# Software Education for Changing Computing Technology

**Benjamin Cooper[1] and Steve E. Watkins[2]**
**[1] CMC Technologies Limited LLC and [2]Missouri University of Science and Technology**

## Abstract

Software education has been dominated by procedural-based programming languages such as BASIC, FORTRAN and C, and before that, the assembly languages. The primary reason that this methodology has held such sway in education was that it allowed quick action for the first major users of computers. This approach was the most straight-forward means of utilizing hardware that, over the last 60 years, has gotten faster and more complex through smaller and more densely packed elements. However, traditional advances as described by Moore's law are now reaching both physical and economic limits. Pure object-oriented programming approaches offer benefits for hardware that is highly parallel and that is of non-traditional design. This work describes the evolution of computational technology, explores features of pure object-oriented languages such as Squeak Smalltalk, and discusses proactive curricula options.

## Introduction

Programming literacy and proficiency are key curricula concerns whether as an independent specialty or as a fundamental component of other engineering specialties. Software education typically emphasized procedural-based programming languages. Such programming approaches, which use sequential calculations and simple decision-making processes, are the traditional option, but they are not the only possible methodology. Alternative approaches have been explored with advanced features. Also, software choices are not unrelated to the hardware. Improvements in computational capabilities require both hardware and software considerations.

Students that are educated today must be prepared for the next generation of digital technologies. Traditional advances that rely on smaller and more densely-packed elements is reaching physical and economic limits. Important curricula questions are "what software education is the best preparation for these new technologies?" and "will engineering education be proactive or reactive to these important developments?" Computing hardware is changing. Future advances cannot follow the traditional path of faster and more densely packed devices. Economic limits, driven by physical limits, have been reached. Next generation digital technologies will incorporate more parallelism and may exploit new computing approaches.

This paper proposes proactive curricula options that build upon a pure object-oriented programming methodology. In particular, the concepts of encapsulation and message passing have features that will complement parallel hardware structures and new digital technologies. The evolution of computing hardware and procedural-based software is described. Object-oriented languages are presented as timely alternatives. Squeak Smalltalk is discussed as one such language and proactive curricula options are proposed for engineering education.

## Overview of Computing Technology

The computer industry is entering into an era of profound change. Its most powerful technique for production is reaching inherent limits. The opportunities to expand photolithography below feature sizes of 28nm are few, and very costly. As a consequence, the industry is beginning to perceive an uncertainty about its future. How can the industry avoid stagnation if the tried-and-true methods of the last half century are no longer able to maintain progress? This situation, however, is not the first time in science that a paradigm has run its course. In other fields, researchers have followed paths of inquiry that have come to an end. But history also shows that stagnation is not a necessity. In fact, many times when one path in science and technology comes to an end another trail is found, generally arising from some obscure perspective. If a new trail can be found for digital technology, it could lead to a new ways of doing things. What the industry could be in ten years from now could be very different from today.

### *A. The Past*

To understand the possibilities for the future, one must have some conception of the past. Fortunately within this subject, we have one advantage when compared to many others. The history of computing hardware and software is sharp and well defined [1]. An important first 'physical' manifestation of software, as applied to electronic digital computers, occurred when ENIAC went live in 1946. Prior to the ENIAC, there were other electrically based “computing” devices, but they can be characterized as calculators. They were machines that ran a straight sequence of instructions to completion, then stopped. But with the ENIAC, digital science took a major step towards realizing Turing's magnificent concepts of computing. With this machine, more than one path was open to the calculating process; a decision could be made based upon the memory of a previous state. Procedural programming arose from the merger of simple decision making capability of the conditional instruction with the sequential features of simple calculators.

But as with all first advances in science, the ENIAC was built when costs were high and performance limited. Consequently, the architecture used by John Mauchly and M. Presper Echkert was of the simplest type. And it was upon this base that the first evolution of computers developed. For most of the era of mainframes, the programming methodology was procedural. Only towards the end of this era did a few exceptions arise. They emerged from two sources. One involved hardware. In the last days of mainframes, the designers of a number of machines moved away from the simple computer architecture of Mauchly and Echkert. This facilitated changes in the way programing was done. The second rumbling of change came from brilliant insights done by several software pioneers. Ivan Edward Sutherland pushed the bounds with the way he created Sketchpad. Then Ole-Johan Dahl and Kristen Nygaard put forth a number of versions of Simula that challenged the standard methodology. Sutherland was recognized for his work with the 1988 Turing Award from the Association for Computing Machinery (ACM). Dahl and Nygaard were recognized jointly with the 2001 ACM Turing Award.

