Editor-in-chief: Steve McConnell, Construx Software, software@construx.com

The Best Influences on Software Engineering

Steve McConnell

I wanted to get some perspective on the best influences we've seen during software engineering's first 50 years. After drafting an initial list of influences, I turned to our advisory boards. *IEEE Software* has active editorial and industrial advisory boards comprising leading experts from around the world. What follows is our discussion of the best influences on software engineering. As you will see, we had some significant differences of opinion!

Any list like this is bound to contain significant subjectivity. As board member Marten Thomas pointed out, this discussion might not tell us much about software engineering's prospects, but it does say a lot about our attitudes at the beginning of the 21st century.

Reviews and Inspections

One of the great breakthroughs in software engineering was Gerald Weinberg's concept of egoless programming—the idea that no matter how smart a programmer is, reviews will be beneficial. Michael Fagan formalized Weinberg's ideas into a well-defined review technique called Fagan inspections. The data in support of the quality, cost, and schedule impact of inspections is overwhelming. They are an indispensable part of engineering high-quality software. I propose Fagan inspections as one of the 10 best influences.

Chris Ebert: Inspections are surely a key topic, and with the right instrumentation and training they are one of the most powerful techniques for defect detection. They are both effective and efficient, especially for upfront activities. In addition to large-scale applications, we are applying them to smaller applications and incremental development.

Terry Bollinger: I think inspections merit inclusion in this list. They work, they help foster broader understanding and learning, and for the most part they do lead to better code. They can also be abused—for instance, in cases where people become indifferent to the skill set of the review team ("we reviewed it, so it must be right"), or when they don't bother with testing because they are so sure of their inspection process.

The Best Influences on Software Engineering

- Reviews and Inspections
- Information Hiding
- Incremental Development
- User Involvement
- Automated Revision Control
- Development Using the Internet
- Programming Languages Hall of Fame: Fortran, Cobol, Turbo Pascal, Visual Basic
- Capability Maturity Model for Software
- Object-Oriented Programming
- Component-Based Development
- Metrics and Measurement

Robert Cochran: I would go more basic than this. Reviews of all types are a major positive influence. Yes, Fagan inspection is one of the most useful members of this class, but I would put the class of inspections and reviews in the list rather than a specific example.

Information Hiding

David Parnas's 25-year-old concept of information hiding is one of the seminal ideas in software engineering—the idea that good design consists of identifying design secrets that a program's classes, modules, functions, and even variables and named constants should hide from other parts of the program. Whereas other design approaches focus on notations for expressing design ideas, information hiding provides insight into how to come up with good design ideas in the first place. Information hiding is at the foundation of both structured and object-oriented design. In an age when buzzword methodologies often occupy center stage, information hiding is a technique with real value.

Terry: Hear! Hear! To me this technique has done more for real, bottom-line software quality than almost anything else in software engineering. I find it worrisome that folks sometimes confuse it with abstraction and other concepts that don't enforce the rigor of real information hiding. I find it even more worrisome that some process-oriented methodologies focus so much on bug counting that they are oblivious to why a program with good information hiding is enormously easier to debug—and has fewer bugs to begin with—than one that lacks pervasive use of this concept.

Chris: Surely agreed; it's far too often missing in current designs.

Incremental Development

The software engineering literature of the 1970s was full of horror stories of software meltdowns during the integration phase. Components were brought together for the first time during system integration. So many mistaken or misunderstood interface assumptions were exposed at the same time that debugging a nest of intertwined assumptions became all but impossible. Incremental development and integration approaches have virtually eliminated code-level integration problems on modern software projects. Of these incremental approaches, the daily build is the best example of a real-world approach that works. It minimizes integration risk, provides steady evidence of progress to project stakeholders, keeps quality levels high, and helps team morale because everyone can see that the software works.

Terry Bollinger

Chris Ebert

Dave Card: I wouldn't include daily builds as one of the best influences. I see it as a brute-force method of integration. It's kind of like having a housing contractor confirm that each window is the right size by holding it up to a hole in the wall, and then chiseling or filling the hole as necessary to make the window fit.

Terry: I disagree with Dave. Incremental integration, including daily builds, represents a retreat from the earlier, ultimately naive view that everything could be defined with mathematical precision up front, and all the goodies would flow from that. Such philosophies overlooked the inherent limitations of the human brain at foreseeing all the implications of a complex set of requirements.

