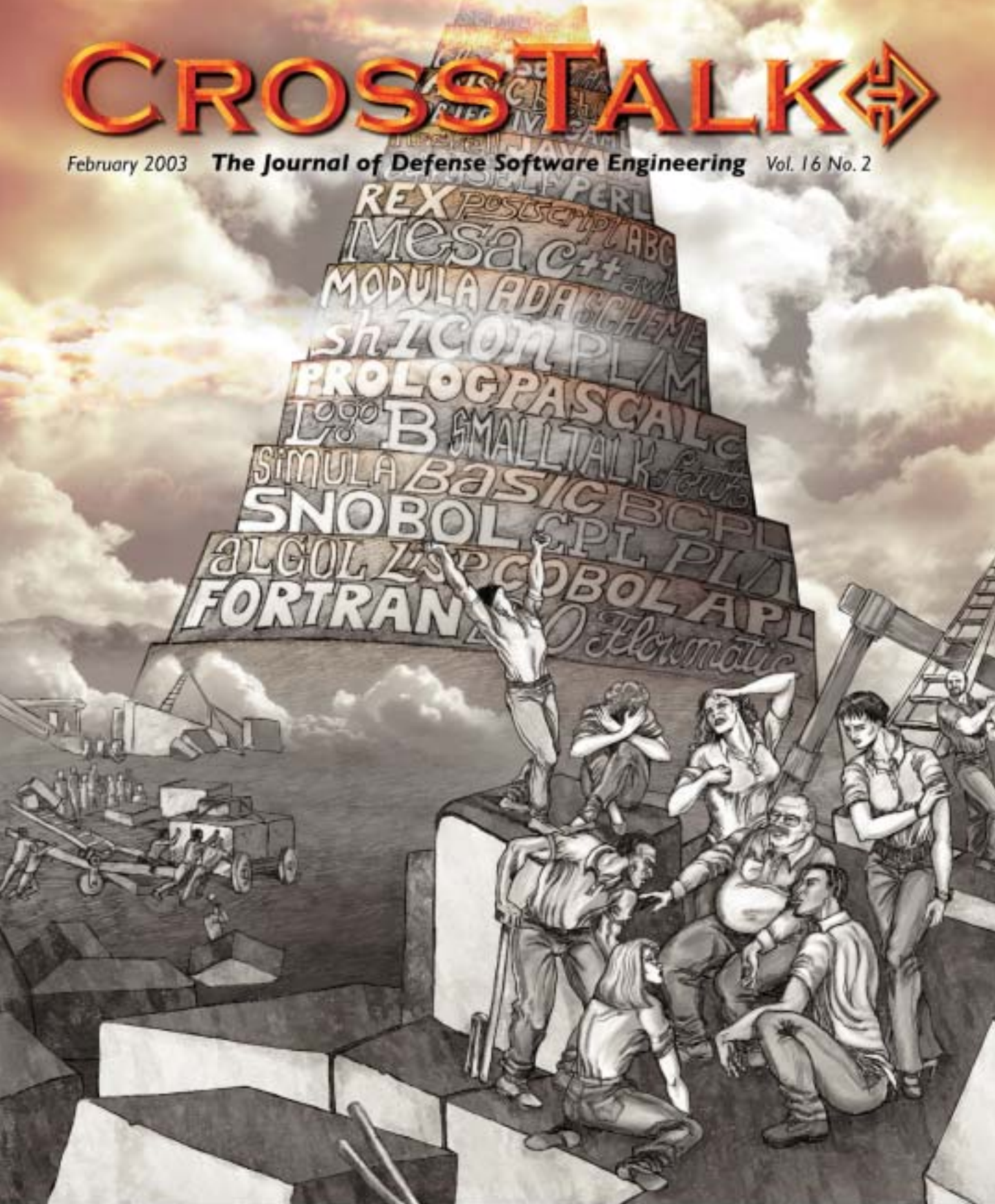
CROSSTALK

February 2003 The Journal of Defense Software Engineering Vol. 16 No. 2



PROGRAMMING LANGUAGES

Programming Languages

ON THE COVER
Cover Design by Kent Bingham, inspired by Gustave Doré's biblical illustration.

Software Engineering Technology

Departments

CROSSTALK

SPONSOR *Lt. Col. Glenn A. Palmer*
PUBLISHER *Tracy Stauder*
ASSOCIATE PUBLISHER *Elizabeth Starrett*
MANAGING EDITOR *Pamela Bowers*
ASSOCIATE EDITOR *Chelene Fortier*
ARTICLE COORDINATOR *Nicole Kentta*
CREATIVE SERVICES COORDINATOR *Janna Kay Jensen*

PHONE (801) 586-0095
FAX (801) 777-8069
E-MAIL crosstalk.staff@hill.af.mil
CROSSTALK ONLINE www.stsc.hill.af.mil/crosstalk
CRSIP ONLINE www.crsip.hill.af.mil

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail or use the form on p. 30.

Ogden ALC/MASE
7278 Fourth St.
Hill AFB, UT 84056-5205

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CROSSTALK editorial board prior to publication. Please follow the Author Guidelines, available at www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf. CROSSTALK does not pay for submissions. Articles published in CROSSTALK remain the property of the authors and may be submitted to other publications.

Reprints and Permissions: Requests for reprints must be requested from the author or the copyright holder. Please coordinate your request with CROSSTALK.

Trademarks and Endorsements: This DoD journal is an authorized publication for members of the Department of Defense. Contents of CROSSTALK are not necessarily the official views of, or endorsed by, the government, the Department of Defense, or the Software Technology Support Center. All product names referenced in this issue are trademarks of their companies.

Coming Events: We often list conferences, seminars, symposiums, etc. that are of interest to our readers. There is no fee for this service, but we must receive the information at least 90 days before registration. Send an announcement to the CROSSTALK Editorial Department.

STSC Online Services: www.stsc.hill.af.mil
Call (801) 777-7026, e-mail: randy.schreifels@hill.af.mil

Back Issues Available: The STSC sometimes has extra copies of back issues of CROSSTALK available free of charge.

The Software Technology Support Center was established at Ogden Air Logistics Center (AFMC) by Headquarters U.S. Air Force to help Air Force software organizations identify, evaluate, and adopt technologies to improve the quality of their software products, efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery.

We've Come a Long Way From Machine Code to Current Programming Languages

As I reviewed this month's CrossTalk theme articles on programming languages, I found it interesting to look back at the evolution of programming during the last 30 years. You can probably guess my age when I admit that the first programs I wrote in college were in machine code.

Lt. Col. Thomas M. Schorsch and Dr. David A. Cook begin this issue with their article *Evolutionary Trends of Programming Languages*. The authors discuss the typical generations of programming languages defining first, second, and third generations, then admit that there is probably no general agreement on what constitutes fourth, fifth, and future generations of languages. (Of course, I actually related to writing programs in a first-generation or machine-code language. One assignment I remember required us to actually write a fully functional program with a limitation of 100 bytes of storage.) Schorsch and Cook go on to describe several general evolutionary trends that have influenced programming languages, as well as some specific recent advances. After discussing many different languages and how they came about, the authors conclude that throughout this total evolution the basic role of a programming languages does not change. This role is to allow the developer to easily express abstract ideas in a language that a machine can execute.

Dennis Ludwig's article, *Language Considerations*, proposes some pertinent ideas for dealing with the question: What programming language should I use for my new project? He includes some real-world examples of how this decision has been made in the past. He then suggests some decision-making parameters that could be formalized into a decision table that could form the basis of a decision-making process.

In *SEPR and Programming Language Selection*, author Richard Riehle laments the misunderstanding and misuse of the 1996 memo from Assistant Secretary of Defense Emmett Paige. He contends that many readers mistakenly assumed the memo's intent was a license to abandon Ada rather than advice to include language selection as part of a rational evaluation step. He discusses some criteria used to evaluate the selection of a language for a particular purpose or project. Riehle contends that the strengths and weaknesses of the more popular languages should be well understood so that the decision whether to choose them or to reject them is based upon consideration of sufficient specific criteria reflecting the project's full life-cycle needs.

We are fortunate in this issue to also get an international perspective on the growth and challenges of the global software market. François Coallier, chairman of Sub-Committee 7 (ISO/IEC JTC 1/SC7), in his article *International Standardization in Software and Systems Engineering* provides an introduction to international standardization in information technology. This article provides status and describes the current activities in international software and systems engineering standardization. Coallier explains why all of these are important for professionals and organizations in the software arena.

Following this ISO tutorial is an application example from another international author, Dr. Paolo Donzelli of Italy. In *An Enterprise Modeling Framework for Complex Software Systems*, Dr. Donzelli describes a goal-oriented, agent-based Enterprise Modeling Framework where advanced requirements engineering techniques are combined with software quality modeling approaches. This provides an environment within which stakeholders and analysts can easily cooperate to discover, verify, and validate the requirements for a new software system.

Lastly, CrossTalk's Associate Publisher Elizabeth Starrett reports on ways to expand your people skills in *Highpoints From the Amplifying Your Effectiveness Conference*. This conference uses out-of-the-box techniques to teach personal skills to complement technical skills for a total package that can help you improve your organization, project, or process.

I hope that this issue will prove useful in your efforts to understand the challenges associated with the myriad of programming languages available, and how to make some practical choices for your particular needs.

H. Bruce Allgood
Deputy Director, Computer Resources Support Improvement Program

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.

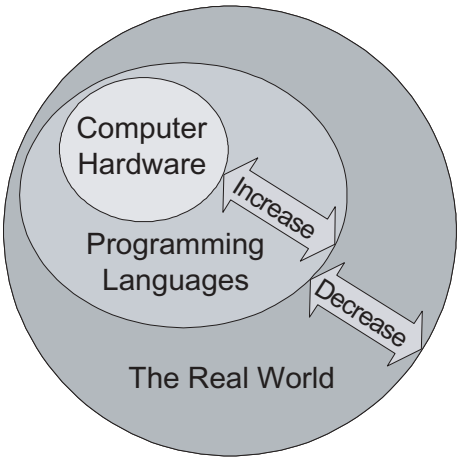


Figure 1: *Distance Between Programming Languages and the Real World Decreases*

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 pointcuts, 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

Language Considerations

Dennis Ludwig
Aeronautical Systems Center

A major question asked when beginning a project is, "What programming language should I use?" This article will provide some ideas to help make this choice. First, it will present some real-world examples of how this decision has been made in the past, and then some decision-making parameters will be explored. It is intended that these ideas could be formalized into a decision table that could be the basis of a decision-making process.

Back when the 6502 microprocessor was competing with the 8080 microprocessor, my college professor posed a question to our class. Why did IBM choose to use the Intel 8080 for its personal computer when he considered the 6502 to be a superior microprocessor?

His answer was that the IBM engineers used the 8080 because they were familiar with it. They knew how to program it, and they had to get a product out the door fast. So they used the product they knew best.

Similarly, during the time the government was under an Ada mandate, which meant all new software development would be Ada, I knew of a company that presented a proposal to develop a flight line electronic warfare tester using Fortran. Why? Their microwave engineers knew Fortran best.

Obviously, one common method of choosing a programming language for a new project is to let the engineers decide, and go with the language they already know. However, there can be problems with this approach. System design should have a team approach that considers all life-cycle phases. This includes asking, "Will the design engineers be around for the maintenance phase?" "Will the choice be made on what is best for the project, or on what is best for the engineers?"

The project manager must ensure that the language selection technique being used is not pure careerism, which is making decisions based on what is best for someone's career instead of on what is best for the organization. For example, during the era of the Ada mandate, professional magazines were listing plenty of job offerings for C++ programmers, but none for Ada programmers. Although Ada is a superior language with better array handling facilities, stronger typing, more readability, etc., marketable programmers were perceived to be the ones with C++ experience. More recently, the trend among programmers has been to push Java as the language to handle all projects. Have you checked the job listings lately?

It would be easy to simply let market demands decide which language to use. Some people feel that they cannot go wrong with the most well-known language. Their rationale is that if it were not the best, it would not be the most well known. Currently, that means using Visual C++ or C# hosted on a Microsoft Windows operating system. For many projects, this will do the job. However, a popularity contest is not the best way to make what is essentially a technical decision. A software product that works very well in one project may not be suited for another under different circumstances.

> *"The project manager must ensure that the language selection technique being used is not pure careerism, which is making decisions based on what is best for someone's career ..."*

Quickly changing markets could leave a project stranded with an obsolete software base.

Another method of choosing a programming language could be called the Dilbert method. Under this scenario, executives lament that they are not able to find Ada programmers because the local schools are not teaching Ada. Meanwhile, their respective companies are advertising to hire C++ programmers, but not Ada programmers. The executives whine that they do not have any control over their advertising departments. So we are led to believe that the advertising or human relations departments make the programming language decisions. The Dilbert rationale is that since the languages major companies advertise for in trade journals determines what students want to learn, this then determines what colleges teach. Since major companies rely on colleges to put out the latest, or at least the best, technology, they advertise for what is most popular with the colleges.

This author believes our society is currently trapped in an unhealthy cycle that is propagating an inferior language. A way to improve the process of deciding which language would be best for a particular development project should be developed and documented.

Language Selection Process

The first step in a language selection process is to decide what computer hardware and operating system will be used. Defining the hardware and operating system greatly narrows the choice of available languages. If an Atari 800 were going to be used for the brains of a robot pet, then a compiler or assembler for that machine would be useful. If plans call for using RSX-11 on a program decision package, then that environment will bind the choice of languages.

To help in choosing an operating system, ask the following three questions: "Will direct access to the input/output (I/O) ports be necessary?" "Will multitasking be needed?" "What support will be available?" If direct control of I/O ports is needed, then an operating system like DOS or Linux or a real-time operating system that provides this service will be required. However, Windows will not allow it. If multitasking is required, then DOS drops out of consideration. If direct access to I/O ports is not required, then Windows can be considered. The chosen operating system will constrict the language choice.

Lastly, the availability of support maintenance for an operating system also factors into the final choice. Operating systems themselves are large computer programs that have bugs and will require support. Unfortunately, operating systems are numerous but few have good support.

(For more information on operating system considerations, see [1].)

The next step in the language selection process is to define or decide on a design method. Some considerations are to create a flow chart, top-down design, or object-oriented design. If an object-oriented design is required or desired, then strictly procedural languages like Fortran drop out of consideration. Some languages that claim to be object oriented are Ada 95, C++, Common Lisp, and Smalltalk. Ada 95 and C++ can also be used as procedural languages using flow charts or other design tools.

The following list itemizes some things to consider in choosing a language. Using this list, managers could build a decision matrix, assign weights, and arrive at a decision that would be documented.