Then in the 1970s we see the full push towards making processors by photolithography. Like the first mainframes, costs were high and capacities limited. Consequently, the early developers of the microprocessor used the same basic architecture as Mauchly and Echkert. But a profound difference existed between the first and second evolutions of electronic computing. The manufacturing capacities that gave rise to the microprocessor had vastly greater potential than that used in mainframes. Photolithographic manufacturing allowed the time between successive generations of product to be very short. In 1965, Gordon E. Moore made this observation for semiconductor product cycles since 1959 [2], but the industry expectation, known as Moore's law, has guided planning, development, and research ever since with high accuracy. The economic consequence has been that products, once started, developed through a rapid number of small accumulative steps. And once a manufacturer traveled some distance down this product development path, major design changes to a product–such as changing the overall architecture of a processor–became financially impossible. Hence, while the complexity of semiconductor chips and computing applications have dramatically increased, the basic architectural approaches have remained largely similar. After the first choice was made for a given product, it did not change at its fundaments throughout the 40 years in which photolithography evolved to its present advanced state.

This type of product evolution is now nearing its end. As molecular and wavelength constraints become more critical [3], the economic benefits for increasing device density and increasing chip performance as a function of manufacturing cost are no longer present. In particular, the mask costs have substantially increased. Figure 1 illustrates these decreasing economic benefits [4].

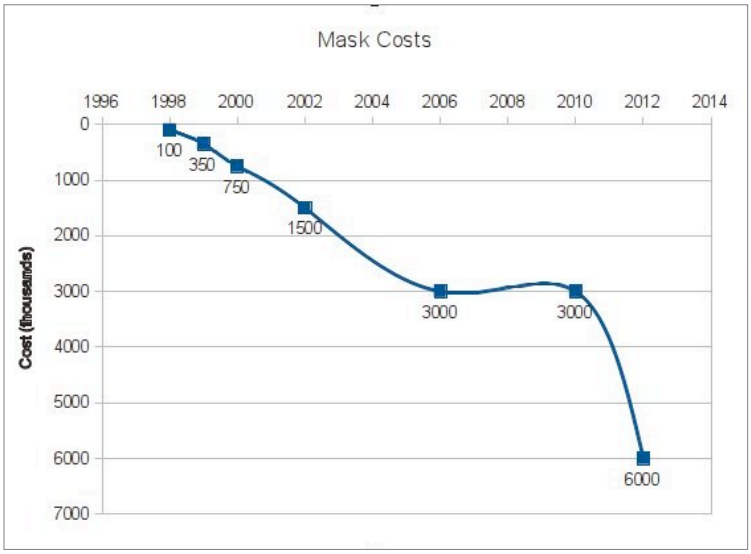


*Figure 1. The End of Moore's Law, Economics as Final Arbiter*

Shang-yi Chiang, the Senior Vice-President of Research and Development for TSMC, shared the following regarding the challenge of costs for lithography at a customer event in Japan during March 2010 [5]. In previous technology transitions, TSMC had been able to keep to a 15% increase in wafer processing costs.

> One challenge I'd like to share with you is the wafer cost. It's really steep. …
> (If you buy one EUV [extreme ultraviolet lithography] tool with a matched track, it will cost about $80M for just one tool. Because the EUV tools are relatively heavy, a special clamp is needed, attached to the fab ceiling to lift the EUV tool into place and for maintenance.)
> I was shocked to sign a purchase order a couple of weeks ago, a 1.9 million euros ($2.56M) purchase order for a clamp. This clamp is a custom-made clamp only for EUV. This tool is so heavy, no other tool can lift it up, and this custom-made clamp costs us 1.9 million euros, just to buy a clamp. It's really shocking.
> (If other costs increase like the lithography vector's,) Moore's Law will end very soon. Nobody will go to the new generation, … It's not because of physics, it's because of cost.

*B. Dominance of Procedural-Based Methodology*

When a processor is built upon a basic architecture and when the computing tasks are simple, but computationally intense, procedural-based programming is, without a doubt, the most direct way of doing things. This method gives the best result for speed. It also allows for the lowest cost of operation at the hardware level. And for most of the last half century of computing, the vast majority of the core tasks implemented in this technology were relatively simple.

