Evolutionary Trends of Programming Languages

Lt. Col. Thomas M. Schorsch
United States Air Force Academy

Dr. David A. Cook
Software Technology Support Center/Shim Enterprises, Inc.

Programming languages are the tools that allow communication between the computer and the developer. Far from being a static tool, programming languages evolve – they are created, constantly change, and frequently disappear over the course of their use. This article discusses the needs and forces that have shaped the evolution of programming languages, and discusses various evolutionary paths of programming languages in current use.

A programming language allows a developer to translate logical real-world actions into operations that can be performed on computer hardware. In effect, it is a way to translate concrete real-world desires into computer-world operations.

Programming languages advance by extending the number of operations programmers can perform without thinking about them – thus making it easier to say the things they want to say. In effect, these advances hide the complexity of what is going on underneath the hood and raise the level of abstraction that programmers think about when they program.

If a programmer wants to say something to the computer, and he/she finds that the current language has difficulty in saying it, then he/she develops a new language or extends an existing language. Advances in programming languages tend to increase the intellectual distance between program statements and what the computer hardware actually does. The language then does more of our work, while decreasing the distance between the programs written and the real world, allowing us to solve real-world problems in the context and language of the real world. On a subtler note, programming languages, software, computer scientists, etc. exert an influence on the real world also, drawing it ever closer to the software realm (see Figure 1).

Inevitably, a new programming language enables a programmer to express an idea or concept in a simpler, more readable manner than what had come before. This simpler, more readable manner allows us to create code that is easier to verify, easier to code, and easier to debug. In essence, the more powerful a programming language is, the easier it is to express complex ideas in a simple manner.

Typical Generations of Programming Languages

The first generation of programming languages, machine codes, is the actual binary codes that the computer hardware directly executes. To program directly in machine code, one must be completely familiar with the individual computer being programmed, including its architecture and its native Central Processing Unit (CPU) instruction set. Programming in a different computer's machine language is like switching from Spanish to German.

The second generation of programming languages, assembly languages, was little more than mnemonics (symbols) on top of machine language instructions.

There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs [2].
— Lawrence Flon

Typically, one assembly language operation is translated into a single, equivalent machine-code operation. When programming Assembler, we still need to understand how the CPU works, and what the command set is. However, we can forget about the codes underlying the instructions and think about the CPU-level activity that is necessary to accomplish the task.

Third generation programming languages are CPU and machine-code independent. Early third generation languages, like Fortran and COBOL, are not completely pure, as many of their data types and control structures derived directly from machine-code operations. Later languages, like Ada and Pascal, were designed specifically to be machine-independent.

There is no general agreement on what the fourth, fifth, and future generations of programming languages are. Some argue that non-procedural languages (or declarative languages), artificial intelligence languages, code generation applications, or object-oriented languages are all contenders. Part of the reason there is no general agreement is that unlike computer hardware generations, later programming languages did not supplant earlier programming languages but instead solved domain specific problems or complemented existing third generation languages.

At one time, it was projected that there existed more than 450 languages being used to develop Department of Defense (DoD) applications [1]. Web sites like <http://oop.rosweb.ru/Other>, <www2.latech.edu/~acm/HelloWorld.shtml>, <http://directory.google.com/ Top/Computers/Programming/Languages>, and <http://sk.nvg.org/lang/lang.html> list more than 2,000 programming languages and <www.levenez.com/lang> shows the evolutionary path of many programming languages in terms of which languages begot others.

However, rather than debate what exactly constitutes a fourth, fifth, or future generation of programming languages, this article describes several general evolutionary trends that have influenced programming languages, as well as some specific recent advances.

Machine-Independent Programming

An ongoing evolutionary trend with one of the longest histories is that of reducing the dependency of programming languages on any particular computer's hardware. The evolutionary goal of machine-independent programming has been to be able to write a program once that could then be run on multiple types of hardware. This would free the application program from the particular hardware on which it was developed.