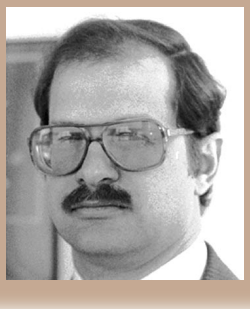
If you really want to see the daily-build approach in ferocious action, take a look at the open-source community. When things get busy there, releases might even be hours apart. Such fast releases also reduce the human setup time for returning to the context of each problem.

Steve McConnell: At a level of abstraction higher than daily builds, Barry Boehm's spiral life-cycle model is an incremental approach that applies to the whole project. The model's focus on regular risk reduction is appealing, but the model is so complicated that only experts can use it. I'd give it the prize for the most complicated and least understood software engineering diagram of the 20th century. Unlike daily builds, it seems like a great incremental-development approach that has had little impact in practice.

Terry: I disagree. The daily build you touted earlier really is closer to the spiral model than to traditional

phased design. And where do you think interface prototyping goes? It surely is not in the original phased-development models of the '70s and '80s. Barry Boehm noticed that reality was closer to the daily-build model than to the rigidly defined phased models of that era.

Chris: I too disagree with Steve's conclusion. The spiral-model diagram was too complex because it tried to put everything into one picture. But the combination of risk management and increments that the spiral model proposes are highly successful when blended together.



Dave Card

Robert L. Glass: In my experience, the spiral model is the approach that most good software people have always used; they just didn't call it that. Good programmers never did straight waterfall; they might have pretended they did to appease management, but in fact there was always a mix of "design, code, test a little" thrown in along the way.

User Involvement

We've seen tremendous developments in the past several years in techniques that bring users more into the software product design process. Techniques such as joint application development sessions, user interface prototyping, and use cases engage users with product concepts in ways that paper specifications simply cannot. Requirements problems are usually listed as the number-one cause of software project failure; these techniques go a long way toward eliminating requirements problems.

Wolfgang Strigel: I think user interface prototyping is especially important because it addresses the fundamental engineering concept of building models. No architect would build a bridge without a model to see if those who pay for it will like it. It's really simple: if you plan on ignoring the customer, don't prototype.

Karl Wiegers: Use cases, sensibly applied, provide a powerful mechanism for understanding user requirements and building systems that satisfy them.

Steve Mellor: Whatever techniques are applied, we need to be clear about context. When we talk about use cases, for example, they can be very helpful in a context where there is little communication about the problem. Use cases put the software folks face to face with the clients and users, and put them in a position to learn the vocabulary of the problem. But use cases are appalling at helping to create an appropriate set of abstractions. Should you use use cases? Depends on your context.

Terry: I think these techniques tie back to the value of incremental development. And this includes all forms of prototyping—communication protocols, subsystem interfaces, and even the method interfaces to an object class. There's a lot of merit to trying out different approaches to discover what you missed, misunderstood, or got out of proportion.

Automated Revision Control

Automated revision control takes care of mountains of housekeeping details associated with team programming projects. In the Mythical Man-Month *in 1975, Fred Brooks' "surgical team" made use of a librarian. Today, software handles that person's function. The efficiencies achieved by today's programming teams would be inconceivable without automated revision control.*

Terry: The promise is high, and the fundamental concept is solid. I don't think our associated methods for organizing large numbers of programmers efficiently are good enough yet to make the value of this technology really shine. For example, the really interesting code still tends to get written by relatively small teams for whom the benefits of auto-

mated revision control are much less conspicuous—for example, many of the open-source efforts.

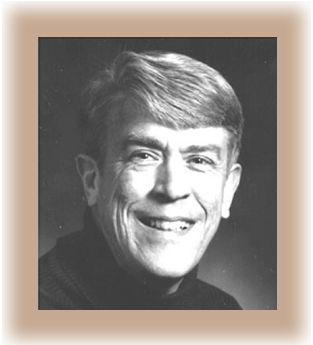
Perhaps the most important software engineering innovations of the next century will come not from more intensive methods for making people more organized, but rather from learning to automate the overall software process in ways that we do not currently understand very well, but that revolve around the overall ability of a computer to remember, communicate, and organize data far better than people do.

I think this kind of innovation will be closely related to methods such as automated revision control. Our current planning tools, which too often just try to automate paper methods, certainly don't seem to be there yet.

Development Using the Internet

What we've seen with open-source development is just the beginning of collaborative efforts made possible by the Internet. The potential this creates for effective, geographically distributed computing is truly mind-boggling.

