Henry F. Ledgard

The Emperor with No Clothes

Examining the software problem from a scientific standpoint.

"If you can't measure it, you can't improve it."
—W. Edwards Deming

"However, just as a sailor can sense a changing sea, I note subtle signs that point to a marked transformation, a disruptive technology, on the horizon"
—Grady Booch

The true problem with software is hardware. We have been seduced by the promise of more and more and have become entranced under the spell of Moore's Law. Continued progress in hardware is not a friend, but our nemesis. We have been shielded by hardware advances from confronting our own incompetence as software professionals and our immaturity as an engineering profession.
—Larry Constantine

Is it possible that the software industry is finally about to make a bold step? Are we ready to uncover our eyes and look at the software problem—from a scientific standpoint?

We have, so far, experienced different theories of software construction paradigms in the last three decades, such as structured programming in the 1970s, and object-oriented programming (OOP) and CASE in the 1980s. Although OOP dominates current beliefs, there is no measure of the productivity of this methodology relative to another.

A View of History

Relevant to our view is the C language. C started in the early 1970s with an attempt to build a Fortran compiler on the PDP-7 at Bell Laboratories, and the desire to build compilers and utilities for Unix in a language for which it was easy to write a compiler [5]. Coming out of Bell Laboratories was a plus. And it was good—in the time and context for which it was designed. As time passed, investments in C grew. The ACM and IEEE became sources for significant publications on Unix and C.

In parallel with the development of C and Unix in the 1970s, there was a totally separate movement to improve programmer productivity. International conferences were held that pointed to the lack of a disciplined approach to building software. Topics ranged from configuration management to top-down design, structured programming, one-in one-out control structures, and goto-less programming. The concepts for improving programmer productivity during this movement were sound. The problem was that no technology (language) existed to implement the high-level design concepts and programmer disciplines described.

The interest in productivity also led to CASE. This was possibly a solution to the software problem, going from specifications to working code in a disciplined way. After about five years, it was realized that once programmers started to code, CASE did not really help.

In parallel with the development of CASE came the C++ language [7]—a super set of C, but still a simple language. As described in the initial paper by Bjarne Stroustrup, C++ made up for some obvious faults in C. This evolved with OOP and classes. OOP and classes are now a dominant part of the programmer landscape, and in Java as well.

Java was developed in the mid-1990s, and was originally aimed at building Web pages. It, in turn, attempted to solve some C++ problems. Java eliminated the C++ facilities for pointers and memory management—the three largest causes of software bugs. In short order, Java has evolved as direct competition to C++. There is no evidence Java is any more productive than C++ for developing large pieces of software.

MARTIN MAYO

The OOP Paradigm

The OOP paradigm is derived from the notion of abstract data type (ADT). Since this is an algebraic concept, ADT works well on ~~non~~mathematical objects. In real-world programs, however, we encounter nonmathematical objects far more frequently than mathematical objects. Programmers deal with functions, common data, intermediate structure, complex operations, and the like. Therefore, in order to map those mathematical objects to algebraic-type-based classes, programmers often have to twist the structure.

The results tend to be far from a natural representation of everyday experiences [3], contrary to what many OO supporters advocate. Thus, much of what can be observed as OOP is just a variation of "encapsulation." It is my contention that many OO designs are really a complex form of "encapsulation" that is not an OO approach.

Consider constructing an inventory program. Such a program requires many files, a few packaged databases, and various user interactions. If one pursues OO software construction, those objects are implemented as classes and subclasses with deep and complex inheritances. In general, one cannot treat a class as a complete black box. Therefore, the programmer who intends to reuse a class must first understand some of the inner workings of the class. If the class is a subclass in a class hierarchy, one has to traverse the path of the inheritance to find the method or data one wants to reuse. In a sense, the programmer has to trace through the labyrinth of class hierarchy.

This is known as the "yo-yo" problem, and has existed since the very beginning [4] of OOP. We can say that hiding information about data structures does not facilitate the user's understanding and ability to support the program.

The reality is, OOP is difficult to learn, taking the average programmer about two years to master. With the world of software persuaded to move to this new technology, programmers who make the required career learning investment quickly become valuable (indispensable), particularly after they write some important code. No one else can understand it. If one wants a raise, one threatens to leave.

Even the programmers themselves are arguing about the difficulties in dealing with the high level of complexity being placed upon them today.

The Productivity Failure

Numerous articles have been published in many periodicals, from *IEEE Software* and ACM journals, to *Business Week* and the *Wall Street Journal*. Mountains of PR suggest the great productivity improvements we could expect from OOP technology. Every programmer concerned about this technology learned to embrace this view. The truth is, the productivity improvements never happened. In fact, studies have shown software productivity as a whole has been in a decline for more than a decade. The five years prior to 1996 (the heydays of OOP), a period of general growth, software productivity was negative, more negative than any other industry [2].

What is most interesting is that no one appears to be talking in terms of productivity (for an exception, see [1]). They are all lost in the mazes of theoretical beauty of their creations. According to quality control expert Deming, if you can't measure it, you can't improve it. So why is it so difficult to bring computer science into the world of hard science and to start measuring the real results of some of these hypotheses?

In the 1960s, people actually measured language understandability (see [6])—a major factor in

software productivity. The switch from assemblers to higher-level compiler languages was a clearly measurable improvement in productivity. But it was difficult to effect the change. This was because the programmers that knew assemblers were indispensable. It was much easier for someone else to understand high-level language code. When the existing masses have a vested interest in the current approach, change that decreases the value of their investment is not in their best interests.

The field has yet to measure the productivity of competing software development environments. The software industry deserves some objective investigations in this area. In today's situation, one can say that, without measures from repeatable experiments, software is not a science.

It is time we uncover our eyes and face the software dilemma we have today—an industry with no scientific measure of productivity, just emperors with no clothes. C

Henry Ledgard (hledgard@eng.utoledo.edu) is a professor in the Electrical Engineering and Computer Science Department at the University of Toledo.

References

1. Cartwright M. and Shepperd, M. An empirical investigation of an object-oriented software system. *IEEE Trans. Softw. Eng.* (Aug. 2000).
2. Cave, W.C. Software survivors. *Software Developer and Publisher* (July/Aug. 1996).
3. Hatton, R. Does OO sync with how we think? *IEEE Software* (May/Jun. 1998), 46–54.
4. Kaehler, T. Patterson, D. *A Taste of Smalltalk*. Norton, 1986.
5. Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1973.
6. Ledgard, H., Whiteside, J., and Singer, A. The natural language of interactive systems. *Commun. ACM* 18, 11(Nov. 1975).
7. Stroustrup, B. What is object-oriented programming. *IEEE Software* (May 1988).