

Learning from Hardware: Planning

Watts S. Humphrey

There has been a long-term effort to apply traditional engineering methods to software. While some portray these methods as the answer to software's many problems, others argue that they are rigid, constraining, and dehumanizing. Who is right? The answer, of course, is that the appropriateness of any method depends on the problems you are addressing. While highly disciplined methods can be bureaucratic and reduce creativity, a complete lack of structure and method is equally if not more damaging.

Engineering and Craftsmanship

One way to look at this issue is as a continuation of the craft versus engineering debate that has raged for over a century. In his recent paper, Kyle Eischen describes the long-running argument about individual craftsmanship versus structured, managed, and controlled engineering methods [1]. Before the advent of software, craft-like methods had always had a serious productivity and volume disadvantage. There simply were not enough skilled craftsmen to meet society's demands for quality goods and services.

Factories were developed as a way to meet the need for volumes of quality goods, and manufacturing plants continue to serve these same needs today. The objective, of course, was not to destroy the crafts but to devise a means for using large numbers of less-expensive workers to produce quality products in a volume and for a cost that would satisfy society's needs. However, the widespread use of factories has required orderly processes and products that were designed to be economically manufactured in volume. This has led to many of today's engineering practices.

While these volume-oriented engineering methods have generally been effective, they have also often been implemented improperly. This has caused resentment and has reduced engineering efficiency and produced poorer quality products. However, as Deming, Juran, and many others have pointed out, this is not because of any inherent problem with engineering methods but rather with how these methods have been applied [2, 3, 4, 5, 6].

The Software Problem

As far as software is concerned, our current situation is both similar and different. The need again is to economically produce quality products in volume. Further, since the volume of software work is increasing rapidly, there is a need to use larger numbers of workers. While this might imply the need for factory-like methods that use lower skilled

people, this cannot be the case with software. This is because software is highly creative intellectual work.

The push toward more of an engineering approach for software is not caused by a shortage of skilled people. Even though we have periodically had programmer shortages, these shortages have not been caused by a lack of potential talent. There appears to be an almost unlimited supply of talented people who could be trained for software jobs, if given suitable incentives. The push for engineering methods comes from a different source, and it is instructive to examine that source to see why we face this craft-engineering debate in the first place.

The Pressure for Improvement

The source of pressure for software process improvement is the generally poor performance of most software groups. Products have typically been late, budgets have rarely been met, and quality has been troublesome at best. From a business perspective, software appears to be unmanageable. Since software is increasingly important to most businesses, thoughtful managers know that they must do something to improve the situation.

From a management perspective, software problems are both confusing and frustrating. Businesses require predictable work. While cost and schedule problems are common with technical work, most engineering groups are much better than we are at meeting their commitments. Senior managers can't understand why software people don't also produce quality products on predictable schedules and with steadily declining costs. They need these things to run their businesses and they expect their software groups to be as effectively managed as their other engineering activities. This management unhappiness spans the entire spectrum from small one- or two-person software projects to large programs with dozens to hundreds of professionals.

Software Background

While the performance of software projects has been a problem for decades, software people have not historically applied traditional engineering methods. They have not typically planned and tracked their work, and their managers often either didn't believe the software problems were critically important or they didn't know enough about software to provide useful guidance.

This situation is now changing, and the pressure for better business results is causing the software community to apply the principles and practices that have worked so effectively for other engineering groups. Among the most important of these practices is project planning and tracking. Therefore, the pertinent questions are

Do engineering planning and tracking methods apply to software?
If they do, need they be rigid and constraining?

To answer these questions, we need to look at why planning and tracking were adopted by other engineering fields and to consider how they might be used with software.

Plan and Track the Work

For any but the simplest projects, hardware engineers quickly learn that they must have plans. The projects that don't have plans rarely meet their schedules and, during the job, nobody can tell where the project stands or when it will finish. On their very first projects, most hardware engineers learn to make a plan before they commit to a schedule or a cost. They also learn to revise their plans every week if needed and to keep these plans in step with their current working situation. When engineering groups do this, they usually meet their commitments.

