

The State of the Software Industry

Henry Ledgard (University of Toledo) - October 2007

INTRODUCTION

Review of the software literature reveals widening disparities in thinking about the field. By many measures, it is the most important industry in the world. Its applications appear unlimited. Yet, experienced technologists characterize the field as a craft - not an industry - in its infancy, [SR]. Authors, in top slots at Microsoft and other software companies, are genuinely concerned about the future of the industry, [GR], [GU], [PO], [SU1]. Repeated observations are that we do not seem to learn from history, and that we lack the scientific discipline to investigate claims of what is “best.” “There is no such thing as architecture - just spaghetti code”, [CU]. It is important to understand that what may appear to be negative comments by many authors are sincere calls for improvement in the way we build software.

As an example of the history issue, many concepts from the structured programming era of the 1970s were based upon comparisons of a large body of languages in use. Software quality measures were developed, being characterized by reliability, run-time speed, and time and cost of support, i.e., to make corrections and enhancements. GO TOs were shown to decrease understandability, which affected reliability and support costs. Based upon comparisons, the use of GO TOs quickly became extinct. But most of the other conclusions, such as top-down design, improved declaration of data types, creation of hierarchical data structures, restricting data declarations to appear before instructions, and the full use and benefit of one-in one-out control structures have been forgotten.

C-based languages (C / C++ / C# / Java / etc.) have become so popular, they are the only ones taught in schools and used for new projects. FORTRAN, COBOL, PL1, Pascal, Basic, and other language environments are considered “legacy,” and becoming extinct. Students in Computer Science learn only C-based languages to gain employment.

Comparing software literature to that of other scientific fields, e.g., communications, control, or circuit theory, one observes a lack of well-defined measures, and only occasional results that compare measures based upon experiment. Personal experience is often cited as proof, yet it would be considered conjecture in a scientific evaluation.

Following the Moore’s curve, CPU clocks speeds have doubled every 18 months for the past 25 years. These speed increases covered the fact that software productivity and software run time speeds have been going down. This led to (Niklaus) Wirth’s Law: “Software gets slower faster than hardware gets faster.” But the clock has stopped. Moore’s law no longer supports increases in speed without using multiple CPUs. Current approaches to programming are headed for negative exposure on two fronts - run-time speed and productivity. As poor productivity moves jobs off-shore, and programming multiple processors to maintain speed proves much more difficult, we must rethink our approach. To find a direction for improvement, let’s seek the truth about software history.

PERTINENT HISTORY

FORTRAN and COBOL

For those in the scientific realm, it took John Backus' IBM team much longer to design the Fortran language and build the compiler than he had anticipated. But it provided a huge leap in programmer productivity. After enough run-time speed comparisons, and the realization that the FORTRAN translators were getting better, it was apparent that the average programmer could not beat the FORTRAN compiler. Anyone in a competitive scientific environment quickly became an expert in FORTRAN. It provided the ability to quickly implement, test, and improve complex scientific transformations, such as complex matrix inversion. Its floating point structure and numeric array handling was tuned to arithmetic processors, making run-times fast. Comparisons still show that matrix inversion in FORTRAN is faster than in C++.

FORTRAN inspired a blue ribbon team to be formed to build an equivalent language for the business world. Measured by lines of code produced in the shortest amount of time, the greatest leap in software productivity was that of COBOL. It provided an approach to processing data that was extremely fast. It was designed to be very close to the machine, particularly with its hierarchical data structures that are easy to declare, manipulate, and most important - understand. It was easy to map compiler generated assembler code directly to the COBOL source. This is because memory is mapped as WYSIWYG in COBOL. It prompted IBM to access memory by the byte in the IBM 360.

Working with databases in COBOL allows the programmer to map memory WYSIWYG and redefine those areas of memory using different templates. This can provide dramatic improvements in run-time speed. Moreover, readability - or understandability - is extremely high. By following simple practices, one programmer can easily pick up where another left off. But COBOL's major flaw was that it did not support the scientific community. It had no floating point arithmetic and no scientific libraries - until some special versions became available - too little too late. This was not good for the academic environment.