Control structures were the first to be freed from the tyranny of the computer hardware. The initial control structures were simple jump statements where instructions followed each other sequentially until a jump command caused it to start executing a different sequence. In Fortran, the GOTO command, both to line numbers and later to symbolic labels evolved out of machine code jumps. Fortran also had a primitive for loop, the DO statement and an IF statement.

Algol popularized structured control statements where the statement itself could have substatements and ushered in the structured programming revolution and the *GOTO-considered-harmful* debate. Prior to 1968, most of the commonly used programming languages routinely used GOTO. Starting in the late 1960s, the programming community debated if the use of GOTO was useful, necessary, and/or harmful to good programming practices.

This debate started with the seminal paper "GOTO Statement Considered Harmful" [3]. In this paper, author Edsger W. Dijkstra said, "The quality of programmers is a decreasing function of the density of GOTO statements in the programs they produce." Although this topic was hotly debated for several years, it is now generally recognized that the GOTO statement decreases program understandability and quality. With the structured programming revolution, thereafter followed case statements, generalized loops, tasks and co-routines, exception handling, and parallel programming.

Another fruitful area of evolution toward machine-independent programming has been with data structures. Initial data items were limited to those that had direct hardware representations (i.e., various-sized integer data types and then later floating-point data types.) Later came logical data, characters, strings, Booleans, and enumerated types. For years, COBOL was the ultimate language in terms of representing and manipulating data. Arrays were initially physically adjacent integers or floating-point data; gradually, more generalized arrays, records, and nested data structures appeared. Later came strong data typing, user-defined data types, and dynamic data structures. Pointers, which were present since the very beginning, evolved to become more structured and have often been left out of modern languages or have been restricted across a number of dimensions.

Once language elements were divorced from computer hardware elements, entire languages could be made more compatible across different hardware platforms. One of the goals in designing the Ada programming language was that an Ada program could be transported to any other computer and need only be recompiled on a validated Ada compiler in order for it to be executed. Another method for making programming languages cross-platform compatible was to develop a virtual computer (called a virtual machine) that replaced the computer hardware as the target on which the programming language ran.

The Rise of Virtual Machines

A virtual machine (VM) is a program that creates an artificial or abstract computer running on top of an existing computer. The VMs hide the normal computer hardware behind a simpler or different computational model. The earliest VMs enabled computer scientists to create programming languages specifically for new and different computational models. Lisp (a functional language) and Prolog (a logic language) are the earliest programming languages to run on top of a VM.

Functional languages, in their purest form, eliminate loops, GOTOs, assignment statements, and all forms of side effects. Their VM does not support such constructs. Functional languages retain IF statements and simulate loops with self-referencing functions (i.e., recursive calls).

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities [5].
— Edsger Dijkstra

Logic languages on the other hand, eschew direct control by the programmer entirely in favor of a VM: An answer is not so much computed as it is deduced from programmer-supplied facts and rules. The VM determines which facts to use and which rules to apply to solve the problem.

To transport these programming languages to other hardware platforms, one must only develop a VM for that system. In the 1970s, to make it easier to port the Pascal language to different computers, a Pascal VM was developed that accepted an intermediate language called P-code. The intermediate language is so named because it is an intermediate step between the original programming language and the computer hardware language. Pascal code was compiled to P-code, which was then interpreted by the Pascal VM. At the time, this concept did not catch on because executing an intermediate code program on a VM was much slower than executing an equivalent compiled program.

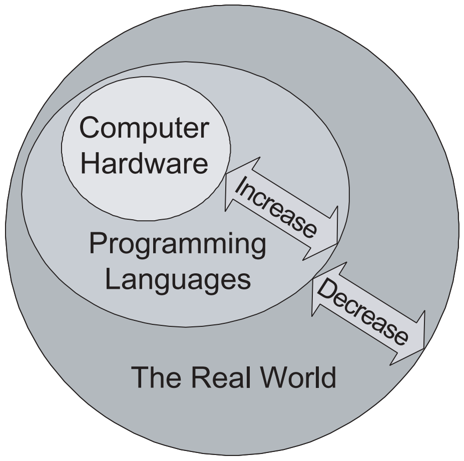

Figure 1: *Distance Between Programming Languages and the Real World Decreases*

The Java programming language was expressly designed to be compiled to a VM. The Java virtual machine (JVM) is a self-contained operating environment.

