Making a Difference in the Software Century

Barry Boehm
University of Southern California

In the 21st century, software engineers face the often formidable challenges of simultaneously dealing with rapid change, uncertainty and emergence, dependability, diversity, and interdependence, but they also have opportunities to make significant contributions that will make a difference for the better.

I feel very lucky to have been born in the US in the 1930s and to have had a chance to participate in the formation of a whole new discipline of software engineering. I think those of you just entering the software engineering field have an even more exciting prospect ahead of you. I believe that at least the next few decades will make the 21st century the Software Century. Software will be the main element that drives our necessary capabilities and quality of life, and people who know how best to develop software-intensive systems will have the greatest opportunity to make a difference in the results.

This will be very satisfying for software engineers, but it will impose large responsibilities to provide excellence in the software-intensive systems that are developed and in the services that they provide. Here are the main challenges that I believe 21st-century software engineers will need to address: increasingly rapid change, uncertainty and emergence, dependability, diversity, and interdependence.

RAPID CHANGE: AVOID THWADI

Gone are the days when people could spend their entire careers doing the same thing in the same ways. A strong case can be made that the most significant challenges facing 21st-century organizations will be their ability to adapt to rapid and unpredictable change in more rapid and appropriate ways than their competitors. Evidence of this accelerated pace of change can be seen in Figure 1, showing adoption of new technologies, and in such books as *The World Is Flat*,[1] showing the change in consumer preferences, international competition, and business practices.

All of this change comes at a price. People enjoy the fruits of change but generally dislike having to modify their behavior. Software engineers concerned with improving their process maturity often think that Level 5 (optimizing) is where they become mature and optimize their processes for all time. This runs the risk of being overoptimized and going the way of the dinosaurs.

There will be a lot of tensions between people and organizations rapidly adapting to change and those who prefer not to. A good example nowadays is the interaction between software developers trying to be adaptive to change within fixed-price, build-to-specification software contract structures determined by administrators practicing THWADI (That's How We've Always Done It). In the commercial world, better contact structures such as "shared destiny" models are being developed; getting these refined and more generally adopted will be important to software success.

Of course, some THWADI is good. We will need to separate obsolete practices from enduring principles that need to be conserved.

Some other implications for software engineers' careers are that learning how to learn will be more important

Excerpted with permission from *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*, R.W. Selby, ed., copyright © 2007, IEEE CS Press-John Wiley & Sons.

Published by the IEEE Computer Society
0018-9162/08/$25.00 © 2008 IEEE

than learning things, and that some goals, such as trying to determine "the" best architecture for a system, will be better thought of as concurrently determining architecture baselines along with architecture evolution processes. The evolution of the Arpanet architecture into the architecture of the Internet is a good example.

UNCERTAINTY AND EMERGENCE: CONSIDER BITAR AND IKIWISI

Rapid change can be relatively predictable, as in the case of Moore's law, which has consistently predicted a doubling of computer power every 18 months for decades, and even has a relatively predictable point a dozen or so years from now at which the doubling will slow down and eventually hit limits. When rapid change is predictable, we can plan for it and organize the architecture to accommodate it, either via planned upgrades or by defining modules to encapsulate sources of change, so that the side effects of the changes are limited to one module, as described in David L. Parnas's "Designing Software for Ease of Extension and Contraction."[2]

However, many sources of change are relatively unpredictable, such as the emergence of new technologies (global positioning satellites or the World Wide Web), changes in consumer demand patterns, changes in leadership personnel with differing priorities, or changes in desired user interface characteristics that are not prespecifiable but emerge with use. Frequently, when users are asked to specify in advance how they would like to interact with a new application, they will provide the IKIWISI (I'll Know It When I See It) response.

