

Sharing What We Know About Software Engineering

Michal Young
University of Oregon
michal@cs.uoregon.edu

Stuart Faulk
University of Oregon
faulk@cs.uoregon.edu

ABSTRACT

Software engineering research has long borrowed and adapted ideas from other disciplines to adapt to the peculiar context of building software. That context is less and less peculiar, as automation and communication transform other fields, and it is time for us to consider how approaches developed in software engineering can be transferred and generalized to other fields. Considering generalization of software engineering to domains outside computer science has implications for both software engineering research and education.

Categories and Subject Descriptors

D.2 [Software Engineering]: General

General Terms

Design

1. INTRODUCTION

It remains commonplace to point out differences between software engineering and “real” engineering (or between computer science and “real” science) as flaws in software engineering research and practice. Others, though, have long noted the peculiar challenges of engineering “pure thought-stuff” [1]. Software engineering research (sometimes led by practice) has adapted and extended engineering methods to these challenges.

Wing [7] has ably summarized the value of *computational thinking* to those outside computer science, owing to the growing role of computation as a tool as fundamental as mathematics. We argue that these intellectual tools, particularly those from software engineering, generalize even further, that in fact many other fields of engineering increasingly face challenges that were confronted first in software engineering. The distinction is subtle but important: While the importance of computational thinking is largely in the growing role of computation across all fields, principles and methods in software engineering are not limited to direct use

of computation. Indirect effects of computation and communication technology make engineering of physical artifacts more and more like engineering of software.

Software engineering research has long taken ideas from other disciplines to adapt to the peculiar context of building software. It is time for us to consider how approaches developed in software engineering can be transferred and generalized to other fields. Considering this question will have effects on software engineering research as well education.

2. WHY SOFTWARE WAS SPECIAL (BUT NO MORE)

The essential distinction between software and other engineered artifacts has always been the absence of fabrication cost.¹ In conventional engineering of physical artifacts, the cost of materials and fabrication has dominated the cost of design and placed a check on the complexity of artifacts that can be designed. When one bottleneck is removed, others appear, and software engineering has therefore faced the essential challenges of complexity and the cost of design to an extent that conventional engineering has not.² Software engineering has focused on issues in managing complexity, from process to modular design to cost-effective verification, because that is the primary leverage point when the costs of materials and fabrication are nil.

Automation and communication technology are reducing the role of materials and fabrication in a variety of physical products. While this is sometimes from removing the physical embodiment of what was always essentially an information good (e.g., replacing maps with mapping services, books with e-books, etc), a very similar effect is obtained when design is disaggregated from fabrication. In computing we are familiar with the example of fabless chip producers, who produce chip designs that can be produced by independent chip foundries. Similar disaggregation is taking hold in fields outside electronics as well, such as the bicycle factories in Taiwan that serve as the “fabs” for bicycles

¹This peculiarity has troubled some to such an extent that they have tried to imagine programming as a kind of fabrication step. The analogy cannot withstand scrutiny. Space and focus does not permit a detailed argument here, but we take as a basic premise that programming is instead a step in the detailed design of a computation, which is the primary artifact produced by software engineers.

²There are other fields of design which likewise have little or no fabrication cost (legislation, for example, and music), but these have not traditionally been considered fields of engineering, nor used (and thus been forced to adapt) engineering methods.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-4503-0427-6/10/11 ...\$10.00.

designed elsewhere. While the physical artifact (e.g., the bicycle) still provides a useful check on complexity, sharing fabrication facilities and technology among competing manufacturers leaves design as the differentiating factor for competition.

3. WHAT WE KNOW

Software engineering is still a young field, but not all of the differences between software engineering and more conventional fields of engineering and science are flaws. Some reflect progress in coping with the challenges of pure design earlier and more intensely than other fields.

3.1 Designing the Design Process

Every field of engineering (and many fields outside engineering, e.g., cartography, advertising, and movie-making) have explicit design processes that are often formalized, studied, and taught to at least some degree. What is less common is treating the process of design itself as a thing to be designed. It is commonplace in software engineering not only to contrast different approaches to software development, but also to reason about what makes one approach more suitable in some situations (say, when requirements are unclear because the application domain is relatively unprecedented) and another approach more suitable in others (say, when the software will control a safety-critical system). It is commonplace not just to adopt and follow a prescribed “best practice”, but also to combine and alter features of different design processes in systematic ways. Leading designers of software systems are often also designers of processes for designing software.

It can be difficult for software engineers to perceive how advanced software engineering is relative to other engineering fields with respect to designing the design process. We envy the discipline and predictability in the design processes for bridges and automobiles, or at least what we imagine those processes to be. But we are fooled, because those design processes seem orderly and predictable only because they are relatively stagnant, and because the cost of the physical artifacts so dominates the costs of inefficient design.³ In fact, as the efficiency of design itself becomes dominant in a wider array of fields, it will be important to share what we know about designing the design process.

