Can Practitioners Neglect Theory and Theoreticians Neglect Practice?

Manfred Broy, *Technical University of Munich, Germany*

Theory helps engineers in other disciplines create and understand methods, evaluate results, and optimize processes. Does it have a role in software engineering as well?

Software engineering is by definition part of a practical discipline,[1] but scientific insight and knowledge are not its primary measures of success—rather, its practitioners are chiefly focused on building and further evolving software systems in a timely manner at a reasonable cost and quality. The measure of success of the discipline is the extent to which it provides concepts, methods, techniques, and processes that allow developers to master the task of software evolution. In addition to software development and maintenance, this task presents diverse challenges, including requirements analysis, specification architecture, implementation, integration, and validation.

Engineering disciplines must be based on scientific practices and theory to justify their approaches and to give scientific evidence for why and where their methods work properly. Fortunately, many different kinds of scientific methods—including those related to theoretical concept analysis, empirical and experimental research, as well as related concepts from disciplines such as psychology and economics—can help with this task. Researchers can create, adapt, and further develop these scientific methods to more properly support the discipline of software engineering.

CAN THEORY IMPROVE SOFTWARE ENGINEERING?

A recent article by Ivar Jacobson and Ian Spence on theory in software engineering included the following thought-provoking statement: "Our greatest challenge: understanding how to build great software."[2] But what exactly is "great" software? Do methods exist to distinguish between great and not-so-great software? Even simple questions like these use a theory of quality as the basis for software construction and evaluation.

Looking more closely at methodology, if the task of software engineers is to build and further evolve software systems, and if the methodological and technical maturity of software engineering is insufficient, who is responsible for improving development methods? Research and improvement initiatives from both industry and academia have sought better methodology in software engineering, but they have encountered several challenges:[3]

- establishing precise and adequate terminology (for instance, better terms than "great" for addressing software quality);
- developing techniques for building software of adequate quality;[4] and
- identifying methods for measuring, judging, and evaluating both the software itself and the effectiveness of the processes, concepts, and tools used to develop it.

Suggesting new engineering methods without their careful justification is of limited help. Unfortunately, researchers often advocate methods without offering

0018-9162/11/$26.00 © 2011 IEEE Published by the IEEE Computer Society

substantial analysis or evidence as to how well, under which constraints, and why they work. For example, the so-called *agile manifesto* states several principles that seem to have promise, but it does not provide scientific evidence of the extent to which agile methods actually help with building software of adequate quality. Software engineering needs theory to evaluate the efficiency and effectiveness of development techniques, which would ultimately result in justified and empirically verified methodological knowledge.

The more complex an engineering discipline is, the more important its theoretical foundation—and software can become very complex.

CRAFT, SCIENCE, OR ENGINEERING DISCIPLINE?

Identifying the differences between craft, scientific theory, and engineering helps in understanding the role of theory in engineering:

- *Craft* applies traditional techniques to produce goods and provide services.
- *Science* aims to gather, verify, and document knowledge and insight through research.
- *Engineering* applies both knowledge and scientifically analyzed and justified methods to develop and produce technical products.

Applying scientific theories to create, analyze, and justify methods and techniques is essential to move something from craft to engineering. For craft, it is not important to understand why methods work or how they could be proven and optimized; it is sufficient if they produce satisfactory results. In contrast, for engineering, the questions of how to construct new methods systematically, why different techniques work, and how scientific methods can justify, relate, prove, and optimize them are key.

Craft and engineering have much in common because they have similar overall goals: creating constructive solutions to practical problems in terms of products and services. Consequently, the separation of craft from engineering might not seem clear—for example, some might say that software development as practiced today is to a large extent a craft. However, by definition, craft applies traditional techniques. Because software development is still a young field, it does not have a large number of such techniques. In some respects, the discipline works in an ad hoc manner, applying "best practices" that often cannot be called well proven. The evolution of traditional techniques requires time, and the software field is still experiencing fast-moving technical progress.

Other subtle differences emerge as well. In contrast to traditional craftsmanship, which typically involves smaller groups, software projects and their budgets can be massive. In addition, development teams for huge, complex software systems must be managed using organizational team structures and management processes. Further challenges arise with the development of large families of software systems—to evolve product lines and families strategically requires something more rigorous than typically found in craft, which brings us full circle to theory.

To successfully accomplish complex tasks, theory proves indispensable. The more complex an engineering discipline is, the more important its theoretical foundation—and software can become very complex.

WHY THEORY?

Successful software engineering requires insights into many aspects of software and its evolution, such as

- methodology, including requirements engineering and specification, architecture and design, code quality, integration and verification, deployment and migration, and modification and improvement of software systems;
- organizational competency and management of software evolution;
- domain-specific knowledge;
- technical information about hardware;
- human-machine interaction; and
- the economic aspects of software, marketing, and entrepreneurship.