1. **Speed.** It is hard to beat assembly language for speed, but some compiled languages with optimized compilers can match it. Avoid interpreted and scripting languages if possible because they are slower.
2. **Operating System.** Defining the hardware and operating system greatly narrows the choice of available languages.
3. **Program Size.** There is a difference between programming large systems and writing a small program to calculate a mortgage amortization. For large system programs, a large system programming language with clean interfacing is required. Ada would be a good choice here.
4. **Reuse and Cost.** If it has already been coded, why reinvent it? If the need is for a small neural network, buy a book with the code included. Then it can be rewritten to make it faster or more robust if needed.
5. **Engineers' Knowledge.** It takes two years to learn a language like Ada or C++. There are two-week courses and 21-day instruction books for most languages, but it takes two years of really working with the language to become good. Look at what languages are already being maintained in the organization. Some synergy could be gained from staying with what is already being used, considering any trade-offs with obsolescent factors. Or ask, "Is it time to upgrade the workers' knowledge?"
6. **Required Pointers.** Some languages like early versions of Fortran, Java, and Basic do not have pointers. Others do, including C, C++, Delphi, Ada, and the latest version of Fortran.
7. **Other Data Structure Considerations.** Will enumerated types or records (such as struct in C) be required? Will array slicing be required or helpful? Is string manipulation built into the language, provided as a library, or will it have to be coded separately?
8. **Garbage Collection.** The reclamation of heap-allocated storage after its usefulness in a program is called garbage collection. Automatic garbage collectors can be useful for some applications, but can be damaging in other situations because it relinquishes control of the program. For some languages that do not have automatic garbage collection, like C++ and Ada, a routine could be written, if required. If you do not need it, it does not matter. If garbage collection interferes with what you need to do, then choose a language that does not have it. Not having automatic garbage collection gives the programmer more control.
9. **Reliability.** This can be one category or several such as information-hiding capabilities, readability, or strong vs. weak typing rules. Typing strength is a matter of opinion. Every book the author has on C++ claims that it is a strongly typed language, but any language that intrinsically converts floats to integers has no business calling itself strongly typed. This is another issue to be addressed in the organizational software-engineering handbook.
10. **Standardization.** Will one company's compiler produce the same results as the compiler from another company? Will a later version of the same compiler work with earlier code? The last question has been a problem with C++ because the language has changed rapidly over the years. Will the language be around in 10 years or more? Ada's well-documented standard and the Association for Computing Machinery (ACM) special interest group assures its success. On the other hand, the former lack of standardization for C++ impeded portability and programmer efficiency, and even though it now has a standard, it is not being followed. Furthermore, Microsoft C++ is not the same as Borland C++. And while Pascal and C are also standardized, Java is not.
11. **Compiler Tools Like Debuggers.** If a graphical user interface (GUI) is mandated or desired, that is also considered because the compiler would have to interface with the GUI tool. It would take considerable research to get honest ratings in this category. The operating system would be a major player in this consideration. If Microsoft Windows were the platform, Visual Basic, Visual C++, and Ada would be major contenders. But if Unix or Linux platforms would be users as well, then the visual languages would not be considered.
12. **Parallel Processing.** If parallel processing is required, some languages shine brighter than others do. The operating system would also be a consideration here. For most applications, the decision matrix would have *not applicable* here. Ada tasking features make it a good choice for parallel processing needs, but other languages, such as Pascal or C++, could be pressed to perform in this arena.
13. **User Base.** Be careful here. C++ is touted as the most popular programming language today, and it claims a large user base. Because Borland C++ differs from Microsoft C++, and almost every compiler sold has its own brand of C++ (due to lack of standardization), the claim to a large user base can be misleading. The backward compatibility to C has been used as an excuse to expand the user base, but only very trivial programs are actually backward compatible. The ACM has a user group for Ada, and that assures a viable user base for this language. Because Ada is well standardized, the users are truly more portable.
14. **Specialized Areas.** ATLAS is a standard language for automatic test equipment. Although most modern test stations use some form of BASIC or C, keep in mind that there are languages developed for special purposes. Forth was developed to control telescopes, but has been expanded into a

"The language decision is a fundamental design decision that will affect production, testing, training, and maintenance, so the entire system life cycle should be considered."

powerful, flexible language. Lisp and Prolog are languages associated with artificial intelligence. However, general-purpose languages are often used in specialized areas. For example, array slice capabilities of Ada make it a powerful language for database operations.

With these considerations and others that may be program-dependent, a payoff table or decision matrix could be built to make a decision based on program requirements. A good process would eliminate programmer bias and focus on the project. It would also provide documentation that could be used to build experience for decisions in future projects. For more information on developing decision matrixes and using weights in decision making, see [2, 3]. Note that in management jargon, a decision matrix is also known as a payoff table.

Other Considerations

To further enhance the process, the decision maker should be familiar with some basic software engineering concepts. For example, suppose a group inaccurately argues that they want to use C++ instead of Ada because they do not think information hiding is a good idea. Information hiding is not an Ada concept, but a software engineering concept that is also used by C++ in the form of file scope and class scope. The concept is associated with Ada because this language has constructs (packages) that make implementing it easier. A small pamphlet explaining this and other software-engineering concepts would be useful for managers to dispel misconceptions.

Unfortunately, software engineering is not an exact science, but a field full of opinions, contradictions, and poorly defined words and concepts. For example, the concept of *object oriented* has changed over the years. A software-engineering handbook would help to standardize some of the concepts, even if the definitions are just accepted in your organization. It would have to be a *handbook* and not an encyclopedia, or busy program managers will never read it.

Another problem in language consideration is knowing them all well enough to make decisions. It is difficult to judge if a language will meet your requirements without knowing it. Yet no one has time to learn the syntactic structure of all of the available languages. For example, an excuse to use C on a project instead of Ada could be that "the array sizes had to be dynamically allocated, and Ada could not do that." Of course, Ada could do that, but the person did not know that. Thus the decision would be made on incomplete or inaccurate information.

As another example, consider that Java does not have pointers. However, someone who knew the language could build a linked list using the Vector class in java.util. Keep in mind that insertions and deletions are less efficient (see <www.cafeaulait.org/javafaq.html> question 4.1). Language summaries similar to one found at <www.cs.rochester.edu/u/scott/pragmatics/a.html> could be expanded and made part of the process documentation [4].

In the future, the line between operating system and language may get blurred. For example, Niklaus Wirth, the creator of Pascal, along with Jurg Gutknecht have created Oberon, which is an integrated software environment that can run on bare hardware or on top of a host operating system. Oberon is also the name of a programming language. It is the author's understanding that the operating system and the language were developed to operate closely together (see <www.oberon.ethz.ch>).

Language vs. Process

Some people feel that the process and tools are more important than the language. A common belief is that if the design is good, the language selection will not matter. This author believes that the process, tools, and design are dependent on the language. The language decision is a fundamental design decision that will affect production, testing, training, and maintenance, so the entire system life cycle should be considered.

If the popular language today is not around during the maintenance phase of the system, life-cycle costs will greatly increase. Languages that are not standardized may be around, but they might be drastically different from the original, or even from another language going by the same name (as is the case with C++). Front-end costs to move to a different language would include training and purchasing tools. Back-end costs to maintain an obsolete system would be constant training and trying to purchase or maintain obsolete tools. Jovial programmers, for instance, are hard to find. COBOL is another example, as those who fought the Year 2000 battle found out.

Conclusion

Operating systems are complex programs. Compilers are complex programs. Choosing which combination of these complex programs best suits a need is not an easy process, and most often, this process is not documented. This article provides an idea for making and documenting the language decision process using a decision matrix. The table elements presented are not intended to be inclusive, or to even all be necessary. The idea is to use a decision process that is documented and able to be improved.◆

References

1. Silberschatz, Galvin. Operating System Concepts. 6th ed. Indianapolis, IN: John Wiley & Sons, June 2001.
2. Babcock, Daniel L. Managing Engineering and Technology. Prentice-Hall, Inc, 1991.
3. Griffin, Ricky W. Management. Houghton Mifflin Company, 1984.
4. Scott, Michael L. Programming Language Pragmatics. San Francisco: Morgan Kaufmann Publishers, 15 Jan. 2000.

Additional Reading

1. Ghezzi, Carlo, and Mehdi Jazayeri. Programming Language Concepts. John Wiley and Sons, 1987.
2. Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. Essentials of Programming Languages. The MIT Press, 2001.
3. Jones, Richard, and Rafael D. Lins. Garbage Collection, Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons, 1996.

About the Author

Dennis Ludwig is a computer engineer at the Simulation and Analysis Facility, Aeronautical Systems Center at Wright-Patterson Air Force Base, Ohio. He has worked with software for more than 20 years. He has a bachelor's of science degree in electrical engineering from Louisiana Tech University, a master's of science degree in administration from Georgia College, and a master's degree in engineering from Mercer University.

ASC/HPMI
2210 5th Street Bldg. 146
Room 122B
Wright-Patterson AFB, OH 45433
Phone: (937) 255-7887
DSN: (785) 255-7887
E-mail: dennis.ludwig@wpafb.af.mil

SEPR and Programming Language Selection

Richard Riehle
Naval Postgraduate School

When former Assistant Secretary of Defense for Command, Control, Communications, and Intelligence Emmett Paige issued a memo in 1996 abrogating the Department of Defense's single-language policy, he included a clause designating programming language selection as a part of the Software Engineering Process Review (SEPR). The intent of his memo and its realization have not yet converged. Many readers of the memo mistakenly assumed his intent was a license to abandon Ada rather than advice to determine language selection as part of a rational evaluation step in the SEPR. This article addresses that confusion. Many of the references to Paige in this article originate in private conversations and correspondence between Paige and the author.

Former Assistant Secretary of Defense for Command, Control, Communications, and Intelligence Emmett Paige issued a memo in 1996 abrogating the Department of Defense's (DoD) single-language policy. His memo included a clause designating programming language selection as a part of the Software Engineering Process Review (SEPR). Many who read the memo mistakenly assumed his intent was a license to abandon Ada rather than advice to determine language selection as part of a rational evaluation step.

As a consequence of misinterpreting Paige's memo, the DoD is retrogressing toward the situation prior to 1983; a period sometimes described in *Tower-of-Babel* terms. Projects are adopting a language *du jour* policy that is destined to restore the havoc experienced prior to the single-language policy. For example, one DoD software group has replaced all of its Ada code with Perl. While Perl is a perfectly good scripting language, the decision to use it instead of Ada for this application represents a substantial failure of management to understand its long-range responsibilities. Program managers are once again required to cope with a multiple-language policy rather than a single-language policy.

When the Ada mandate was relaxed, some in the software community asked, "If the DoD cannot manage a single-language policy, how can it be expected to manage a multiple-language policy?" Although it was not explicitly stated in his memo, Paige did not want a return to the days of more than 400 DoD programming languages. He included the SEPR clause intending to provide guidance for this new multi-language policy.

Most DoD program managers are unprepared to make decisions regarding language choice. They are at the mercy of each contractor. In the future, we will see DoD software written in Java, C++, C#, Ada, Eiffel, Ruby, Perl, Python, Smalltalk, Fortran, COBOL, Euclid, Lisp, Prolog, Haskell, or even some proprietary languages invented by contractors.

Compilers for these languages, with the exception of Java and Ada, implement dialects that fail to correspond to a published standard. Gone are the days when a program manager would be able to insist on a validated compiler for the chosen programming language. No one even pretends that C++ compilers can be validated. In my conversations with program managers, I find that many have simply abandoned the language decision to the contractor. While this is expedient in the short term, it is likely to create major problems in the future. The Tower of Babel is slowly being rebuilt.

> *"Program managers are once again required to cope with a multiple-language policy rather than a single-language policy."*

More Mature Selection

At first, many in the software community were surprised that someone with Paige's appreciation of Ada would issue a memo abrogating its mandate. In meetings with Ada advocates subsequent to the SEPR memorandum, Paige emphasized several key points. He noted that, with more than 50 million lines of Ada code deployed in operational weapons systems, Ada had proven it could do the job it was intended to do. He also noted that, instead of unifying the software community within the DoD, Ada had become a rallying point for bickering among contractors and military officials. Its image was being tarnished through flurries of e-mail and other correspondence, often by some of the very people who were its advocates. Paige came to believe that Ada was good enough to stand on its own against alternative choices.

His decision coincided with the advent of the Ada 95 standard. In 1995, Ada changed from a military standard (MIL-STD 1815A) to an ISO/ANSI standard (ISO8652-1995), which made Ada 95 a powerful language for real-time, embedded object-oriented systems. Those who discovered that fact are enjoying the benefits of Ada while those who have chosen error-prone languages such as C++ have a long struggle ahead of them. The DoD will not be the beneficiary of that struggle.

Paige envisioned the SEPR as an essential part of any DoD software engineering effort. We know that successful software engineering projects start at a high level of abstraction. Before adopting tools, languages, and methods, we need to answer a question at the highest level of abstraction: "What problem are we trying to solve?"

As a project moves forward, even under agile processes, there is a succession of reviews to decide on methods, tools, and languages. We use the plural of language because more and more contemporary projects are implemented in multi-language environments. We would not choose C++ where HTML is appropriate nor would we choose Ada where we could more effectively use MATLAB. Java would be inappropriate for high reliability mathematical applications but might be perfect for displaying some of the results of those computations. XML is of growing importance, and XML works well with both Java and Ada.

In other words, we select the appropriate language for the problem to be solved. This is analogous to selecting the right tool for the job at hand. A pipe wrench does a different job than a box

wrench. When there is a chance we might break off the head of a bolt with a long-handled wrench, we use a torque wrench. Of course, we must know something about torque wrenches before we use them.

Paige's SEPR memo suggests that programming language selection should be a carefully considered process with the decision made on the basis of criteria derived from the project requirements. Although it was noted earlier that Paige was responding to a controversy, it was that controversy that led to his realization that the DoD needed a more mature approach to programming language selection. This, in turn, led him to the decision to include programming language choice in the SEPR.

One important benefit of abrogating the mandate has been the democratization of Ada. When the mandate was in place, Ada compiler publishers had the DoD as a captive customer. They could charge whatever they wanted for their technology. Prior to the Ada 95 standard, Ada compilers and tools were priced so commercial software developers could not afford them. When that captive DoD audience diminished, many Ada compiler publishers vanished or merged. In the absence of the mandate, compilers and tools, downloadable by anyone with access to the Internet, are now free.

Programmers worldwide are now experimenting with Ada. More non-DoD developers are quietly using it. Contemporary Ada has been adopted by the United States' friends and enemies. For example, Iranian and Chinese military software engineers are now using Ada. It is not as popular as C++ or Java, but it is equal to those languages in every way and better in some respects. At this stage, the cost of Ada technology should not be any greater than for C++ technology. Ada compilers are no more difficult to create than C++ compilers. They can be hosted on any computer in existence or being planned.

Programming languages are designed according to different goals. The SEPR must consider its language choice with an understanding of those goals, along with other factors. Those other factors include mission requirements such as targeted platform, expected level of dependability, maintenance ease, compiler availability, development tools, and environments as well as others. The factors should be determined by defining the criteria appropriate to the software product requirements.

