Research toward an Engineering Discipline for Software

Mary Shaw  
Carnegie Mellon University  
Institute for Software Research  
Pittsburgh PA, USA  
mary.shaw@cs.cmu.edu

ABSTRACT  
Software engineering should aspire to be a true engineering discipline. We have made good progress in some areas, but a number of aspects of practical engineering are under-represented in our research portfolio. We have been slow to move beyond well-delimited systems developed by professional programmers to systems integrated from multiple public sources that evolve in the hands of their users. We have focused on formal reasoning and systematic testing to the detriment of qualitative and incremental reasoning supporting cost-effective, rather than perfect solutions. We have been slow to codify our results into unified theories and practical reference material. To establish a true engineering discipline for software, we need to broaden our view of what constitutes a “software system” and we need to develop techniques that help to provide cost-effective quality despite associated uncertainties.

Categories and Subject Descriptors  
D.2.0 [Software Engineering]: General

General Terms  
Design, Documentation, Economics, Human Factors, Reliability

Keywords  
software engineering research, ultra-large scale systems, end user software engineering, software design, low-ceremony evidence, everyday software, handbooks

1. INTRODUCTION

At Carnegie Mellon we understand software engineering to be “the branch of computer science that creates practical, cost-effective solutions to computing and information processing problems, preferentially by applying scientific knowledge, developing software systems in the service of mankind” [20]. The technical foundation of software engineering is a body of core computer science concepts. This technical knowledge is applied through a body of engineering knowledge about the pragmatics of design and problem solving. These are complemented by the social and economic context of the engineering effort to provide a basis for shaping engineered artifacts for their intended use.

In this view, software engineering should be an engineering discipline, on a par with the classical engineering disciplines, though with its own distinctive character – software engineering’s paradigm is computational; software’s design-intensive nature minimizes manufacturing cost; and software’s symbolic and abstract basis makes it more constrained by intellectual complexity than by fundamental physical laws. Despite these differences, software engineering shares with classical engineering the need for design techniques to reconcile conflicting constraints and achieve cost-effective results, as well as reliance not only on established scientific knowledge but also on systematically codified observations drawn from experience.

In the four decades since our field was named, we have made substantial progress in many areas, especially program development and maintenance, management and development processes, and formal foundations and analysis techniques. Notwithstanding this progress, we still lack a firm engineering basis for our field.

In those same four decades, software has become pervasive. Our society and our economy depend on software embedded in the infrastructure, on software composed of independent resources that evolve independently, and on software created and modified by people who are not trained in software engineering. Not only is software at the heart of our physical infrastructure and economic system, it has become a critical information and communication resource as the public at large comes to increasingly depend on software (modern automobiles and airplanes are increasingly controlled by software rather than mechanical linkages, and three quarters of American adults now use the Internet [16]).

The current center of attention in software engineering research is evident in the topics of conferences and in the distribution of topics within conferences. For example, the past two months of SEWORLD conference and symposium announcements [1] are dominated by conferences on formal theories, software processes, and topics dealing with the development and maintenance of code by professional programmers. This is consistent with what I have seen on occasions when I have inventoried the topics of papers in conferences: the most common topics are consistently related to development processes and cost estimation; analyzing, verifying or debugging code; and integrating and testing collections of program components. Although neither of these observations is rigorous, they are both strongly suggestive.

For software engineering to become a genuine engineering discipline we must broaden its scope to be more representative of the artifacts and activities of practical software; we must expand its range of models to include qualitative and approximate models; and we must establish a culture that rewards organizing knowledge for routine use. This position paper surveys these areas, offering suggestive examples but not fully-fleshed-out research programs.
2. BROADEN THE SCOPE OF “SOFTWARE”

Software systems are no longer just clearly-delimited, tightly-integrated collections of code modules developed as discrete projects, amenable to precise specifications and validation against those specifications. Rather, they are now composed, often opportunistically and dynamically, from independent components and from resources other than code modules. Further, software is often developed, modified, or tailored by people whose principal responsibility is not software development. In addition, software is deeply embedded in the public infrastructure, and as a result the boundary between the software system and its environment is often indistinct, so the capability of the system arises from the activities of its users as well as the core software. The scope of “software engineering” must be expanded to encompass the varieties of software that exist in the real world.