"This design has two advantages:
- System Independence. A Java application will run the same on any JVM, regardless of the hardware and software underlying the system.
- Security. Because the JVM has no contact with the operating system, there is little possibility of a Java program damaging other files or applications" [4].

The JVM is so small and compact that it can easily be downloaded and installed over the Web. While it still runs slower than compiled code, the benefits have been enormous. Microsoft has developed a similar language called C# (pronounced C sharp) with its intermediate language, Microsoft Intermediate Language (MSIL) and associated VM.

In the future, very few programming languages will be compiled to machine code directly. Instead, VMs like the Java virtual machine or the Common Language Runtime (CLR), the virtual machine for C#, will be the intermediary. Only those applications that need additional speed will use just-in-time compilers to compile the intermediate code (Java byte code, MSIL, or others) into machine code. Thus, most languages will have at least a two-step translation process: compiler to compiler, or compiler to interpreter. Remember, it was not that long ago when assembly level programmers scoffed at languages that needed compilers because they believed a compiler could never produce code that achieved the speed of a hand-coded assembly.

In addition, the existence of VMs, and the intermediate languages that run on them, will be a boon for other languages as it will make it easier to port new languages to multiple machines. Rather than creating a compiler or interpreter for a new language that has computer hardware as the target language, programmers merely produce intermediate code for a VM. The JVM already has more than 160 experimental, research-oriented, and commercial languages that use Java byte code as the intermediate language [6].

Programming Language Interoperability

One reason so many programming languages have been developed is that language developers designed different languages to solve different types of problems. In theory, a software developer would be able to pick the right language for the task. In practice, it has been difficult to integrate different programming languages so developers tend to stick with general-purpose languages.

To further their use, programming language designers feel compelled to make their languages more appealing by adding new features and language constructs until the languages become very complex to use and master. No one knows what feature or capability will be a success in the end, so language designers add new features to existing languages to make them more competitive. Pl/1, the Algol family of languages, Ada-Ada 95, and C-C++ all suffered from this problem. For example, Ada was designed to be the programming language for the DoD, supplanting almost all others. Even with Ada, it was felt necessary to periodically update the language to ensure that it had features and capabilities necessary to make it competitive in current environments, hence Ada 95. Language bloat through feature addition is a naturally occurring phenomenon.

On a related note, developing a new programming language has often been hindered by the lack of existing libraries and components for that language. Much of the power of today's programming languages comes from their ability to use existing libraries of code. It is possible to design bridges between new and old languages so that the other's libraries can be accessed, but it is an endless effort that must be done for each one [7].

C++ was built as a superset of C to take advantage of all of the existing C programmers and all of the existing libraries of code. Many would argue that a completely new and clean design would have resulted in a much better language. In the same vein, there are probably millions of lines of Fortran libraries in existence. Fortran keeps evolving to include new features, but backwards-compatibility with existing libraries is still possible. Were it not for all of the Fortran libraries in many engineering application areas, the developers would probably have switched to a newer language years ago. What is needed is a mechanism that enables programming languages to interoperate, and yet be independent of any particular programming language.

The first steps toward this goal were for languages to be able to make external calls, i.e., calls to a procedure or function that is in a different language and to exchange data in that call. Most modern languages have a mechanism that enables them to make an external call. However, few programming languages have that capability defined as part of their language definition, and none have such clearly defined routines for converting data elements between programming languages like Ada does [8]. Programming language and machine-independent data representation standards such as External Data Representation, Network Data Representation, and eXtensible Markup Language were developed to make it easier to exchange data between different programming languages on different computing platforms.