Too often, language is chosen by the programmers, the contractor, or through some *ad hoc* decision-making process that has little to do with the underlying requirements. It is not unusual to see programmers make the language decision in pursuit of career goals. Do not let the programmers decide what language will be used for a sensitive DoD project. Long after the original programmers are gone, the software will require continued maintenance. Language choice must be a management decision based on what is best for the long-term health of the final software product.

It is rarely difficult for programmers to learn the language needed for the project. Ada, well taught, is as easy to learn as any other contemporary language. Our tools, including our programming languages, must help us meet the mission requirements with minimal error and maximum reliability.

"Too often, language is chosen by the programmers, the contractor, or through some ad hoc decision-making process that has little to do with the underlying requirements."

Reliability is an essential criteria for DoD weapons systems. A pilot attempting a carrier landing will be greatly annoyed if greeted by a heads-up display that announces, "Sorry. System error occurred. Please reboot." That is the kind of thing that can happen if we make the wrong choices.

So how should the programming languages be selected during the SEPR? I believe the answer lies in ideas: context and criteria. Can criteria be defined in the context of the future product? Lloyd Mosemann, senior vice president of Corporate Development for Science Applications International Corporation and former deputy assistant secretary of the Air Force for Communications, Computers, and Logistics, often cites predictability as an essential characteristic of DoD software. Predictability is an essential property of any engineering effort. Software engineering is not any different.

Can DoD program managers give guidance and direction about programming language choice? Should they simply provide context and criteria and let the software contractor choose the programming language? I believe the program manager, representing the DoD, should have a role in the programming language decision. The sad fact is that I see many contractors making the programming language choice on the basis of convenience, résumé building, and other factors that have little to do with product quality. Some consultants make recommendations based on poorly chosen criteria. Worst of all, language choices are made on the basis of what is currently popular rather than on what is best for a particular project.

Language Options

Because so many languages are available, this article focuses on object-oriented programming (OOP) languages. There are many OOP language choices available, including Smalltalk, C++, C#, Java, Ada, Eiffel, Modula-3, and Object COBOL. This list could be longer, especially if it included some of the excellent new languages, such as Ruby, or discussed the value of functional languages such as Objective CAML and Haskell.

Each language has benefits in given application domains. Some are better for one domain than another. Advocates will make the case that a favorite from the list is great for every kind of application, but that claim must be supported by the criteria declared for the targeted domain.

The language decision must recognize the difference between two issues: *expressibility* and *expressiveness*. Nearly every programming idea can be expressed in your favorite language. *Expressiveness* is about how well a language maps its solution space to the problem space. The ability to express an idea is called *expressibility*. The ease of expressing that idea is called expressiveness. For example, we might be able to compute Bessel functions in Lisp, but Fortran better expresses mathematical functions than Lisp. Lisp is more expressive of ideas related to artificial intelligence.

A DoD contractor for whom I once worked was assigned to create materials management software for a specialized Navy environment. At that time, the contractor was a Fortran programming shop. Some members of our team, comfortable with Fortran, insisted we could do the entire project in Fortran. From the perspective of *expressibility*, they were right. Others on the team made the case for COBOL. The COBOL advocates correct-

ly pointed out that COBOL was designed to express exactly this kind of application. The Fortran advocates correctly pointed out that they could do anything in Fortran that the other group could do in COBOL. The COBOL advocates won the day. The deciding factor was one of expressiveness over *expressibility*. Although Fortran could express the required programming solutions, management decided that COBOL was more expressive of those solutions.

Expressiveness is one of the earliest issues to consider before choosing other criteria. However, one needs to exercise some care about this. Some special purpose languages are expressive for specific applications, but targeted so narrowly they fail to meet other requirements. Also, many expressive languages are proprietary, useful only on one operating system, or poorly supported. For example, Visual BASIC is popular for programming on Microsoft platforms but is nonportable when considering other environments.

The program manager and the contractor together formulate the relevant criteria. Are the applications computational/numerical? Is nonportable software OK? Must we be able to deploy on multiple operating systems? Will this application require a lot of string manipulation? Is there an embedded real-time requirement? Must we interface with other languages? Is the application short-lived or long-lived? What is the cost of a software failure? Do we expect a lot of enhancements during the life cycle of this product? Is this a graphics product? What are the human-machine interface requirements? What are the software architecture considerations? Can we get efficient code from the available compilers? Is there a technology transition cost? The list of possible questions continues.

As you develop criteria, try to include a numerical weight and rating. This is a good place to use a spreadsheet listing languages and criteria with weights for the various criteria (see Table 1). Have more than one person assigning the scores on separate versions of the spreadsheet. You will be surprised by the disparity of viewpoints. Gather those involved in the scoring process and encourage a discussion that excavates biases, predilections, and perversions that may have influenced each person's scoring. The final score on the spreadsheet should be one of your indicators, but not the only indicator.

Language Criteria

Consider Table 1 in which sample criteria are weighted in favor of a safety-critical weapon systems. The entries will vary according to each kind of system. To avoid introducing too much bias into the chart, and to acknowledge that projects will evaluate criteria differently, I have masked the names of the languages. Your evaluation would insert the languages of interest in the spreadsheet.

		Language					
Criteria	**Weight**	**A**	**B**	**C**	**D**	**E**	**F**
Object-Oriented Programming (OOP)	3	3	3	4	5	4	5
Built-in Concurrency	5	5	0	3	2	0	0
Safety-Critical Features	5	5	3	3	4	3	1
Ease of Learning	3	3	2	4	4	4	5
Java Virtual Machine	2	3	0	5	2	0	0
Portability	4	4	3	4	4	3	4
Relevant Component Libraries	4	4	4	4	4	4	2
Open Source Compilers	2	4	4	2	2	2	5
Pre-, Post-, and Invariant Assertions	3	3	1	4	5	2	2
Type Safety	4	5	3	4	4	4	2
Development Tools	3	3	4	5	5	3	2
Language Maturity	4	4	4	3	4	1	1
Market Penetration	3	2	5	5	1	0	1
Microsoft Windows	2	4	5	1	4	3	3
Linux	4	4	3	1	4	1	5
Other Unix	4	4	3	1	4	1	5
Generic Templates	4	5	5	0	4	0	0
ISO/ANSI Standard Compliance	5	5	4	0	0	0	0
Interoperability	4	5	4	3	3	2	1
Raw Totals:		75	60	56	65	37	44
Weighted Totals:		279	214	192	230	128	146

Table 1: *Language Criteria Spreadsheet*

Let me emphasize that Table 1 will look different after you have combined the scores from more than one person. Where the example shows a score for OOP of five for Language D and four for Language E, someone else might score these differently. Also, someone might take issue with a score of one for Language E on Microsoft Windows. Even if no one argues about zero for built-in concurrency, they might argue that external (operating system-based) concurrency is a close enough equivalent.

Some Language Choices

Some of the OOP language choices were named earlier. The following sections contain a few pros and cons of some languages. I have chosen to mention Smalltalk, C++, Ada, Java, Eiffel, and Object COBOL. If there were enough space, I would have included some other favorites such as Modula-3, Haskell, and Ruby. Also, logic languages such as Prolog often have an important place in DoD applications. While this is all a matter of opinion, it is opinion derived from experience as well as study of the given examples.

Smalltalk

This is still the gold standard for OOP. Whenever someone writes about OOP, they compare their favorite language to Smalltalk. It is fun to program in Smalltalk. Although it has fallen out of popularity during the past several years, it comes with a powerful development environment, a large selection of libraries, and a small but powerful collection of features for building interactive applications. It is not a type-based language and does less compile-time checking than other languages. It is excellent for applications where dynamic binding is beneficial. It is not appropriate for safety-critical or embedded weapon systems applications. Smalltalk is portable enough for most situations. I personally like Smalltalk, but must realistically acknowledge its limitations.

C++

This language gained a large following during the 1990s. It is losing some ground to Java. C++ has both severe critics and committed advocates. It is a general purpose OOP language that became an ISO standard in the late 1990s. Few compilers support the full ISO standard so develop-

ers often design around a subset of the C++ language to achieve portability. The language has a type system built on predefined primitive types and designer-defined classes. A well designed class is supposed to behave like a predefined type.

The C++ type system has weaknesses. Among these are the notion of structural equivalence, the potential for unruly pointers, and the excessive reliance on predefined types as primitives. Typecasting in C++ is fraught with potential dangers. C++ has borrowed a number of features from Ada, including genericity, exception handling, and dynamic memory allocators.

It is a satisfactory language for noncritical software such as windowing applications and graphics. It is a poor choice for any kind of safety-critical application. It is probably a terrible choice for weapon systems, radar, space applications, or flight-control software. Validation suites for C++ do exist. Unlike Ada, C++ is held to a lower standard and no program manager or contractor ever suggests requiring a validated compiler.

Ada

The ISO 1995 Ada standard is a great improvement over the original 1983 standard. The primary goal of Ada is to maximize the amount of error detection a compiler can perform as early in the development process as possible. This has led to the strongest type-safety model of any contemporary language supplemented with a set of rules for scope and visibility that prevents name collisions, structural equivalence problems, dangling pointers, pointers to nonexistent data, along with many other problems. Ada supports several levels of object technology, including inheritance, polymorphism, dynamic binding, and genericity. However, OOP is optional and can be avoided when inappropriate for a particular application. Ada supports built-in concurrency and embedded real-time systems.

It is still the best choice for DoD software such as safety-critical, real-time weapon systems and flight-control software. For interoperability, Ada 95 is probably the most hospitable language of all. It directly interfaces with several legacy languages as well as with C++ and XML. Ada compilers used for DoD software are held to a higher standard than any other language. That is, the compilers for Ada are always required to pass the validation suite before they can be used in DoD applications. Safety-critical applications continue to be best served by Ada.

Java

Not since PL/I has a language been introduced into the marketplace with so much hyperbole. Java is a language that promises to become better with time. At the elementary level, it is comparatively easy to learn. It is, in many respects, safer than C++. Indirection in Java does not use computational pointers, thereby reducing some of the risks encountered with C++. It also has automatic garbage collection, which some see as a blessing and others as a curse. Java consists of three basic parts: the language, the libraries, and the Java Virtual Machine (JVM). The most important contribution of Java is not the language. The language actually contributes very little new and actually represents a step backward in some respects. Rather, the important thing about Java is the libraries and the JVM. In particular, the JVM permits a high level of portability for compiled applets.

Java includes a capability for concurrency but falls short of the concurrency model of Ada. Java provides none of the deterministic behavior expected for a hard, real-time software environment. Like Smalltalk, Java is fun for programming, largely because it appeals to the instant gratification that entices so many of us. The language includes a disclaimer that it is not intended for safety-critical applications. Do not use it for weapon systems or applications where high reliability is required. Java is not yet a standard. It seems to be an evolving product that will probably stabilize in the next couple of years.

"With the abrogation of the DoD's single language policy, we need to take care not to fall into the trap of avoiding Ada or becoming too attached to it."

Eiffel

This is a relative newcomer, but older than Java. It is everything one could get from Java, except the bytecode. Grady Booch, chief scientist at Rational Software Corporation, in an off-the-cuff remark at a software conference said of Eiffel, "Eiffel is what C++ could have been if C++ had not been dependent on C."

The language permits a stronger typing model than Smalltalk but still emphasizes a pure OOP approach. For applications programming it is a better choice than C++ because it permits a more natural form of expressiveness than C++. When considering type safety, Eiffel is probably more reliable than C++. Eiffel includes a powerful development environment along with a full library of generic reusable components. Eiffel does not enjoy the large audience of users it deserves. Most Eiffel compilers are C Path, meaning they generate intermediate C code. Eiffel also has a built-in capability for programming with assertions. Assertions are pre-, post-, and invariant conditions that can be applied at many levels within an Eiffel module. The designer of the Eiffel language, Dr. Bertrand Meyer, calls the use of this feature *design-by-contract*. None of the other languages mentioned so far are as robust in this regard. Even Ada, which supports a kind of range constraint assertion, does not yet support assertions such as those found in Eiffel.

Eiffel is still not my first choice for safety-critical weapons systems, but it is probably a better choice than either C++ or Java. There is no ISO standard for Eiffel, but there is an international body called Network Information and Control Exchange that oversees its progress. A program manager once told me, "The only two languages I would consider are Ada and Eiffel." If you have not yet looked at Eiffel, you might consider doing so.

Object COBOL

If you are currently programming in COBOL, Object COBOL makes sense. Many COBOL shops are making the mistake of converting to Java, or worse, C++. The syntax of Object COBOL will be familiar to your programmers. They can learn OOP on the job using this familiar syntax. Everything you liked about COBOL is still there, but you can enjoy inheritance, information hiding, encapsulation, polymorphism, dynamic binding, and everything else expected from an OOP language. Contemporary COBOL is a language with expressive power required for business and business-like applications. It includes features that will make your current COBOL programmers and systems analysts more productive than they would be after retraining in some other language. Object COBOL can

improve communications between clients and systems analysts as well as between systems analysts and programmers. Object COBOL is not appropriate for safety-critical weapons systems, but it is an excellent step into the future if you are already in a COBOL programming environment.

Summary

The programming language selection process for DoD software systems is too often made on the basis of inadequate criteria. With the abrogation of the DoD's single language policy, take care not to fall into the trap of avoiding Ada or becoming too attached to it. It is important to recognize the strengths and weaknesses of more popular languages such as C++ and Java, and understand when to choose them and when to reject them.

Paige's SEPR memorandum suggests a direction without specifying too many details. As we implement his suggestions, we must do so on the basis of carefully defined criteria and with sufficient knowledge to understand the contribution of each of the alternatives in terms of those criteria. Also, selecting the language is not sufficient. We must insist that the compiler for the language we choose conforms to the highest possible set of standards available. Unlike commercial software, the safety of our uniformed personnel and the success of our military missions depend of the reliability of the choices we make.◆

About the Author

Richard Riehle is a visiting professor at the Naval Postgraduate School. He also owns AdaWorks, a small company dedicated to Ada consulting and training. His book "Ada Distilled" may be downloaded free from <www.adaic.org>.

Naval Postgraduate School
Computer Science Department
Spanagel Hall
Monterey, CA 93943
E-mail: richard@adaworks.com or rdriehle@nps.navy.mil.

WEB SITES

Data and Analysis Center for Software

http://dacs.dtic.mil

The Data and Analysis Center for Software (DACS), a Department of Defense (DoD) Information Analysis Center, is designated as the DoD software information clearinghouse, serving as an authoritative source for state-of-the-art software information and providing technical support to the software community. From its home page, the user can access more than 30 specific technical topic areas related to software engineering and software technology, including programming languages. The DACS offers a wide-variety of technical services designed to support the development, testing, validation, and transitioning of software engineering technology.