Terry: Internet interactions let interesting sorts of interactions happen, and not just with source code. I think this one is important simply because it isn't very easy to predict—for example, it might lead to approaches that are neither proprietary nor open source, but that are effective in ways we don't fully foresee or understand yet.



Robert Glass

Wolfgang: Yes, although it stands on the foundation of best practices in software design and development of the 20th century, it brings new aspects that will change the face of software development as we know it.

Programming Languages Hall of Fame: Fortran, Cobol, Turbo Pascal, Visual Basic

A few specific technologies have had significant influence on software development in the past 40 years. As the first widely used third-generation language, Fortran was influential, at least symbolically. Cobol was arguably at least as influential in practice. Originally developed by the US Defense Department for business use, most practitioners have long forgotten Cobol's military origins. Cobol has recently come under attack as being responsible for Y2K problems, but let's get serious: Y2K is only a Cobol problem because so much code has been written in Cobol. Does anyone really think programmers would have used four-digit instead of two-digit years if they had been coding in Fortran instead of Cobol?

Chris: The real added value of Cobol was to make large IT systems feasible and maintainable, and many of those are still running. The "problems" with Cobol weren't really about Cobol. The problems were with the programmers who happened to be using Cobol. Today, many of these same programmers are using or abusing Java and C++, and the legacy-problems discussion will surely come back someday with regard to those languages.

[*IEEE Software*'s next issue (March/April 2000) will focus on recent developments and future directions in Cobol. —*Editor*]

Turbo Pascal, the first integrated editor-compiler-debugger, forever changed computer programming. Prior to Turbo Pascal, compilation was done on disk, separate from editing. To check his work, a programmer had to exit his editing environment, compile the program, and check the results. Turbo Pascal put the editor, compiler, and debugger all in memory at the same time. Making compile-link-execute cycles almost instantaneous shortened the feedback loop between programmers' creating code and executing the code. Programmers became able to

Editorial: All submissions are subject to editing for clarity, style, and space. Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *IEEE Software* does not necessarily constitute endorsement by the IEEE or the IEEE Computer Society.

To Submit: Send 2 electronic versions (1 word-processed and 1 postscript or PDF) of articles to Magazine Assistant, *IEEE Software*, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314; software@computer.org. Articles must be original and not exceed 5,400 words including figures and tables, which count for 200 words each.

IEEE Software

experiment in code more effectively than they could in older, more batch-oriented environments. Quick turnaround cycles paved the way for effective code-focused development approaches, such as those in use at Microsoft and touted by Extreme Programming advocates.

Terry: I think the real message here is not Turbo Pascal, but rather the inception of truly integrated programming environments. With that broadening (integrated environments, with Turbo Pascal as the first commercial example), I would agree that this merits inclusion in the top 10.

Wolfgang Strigel

Chris: Aside from better coding, such programming environments also made the incremental approaches that we discussed earlier feasible.

Academics and researchers talked about components and reuse for decades, and nothing happened. Within 18 months of Visual Basic's release, a thriving market for prebuilt components had sprung from nothing. The direct-manipulation, drag-and-drop, point-and-click programming interface was a revolutionary advance.

Chris: Even more relevant is the single measure of speed of penetration—VB is one of the fastest growing languages and is still going strong. Only Java has passed it in speed of adoption.

Terry: VB provided the benefits promised (but for the most part never really delivered) by object-oriented programming, mainly by keeping the "visual" part of the object paradigm and largely dispensing with the complex conversion of objects into linguistic constructions. VB is limited, but within those limits it has had a profound impact on software development and has made meaningful development possible for a much larger group of people than would have been possible without it.

Kudos to Microsoft for this one—and that's coming from a guy who coedited the Linux issue!

Capability Maturity Model for Software (SW-CMM)

The Software Engineering Institute's SW-CMM is one of the few branded methodologies that has had any effect on typical software organizations. More than 1,000 organizations and 5,000 projects have undergone SW-CMM assessment, and dozens of organizations have produced mountains of compelling data on the effectiveness of process improvement programs based on the SW-CMM model.

Dave: I would consider the software CMM to be both a best influence and a dead end. Steve has explained the case for a best influence. However, I have also seen a lot of gaming and misuse of the CMM that led to wasted and counterproductive activity. Even worse, it has spawned a growth industry in CMMs for all kinds of things—apparently based on the premise that if your model has five levels it must be right, regardless of whether you have any real knowledge of the subject matter. It reminds me of the novel, *Hitchhiker's Guide to the Galaxy*, where the answer was 42—except that now it's 5.