Software engineering also deals with numerous abstract concepts that are often hard to understand because of the imprecise terms used to describe them, such as *modularity*, *compatibility*, and *tracing*. These notions often appear in publications or in practice without establishing a consistently clear understanding of what they mean or where and why they are important. This is why theory matters—it helps determine and evaluate the concepts that provide the basis for identifying terminology and developing engineering methods.

Theory and methods

An engineering discipline without a theory cannot work. But software engineering also requires methods to help solve the problems that arise when developing and evolving a software system. For methods to be effective, researchers must evaluate, justify, and classify them to determine how they work best and to identify their limitations.

Like other disciplines, software engineering has many facets, such as people, management, and teams; economy and cost; methodology and development processes and tools; human-machine interaction factors; and actual technology—hardware, operating systems, and programming languages.

Researchers can use theory to interpret and explain obstacles in engineering practice and provide constructive foundations for engineering methods and tools. How much theory and methods can contribute to engineering is evident in the field of compiler construction. Applying theory systemically in this field using context-free grammars and compiler generators has turned it into one of the first mature areas of software construction.

Logical theories also play a prominent role in some aspects of software engineering because developers can use mathematical logic to deal with the formality. Research in formal techniques is indispensable and can lead to insights, perhaps finally resulting in solid engineering methods. However, there is a considerable gap between conducting research on formalization and formal theories, and developing tractable methods. Of course, such methods will also require a careful analysis of scalability, tractability, applicability, and efficiency as well as effectiveness.

In spite of the reported benefits of applying formal methods to some areas of software engineering, some counterarguments still exist as to their practicality.[5] Formal methods

- do not scale, per se;
- are too difficult for engineers who are not properly trained;
- if not carefully designed, do not capture essential aspects; and
- are not cost-effective if they are not well integrated into development processes.

Perhaps part of the problem is that when the term *formal theory* is mentioned in software engineering circles, *formal methods* come to mind. But reducing formal theory to formal methods is both too narrow and misleading. Formalization is a scientific method that can help analyze, validate, and create engineering methods, but formal methods should not be regarded as a silver bullet for software evolution. As David Parnas says, "Our role model should be engineers, not philosophers or logicians."[6]

Theory and education

Educating software engineers without giving them theoretical foundations is unthinkable. Even if some theories are not immediately applicable to practical engineering, they give a framework of understanding. One example is assertion logic. Even if formal proofs of programs through assertions are not used in practice, understanding the ideas of Hoare triples, loop invariants, and termination proofs is a valuable conceptual exercise. The same applies to algebraic data types, relational data models, and modular models of software architectures.

It is essential, however, to teach theory the right way. Students need to get a deep understanding of the basic engineering concepts and their theoretical justification to relate them to practical tasks. Applied topics such as operating systems, protocols, and databases should also be taught and explained in terms of adequate theories.

Logical theories also play a prominent role in some aspects of software engineering because developers can use mathematical logic to deal with the formality.

Software evolution is an engineering discipline based on informatics—the scientific discipline of information and information processing. Several related aspects of theory are important for software engineering.

Classical areas such as formal languages, computability, and complexity theory are obviously important for software engineers. How can researchers develop ambitious software systems without understanding the limits of computability and decidability? How can they deal with performance without understanding computational complexity? How can they design software engineering tools for modeling or programming without understanding formal languages?

Techniques for describing the semantics of programming languages, program specification, and program verification are parts of a theory of programming in the small. Certainly, programming in the small is a basis of software engineering, but it is not software engineering per se. Indeed, software engineering is concerned with programming and developing in the large. Some theories directly address the foundations of programming in the large and lay the foundations for software engineering.[7] Examples include ways of modeling and mastering key aspects of software systems such as requirements, architecture, interfaces, composition, and quality.

In addition to the foundational theories of informatics, software engineering also must be based on more general subfields of informatics such as the theory of data types, databases, hardware structures, operating systems, protocols, and so on.

Although using formal techniques might not always be directly practical in software engineering, they help in

understanding the basic software engineering questions and concepts. How could we understand notions such as correctness, specification, or verification without appropriate theory? Even if in practice such techniques often are not applied in the rigorous way formal methods suggest, at least understanding them is useful.

Another interesting foundation of software engineering is mathematical logic, a field that has developed completely independently of software engineering. There is a close relationship between computing and mathematical logic, and many software engineering issues are closely related to logic. The reason is obvious: software is by nature a formal artifact. Because it is formal, computing machinery can execute software with interpretations independent of human beings. Hence, software exhibits behaviors and properties that are subject to formal analysis, even if a programming concept is not formalized or if software is written in ad hoc languages and executed on machinery by ad hoc interpreters.