3.2 Abstraction and Evolution

Abstraction is an essential step in any field of design, from aircraft to drugs to drugstores. Often the designer creates several abstractions of the same artifact (e.g., one to ensure airfoils have sufficient lift, another to ensure restroom facilities are appropriately distributed in the passenger cabin). What is rare in engineering, but commonplace in software engineering, is explicit study of how to invent and employ new abstractions.

In software engineering, choice of abstractions is tied directly to characteristics of the software design process, and particularly to design artifacts as objects of value, to be reused and modified over time. Since the pioneering work of Parnas [4], we have understood that the choice of an abstraction (especially, but not only, the abstraction presented by

³A reviewer of of this paper noted that automobile design has already diverged from our idealized conception of an orderly design process with electrification, and is experiencing the classical software engineering problems of complexity.

a module interface specification) has direct practical consequences in making some design changes cheaper and faster, and others impractical. To a lesser degree, we understand how the cost and pace of design depends in part on the abstractions we choose to make explicit or formal at different stages of development, and how the pressures affecting those changes may evolve over time.

Imperfect and evolving as our understanding of abstraction may seem, and as difficult as we find imparting the art of abstraction to students, software engineering and computer science are steps ahead of other fields in both understanding and pedagogy of abstraction as a design step. The difficulty in pedagogy, in particular, is largely because students do not encounter invention of abstractions as a design step in other fields. They certainly encounter abstractions in mathematics, in physics, even in social sciences, but in no other field are they likely to encounter explicit instruction in devising and evaluating novel abstractions as a way of decomposing complex problems into independently solvable sub-problems, little more how to devise and evolve abstractions that will continue to be useful as problems evolve, or across related problems.

3.3 Notation Design

Like abstractions, useful notations are an important intellectual tool in many disciplines. What is less common is notations as subject of explicit inquiry. In software engineering, not only conventional programming languages but a variety of other notations, from requirements descriptions to configuration descriptions to test plans have long been studied.

Important understandings developed in the overlap and intertwining of programming languages and software engineering research include the ways in which notational power and flexibility are traded for analyzability, how and to what extent portions of a large notational artifact can be independently analyzed and the results efficiently combined, and how notational design impacts the extent to which useful diagnostics can be extracted from analysis and use (that is, compiler and run-time error messages). There is, to our knowledge, no comparable field of study for notations developed in other disciplines, and little guidance to developers of notations outside computer science.

Software engineering researchers also study aspects of notation design outside the core concerns of mainstream programming language research, including interface languages, diagrammatic notations, and markup languages. Our growing understanding of notation design is closely tied to our understanding of software processes, evolution, and use of abstraction. For example, attempts to separate presentational, content, and structural markup in HTML⁴ are a classic exercise in information hiding and modularity, applied in the context of notation design.

⁴See, e.g., <http://www.w3.org/TR/WCAG10/#content-structure>. One may also find in logs of design discussions the question of voice rendering as an acid test for distinguishing markup that is too presentational: `<emph>` can be considered structural content because one may imagine rendering it in many forms, including voice, but `` (bold) is excessively presentational because it has an interpretation only in printed text. Testing against hypothetical implementation changes is precisely how information hiding principles are applied to determine whether an abstract interface hides the design secrets of a module.

Less studied, but equally important, is the evolution and codification of conventions into notations, including programming languages. Ryder and Soffa's study of the interplay between software engineering and programming languages [5] is a beginning. It seems likely that understandings of other domains and the notations used in those domains similarly co-evolve, and that lessons from the evolution of software engineering concepts and notations could be generalized.

3.4 Meta-Engineering

The unique power of these conceptual structures becomes increasingly evident as they are applied to themselves. While "abstractions of abstractions" or designing "processes to design processes, to design processes" may sound like convoluted self-indulgence, such thinking is foundational to a growing understanding of, and facility with, meta-engineering: the engineering of engineering practices. While all engineering disciplines necessarily examine their own practices, this tends to be a discipline distinct from the engineering practice itself. Constrained by material processes and immutable physics, emergence and adoption of new practice tends to be slow. This is not the case with software engineering.

This self-reflective approach to meta-engineering is exemplified in the development of software product line engineering and, more recently, domain-specific modeling. Product-line engineering is, among other things, a process-development process. Developing a software product-line requires developing a set of assets from which members of a family of software systems can be produced quickly and easily. A critical asset of any product-line is the process for using the other assets (code, generators, etc.) to produce individual members of the software product line. This process (called the "application engineering process") is, itself, a product of an earlier phase of the product-line process (domain engineering). In short, the product-line process embeds a process for designing and developing application engineering processes tailored to the assets and generation methods deployed.