Nancy Mead

One of the unanticipated effects of the CMM has been to make the effective body of software engineering knowledge much shallower. My expectation was that an organization that was found to be weak in risk management, for example, would read one of the many good books on risk management or get help from an expert in the field. Instead, many (if not most) organizations take the four pages in the CMM that describe risk management and repackage it as four to six pages of their local process documentation.

Overall, I believe the CMM has helped a lot. However, I think it does have some harmful side effects that can be mitigated if the community is willing to recognize them. Moreover, there are limits to how far it can take us into the 21st century.

Nancy Mead: I don't think the CMM per se is significant—it's the recognition of a need for standardized software processes, whatever they might be.

Robert: Standardization is key. Another example is the recent effort to put a good structure on rapid application development using the Dynamic Systems Development Method. DSDM is not well known in the US, but interest is rapidly growing on this side of the Atlantic.

And don't forget the many thousands who have used ISO 9000 as their SPI framework. I would say that SPI using any of the recognized frameworks (or indeed a mixture of them) is where benefit is gained. They give a structure and coherence to SPI activity.

Terry: I do think that some concept of process improvement probably belongs in this list, including both ISO 9001-3 and maybe the first three levels of CMM. But I disagree that the CMM taken as a whole has been one of the most positive influences of the 20th century.

Wolfgang: This one can really stir the emotions. Despite its simplicity, the main problem with the CMM is its rigid and ignorant interpretation. The concept itself is sort of a truism. I believe the biggest accomplishment of CMM is in the Software Engineering Institute's success in marketing the concept. It certainly has raised the awareness within thousands of companies about "good software engineering practices." Universities were woefully slow in recognizing software engineering as a topic, and the SEI jumped into the fray, educating not only new grads but old professionals

as well. Its problems get aggravated with people reading too much into it. The grading is questionable and not the most relevant part of the CMM. I disagree that ISO 9000-3 would merit a similar place in the hall of fame. After all, with ISO 9000-3 you can diligently do all the wrong things, and you will be compliant as long as you do them diligently.

It may be time to throw away the old CMM model and come up with an update that takes into account the experiences from the last 10 years. I don't think we need a new model or more models; just taking the cobwebs out of the current one will carry us for the next 10 years.

Chris: Despite the argument about ISO 9000, the CMM is definitely the standard that sets the pace. At present, it is the framework that allows benchmarking and that contributes most to solving software engineering crises—just by defining standard terminology and condensing industry best practices. This is my personal favorite "best influence."

Object-Oriented Programming

Object-oriented programming offered great improvements in "natural" design and programming. After the initial hype faded, practitioners were sometimes left with programming technologies that increased complexity, provided only marginal productivity gains, produced unmaintainable code, and could only be used by experts. In the final analysis, the real benefit of OO programming is probably not objects per se, but the ability to aggregate programming concepts into larger chunks than subroutines or functions. This is certainly one of the strongest influences. Is it one of the best?

Karl Wiegers

Terry: Whatever real benefits people got from OO, they mostly came from the use of traditional Parnas-style information hiding, and not from all the other accoutrements of OO.

OO is a badly mixed metaphor. It tries to play on the visual abilities of the human brain to organize objects in a space, but then almost immediately converts that over into an awful set of syntax constructions that don't map well at all into the linguistic parts of our brains. The result, surprise, surprise, is an ugly mix that is usually neither particularly intuitive nor as powerful as it was originally touted. Has anyone done any really deep inheritance hierarchies lately? And got them to actually work without ending up putting in more effort than it would have taken to simply do the whole thing over again?

Steve Mellor

Chris: Agreed, because many people haven't taken the time to understand OO completely before they design with it. In a short time, it has created more harm to systems than so-called "bad Cobol programming."

Takaya Ishida: I think it is questionable whether Cobol and OO should be on the 10 Best List or 10 Worst List. It seems to me that Cobol and other popular high-level procedural languages should be blamed for most of the current problems in programming. It is so easy to program in such languages that even novice programmers can write large programs—but the result is often poorly structured. With OO languages such as Smalltalk, programmers take pains to learn how to program, and the result is well-structured programs. Having programming approaches that can be used only by experts may be a good idea. Our past attitude of easy-going programming is a major cause of the present software crisis, including the Y2K issue.

Terry: I really like Smalltalk-style OO, and I'm optimistic about Java. The real killer was C++, which mixed metaphors in a truly horrible fashion and positively encouraged old C programmers to create classes with a few hundred methods in them, to make a system "object oriented." Argh! So maybe the culprit here is more C++ than OO, and there's still hope for object-oriented programming.