In such cases, using a sequential, requirements-specification-first waterfall model will generally be a recipe for a nonresponsive system and a great deal of expensive rework. It will be better to accept that your project will start at the left of a Cone of Uncertainty, as shown in Figure 2. This figure was derived from several years of experience in preparing cost estimates at various stages of system definition at TRW.[3] It was dubbed the Cone of Uncertainty by Steve McConnell.[4]

The best way to narrow a Cone of Uncertainty is to use the BITAR (Buy Information To Avoid Risk) strategy. This involves identifying candidate investments in reducing uncertainty such as prototyping, simulating, modeling, benchmarking, reference checking, COTS evaluation, or market trend analysis; analyzing their relative

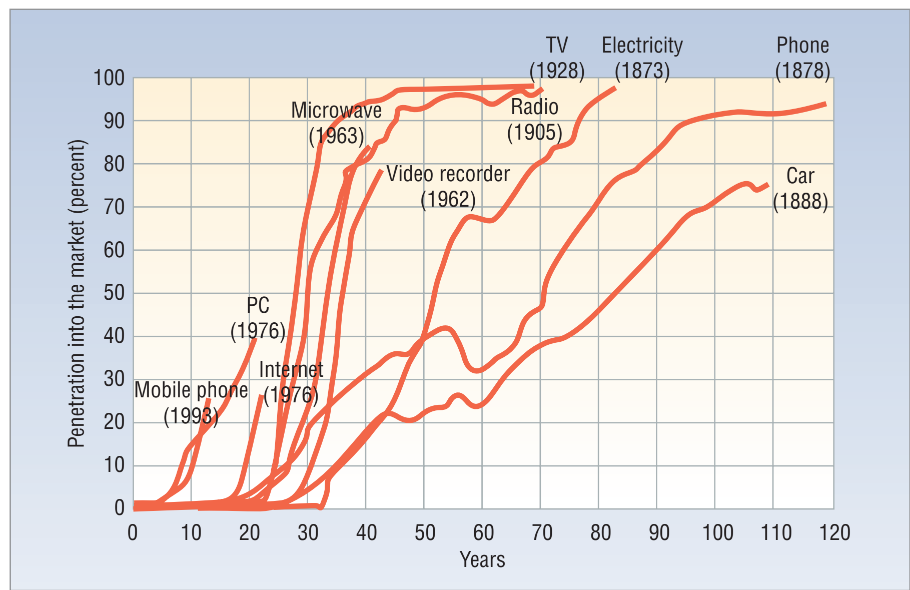


Figure 1. Accelerating the pace of change (courtesy of Microsoft).

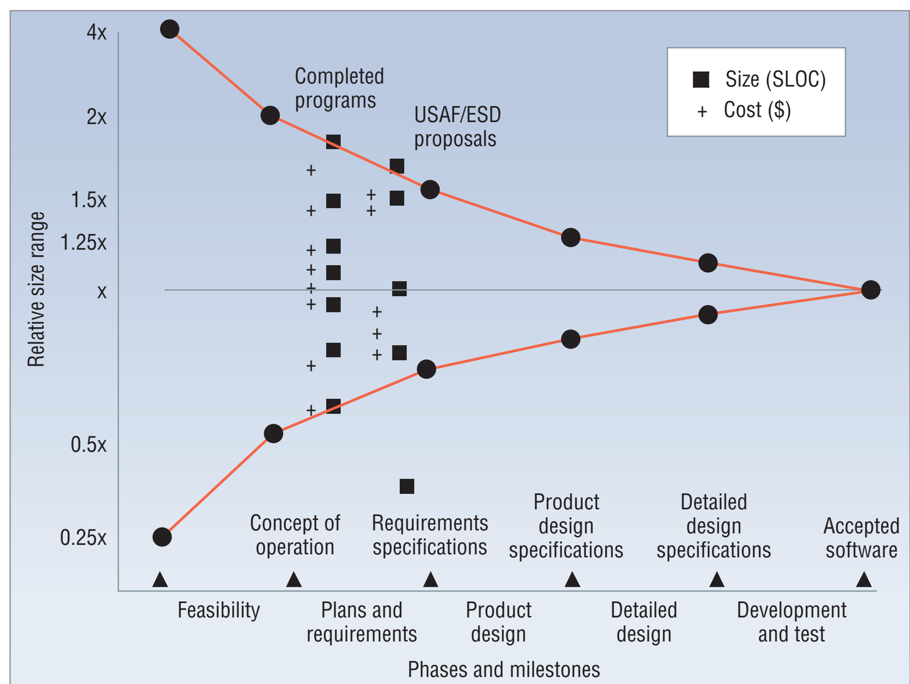


Figure 2. The Cone of Uncertainty in software cost and size estimation.