2.1 Beyond Code

The principal focus of software engineering has been program modules written in general-purpose programming languages, generally to an explicit, if not formal, specification. These modules are composed into systems, possibly with known modules acquired from third parties. In practice, however, software systems incorporate resources such as data, media, dynamic state, real-time data feeds, properties of operating context, interaction protocols, user interface designs, scripts, high-level architecture, design intent, and metadata required for compliance with policies and changing requirements. Web pages, for example, incorporate many of these elements: layout (HTML), data (XML), dynamic behavior (JavaScript et al), and layout state change (implicit, lacking good abstraction or notation) coexist in an uneasy alliance. The organization of large systems is guided by a maturing understanding of software architectures that recognizes components of diverse types that interact in specific ways, [7][21] but the current focus on object-oriented development environments has distracted attention from the architectural role of other sorts of resources.

We have good abstractions for only a few of these types of resources, and even the abstractions we do have are not well integrated with programming languages. Research in this area should refine existing abstractions, defining new ones as necessary, for these non-code resources and integrate them into software development languages and tools in such a way that they complement existing programming languages.

2.2 Beyond Programmers

Most software engineering research focuses on software developed by professional programmers using explicitly defined development processes.

In practice, however, people who develop, compose, script, or tailor software but whose principal responsibilities lie elsewhere (often called “end user programmers”) are coming to vastly outnumber professional programmers [18]. Software engineering is only beginning to support the lightweight development techniques appropriate for many of these developers. Worse, support for engineering considerations such as reliability, evolvability, suitability for task, security, and privacy is even less robust. Yahoo Pipes [26] is a good example of a task-specific language that supports abstractions appropriate for merging and filtering RSS feeds in an accessible manner, but its support for engineering aspects of construction and maintenance is meager at best. Lightweight languages such as scripting languages have been neglected for decades; Brooks identifies the greatest flaw in JCL (Job Control Language, a scripting language for IBM’s OS/360 designed in the mid-1960’s) as its designers’ failure to recognize that it was actually a programming language [3].

A research community in end user software engineering is emerging [9], and useful results are being published (a survey of current research will appear soon [15]). Research in this area should address not only engineering considerations for individual programs but also issues associated with composing separate software resources, such as spreadsheets with web pages. The challenges include not only the technical issues of raising lightweight languages to the level of design and support that we expect for traditional programming languages, but also establishing models that allow the broader population to use them effectively to develop dependable solutions to their own problems.

2.3 Beyond Integration of Program Components

Software engineering research usually assumes that a software development project has a project manager who controls the system configuration and who is aware of, and possibly in control of, changes to components. As more software resources are provided via the Internet and move to “the cloud” (that is, to distributed servers controlled by service providers rather than by the owner of the data and software), control of system configuration moves farther from the system developer. In this setting, many resources are underspecified, autonomous (they are independently created and managed and may change structure or format without notice), and heterogeneous (they are packaged with different interface assumptions rather than all satisfying a common API). The resources are often used opportunistically for purposes other than their original intended purposes. Web “mashups” exemplify this style; a common function of mashups is to aggregate information from multiple web sites [24], for example by merging information resources with Google Maps [11]. Service-oriented architectures (SOA) also exemplify many aspects of this style, though they usually draw components from a common family to ease integration (or “orchestration”).

Further, the emergence of ultra large-scale systems [23] brings new challenges for developing and controlling systems that allow for only limited central or hierarchical control; that must respond to conflicting, possibly unknowable requirements arising from social and political considerations; that must operate and evolve continuously, that have only indistinct system boundaries; and for which the character of the system emerges from the behavior of its users, not solely from the software.

Emerging research on mashups and SOAs has begun to address the richness of modern network-based software; work on self-adaptive systems offers help with reacting to changes in resources [4][6]; and the CONNECT Project is developing techniques for integrating heterogeneous components [8]. Health care initiatives are beginning to call for integration of information from many sources, with consideration for the quality and semantics of the information from each source [17].
Open research opportunities include composition techniques that monitor the individual resources to ensure proper behavior and respond to anomalies; abstractions that allow automatic replacement of defective resources (e.g., a “weather forecast” that might be instantiated from either weather.com, accuweather.com, or noaa.gov), models for defining and enforcing policies on the envelope of permitted behavior rather than specifying incidental details; and architectures that control the interactions among subsystems to isolate faults and simplify troubleshooting.

