as most people believe, that it is not as successful as its proponents claim, and, therefore, that its central place in the CS curriculum is not justified.

## Is OOP dominant?

In assessing the dominance of OOP, we have to watch out for proxies. The extensive use of languages that support OOP proves nothing, because languages are chosen for a myriad of reasons, not necessarily for their suitability for OOP, nor for the suitability of OOP itself. Similarly, the use of a CASE tool that supports OOP is another proxy; these tools might just be

convenient and effective for expressing the software design of a system, whether OOP is being used or not. Furthermore, many practices associated with OOP, such as decomposing software into modules and separating the interface from the implementation, are not limited to OOP; they are simply good software practice and have been supported by modern programming languages and systems for years.

The classical definition of OOP was given by Peter Wegner [9, p. 169]:

object-oriented = objects + classes + inheritance.

The Java Swing GUI library, which makes massive use of inheritance, is frequently mentioned as a successful example of software that was designed using object-orientation and it certainly fits Wegner's definition. Is this style of programming truly dominant?

There is a claim that 90% of all code is being written for embedded systems [7, p. 357]. I could not locate the author's source, but it doesn't really matter since the claim is just as suspect as the claim that OOP is dominant. However, embedded system development is surely an important field of software, and, based upon my experience, I do not believe that OOP has a significant contribution to make here, because the main challenges are not in the *software* design itself. The challenges arise from an "unfriendly environment": getting proprietary hardware to work, obtaining meaningful requirements while the system itself is being designed, integration with non-standard networks and busses, and, above all, figuring out how to test and verify the software.

Consider another field where the dominance of OOP is questionable. The development and implementation of new algorithms form the heart of many applications areas like numerical simulation (for example, climate modeling) and image processing (for example of satellite imagery). The challenges arise from mathematical difficulties and demands for performance, and OOP has little to contribute to meeting these challenges.

Not only is there no evidence to back up the claims for the dominance of OOP, but there is criticism of OOP, some of it quite harsh [3, 8]. I, too, have found OOP to be extremely disappointing and I would to explain my position from a personal perspective.

**What the "real world" is really like**

Suppose you ask your students to design OOP software for a car; you would probably give a good grade for:

```
abstract class Brake {
  public abstract void applyBrakes();
}
class DiskBrake extends Brake { ... }
class DrumBrake extends Brake { ... }
```

The only problem is that the real world doesn't work this way. A wonderful image in [5] shows the schematic of the computer system of the Mercedes-Benz S-class car. The legend says that there are over 50 controllers, 600,000 lines of code, hundreds of bus messages, thousands of signals and three networks. The details of this system are proprietary, but I am confident that no one sat down and used OOP to "design the software," for example, by deriving classes as shown above. Almost certainly, the various subsystems were subcontracted to different companies who jealously guard their software because they are engaged in merciless competition.

The interface to the brake system will be implemented by network protocols and bus signals, and the commands to the brakes will be given as bits and bytes (or even by a hardware specification like "apply the brakes when lines 1 and 5 are asserted continuously for at least 10 milliseconds"). An abstract specification like `void applyBrakes()` is meaningless here. More importantly, what is likely to be changed is the *interface*, contrary to the OOP approach, which assumes that different implementations will be "swapped" at a single interface. Let us imagine that at some time in the future the brake manufacturer is asked to supply systems to Daimler archrival BMW. The mechanics, hydraulics, electronics and algorithms will be re-used, but the network protocols and bus signals will certainly require significant modification to fit the systems architecture used by BMW.

I believe that industrial systems are successful because the decomposition is not into *classes*, but into *subsystems*. The Mercedes-Benz car has, on the average, 600,000 / 50 = 12,000 source code lines per controller, so each individual subsystem can be developed by a relatively small team in a relatively short time. There is a need for talented systems engineers to specify and integrate the subsystems, but there is no overall grand software design where OOP might help.