Language serves not only to express thought but to make possible thoughts which could not exist without it [9].
— Bertrand Russell

Another step in the evolutionary path has been to enable components to be built in nearly any programming language that can then be accessed by nearly any other language. In essence, by making the code libraries more open and non-language specific, it is easier for languages to rely on the strengths of other languages instead of incorporating all of the necessary features themselves.

Current technologies that enable language-independent programming are Dynamic Link Library, Component Object Module, and Common Object Request Broker Architecture. Each of these technologies has enabled a service to be made available, and yet shields the calling programming language from the called programming language. These technologies enable functionality to be built and shared independently of the language and machine by developing a standardized calling model that is programming-language neutral.

The latest step in the language interoperability evolutionary trend is the dot-net environment and the CLR. In this environment, classes and objects in one language can be used as first-class citizens in another. Not only can one language call services in another language, but it can inherit from the classes of another language, declare variables based on types declared in another language, handle thrown exceptions from a routine in a different language, and debug across languages [7].

The trick is, not only is there an intermediate language, but an intermediate type system exists as well that retains high-level data-type information such as classes and inheritance hierarchies. Once a program is compiled into the dot-net architecture, its language of origin disappears, and it becomes language neutral. Consequently, other dot-net aware languages (actually their compilation systems) can access those types. The language interoperability evolutionary trend and the machine-independent programming evolutionary trend intersect under the dot-net architecture.

Increasing Modularity

Software designers reduce the complexity of software by decomposing difficult problems into smaller, easier to solve pieces. Initially this concept of modularity was supported in programming languages by procedures, functions, and user-defined data structures. Eventually, the evolutionary paths of control and data abstraction merged into larger structures. The ideas of encapsulation and information hiding, which are two key parts of modularity, led to evolutionary improvements in programming languages to support those concepts.

Programming languages evolved to provide support for modularity by making it easier to create abstract data types (such as a stack, set, queue, or hash table) by allowing separate code units that can be compiled and by syntactically supporting modules, packages, and namespaces. Object-oriented programming is a form of modularity. Although the first object-oriented language, Simula, was developed in 1965, other languages did not adopt that paradigm until the mid-1980s.

A final unit of functional modularity is the framework. A framework is much larger than an abstract data type or a class hierarchy. A graphical user interface (GUI) framework, for example, contains all of the necessary routines and classes to make programming user interfaces easier.

Programming languages have evolved to provide a wide variety of syntactic and semantic supports for modularity and information hiding, but not all forms of modularity are equal. Coupling refers to how many other modules a module references. Cohesion refers to how single-minded a module is – a way of measuring how many things a module accomplishes. A highly cohesive module is one that solves a single problem; a low-coupled module is one that is self-contained and has few ties to other modules. A module that is highly cohesive with low coupling is easier to maintain because it has fewer dependencies.

To date, programming languages provide little syntactic support to facilitate the creation of highly cohesive and low-coupled modules (other than just making it possible). This is problematic because there are some facets of a problem that crosscut normal module boundaries. When programmers combine different facets of the problem into a single module, the code is longer, less readable, and less easy to maintain, reuse, and evolve.

Programming languages have instruction and data topologies that have evolved as programming languages evolve [10]. During the first 20 years of computing, programming languages supported mixing code and data (assembly languages) or had global data structures (Fortran common statements) resulting in poor cohesion and high coupling. In the next 20 years, module-oriented programming languages evolved that enabled programmers to place related routines and data structures within the same module and provide limited access via exported routines thus providing direct support for encapsulation and information hiding, which can be used to improve both cohesion and coupling. The last 15 years have seen the rise of object-oriented programming languages that enable programmers to decrease the coupling and increase the cohesiveness of their program designs even further.

Unfortunately, different facets of a problem often defy being easily separated into cleanly modularized subunits. Components of a system are usually arrived at by decomposing a problem's functionality. Other aspects of the problem such as performance, security, communication, synchronization, failure handling, persistence, integrity and error-checking rules, design patterns, and concurrency often crosscut the boundaries of the functional components. These crosscutting aspects necessarily increase the coupling and decrease the cohesiveness because our current programming languages have no other way to deal with them.