The product-line process further illustrates the power of this paradigm in that it at once applies and is a product of meta-engineering, deploying recursively the concepts of the design-of-design, abstraction, evolution, and language development. Abstracting the common requirements of all the prospective systems in the product-line is the basis for developing common assets. Specifying precisely which member of the product-line should be produced requires developing a domain-specific language (the application modeling language). Deploying the product-line in any real setting requires thinking ahead about how the embedded processes, abstractions, languages, and other assets will evolve over time.

Likewise, the product-line process itself is necessarily the product of meta-engineering. For example, this is evident in the specification of the FAST product-line process [6], which is explicitly described as a product of process development. A process modeling notation and tool are introduced to specify the process. Further, the process specified is itself an abstraction, representing the activities, artifacts, and roles common to a family of product-line processes.

We see the continuing extension of this thinking in emerging approaches to domain-specific modeling, software factories, and supporting tools. These approaches explicitly

address the processes of meta-engineering in providing models, languages, and processes for building coherent sets (e.g., domain-specific) of models, languages and processes. Supporting tools (e.g., Metaedit+ [3]) provide languages and tools for building product-line languages and tools. Such generator-generators provide the capabilities to define new modeling languages and specify their interpretation in software. Output of the tool is a tool that embeds the domain model and language, generating members of the software-family from specifications in the language.

The real-world power of the approach have been amply demonstrated. Processes for process improvements (e.g., the CMMI) have become standardized and there are tools for developing abstract process models and deploying instances (e.g., Eclipse process modeler). Product-line approaches are now deployed in industrial applications from automobiles to medical systems and cell phones. Our own research has applied these concepts to domains as diverse as new processes for developing massively parallel programs [2] and self-adaptive assistive device product-lines for cognitively impaired individuals.

Software engineering's rapidly evolving meta-engineering capabilities have no parallel in more traditional disciplines. Of particular importance are the generalizable methods needed to analyze new domains and apply the principles and practices of meta-engineering to create new processes, methods, and tools. As software becomes increasingly ubiquitous and design supplants manufacturing in more and more fields, the need for such capabilities must grow. The necessary cognitive tools are being developed in software engineering.

This sampling of generalizable, transferable understandings from software engineering research (and practice) is hardly exhaustive. We have not touched upon what is understood about requirements engineering, or management of distributed teams, or validation and verification. In all of these, software engineering research and practice have developed approaches that were initially peculiar to the engineering of software, but which are likely to be increasingly relevant to other fields.

4. IMPLICATIONS FOR RESEARCH AND EDUCATION

If one accepts the argument that understandings developed in software engineering research can and ought to be shared with other fields, then what should we as software engineering researchers do? Mere evangelism is of doubtful value, but there are pragmatic steps we can take in both research and education.

Many of us already participate in collaborative, cross-disciplinary, applications-oriented research. In such research, we surely ask ourselves regularly both what we bring to the discussion with our peers in other disciplines (more, we hope, than just facility in building useful software), and what understandings we can bring back to computer science.

Taking seriously the proposition that software engineering concepts are generalizable adds a bit to both questions. We can ask ourselves, what is the relation of the problems encountered in another domain to common problems in software engineering? How can software engineering concepts be applied in that domain, in ways that extend beyond engineering software for the domain? What fails to generalize, or

requires adaptation, and why? By pursuing these as questions, we avoid off-putting hubris. And to the extent we find even partial answers, we contribute in a more fundamental way both to the field of application and to software engineering research.

Many of us are also educators. Wing has argued persuasively that computational thinking skills are valuable to students in all disciplines, and lists abstraction and separation of concerns among the exemplars of computational thinking [7]. Concerning ourselves explicitly with how software engineering concepts generalize to other fields also prepares us to better communicate those concepts to our students, including those who will ultimately pursue other fields of inquiry.

5. SUMMARY

Software engineering was initially unique among engineering disciplines in that, having little or no costs of materials and fabrication, it was forced to grapple with other problems. Ironically, the success of computation and communication technology is making software engineering less unique. In other fields, the cost of materials and fabrication are either declining relative to the cost of design, or by being disaggregated are becoming less significant in competition. Thus concepts and approaches that were developed first for software engineering are becoming increasingly relevant in other fields. Generalization of software engineering approaches to other fields should be an explicit goal of future software engineering research, and can enrich not only those other fields but also our understanding of software engineering.

6. REFERENCES

- [1] F. P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [2] S. Faulk, E. Loh, M. L. V. D. Vanter, S. Squires, and L. G. Votta. Scientific computing's productivity gridlock: How software engineering can help. *IEEE Des. Test*, 11(6):30–39, 2009.
- [3] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons and IEEE Computer Society Press, 2008.
- [4] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.
- [5] B. G. Ryder, M. L. Soffa, and M. Burnett. The impact of software engineering research on modern programming languages. *ACM Trans. Softw. Eng. Methodol.*, 14(4):431–477, 2005.
- [6] D. M. Weiss and C. T. R. Lai. *Software product-line engineering: A family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [7] J. M. Wing. Computational thinking. *Commun. ACM*, 49(3):33–35, 2006.