Is the Software Engineering State of the Practice Getting Closer to the State of the Art?

Donald J. Reifer, *Reifer Consultants*

How do the state of the art and the state of the practice of software engineering differ? Having surveyed them using current frameworks, the author identifies missing research areas and good practices and identifies eight critical success factors that can help managers close the gap.

For years, my friends have complained about the difficulty of using the state of the art in software engineering technology. Because this technology isn't tried and true, many argue that using new processes, methods, and tools in a production environment is too risky. Recently, however, management seems responsive to taking more risk. To get products to market more quickly than the competition, they are willing to try new approaches such as Extreme Programming,[1] COTS-based (commercial off-the-shelf) construction,[2] and agile methods.[3] They are also trying to put new lifecycle paradigms to work, such as Microsoft's synch-and-stabilize[4] and Rational Unified Process,[5] because they make progress more visible during development. Although these approaches might decrease development time, many in the software community ask, "Do they make it easier to take advantage of the state of the art to build quality software products more quickly and inexpensively?" While some would argue they do, others would ask for hard data to substantiate these claims. Even under the best conditions, such evidence is just emerging. Many times, the business case justifications don't justify the required investment to put new technology into operational use throughout the organization. It is no wonder that many old-timers such as myself remain skeptical. To make the leap, we must be given a compelling reason.

To answer the question just posed, I captured what I believe is the current state of the art and the state of the practice in software engineering. I define *state of the art* in terms of "best theory," which is often not yet implemented. In contrast, the *state of the practice* refers to the current practices software engineers use on the job to get their work done.[6] These practices represent what is being used,

0740-7459/03/$17.00 © 2003 IEEE

as opposed to what could or should be used. Then using these taxonomies, I recommend ways for you to close the existing gaps between theory and practice.

State of the art

The first taxonomy establishes my basis for assessing the current state of the art in software engineering (see Table 1). I based this taxonomy on roadmaps established by leading members of the research community to identify research thrusts and issues they need to tackle as they enter the new millennium.[7] In this table, I condensed the roadmap information along with theoretical areas that academia and industry are pursuing. For the most part, these current research directions were emerging trends three years ago when the roadmaps were developed. All in all, I believe the roadmap authors did a fine job of ferreting out what most in academia and industry believe was the state of the art of software engineering technology in the year 2000.

Some of the research thrusts are related to new approaches that have come into vogue in the last few years. For example, research into adaptive architecture and self-monitoring is clearly needed to address issues associated with COTS guideline developments.

State of the practice

I used the Certified Software Development Professional examination topics to describe the framework I selected for the state of software engineering practice.[8] As Table 2 shows, this examination defines the skills that practicing software engineers must have to be considered professionals by certification proponents. These are obvious skills, and the life cycle's activity-oriented coverage offers few surprises. Surprisingly, as noted in the Missing Current Practices column, the push toward quick-to-market approaches such as agile methods[3] was not considered part of the exam. Also missing was the current business emphasis that many in industry[9] were advancing and the push toward the use of COTS and component-based methods and product line management concepts. I included these missing current practices as examples of practitioner trends. I hope that Table 2 will stimulate debate over what the CSDP exam does and doesn't include.

Additionally, the CSDP exam should also emphasize nonwaterfall development models, quick-to-market practices, and collaboration approaches to software development that are being used by distributed workforces across geographical distances. By adding these practices, industry might be able to quicken closing of the gap between the software engineering state of the art and the state of the practice.

Of course, practice can and does lead to research as practitioners try to put technology to work. For example, research issues abound with the use of COTS packages. Researchers are trying to develop methods to wrap and condition COTS software so that it can be easily integrated into brown-field systems (those whose development is constrained by mixes of new, legacy, and COTS software, in contrast to green-field systems, whose development tends to be new and unconstrained). They must devise new models for estimating and metrics for assessing quality, because old ones do not work when COTS software is mixed with legacy and new code.[10] In such systems, both researchers and practitioners must reexamine underlying premises as they try to address new phenomena (such as the COTS renewal and refresh cycle) and as they try to make existing methods, tools, and architectures work under new operating conditions.

How big a gap?