JOVIAL Lives

www.jovial.hill.af.mil

JOVIAL Lives is the official home page of the U.S. Air Force JOVIAL program office, whose mission is to provide current and future customers with superior service, support, and distribution of the best JOVIAL compilers available and JOVIAL/MIL-STD-1750A Integrated Tool Sets (ITS), which are software support tools used for development and maintenance of MIL-STD-1750A target applications. In compliance with the Air Force's policy of software reuse, this office provides a cost-effective way to maintain and modernize existing quality software products.

Ada Power

www.adapower.com

Ada Power developer resources and tools Web site was formed to contribute back to the Ada community and to help advocate this powerful language. The site features examples of Ada source code, including illustrating various features of the language and programming techniques, various interfaces to popular operating systems, various algorithms, a collection of packages for reuse in Ada programs, inplementation articles, numerous Ada links, and more.

Computing Research Association

www.cra.org

The Computing Research Association (CRA) includes more than 200 North American academic departments of computer science, computer engineering, and related fields; laboratories and centers in industry, government, and academia engaging in basic computing research; and affiliated professional societies. CRA's mission is to strengthen research and education in the computing fields, expand opportunities for women and minorities, and improve public and policy-maker understanding of the importance of computing and computing research in our society.

Ada Information Clearinghouse

www.adaic.org

The Ada Information Clearinghouse (AdaIC) was formed to ensure continued success of Ada users and promote Ada use in the software industry. The AdaIC has served a community of software engineers, managers, and programmers for more than 15 years. The Web site provides articles on Ada applications, databases of available compilers, current job offerings, and more. The AdaIC is managed by the Ada Resource Association, a group of software tool vendors that supports the use of Ada for excellence in software engineering.

IEEE Computer Society

www.computer.org

With nearly 100,000 members, the International Electrical and Electronics Engineers Computer Society (IEEE-CS) is the world's leading organization of computer professionals. Founded in 1946, it is the largest of the IEEE's 36 societies. The IEEE-CS's vision is to be the leading provider of technical information and services to the world's computing professionals.The Society is dedicated to advancing the theory, practice, and application of computer and information processing technology.

International Standardization in Software and Systems Engineering

François Coallier
École de technologie supérieure

The last decade saw an increase in the phenomenon of globalization and concurrently the pervasiveness of computing in our society. This article provides an introduction to international standardization in information technology, gives a status and describes current activities in international software and systems engineering standardization, and explains why all of these are important for professionals and organizations in this area.

The last decade saw an increase in the phenomenon of globalization and concurrently the pervasiveness of computing in our society. Products based on the information and communication technologies (ICTs) and software intensive systems are now ubiquitous in industrialized societies, whether for commercial, industrial, defense, or domestic applications. The global information technology (IT) procurement industry, which includes telecommunication equipment, computer systems hardware, software licenses, semiconductors, and IT services, should now be around $1.4 trillion, according to Gartner Dataquest [1].

As a direct result of the use of computerized devices, the world is now very dependent on software systems. ICT-based products are software-intensive systems and the software in them is essential to their functioning.

The ability to design and implement ICT systems and products has greatly improved in the last 10 years. A recognized core body of knowledge in software engineering now exists – a sign that software engineering is maturing into a recognized profession. Challenges still abound because of the pressure to build even more complex applications and products in an ever-shorter time frame (a *Web year* is three months) [2].

In response to these market needs, there has been significant development in international standards in software and systems engineering in the last decade. This article will give an overview of these developments as well as the context in which they are happening.

The International Standardization Context

Standards are essentially either a *de jure* (formal) or a *de facto* (current state of things) mandatory set of conventions and/or technical requirements or practices [3]. Standards can be classified into the following categories:

- Organizational standards such as internal company standards.
- Market standards (de facto) such as the VHS format.
- Professional standards developed by professional organizations such as the Institute of Electrical and Electronics Engineers (IEEE).
- Industry standards developed by industrial consortia such as the World Wide Web Consortium, and the Organization for the Advancement of Structures Information Standards.
- National standards developed or adopted by national standards organizations such as the American National Standards Institute.
- International standards developed or adopted by formal international standards organizations such as ISO.

A given standard may be developed in one environment (market, professional, industry, or national) and migrate into a formal international standard. Market, professional, and industry standards may also represent an international consensus or de facto state. The difference with the formal international standards is in the degree of the breadth and formality of this consensus. This will become clearer later in the article.

Formal international standards in the ICT are developed by the following organizations:

- International Telecommunication Union, founded May 17, 1865. This is the international organization within the United Nations System where governments and private sectors coordinate global telecom networks and services.
- ISO (International Organization for Standardization), founded in 1947. The mission of ISO is to promote the development of standardization and related activities in the world with a view to facilitating the international exchange of goods and services, and to develop cooperation in the spheres of intellectual, scientific, technological, and economic activity.
- International Electromechanical Commission (IEC), founded June 1906. This is the leading global organization that prepares and publishes international standards for all electrical, electronic, and related technologies.

In 1987, ISO and IEC joined forces and put in place a Joint Technical Committee 1 (JTC 1) with the following mandate:

Table 1: *Current JTC 1 Subcommittees* (**Note:** 10 Jan. 2002, taken from <www.jtc1.org>)

Technical Areas	JTC1 Subcommittees and Working Groups
Application Technologies	SC 36 – Learning Technology
Cultural and Linguistic Adaptability and User Interfaces	SC 02 – Coded Character Sets SC 22/WG 20 – Internationalization SC 35 – User Interfaces
Data Capture and Identification Systems	SC 17 – Cards and Personal Identification SC 31 – Automatic Identification and Data Capture Techniques
Data Management Services	SC 32 – Data Management and Interchange
Document Description Languages	SC 34 – Document Description and Processing Languages
Information Interchange Media	SC 11 – Flexible Magnetic Media for Digital Data Interchange SC 23 – Optical Disk Cartridges for Information Interchange
Multimedia and Representation	SC 24 – Computer Graphics and Image Processing SC 29 – Coding of Audio, Picture, and Multimedia and Hypermedia Information
Networking and Interconnects	SC 06 – Telecommunications and Information Exchange Between Systems SC 25 – Interconnection of Information Technology Equipment
Office Equipment	SC 28 – Office Equipment
Programming Languages and Software Interfaces	SC 22 – Programming Languages, Their Environments and Systems Software Interfaces
Security	SC 27 – Information Technology Security Techniques
Software Engineering	SC 07 – Software and Systems Engineering

> "Standardization in the Field of Information Technology: Information technology includes the specification, design, and development of systems and tools dealing with the capture, representation, processing, security, transfer, interchange, presentation, management, organization, storage, and retrieval of information" [4].

JTC 1 presently consists of of the subcommittees lists in Table 1.

International standards can come into being through different processes:

- As a proposal that is developed in working groups through the *standard* six-stage process described in Table 2 (three to five years from initiation to publication).
- As a proposal with a base document that can be internally *fast-tracked*, e.g., processed through a compressed schedule (about two years).
- As a proposal with a complete document that can be fast-tracked by JTC 1 (one four-month ballot, less than one year).
- As a proposal with a complete document that can be proposed by external (but recognized) organizations and fast-tracked as a four-month ballot known as the Publicly Available Standard process (one to two years).

It is a misperception that the development of formal international standards always takes an exceeding amount of time. When this is the case, it is usually due to one (or a combination) of the following reasons:

- The topic is new thus it takes time to develop a unified international view.
- International consensus on the topic is weak due to positions that are difficult to bring together.
- Management of the development process is suboptimal.

This brings us to another key concept in standardization work: the notion of *consensus*. Standards represent a consensus, and the essence of the ISO standard process is the achievement of a proper level of consensus. ISO defines consensus as the following:

> "General agreement, characterized by the absence of sustained opposition to substantial issues by any important part of the concerned interests and by a process that involves seeking to take into account the views of all parties concerned and to reconcile any conflicting arguments" [5].

This essentially means the following:

- That all the parties involved were able to voice their views.
- That the best effort was made to take into account all of the above views and resolve all issues (meaning all comments tabled during a ballot).
- That nearly all or (ideally) all the parties involved can live with the final result.

As ISO notes in its guidelines, consensus does not mean unanimity. The minimal numerical for international standards adoption in ISO is a two-thirds majority of the participating members (e.g., countries) voting. For technical reports (e.g., guides), it is a simple majority.

So what added value do international standards bring in addition to a well-known brand? They bring the following:

- They represent an international consensus attained through a very rigorous and uniform process.
- They usually represent a set of conventions and/or technical requirements or practices that are relatively stable.

In addition, the international standardization process makes it relatively difficult and costly for special interest groups to take over a given standardization project, especially if the topic is controversial. This would mean controlling many country delegations, as well as liaison organizations. For a project to be accepted in ISO, at least five countries must be willing to contribute experts. In Subcommittee 7 (SC7) of JTC 1, there are currently 27 participating countries.

Evidently, nothing is perfect. Industry and de facto standardization dominate in some fast-evolving ICT areas. The international standardization process is not built to accommodate the requirements in these areas, especially when the technology and the market are unstable. On the other hand, once things stabilize, industry standards developed by an industrial consortia (for example, the object management group or OMG) should be able to migrate to the formal international scene using one of the compressed processes presented earlier in this article. This has been happening in the ICT since JTC 1 was created.

Stage No.	Stage Name	Stage Description
0	Preliminary	A study period is under way.
1	Proposal	A new project is under consideration.
2	Preparatory	A working draft is under consideration.
3	Committee	A committee draft /final committee draft is under consideration.
4	Approval	A final draft international standard is under consideration.
5	Publication	An international standard is being prepared for publication.

Table 2: *Standard Six-Stage Process for the Development of International Standards*

International Software and Systems Engineering Standardization

The SC7 has the mandate within JTC 1 for, as described in its terms of reference, standardization of processes, supporting tools, and supporting technologies for the engineering of software products and systems.

The origins of SC7 go back to ISO/Technical Committee (TC) 97, initiated in 1960 for international standardization in the field of *information processing*. When JTC 1 was established in 1987, ISO/TC97 was combined with IEC/TC83 to form JTC 1/SC7, with *software engineering* as its initial title and area of work. This was extended to software and systems engineering in 2000.

There are currently 69 published international standards under the responsibility of SC7. By early 2004, this will rise to 81 if the work proceeds as planned. As illustrated in Figure 1 (see page 20), the availability of so many international standards in software and systems engineering is a recent occurrence.

In 1990, only eight standards were under the responsibility of SC7. One of the eight standards was on software documentation (ISO/IEC 6592, last revised in 2000), the balance being diagramming and charts standards, six of them still existing as *legacy* standards. At the same time, the IEEE already had a substantial collection of 14 software and systems engineering standards, a collection that grew to 27 in 1994 and presently stands at about 50 standards. To get the complete picture, it is good to keep in mind that the first software engineering standard was a U.S. military standard in 1974, and the first IEEE software engineering standard was published in 1979 (software quality assurance plans) [6].

The increase in the number of published international standards in software and systems engineering in the last 10 years is due to the following:

- The increased dependencies of our global society and economy on the ICT and software intensive systems.
- The maturing of the software and systems engineering profession in the 1990s due to the work of professional organizations such as the IEEE, the European Strategic Program for Research in Information Technology (ESPRIT) projects, and Japanese IT research initiatives to name a few.
- The dedication of all the technical

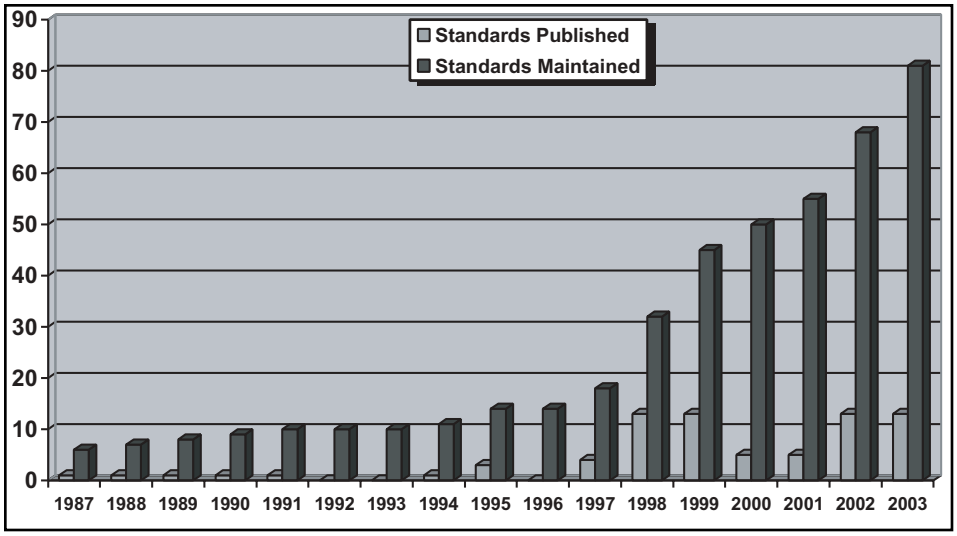


Figure 1: *Evolution of Published International Standards in Software and Systems Engineering*

experts and professionals who work on these standards.

In background to all this – as mentioned at the beginning of this article – is the very significant expansion of the ICT market during the 1990s, driven by Moore's Law[1] and the Internet.

Another evolution took place in the standardization area. The Computer Society of the IEEE (IEEE-CS) saw its membership become more international, with more than half now coming from outside the United States. The IEEE-CS has adopted key international standards from SC7 such as ISO/IEC 12207 (available in an IEEE edition). Also, as we will see in some examples that follow, IEEE standards are being considered by SC7. This means that the two sets of standards should become more integrated with time. SC7 and IEEE-CS are presently working together to become more systematic in their relationship.

The SC7 standardization portfolio can be presently divided into the following areas of work:

- **Legacy Standards.** These are essentially legacy information processing standards that SC7 still has in its portfolio (six standards).
- **Software and Systems Engineering Processes.** These are standards that describe good software and systems engineering practices, as well as standards to consistently assess organizational software and systems engineering practices against a given benchmark (19 standards, eight active projects).
- **Software Systems Products.** These are standards that allow developers, purchasers, and buyers to size and document software products, as well as to express, measure, and evaluate the quality of the software that is produced and its contribution to the final product or application systems (25 standards, six active projects).
- **Enterprise Architecture.** These are standards to integrate IT and business systems definitions and to provide the software and systems engineering tools to implement enterprise information systems (12 standards, nine active projects).
- **Software Engineering Environment.** These are standards that make it easier to use software-engineering environments and to reuse and re-deploy the data contained in them (two standards, one active project).
- **Software and Systems Engineering Formalisms.** These are standards for formal representations and modeling of software and systems (five standards, two active projects).
- **Software Engineering Body of Knowledge.** These are guidelines that establish the appropriate set(s) of criteria and norms for the professional practice of software engineering upon which industrial decisions, professional certification, and educational curricula can be based (one active project).
- **Management of Software Assets.** These are standards that will describe the basic requirements of a software asset management environment (one active project).

Let us look in greater detail into the last seven areas, focusing on key standards and projects.