Some years ago, I was put in charge of a large software group that was in serious trouble. Their current projects had all been announced over a year earlier, and the initial delivery dates had already been missed. Nobody in the company believed any of the dates, and our customers were irate. The pressure to deliver was intense.

When I first reviewed the projects, I was appalled to find that no one had any plans or schedules. All they knew was the dates that had been committed to customers, and nobody believed them. While everyone agreed that the right way to do the job would be to follow detailed plans, they didn't have time to make plans. They were too busy coding and testing.

I disagreed. After getting agreement from senior management, I cancelled all the committed schedules and told the software groups to make plans. I further said that I would not agree to announce or ship any product that did not have a plan. While it took several weeks to get good plans that everyone agreed with, they didn't then miss a single date. And this from a group that had never met a schedule before.

The Key Questions

If planning is so effective for everybody else, why don't software people plan? First, software people have never learned how to make precise plans or to work to these plans. They don't learn planning in school, and the projects they work on have not generally been planned. They therefore don't know how to plan and couldn't make a sound plan if they tried. Second, nobody has ever asked them to make plans. When plans are made in the software business, the managers have typically made them and the engineers have had little or nothing to do with the planning process. The third reason that software people don't plan is that, without any planning experience, few software people realize that

planning is the best way to protect themselves from unrealistic schedules. The fourth reason is that management has been willing to accept software schedule commitments without detailed plans. When management realizes the benefits of software plans, they will start demanding plans and then, whether we like it or not, software people will have to plan their work.

The Answers

So the answer to the first question, “Do these engineering methods apply to software?” is a clear and resounding yes. The answer to the second question, “Are these engineering methods really rigid and constraining?” depends on how the methods are introduced and used. Any powerful tool or method can be misused. The guideline here is this: Does the method’s implementation assume that some higher authority knows best, or is the method implemented in a way that requires the agreement and support of those who will use it?

Any method that requires unthinking obedience will be threatening and dehumanizing to some, if not to all, of the people who use it. This is true whether the method requires you always to plan, refactor, or document, just as much as if the method requires that you never plan, refactor, or document. All methods have costs and advantages, and any approach that dictates how always to do something is rigid and constraining. The key is to learn the applicable methods for your chosen field, to understand how and when to use these methods, and then to consistently use those methods that best fit your current situation.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Noopur Davis, Julia Mullaney, Bob Musson, and Marsha Pomeroy-Huff.

In closing, an invitation to readers

In these columns, I discuss software issues and the impact of quality and process on engineers and their organizations. However, I am most interested in addressing the issues that you feel are important. So, please drop me a note with your comments, questions, or suggestions. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

References

- [1] Eischen, Kyle. "Software Development: An Outsider's View." *IEEE Computer* 35, 5 (May 2002): 36–44.
- [2] Crosby, Philip B. *Quality is Free, The Art of Making Quality Certain*. New York: Mentor, New American Library, 1979.
- [3] Deming, W. Edwards. *Out of the Crisis*. Cambridge, MA: MIT Center for Advanced Engineering Study, 1982.
- [4] Humphrey, W. S. *Managing the Software Process*. Reading, MA: Addison-Wesley, 1989.
- [5] Humphrey, W. S. *Winning with Software: An Executive Strategy*. Reading, MA: Addison-Wesley, 2002.
- [6] Juran, J. M. & Gryna, Frank M. *Juran's Quality Control Handbook, Fourth Edition*. New York: McGraw-Hill Book Company, 1988.

The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University. This article is intended to stimulate further discussion about this topic.

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.

® Capability Maturity Model, Capability Maturity Modeling, Carnegie Mellon, CERT, CERT Coordination Center, CMM, and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

SM Architecture Tradeoff Analysis Method; ATAM; CMM Integration; CURE; IDEAL; Interim Profile; OCTAVE; Operationally Critical Threat, Asset, and Vulnerability Evaluation; Personal Software Process; PSP; SCAMPI; SCAMPI Lead Assessor; SCAMPI Lead Appraiser; SCE; Team Software Process; and TSP are service marks of Carnegie Mellon University.

TM Simplex is a trademark of Carnegie Mellon University.