For example, many typical applications have error-handling code that crosscuts module boundaries and spans the application. Similar bits and pieces of error handling code are scattered throughout the application. A design change that affects error-handling code will necessarily affect all of those scattered bits and pieces [11]. All crosscutting modifications to the code affect readability and maintainability, increase coupling, and decrease cohesion.

Programming languages currently only support composing different components during run-time by procedure or method invocation and during development time by inheritance. Software developers are forced to manually compose the different aspects in the code, which can cause similar code to be scattered across an application and can cause existing code to become a tangled mess of differing concerns. Aspect-oriented programming languages address the different facets in clean, modularized ways. Aspect-oriented programming languages separate different aspects of the problem into different, easily maintainable modules and then automatically weaves the aspects together (using an interpreter, compiler, or preprocessor) just prior to normal processing.

The city's central computer told you? R2D2, you know better than to trust a strange computer [12].
— C3PO

The most advanced general-purpose aspect-oriented programming language, AspectJ, is an extension of Java (<https://aspectj.org>). AspectJ uses pointcuts to specify join points in the normal Java code and uses Advice to specify additional Java code to be executed at the join points[1]. The pointcut and Advice code are maintained separately and the AspectJ compiler weaves the Advice Java code into all the specified join-point code locations, thus the different crosscutting concerns can be developed and maintained separately eliminating a tangled mess of differing concerns.

Aspect-oriented programming has barely broken out of its research roots[1], but it is already having an influence on language design. Currently there are aspect-oriented programming extensions being made to a variety of programming languages (several Java variants, C, C++, C#, Ruby, Perl, Python, and several Smalltalk variants). Links to those and other domain specific, aspect-oriented programming languages can be found at <http://aosd.net/tools.html>.

Scripting Languages

Scripting programming languages, also called glue languages or integration languages, are not designed for developing large-scale applications from scratch (or with the help of a large class library). They leave that task to mainstream, or system programming languages. Instead, scripting languages construct applications by gluing together pre-written components. Scripting languages may seem in some ways to be an evolutionary throwback, but in reality they are just programming languages that are being optimized (evolved) along different lines.

Scripting languages originated as command languages for computer operator tasks. Job Control Language in the '60s and Rexx in the '70s were early IBM mainframe scripting languages. The original Unix scripting language developed in the '70s was sh, and has since been followed by csh, bash, ksh, and others. The Unix shell script languages made it easy to create new applications by composing existing applications that piped and filtered data from one application to the next. The ease with which new applications were created was probably the most important reason for Unix's popularity among application developers [13].

In the late '80s, scripting languages took a major evolutionary leap with the development of Perl and Tcl. Perl grouped together some of the Unix text processing applications (sh, sed, and awk) and added more sophisticated input and output statements and control statements. Perl has become the primary means of creating on-the-fly common gateway interface scripts for dynamic Web pages [14]. Tcl started out as an embedded command language for end-user tailoring of the application.

Tcl/Tk extended Tcl so that it can easily create GUI's in Windows, Mac OS, and the Unix X windowing system.

In the almost 15 years since, many other scripting languages followed (Visual Basic, Python, JavaScript, Icon, Ruby, etc.) for many purposes (rapid integration of Web, database, and GUI components; system management; automated testing; Web scripting; etc.). The creators of these scripting languages designed them to be flexible and very powerful. Most scripting languages are interpreted instead of compiled, dynamically typed, perform automatic conversions between types when needed, have loose and forgiving syntax, have powerful text manipulation and input/output capabilities, and can often create and execute additional code on the fly. These language features make scripting languages extremely useful for rapidly interfacing with legacy applications, acquiring and manipulating data from those applications, and either displaying the data to the user or sending it on to some other application.