**Natural and intuitive**

In the 43 years since I first learned to program, I have frequently become excited about developments in programming, such as pattern matching (which I first encountered in SNOBOL) and strong type checking (a revelation when I first learned Pascal), and I found that these new constructs *naturally and intuitively* supported solutions to programming tasks. I have *never* had the same feeling about OOP, despite teaching it, writing textbooks on OOP languages, and developing pedagogical software in Java. During all this time, I found only one natural use of inheritance. (I developed a tool for learning distributed algorithms [2] and found it convenient to declare an abstract class containing the common fields and methods of the algorithms and then to declare derived classes for specific algorithms.) Isn't it just possible that my inability to profit from OOP reflects a problem with OOP itself and not my own incompetence?

I am not the only one whose intuition fails when it comes to OOP. Hadar and Leron recently investigated the acquisition of OOP concepts by *experienced* software developers. They found that: "Under the demands of abstraction, formalization, and executability, the formal OO paradigm has come to sometimes clash with the very intuitions that produced it" [6, p. 45]. Again, isn't it just possible that the intuition of experienced software engineers is perfectly OK, and that it is OOP that is not intuitive and frequently even artificial?

**Reuse from the trenches**

One of the strongest claims in favor of OOP is that it facilitates reuse. I would like to see evidence to support this, because, in my experience, OOP makes reuse difficult, if not impossible. Here I would like to describe two attempts at reuse where I truly felt that OOP was the problem and not the solution. I would like to emphasize that—as far as I can judge—these programs were designed according to the principles of OOP, and the quality of the design and programming was excellent.

I developed the first concurrency simulator for teaching based upon a Pascal interpreter written by Niklaus Wirth. Several years ago, while looking for a modern concurrency simulator, I found a third-generation descendant of my simulator: an interpreter written in Java, extended with a debugger that had a Swing-based GUI. I wished to modify this software to interpret additional byte codes and to expand the GUI by including an editor and a command to invoke the compiler.

The heart of an interpreter is a large switch/case-statement on the instruction codes. An often-cited advantage of OOP is its ability to replace these statements with dynamic dispatching. In the Java program, an abstract class for byte codes was defined, and from it, other abstract and concrete classes were derived for each of the byte codes. I simply found it more difficult (even with Eclipse) to browse and modify 80 classes than I did when there were 80 alternatives of a case-statement in Pascal.

This was only an annoyance; the real problem quickly surfaced. The extreme encapsulation encouraged by OOP caused untold complexity, because objects have to be passed to various classes via constructors. For example, in the original program, when a button is clicked to request the display of the history window, the statement performed in the event handler is:

```
getDebugger().getDebuggerFrame().getWindowManager().
      showHistoryWindow(
          getDebugger(), getDebugger().getInterpreter());
```

Well, the history window is derived from an abstract window class, so OOP makes sense here, but there is *one* debugger, *one* debugger frame, *one* interpreter, and *one* window manager. Why can't these *subsystems* be declared publicly (and implemented privately) without the baggage of allocated objects and constructors? My attempt to modify the software was continually plagued by the need to access one of these subsystems from a class that had not been passed the proper object. This resulted in cascades of modifications and complicated the task considerably; in addition, it led to a decline in coherence and cohesion. As a result of this experience, I have ceased to automatically encapsulate everything; instead, I judge each case on its own merits. In general, I see nothing wrong with declaring record types and subsystem objects publicly, encapsulating only the implementation of data structures that are likely to change.

My second attempt at reusing OOP software involved a software tool VN that I developed for learning nondeterminism. It takes as input the XML description of a nondeterministic finite automaton that is generated by an interactive graphical tool for studying automata. To facilitate using VN as a single program, I decided to extract the graphics editor from the other tool. But OOP is about *classes* and Java enables the use of any public declaration anywhere in a program just by giving its fully expanded name. There were just enough such references to induce a cascade of dependencies when I tried to extract the Java package containing the graphics editor.

This is precisely the issue that I raised with the imaginary brake system. What I wanted to re-use was the *implementation* of the graphics editor even if that meant modifying the *interface*. I saw that I would have had to study many of the 400 or so classes in 40 packages, just to extract one package. The effort did not seem worthwhile, so I gave up the idea of reusing the package and included the (very large) jar file of the other tool in my distribution.

