CrossTalk and Software—Past, Present and Future

A Twenty-Five Year Perspective

David A. Cook, Ph.D., Stephen F. Austin State University

Abstract. Since its initial issue, CrossTalk has helped guide software development throughout the DoD. As CrossTalk celebrates its 25th anniversary, it is educational to see how much software has changed and evolved over the lifetime of CrossTalk—and where the future might be leading us. This article discusses several of the forces that have shaped software and developmental languages over the last 25 years and also tries to see where the future will be taking us.

In the Beginning: Pre-CrossTalk

Although 25 years is a short span of time, it is actually a very long time in terms of software evolution. Twenty-Five years is over one-third the entire life span of computers—after all, the ENIAC only dates from 1946 [1].

One could also argue that some of the most important changes in computers and software occurred in the last 25 years—after all, the commercialization of the Internet did not begin until the mid 1990s. Standardization of TCP/IP itself did not begin until the 1980s [2]. The replacement of the large mainframe computers with desktop "microcomputers" did not happen until the late 1980s. Of course, lots of software development was accomplished prior to the existence of CrossTalk. In the early days of software development, however, it was normal for developers to need intimate knowledge of the target hardware.

Back in the 1950s and even into the 1960s, machine code was used for many applications—and the only tools available were assemblers. Even when working with assembly language (which was much simpler to understand than machine code), developers had to have extensive knowledge of the hardware that the final software would be deployed upon. The tools that were available during these early days were relatively simplistic. The developer was closely tied to not only a machine, but occasionally tied to a particular model and configuration. The interface between the developer and the hardware was direct—and hard to learn and master. The developer had to understand not just the problem space, but also had to be a master of the hardware. At best, an assembler abstracted away some of the hardware, but not all. Developers still were tied to hardware—and had to understand it to develop any code [3].

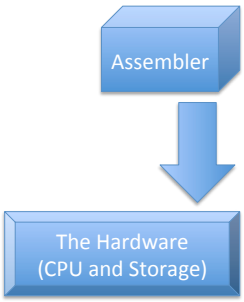

Figure 1 – Adding a tool to abstract away part of the actual computer

The Era of CrossTalk–The Early Days

The Quantity of Programming Languages

Twenty-five years ago, compilers and languages proliferated. There were many reasons for the creation of a new programming language [4], and the result was that by the 1980s, more than 2,000 programming languages existed [5]. Often companies or projects created a new language because their proposed software needed a combination of features not found in an existing language. Because the machines (and storage) of the time were limited, trying to add additional features to a language that already had features they might not need would simply increase compile time. Back in the 1980s it was not unusual for compile time to run to minutes per line! Adding new features to existing languages simply made compile time worse. It was more attractive to start fresh—and develop a language that had only the exact features needed for a project.

By the time CrossTalk came along, it was reasonably well recognized in the DoD that a minimal set of languages would make software maintenance easier but allow more transfer of knowledge and reuse of code throughout the DoD. Simply put, it is not cost efficient to maintain systems in thousands of languages, nor is it wise to have a software development workforce that is segmented by knowledge of so many niche languages.

While it was recognized that such a minimal set of languages needed to include some legacy languages (JOVIAL, CMS, Fortran, COBOL), the DoD also wanted to develop a language that it hoped would meet everybody's programming needs. During the infancy of CrossTalk, Ada was developed and heavily promoted by the DoD as a language that would unify software development needs. For numerous reasons (many political), Ada never became the huge success that the DoD envisioned. Commercial languages that dominate today's software development market include Java and C (and the descendants of C, such as C++ and C#). To understand the forces driving language design and language selection, it helps to examine a programming language from the perspective of what it provides to the developer.

The Quality of Programming Languages

Early high-level languages provided "machine transparency" to the developer. Without having to know and master such concepts as word size, memory size, how many registers were available, etc., the developer could spend less time concentrating on "what platform the solution will be implemented on" and more time on just understanding the problem. A "good" programming language let the programmer focus on the problem, rather than the hardware—but at the same time, provided enough features to permit the majority of general-purpose software tasks to be easily accomplished.

The earliest compilers were adequate for basic generalized programming needs. They provided the developer with a way to abstract themselves yet one step further away from the machine. In essence, the compilers were a tool that provided input to another tool (the assembler), which, in turn interfaced with the hardware.