Yet, even in the business programming market, COBOL was resisted. In the 1960's, when the financial industry in New York City was going through conversions to new IBM-360s, costs to upgrade to new assembler programs were extremely high. At that time, experienced programmers insisted that accounting applications could only be written efficiently in assembly language. What they were really concerned about were high school graduates moving into Manhattan and dramatically lowering the cost of building and supporting new software using COBOL.

Data processing managers had to fight to dislodge their company's software assets from the assembly language programmers, turning them over to a younger, less skilled workforce who could write code that everyone could understand - in COBOL. In that highly competitive economic environment, it was only a matter of time. The cost of software development and support plummeted with COBOL, and the leftover time and money was spent developing more sophisticated applications. So where has it gone?

C-Based Languages

To better understand the foundation of current approaches to software, it is helpful to review history as compiled from many sources as well as personal experience by Don Anselmo, former Director of the Computer Development Laboratory at AT&T Bell Laboratories, see for example, [AN2]. In the late 1960's Bell Labs withdrew people assigned to the failing MULTICS project, a joint effort with MIT and GE to produce a timesharing environment. These people were left without a demanding project or a convenient environment to share their work.

One of these people, Ken Thompson, wrote a simulation of a shared file system, and also a game "Space Travel," in FORTRAN on a GE 635 computer. However, it was difficult to control the space ship on the GE machine, and expensive to run. Thompson found a little-used PDP-7 that provided a good graphic display, but it's environment for porting the software made it difficult to work with. So he set about to write a FORTRAN compiler for the machine - but quickly abandoned that effort as too ambitious.

What he produced instead was a compiler for a language he called B - a revised version of the Basic Combined Programming Language (BCPL) from the UK. His principal concerns were to provide a Spartan syntax to keep the compiler simple to write, and also keep the compiler small to fit into the PDP-7's tiny memory. The compiler produced interpreted code and was inherently slow, but it served its purpose in porting the game. As described by Anselmo, Dennis Ritchie's modifications to make it run faster for the newer PDP-11 (it had a 24K byte memory) did not change that approach. It was then called NB for "New B", and finally C.

At this point there was no corporate objective or plan to develop a product. This was a project intended to create an environment in which a small group of researchers could share their work. Years later, another Bell Labs researcher, Bjarne Stroustrup, made improvements to C to provide for Object Oriented Programming. This language was called C++. This movement has grown to include Java, C#, and a host of other C-based languages.

The Rise Of OOP

Using C++ and OOP, programmers moved further from the machine, hiding data behind layers of abstraction. This made it slow as well as hard to understand. In the 1980s, Moore's Law came into full force and CPU speeds were doubling every 18 months. This led to (Niklaus) Wirth's Law:

"Software gets slower faster than hardware gets faster."

In the late 1980s, history was being repeated when the New York financial community started moving to UNIX platforms to save on the costs of mainframes. Moving from COBOL to C++, transaction processing times grew to be huge. By 1994, the market for mainframes came back to life. Marketeters credited this to what they called the "UNIX After Market" - after buyers realized that the cost to port their software to UNIX and C-based languages was greater than the hardware savings gained from inexpensive platforms. And it was not just the cost of the port, or of building new applications. It was the loss in transaction processing speed and the cost of supporting the software over the long term.

Apart from the OO concept, there are many reasons why C-based languages are difficult to work with. In his book "Expert C Programming - Deep C Secrets," [VDL], van der Linden describes problems in the use of C and C++ in a production environment at SUN, questioning the burden of translation. *Should it be on the programmer, or on the language translator?*

"C's declaration syntax is trivial for a compiler (or a compiler-writer) to process, but hard for the average programmer. Language designers are only human, and mistakes will be made. ... Ambiguity is a very undesirable property of a programming language grammar, as it significantly complicates the job of a compiler writer. But the syntax of C declarations is a truly horrible mess that permeates the *use* of the entire language. It's no exaggeration to say that C is significantly and needlessly complicated because of the awkward manner of combining types."

Neither sound economics nor good science can be cited as the justification for this movement. When engineers think of comparisons, they think of benchmarks that produce clear-cut economic measures. But the software field appears to be short on both repeatable tests and benchmarks. Today, programmers still question why matrix inversion routines run faster in FORTRAN than in C++. In fact, most programmers are shocked when told the real history of C. So what were the underlying driving forces at work to make C-based languages the programmers' choice?