**Paradigms**

I know what your next question is going to be: What *paradigm* to you propose instead of OOP? Ever heretical, I would like to question the whole concept of *programming paradigm*. What paradigms are used to design bridges? My guess is that the concept of paradigm does not exist there. Engineering design is done by using technology to implement requirements. The engineer starts from data (length of the bridge, depth of the water, characteristics of the river bed) and constraints (budget, schedule), and she has technology to use in her design: architecture (cables, stays, trusses) and materials (steel, concrete). I simply don't see a set of alternative "paradigms" for building bridges.

Similarly, the software engineer is faced with requirements and constraints, and is required to meet them with technology: computer architectures, communication links, operating systems, programming languages, libraries, and so on. Systems are constructed in complex ways from these technologies, and the concept of programming paradigm is of little use in the real world.

**Hegemony**

It is easy (and not incorrect) to dismiss what I have written as personal opinion and local anecdotes, just as I have dismissed OOP as based upon personal opinion and local anecdotes without solid evidence to support its claims. But the difference between me and the proponents of OOP is that I am not making any hegemonic claims for my opinions. I do not believe that there is a "most successful" [4, p. 33] way of structuring software nor that any method is "dominant." This hegemony is particularly apparent in CS education, as evidenced by the objects-first vs. objects-later debate that concerns teaching OOP to novices. No one questions whether OOP is at all appropriate for novices, and no one suggests an *objects-as-an-upper-level-elective* approach or an *objects-in-graduate-school* approach. Perhaps the time has come to do so.

**Conclusion**

Let me conclude with a “to-do list”:

- Proponents of OOP should publish analyses of successes and failures of OOP, and use these to clearly and explicitly characterize the domains in which OOP can be recommended.
- Software engineers should always use their judgment when choosing tools and techniques and not be carried away by unsubstantiated claims. Even if you are constrained to use a language or tool that supports OOP, that in itself is not a reason to use OOP as a design method if you judge that it is not appropriate.
- Educators should ensure that students are given a broad exposure to programming languages and techniques. I would especially like to see the education of novices become more diverse. No harm will come to them if they see objects very, very, late.

**References**

[1] Astrachan, O., Bruce, K., Koffman, E., Kölling, M., and Reges, S. 2005. Resolved: objects early has failed. *SIGCSE Bull.* 37, 1 (Feb. 2005), 451-452. DOI= http://doi.acm.org/10.1145/1047124.1047359.

[2] Ben-Ari, M. 1997. Distributed algorithms in Java. *SIGCSE Bull.* 29, 3 (Sep. 1997), 62-64. DOI= http://doi.acm.org/10.1145/268809.268840.

[3] Gabriel, R. 2002. *Objects Have Failed: Notes for a Debate.* http://www.dreamsongs.com/Files/ObjectsHaveFailed.pdf (retrieved 17 May 2009).

[4] Gries, D. 2008. A principled approach to teaching OOP first. *SIGCSE Bull.* 40, 1 (Feb. 2008), 31-35. DOI= http://doi.acm.org/10.1145/1352322.1352149

[5] Grimm, K. 2003. Software technology in an automotive company: Major challenges. In *Proceedings of the 25th international Conference on Software Engineering* (Portland, Oregon, May 03 - 10, 2003). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 498-503.

[6] Hadar, I. and Leron, U. 2008. How intuitive is object-oriented design?. *Commun. ACM* 51, 5 (May. 2008), 41-46. DOI= http://doi.acm.org/10.1145/1342327.1342336.

[7] Hartenstein, R. 2004. The digital divide of computing. In *Proceedings of the 1st Conference on Computing Frontiers* (Ischia, Italy, April 14 - 16, 2004). CF '04. ACM, New York, NY, 357-362. DOI= http://doi.acm.org/10.1145/977091.977144.

[8] Jacobs, B. *Object Oriented Programming Oversold!* http://www.geocities.com/tablizer/oopbad.htm (retrieved 17 May 2009).

[9] Wegner, P. 1987. Dimensions of object-based language design. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (Orlando, Florida, United States, October 04 - 08, 1987). N. Meyrowitz, Ed. OOPSLA '87. ACM, New York, NY, 168-182. DOI= http://doi.acm.org/10.1145/38765.38823.