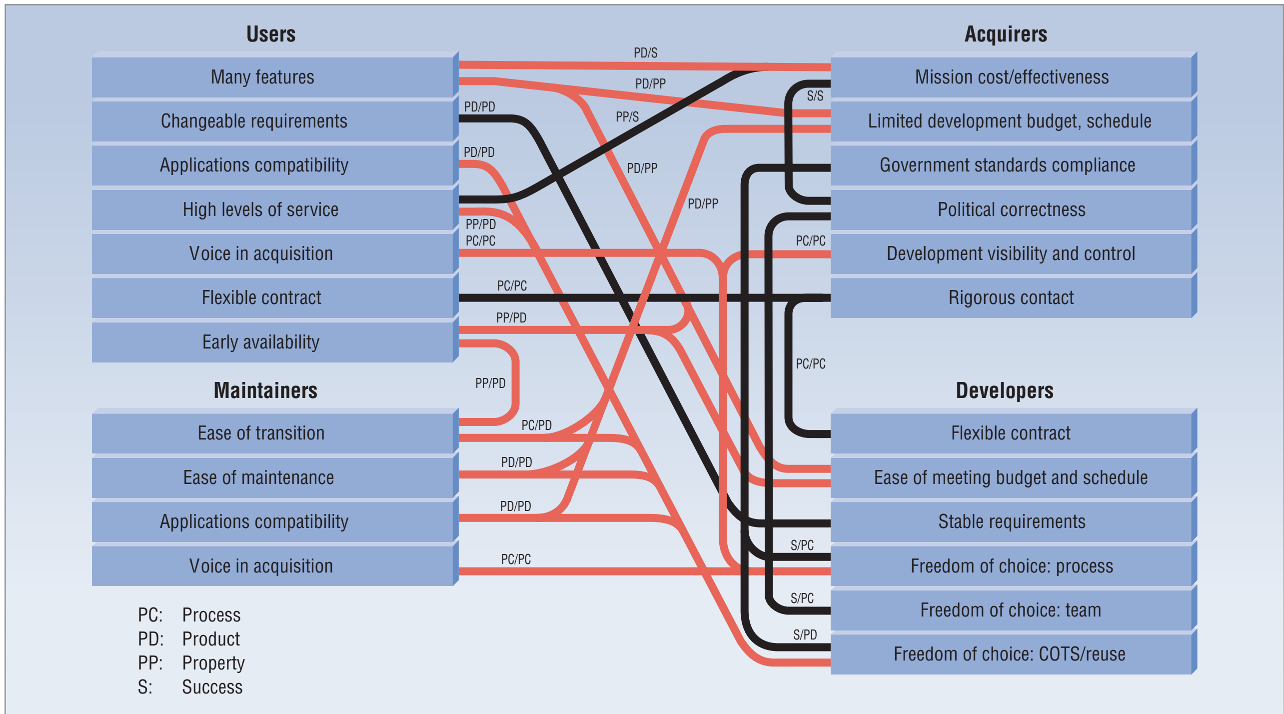


Figure 3. Key-stakeholder value propositions and their potential conflicts. (The red lines show model clashes from the MasterNet system.)

costs and benefits; and choosing the most cost-effective ways to reduce your uncertainty and risk.

However, it will also be important to organize future projects with enough agility to be able to adapt to the additional emergence of new sources of unpredictable change. This involves building agility and adaptability into your projects' staffing, organization, product architectures, and process architectures.

DEPENDABILITY: CONSIDER DAVAS

Along with improving agility, future projects will need to improve the dependability of the software they produce, as software is becoming the dominant source of competitive differentiation in organizations' products and services. As they consider DAVAS (Dependability As Value Assured to Stakeholders), simultaneously achieving and improving agility and dependability will be one of the biggest challenges for 21st-century software engineers.