Software and Systems Engineering Processes

Four standards are the cornerstones of this area:

- The ISO/IEC 12207 Software Life-Cycle Processes was published in 1995 and amended in 2002.
- The ISO/IEC 15288 Systems Life-Cycle Processes was published in 2002. It was developed with a strong participation of the International Council on Systems Engineering (INCOSE).
- The ISO/IEC TR 15504 Software Process Assessment series was published in 1998 and 1999 as technical reports. They are currently being revised, with their scope widened to cover any type of processes and upgraded to international standards. The Capability Maturity Model® Integration℠ (available through the Software Engineering Institute) is compatible with the current version of ISO/IEC 15504 [7].
- The ISO/IEC 9000-3 Guidelines for the Application of ISO 9001 to Computer Software was transferred to SC7 from another ISO committee (ISO/TC176) and is currently undergoing a revision to be aligned to the 2000 version of ISO 9001.

The relationship between these four standards is illustrated in Figure 2.

These key standards, including the recently published ISO/IEC 15288, are well known in the software and systems engineering community. Since these standards were developed on different timelines, it is normal that differences crop up among them. This is why a harmonization project between 15288 and 12207 is under serious consideration in SC7.

The following complements this top-level set of standards:

- ISO/IEC TR 15271 – Guide to ISO/IEC 12207.
- ISO/IEC 14764 – Software Maintenance.
- ISO/IEC TR 15846 – Configuration Management.
- ISO/IEC 15910 – Software User Documentation Process.
- ISO/IEC 15939 – Software Measurement Process.
- ISO/IEC TR 16326 – Guide for the Application of ISO/IEC 12207 to Project Management.

The following two standards should join this set by the third quarter of 2003:

- ISO/IEC TR19760 – Guide for ISO/IEC 15288.
- ISO/IEC 16085 – Risk Management.

It is interesting to note that the ISO/IEC 16085 is a fast-track of IEEE 1540:2001.

Software Systems Products

There are five main sets of standards in this area:

- The 9126 series on *software quality char-*

® Capability Maturity Model is registered in the U.S. Patent and Trademark Office.
℠ Capability Maturity Model Integration is a service mark of Carnegie Mellon University.

acteristics. The initial standard was published in 1992. This standard is currently being revised and expanded into a three-part document.

- The 14598 series on *software product assessment*. Initially published between 1998 and 2001, this six-part standard got significant inputs from the Software Certification Program in Europe, ESPRIT project of the early 1990s.
- The 14143 series on *functional size measurement*. This five-part standard publication will span from 1998 through early 2003.
- A group of four *functional size counting methods* standards (19761, 20926, 20968, 24570). These are in the final approval stage, three of which are fast-tracked through the PAS process.
- A block of *software and systems reliability* standards. These include the standard on Systems and Software Integrity Levels (15026) and the recently transferred project from IEC/TC 56 Guide to Techniques and Tools for Achieving Confidence in Software (IEC 16213).

The 9126 and 14598 standards are currently being integrated and reworked into the new 25000 series titled "Software Product Quality Requirements and Evaluation" (SQuaRE). The architecture of the SQuaRE standards is given in Figure 3.

Enterprise Architecture

The enterprise architecture standards of SC7 currently consist of a series of documents on open distributed architecture (principally 10746 and 13235 series, 14750, 14752, and 14753). This work is being carried cooperatively with the OMG, which is fast-tracking many of the documents. More details on the applications of these standards can be found in [10].

Software Engineering Environment

Published standards in this area cover the evaluation (14102) and the adoption of CASE tools (14471).

Software and Systems Engineering Formalisms

A key standard in this area is the Unified Modeling Language (19501) that is currently being fast-tracked from the OMG.

Software Engineering Body of Knowledge

This is a cooperative project with the IEEE-CS to publish their Software Engineering Body of Knowledge as an ISO technical report – ISO/IEC TR 19759 [11]. This project is near completion.

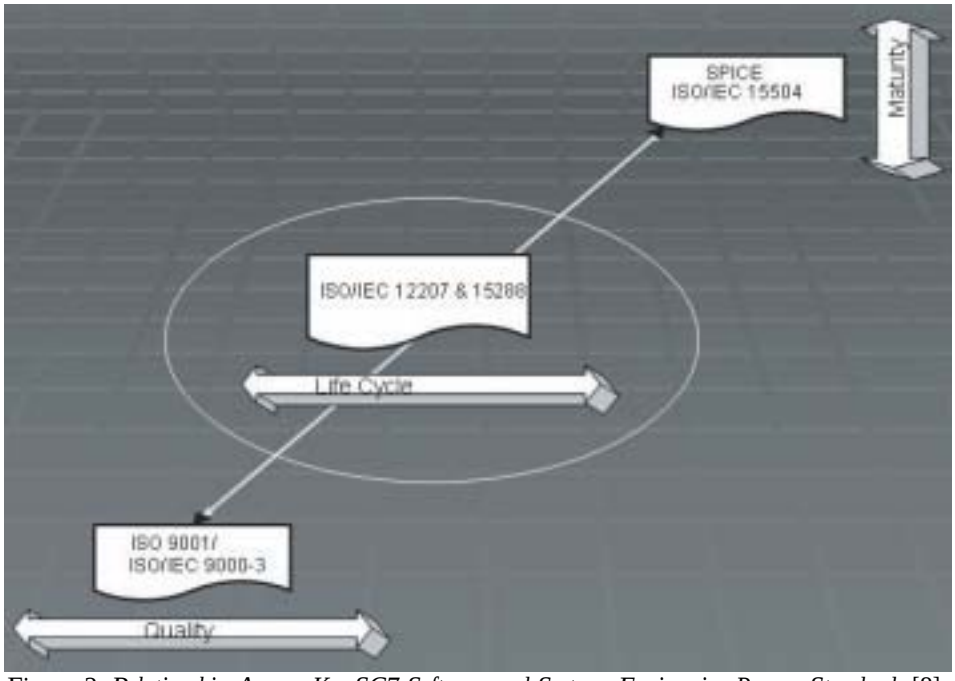


Figure 2: *Relationship Among Key SC7 Software and Systems Engineering Process Standards* [8]

Management of Software Assets

This is a new project (19770) initiated in 2001 that aims to develop a standard on a software asset management process. A first working draft of this standard has been published on the SC7 Web site.

Conclusions

The increase in international software and systems engineering standardization is a consequence of both the continuing growing importance of the ICT and the ICT-based systems, products, and services in the global economy as well as the growing maturity of the software and systems engineering disciplines.

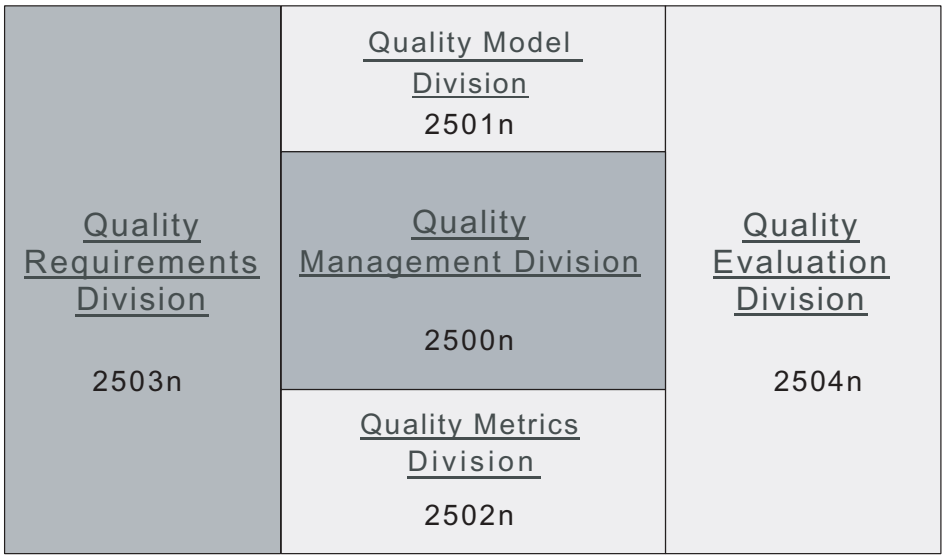
The SC7 will strive to fulfill its mandate and deliver to the international software and systems engineering community the tools it requires in the global information society. This will be done in cooperation with other standards-developing organizations, not only the national standards organizations but also, increasingly, professional and industrial ones.

Additional Information

The SC7 Web site <www.jtc1-sc7.org> provides more information. All JTC 1/SC7 standards can be purchased directly from ISO or from the American National Standards Institute at <http://webstore. ansi.org/ansidocstore/default.asp>.

It is necessary to be accredited with a national body to participate in the development of SC7 standards. In the United States, the contact for the U.S. Technical Advisory Group on software and systems

Figure 3: *SQuaRE Architecture* [modified from 9]



COMING EVENTS

February 24-27
Software Engineering Process Group Conference
Boston, MA
www.sei.cmu.edu/sepg/

February 25-26
Data Mining Tech. for Military and Government Applications Forum
Washington, D.C.
www.worldrg.com

March 10-13
Software Test Automation Spring '03
San Francisco, CA
www.sqe.com/testautomation/

March 24-28
International Symposium on Integrated Network Management
Colorado Springs, CO
www.im2003.org

March 31-April 2
ACDM's Annual Technical and Training Conference
San Diego, CA
www.acdm.org/main.htm

April 8-10
FOSE 2003 (Federal Office Systems Exposition)
Washington, D.C.
www.fose.com

April 28-May 1
Software Technology Conference 2003
Salt Lake City, UT
www.stc-online.org

May 3-10
International Conference on Software Engineering
Portland, OR
www.icse-conferences.org/2003

engineering is Mike Gayle at s.m.gayle@jpl.nasa.gov.

Acknowledgements
The author would like to thank Claude Laporte, James Moore, Perry DeWeese, Robert Frost, Gilles Allen, Alastair Walker, Hans Daniel, Dennis Ahern, Witold Suryn, and Alain Abran for reviewing this article and providing feedback.

References

1. R. Fulton. "Predicts 2002 – What's Ahead for the IT Industry?" Gartner Research, 1 Aug. 2002 <www.adabas natural4ever.com/industry_news/media/predicts_2002_whats_ahead_for_the_it_industry.pdf>.
2. Booch, Grady. "The Illusion of Simplicity." Software Development Feb. 2001 <www.sdmagazine.com/documents/s=734/sdm0102m/0102m.htm>.
3. Institute of Electrical and Electronics Engineers. Software Engineering Terminology. Adapted from IEEE Std. 610-12. New York, 1990.
4. ISO/IEC Guide 2:1996 <www.iso.org/sdis>.
5. Procedures for the Technical Work of ISO/IEC JTC 1 <www.jtc1.org>. Found under Procedures in the public area.
6. Moore, James W. Software Engineering Standards: A User's Road Map. ISBN 0-8186-8008-3. New York: IEEE CS Press, 1979.
7. Software Engineering Institute. Capability Maturity Model® Integration℠ Pittsburgh: Software Engineering Institute, Mar. 2002.
8. Subcommittee 7/WG7. "ISO/IEC 15288 Marketing Presentation." SC7 document N2646R <www.info.uqam.ca/Labo_Recherche/Lrgl/sc7/N2601N2650/07N2646R%20W07N06110Version%202%20%20IEC%2015288%20Marketing%20Presentation.pdf>.
9. Azuma, Motoei. SQuaRE: The Next Generation of the ISO/IEC 9126 and 14598 International Standards Series on Software Product Quality. Proc. of European Software Control and Metrics – Escom 2001, London, 2-4 Apr. 2001 <http://dialspace.dial.pipex.com/town/drive/gcd54/conference2001/papers/azuma.pdf>.
10. Putman, Janis R. Architecting with RM-ODP. New Jersey: Prentice Hall PTR, 6 Oct. 2000.
11. Abran, Alain, James W. Moore, Pierre Bourque, Robert Dupuis, and Leonard L. Tripp. Guide to the Software Engineering Body of Knowledge SWEBOK. ISBN 0-7695-1000-0. New York: IEEE CS Press, 2001.

Note
1. Moore's Law <www.webopedia.com/TERM/M/Moores_Law.html>.

Further Information
1. IEEE-CS Software Standards: <http://standards.computer.org/sesc>.
2. IEC: <www.iec.ch>.
3. INCOSE: <www.incose.org>.
4. OMG: <www.omg.org>.
5. ISO: <www.iso.ch>.
6. JTC 1: <www.jtc1.org>.
7. SC7: <www.jtc1-sc7.org>. Provides information on current SC7 activities, published standards, and more. Many documents are available for download.

About the Author

François Coallier is professor of software and information technology (IT) engineering at the École de technologie supérieure. He has nearly 22 years of industrial experience in one of Canada's largest companies, where he held various engineering and managerial positions in engineering, quality engineering, IT procurement, IT infrastructure deployment and operation, and IT enterprise architecture management. Coallier is currently the international chairman of the Joint ISO and International Electromechanical Commission (IEC) subcommittee responsible for the elaboration of Software and Systems Engineering Standards ISO/IEC JTC1/SC7, and also a fellow of the American Association for the Advancement of Science. He has a bachelor's of science degree in biology from McGill University, a bachelor's degree in engineering physics, and a master's of science degree in electrical engineering from Montreal's École Polytechnique.

Département de Génie Électrique/Department of Electrical Engineering
École de technologie supérieure
1100, rue Notre-Dame Ouest
Montreal, Quebec
Canada H3C 1K3
Phone: (514) 396-8637
Fax : (514) 396-8684
E-mail : fcoallier@ele.etsmtl.ca

An Enterprise Modeling Framework for Complex Software Systems

Dr. Paolo Donzelli
Office of the Prime Minister

This article presents a goal-oriented, agent-based Enterprise Modeling Framework where advanced requirements engineering techniques are combined with software quality modeling approaches to provide an environment within which the stakeholders and the analysts can easily cooperate to discover, verify, and validate the requirements for a new software system. The framework assists and drives the stakeholders to an early definition of the desired system functionality and quality attributes, while supporting the redesign of the encompassing organization to better exploit the new system's capabilities. Although beneficial to a wider class of software systems, the framework has been applied here to improve the requirements engineering process for synthetic environments, which are complex software-intensive systems that typically comprise distributed interactive simulations of real-world systems. They are increasingly used to support vitally important operational, political, and economic decisions in a variety of industrial and governmental settings.

Systems requirements define what the system is required to do and the circumstances under which it is required to operate. The branch of software engineering concerned with all the activities involved in discovering, documenting, validating, and maintaining system requirements is *requirements engineering* (RE). Since difficulties with requirements are still a major contributory factor to project failures, leading not only to late and over budget deliveries but often also to systems significantly different from the stakeholders' expectations, RE is one of the most crucial steps in system development.

