There is no software engineering crisis

Alex Colburn Jonathan Hsieh Matthew Kehrt Aaron Kimball

January 16, 2008

Introduction

There is no software engineering crisis. In order to determine what is meant by the phrase "software engineering crisis," it is informative to define both the term crisis and the term software engineering. We define crisis to mean a "crucial or decisive point or situation; a turning point." [1] While not part of the definition, the usual connotation of a crisis further implies an impending disaster as a result doing nothing or making the wrong choices. The IEEE defines software engineering as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" [3]. When considering the complete phrase "software engineering crisis," it is also helpful to consider the historical context of the term and its subsequently use. We ask is software engineering really at a turning point? Are there dire consequences if we make the wrong choice on how to proceed? To answer these questions we consider the "software crisis" described in 1968 by NATO. We note that we are not currently in the same "software crisis", or for that matter, any subsequent crisis. We also debunk the claims that we are in the midst of a perpetual and "chronic software crisis." by considering the major categories of historical crisis claims. We conclude that the ever-increasing capabilities, ambitions, and expectations placed upon software and software engineers are not signs of a crippling crisis but a natural by-product of healthy exploration and evolution of increasingly improved software.

The Crisis of 1968

If we are in the midst of a "software crisis" today, it is surely a different one than what was described when the term was first coined and debated in 1968. Discussions at the NATO Software Engineering Conference specifically lamented the difficulties in predictably producing correct, understandable, and verifiable computer programs. [5] The result of the debate and following meeting in 1969 created a field that would be the long-term antidote to the crisis: software engineering. To use a medical analogy, they finally had a diagnosis, and realized then that addressing the problems would require a "lifestyle-change" approach as opposed to hoping for a breakthrough panacea that addresses individual symptoms. One

could truly claim that this was the turning point that identified the root of the difficulties in building software products.

Even during the conference, the use of the term "software crisis" was criticized. Kolence stated:

> I do not like the use of the word crisis. Its a very emotional word. The basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities and our theories and methods of design and production at this time. There are many areas where there is no such thing as a crisis sort routines, payroll applications, for example. It is large systems that are encountering great difficulties. We should not expect the production of such systems to be easy. [5]

This comment makes two key points: first that there are areas that are considered solved, and second is that pushing past the previously known boundaries will always incur new difficulties.

A Chronic Software Crisis?

Could software engineering really be in a state of crisis for the past forty years? Are we in a "chronic software crisis"? [2] We believe not.

Throughout the decades there have been a series of problems that have successively been handled and integrated from research into industrial best practices. The history of the software engineering crises takes on different focuses: the inability to build large software systems, the inability to budget and schedule, the lack of productivity and the quality of software. We believe that many of these problems have been solved today, and turned difficult problems in the past into mundane problems today. Any remaining problems are not crises, but a sign of an active and vibrant field. Any crisis that may have existed in 1968 has been resolved by vast improvements in maturity of techniques for managing large software projects, which has led to higher levels of productivity in production of code and to higher quality code.

Maturity

Today, computing is still considered a young field; in 1968 it was merely in its infancy. Many of the systems we take for granted today could not even have been conceived of back then. At the time there were many systems they wanted to build but could not. Every project required new exploration and research efforts but were being sold as production jobs. Building software was a craft where every developer not only created the end product, but often created their own tools to aid in its construction.

For example, d'Agapeyeff discusses how developers could not trust the file handling routines provided by the manufacturer's control program. [2] Programmers had to rewrite them to suit their application better. Today we would call this the operating system API.

While there is active research in the area, very few people roll their own file system – the overwhelming majority trust and use the mature routines that are provided. Claiming that we are in the same software crisis today as faced at the NATO conference is absurd. While there are difficulties when developing large software systems, there is a multi-billion dollar industry that reliably produces robust computer programs, improves existing programs and maintains them.

Similarly, we also have the tools for managing large scale software projects The SWEBOK project has published hundreds of pages on the modern body of software engineering knowledge. [3] They categorize large amounts of knowledge of mechanisms for producing large software projects. There is evidence that successful software development processes and project management can and has been duplicated in other environments. One example is Motorola's CMM5 compliant team in India [2]. Though not perfect, it has shown improvements in process, and seems to be duplicable.