The missing research areas and current practices provide insight into how closely the state of the practice trails the state of the art. While researchers seem to focus on developing new paradigms, languages, algorithms, methods, and tools, practitioners still seem to have difficulty putting these innovations to work. In contrast, researchers seem to have equal difficulty scaling their work to address problems practitioners face as they try to deliver quality solutions per demanding schedules. Although these problems exist for many reasons, the major contributors are the lack of an underlying science-based underpinning for software engineering along with the small number of measurement-based case studies on which to base decisions about what to use and when. In addition, overemphasizing the move to software process improvement during the past decade has led to a lack of focus on the product-based enhancements (improved performance, quality, interoperability, and so on) that projects need to thrive in the new environments (platforms, organizations, and so on).

Practice can and does lead to research as practitioners try to put technology to work.

Table 1

State-of-the-art taxonomy

Category	Title	Research thrust areas	Missing research areas
1	Software process	Formal modeling methods Process simulations	Merger between software and systems
2	Requirements engineering	Elicitation approaches Modeling approaches Families of products and COTS selection guides	Requirements viewed as a learning experience, not a specification
3	Reverse engineering	Program dicing, slicing, and understanding approaches Database reverse engineering techniques Reasoning tools	More emphasis on achieving desired levels of machine performance
4	Testing	Testing of component-based systems Use of testing artifacts to assist in development	Test-as-you-go methods that make test progress visible
5	Software maintenance and evolution	More effective maintenance tools and methods More formalism Service-based model of software	Short-life systems such as those on the Internet
6	Software architecture	Support for dynamic coalitions Adaptive architectures Self-monitoring and -healing systems	More emphasis on reconfigurable architectures
7	Object-oriented modeling	Aspect-oriented programming Domain-specific modeling	Object-oriented metrics
8	Software engineering for middleware	Distributed architectures Network-of-networks middleware concepts	High-performance distributed systems
9	Software analysis	Checking code conformance to design approaches Abstract design models	More emphasis on behavior models
10	Formal specification	Constructive methods for specification Nonfunctional needs analysis methods	Figuring out the domain of applicability
11	Mathematical foundations of software engineering	An underlying math basis for programming Better understanding through better tools	More emphasis on statistical methods
12	Software reliability and dependability	User-centered measures of reliability Better understanding of issue of variation	Reliability prediction done a priori instead of a posteriori
13	Software engineering for performance	More formalism via UML Performance models integral to methods	Impact of nontraditional hardware architectures
14	Software engineering for real time	Temporal architecture description techniques Systematic validation techniques Fault-tolerant methods	New scheduling algorithms for nontraditional hardware architectures
15	Software engineering for safety	Runtime monitoring to detect faults in real time Methods for safety analysis of COTS and product lines	Self-healing systems
16	Software engineering for security	Adaptive architectures based on security characteristics Better protection models, methods, and tools	Better protection models, methods, and dynamically reconfigurable systems (based on threat detection)
17	Software engineering for mobility	New and novel architectures, protocols, algorithms, and middleware	Dynamically configurable mobile systems
18	Software engineering tools and environments	New methods, formalisms, and tools Adaptation of XML, JavaBeans, and message brokering for tool integration	Self-configurable toolsets (based on topology of the environment in which it operates)
19	Software configuration management	Interoperable tools used in collaborative environments	Daily-build support systems and make utilities
20	Databases in software engineering	Intelligent querying Innovative data modeling tools Interoperable models	Fusion of methods
21	Software engineering development	Exploitation of Web engineering, languages, and tools for software engineering	Use of the Web for collaborative development
22	Software economics	Reasoning models and tools that deal with uncertainty Multiattribute decision models	Application of real options theory in software engineering
23	Empirical studies of software engineering	Empirical underpinning for software engineering decision making	Underutilized experimentation methods
24	Software metrics	Innovative metrics that support new process paradigms and methods	Practical measures of success
25	Software engineering education	Instilling an engineering attitude in education programs Keeping education current in face of rapid change Distance education	More linkage with industry needs