Component-Based Development

Component-based development has held out much promise, but aside from a few limited successes, it seems to be shaping up as another idea that works better in the laboratory than in the real world. Component-version incompatibilities have given rise to massive setup-program headaches, unpredictable interactions among programs, de-installation problems, and a need for utilities that restore all the components on a person's computer to "last known good state." This is one of the best prospects for the 21st century, but probably not for the 20th.

Robert: The reuse (component-based) movement might have trouble getting off the ground, but some related developments have shown great promise. Design patterns are a powerful medium for capturing, expressing, and packaging design concepts and artifacts—design components, if you will. The patterns movement looks to be the successful end-run around the reuse movement. Similarly, we've been very successful in domain generalization. Compiler-building was perhaps the first such success many decades ago.

Steve: Yes, 30 years ago a person could get a PhD just for writing a compiler, but today compiler-building domain generalization has been such a success that compiler building is usually taken for granted.

Robert: There have been too few successes other than compiler build-

ing. I would assert that the enterprise resource planning system, such as SAP's, is the most recent and most successful example of building general-purpose software to handle a whole domain—in this case, back-office business systems.

Robert Cochran

Terry: The components area has indeed been a major headache, but it's sort of an ongoing problem, not one that has been fully recognized as a dead end. Visual Basic is an example of how component-based design can work if the environment is sufficiently validated ahead of time.

The fully generic "anything from anybody" model is much more problematic, and is almost certainly beyond the real state of the art—versus the marketing state of the art, which generally is a few years to a few centuries more "advanced" than the real state of the art.

Chris: Work on components helped to see the direction toward frameworks, reuse, and pattern languages. So it is a useful intermediate step. Thinking in components is still a good design principle and entirely in line with the point made earlier about information hiding.

Takaya Ishida

Metrics and Measurement

Metrics and measurement have the potential to revolutionize software engineering. In the few instances in which they have been used effectively (NASA's Software Engineering Lab and a few other organizations), the insights that well-defined numbers can provide have been amazingly useful. Powerful as they can be, software process measurements aren't the end; they're the means to the end. The metrics community seems to forget this lesson again every year.

Dave: I would include metrics as a dead end, but I also would include measurement in the list of top prospects for the 21st century. The word "metrics" doesn't appear in most dictionaries as a noun—and that is a good indication of how poorly measurement is understood and why it is consequently misused in our discipline. The ability to express things meaningfully in numbers is an essential component of any engineering discipline, so I don't see how we can have software engineering in the 21st century without measurement.

Deependra Moitra: Any problems we have in this area are not with metrics but with the metrics community. We cannot blame an idea, concept, or approach if people mess up using it. I can't imagine performing my job without metrics, especially in a geographically distributed development mode.

Maarten Boasson

Terry: Metrics are a problem. Our foundations are badly off in this area, because we keep trying to apply variations of the metrics that worked well for manufacturing to the highly creative design problems of software. It's a lot like trying to teach people how to write like Shakespeare by having them constantly check their words for spelling errors.

Chris: Entirely true, but still metrics as such are of big value for the practical management of teams and projects. We should see metrics as a tool (or means). Metrics usage has matured dramatically over the past 10 years. A metrics discipline in software engineering is as necessary as theoretic foundations are in other engineering disciplines to avoid software development approaches that are built on nothing more than gut feeling.

Wolfgang: Regardless of terminology (metrics or measurements), I believe that this should become another top contender for the next century. How can we claim to be professionals if nobody can measure some basic facts about the development activity? How can we manage activities that cannot be measured? Measurement failed in this century not because it is a bad concept, but because our profession is still in its infancy. There is no other engineering discipline that would not measure product characteristics. And there is no other manufacturing activity that would not try to measure output over input (which equals productivity). The concept of measurement isn't at fault; the problem is that consumers have become too tolerant of faulty software and a red-hot marketplace in which inefficiencies do not cause companies to fail. If there were an oversupply of software developers, we would see a lot more measurement applied to optimize the process.

Summary

Maarten Boasson: The greatest problem in the software engineering community is the tendency to think that a new idea will solve all problems. This tendency is so strong that previously solved problems are forgotten as soon as a new idea gains some support, and consequently problems are re-solved in the new style. This is a perfect recipe for preventing progress!

