Software Engineering is indeed in a crisis

Michael Buettner, Peng Dai, Suporn Pongnumkul, Richa Prasad

The IEEE defines software engineering as, "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software." As software is the result of software engineering, if software is in crisis then software engineering itself is in crisis. We argue that software engineering is indeed in crisis, as it has failed to significantly mitigate the "software crisis" of the 1980s which was characterized by over-budget and over-time projects of low quality. Additionally, as the problems of software are fundamentally insoluble using the techniques of engineering, software engineering will continue to resemble a craft more than an engineering discipline and we can expect no significant change from the current state of the industry.

1. Software in Crisis

When it comes to large scale software development, the statement: "Good, fast, cheap - choose two," is often woefully optimistic. According to Jim Johnson, the founder and chairman of the Standish Group, "People know that the more common scenario in our industry is still: over budget, over time, and with fewer features than planned." The 2004 Standish CHAOS report, which researches the reasons for IT project failure in the United States, shows that 18% of software projects fail, 29% succeed, and 53% are "challenged". While absolute numbers such as these have limited value, a comparison of yearly CHAOS reports shows that while projects have become more successful, the problems that characterized the "software crisis" of the 1980s are still the common case. In addition, the number of successful projects, and the number of projects that are over-budget and over time (about 50% and 80% respectively), has remained relatively constant from 1998-2004 [1]. This suggests that the beneficial impact of software engineering innovation has leveled off. As projects are increasing in complexity, simply holding our ground with respect to failures is admirable. However, with multi-core technologies poised to pull the rug from under our traditional programming paradigms, software engineering may find itself unable to maintain even this modest feat.

Due to the physical limitations of transistor density in single core processors, higher computation power in the future is expected from multi-core processors. However, we are currently ill-equipped to make use of this parallelism. While software written for a single processor can expect a performance increase proportional to the future increase in processor speed, this is not the case with an increase in the number of cores in a processor. In [2], Lee asserts that threads, the current method of concurrent programming, will be unable to exploit the parallelism of multi-core systems. In effect, he states that nothing short of an entirely new concurrent coordination language will allow us to make use of these advancements in architecture. Unfortunately, this language is as of yet unnamed and un-described, while multi-core processors are already on our desks. Further evidence that we will be unable to exploit the increase in processor cores is shown in [3], where the authors port Linux to 16 and 32 core architectures that use transactional memory, which has been touted as a solution for providing concurrency in multi-core systems. The results of the study show that performance was on par with a single processor system as the synchronization costs dominated with increasing number of cores. The

situation will be worsened by the fact that chip manufacturers plan to scale to over 1000 cores in the coming years.

If there was in fact a software crisis during the 1980s, then the state of software is still in crisis as projects are still over-budget, over time, and of low quality. Consider Microsoft Vista as the quintessential case [14][15][16][17]. While we rarely hear about the software crisis today, this is due to a change of terminology and not a change in situation.

2. Failure of Current Software Engineering Practices

Since software engineering is the process by which software is produced, it stands to reason that a critical analysis needs to be performed of current software engineering practices in order to solve the software crisis. In fact, it is common knowledge that there is widespread observance of poor software engineering practices. For instance, most software developers do not adhere to even the basic models in place to cope with changing design requirements. Lack of unit-tests, global variable usage, and undocumented code are some of the poor practices that guarantee bug-ridden software. It is also true that often the requirement specification for new software is not very clear; such as when the software is to help create new markets. In such scenarios, it is inevitable for the design to be imprecise, which may lead to errors in code depending on how far the software development process has progressed. The lack of well documented, formal processes in software engineering has led to developer-specific code, which is difficult to transition to a new software engineer, thus creating problems in its maintenance and reuse in new software.

There are three main aspects of software engineering that garner scrutiny:

1. Software Developers
2. Innovation of Tools and Models
3. Currently used Tools and Models

2.1 Software Developers

In his book, *Software Engineering Economics*, Barry Boehm argues that individual and team productivity is the leading predictor of software costs; it's twice as significant as product complexity [12]. Software companies have long complained of the lack of good software engineers. This may be due to the fact that in software engineering, unlike most other engineering disciplines, it is more common for software engineers to have disparate educational backgrounds; from no formal training to a doctorate in the subject. Thus, there is no guarantee of a reasonable level of competence and homogeneity among software engineers in their approach to developing software.