System programming languages (C, Ada, Java, C++, etc.) are designed to develop applications from scratch with the help of a few class libraries. Scripting languages assume the existence of the necessary components and quickly and easily join those components together to form a larger application. System programming languages have high overhead in terms of their structure (try writing a Hello World! program in Java). Scripting languages can do quite a lot with just a few lines. A single line of scripting code may execute hundreds of machine code instructions where a system language may only execute tens of machine code instructions [15].

Scripting languages will never replace system languages, as scripting languages are not very good at programming complex algorithms and complex data structures, or for manipulating large data sets. However, scripting languages have their own strengths, including easily connecting pre-existing components, robustly manipulating a variety of data types from a variety of sources, rapidly developing GUIs, straightforward text manipulation, and creating and executing code on the fly. Scripting languages are the duct tape of the programming world.

Scripting languages are still very young compared to system languages. In all likelihood, many more evolutionary improvements will be made to them to make their strengths even greater. We predict that the easy work of complex algorithms, elaborate data structures, and brute force processing of large data sets will continue to be accomplished by system programming languages. More and more reusable components and services will be constructed using systems programming languages. However, the more difficult part of programming, that of developing a robust, easy-to-use application that is easily extended and modified as requirements change and the operational environment varies, will become more and more the job of scripting languages. Both types of languages will continue to evolve, but toward their strengths.

Conclusion

The evolutionary path of programming languages has not been without its share of dodos and passenger pigeons: the Algol by-name parameter passing mechanism and the dynamic scoping semantics of Lisp to mention two. Many languages are introduced with great fanfare and then die unnoticed (PL/1, Modula-2). Some language features (such as unrestricted pointer use and GOTOS) are historical relics: They are generally regarded as bad and unsafe, but they continue to be included in languages (C++).

There will always be things we wish to say in our programs that in all known languages can only be said poorly [16].
— Alan J. Perlis

As the developers' needs have evolved, so have the abilities of programming languages evolved. If a programming language is not expressive enough, then it must evolve to allow its users the ability to articulate their abstractions or it will become extinct.

At one time, many believed that a single multi-purpose programming language would allow developers to standardize. However, the wide variety of problems that need solving and the diverse philosophies of developers have appropriately led to different languages for different purposes. Certain domains will continue to have special-purpose languages that focus on the features that are unique to the applications of that domain; those languages will continue to evolve and be optimized for those domains (e.g., ProModel – a simulation language, and MATLAB – an engineering language are examples of this).

General-purpose languages will also continue to evolve by incorporating new features and programming paradigms. These general-purpose languages must also shed features that become outmoded, and be redesigned to become leaner and meaner in order to try to eliminate bloat and regain simplicity. Programmers do not need to use complex languages, there is enough complexity in the world for them already.

During all this evolution though, the basic role of a programming language will not change – allowing the developer to easily express abstract ideas in a language that a machine can execute. Future advances in programming languages will only be made possible by the evolutionary advances (and cullings) being made today. In the near future, the general evolutionary trends of increasing machine independence, increasing programming language interoperability, and increasing modularity will continue.◆

"Are you quite sure that all those bells and whistles, all those wonderful facilities of your so called powerful programming languages, belong to the solution set rather than the problem set" [17]?
— Edsger Dijkstra

"The limits of your language are the limits of your world" [18].
— L. Wittgenstein

References

1. Hook, Audrey A., et al. "A Survey of Computer Programming Languages Currently Used in the Department of Defense: An Executive Summary." CrossTalk 8.10 (Oct. 1995) <www.stsc.hill.af.mil/crosstalk/1995/10/ index.html>.
2. Flon, Lawrence. "On Research in Structured Programming." SIGPLAN Notices 10:10 (Oct. 1975).
3. Dijkstra, Edsger W. "Go To Statement Considered Harmful." Communications of the ACM 11.3 (Mar. 1968): 147-148.
4. Webopedia. Online dictionary and search engine for computer and Internet technology <www.webopedia. com>.
5. Dijkstra, Edsger W. Selected Writings on Computing: A Personal Perspective. Springer-Verlag, 1982.
6. Tolksdorf, Robert. Programming Languages for the Java Virtual

Machine. <http://grunge.cs.tu-berlin.de/~tolk/vmlanguages. html>.