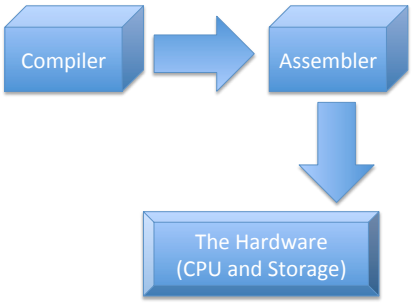

Figure 2 – Adding one more level between the developer and the hardware

One of the driving forces behind software development has been, oddly enough, a hardware force—Moore's Law [6]. Moore's Law (Gordon Moore was one of the co-founders of Intel) was that the number of components of an integrated circuit doubles about every two years. The law (more of an observation) has proven to be uncannily accurate over the last 50 years. And the law has been expanded to cover the capabilities of many digital electronic devices that are strongly linked to Moore's Law: processing speed, memory capacity, disk capacity, and even the number and size of pixels. Because this law says that everything doubles every two years, then the capacity of computers (in terms of speed, memory, and storage) is exponential. From Moore's Law comes what I refer to as Cook's Observation of Unwanted Space—every CPU cycle and byte of storage will eventually become used. Back in the 1960s, the Titan missile used less than 2,000 lines of code. The F-35 Joint Strike Fighter uses around 25 million [7].

The Recent Past and the Present

Software of Today

How is it possible that computer speed, memory, and storage are doubling every two years, but we are continuing to use mainstream languages (such as Java and C++) to develop modern software systems that demand more and more capabilities? We manage to accomplish this by continually updating just the language, but by continuing to create and update extensive libraries and templates to assist us with coding. Granted, we continue to update modern languages (Java is up to Version 7, Update 15, while C++ is now at C++11, with revisions planned ahead for C++14 and C++17). These changes, however, are evolutionary, not revolutionary. It is pretty much a guarantee that C++ code that runs today will still run with the latest version of the compiler in 2017. And no language is currently on the horizon to displace either Java or C++ from their dominant positions.

Instead, rather than develop newer and newer languages, we now extend our current software capabilities by writing support libraries and "importable" code (templates, generics) to extend the capabilities of our languages. We are adding additional tools (libraries) to support the compiler (another tool) to eventually/probably be converted to assembly language and then executed on the target machine.

Back in the 1970s and 1980s, the lack of the Internet made it difficult to share languages. Languages came into existence, were used for select projects, and disappeared in relative isolation. Languages tended to belong to a single project, or a single company. In the present, however, we can easily share languages and libraries. And because so many needed language features are common throughout much of the development community, new ideas for language features are easily and quickly shared. We can easily add standardized features (typically by including a new library or adding features to existing libraries) to languages that are standardized. Our extensibility is now managed by a mutually agreed upon standardization of languages. Rather than writing a new language, we have enough spare capability to add the libraries and compiler features to let the existing languages evolve.

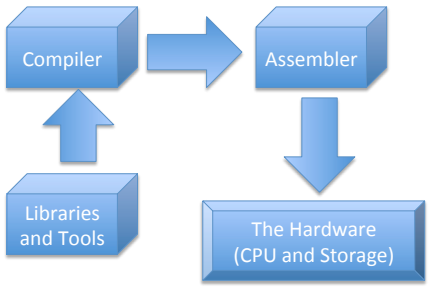

Figure 3 – using Libraries and Tools to further distance developers from the hardware

Coupled with the ability to "expand" languages through the use of libraries, we also have several other forces shaping how we develop software. These factors will have a tremendous effect on the software development of tomorrow.

The Near Future – Forces That Will Affect How We Develop Software

Distributed Computing

In the 1990s, we viewed distributed processing and parallel processing as the wave of the future. While both predictions have somewhat become true, it is not in a way that we ever envisioned 25 years ago. When CrossTalk first started publishing, 20 to 30 pound laptops were about as "portable" as computers could be. Back in the 1960s, when Star Trek first debuted, Star Fleet ensigns walked around the ship carrying PADDs, or Portable Access Display Devices. These devices, which seemed to be portable computers with access to the "Computer" were obviously a pipe dream. Now, as ultra books, full-fledged and high-powered laptops, smart phones and tablets abound we "distribute" computing and require software that equally distributes tasks as necessary. Mainstream languages now have extensions or specialized frameworks to allow developing software that runs on multiple platforms (from the large to the small).

Storage Issues

In the near future, several trends are going to affect how we develop software. The first is data storage. In the 1960s and 70s, the storage medium of choice was (as any addict of late-night really old science fiction movies can tell you) magnetic tape (for large data storage) and punched card (for individual programs). By the 1980s, floppy disks (8", 5 ¼" and later 3 ½") had become the medium of choice for individuals, while disk

storage was the standard for large data stores. By the 1990s, individual developers were using CD and DVDs for storage. By the 2000s, most developers had embraced flash storage with capacities up to 32GB being common. In all of the above examples, the devices for individual storage were "personal" under the total control of the developer. Now, however, cloud storage is becoming the standard. It is possible to obtain totally free cloud storage ranging from 5GB to 50GB. The side effect of this easy to obtain and easy to use (and extremely portable) storage is that the possession and protection of code and individual data is no longer under the developer's control.