There is also the issue that it is not sufficient for only individuals intending to pursue software engineering as a career to demonstrate a formal background in the field. Today, many companies in niche fields, such as Boeing in aerospace engineering, let engineers from a non-software engineering background develop their software. They fail to realize that while it may be relatively easy to write software, it is very hard to write good software – bug-free, extensible, easily maintainable software. A consequence of this has been a large contribution to the increasing mound of bad software.

If we consider the former situation from the opposite perspective, we note that the demands on software engineers are unique in that they are often required to produce software for fields that are beyond their area of expertise. For instance, companies in the medicine and business domains are two significant recruiters of software engineers. This often leads to software developers learning the requirements along the development process and probably never being truly aware of all possible boundary conditions beforehand. Thus, it is inevitable for the final product to be bug-ridden, behind schedule, and over-budget.

Besides the lack of a formal background among all software engineers and the interdisciplinary nature of software engineering, another issue to ponder is whether the education provided in the field is adequate [13]. Firstly, computer science students adopt an ad-hoc approach to software engineering from early on since the formal software engineering courses occur at a much later stage in the curriculum, if at all. The adage of "Old habits die hard" fits well here. Secondly, often computer science courses are extremely product oriented with little or no emphasis on the software engineering process. For instance, the focus is usually on whether the 'best' algorithm was used to create an efficient working final software. While this is important, it is also vital for students to learn the value of understanding the software process from the beginning to the end and beyond. Students need to be aware of the relation between good software engineering practices and good software. The inadequacy of software engineering education is not contained to universities alone since the computer science graduates enter the workforce with the same poor practices.

2.2 Innovation of Tools and Models

Innovation in the field of software engineering has been rapid due to the urgent need to solve the software crisis. Many software tools and models have been invented such as Object Oriented Programming, Structured Programming, Expert Programming to name a few. Despite this seemingly quick generation of useful solutions, in 1985, Redwine and Riddle [4] found that "it takes on the order of 15 to 20 years to mature a technology to the point that it can be popularized and disseminated to the technical community at large." This could be due to two potential reasons: the intransigence of the practitioners, and the irrelevance of the innovation. We can easily see that adopting a new technology involves high overhead cost but it is not necessary for all software development processes to pay such a high cost. One would especially hope that high security and correctness critical areas would be more willing to adopt new software engineering tools and practices because of their sensitive nature. This is in fact far from the truth as study [5] shows that even in medical applications, software engineering tools and principals are not being applied much, which indicates the reluctance of programmers to adopt new software engineering tools.

As for the relevance of research, Potts [6] explained, in 1993, that the limited impact of software engineering research on the real world is caused by the research-then-transfer model, where research is done in a research lab and by researchers who do not possess industry experience. He proposed that software engineering research should be industry-as-laboratory. Even though there is more industry-as-laboratory research since then, Fichman and Kemerer's study [7] in 1999 indicates that the 'assimilation gap' between the first acquisition of a new technology and its 25% penetration into software development organizations is found to be 9 years for relational databases, 12 years for Fourth Generation Languages, and a much longer

period for computer aided software engineering (CASE) tools. Therefore, to date, both - intransigence of practitioners and irrelevance of innovation - are causing slow software engineering technology transfers. It is also alarming to note that there have been few, if any, studies on the causes of slow transfer of technology.

2.3 Currently used Tools and Models

There is a plethora of tools and models currently available for software engineers to use. Some of the directions in which software engineering is developing include [18]:

1. Aspect-oriented programming
   - Aim: Developing new languages for more systematic programming
   - Criticism: Inherent ability to create unpredictable and widespread errors. Slow learning curve
2. Agile software development
   - Aim: Emphasizes development iterations throughout the life-cycle of the project
   - Criticism: Only useful when practiced by programmers of above average capability. Increases likelihood of scope creep.
3. Experimental software engineering
   - Aim: Scientific attempts to understand software by doing experiments on them
   - Criticism: Results of any one study cannot simply be extrapolated to all environments because there are many uncontrollable sources of variation between different environments [19]
4. Model Driven Software Development
   - Aim: Code generation from models
   - Criticism: Cost and feedback process are some of the issues seen in models such as the Waterfall Model or Spiral Model
5. Software Product Lines
   - Aim: Emphasizes extensive, systematic, formal code reuse
   - Criticism: Assumes all products neatly fit into a product line

As a result none of these ideas have been widely accepted. This has lead to many great ideas and tools that are not used in practice, because none of them have proven to be significantly useful.