When I was working on the NASA High-Dependability Computing program with Vic Basili and others, we found that one of the biggest challenges was just defining "dependability." There were several cases in NASA and elsewhere in which dependability was equated with system mean time between failures or system uptime, where this emphasis led to the loss of other system properties that users were counting on. In one case, the system administrators were incentivized on uptime and ran the system in a way that produced very high uptime, but also produced delays of up to two weeks for users depending on rapid response when requesting data or services from the system.

This indicates that when you need to produce a dependable system, it will be important to identify your most success-critical stakeholders and what value propositions their success most depends on. Figure 3 shows that these stakeholder value propositions are likely to conflict with each other, even for the four main stakeholders in a software-intensive system. The end users' success will depend on the system having many features whose nature and priorities they can change at any time; operating smoothly, quickly, and reliably; being compatible with their other applications; and being available right away. The sales users' success will depend on having fancy displays that entice customers. The administrative users' success will depend on having extensive self-monitoring, audit trails, and trend analysis capabilities.

The acquirers will have limited resources, and their success will depend on having a business case for each feature and having a completely stabilized set of requirements. The developers' success will depend on minimizing the risk of overrunning the budget and schedule, which will imply having the ability to make infrastructure decisions and to reuse previously developed software. The maintainers' success will depend on how well the system and its software are designed and structured for ease of maintenance and how compatible they are with the other artifacts being maintained.

The red lines in Figure 3 show actual conflicts among stakeholder dependability value propositions that were not resolved in one of the classic failed software projects, the Bank of America MasterNet system.[5] The black lines

Table 1. Cultural differences between the US and Thailand.

Country	Power distance		Uncertainty avoidance		Individualism		Masculinity		Long-term orientation	
	Index	Rank	Index	Rank	Index	Rank	Index	Rank	Index	Rank
US	40	38	46	43	91	1	62	15	29	27
Thailand	64	21-23	64	30	20	39-41	34	44	56	8

were additional conflicts we found in analyzing other failed projects. The S, PD, PC, and PP annotations on the lines indicate whether a line primarily reflected conflicts among the project's success models, product models, process models, or property models. One implication of the diversity of model clashes on failed projects is that current "model-driven development" initiatives need to consider more than just product models. A 2003 empirical study on model clashes found that product-product model clashes were only about 30 percent of the total number of model clashes found.[6]

The fact that there are so many potential conflicts among even the four main success-critical stakeholders means that there are many potential win-lose situations that a project can get into. A win-lose situation usually turns into a lose-lose situation, as happened with the cancellation of the MasterNet project when the acquirers found that their budget was being overrun, the users were unhappy with the system's performance, and the maintainers were unhappy with the incompatibility of the developer's Prime Computer solution with their other IBM mainframe applications.

In such a situation, it is important to manage stakeholders' expectations, invest more effort into ensuring feasibility of the proposed solution with respect to all the stakeholders, and to negotiate a mutually satisfactory or win-win set of specifications, plans, and resources before proceeding into development. When negotiating requirements, it is better in the early stages to replace words that have nonnegotiable connotations such as "requirements" (things claimed by right and authority) with words like "objectives" and "goals."

Finally, to converge on a software-intensive system that is mutually satisfactory to a diverse set of stakeholders in such areas as banking, medicine, public services, manufacturing, transportation, or commerce, it is increasingly important for software engineers to be knowledgeable not only about software concepts and techniques but the concepts and techniques of the organizations using the software. In the 21st century and beyond, there will still be important career paths for people developing the digital infrastructure, but it will be people understanding the tradeoffs between software feasibility and applications-domain feasibility that will be most influential in architecting and developing high-value software-intensive systems.