Procedural-based computing has remained in the forefront of the software industry for four reasons. First, the core computing task were simple ones as described above. The second reason for this dominance has been short and predictable production cycle described by Moore’s law and driven by advances based on photolithography. During its massive grow phase, no one, no matter their level of genius, could overcome its profound influence. Some did try, especially in the 1980s, and they failed quickly. The development of the processor has followed a technical path that made procedural-based programming the 'natural choice'.

The third reason for this dominance relates to the second. Manufacturing through photolithography flooded the world with architecturally simple processors. This fostered a vast build-out of procedural-dominated programs. This includes all of the presently conceived object-oriented programs that have compromised upon the concepts of encapsulation and message passing, such as those written in C++ and Java. This vast overhang of programming achievements has, to date, cemented the overweening influence of the procedural methodology.

As for the last reason that procedural-based programing has remained in overall control, it is, without doubt, the most profound. It is also the one that is best overcome through education. The biggest reason that the simplest programming methodology has held sway is it is the one that

most easily fits with the innate thought processes of human beings. At the conscious level, human thought is essentially sequential. So it is not surprising to find that this methodology would be the first to be used by us to instruct our machines. Also, the need for procedural programming has lead to procedural programming methods being taught. Newly-minted engineers and scientist use the tools that with which they are most familiar.

*C. The Changing World of Software*

The scientific and technological forces that have brought the software industry to where it is now have already begun to change. Moore's Law is coming to an end for hardware. The juggernaut of photolithography is slowing. But upon closer examination, the truth is even more harsh for the software industry. Moore's Law's direct impact came to an end when a major trend in hardware development ended nearly a decade ago. That trend was where each new generation of processor ran faster than its predecessor. That is no longer true. Today, each new generation of processor runs at about the same speed of its recent predecessors. Figure 2 shows how the rate of increase in processor speeds is decreasing [4].

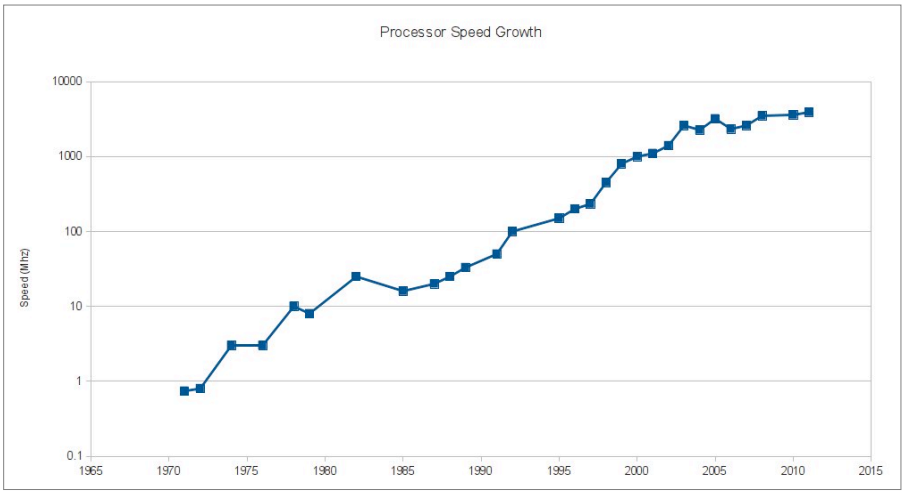


*Figure 2. Processor Speeds*

As a consequence of this difficulty, the computer hardware industry sought means to continue increasing computing performance. The notion that next generation computers are simply built around faster processors has been largely abandoned. A common direction has been toward parallelism, although many short-term solutions have been proposed [3]. Whether parallel structures or new architectures, Procedural-based programming approaches can no longer be consider the obvious answer. Relatively speaking, it was easy for the hardware industry to make fundamental changes; especially when they had no choice. Yet the situation is not nearly so easy for the software industry. Software approaches developed and ingrained over decades may be poor fits for the future technology.

*D. Limitations of Procedural Programming*

A critical issue in computing science is time. For a number of problems we can identify, we can, in the abstract, see the way to solve them by procedural programing.[1] If the time for solution is allowed to be elastic to the point of approaching infinity, these envisioned computational issues can be 'solved' using the simplest of computing methods. Even the 'traveling salesman' with ten thousand stops can, if the 'computing system' is allowed to run for a long-enough time, be resolved. The obvious problem with this procedural solution is that it may not only exceeds the time available in a implementation, but it may go beyond any practical scale.

