In the July/August 1997 issue of IEEE Software, Steve McConnell issued a “call to action”: an invitation to software practitioners to join the fray, to share their words “braced by labor and invention” with the magazine’s readers. That call, combined with the support of Robert Glass, led to this focus section. We gladly join the fray, partly to look back but primarily to look ahead.

Before continuing, we propose common ground. We agree with F.G. Harold’s concept of “optimal heterophily” and use the term software engineering to include information technology.

Y2K presented a formidable (albeit well-defined) challenge: find the date problems and fix them (see our “Congratulations” in the accompanying sidebar). Post-Y2K presents a more formidable challenge: a set of issues more complex, less well defined, with solutions more resistant to prescription. With Y2K behind us, we address the question “What’s next for Cobol?” in terms of how it came to be what it is and how it will likely continue to change.

A Brief History

In January 1960, Cobol was officially released to the world. Inspired by Grace Hopper’s Flow-Matic, Cobol embraced English
as its medium of expression. An unpopular notion among the digital elite of the day, scientists and mathematicians took a dim view. Hopper, a mathematician herself, understood the secret world of symbols and was undaunted by naysayers who insisted symbols were the only diet of computers. In contrast, most other popular programming languages such as C++ and Java are founded on symbols, allied with the language of mathematics.

Both sides have largely misrepresented the words–symbols debate, igniting endless discussions based more on taste than on substance. Critics call Cobol wordy. Proponents call Cobol self-documenting. Each is right but misses the point.

Reading for Sound

A programming language can be read and written. The word “read,” however, carries with it a heavy assumption of silence. And in that wrong assumption of silence is the key that differentiates Cobol from other programming languages, explains its longevity, clarifies its maintainability, and captures its essence. Unlike other popular programming languages, Cobol can be read aloud, or pronounced, where pronounce means “to articulate words; to speak; voiced in phonetic components.”

English has 44 phonemes that are the basic sound bites of all words. Research conducted in the ’90s found that phonemes play a critical role in comprehension. The brain reads by breaking words into sounds (phonemes).

Phonetics and whole-language are diametrically opposed methods of reading. Phonetics emphasizes the use of phonemes while whole-language advocates grasping whole words as indivisible units. In many ways, Cobol is to phonics as its more symbolic cousins are to whole-language. Yale University found that by associating sounds with words, people rapidly remember and process what they read. Cobol programs can be rapidly read and remembered because they tap into our linguistic centers where phonemes are processed. Cobol code is based on a culture of sound as the mnemonic amplifier for understanding. Phonetic reading (processing sounds) is the human equivalent of parsing.

It is not only a question of sounds versus symbols. Symbolic languages place a premium on reduction at the cost of clarity. The Cobol culture finds pride not in reducing three operations to one by taking side-effect shortcuts, but by producing understandable systems through the clarity of the language and by rejecting the “Law of Parsimony” (as in “Steps 5–11 are self-evident and are not shown below” in your Introduction to Algebra textbook).

The syntax, the semantics, and the culture of Cobol are bound to the power of English. Its dual nature—pronounceable and parsable—presents two voices to its two distinct audiences: one human, the other machine. English evolves as a vehicle of thought to capture ideas and to convey information. Cobol evolves for the same reasons, taking to objects and the Web as naturally and easily as English, while maintaining its “conceptual integrity” by constructing computable entities cut directly from the cloth, and in the language, of the problem domain.

Y2K Costs and Benefits

Y2K clearly illustrated the extent to which IT still relies on Cobol. According to Jim Sinur, a Gartner Group vice president and research director, the immediate and long-term benefits from our $300- to $600-billion Y2K investment include

- learning how to manage large software projects,
- greater discipline in our profession,
- cleaner portfolios,

Societies create vehicles to promote change from one state to another—commonly from the certain to the unknown (the graduation ceremony is one). Y2K was the public’s chosen institutional vehicle of change to cross the millennium. Tests brought reassurance as systems set forward survived, thus lessening fears and cushioning the passage.

A century of unparalleled technical achievement chaperoned out in a minute-by-minute digital pulse-checking moment and Nietzsche’s “long obedience in the same direction” held steady (Open Sources, C. DiBian, S. Okman, and M. Stone, eds., O’Reilly and Associates, Sebastopol, Calif., 1999, p. 147). Our congratulations to every programmer who, instruction by instruction, helped pierce the divide. Fittingly, the world’s population joined you on the eve of the new millennium as both witness to and participant in a job well done.
That philosophy of extinction has been replaced with one of extension and inclusion.