Security Issues

Even before 9/11, military applications were routinely developed with a high level of security in the actual developed application. The events of 9/11 made security an integral part of almost all DoD system and development processes. With distributed computing (using smart phones, laptops and tables) and the use of cloud storage, DoD applications require specialized and higher levels of security during development. They also require a language (and operating system and network) that permits the applications to run with a relatively high degree of security.

In the 1980s and 1990s, software was developed mostly onsite, and typically run from a dedicated (and protected) client. Now, however, software development, execution of the applications, and code and data security are no longer necessarily centralized. When you combine the potential for terrorism and the potential for catastrophic failure of storage, applications will require unprecedented levels of security and redundancy. This has not been primarily a software issue in the past (it was handled by the operating system, network, and even manual processes). However, as redundancy and security will become more and more of a requirement for all levels of software in the future, I expect to see many security features become part of mainstream programming languages.

Trend To Graphical Languages or Graphical Front-Ends

Since the early 1950s, we have tried to use graphical methods to capture requirements and develop systems. We have tried flowcharts, State Transition Diagrams, Data Flow Diagrams, and the Unified Modeling Language. All work to help, but none are full-fledged enough to actually capture a full set of requirements for a large-scale system and produce executable code. Some (such as UML) come close.

In some areas, there do exist graphical interfaces that can create a complete executable system. For example, in the field of Modeling and Simulation, the language Arena (among others) allows an experienced user to capture requirements, develop the model, and execute the simulation under a variety of constraints [8].

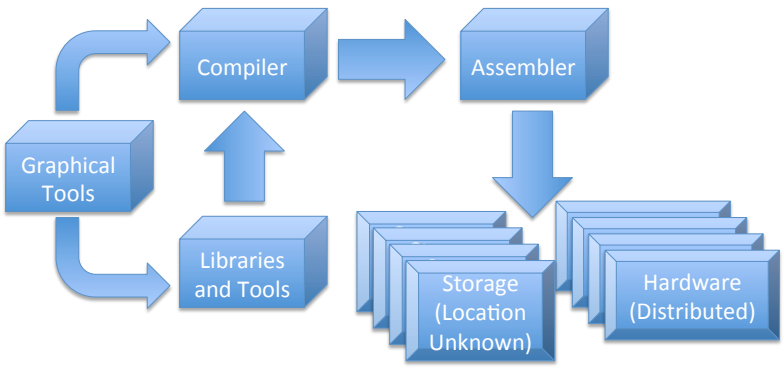

Figure 4 – Adding a Graphical Interface – even more removed from hardware, wherever it is, and wherever your storage is!

The Not-So-Near Future

Back in 1997, I was privileged to attend the ACM (Association of Computing Machinery) 50th Anniversary celebration, in San Jose. While there, a group of luminaries was present, and each was asked to briefly speak for 10 minutes or so on "What The Future Holds." I remember little about who spoke, or what they said, except for one speaker (whose name I cannot remember). He said, "10 years ago, we did not see the Internet coming, so who are we to predict the future?" I feel the same way. Things that we never envisioned as possible are now real. I can be standing in the middle of a cornfield in Nebraska, and given a decent 4G signal, have accessible to me almost all recorded history.

In the 1960s, when Star Trek had tablets disguised as PADDs, and cell phones and Bluetooth earpieces disguised as communicators, we could not comprehend a future with such wonderful devices. Now, I can wear a small device in my ear, tap it, and simply say, "Siri, please tell me the weather in London." I get results within seconds. The boundaries between normal life and computer usage are almost non-existent. Cars, appliances, even shoes are integrated in the ever-expanding computer-driven daily life.

I feel that software will continue to follow two separate paths—large-scale and non-traditional. Large-scale traditional software development (like much of the software developed within the DoD) will evolve slowly. Granted, I used Fortran in the 1960s, and now use C++, but the process is almost the same. Requirements, analysis, design, implementation, testing, maintenance—some things will probably not change for a long, long time. Niche software will come and go. A few new languages will be developed for specialized applications. It will be very difficult to create a new language that can overcome the developmental inertia that C++ and Java now hold. This language might continue to evolve (such as C# or Objective-C), but look at the staying power of Fortran. It was released commercially in 1957, and still maintains a strong "foot in the door" for many engineering applications. It appears than once a language becomes mainstream it remains a development tool for years and years to come.