DIVERSITY: AVOID OSUFA

The NASA example indicates that a value-neutral, one-size-uniformly-fits-all (OSUFA) definition of dependability as system uptime is often unlikely to succeed for other success-critical stakeholders. This conclusion becomes even more pronounced in considering the trends toward cross-cultural globalization described in such books as *The World Is Flat*.[1]

An example in the area of software capability maturity models is that in Thailand: Only 17 of 380 software-developing companies decided to use the US-developed Software Capability Maturity Model (CMM), and only three of the 17 companies proceeded to increase their CMM level. Much of the difference in usage can be explained by the differences in the more individualistic, masculine, short-term-oriented US culture and the more community-oriented, feminine, long-term-oriented Thai culture, as shown in Table 1.[7]

Some other cultural differences that cause difficulties for OSUFA practices are provided in E.T. Hall's *Beyond Culture*.[8] One example is the difference between polychromatic cultures, in which interruptions during task performance such as blinking e-messages and pop-up windows are welcomed, and monochromatic cultures in which such interruptions cause frustration as people try to focus on the closure of one task at a time. Hall also found that contracts in a high-content culture such as the US averaged 10 times as many words as contracts in a high-context culture such as France.

OSUFA contracting practices and process standards are a particularly poor match to another 21st-century trend toward large software-intensive systems of systems (SISOS). These types of systems add scalability to the simultaneous challenges of achieving high agility and high dependability.

Figure 4 compares the average change request processing times for two recent large-scale SISOS using OSUFA high-content contracting instruments and processes. Having to deal with many contract-affected changes that average 141 workdays to achieve closure is not a recipe for success in developing change-responsive and on-schedule large SISOS.

"Some Future Trends and Implications for Systems and Software Engineering Processes" provides some alternatives to OSUFA processes and contracting practices for current and future large-scale SISOS.[9]

INTERDEPENDENCE: CONSIDER TANIA

In large-scale SISOS, the component systems no longer have the luxury of operating autonomously. They act on information from other SISOS elements and supply information to them. Whereas before they may not have been concerned with such issues as interoperability and network security, now these issues are necessary considerations. Each component system is no longer an island unto itself, and a SISOS is not an island either. Many of them must interoperate with more than 100 separately evolving systems with which they need to stay synchronized.

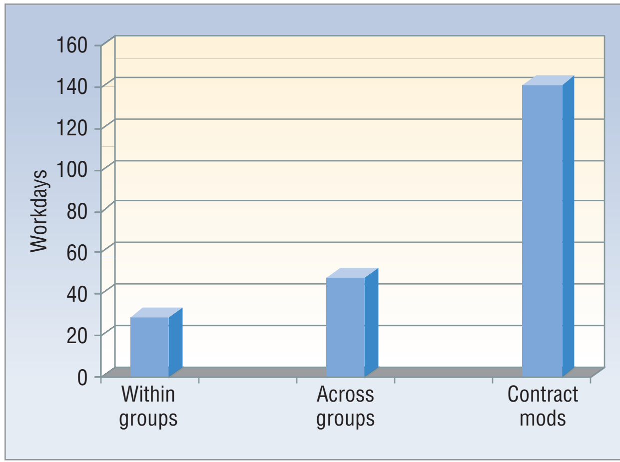


Figure 4. Average change request processing times for two large SISOS in workdays.

Beyond this, it is important to recognize that TANIA (There Are No Islands Anymore) applies not only to the SISOS and its elements, but also to the processes, methods, and tools used to define, design, develop, deploy, and evolve it. No set of requirements, designs, or plans is an island. Nor can software engineering, hardware engineering, or human-factors engineering be considered as islands. They all need to be engineered concurrently. And this concurrency needs to be incrementally synchronized and stabilized to avoid divergence, chaos, or analysis paralysis. Figure 5 provides one vision for how to do this, based on extensions of the anchor point milestones[9] provided in the Incremental Commitment Model.[10]