As technologies advance, they allow designers to envision systems that are increasingly becoming integral parts of the encompassing organizational processes. As this occurs, attention is being more and more focused on the very early phases of RE. The development of a successful system, that is a system able to address the stakeholders' real needs and suitable to evolve to meet ever-changing organizations' demands, relies on a firm understanding of the organizational context in which the system has to function. In other words, the system and its context need to be treated as a larger social-technical system, whose overall needs are the ones to be fulfilled [1].

Consequently in RE, appropriate process modeling techniques are typically advocated [2, 3] to help understand the organizational context (as it exists), envision possible solutions (as they could exist with the new system in place), and compare feasible alternatives. Within such a perspective, this article introduces an Enterprise Modeling Framework (EMF) explicitly designed to support discovery, verification, and validation of both *user-oriented* and *organization-oriented* system requirements [4, 5] by assisting dialogue between the analysts and the stakeholders and negotiation among the stakeholders.

In the remainder of this article, the EMF is introduced and its main characteristics are briefly described. Next comes some extracts from a practical application, and the article concludes by discussing some of the observed benefits.

"As technologies advance, they allow designers to envision systems that are increasingly becoming integral parts of the encompassing organizational processes."

The EMF

The EMF is designed to allow analysts to deal with the *what* and *how* of the organizational context, i.e., the tasks performed by the organization, and the way in which they are performed. It also allows the analysts to model explicitly the why, i.e., the underlying reasons, expressed in terms of organizational goals. This enables the analysts and the stakeholders to focus on the *right* system for a given context, to design (or redesign) the encompassing process that fully exploits the system's capabilities [6], and to improve their capacity to identify, justify, and validate the system requirements [2, 3, 7].

In particular, the modeling effort is tackled by breaking the activity down into more intellectually manageable components, and by adopting a combination of different approaches on the basis of a common conceptual notation: *Agents* are used to model the organization, whereas *goals* are used to model agents' interactions.

According to the nature of a goal, we distinguish between *hard* goals and *soft* goals [2, 3]. For a hard goal, the achievement criterion is sharply defined (e.g., "buy a computer"); for a soft goal, it is up to the goal originator, or an agreement between the involved agents, to decide when the goal is considered to have been achieved (e.g., "buy a fast computer"). In comparison to hard goals, soft goals are highly subjective and strictly related to a particular context (what is meant by "fast computer?"). The EMF, therefore, spawns three interrelated modeling efforts: the organizational modeling, the hard goal modeling, and the soft goal modeling (see Figure 1 on page 24). In this way, separating goal modeling from organizational modeling helps reduce the problem complexity [4].

During *organizational modeling*, the organizational context is modeled as a network of interacting agents (any kind of active entity, e.g., teams, humans, and machines, one of which is the target system) [2, 8], collaborating or conflicting in order to achieve both private and organizational goals. Once identified, goals are translated and implemented through continuous refinement into operational terms as tasks with constraints.

The *hard goal modeling* seeks to determine how to achieve hard goals by decomposing them into more elementary subordinate hard goals and tasks, where tasks are well-specified activities that someone or something within the organization has to perform. So, for example, the hard goal "buy a computer" will be translated into a set of actions (tasks) necessary to procure a computer: organize a

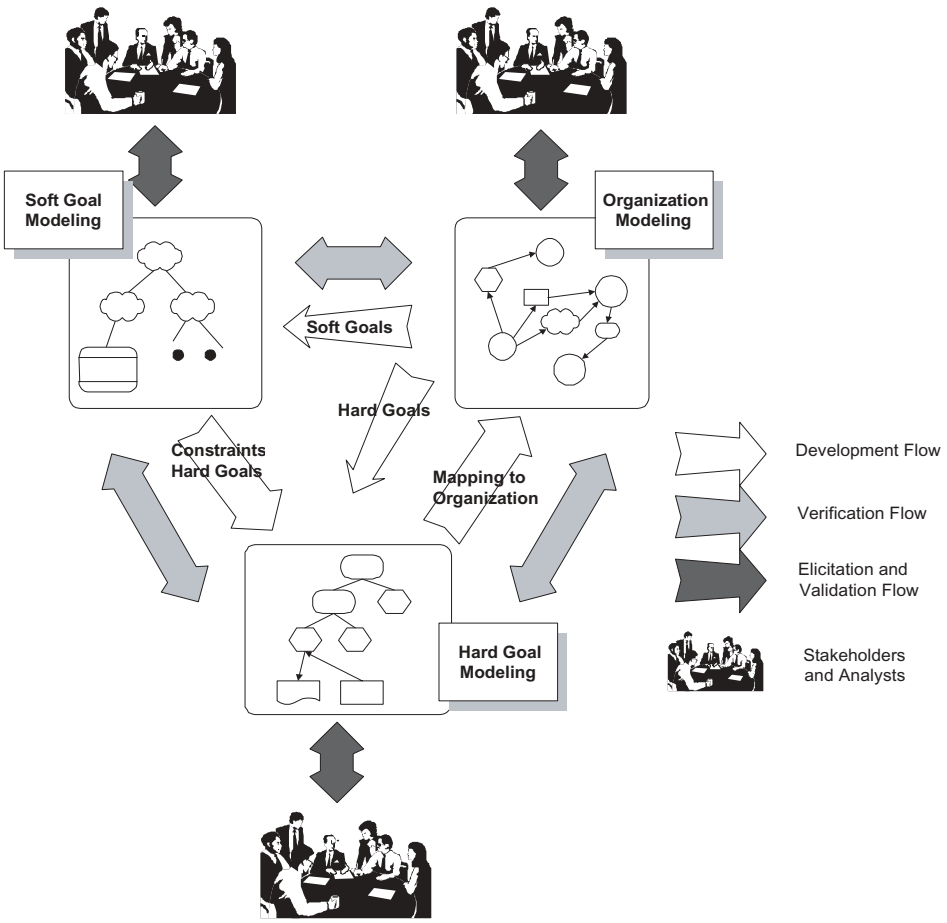
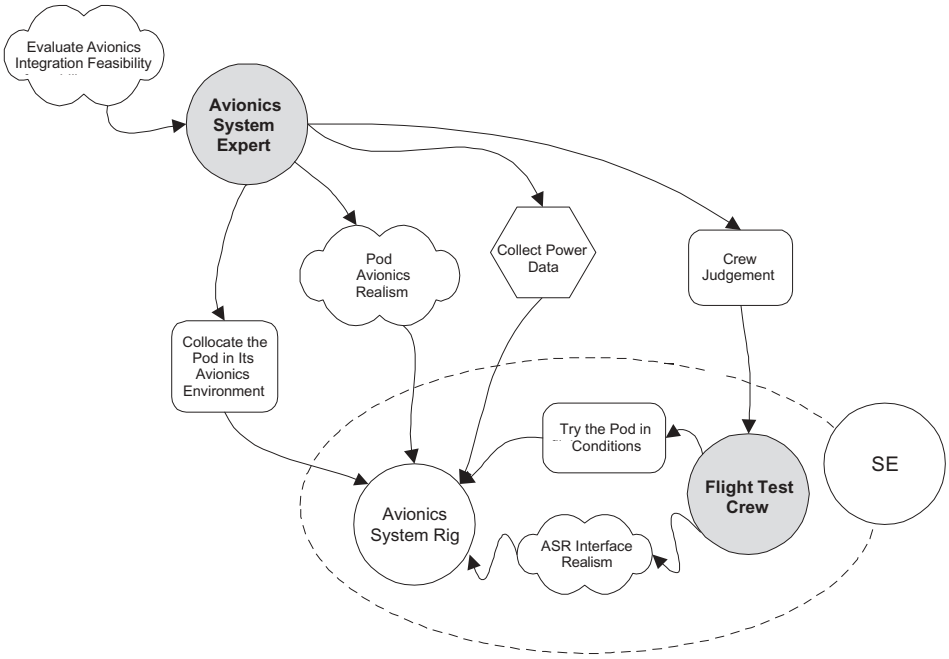


Figure 1: *The Enterprise Modeling Framework*

competitive tender, inform the bidders, etc. The EMF then draws upon these [7], although a more informal approach based on a combination of natural language and a semiformal graphical notation is preferred [2].

The *soft goal modeling* aims at producing operational definitions of the soft goals sufficient to capture explicitly the semantics that are usually assigned implicitly by the user [9], and highlight the system quality issues from the outset. A soft goal is refined in terms of subordinate soft goals, hard goals, tasks, and constraints. Constraints are associated with hard goals and tasks to specify the corresponding quality attributes. So, for example, the soft goal "buy a fast computer" will spawn the hard goal "buy a computer" and a set of associated constraints, e.g., CPU speed, memory size, cache characteristics, etc., that specify the computer quality attribute *fast* according to the stakeholders' perception.

Figure 2: *The Organizational Model to Perform the Integration Study*



Applying the EMF

To investigate the feasibility of equipping an aircraft with a new avionics subsystem, a particular kind of ground-based equipment capable of providing the characteristics of the aircraft and of its components is usually adopted. Such equipment, or Avionics System Rig (ASR), is an example of a synthetic environment (SE) or of a subcomponent. In fact, a SE would typically consist of a mixture of real and simulated equipment [10], and would even encompass human operators. Furthermore, a SE can have different degrees of complexity, depending on the kind of information it provides (i.e., on the decision-making process that it has to inform).

In [5], EMF is applied to support the requirements engineering process for a hypothetical ASR that is needed to investigate the feasibility of providing an aircraft with a thermal pod. A thermal pod is an infrared device that is, for example, normally used on aircraft and helicopters dedicated to *search and rescue* and *anti-fire* roles. Although hypothetical, the case study is firmly based upon a real application where SE's have been used for a similar purpose [10]. For brevity in this article, the presentation is limited to a few extracts from this example application.

Shown in Figure 2 is a simple organizational model, within which the ASR is employed. Circles represent agents, and dotted lines are used to bind the internal structure of complex agents; i.e., agents containing other agents. Consequently, the *avionics system expert* is a simple agent that interacts with the SE to collect information necessary to assess the feasibility of equipping the aircraft with the thermal pod.

The SE agent in Figure 2, instead, is a complex agent encompassing the agents ASR and *flight test crew*. As stated earlier, it is typical of SE's to encompass human operators who have to interact with the simulated environment, but who are not direct stakeholders of the process that the SE is designed to support. In this case, for example, the flight test crew is part of the overall SE, so that the effects

of the thermal pod integration on crew performance can be determined. However, it is the avionics system expert who is the primary (feasibility study) process owner.

Agents interact by exchanging goals and tasks. Clouds represent soft goals, rounded-rectangles represent hard goals, and hexagons represent tasks. Thus in Figure 2, the avionics system expert receives from the enclosing domain the soft goal of "evaluate avionics integration feasibility" for the new pod. Goals, tasks, and agents are connected by dependency-links, represented by arrowhead lines. An agent is linked to a goal when he/she needs that goal to be achieved; a goal is linked to an agent when he/she depends on that agent to be achieved. Similarly, an agent is linked to a task when it wants the task to be performed; a task is linked to an agent when the agent has to perform the task. By combining dependency-links, we can establish a dependency among two or more agents [2].

Each agent works as a goal transformer. Having received a goal, an agent will operate according to his/her own experience, knowledge, or position within the organization in attempting to achieve the goal. He/she will decide how to achieve the goal in terms of tasks and subordinate goals, and may choose to depend on other agents by passing out some of these tasks and subordinate goals. For example, the soft goal model in Figure 3 explains the behavior of the avionics system expert.

In order to achieve the received goals, the avionics system expert will need to "observe the pod in its avionics environment" and will require a *crew judgement*. The first soft goal will spawn some precise goals that the SE agent has to satisfy. These include the hard goal "collocate the pod in its avionics environment" with the associated soft goal "pod avionics environment realism," and the hard goal "monitor avionics system behavior," which leads to the task "collect power data."

The latter will require that a new agent, named flight test crew, will have to be included in the SE to operate with the ASR. In Figure 3, the A annotation on each decomposition line indicates that all these goals must be satisfied (and refined), whereas the goals and tasks in bold outline are those that the avionics system expert will pass out (see Figure 2) and are not further refined. To be able to express its opinion, the flight test crew in Figure 2 will require the possibility of *flying* the new pod, which places two other goals on the ASR: a hard goal "try the pod in flight conditions," and a soft goal "ASR interface realism."

By analyzing and refining the goals imposed on the SE agent, the final requirements for the ASR can be obtained. For example, by modeling the soft goals "pod avionics environment realism" and "ASR interface realism" (identified in Figure 2), a clear idea of the needs of the *avionics system expert* and *flight test crew* agents can be gained and translated into *functional* and *nonfunctional* requirements for the ASR.

Figure 4 provides this soft goal model for the ASR. It shows how the soft goal "pod avionics environment realism" from the avionics system expert spawns the following soft goals: "sensors and aircraft model realism," "avionics system realism," and the same "ASR interface realism" as was generated by the flight-test crew.

The soft goal "sensors and aircraft model realism" leads to a well defined set of constraints, which are represented by rounded-rectangles with one horizontal line and form nonfunctional requirements regarding both the sensors' tolerance and the aircraft model's capabilities. In this way, the flight test crew will be able to fly quite realistic low-level missions to test the pod.

The soft goal "avionics system realism" will translate into constraints that define what kind of equipment should be used. For example, the avionics system expert wants to be able to interface the pod with the real on-board computer.

Finally, the soft goal "ASR interface realism" will refine into both hard goals and constraints by mutual agreement of the two agents that have independently generated this same soft goal.

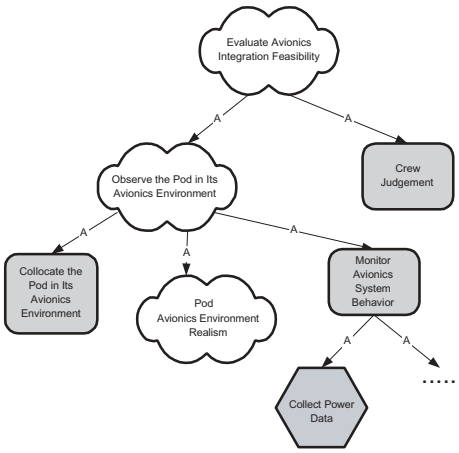


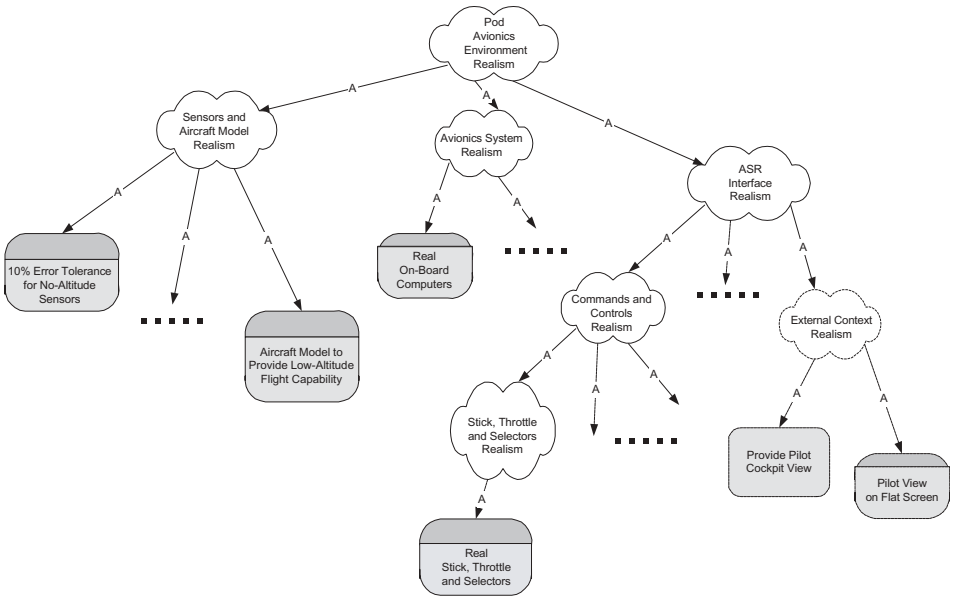
Figure 3: *The Soft Goal "Evaluate Avionics Integration Feasibility"*

In particular, resolving requirements clashes in this area result in the ASR providing the pilot cockpit view (a functional requirement) on a flat screen (a nonfunctional requirement). Importantly, this last goal demonstrates that different agents may have different opinions [11], and that *soft goal models allow the analysts to detect clashing requirements* at the early stages of a new system development and simultaneously provide a way to resolve them.