Procedural-based programming is limited by time constraints, even more than it is with the developing economic issues with photolithography. Simple sequential solutions, at both the hardware and software levels, will need to be replaced with the approach of parallelism. If we do not break up the problems that we now face into 'small' pieces, then act upon them in parallel, we will not have enough time to even get an approximation to the answers we seek.

It must be said that time constraints has always been a problem in the industry. But fortunately in the early stages of its development, the most pressing initial problems–such as generating ballast tables–could be solved reasonably well by the procedural method. And as the industry began to grow, each new generation of machines made by the computing industry could be made to go faster. This was the first method of staying ahead of time issues.

But as stated above, this technological approach came to an end nearly a decade ago. Then, the hardware industry struck off into the direction of parallelism, as stated above. But the approach taken was not well grounded in theory or in long-term considerations. Rather, the new approach tried to preserve as much value in the present products as possible. The industry attempted, as much as possible, to remain as backwards compatible with the prior achievements with software. The problem with a backwards-compatible approach is that the software industry has, over the last four decades, focused its efforts on extracting as much performance from sequential programming as possible. Most of these achievements, unfortunately, are incompatible with parallel processing[2]; especially when efforts are made to enter deeply into parallel processing, where the main issue is to break the problem into the optimum number of pieces to solve simultaneously.[3]

---

1 The one major issue that makes this particular discussion even more difficult is Turing's concept of the halting problem. If we extend this discussion to a greater scope of problems than those that we have a high degree of confidence will terminate, then the discussion must extend beyond parallelism and begin to incorporate the concepts of heuristic solutions. Also, we would need to delve into the ideas of approximation.

2 The two issues that must be handled very differently between an environment that runs sequentially and one that runs in parallel is data typing and 'the reuse of code'. In fact, the two environments demand the exact opposite approaches to both issues.

3 One of the main issues in parallel processing is how many pieces should a given problem be broken down to do parallel analysis. There are many factors that play an important role in answering this question. Two of the major issues is the power of each of the computational units and the speed and power of the interconnects between the units. A complex issue that will demand an ever growing analysis from the computational industry as it moves forwards.

**Software Programming**

If procedural-based programming is not the obvious choice, the next question is, what is a more mature or optimal approach to structuring computational environments? Often, a different perspective can be found in other seemly-unrelated fields. One of the greatest places to gain insights into complexity is biology [3]. However, seeking insight from biological systems is not new. In the 1960s, Alan Kay (another winner of the ACM Turing Award, 2003) once asked the simple question, "How can biology fill our world with systems that involve vast complexity and intricacy, of individual systems composed of hundreds of trillions of subcomponents working harmoniously together, and we cannot produce a few systems that, if we are lucky, have one millionth of that intricacy?"

Alan Key was a molecular biologist turned programmer who has been described as the "father of the personal computer" [6]. His answer from biology was simple: encapsulation combined with message passing. That is the heart of an alternative approach to building complex systems that involve the use of parallel action. It is the basis for the functioning of multicellular organisms and it has proven successful over the history of biology on Earth.

The biological model appeared four decades before its 'time'. The main progress toward this biological model of object-oriented programming was in a modified hybrid beast, so that it could stay compatible with the traditional programming paradigm. The juggernaut of photolithography and the first processors it produced were just too much. Other ways of doing things could not compete, especially if those other ways required 'painful' educational effort of those in the industry. And during this growth period, where the next generation computers were faster than their predecessors, it was always possible to extract more advantage from procedural-based programs. So during this time, there appeared no real need to leave behind traditional thinking and go through the pains of optimizing the programming approach.

Since the situation has changed, research and educational challenges are to address the issue of optimizing the programming approach. Procedural-based programming was dominant for historical and economic reasons, not because it was the optimal approach. Also, the compromises made in creating hybrid object-oriented-lite programming were not optimal. We would argue that early efforts at object-oriented programming were useful in that they incorporated core concepts, but they also incorporated aspects of procedural programming and tended to rely on intuition rather than a high-level inspirational model. [4] The reason for this was often to make best use of available hardware rather than best programming practices. These programming languages would include Sketchpad, Simula, and Lisp. What then is pure object-oriented programming?