One might argue that high-level programming languages greatly decrease the complexity of programming. It is true that high-level languages make complex programming easier. However, the languages themselves are not a solution [8]. Although they help break a complex problem into pieces, the sub-problems themselves can still be quite complex. Today we have an increasing number of advanced high-level languages, but the situation has not changed significantly. Object-oriented programming is an old idea, but it was not until the last decade that it is has become popular. Its real contribution to ameliorating the software crisis is in removing difficulties from design expressions, where there is only limited functionality due to the small portion of type-related issues.

Automatic programming and graphical programming are two realistic and popular ideas. However, they are no more than good pictures. For example, it is extremely hard to characterize a problem with few parameters, which is the requirement of automatic programming. The lack of existing sub-solutions is another problem of automatic programming, and it is not trivial to simply reuse an existing solution due

to the variance of requirements, hardware constraints, etc. The problem of graphical programming software is the difficulty in describing a program completely via diagrams, as many programs do not land themselves to complete description via graphical modeling languages.

3. Possibility of Good Software Engineering

3.1 Software Engineering is not Engineering

Unlike engineering disciplines such as hardware engineering, which have set procedures for arriving at a solution, software engineering takes on a more ad-hoc approach. Software developers often rely on their talents and skills and usually mature to have a unique coding style. On the other hand, they also need to emphasize reproducible, quantifiable techniques – a requirement of any engineering discipline. It is this divergence between approach and targeted outcome which is responsible for much of the failure of software engineering.

The question then arises whether we can make software engineering more like other engineering disciplines. In order to do so, we would have to discard all adhocracy. This means that we could no longer simply add or remove parts of the code according to changing requirements, but instead would produce entirely new pieces of software for each enhancement. Management would have to restrain itself from pushing developers at the last minute to release software that takes advantage of new hardware architectures. Besides the unreality of such an approach, it is unlikely that bug-free, all requirements met, and on-budget software would be produced. Thus, software development is unlikely to ever morph into a strict engineering discipline.

3.2 Software Engineering is not Mathematics

One line of work in software engineering that hopes to make software engineering more like mathematics is formal methods and verification. Ideally, program verification will give users confidence that the programs have certain properties and will not fail. However, even published mathematical proofs can be wrong as evident in cases where the proof was believed for a decade before someone found a fatal flaw, or when two proofs are contradictory but neither can be discredited [9]. The mathematics proofs, however, can be read, and the mathematician community as a whole gradually verifies the proofs and correct mistakes. This process allows people to believe in the outcome of mathematics. Nevertheless, this is not the case in software verification. The verification cannot be read and you either blindly believe it, or do not believe in it. Therefore, formal verification cannot serve the same purpose for software engineering as proofs do for mathematics.

3.3 Software Engineering Cannot Be Codified

SWEBOK[10], a recent effort by the ACM that tries to solve the software quality problem by improving the competency of software developers. It was intended to provide a body of knowledge for the industry, and software engineers could then be certified based on that minimum set of knowledge. This seemingly workable approach failed finally due to the complexity in the software development behavior itself. For example, there is no general agreement upon essential knowledge for programmers. It is also hard to quantify the requirements of different

roles in a software development team. In [11], the authors state that even a conceptually clear and generally accepted organizing principle is lacking for a software engineering body of knowledge, and is unlikely to exist any time soon. In effect, software engineering techniques cannot be resolved into a defined set of principles in the same manner as techniques of other engineering disciplines.

4. Conclusion

The current state of software, while measurably better than during the so called crisis of the 1980s, is still plagued with problems. It appears that the easiest solution is to double our budget and time estimates. However, the problem of quality may not be so easily remedied. In particular, the emergence of highly parallel multi-core processors will challenge the way in which we must think about, and abstract, software problems. Currently, there is not only no silver bullet for this problem, there are no clear roads forward for addressing this paradigm shift.

Ignoring the imminent problem of parallelism, the current practices of software engineering have failed to produce real solutions for creating on-time, in budget, quality software. This is partly due to software engineers not making use of the best practices, but also is caused by the slow transfer of software engineering techniques from research into industry. However, even those techniques that have come into prominence, such as high level programming languages and graphical modeling techniques, have not resulted in rapid and error free software.