Conclusions

The application example described demonstrates the feasibility of the suggested approach. It also shows the benefits offered during the early phases of RE. This is the time when analysts and stakeholders have to cooperate to understand and reason about the organizational context within which the new system

Figure 4: *Soft Goals Imposed on the ASR*



has to function. They must identify and formalize not only the system requirements, but also the organizational setting that better exploits the new system's capabilities.

In particular, benefits can be observed in terms of requirements discovery and early validation. Discovery and validation are improved because of the visibility of decisions made by the stakeholders as a result of explicit organizational and goal modeling. Each type of EMF model provides a specific knowledge representation vehicle that the analyst can use to interact with the stakeholders to capture requirements, reason about them, and eventually reach an accepted formulation.

Soft goal models force the stakeholders to reason about their own concepts of quality (for example, the concept of *realism* in Figure 4). Hard goal models allow the stakeholders to understand and validate their role within the organization, whereas organizational models provide management with a clear view of how the business process will be changed or affected by the introduction of the new system (see Figure 2).

The resulting models also suggest that EMF offers potential benefits in the *post-deployment phase*. The clear links established between organizational goals and system requirements, in fact, allow the analysts to quickly identify the influence of organizational changes on the final system requirements, supporting both *system maintenance* and reuse in different application contexts.

Although EMF addresses the early stages of the RE process, the possibility of combining its outcome with techniques more suitable for dealing with further system development phases has been investigated [12]. For example, initial results suggest that EMF can be usefully applied as a forerunner to object-oriented approaches such as those based upon the unified modeling language [13].

Finally, the general principles upon which the EMF is based allow it to be deployed for a larger class of computer-based information systems beyond SE's. For example in [14], EMF has been applied to define the requirements for a workflow-based document management system. Whereas in [15], it has been adopted to analyze the organizational impact and advantages of introducing a corporate smart card as enabling platform for accessing and using different e-services delivered by the organizational information technology system (e.g., digital signature, certified e-mail, documents management systems, etc.).◆

References

1. Fickas, S., and B. R. Helm. "Knowledge Representation and Reasoning in the Design of Composite Systems." IEEE Transactions on Software Engineering 18.6 (June 1992).
2. Yu, E. Why Agent-Oriented Requirements Engineering? Proc. of 3rd Workshop on Requirements Engineering for Software Quality. Barcelona, Catalonia, June 1997.
3. Loucopulos, P., and V. Karakostas. System Requirements Engineering. UK: McGraw Hill, 1995.
4. Donzelli, P., and M. R. Moulding. Developments in Application Domain Modeling for the Verification and Validation of Synthetic Environments: A Formal Requirements Engineering Framework. Proc. of the Spring '99 Simulation Interoperability Workshop. Orlando, Fla., Mar. 1999.
5. Donzelli, P., and M. R. Moulding. "A Unified Approach to the Verification, Validation and Accreditation of Synthetic Environments: A Requirements Engineering Framework." Cranfield University Technical Report Ver. 1. SE027E/TR1. Dec. 1999.
6. Hammer, M., and J. Champy. Reengineering the Corporation: A Manifesto for Business Revolution. London: Nicholas Brealey Publishing, 1995.
7. Dardenne, A., A. Van Lamsweerde, and S. Fickas. "Goal-Directed Requirements Acquisition." Science of Computer Programming 20 (1993).
8. D'Inverno, M., and M. Luck. Development and Application of an Agent-Based Framework. Proc. of the First IEEE International Conference on Formal Engineering Methods. Hiroshima, Japan, 1997.
9. Cantone, G., and P. Donzelli. "Goal-Oriented Software Measurement Models." European Software Control and Metrics Conference. Herstmonceux Castle, East Sussex, UK, Apr. 1999.
10. Donzelli, P., and R. Marozza. Laser Designation Pod on the Italian Air Force AMX Aircraft: A Prototype Integration. Proc. of the NATO/RTO SCI Joint Symposium on Advances in Vehicle Systems Concepts and Integration. Ankara, Turkey, Apr. 1999.
11. Van Lamsweerde, A., R. Darimont, and E. Letier. "Managing Conflicts in Goal-Driven Requirements Engineering." IEEE Transactions on Software Engineering 24. 11 (Nov. 1998).
12. Donzelli, P., M. R. Moulding. Application Domain Modeling for the Verification and Validation of Synthetic Environments: From Requirements Engineering to Conceptual Modeling. Proc. of the Spring 2000 Simulation Interoperability Workshop. Orlando FL., Mar. 2000.
13. Rumbaugh, J., I. Jacobson, and G. Booch. The Unified Modeling Language Reference. Addison-Wesley, 23 Dec. 1998.
14. Antonelli, C., P. Donzelli, Masroianni, R. Setola, S. Vinti, and S. Tucci. A Web-Based Workflow Solution to Support the Italian Government Agenda Definition Process. Proc. of the Italian Automatic Computation Association 2000 Conference – Le Technologie dell'Informazione e della Communcazione come motore di sviluppo del Paese. Taomina, Italy, 27-30 Sept. 2000.
15. Donzelli, P., and R. Setola. Putting the Customer at the Center of the IT System – A Case Study. Proc. of the EuroWeb 2001 Conference – The Web in the Public Administration. Pisa, Italy, 18-20 Dec. 2001.

About the Author

Paolo Donzelli, Ph.D., is an advisor with the Department of Informatics of the Office of the Prime Minister in Rome, Italy. A former engineering officer with the Operational Testing Center of the Italian Air Force, Dr. Donzelli was a senior research fellow with the Computing Information Systems Engineering Group, Royal Military College of Science, Cranfield University, United Kingdom. He has a variety of interests in the software engineering area, and his doctorate thesis was in software process quality modeling.

Presidenza del Consiglio
dei Ministri
Ufficio per l'Informatica,
la Telematica e la Statistica
Via della Stamperia 8, 00187
Roma, Italy
E-mail: p.donzelli@governo.it

Highpoints From the Amplifying Your Effectiveness Conference

Elizabeth Starrett
CrossTalk

Improvements within an organization require people skills as well as technical skills. The Amplifying Your Effectiveness Conference held last November in Phoenix, Ariz., provided presentations for both sets of skills.

Improvement to an organization, project, or process usually requires more than just technical skills; it requires people skills. The Amplifying Your Effectiveness (AYE) conference strives to give attendees the personal skills required for improvement. As one of the AYE hosts, Jerry Weinberg states, "[The AYE] conference looks at technical problems from this human point of view."

The AYE 2002 conference was held in Phoenix, Ariz., from Nov. 3-6, 2002. The attendees were from technical positions such as project managers, software developers, software testers, and quality assurance personnel. While they were seeking new information on performing their jobs better, they also appreciated the atmosphere of the conference.

The AYE conference may be different than other conferences you have attended. Attendance is limited to the first 100 registrants and use of slides during a presentation is discouraged. The AYE hosts want speakers to talk to attendees, discussing information from their own experiences instead of simply reading from slides. As a result, speakers sit in a circle with the presentation attendees and discuss the topic of the presentation. The presenter notes key points on a flip chart during the discussions.

The range of issues at the AYE conference dealt with personal skills such as writing, presenting, dealing with different personalities, working with differences, working under stress, enhancing personal influence, effective feedback, and change-agent skills. Technical presentations addressed quality vs. speed, technical reviews, service level agreements, sharing project status, and dealing with project problems.

All the presentations included an activity to use the ideas shared in the presentation. The presenters then called on the attendees to provide their solutions before sharing solutions from the presenters. One example of this was the presentation "Building Writing Skills and Confidence: A Writing Workshop" by Johanna Rothman and Naomi Karten (two well- respected authors). During this presentation, the participants were challenged to write a story on a variety of topics. The results of the exercise were then shared, along with several tips for effective writing. These tips are included in the "Tips and Ideas for Building Writing Skills and Confidence" box.

When asked, a repeat attendee said she liked the conference because she preferred the focus on people skills vs. technical skills. One of the key attributes of this conference was the personal touch from each host. When not presenting, the hosts made themselves available to all conference attendees to answer questions and discuss issues. This invitation was available during the welcome dinner, daily lunches, and even anticipated for people that may be sitting in the halls not interested in any presentations. However, I was told the hosts never found this last situation at the AYE conference.

"The Amplifying Your Effectiveness conference is geared to help those with technical strengths tap into their interpersonal and relational talents."

A Few Example Models

Popular ideas shared at the conference included the Satir method and models. The Satir tools were developed by Virginia Satir, a family therapist, as a "self-esteem tool kit" and are aimed at helping people realize that they own the resources necessary to deal with daily situations but often forget to use them, especially when feeling powerless. Following are the tools included with this kit:

- **Wisdom Box.** The ability to know what is right and what is not right for you.
- **The Golden Key.** The ability to open up new areas for learning and practicing, and to close them if they do not fit for you at this time.
- **The Courage Stick.** The courage to

CONTINUED ON PAGE 30

Tips and Ideas for Building Writing Skills and Confidence

1. Learn from others. Find role models – people whose writing you like – and study their style. Read with intentionality. Notice what strikes you as good or bad writing.
2. Writing anything is better than writing nothing. Practice makes less imperfect. Make every writing opportunity an opportunity to develop your writing skills.
3. All good writing starts with the initial rough draft. Your first draft is just the starting point. Learn to trust yourself and the process.
4. Don't fall in love with your own words. Edit ruthlessly. Focus on tight writing. Become best friends with your delete key.
5. Write like you speak. Eschew terminological obfuscation and fancy formality. Write in a conversational me-to-you tone.
6. Let your subconscious do your writing for you. A great deal of writing happens when you are away from the keyboard. Write, put it away, then look at it later on with fresh eyes.
7. When (not if) you get stuck, notice your writing "shoulds." Acknowledge them and set them aside. Take a break, then write an email about what you are stuck writing about.
8. Find a setting that is conducive to writing. Use your favorite font. Play your favorite music. Find your favorite location. Use whatever approach works best for you.

THE FIFTEENTH ANNUAL

Software Technology Conference

Strategies & Technologies: Enabling Capability–Based Transformation

28 APRIL - 1 MAY 2003 • SALT LAKE CITY, UT

Software design has long been viewed as an art. Software has also become so pervasive that many endeavor to reinvent software design as a science. Many of the papers presented during past years have focused on how to accomplish this seemingly perpetual task. This year, STC will bring the art of developing software a step closer to a science by focusing on how strategies and technologies can bring about transformation. A strategy can be viewed as a science of implementing policy, a specific or defined method to reach compliance. Technologies are the tools for implementing strategies. Implementing strategies through the use of technologies brings science into the art of software design. Join us in defining how best to use strategies and technologies as a basis for transforming our capabilities by participating in STC 2003.

About STC

The best way to describe STC 2003 is that it will be jam-packed! This year's conference will include more than 180 events to choose from, including general sessions, luncheons, plenary sessions, and presentation tracks. If you work with software, STC provides outstanding training and networking opportunities. Some organizations report that they must send a small army to absorb all the information that is important to their organizations.

In its fifteenth year, STC is the premier software technology conference in the Department of Defense and is co-sponsored by the United States Army, United States Marine Corps, United States Navy, United States Air Force, the Defense Information Systems Agency (DISA), and Utah State University Extension. We anticipate over 2,500 participants this year from the military services, government agencies, defense contractors, industry, and academia.

STC is Endorsed by:

Lt Gen Harry D. Raduege, **Jr.**, Director, Defense Information Systems Agency

LTG Peter M. Cuviello, GS Chief Information Officer/G6, U.S. Army

John M. Gilligan, Chief Information Officer, U.S. Air Force

RADM Kenneth D. Slaght, Commander, Space and Naval Warfare Systems Command, U.S. Navy

Robert L. Hobart, Deputy Commander, Command, Control, Communications, Computers, and Intelligence/Integration (C4II), U.S. Marine Corps.

IEEE Computer Society CSDP Preparation Course and Examination

STC is happy to partner with the IEEE Computer Society to offer a preparation course and examination for the Certified Software Development Professional (CSDP) program at STC 2003. The CSDP is the Computer Society's certification program for software professionals. Developed by industry experts in a rigorous three-year process, the new CSDP credential is intended for software engineers, software developers, software program managers and other professionals with 5 or more years of experience. The CSDP exam is designed to measure an individual's mastery of the fundamental knowledge required to perform the functions of an experienced software engineer. The CSDP is the only certification for computing professionals that carries the brand, reputation, and standards of the IEEE Computer Society. Complete details about CSDP are available at <http://www.computer.org/certification>.

Conference Highlights

STC will host three concurrent speaker luncheons on Monday featuring **Stephen J. Mellor,** Founder & Vice President, Project Technology, Inc.; **Dr. David A. Cook,** Principal Engineering Consultant, Shim Enterprise, Inc./Software Technology Support Center; and **Charles Thomas Burbage,** Executive Vice President & General Manager, Joint Strike Fighter-Lockheed Martin Aeronautics Co. The co-sponsors will join together to discuss the services perspective on strategies and technologies in a panel discussion Tuesday morning.

Wednesday's plenary session speaker is **David W. Carey,** Vice President, Information Assurance, Oracle Government, Education & Healthcare. Thursday, **Dr. William E. Halal,** Professor of Management, George Washington University, will address the conference. STC will be capped off in the Thursday afternoon Closing General Session with **Tim Border & Doug Nielsen**, two of America's most sought-after motivational speakers in the areas of personal development, motivation, leadership, change, and customer service.