4 We recognize that object-oriented programming is not universally recognized as an approach of choice, or even one suited for parallelism. In some case object-oriented programming is criticized for just the features that we are praising. This debate will certainly continue into the future.

*A. Pure Object-Oriented Programming and Hardware Considerations*

We maintain that the ideological connection between object-oriented programming and molecular biology is essential. The insight of Alan Kay is that biological systems have withstood the rigorous testing of an untold number of interactions for multicellular organisms. Here failure results in the most severe consequence of death. This connection gives a model with which to validate programming features. Core concepts from biology are encapsulation and message passing.

Encapsulation can be considered a construct that bundles data with the methods that operate on that data and where no outside methods can directly impact or affect the internal data. For those in the computing world, the term data has become well defined. Methods can be construed as short sequences of code that are called on in a similar way that functions are used in procedural-based programming. But in pure object orient programming, well articulated mechanisms are establish to assure what the short sequences of code can or cannot do with regards to the various sets of data within the system.

In the realm of biology, the primary identifier of encapsulation is the cell. The data, in the broadest sense, is the activated DNA, as well as some expressions of RNA. The methods generally consist of proteins acting as enzymes and that are activated and inactivated through various chemical energy mechanisms, one of the more common forms being phosphorylation and dephosphorylation done through the use of Adenosine-5'-triphosphate. But in recent work, RNA mechanisms are also turning out to carry out critical enzymatic function within the cell, and they may also be influenced from outside the cell. As such, these latter mechanisms would also constitute cellular methods

Message passing is the way one encapsulated object influences one or more encapsulated objects. In biology this is accomplished by one cell releasing into the interstitial space a small molecule that other cells can respond to through surface proteins; proteins that create chemical energy changes within the cell. In the realm of computer science, a communication network is created to establish a means by which one encapsulated object can send a message to one or more encapsulated objects. And when an encapsulated object receives a message, a corresponding method is triggered. In computer systems, the transmitting and receiving object may be one and the same. This communication network, along with the sending of messages, constitutes the message passing mechanism.

Such a model facilitates a parallel, hardware architecture. Furthermore, it is suited for co-design schemes for considering both hardware and software during the design process. In the current environment, powerful tools exist that could allow programmers have a strong voice in what was put into next generation products. First, vast chip manufacturing capacities are available in the world that anyone can access. Second, the design tools to create chips are widely available. They can be used by anyone with the desire to create their own products. This situation is reminiscent of the close interaction between hardware and software designers in the age of mainframe computers. The flexibility to optimize hardware and software together has never been greater.

*B. Software Education in Engineering Curricula*

Computer programming literacy is a fundamental component of technical education. Computer science is a well-established field in its own right. Computer engineering degree programs have grown and they have strong programming requirements. Other engineering fields typically require some course or familiarity in programming. Many STEM-based pre-college programs have a strong programming component, e.g. LEGO Mindstorm Robotics competitions. However, unless a student specializes in computer programming, the exposure may be limited to a single language or limited languages. For instance, in electrical engineering at Missouri University of Science and Technology, the programming language of choice is C++ to fulfill the requirement. The selected language is often driven by faculty preference and immediate industry needs. New graduates tend to use the tools they know.

Structured education, as the primary means of transmitting knowledge to the next generation, can, as knowledge advances, progress upon one of two paths. The first possibility is that of being reactive, of responding to major changes only after those changes affect technology in a profound way. Then, there is the proactive choice. The path taken when significant changes are arising, but before those changes have widening consequences. We are entering an era of change in both hardware and software. Providing options for science and engineering students to explore the object-oriented programming is the proactive choice.

## Curricula Development

*A. An Example Language: Squeak Smalltalk*

Smalltalk has many versions. Squeak Smalltalk is open-source language based on object-oriented methodology [7]. It is the current version of a programming language originally developed as XEROX PARC in the 1970s. It has been successfully applied to many areas in education, recreation, and industry. For instance, it is a language of choice for the One-Laptop-Child initiative and has been shown to facilitate learning at early ages [8]. A commercial version of Smalltalk has also played a powerful, but quiet role in banking and finance, both as direct means of analysis and building upon a powerful object-oriented database constructed. The features of Squeak Smalltalk are compared to those of C++ and Java in Figure 3.