With such tools, we can handle scale; the multitude of massive complex and successful pieces of software demonstrates this. Examples include operating systems like MS Vista, and large suites of office applications. The IBM 360 had 20M lines of code, more recently Vista has 50+ million lines of code. There are only a small number of projects at this scale; these are large, risky pieces of software. Moreover, these projects have had problems with delays and budget overruns. These projects, however, are orders of magnitude larger and more complex than the majority of projects. In any field, increasing orders of magnitude cause existing techniques to break down. Projects of this nature would only be pursued if the reward were worth the effort. Lessons learned from these experiences gradually improve the prospects for future projects. This an evolutionary change, not a acute change due to a crisis.

Productivity

Over the past four decades, software has become ubiquitous, and our reliance on it has increased dramatically. In large part, this is because we have been able to produce very large amounts of it. In forty years, we have solved many problems which has in turn given us a plethora of tools that reduce the amount or work required to construct software. Difficult or error-prone tasks are encapsulated into these tools and have resulted in greatly increased productivity. Developing any solution always starts with building better tools. When we cannot make them better we can then focus on building with better methods.

For example, designing GUIs was once an arcane discipline but was simplified with the advent of RAD tools and later almost trivialized with HTML and JavaScript. At one point, building 3-tiered database-backed distributed applications was considered a large and difficult project prone to failure. At first this required building custom servers and clients, but later became simplified with HTTP/CGI, further simplified with servlet-like technologies, and almost trivialized with today's frameworks such as Ruby and Django. Each successive set of tools reduced the amount of work required by an order of magnitude. This improvement is not a sign of crisis: it is a sign of progress.

Quality

In 1969, the conference members shuddered to think about systems where software could be potentially life threatening. Today we have many systems such as fly-by-wire and medical devices that have life or death implications. This coincides with what seems to be a recent emphasis on problems of quality and security of software systems. Software is in fact ubiquitous – today's society would cease to function if all software were removed. Some companies trust software enough use it to trade billions of dollars. [4] This must mean that software development has actually become robust enough that it can be trusted in situations of potentially extreme danger or difficulty.

One could claim that Y2K was a software crisis due to difficulties of software engineering. In fact, the public was fed an tale of impending Armageddon due to Y2K-related computer glitches. On January 1, 2000, we woke up to find that the world had not descended into chaos. This non-occurrence shows two things. First that software practices allowed people to fix or mitigate the problems that existed. Second, it shows that software and the engineering practices used to build these programs were more robust than initially perceived. This crisis only existed because existing software was robust enough that it outlived its perceived lifetime!

There are many examples of long-running large-scale software systems. The SABRE airline reservation system built in the 70's still lives today, as do many of the early payroll systems. In fact, much code is reused – using techniques that include encapsulation of legacy programs, libraries. Even the open source movement encapsulates lessons learned and gives new development efforts a more stable base to build upon. Even if problems exist in new code, techniques such as automated software updates have become possible, and are accepted and deployed on millions of devices.

Shifting Expectations

Overall, the state of software engineering has greatly improved since 1968. We can now reliably produce programs that would be inconceivable even a decade ago. The impression of a software engineering crisis is due to constantly increasing and shifting expectations. Every time a challenge is met, a new one appears. After the 1968 crisis, the productivity crisis, and Y2K crisis and the Internet crisis, we will likely find yet another problem branded as "a crisis." As successes become routine and initially difficult tasks become nearly trivial, ambitions for software engineering projects are set ever higher. As confidence increases and success rates improve, the specter of another "software engineering crisis" rears its head because of expectations that are too high.

One issue is that we have the ability to integrate new research into production systems without fully exploring all the consequences. We are still exploring the design space for many of the systems that are being built. Many large efforts are unique engineering efforts as opposed to incremental updates on prior work. This ability to continually improve software gives us constant discomfort. We are not in a crisis – if anything, it is a problem of managing

expectations.

Conclusion

We argue that there is no crisis in software engineering. The 1968 NATO conference discussed the so-called "software crisis" of the time. It is debatable that such a crisis existed at the time, but it is certain that such a crisis has not persisted for four decades. The past four decades have seen immense improvements in the state of software engineering, making it impossible to say that any crisis that may have existed still exists now. Improvements in techniques for managing software projects have lead to improvements in productivity of software creators and to improvements in the created software. Any apparent crisis in software engineering today is due merely to the volatile nature of the modern software industry, but this is more an indicator of the health of the industry than of any problem.

References

[1] The American Heritage Dictionary of the English Language, Fourth Edition. Houghton Mifflin Company, 2004.
[2] W. Wayt Gibbs. "Software's Chronic Crisis", Scientific American, September 1994.
[3] SWEBOK Guide, 2004 version.
[4] Y. Minsky. Caml Trading. In Proc. POPL, 2008
[5] P. Naur and B. Randell, (Eds.). Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO, 1969.