Special Sessions

Sponsored track presentations will be offered throughout the week by the following organizations: Defense Information Systems Agency (**DISA**), Institute of Electrical and Electronics Engineers (**IEEE**), International Council on Systems Engineering (**INCOSE**), Joint Strike Fighter (**JSF**), Office of the Secretary of Defense (**OSD**), Software Engineering Institute (**SEI**), and Software Technology Support Center (**STSC**).

Track Topics on the Conference Agenda

- •Acquisition
- •Architectures
- •Business Technologies
- •Development
- •Innovations
- •Inspections
- •Interoperability
- •Management
- •Metrics
- •Processes
- •Programming
- •Quality
- •Requirements
- •Security
- •Testing
- •Tools
- •Trends

STC 2003 Exhibiting Organizations (As of 11/18/02)

Ada Core Technologies, Inc.
AFIT Software Professional Development Program
Army Small Computer Program
Barrios Technology, Inc.
Battelle
BMC Software
Boeing Co.
Borders Books & Music
CDW Government, Inc.
Centech Group, Inc.
Cognos Corp.
Compliance Automation, Inc.
Crystal Decisions
DCS Corp.
DDC-I, Inc.
Defense News Media Group
DISA
EDS PLM Solutions
Galorath, Inc.
HQ Standard Systems Group
IBM
IEEE Computer Society
Integrated System Diagnostics, Inc.
Joint Advanced Weapons Systems Sensors Simulation and Support
JOVIAL Program Office
Marine Corps System Command
Microsoft
Military Information Technology
NASA GSPC SEWP
Northrup Grumman Information Technology
Objective Interface Systems, Inc.
OO-ALC/MAS
pragma Systems Corp.
Praxis Critical Systems Ltd.
Quality Plus Technologies, Inc.
Quantitative Software Management, Inc.
Rational Software
Real-Time Innovations, Inc.
SAIC
Sciforma Corp.
SEEBEYOND
Software Productivity Consortium
Software Technology Support Center (STSC)
Space Dynamics Laboratory
SPAWAR
TeamQuest Corp.
Tecolote Research, Inc.
TeraQuest Metrics, Inc.
The Aerospace Corp.
The Software Revolution, Inc.
U.S. Air Force
USAA
USACECOM SEC
Utah State University Innovation Campus
Vitech Corp.
Wind River

Special Events

STC 2003 features many networking opportunities such as the Opening Welcome Reception on Monday evening and the Fifteenth Anniversary Progressive Social featuring "Still Surfin'" musical entertainment on Wednesday.

Registration

Completed registration form and payment must be received by 24 March 2003 to take advantage of the early registration fees. Credit cards will not be charged until 1 April 2003. The conference fee structure for STC 2003 is as follows:

Discounted registration fee (paid by 24 March 2003):

Active Duty Military/Government*	$645
Business/Industry/Other	$775

Regular registration fee (paid after 24 March 2003):

Active Duty Military/Government*	$715
Business/Industry/Other	$845

* *Military rank (active duty) or government GS rating or equivalent is required to qualify for these rates.*

Housing

The Housing Bureau of the Salt Lake Convention and Visitors Bureau (SLCVB), using the online Passkey system, handles housing reservations. Housing has been available since May 2002; therefore, some government rate guestrooms at specific hotels may not be available. To access the Passkey system, log on to the STC Web site at www.stc-online.org and select the Housing Reservation button. If you prefer to make your reservation using a traditional method, a PDF version of the housing form is available online.

Delta Airlines Special Discounted Airfare

Delta Airlines is our official host airline for all STC 2003 attendees. Take advantage of the five-percent discount off Delta's published round-trip fares within the continental U.S. A ten-percent discount is offered on Delta's domestic system for travel to STC 2003 based on the published unrestricted round-trip coach (Y06) rates. Book your flight by calling Delta Meeting Network® Reservations at 1-800-241-6760, Monday-Sunday 8:00 a.m. – 11:00 p.m. Eastern Time, or have your travel agent call for you. You must refer to File Number **189840A** when making your reservations.

Trade Show

STC 2003 will again feature its accompanying trade show, providing 180+ exhibitors the opportunity to showcase the latest in software and systems technology, products, and services. This year's schedule has been adjusted to allow participants more time to interact with the vendors without conflicting with conference presentations.

Exhibit space is sold in increments of 10' x 10' at a rate of $1575 if application is received on or before 14 February 2003. Should space still be available after this date, booth space will be sold at the rental rate of $1775 per 10' x 10' space. Special fees and restrictions may apply to certain types of booth space. Complete trade show rules, regulations, and updated hall layout are available on the STC Web site.

All badged exhibit personnel wishing to attend the entire conference are eligible for a discounted conference registration fee. Please utilize the conference registration form that was mailed to the exhibit manager in early January to register for the conference.

www.stc-online.org

Source Code: CT5A

General Information
stcinfo@ext.usu.edu
435-797-0423

Technical Content Inquiries
stc@hill.af.mil
801-777-7411

Trade Show Inquiries
stcexhibits@ext.usu.edu
435-797-0047

Media Relations
stcmedia@ext.usu.edu
435-797-0089

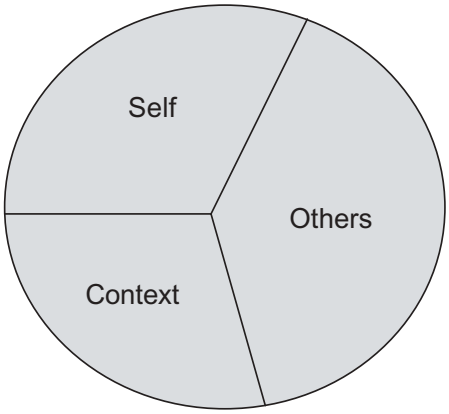


Figure 1: *Congruence Model*

CONTINUED FROM PAGE 27

try new things, and to risk failure.

- **The Wishing Wand.** The ability to ask for what you want, and to live with not getting it.
- **The Detective Hat.** The ability to examine data and to reason about those data.
- **The Yes / No Medallion.** The ability to say *yes*, the ability to say *no* (thank you), and the ability to mean what you say.

Another model discussed was the Congruence Model (Figure 1). The intent of this model is to remind the user that in dealing with different situations, the person should consider oneself, others involved in the situation, and the context of the situation. The inclusion or exclusion of any of these elements results in a stance that that may be blaming, placating, overly reasonable, or congruent.

A similar model shared at the conference was the Thomas-Kilmann Conflict Mode Instrument (TKI) (Figure 2). This model describes possible reactions to conflict based on consideration of self vs. consideration of others. The five possible reactions are displayed in Figure 2.

The TKI reactions are as follows:

- **Competing.** The goal is to win.
- **Avoiding.** The goal is to delay or avoid.
- **Compromising.** The goal is to find a middle ground.
- **Collaborating.** The goal is to find a win-win situation.
- **Accommodating.** The goal is to yield.

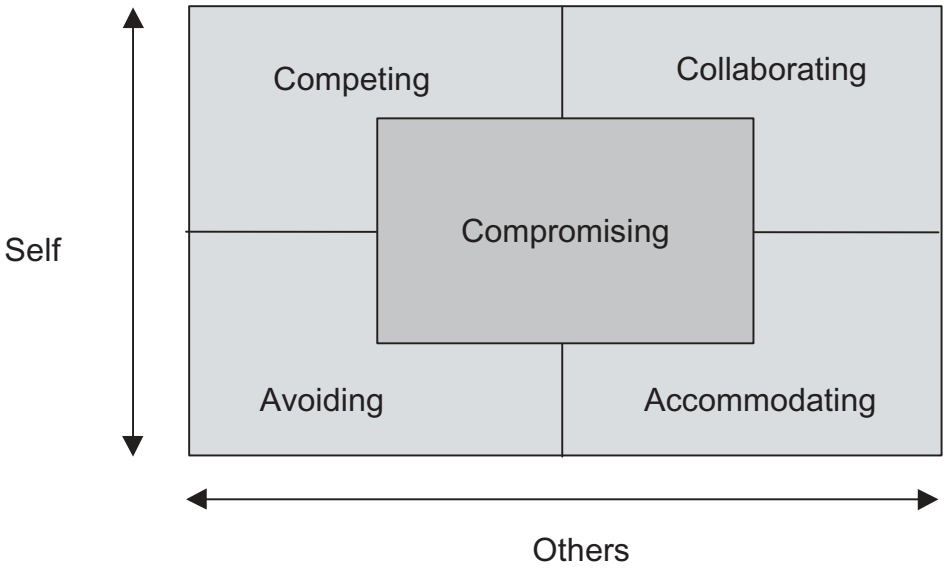
To use the models when in an uncomfortable confrontation, the user should take a moment to consider where he or she is relevant to the model and where he or she would like to be. Then, modify behavior appropriately.

Many attendees considered the closing session to be the highlight of the AYE conference, since it was an opportunity to network those who still had questions or other needs with people who could help.

The AYE conference is geared to help those with technical strengths tap into their interpersonal and relational talents. It aims to demonstrate that these types of talents can be learned in the same way that technical skills can be learned.

In the interest of sharing technical expertise, the AYE 2003 conference is providing one scholarship to each of the winning projects of CrossTalk's 2002 U.S. Government's Top 5 Quality Software Projects contest. This will ensure attendees the opportunity to interface with some of the best software developers in the United States. Readers interested in learning more about this conference can access the AYE Web site at <www.ayeconference.com>.◆

Figure 2: *Thomas-Kilmann Conflict Mode Instrument*



The First Book of EPP

The foldout poster in this issue of CrossTalk based on Eric Levenez's research is a fascinating pedigree. Adding fruit to that tree, BackTalk offers the First Book of Etymology of Programming Patois (EPP).

1. In the beginning was the computer. The computer was void of instruction and darkness was upon the face of the screen.

2. Engineers said, "Let there be light," and there was light. Engineers saw the light and divided the light from the darkness. They called the light Software and the darkness Hardware.

3. The prophet Moore foretold of computing hardware whose capacity swelled as its size shrunk. Following Moore, the seer Parkinson dreamt of a growing cistern filled with software that incessantly replenished the cistern such that it was forever full. Engineers were perplexed and dismayed.

4. Assembly, the first to take on the paradoxical Goliath, was a young, brash, autodidact language. He was disdained by the scholars who thought him more idiom than language. Undeterred, his speed bridled the giant. While arduous to preserve, engineers swore their alliance to Assembly till the day their languages are without root or branch.

5. Fortran followed Assembly. Blessed with longevity and filled with virility Fortran begat Algol. Intelligent yet often misconstrued, Algol was a captivating mistress who brought forth several programming languages.

6. Algol's firstborn was CPL who begat BCPL who begat B who begat C who with his objective son C++ emerged as a fruitful and prominent language. A true blue-collar language, C was healthy, reticent, and, at times, explosive.

7. The malodorous language B-O begat Flow-Matic, of infomercial fame, who in turn begat COBOL, a lad who exited the womb in a three piece suit and briefcase in hand.

8. COBOL seized an incestuous opportunity to pair the mistress Algol and Fortran the IV, Fortran's great great grandson. The union engendered Algol's second language PL/I. Harsh to the eye and large in the thigh, suitors eschewed PL/I who languished in obscurity.

9. Algol's third language, Simula, was ahead of her time and crowned the Princess of Objectivity. Simula was beguiled by Lisp, a parenthetical language from the League of Ivy, and they begat Smalltalk who was offered at the altar of Xerox. Several years later, Simula's daughter collaborated with C to begat C++, an objective, robust yet capricious language.

10. Lisp also beguiled Algol and her fourth language, Scheme, was born. Favoring elegance over practicality, Scheme had a proclivity for genetics but was a stranger to success. Scheme had a tryst with Sir Lambda of Calculus that produced Haskell. Haskell was a lazy polymorphic language who begat Eddie, the annoying friend of Wally who was the brother of Beaver.

11. Algol's fifth language, Pascal, resembled Algol's great great grandson C and had a penchant for higher learning. Pascal begat three daughters: Modula, who followed her father into education; Mesa, who dabbled in woodwork; and Ada, the love child of the Sirens of War who was reared to be the *only* language. In the right hands this well-defined, multitasking, polymorphic prodigy could be an industrious lady. Yet, in the wrong hands she was a fickle corpulent popinjay. The three sisters now languish in an old code home.

12. Algol's last progeny Algor, born with the mark of the ass, inspired the movie "Love Story" and invented the information superhighway on which all languages would, in time, journey. Algor, like his mother, fell from an ivory tower. Many blame the Earl of Chad, although suicide is a more likely suspect.

13. Prolog from Ether was a self-made language that preferred matching solutions to solving problems. For common users, Prolog induced recursive headaches and algorithmic nausea. For fuzzy addicts, he excelled in fabricating exquisite neural lattices.

14. In the Land of Ice, the language Snobol manipulated strings to scale Mount Data. This pioneer, now obsolete, journeyed on a virtual machine – a concept that would be instrumental in the navigation of Algor's superhighway.

15. Rooted in small towns and villages across the country, BASIC was the language of the mediocre. At an early age, BASIC shunned complexity and took a vow of isolation. BASIC was reincarnated as Visual BASIC and pursued Nirvana, which disbanded after Cobain's grungy demise. Limited by scale, many discarded Visual BASIC as slang but his ubiquitous force continues to appeal to the masses.

16. C begat Awk and Awk begat Nawk and Nawk enticed Sh, the quiet interpreter, who begat Perl. A natural navigator, Perl toils on the River Script extracting, distributing, and dealing data. Often used but seldom loved, Perl is known as the Swiss Army Chainsaw for the callous methods she employs to extricate quandaries aptly referred to as Perl Jams.

17. Angered by his great-granddaughter's lack of classiness, C connived with ABC and Modula-3 to supplant and string up Perl. They hatched Python, a smooth broker who haunts the banks of the River Script constricting the interrupts of his rival Tcl; not to be confused with TLC who lost her left eye in Honduras.

18. In a defiant move, Perl collaborated with Python, Smalltalk, and Ada's towering daughter, Eiffel, to ameliorate their dominion of the River Script. The result was Ruby. Inheriting Perl's resiliency, Python's elegance, and Smalltalk's autonomy, Ruby traverses the River Script and Algor's Web- encrusted highway with ease.

19. During the French revolution ML begat SML, a nasty language who tortured his users to the delight of his father. While incarcerated in the Palace de Cristal, SML begat CAML. After contracting emphysema, CAML begat CAML Light who moved to Ireland and begat O'CAML who objectively surmounted his innate humps.

20. Mesa begat Cedar who joined with Scheme, Smalltalk, and C++ to harvest a solid language called Oak. Oak begat Java who was sun-roasted for mass distribution. Brewed in early popularity, Java has percolated through harsh rays of criticism on her way to maturity. Only time will tell if she will amount to a hill of beans.

21. And thus ended the first period of the proliferation of programming patois.

— Gary Petersen
Shim Enterprises, Inc.

CrossTalk / MASE
7278 4th Street
Bldg. 100
Hill AFB, UT 84056-5205

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737