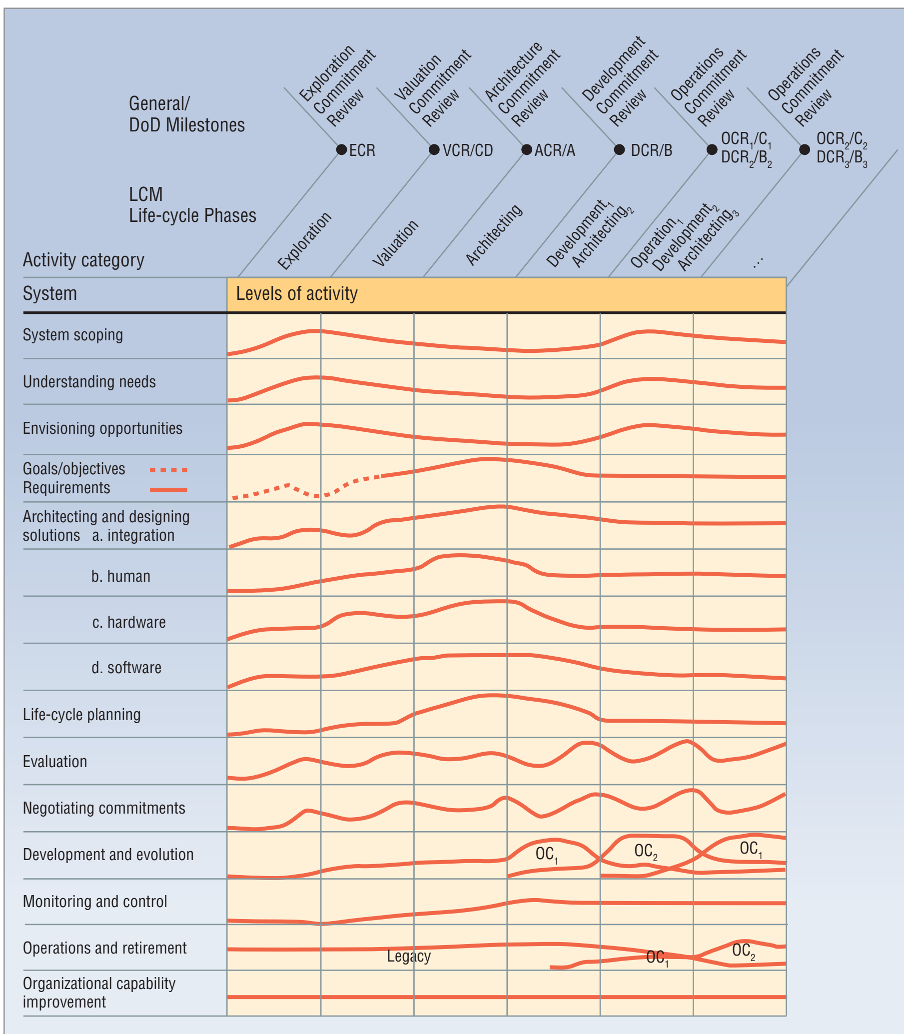


Figure 5. Representative levels of effort by phase in the Incremental Commitment Model.

The recent Software Engineering Institute report on *Ultra-Large-Scale Systems*[11] makes a good case that such SISOS are best considered as ecosystems, which contain internal ecologies among their component systems, and which participate in external ecologies within which they need to stay viable. The report includes numerous good recommendations for improving SISOS acquisition, requirements engineering, design, multidimensional and dynamic quality assurance, and evolution.

A further important point in defining this kind of ecosystem is that considerably more effort is needed up front to avoid getting its concurrently engineered requirements, designs, and plans pointed toward an ecological crisis. Increasing interdependence and the rapid pace of change mean thinking about not just first-order fixes but also their second-order and

maybe third-order effects. The best book I've read recently is *The Logic of Failure*.[12] It shows through a wealth of examples and ecology simulation games that people with their built-in, fight-or-flight survival skills and tendency to apply quick first-order fixes have a horrible track record in managing ecologies, but that they can get better at managing ecologies with practice and with better tools for understanding side effects.

The need for more up-front investment is corroborated by the analysis summarized in Figure 6 from "Spiral Acquisition of Software-Intensive Systems of Systems."[13] It shows that the "sweet spot" for "how much up-front architecting and risk resolution effort is enough" increases as the size of the SISOS increases. This effort is not just in writing specifications and plans but, more importantly, in exercising models, simulations, prototypes, and partial SISOS implementations before committing to go forward into SISOS development.