- A reduction in the legacy-code fear factor,
- Revamped testbeds,
- Better testing and software development methods, and
- The introduction of many third-party software tools and vendors.

Y2K answered the question, “Where are we now?” In addition to framing the moment, Y2K also pointed where we are heading. From the late ’80s to the mid ’90s, the client-server crowd predicted Cobol applications would quickly be rewritten in other languages and rehosted on fat clients and fatter servers. Like the little Dutch boy who found out the hard way that the dyke was not protecting the land from the ocean, initial attempts to rewrite or rehost many Cobol applications met the same fate. That philosophy of extinction has been replaced with one of extension and inclusion. Cobol applications are, by and large, too critical and too valuable to consider replacing en masse.

Perhaps the most important Y2K side-effect provided Cobol with something it had always lacked: a broad-based community of developers focused on providing Cobol with options. With too much code to be repaired manually, a market of software remediation tools came of age. The tools did not turn into pumpkins at the stroke of midnight. Instead they kicked off the other shoe, took off the gown, put on a Spidergirl outfit, and swung onto the Web.

Application Architectures Post-Y2K

As the millennium gathers momentum, the move away from monolithic to multilingual systems will accelerate. Seismic activity radiating out from e-commerce epicenters has established the need for refactoring legacy applications along more accessible Internet fault lines. Driving the refactoring efforts are, in addition to e-commerce, a need to accelerate products to market, improve business processes, enhance customer service, and support mass customization.

Components—pieces of software providing services—and middleware—any software sitting between a user and a database or legacy application—will play pivotal roles in integrating application silos into enterprise-wide systems. Distributed in three-tier architectures (presentation, business logic, and database), enterprise-wide applications will be connected using the plumbing, protocols, and services of the W3C Consortium, Corba, DCOM, and Enterprise JavaBeans.

Integration presents opportunities and risks; Cobol can participate in the refactoring and integration. In an ironic twist, in many ways what will be needed to extend legacy Cobol code is not less Cobol, but more.

- On the desktop: Cobol IDEs can input Cobol source code and output Java byte code, run as Cobol virtual machines inside browsers, invoke and communicate with JavaBeans and with COM and Corba objects, use ActiveX controls, and generate Cobol graphical interfaces.
- On the server: Cobol can interoperate with Corba, DCOM, and Enterprise JavaBeans; work with and control HTML and dynamic HTML; and be the CGI language working with form data and named environment variables. Cobol’s Accept and Display verbs can receive and present name-value pairs. “Cobol as the gatekeeper brings the power of Cobol’s preeminent data-manipulation features into play and allows the gateway to talk to other programs, databases, and transaction monitors.”
- On the mainframe: Cobol’s strict adherence to backward compatibility allows Cobol programs from the 1960s to intermix object-oriented and procedural statements to invoke other objects or be invoked. Cobol code can be wrapped and participate as large-grained components, or cleaved at the business-rule level to participate as smaller-grained objects.

As Steve McConnell said to Robert Glass, it is the language in combination with the integrated development environment and its facilities for an intended environment that determines the suitability of a language. Cobol IDEs have learned from their competitors and now offer everything from visual programming to Web and SQL wizards.

Article Overview

Cobol is a community of academicians, practitioners, businesses, and standards committees, each with their own separate
thread of control, each influenced by the other. This issue’s articles address the problems facing three of the intertwined interests; each group should find interest in the others.

**Academia**

Don Carr and Ron Kizior surveyed almost 3,000 CIS and IS programs and 5,000 businesses about the future of Cobol. One of the most extensive studies of Cobol in recent history, their findings are presented in “The Case for Continued Cobol Education.”

In “Cobol in an Object-Oriented World: A Learning Perspective,” Bill Hardgrave and E. Reed Doke investigate approaches to teaching object-oriented programming, the role of Cobol, and the relationship between Cobol and Java.

Jean Sammett is a well-known and highly regarded figure in the world of programming languages and Cobol. One of the original members on the Short-Range Committee, the committee that defined the first specification of Cobol, she tells how Cobol came to be in this insider view of history, entitled “The Real Creators of Cobol.”

**Business**

The world is data. In “Legacy Integration—Changing Perspectives,” Frank Coyle explores how the new focus on data, driven by the Web and several Web standards, including XML, match and map into Cobol’s data-centric design.

What did we learn from Y2K? What did we get for our money? Are we smarter now or will we make the same mistakes again? Leon Kappelmann explores these questions in “Some Strategic Y2K Blessings.”