In addition to the utility of dual concepts of encapsulation and message sending, the object-oriented approach facilitates the reinvention of each successive generation or version of the programming language. Alan Kay stated at OOPSLA ‘97: “I think one of the things we liked the most about Smalltalk was not what it could do, but the fact that it was such a good vehicle for bootstrapping the next set of ideas we had about how to do systems building.” He also went on to say: “The thing I am most proud of about Smalltalk, pretty much the only thing, from my standpoint, that I am proud of, is that it has been so good at getting rid of previous versions of itself.”

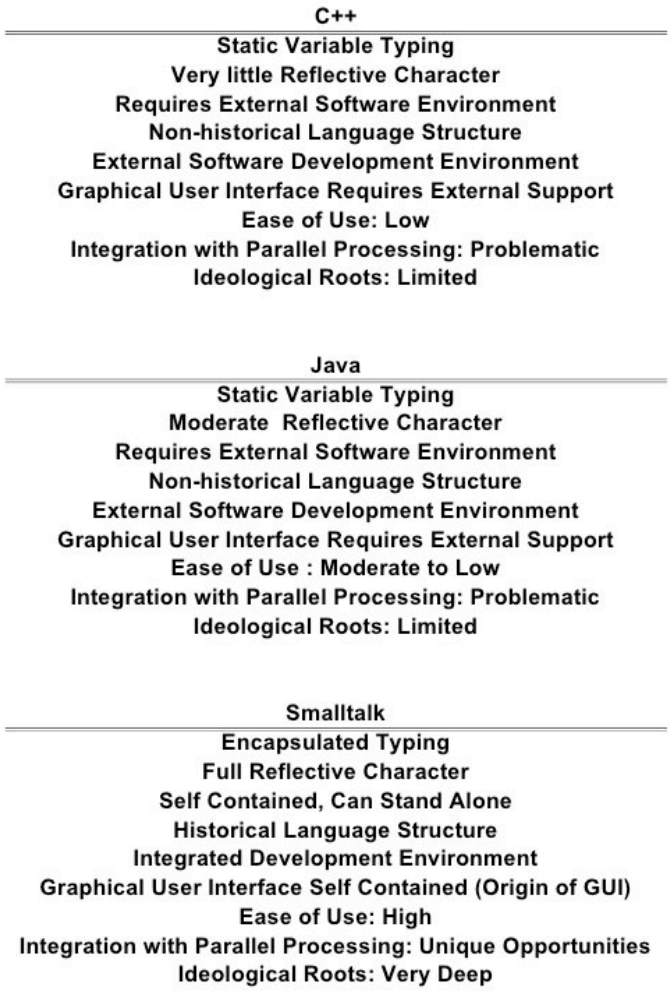


*Figure 3. Features of C++, Java, and Smalltalk*

## *B. Curricula Discussion*

Changes in computing technologies will have a profound effect on the course of science and engineering in the coming years. Software options as a supporting resource for various science and engineering fields and the programming languages that are required and offered through educational programs should be a topic for discussion in educational circles. The comparison of programming languages is not trivial [9,10]. Beyond pure intrinsic characteristics of a language, a programs performance is influenced by the programmer [9]. Extrinsic factors related to institutional support, industry support, etc. must be considered [10]. In addition, business considerations may play a dominant role. Languages that are implemented as compiled code provide considerable protection of the intellectual property in a given program.

Traditional programming languages and procedural programming approaches should not be default choices. Advances in computing technologies and innovation based on computing resources may be better served by alternatives. Proactive efforts by science and engineering educators can include:

- Research studies comparing software features,
- Research studies of hardware/software co-design,
- Debate on optimized programming (e.g. object–oriented programming verses other approaches),
- Re-examination of required programming courses,
- Research into programming pedagogy,
- Programming content in pre-college curricula, and
- Electives or options for non-traditional programming instruction.

We believe that object-oriented programming choices should be part of this effort. Fortunately, such options can serve the current niche implementation areas such as database management for banking and finance. The ease of learning programming languages such as Squeak Smalltalk can assist in incorporating related content into project coursework and can aid in programming instruction for younger students. Also, open-source versions of object-oriented languages such as Squeak Smalltalk are available.