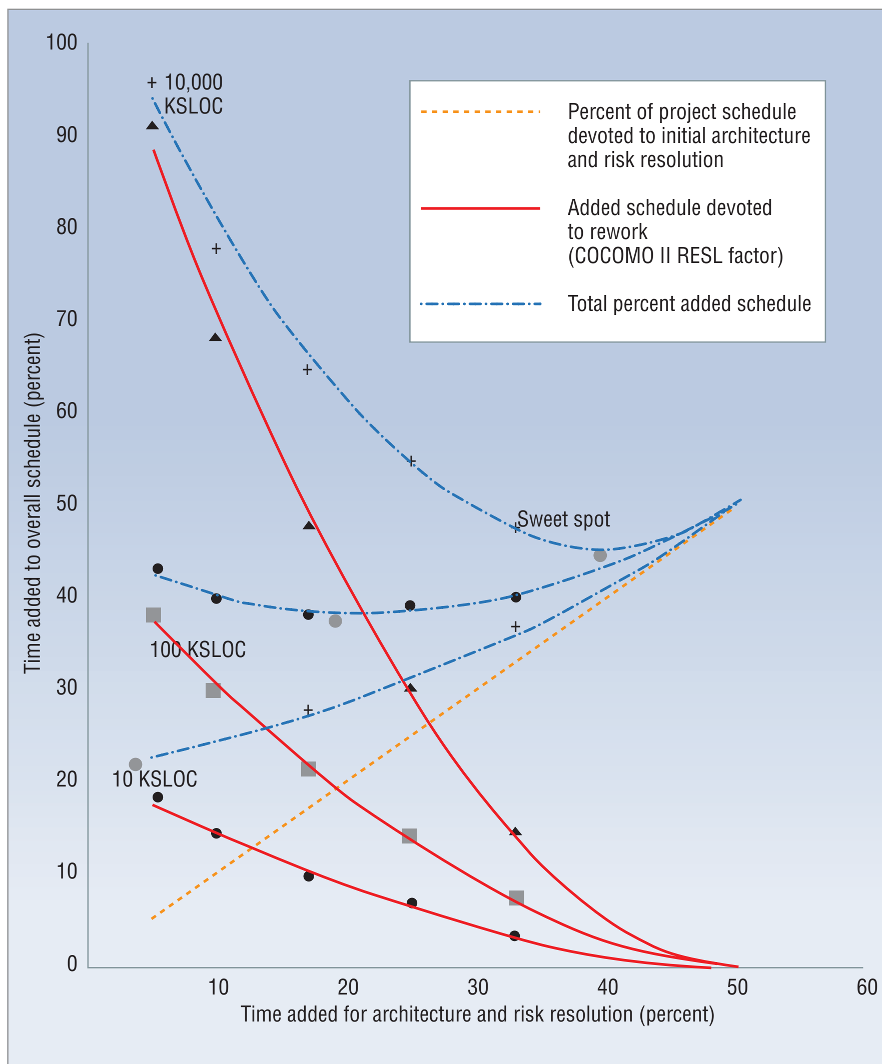


Figure 6. Large SISOS need more time for architecture and risk resolution.

In the 21st century and beyond, software will provide the sensing, communications, and decision support capabilities that enable people to become aware, understand, and collaborate in addressing the problems and opportunities they will have from local and personal levels to global levels. At each level, software capabilities will strongly determine how well people will be able to understand each other and come together to find ways to make their local and global situations more mutually satisfactory and sustainable into the future.

If this software and the collaboration it supports are done well, the world could have a Golden Age going well beyond the dreams of the Athenian and Renaissance philosophers. If it is done poorly, it will exacerbate tensions and mistrust, obfuscate mutual understanding, and cut off many of the options for joint gain that enable people to negotiate mutually satisfactory and sustainable agreements.