Conclusions and Inescapable Facts

The average reader of CrossTalk is probably not the average developer of software. If you read CrossTalk, you probably work on large-scale or real-time systems. These systems are hard! We are always on the cutting edge of technology, trying to do what has never been done before.

I cannot say it any better than Fred Brooks said back in 1986, when he wrote the classic article, "No Silver Bullet—Essence and Accidents of Software Engineering [9]."

In it, he said, "I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure; but they are fuzz compared with the conceptual errors in most systems."

If this is true, building software will always be hard. There is inherently no silver bullet.

Let us consider the inherent properties of this irreducible essence of modern software systems: complexity, conformity, changeability, and invisibility.

Twenty five years later, software is still hard. Software is still complex; still has to conform to bizarre and antiquated interfaces; still requires constant maintenance and updating; and still is essentially invisible, in spite of the graphical tools and process we try to use. And this is not necessarily bad. Using my iPhone to connect to a microprocessor in my shoes so that I can track my daily aerobic exercise history should be invisible—in fact, I want it seamless and thought-free.

But still, how do we create and provide this seamless integration between computers and every facet of our life? How about the really large-scale integration—the aircraft, spaceships, and weapons of tomorrow? Brooks, in the Mythical Man Month anniversary edition (where both the original article and his article "No Silver Bullet Refired" can be found) brings forth that perhaps methodologies are the silver bullet. The more advanced and larger the eventual software application, the more important it will be to have a process to manage the inherent complexity, conformity, changeability and invisibility.

And, as far as I can clearly see, therein lies the future. Processes are important—because of the magnitude of the effort. As the effort gets bigger, the more we need to rely on a process to guide us to completion. Back in the 1980s, when CrossTalk started publication, our computer systems were not exactly small, but they were smaller. For the mid 2010s? Double the CPU speed about 10 times. Then, also double available memory and storage capacity about the same number of times. And now fill up the computer with enough software to consume every clock cycle and byte. It is too big to even comprehend, so you better have a serious process to make it all fit together because without a process to manage the complexity you are not going to be able to get anything to work. In fact, you probably would not even be able to gather enough requirements to start development.

Large-scale projects require large-scale processes, which require relatively strict adherence to process standards. The languages we use are just a supporting role in the software systems we create. Software of the future is a combination of languages, tools, libraries, and most importantly, a process for putting it all together.

As we reflect on CrossTalk's 25 years of publication, I think that I can confidently say that CrossTalk has covered the issues and trends that got us to where we are now. As a frequent contributor and reviewer, I can also say that CrossTalk is already preparing us for the future!

ABOUT THE AUTHOR

David Cook is Associate Professor of Computer Science at Stephen F. Austin State University. He served 23 years in the Air Force, teaching computer science and software engineering at both the USAF Academy and AFIT. He also worked as a consultant to the STSC for 16 years. His fields of interest are software engineering, software quality, and verification and validation of large-scale modeling and simulations. His Ph.D. (in computer science) is from Texas A&M. He has been a columnist and contributing author for CrossTalk for almost all of its 25 years of publication.

E-mail: cookda@sfasu.edu
Phone: 936-468-2508

REFERENCES

1. Bellis, Mary. The History of the ENIAC Computer. February 24, 2013. <http://inventors.about.com/od/estartinventions/a/Eniac.htm>.
2. Leiner, Barry et. al. "www.internetsociety.org." October 2012. Brief History of the Internet. February 18, 2013. <http://www.internetsociety.org/internet/what-internet/history-internet/brief-history-internet>.
3. Cook. "Evolution of Programming Languages and Why a Language is Not Enough to Solve Our Problems." Crosstalk, the Journal of Defense Software Engineering 12.12 (1999): 7-12.
4. Schorsch, Thomas and David Cook. "Evolutionary Trends in Programming Languages." CrossTalk, the Journal of Defense Software Engineering 16.2 (2003): 4-9.
5. Kinnersley, William. The Language List. 1991. February 13, 2013. <http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm>.
6. Shankland, Stephen. Moore's Law: the rule that really matters in tech. October 2012. February 20, 2013. <http://news.cnet.com/8301-11386_3-57526581-76/moores-law-the-rule-that-really-matters-in-tech/>.
7. Venlet, VADM David. "Selected Acquisiton Report F-35." Department of Defense, n.d.
8. Kelton, David W. et. al. Simualtion with Arena. New York: McGraw-Hill, 2003.
9. Brooks, Frederick. Mythical Man Month, Anniversary Edition. Boston: Addison-Wesley, 1995.