The underlying reason for the limited ability of software engineering techniques to solve the problem of creating good software is that software is not reducible by traditional mathematical or engineering techniques. The large input state space and internal complexity of software results in software engineering being more of a craft than an engineering discipline. While crafts can surely be done well, the skills necessary for this can only be learned over time and in a relatively ad-hoc manner. Consequently, universities will never be able to graduate good software engineers as if from a cookie cutter, and the software crisis is unlikely to change in anything more than name.

5. Bibliography

[1] Deborah Hartmann. "Interview: Jim Johnson of the Standish Group", InfoQ. Aug 25, 2006. <http://www.infoq.com/articles/Interview-Johnson-Standish-CHAOS>

[2] Edward A. Lee. The Problem with Threads. Technical Report UCB/EECS- 2006-1, EECS Department, University of California, Berkeley, January 10 2006.

[3] CJ Rossbach, OS Hofman, DE Porter, HE Ramadan, A Bhandari, E Witchel, "TxLinux: Using and Managing Hardware Transactional Memory in an Operating System", SOSP, 2007.

[4] S. Redwine and W. Riddle , Software technology maturation. *Proceedings of the 8th International Conference on Software Engineering, London* (1985), pp. 189–200.

[5] Christian Denger and Raimund L. Feldmann and Martin Höst and Christin Lindholm and Forrest Shull. "A Snapshot of the State of Practice in Software

Development for Medical Devices" ESEM, IEEE Computer Society Press, 2007, pp. 485-487

[6] C. Potts, Software Engineering Research Revisited, IEEE Software, September, 1993, pp. 19-28.

[7] R.G. Fichman and C.F. Kemerer , The illusory diffusion of innovation: an examination of assimilation gaps. *Information Systems Research* **Sept** (1999).

[8] Fredrick P. Brooks, Jr. "No Silver Bullet: Essence and Accidents of Software Engineering", Computer, April 1987. This is also found in Brooks' Mythical Man-Month (25th Anniversary Edition).

[9] De Millo, R. A., Lipton, R. J., and Perlis, A. J. Social processes and proofs of theorems and programs. Commun. ACM 22, 5 (May. 1979), 271-280.

[10] A. Abran, J.W. Moore, P. Bourque, R. Dupuis and L.L. Tripp, Guide to the Software Engineering Body of Knowledge (SWEBOK), Version 1.00, IEEE Computer Society Press, Los Amitos, CA, USA (2001).
[11] David Notkin, Michael Gorlick, Mary Shaw. "An Assessment of Software Engineering Body of Knowledge Efforts", A Report to the ACM Council (May 2000).

[12] Dianna Mullet. "The Software Crisis", University of North Texas. Feb 12, 2007. <http://www.unt.edu/benchmarks/archives/1999/july99/crisis.htm>

[13] Bruce F. Webster. "The Real Software Crisis", BYTE. Jan 1996. <http://www.byte.com/art/9601/sec15/art1.htm>

[14] Gregg Keizer. "Is Windows = st1 />= ST1 />Vista Slower than Windows XP", The Guardian. Dec 6 2007.
<http://www.guardian.co.uk/technology/2007/dec/06/microsoft>

[15] Fuad Abazovic. "Vista gaming will be 10 to 15 per cent slower than XP". The Inquirer. Oct 7, 2006.
<http://www.theinquirer.net/en/inquirer/news/2006/10/07/vista-gaming-will-be-10-to-15-per-cent-slower-than-xp>

[16] Suzanne Tindal. "Windows XP outshines Vista in benchmarking test". CNET News. Nov 27, 2007. <http://www.news.com/Windows-XP-outshines-Vista-in-benchmarking-test/2100-1016_3-6220201.html>

[17] "Testers: Updated Windows Vista still slower than Windows XP". Fox News. Nov 30, 2007. <http://www.foxnews.com/story/0,2933,314141,00.html>

[18] "Software Engineering:: Current Trends in Software Engineering". Wikipedia. <http://en.wikipedia.org/wiki/Software_engineering>

[19] Forrest Shull et. al. "Knowledge Sharing Issues in Experimental Software Engineering". Empricial Software Engineering. Oct 28, 2004.