A Historical Introduction to Software Maintenance & Evolution

Jeff Offutt
George Mason University

January 2018

This essay prints a short historical overview of how software maintenance has changed over the years and how the term "evolution" is not more appropriate than "maintenance."

Traditional Quality Attributes (1980s)

When I was in school in the 1980s, most software projects only cared about two quality attributes:

1. Efficiency of process (time-to-market)
2. Efficiency of execution (performance)

These are still taught as the most important attributes to computer science students. It is usually taught implicitly, as that is how assignments are graded. It might have been true in 1980, but is very seldom true in the 21st century.

A few years ago, I surveyed about a dozen project managers and asked them what the most important quality attributes ("ilities") were to their projects. Every single manager listed the first three in the same order:

1. Reliability
2. Usability
3. Security

The next four varied in order, but were reasonably consistent:

4. Availability
5. Scalability
6. Maintainability
7. Performance & Time-to-market

Note that performance and time-to-market were still in the list, but pretty far down. Why do we have such a mismatch in what we teach and what industry needs? I think a historical view of how software projects have grown is instructive.

Software Projects Across the Decades

In the 1960s, programmers built “tiny log cabins.” Most projects were built by a single-programmer, and were not very complex. The engineering process used was not very important, largely because the design could be kept in one programmer’s short term memory.

In the 1970s, programmers started to build houses. Most teams were small and there was a strong focus on algorithms and programming. The software was larger and more complex, and programmers had to start thinking harder. The lack of emphasis on engineering process led to disasters where projects were not completed, completed late, or failed spectacularly after they were completed. A very important point is that, for most of the industry, the quality of the software did not affect profits. There was very little competition, almost nobody built high-quality software, and expectations were extremely low. But the costs were starting to increase.

Parts of the industry saw these changes sooner than most of the industry. Space systems, aircraft systems, weapon systems, and some other safety-critical applications were building software that was larger, more complex, and had higher reliability needs. In fact, the NATO Science Committee sponsored conferences in 1968 and 1969 that introduced the terms “software crisis” and “software engineering” [1].

In the 1980s, teams of programmers were building office buildings. We needed significant teamwork and communication, including clear requirements and design that could be shared and archived. The software was much more complex and relied heavily on data abstraction. Unfortunately, the use of poor processes and ignorance of the need for process created many spectacular software failures. It was very clear that we no longer had the skills and knowledge for successful software engineering.

This process accelerated in the 1990s, when large teams were creating skyscrapers. Industry needed more than teamwork, communication, and cooperation; we needed totally new technologies, including languages, modeling techniques, and engineering processes. Software development changed completely in the 1990s. New languages (Java, UML, etc) led to revolutionary procedures, and new tools such as IDEs and deployment engines dramatically increased productivity. Sadly, more productivity often meant programmers could simply make more mistakes, faster. Our education fell further and further behind as universities continued to focus on teaching algorithms and theory, using outdated languages, and emphasize speed to the exclusion of all else–performance and time-to-completion. Computer science education became more and more competitive, and students who valued cooperation, communication, and quality moved out of the field.

Two huge waves started in the 1990s that dramatically changed the nature of software engineering. The first was subtle and fundamental–software became competitive. More users and more uses for software led to more software companies, which in turn gave software purchasers choice. The obvious, and probably most significant innovation in the 1990s was, of course, the web. As they say, “the web changes everything,” especially software engineering.

By the 2000s, software companies were using large teams to build integrated collections of continuously evolving cities. Algorithm design and programming was no longer the primary focus of software development, modifying and expanding existing software was much more important. Component-based software, “glue software,” and continuous evolution became essential. New applications, led by web software, made quality attributes such as reliability, usability, and security critical to the success of software. CS education fell so far behind it started to look almost obsolete. Professional engineers often learn more from training courses after college than they did in college. By the 2000s, we had relatively little new software development. Most professional engineers join projects after they start and spend most of their career modifying existing code bases.

The changes have, if anything, accelerated in the 2010s. In addition to web applications moving from novel to standard, software has transformed our mobile telephones into incredibly powerful pocket computers, giving us the ability to shop, register for classes, pay taxes, monitor current events, be entertained, and not get lost. And as a bonus, we can use these computers to communicate. Software is also invading our homes in many ways (IoT), including our thermostats, garage doors, ovens, washing machines, and everything else that uses electricity. In the 2010s, software is ubiquitous and diverse. Software looks like metropolises and small gadgets. Perhaps most importantly, users expect the software to be designed and work better than ever before, and to continuously evolve.

As someone who has seen five decades of software engineering, the pace of change has been exhilarating. We have gone from building log cabins to houses to office buildings to skyscrapers to building the most complicated engineering systems in human history in just half a life-time. Civil engineers took thousands of years to make this much progress, and the most complicated civil engineering products pale in comparison to the complexity of modern software systems. Electrical engineers took a couple of centuries to reach this much complexity. There is simply no way that humans could have kept up. Among other things, that means that all software engineers must be continuous lifelong learners.

Theory, Education, and Practice

What do students learn in college? Primarily how to build houses. Most universities have one general software engineering course that introduces a few concepts about buildings. But the way we build software has changed dramatically since the CS curriculum stabilized in the 1980s.

What can you do? As a professional software engineer, some things turn out to be simple. Program very neatly so others can change your software later. Design to make future changes easy. Follow processes that make changes easy. At a higher professional level, learn from your colleagues, teach your colleagues, and never stop learning. Take as many training courses as you can, and if you don't have one already, go back to school and start that master's degree.

References

[1] NATO Software Engineering Conferences, Brian Randell, 2001. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/