Table 2

State-of-the-practice taxonomy

Area	Topic	Practice thrust areas	Missing current practices
1	Professionalism and engineering economics	Engineering economics Ethics Legal issues and practice Standards	Business case development[13] Metrics and measurement[14]
2	Software requirements	Elicitation Specification Analysis Validation Management	Agile methods Product lines and COTS Security and network defenses[15] System of systems
3	Software design	Architectures Design concepts Design notations Design strategies Human factors System safety	Agile methods Design patterns Interoperability Method integration Product lines and COTS Refactoring[16]
4	Software construction and implementation	Data design Coding Integration Deployment Tuning	COTS Game-based programming Refactoring Web-based development Extreme Programming
5	Software testing	Test strategies Test levels Test design Test execution Test management	Agile methods COTS Performance optimization System level testing Test metrics
6	Software maintenance	Maintainability Maintenance process and activities	Self-healing systems design
7	Software configuration management	Configuration and release control activities	Support for quick-to-market methods
8	Software engineering management	Acquisition management Measurement Organizational management Project management Risk management	People management Virtual project management[17]
9	Software engineering process	Lifecycle models Process definition, measurement, and management	Personal Software Process[18] Rational Unified Process[5] Synch and stabilize paradigm[4] Team Software Process[19] Extreme Programming and agile methods
10	Software engineering tools and methods	Tools and methods to support development, maintenance and management	Collaborative environment Matching methods to problem domains
11	Software quality	Software quality assurance Independent verification and validation	Engineering quality in versus inspections Quality in product and process

This process focus seems to have given rise to agile methods, whose adoption some believe has been stimulated in part in rebellion to the push to achieve higher levels in Software Capability Maturity Model assessments.[11] Although arguably compatible with the CMM,[12] agile methods seem a throwback to the problems many of us faced in the past as the community was pressured to rush to code.

The missing research areas listed in Table 2 illustrate the difficulties practitioners and researchers have in communicating what they need from each other. To resolve these problems, researchers must place more emphasis on the problems industry identifies. Likewise, practitioners must frame their problems so that researchers can address the underlying scientific issues in addition to scalability and adaptability.

Managers who want to take better advantage of the state of the art in their practice and reduce the time it takes to move a technology into practice should do so by selecting their targets carefully.

Don't misinterpret my previous remarks: I don't believe that the state of the art is always superior to the state of the practice. For example, independent of how much benefit a new technology can potentially provide, sometimes it represents too great a risk for a production job. There are many reasons for this situation:

- The technology might be too immature or volatile to put to work in the project's time frame.
- The team might not have time or be ready to adopt the technology.
- The technology might not scale or apply to the problem domain.

However, in many cases, we would like to close the gap and speed technology's introduction.

Bridging the gap

Bridging the gap between the state of the art and state of the practice has proven elusive in the past. The classic example is the over-a-decade time frame in getting the practitioner community to adopt Unix when Bell Laboratories put it into practice. However, timelines don't need to be quite so extended. There have been examples where adoption has occurred more quickly. For example, object-oriented techniques took less than five years to put into practice once industrial-grade methods and tools were brought to market. As another example of bridging the gap, the Software Engineering Institute reported that the average time to reach a CMM Level 2 rating shrunk eight percent from 25 to 23 months in just one year. (Statistics from 1992 to present are available elsewhere.[20,21]) This reduction occurred because the SEI paid considerable attention to providing the supporting infrastructure (training, organization, transition support, and so on) needed when it brought its improvement framework to market.

Managers who want to take better advantage of the state of the art in their practice and reduce the time it takes to move a technology into practice should do so by selecting their targets carefully. They should also take advantage of windows of opportunity and plan accordingly. For example, to cross the chasm more naturally between early-adopter and early-majority organizations,[22] they might propose using the self-modifying architecture they are experimenting with as part of their product line architecture at the start of a pilot project. If they do so, they will reap the benefits. If not, it might take them years to transition the technology into widespread use instead of months. Based on others' experiences, the following eight critical success factors provide managers with action guidelines aimed at better using the state of the art in their practice.

- The technology is considered mature (that is, its use had been proven feasible for a range of applicable domains on a number of pilot and pathfinder projects).
- The body of knowledge (that is, the lessons learned by early adopters and associated practices) related to the technology has been codified and is available in actionable form.
- Hard data associated with using the technology (performance benchmarks, productivity metrics, cost-benefit guidelines, defect rates, and so on) and an initial science-based body of knowledge have been published.
- The rules associated with using the technology are documented, and guidelines for use were published.
- Industrial-strength tools for either use with or for automating the technology have been developed and are readily available on the open marketplace.
- Training and support for introducing the technology have been developed and are available for use.
- People other than the technology's developers are promoting its use for business as well as for good technical reasons.
- The organization planning to use the technology has prepared itself both technically and socially for the changes needed to take full advantage of it (most of the barriers to change are psychological, political, and social, not technical).