7. Meyer, Bertrand. "Polyglot Programming." Software Development May 2002.
8. Ada 95: The Language Reference Method and Standards Libraries. Appendix B. ANSI/ISO/IEC-8652:1995 <www.adahome.com/rm95>.
9. Russell, Bertrand. <www.angelfire.com/realm/firelight63/Words_Russell_Bertrand.htm>.
10. Cook, Dr. David A. "Evolution of Programming Languages and Why a Language Is Not Enough to Solve Our Problems." CrossTalk 12.12 (Dec. 1999).
11. Kiczales, Gregor, et al. Aspect-Oriented Programming. Proc. of In ECOOP '97 Object-Oriented Programming, 11th European Conference. LNCS 1241: 220-242.
12. C3PO. "Star Wars – Episode V: The Empire Strikes Back."
13. Tcl Developer Xchange. History of Scripting <www.tcl.tk/doc/script/scriptHistory.html>.
14. Laird, Cameron, and Kathryn Soraiz. "Choosing a Scripting Language." SunWorld. Oct. 1997 <http://sunsite.uakom.sk/sunworldonline/swol-10-1997/swol-10-scripting.html>.
15. Ousterhout, John K. "Scripting: Higher Level Programming for the 21st Century." IEEE Computer Mar. 1998 <http://home.pacbell.net/ouster/scripting.html>.
16. Perlis, Alan J. "Epigrams in Programming." ACM's SIGPLAN Sept. 1982.
17. Dijkstra, Edsger W. A Discipline of Programming. Englewood Cliffs, NJ: Prentice Hall, 1976.
18. Ludwig, Wittgenstein. Tractatus Logico-Philosophicus 5.6. Trans. by D. F. Pears, B. F. McGuinness, London: Routledge and Kegan Paul, 1961.
19. Clark, Lawrence R. "A Linguistic Contribution to GOTO-less Programming." Datamation 1973. Reprinted in Communications of the ACM 27.4 (Apr. 1984): 349-350.

Note

1. A join point is similar in some respects to the infamous and semi-mythical COME FROM statement [19], which was one of the salvos fired in the famous *GOTO-considered-harmful* debates mentioned earlier in the article. For a formal and correct definition of join points and point-cuts, see <http://aspectj.org/servlets/AJSite>.

About the Authors

Lt. Col. Thomas M. Schorsch, Ph.D., is deputy department head, Computer Science department at the U.S. Air Force Academy. He has served in the Air Force for 17 years in a variety of software-related capacities from application programming to managing the development and installation of a new Cheyenne Mountain Command and Control System. Schorsch has a bachelor's of science degree from the U.S. Air Force Academy, a master's of science degree from the University of Colorado, and a doctorate degree from the Air Force Institute of Technology, all in computer science. His most well-known CrossTalk publication is "The Capability Im-Maturity Model" <www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1996/11/xt96d11h.asp>.

U.S. Air Force Academy
Colorado Springs, CO 80840
DSN: 333-8793
E-mail: tom.schorsch@usafa.af.mil

David A. Cook, Ph.D., is the principal engineering consultant for Shim Enterprises, Inc. Dr. Cook has more than 27 years of experience in software development and software management. He was formerly an associate professor of computer science at the U.S. Air Force Academy (where he was also the department research director) and also the deputy department head of the Software Professional Development Program at the Air Force Institute of Technology. He has a doctorate degree in computer science from Texas A&M University, and he is an authorized Personal Software Process instructor.

Software Technology Support Center
7278 4th Street Bldg. 100
Hill AFB, UT 84056
Phone: (801) 775-3055
DSN: 775-3055
Fax: (801) 777-8069
E-mail: david.cook@hill.af.mil