This behavior suggests (to me, at least) that the field of software engineering, and even computer science, is very immature. This is not anybody's fault, but the simple result of

the very short time that computing has been around. Compared to other fields, we really are hardly out of the nursing stage. What is collectively our fault, however, is the religious fanaticism with which we protect our current beliefs against perceived attacks from people doing things differently. Such behavior probably also existed during the early years of other disciplines, so why don't we learn from their example?

The problems in developing today's and tomorrow's systems are overwhelming; they require many different types of problems to be solved. No other scientific or engineering discipline relies on a single technique for addressing problems, so why are we, so-called professional engineers (and computer scientists), stupid enough to think that our field is fundamentally different in this respect?

So, what do we need to do? First, industrial management has to understand that software engineering is not an engineering discipline like so many others (yet) and that standards, methods, and tools are all likely to be wrong (once we really understand what developing software means). This implies that software development is still much more of an art than management believes it to be and that, therefore, only the best and the brightest should be allowed to develop systems of more than trivial complexity. A side effect of such an approach would be a significant reduction in effort needed for their development.

Second, we should carry out real experiments with novel and alternative approaches to software development, possibly as a cooperative effort between industry and academia. These should not be simplified, typical academic problems, but real-life systems. Such experiments will have to be performed in parallel with the traditional development of the same systems; indeed, no manager will accept the risk involved otherwise (although I would guess the risk is generally smaller than the traditional approach). We should analyze and widely publish the results—whichever way they turn out.

Third, we should reconsider the metric for academic success. The "publish or perish" paradigm guarantees that large amounts of garbage will be published. Also, the really interesting results in multidisciplinary work are practically unpublishable, due to the strong segmentation of the field, with superexperts in epsilon-width fields.

Fourth, we (*IEEE Software*, for example) should continuously stress that problems have not yet been solved, that innovation is essential, and that all claims to having invented THE solution are simply wrong.

Maarten Boasson is head of the Applied Systems Research Department at Hollandse Signaalapparaten in Hengelo, The Netherlands, and professor of industrial complex computer systems at the University of Amsterdam; boasson@signaal.nl.

Terry Bollinger is an information systems engineer at The MITRE Corporation in Reston, Virginia; terry@mitre.org.

Robert Cochran is the managing director of Catalyst Software and director of the Center for Software Engineering at Dublin City University in Dublin, Ireland; robert@cse.dcu.ie.

Dave Card is the technical lead for software measurement at the Software Productivity Consortium in Virginia; card@software.org.

Chris Ebert is the software process manager of Alcatel's Switching and Routing Division in Antwerp, Belgium; christof.ebert@alcatel.be.

Robert Glass is president of Computing Trends, based in Bloomington, Indiana, publisher–editor of *The Software Practitioner*, and editor-in-chief of the *Journal of Systems and Software*; rglass@indiana.edu.

Takaya Ishida is senior chief researcher of corporate research and development at Mitsubishi Electric Corporation in Tokyo, Japan; takaya.ishida@hq.melco.co.jp.

Nancy Mead is team leader for survivable network analysis in the Networked Survivable Systems Program at the Software Engineering Institute, Carnegie Mellon University, in Pittsburgh; nrm@sei.cmu.edu.

Steve Mellor is the founder of Project Technology, based in Tucson, Arizona, and codeveloper with Sally Shlaer of the OO Analysis and Recursive Design method; steve@projtech.com.

Deependra Moitra is general manager of quality at the India Product Realization Center of Lucent Technologies in Bangalore, India; dmoitra@lucent.com.

Wolfgang Strigel is founder and president of the Software Productivity Centre in Vancouver, Canada; wstrigel@spc.ca.

Karl E. Wiegers is principal consultant with Process Impact, a software process consulting and education company in Rochester, New York; kwiegers@acm.org.

The views expressed are the participants' and do not necessarily reflect the views of their organizations.

IEEE Software

How to Reach Us

Writers

For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org), or access computer.org/software/author.htm.

Letters to the Editor

Send letters to

Letters Editor
IEEE Software
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
cbaltes@computer.org

Please provide an e-mail address or daytime phone number with your letter.

On the Web

Access computer.org/software for information about *IEEE Software*.

Subscription Change of Address

Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software*.

Membership Change of Address

Send change-of-address requests for the membership directory to directory.updates@computer.org.

Missing or Damaged Copies

If you are missing an issue or you received a damaged copy, contact membership@computer.org.

Reprints of Articles

For price information or to order reprints, send e-mail to software@computer.org or fax +1 714 821 4010.

Reprint Permission

To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.