Software engineers apply both scientific—mathematical, economic, and social—and practical knowledge when they design and build software systems.

That said, software is radically different from other technical artifacts. In contrast to conventional engineering disciplines, software has no material nature—it is pure logic. It describes dynamic behavior by static means using rigorous formalisms such as programming languages.

However, in practice, software's entirely formal nature is obscured by the technical consideration of execution environments, which have many different mechanisms and contribute to the generation of highly complex behavior. Software is tightly connected with its environment, and the connected electronic, mechanical, and organizational processes usually are not formal, thereby requiring a more comprehensive view of software systems as formal entities, separate from their operational context. This split requires two complementary views: a formal one that captures the interaction between interface and software system contexts, and a pragmatic view directed at the impact of the software on its context.

THEORY IN SOFTWARE ENGINEERING RESEARCH

Software engineers apply both scientific—mathematical, economic, and social—and practical knowledge when they design and build software systems. Theory can help engineers avoid repeating the same mistakes when building new systems. It is depressing to see the extent to which well-established theoretical principles are ignored in pragmatic work, such as in developing UML[8] and Java. One of the frustrations of the field is that we have not managed to bring together theoretical knowledge and practical work in a way that offers a mutual benefit.

Insufficient knowledge about practice in academia and limited competency in practice are key factors in the failure to bridge this gap. Most companies have narrow views about the field of software engineering—even about how to use software processes and the basic rules and principles of the field. Accordingly, it is difficult for them to see the value in theory-based methods. Moreover, due to the lengthy amount of time it takes to introduce advanced incremental theory-based techniques, even if they contribute to cost savings and quality in the long run, it is difficult for nonspecialists to recognize their methodological potential. This is, of course, reinforced in cases in which we do not have enough evidence that approaches based on theory actually contribute practically. In fact, introducing advanced engineering methods brings risks even for management with expertise.

Nevertheless, software engineering theories have made enormous progress in the past 40 years, in particular those related to specification and verification, architecture and design, testing, software project management, and software economics. However, we are still far from comprehensive and established theoretical foundations.

Much more work on theory is needed to master the enormous future challenges related to software evolution, when we can easily foresee having to construct large software systems, software families, or cyberphysical systems. Theoretical foundations will ultimately prove indispensable for dealing with the inevitable behavioral and nonfunctional properties related to safety, security, reliability, usability, and maintainability, especially for highly distributed, iterative networks of software systems.

So why are the results of theoretical software engineering research not applied more often in practice? In part, the answer to this question has to do with the attitudes of researchers working in this field. There are several different approaches to this research.

Theory based on practical observation and experience

In this approach, researchers try to discover theories that help them understand, interpret, and deal with observations and obstacles they have discovered in practice. A typical example would be the pointer and reference structures found in many programming languages. Developing a theory of pointer structures for reasoning about programs with languages like Pascal (with its typical pointer

structures) or with object-oriented languages like Java (with its object identifiers, which are essentially nothing more than pointer structures) is a challenge. Accordingly, a typical scientific approach would try to work out a theory and then use that theory to help in the classical tasks of program specification, analysis, and verification. Further examples would include formalized techniques that deal with architectures of software systems.

Theory in isolation

Another approach is independent observations of practice to develop helpful scientific theories that hopefully can later contribute to a better understanding of the practical issues of software engineering. In this case, researchers work independently from practical issues and just concentrate on getting the theory into clearer form.

From ad hoc methods to theory

This third category characterizes approaches in which practitioners encounter practical problems and then develop ad hoc methods to deal with them. Later, they might try to provide a theory for them. Examples include UML and architectural description languages. Both show that ad hoc solutions often must change fundamentally after careful analysis because ad hoc method constructions typically fail to get everything right in the first cut. This usually becomes evident when working out the theoretical foundations, further indicating that giving foundations to ad hoc methods exposes their shortcomings and superficial complexity.[8]

Obviously, there are crucial differences between these three approaches. It is very dangerous to develop theory and claim that it is practically useful without being knowledgeable and working in close relationship with practitioners. A more promising approach is to work in iterations between steps in developing theory and any attempts to apply them in practice. Doing useful theory requires sufficient capabilities in performing theoretical work. Not all software engineers with deep practical insights and understanding are knowledgeable enough in scientific work to express their insights in adequate theory.

ADDRESSING PRACTICAL QUESTIONS

In the long run, theory for engineering must address practical questions. We do not need theorists who work out beautiful calculi that are too good for the dirty work of practice but claim to contribute to software engineering and its practical problems. Similarly, working out an algebraic calculus and deciding about certain properties independently of engineering issues is dangerous because issues of aesthetics and scientific quality for algebraic calculi might not necessarily coincide with practical needs and tractability in the real world.