3. EXPAND THE RANGE OF OUR THEORIES AND TECHNIQUES

Civil engineering is rooted in theories of statics and dynamic flow, but civil engineering must go far beyond those theories to design structures that serve practical purposes. Engineering design requires reconciliation of conflicting constraints and the selection of solutions based on cost-benefit analyses. Engineers prefer to apply formal systems when possible, but they fall back on approximations, codified experience, and engineering judgment when sound theory is not available. They protect themselves from the resulting uncertainties by including safety factors in their designs and by relying on prior art when possible, innovating only when necessary.

Similarly, software engineering needs to move beyond the scope that is implied by our emphasis on closed-shop systems – projects with delimited scope, professional developers and explicit, preferably formal, specifications.

In a real world of limited resources, it is not practical to complete a full analysis and validation for every system. Systems have different levels of criticality and different consequences of failure. We should seek appropriate levels of quality for each application rather than acting as if we could (or should) do full validation on all. The impact of this cost-effectiveness imperative is that we need to accept approximate, qualitative, or informal assurances in some cases – but we have not developed systematic techniques for deciding how good those assurances need to be in a particular case.

3.1 Design

In software engineering, “design” often means “drawing UML diagrams”. In classical engineering, on the other hand, the design activity establishes what should be expressed in the analog of UML diagrams. Engineering design reconciles the conflicting constraints of the problem in light of the client’s priorities and budget. It seeks not only a solution to the problem, but it compares multiple alternative solutions to find a cost-effective solution that balances priorities, constraints, and costs.

Software engineering would benefit from more research on design in the classical engineering sense. This involves problem setting in Schön’s sense (deciding what to build) [19] as well as problem solving in Simon’s sense (figuring out how to build it) [22]. It involves greater attention to the translation between the expression of the requirement in the problem space and the exploration of the implementation in the solution space. It involves explicit consideration of cost as well as benefit to establish appropriate levels of dependability, performance, and other properties – that is, explicit attention to the client’s utility function.

A particular opportunity lies in finding good ways to represent design spaces – not just systematically representing the design alternatives, but the implications of those choices [2]; [3]; [25].

3.2 Low-Ceremony Reasoning

In software engineering, the “gold standard” for program correctness is the trio of formal verification, systematic testing, and empirical studies of operation. Each of these is a high-ceremony process, with assurance arising from a discrete, often resource-intensive, validation event.

In the real world, however, most of the software that most of us use most of the time is not “correct” according to the gold standard. The release notes for new system releases commonly include a section of “known issues”, many of which are errors. Yet we choose to use these “incorrect” systems because on balance we find it more cost-effective to use them than not to.

Although certain critical systems can justify high-ceremony validation, for most everyday systems our confidence arises incrementally, based on fragments of evidence that may be individually unreliable but that, taken as a whole, provide an acceptable basis for decisions. Such evidence includes reputation, user reports, editorial reviews, advertising claims, and “best X” reports based on linear functions of subjective marks. This leads to a low-ceremony process for developing confidence that is incremental, often informal, context-dependent, and possibly nonmonotonic. Although not suitable for safety-critical systems, it can incorporate many sources of information about the credibility of a product. It is also a much better match than high-ceremony techniques for use by end users.

Recommendation systems, reputation systems, machine learning, auction and betting mechanisms, and other inexact techniques offer ways to aggregate inexact or unreliable evidence. Software engineering would benefit from research on how to harness these techniques to develop confidence in software systems. The results would not be suitable for high assurance systems, but they have promise of being cost-effective for decisions about everyday software. For everyday software, the evaluation process would then involve two questions: (a) How good does this software have to be? and (b) How good do I believe this software to be?

3.3 Dependability and Trust Properties

Software engineering has a substantial body of results on dependability, security, performance, and other quality properties. Nevertheless, our industry still delivers computers that are very difficult for anyone who is not a trained system administrator to install and maintain in a safe state that establishes and preserves these properties.

Dependability, as an example of one of these quality properties, can be addressed at three levels. First, we can analyze the intrinsic dependability of a component in a given environment based on a specific set of attributes, such as availability, reliability, safety, and integrity. This yields an evaluation of the component, but only of the component in isolation. Second, we can analyze contextual dependability, based on the specific needs and priorities of that environment. For a given component, this leads to different sets of attributes and different evaluation results for each distinct environment, reflecting that environment’s needs, tolerances, priorities, and expectations. These results are not transferable to other environments, but they are more useful in
each instance. Third, we can analyze dependability in practice, considering the behavior of users as well – especially the ways that users can be confused, distracted, unmotivated, or noncompliant.