Existing research and curricula efforts with object-oriented programming are important resources for other educators. For instance, Dr. Gene A. Tagliarini, Computer Science at the University of North Carolina Wilmington, uses the Squeak Etoys language to teach an Introduction to Computer Programming (CSC 112) course for non-computer-science majors [11]. Research work by the Computer Science Department at Duke University is based on the Open Cobalt language which is based upon Squeak [12]. Dr. Mark Guzdial, College of Computing at Georgia Institute of Technology, teaches Introduction to Media Computation (CS1315) and Representing Structure and Behavior (CS 1316), is pursuing Dynabook-type [6] research, and has successful pre-college outreach called Georgia Computes [13,14].

**Discussion**

Computer technology evolution as described by Moore's law is at an end. Many approaches for continuing to expand the performance of computing have been proposed, especially approaches using parallelism. The change in computing hardware will have an impact on software. Many, including the authors, believe that the dominance of procedural programming is also at its end. Optimization of programming approaches may be just as importance as changes in hardware. In particular, the approach to parallelism as now being undertaken by industry does not offer a long-term alternative. Major efforts in both the hardware and software fronts are imminent.

What is a more mature or optimal approach to structuring computational environments? Procedural-based methodology has its place. It is the most basic level of interfacing among hardware components. However, other methodologies may grow in importance and application. We propose object-oriented programming approaches as being well suited for parallel hardware structures and new digital technologies. These approaches are based on a biological model and

are built on encapsulation and message passing. The biological model as described by Alan Kay gives an ideological basis. These features support parallelism and eliminate many time constraints in current programming tasks. A vast array of hardware tool and extensive opportunities for affordable chip design are available. Co-design of hardware and software systems have great potential for education as well as industry application.

Engineering education should address changing computing technology. Procedural-base programming is no longer the obvious choice. Object-oriented software such as Squeak Smalltalk offers many benefits. Students should have options to learn and apply such programming methodology.

## Bibliography


1. Computer History Museum, *Revolution, The First 2000 Years of Computing*, Computer History Museum, Mountain View, CA, 2010.
2. C. A. Mack, "Fifty Years of Moore's Law," *IEEE Transactions on Semiconductor Manufacturing,* 24(2), 202-207, (2011).
3. R. K. Cavin, "Science and Engineering Beyond Moore's Law," *Proceedings of the IEEE,* 100(Special Centennial), 1720-1749, (2012).
4. "Chip Design,"(2012) Available WWW: http://chipdesignmag.com/.
5. "TSMC Facing EUV, Wafer Cost Challenges," Institute of Microelectronics, Chinese Academy of Sciences, (2012). Available WWW: http://english.ime.cas.cn/ns/es/201003/t20100302_51092.html.
6. S. B. Barnes, "Alan Kay: Transforming the Computer into a Communication Medium," *IEEE Annuals of the History of Computing,* 29(2), 18-30, (2007).
7. "Squeak Smalltalk," (2012). Available WWW: www.squeak.org.
8. K. N. Rodhouse, B. Cooper, and S. E. Watkins, Programming for Pre-College Education using Squeak Smalltalk," *Computers in Education Journal*, 21(2), 101-111, (2011).
9. L. Prechelt, "An Empirical Comparison of Seven Programming Languages," *Computer*, 33(10), 23-29, (2000).
10. Y. Chen, "An Empirical Study of Programming Language Trends," *IEEE Software*, 22(3), 72-179, (2005).
11. "Computer Science Department," University of North Carolina Wilmington, (2012). Available WWW: http://uncw.edu/csc/.
12. "Open Cobalt," Computer Science Department, Duke University (2012). Available WWW: http://www.opencobalt.org/.
13. "Contextualized Support for Learning Laboratory," College of Computing, Georgia Tech (2012). Available WWW: http://home.cc.gatech.edu/csl/.
14. "Georgia Computes," College of Computing, Georgia Tech (2012). Available WWW: http://gacomputes.cc.gatech.edu/.


## Biographical Information


BENJAMIN COOPER is CTO/Managing Partner of Savant LLC. He is an entrepreneur with experience in several start-up companies. He attended Emery University and the University of California, San Diego. Contact: Benjamin.cooper.2@gmail.com.

DR. STEVE E. WATKINS is Professor of Electrical and Computer Engineering at Missouri University of Science and Technology, formerly the University of Missouri-Rolla. His interests include educational innovation. He is active in IEEE, SPIE, and ASEE including service as the 2009 Midwest Section Chair. His Ph.D. is from the University of Texas at Austin (1989). Contact: steve.e.watkins@ieee.org