First, apparently driven by job security, programmers want to feel that their skills in a language acquired over years is a vital asset, and resist writing code that is easily understood by others. Second, the academic community accepts the esoteric nature of C++ and OOP. Third, throughout the 1980s under CEO Bob Allen, AT&T spent billions marketing UNIX to compete with IBM for the computer market. C and C++ went along for the ride.

SOFTWARE PRODUCTIVITY ASSESSMENTS

Since "The Emperor With No Clothes," [LE], there have been an increasing number of papers disclosing the facts about decreasing software productivity and calling for a scientific approach to software engineering. Prior to that paper, the Standish Group published a report on the software industry, [SG1]. This report substantiated productivity statistics that were published independently by Cave, [CA1]. This was followed by Anselmo on productivity in the software industry, [AN1]. The Anselmo paper analyzed factors affecting software productivity as well as ways to measure it. Based upon emails received, it clearly polarized people into two groups: those preserving conventional wisdom, and those questioning it.

When the Standish Group published its follow on reports in 2003, [SG2] and 2004, [SG3], they served to substantiate that the software industry's productivity trend was still headed downhill. More importantly, denials by naysayers - that the negative productivity numbers were caused by the growth of project sizes - were put to rest. Standish statistics showed that management was cutting project size and complexity to reduce the historically high risks of building software. Not only were project sizes getting smaller, but stated requirements were being reduced as deadlines were missed. Software is one of the few industries in the U.S. with declining productivity. Based upon available measures starting in 1990 it has the worst track record of any - by a wide margin!

In early 2004, Jesse Poore published a Tale Of Three Disciplines, [PO], saying that

“Circuit engineering has done the job right . . .”,
and that compared to software,

“Computer-Aided Design, engineering and manufacturing tools were at the industry’s heart, making abstraction practical and enforcing an affinity of theory, practice, and tools.” . . . “Today, the complexity of circuit engineering matches or exceeds that of any other field.”

In June 2004, Ranum Marcus told how the approach to building software was the root cause of problems with security, [RA], saying that

“the software problem has gotten so bad that the hardware guys are trying to solve it too.”

He went on to say that

“it just seems backward to me that we are trying to solve what is fundamentally a software problem using hardware.”

In September 2004, Groth, [GR], produced a paper regarding the statistics of software productivity, extending the trend described by Cave, and further corroborating the Standish reports. Then in September 2005, Microsoft made the front page of the Wall Street Journal with the biggest software problem of all time, [GU]. This story was further explained by Cusumano, [CU], where he described “the chaotic, spaghetti architecture of Windows” as being no architecture at all. But what does “architecture” mean relative to software?

In September 2006, we hear more about the “Hard Data” describing software productivity from Phillip Armour in Communications of the ACM, [AM], where he solicited inputs on the cause for the decline. This is further supported by Broy in IEEE Computer, [BR]. Based upon his experience, Broy questions the Standish Group’s numbers as possibly too rosey. More importantly, he is in line with Poore on the issue of CAD systems, including discrete event systems theory, modeling, and simulation to support software development.

WHERE DO WE GO FROM HERE?

Today, a new era is emerging that may force us to rethink the approach to building software. We are faced with the realization that semi-conductor expert Jim Meindl’s prophecy was accurate: the Moore’s curve for speed has flattened, [MEI]. As Herb Sutter from Microsoft puts it, “The free lunch is over, ...”, [SU1]. The hardware solution concealing the software problem has just been removed. Instead, we are faced with programming parallel processors if we want to continue providing more functional capability without significant loss in speed.

According to Sutter, programming for two or more processors presents a major problem “because concurrent programming is hard.” What is more obvious is that programming approaches that make software slower on single processor platforms are going to make things *much slower much faster* on a parallel processor (an exponential form of Wirth’s Law). Piling layers of abstraction on top of the existing layers will move us further from the machine faster. So where do we go from here?