A typical example explains this viewpoint. In the 1980s, based on the observation that some of the deduction rules of linear logic showed similarities to phenomena in concurrent systems, some researchers claimed that linear logic would contribute to concurrency theory. However, in the end, linear logic could not contribute at all either to concurrency theory or to the practice of engineering concurrent and distributed software systems. Another example is the POM sets by Vaughn Pratt.[9] Although they presented an interesting step into understanding the structure of concurrent processes, delving deeply into the details of POM sets and Chu spaces proved to be useless in practice, in the end contributing nothing to resolving engineering challenges.

As long as theoreticians do not take into account issues from practice, their work will not be very helpful from a pragmatic viewpoint and will not find an immediate road into practice.

In fact, much of the very basic work and theory is not properly related to practical issues. An example is the field of process algebras, a branch of theoretical informatics that was an active research field a few years ago. Certainly, there is a rich world of process algebras similar to the mathematical groups as studied in group theory. But most pure branches of theoretical informatics study different kinds of process algebras completely independent of any questions related to software engineering practice.

Of course, it is rewarding to work out concepts and their underlying theories to support certain software engineering tasks, such as specification, design, and verification. But this can be done with a mathematical attitude leading to instances of theory that contribute to the understanding of engineering and give a proof of concept for engineering methods. As long as theoreticians do not take into account issues from practice, their work will not be very helpful from a pragmatic viewpoint and will not find an immediate road into practice. There are several reasons why: software engineering lives in a dirty and imperfect world, and many compromises must be accepted in practical software evolution, not just because of a lack of education of the engineers doing the work but also due to many obstacles encountered in practice, including hardware, tools, and organizational structures. Bringing proper theory into the unsystematic world of practice is a difficult endeavor.

Theories that characterize essential engineering concepts through useful relationships between levels of

abstraction in architectures, for example, those based on Galois connections, could contribute to practice. Galois connections are an interesting and deep concept in algebra, but both theoreticians and engineers can find them difficult to understand. However, proving levels of abstractions that form Galois connections could be a helpful observation because it allows choosing levels of abstraction independently and relating them methodologically, bringing together theoretical and tool support issues.

As a matter of principle, engineering disciplines are based on their own scientific foundations. Electrical engineering and mechanical engineering are based on physics. Software engineering is based on logics and theories of information and computation.

But in addition to a theoretical foundation, we need a joint understanding and agreement that, in the long run, software engineering cannot be established without a solid body of scientific theory. In fact, software engineering requires an understanding of theory, what it can offer, and its limits. This requires a comprehensive understanding of practice, its needs, and its open challenges. A structured presentation of theory and its connection to the engineering of software systems is a must in software engineering education.

Acknowledgments

I thank Benedikt Hauptmann and Holger Pfeifer for reading and commenting on drafts of this article.

References

1. P. Naur and B. Randell, *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee*, NATO, 1968.
2. I. Jacobson and I. Spence, "Why We Need a Theory for Software Engineering," *Dr. Dobbs J.*, 2 Oct. 2009; http://drdobbs.com/architecture-and-design/220300840.
3. M. Broy, "The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems," *Computer*, Oct. 2006, pp. 72-80.
4. I. Jacobson and B. Meyer, "Methods Need Theory," *Dr. Dobbs J.*, 6 Aug. 2009; http://drdobbs.com/architecture-and-design/219100242.
5. R.A. De Millo, R.J. Lipton, and A.J. Perlis, "Social Processes and Proofs of Theorems and Programs," *Comm. ACM*, vol. 22, no. 5, 1979, pp. 271-280.
6. D.L. Parnas, "Really Rethinking Formal Methods," *Computer*, Jan. 2010, pp. 28-34.
7. M. Broy, "Toward a Mathematical Foundation of Software Engineering Methods," *IEEE Trans. Software Eng.*, 2001, pp. 42-57.
8. M. Broy and M.V. Cengarle, "UML Formal Semantics: Lessons Learned," to appear in *Software & Systems Modeling*, 2011.
9. V.R. Pratt, "Chu Spaces and Their Interpretation as Concurrent Objects," *Computer Science Today: Recent Trends and Developments*, LNCS 1000, J. van Leeuwen, ed., Springer, 1995, pp. 392-405.

Manfred Broy *is a professor in the Department of Informatics at the Technical University of Munich, Germany. His research interests focus on the theoretical and practical aspects of software and systems engineering. Broy received a Habilitation degree in informatics from the Technical University of Munich. Contact him at broy@in.tum.de.*