**Practitioner**

The next Cobol standard, commonly known as Cobol 2002, is almost complete. Don Schricker, the chairman of J4, the Cobol Technical Committee of the National Committee for IT Standards, explains some of the new features of the language in “Cobol for the Next Millennium.”

Brian Henderson-Sellers, one of the leaders in object-oriented software development, explains an OO methodology and its benefits in “The OPEN Framework for Enhancing Productivity.”

Cobol programmers have not traditionally relied on multivendor toolsets, and many still work with compilers dating from 1968 and 1974. To build new systems and extend legacy applications, Cobol programmers will need to become familiar with new toolsets and IDEs. To this end, Thane Hubbard takes a look at a Cobol GUI tool, Alden Lorents examines a modern Cobol IDE, Steve Shiflet looks at data mining for business rules, and Jon Wessler explores components and rapid application development. We conclude with a taxonomy that lists more than 100 tools divided into 14 categories—only a partial catalog of the tools available.

From what you read in the popular press, you would think the leading lights in our profession are waiting in line to flay Cobol. As we worked on this issue, we found that the true giants in our profession (such as Dave Parnas and Fred Brooks) had nothing negative to say about Cobol. We did, however, find two experts to debate Cobol’s place in today’s IT environment: Frank Coyle in his Point position, “Does Cobol Exist?” and Cay S. Horstmann in his Counterpoint, “Cobol vs. Java.”

Cobol’s future began in November 1989 at a meeting in Scottsdale, Arizona, when a group of Cobol and object-oriented experts sat down to map out the future of the language. This group, which became known as the Scottsdale Symposium, concluded that Cobol was an excellent vehicle for providing object-oriented capabilities for business programmers. The addition of one new verb, invoke, brings objects to Cobol. While the new standard will not be finalized until 2002, many Cobol vendors have accelerated to Internet time and now offer features found in the upcoming standard. Expect OO Cobol to follow the same trajectory as relational databases: reluctance at first, then acceptance. Java’s creator James Gosling puts it in perspective: “When people ask me why I don’t talk about Java’s object-oriented capabilities, I tell them I have a friend and his name is Bob. I don’t tell them Bob is breathing.”

Today, Cobol and Visual Basic each account for 35% of new business applications development, with the remainder divided...
among a handful of languages, according to Matt Hotle, a Gartner Group vice president. Hotle finds that “Cobol is still in the sweet spot of development and extension” and believes “Cobol will be a strong viable language for the next 15 years.” In fact, Gartner Group is advising universities to stay with Cobol due to a coming shortage of qualified Cobol programmers. Robert Glass, when comparing Cobol to Delphi, C++, Java, Visual Basic, and Powerbuilder, concluded that Cobol is still the best language for developing information systems.6

Cobol is built on a powerful idea: an open language that is not owned by any individual, organization, or group of organizations and uses English to talk not to the computer but to its writer and most especially to those who follow.

Despite the disproportionate publicity given the most radical experiments, the majority of leading IT systems continue along a moderate path while absorbing new techniques that contribute to practical, proven benefits. Discarding the novel and sensational, business, Cobol, and IT will continue to evolve together. ☏

Acknowledgments

We thank the more than 30 authors who submitted papers for consideration. The majority were excellent and highly deserving of publication.

References


About the Authors

Edmund C. Arranga is the editor-in-chief of CobolReport.com and coauthor of Object-Oriented Cobol (SIGS Books/Cambridge Univ. Press, 1996). His interests include patterns, Cobol, the data-driven Web, and e-commerce. Arranga is currently working on a book about the Cobol 2002 standard. He graduated from Texas A&M University and is completing his MS in software engineering at Southern Methodist University. He is a member of the IEEE. He can be reached at earranga@objectz.com.

Wilson Price has taught computer information systems at the university level for over 30 years, chairing the computer information systems department at Merritt College, in Oakland, California, for 24 years. He is the author of 38 textbooks on computers and information processing, including six on Cobol. The latest is entitled Elements of Cobol Web Programming (Object-Z Publishing, 1999). His consulting services have ranged from writing system software in assembly language to the design and implementation of an organization-wide database management system. As a principal of Object-Z Systems, his current activities focus on Cobol—in particular teaching OO Cobol and Web programming using Cobol. He received an MS in engineering from the University of Pittsburgh and an MS in mathematics from California State University, San Jose. Contact him at 220 La Es- piraL, Orinda, CA 94563; wprice@objectz.com.