The software engineering results on quality properties are chiefly of the first kind, with some attention to the second. This might arguably have sufficed when the effects of poor dependability were limited to the owner of the system. However, in the highly inter-connected world of the Internet local vulnerability can have global effects. For example, the well-known difficulty in configuring and administering personal computers has led to huge numbers of insecure machines that are infected by malware and co-opted into armies engaged in malicious behavior that can potentially threaten the infrastructure of the network itself. Even if those personal computers were intrinsically safe (in the sense of meeting specifications or standards), indeed if they were contextually safe for their intended use, the complexity of administering them – of selecting an appropriate configuration for the operating environment – is too great for individual personal computer owners, and therefore defeats dependability in practice. Many aspects of usability, of course, lie within the domain of human-computer interaction. However, structural characteristics of systems that make them hard to understand and manage lie solidly within software engineering.

Software engineering would certainly benefit from more research on dependability and trust properties, especially security and privacy. The most pressing need, though, is for research that leads to the development of systems that can be dependable in practice. This might, for example, build on results from self-adaptive systems to develop systems that assumed a greater share of the responsibility for preserving system integrity and stability.

3.4 Codifying Engineering Knowledge for Practical Use
Software engineering research emphasizes novel, innovative results. A notable exception is the work on design patterns [5][10][13], where explication of working solutions is explicitly more values than innovation.

Classical engineering disciplines, however, value routine design over innovative design in most cases where that is practical. To accomplish this, much of the core knowledge is captured in engineering handbooks and other reference material [12]. Software engineering has done this only to a limited extent, mostly for information about specific products – and only slightly for general design knowledge and theories that transcend particular products. We need the analog of engineering handbooks that organize useful theories and pragmatics in a form that allows designers to choose techniques based on the task at hand. This will support opportunistic choices of models as exemplified by Jackson in his introduction to problem frames [14].

For example, modern adaptive software systems usually embed feedback control mechanisms to respond to uncertainties and changes in the external environment. Despite control theory’s well-established body of knowledge about feedback systems, which is referenced routinely in traditional engineering disciplines [12], descriptions of adaptive software systems rarely make the feedback loops explicit. If the feedback loops were made explicit, it would raise validation obligations such as establishing that the current state of the system is adequately modeled, that the control strategy is appropriate to the problem, and that the controller has sufficient command authority over the controlled process. Failing to make the feedback loop explicit leads designers to ignore these obligations [4]. This is particularly problematic when control loops at different levels of abstraction interact. If the control theorists’ understanding of the varieties of control and the associate proof obligations were codified in such a way that software engineers could incorporate that knowledge – and if the control view were treated as a first-class view, on a par with views supported by UML – then it would be much easier to establish correctness of this sort of adaptive system.

An important challenge for software engineering is to develop reference materials that unify and codify the results of our research. The classical engineering model of publishing large tomes at multi-year intervals will not suffice for the rapidly-developing body of knowledge in software engineering, so part of this challenge is to find a way to develop these reference materials online.

4. RECOMMENDATION
This position paper starts with the proposition that software engineering should be an engineering discipline and the research community should establish a basis for that discipline. It identifies discrepancies between the types of software that dominate the attention of the research community and the types of software in practice. It goes on to identify discrepancies between the types of theories and techniques that dominate the attention of the research community and the types of theories and techniques that area appropriate to software in practice, noting research areas that are under-represented in our portfolio.

This analysis intrinsically yields a breadth-first overview, so the specific examples are suggestive, not comprehensive, and none of the examples are fully fleshed out as full drafts of research initiatives. It does not lead to an overarching unified theory that solves all problems of practical software. The world is too rich, too complex, to hope for this.

This analysis does not suggest that current work fails to contribute to the objective. We have thriving communities making substantial progress on theoretical models, development processes, and code-level issues. These are all parts of the engineering discipline we seek. As we plan future research initiatives, though, the points of highest leverage are the under-represented areas, where progress will break new ground.

Needless to say, good models and theories should apply despite changes in applications and platforms. Research that narrowly pursues the current technology or the current fashion is unlikely to have enduring effect. What’s new here is the recognition that intrinsic uncertainties of modern systems lead to issues that are not addressed by current models.

5. ACKNOWLEDGEMENT
This work was supported by the A. J. Perlis Professorship in Computer Science.
6. REFERENCES
[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley 1995.