Yes, the state of the practice seems to be somewhat behind the state of the art in software engineering. That's expected and might not be a problem in your situation. However, the ability to more quickly close the gap represents a relatively new and positive phenomenon. We as a community must continue focusing on further reducing

the gap as we use technology to generate better products more quickly and cheaply. This will enable us to derive economic benefits from technology that will stimulate development of fixes for the holes that we identified in Tables 1 and 2. Also, we must stimulate researchers and practitioners to work more closely together. Better communication would go a long way to resolving the communities' difficulties understanding each other, in turn increasing focus on the issues that count. Finally, researchers must recognize that practitioners live in an imperfect world and thus must address the brown-field issues that constrain how practitioners get their jobs done. Then, practitioners will perceive researchers' solutions as helpful. This in turn will lead to quickened technology adoption.

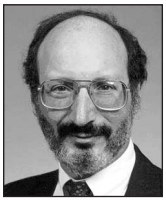
Acknowledgments

I thank Bob Glass for stimulating me to write this article and the *IEEE Software* editorial staff for their many helpful suggestions.

References

1. K. Beck, *Extreme Programming Explained*, Addison-Wesley, 2000.
2. P. Clements and L. Northrop, *Software Product Lines*, Addison-Wesley, 2001.
3. A. Cockburn, *Agile Software Development*, Addison-Wesley, 2002.
4. M.A. Cusumano and R.W. Selby, *Microsoft Secrets*, Touchstone, 1995.
5. P.N. Robillard et al., *Software Engineering Process with the UPEDU (Unified Process for Education)*, Addison-Wesley, 2003.
6. *IEEE std. 610.12-1990, Standard Glossary of Software Engineering Terminology*, IEEE CS Press, 1990.
7. A. Finkelstein and J. Kramer, "Software Engineering: A Roadmap," *The Future of Software Engineering*, A. Finkelstein, ed., ACM Press, 2000, pp. 5–22; www.cs.ucl.ac.uk/staff/A.Finkelstein/fose/finalfinkelstein.pdf.
8. R.H. Thayer and M.J. Christensen, "Appendix C: CSDP Examination Specifications," *Software Engineering, Vol. 2: The Supporting Processes*, 2nd ed., IEEE CS Press, 2002, pp. 427–429.
9. C. Wallin, F. Ekdahl, and S. Larsson, "Integrating Business and Software Development Models," *IEEE Software*, vol. 19, no. 6., Nov/Dec. 2002, pp. 28–33.
10. D.J. Reifer et al., "Scaling Agile Methods," *IEEE Software*, vol. 20, no. 4., July/Aug. 2003, pp. 12–14.
11. M.C. Paulk et al., *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, 1995.
12. D.J. Reifer, "Extreme Programming and the CMM," *IEEE Software*, vol. 20, no. 3., May/June 2003, pp. 14–15.
13. D.J. Reifer, *Making the Software Business Case: Improvement by the Numbers*, Addison-Wesley, 2002.
14. J. McGarry et al., *Practical Software Measurement*, Addison-Wesley, 2002.
15. F.B. Schneider, *Trust in Cyberspace*, National Academy Press, 1999.
16. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
17. P.E. McMahon, *Virtual Project Management*, St. Lucie Press, 2001.
18. W.S. Humphrey, *Introduction to the Personal Software Process*, Addison-Wesley, 1997.
19. W.S. Humphrey, *Winning with Software*, Addison-Wesley, 2002.
20. Software Engineering Measurement and Analysis (SEMA) Team, *Process Maturity Profile of the Software Community*, Software Eng. Inst., briefing, 2001.
21. Software Engineering Measurement and Analysis (SEMA) Team, *Process Maturity Profile of the Software Community*, Software Eng. Inst., 2002.
22. G.A. Moore, *Crossing the Chasm*, HarperBusiness, 1991.

About the Author



Donald J. Reifer is president of Reifer Consultants, a member of *IEEE Software*'s editorial board, and the magazine's Manager column editor. He has an MSOR from the University of Southern California. Contact him at d.reifer@ieee.org.