This is not to say that software is a silver bullet for resolving social problems. Some of the limits to software perfectability are imposed by limits to social perfectability. An example is provided by Conway's law, which states that the structure of a software product reflects the structure of its sponsoring and development organizations. The converse of Conway's law then states that we will be able to create perfect software as soon as we learn how to create perfect organizations, which seems unlikely to happen soon. A recent National Research Council report provides some approaches for better including human and social considerations into software-intensive systems development.[14]

But it is to say that making incremental improvements in developing better software will have significant leverage in incrementally making the world a better place. And it is to say that being a software engineer in the 21st century will be one of the most challenging and rewarding careers available to people. The challenges we have discussed here of simultaneously dealing with rapid change, uncertainty and emergence, dependability, diversity, and interdependence will often be formidable, but they will also be opportunities to make

significant contributions that will make a difference for the better.

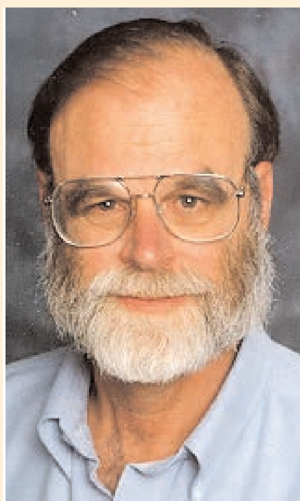
If you have read this far, I hope this means that you are already pursuing or are seriously considering a software engineering career or including software engineering in your portfolio of skills. The world is going to need all the capable software engineers or hardware and human factors engineers that understand software that it can find. And I hope that you will find my observations helpful in addressing the challenges and realizing the benefits you set out to achieve.

References

1. T. Friedman, *The World Is Flat*, Farrar, Straus, and Giroux, 2005.
2. D.L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Trans. Software Eng.*, Mar. 1979, pp. 128-138.
3. B. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.
4. S. McConnell, *Software Project Survival Guide*, Microsoft Press, 1997.
5. B.W. Boehm, D.N. Port, and M. Al-Said, "Avoiding the Software-Model-Clash Spiderweb," *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*, R.W. Selby, ed., IEEE CS Press-John Wiley & Sons, 2007, pp. 743-747.
6. M. Al-Said, "Detecting Model Clashes During Software Systems Development," PhD dissertation, Univ. Southern Calif., 2003.
7. G. Hofstede, *Culture and Organizations*, McGraw-Hill, 1997.
8. E.T. Hall, *Beyond Culture*, Anchor Books/Doubleday, 1976
9. B.W. Boehm, "Some Future Trends and Implications for Systems and Software Engineering Processes," *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*, R.W. Selby, ed., IEEE CS Press-John Wiley & Sons, 2007, pp. 545-571.
10. B.W. Boehm and J. Lane, "Using the Incremental Commitment Model to Integrate System Acquisition, Systems Engineering, and Software Engineering," *CrossTalk*, Oct. 2007, pp. 4-9.
11. Software Engineering Institute, *Ultra-Large-Scale Systems*, CMU/SEI, 2006.
12. D. Dorner, *The Logic of Failure: Recognizing and Avoiding Error in Complex Situations*, Perseus Books Group, 1997.
13. B.W. Boehm et al., "Spiral Acquisition of Software-Intensive Systems of Systems," *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*, R.W. Selby, ed., IEEE CS Press-John Wiley & Sons, 2007, pp. 615-626.
14. R. Pew and A. Mavor, *Human-System Integration in the System Development Process: A New Look*, National Academies Press, 2007.

***Barry Boehm** is the director of the University of Southern California Center for Systems and Software Engineering. Contact him at boehm@csse.usc.edu.*

Tribute to Honor Jim Gray



The IEEE Computer Society, ACM, and UC Berkeley will join the family and colleagues of Jim Gray in hosting a tribute to the legendary computer science pioneer, missing at sea since 28 Jan. 2007.

31 May 2008
UC Berkeley

- General Session: 9:00am
  Zellerbach Hall
- Technical Session: 11:00am
  Wheeler Hall

Registration is required for technical sessions

http://www.eecs.berkeley.edu/ipro/jimgraytribute