The papers by Poore and Broy suggest that solutions may be developed based upon circuit design and CAD tools, both of which require models and simulations. In the case of circuit design, the theory existed to create the models and test the designs. But we must separate the measures of merit for the end design from those for the tools. For example, the end design should support high productivity both in initial development, and more importantly in support for upgrades. The tools used in development must provide for measures of these properties to ensure productivity. If *understandability* and *independence* are the properties that spawn high productivity, then the tools for building and supporting software must ensure these properties.

A new paper by Cave and Wassmer, [CA2], shows how the application of engineering principles can head the software industry in a scientific direction. It defines software architecture in a manner consistent with hardware architecture and describes a CAD system to visualize architectures so that module independence is apparent by inspection. These are the keys that (1) make it easy to use parallel processors; (2) allow linear scaling of complexity; and (3) greatly improve run-time speed and productivity on single as well as parallel processors. This is an example of how we can help the software industry to head in directions that can capitalize on the opportunities for great leaps of growth.

REFERENCES

- [AM] Armour, Phillip G., *Software: Hard Data*, Communications of the ACM, Vol. 49. No.9, Sept. 2006.
- [AN1] Anselmo, Donald and Henry Ledgard, *Measuring Productivity in The Software Industry*, Communications of the ACM, Vol. 46. No.11, Nov 2003.
- [AN2] Anselmo, Donald, *Why Hasn't Software Productivity Improved?*, Software Summit, Washington, D.C., May 2004.
- [BR] Broy, Manfred, *The "Grand Challenge" in Informatics: Engineering Software-Intensive Systems*, Computer, IEEE Computer Society, Oct. 2006.
- [CA1] Cave, W., *Software Survivors*, Software Developer & Publisher, W. Cave, July/Aug1996.
- [CA2] Cave, W. and Wassmer, R, *Getting Closer to the Machine*, Visual Software International, Spring Lake, NJ, Jan. 2007.
- [CA3] Cave, W. and Ledgard H, *The Software Survivors*, visisoft.us/PDF_Files/TheSoftwareSurvivors.pdf, December 2006.
- [CU] Cusumano, Michael A., "What Road Ahead for Microsoft and Windows?," Communications of the ACM, July 2006, pg 21-23.
- [GR] Groth, R., *Is the Software Industry's Productivity Declining?*, IEEE Software, Nov/Dec 2004.
- [GU] Guth, Robert A., "Battling Google, Microsoft Changes How It Builds Software," The Wall Street Journal, Sept 23, 2005, page 1, column 5.
- [LE] Ledgard, Henry F., *The Emperor with No Clothes*, Communications of the ACM, Vol. 44, No. 10, Oct 2001.
- [MEI] Meindl, James D., Keynote Address, 1997 Hot Chips IX Symposium, Stanford Un., Palo Alto, CA.

- [PO] *A Tale of Three Disciplines and a Revolution*, Jesse H. Poore, IEEE Computer Society, Jan 2004.
- [RA] Ranum, Marcus J., *SECURITY - The root of the problem*, ACM QUEUE, June 2004.
- [SG1] "Chaos, Charting the Seas of Information Technology," The Standish Group International Inc. Report, Dennis, MA, 1995.
- [SG2] "Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved by 50%," Press Release, The Standish Group International Inc., West Yarmouth, MA, 2003.
<http://www.standishgroup.com/press/article.php?id=2>
- [SG3] "Standish: Project Success Rates Improved Over 10 Years" Software Magazine, January 2004.
<http://www.softwaremag.com/L.cfm?Doc=newsletter/2004-01-15/Standish>
- [SR] Strassmann, Paul, "From a Craft to an Industry," Ada Symposium, George Mason University, 1992.
- [SU1] Sutter, Herb, The Free Lunch Is Over: A Fundamental Turn Toward Concurrency In Software, Dr. Dobbs' Journal, 30(3), March 2005.
- [SU2] Sutter, Herb, and J. Larus, Software and the Concurrency Revolution, ACM Queue, vol. 3, no.7, September 2005.
- [VDL] van der Linden, P, *Expert C Programming - Deep C Secrets*, SunSoft Press - Prentice Hall, Englewood Cliffs, NJ, 1994.