"The Demise of the Waterfall Model Is Imminent" and Other Urban Myths

Phillip A. Laplante and Colin J. Neill,
Penn State University

In software engineering, HOW COMMON IS COMMON SENSE?

Rumors of the demise of the Waterfall Life-cycle Model are greatly exaggerated. We discovered this and other disappointing indicators about current software engineering practices in a recent survey of almost 200 software professionals. These discoveries raise questions about perception versus reality with respect to the nature of software engineers, software engineering practice, and the industry.

WHY DO URBAN MYTHS EXIST IN SOFTWARE ENGINEERING?

About two years ago, we asked ourselves the question, "What practices are really being used in the specification and design of software systems?" We were under the usual impressions about the demise of the use of the Waterfall model and the adoption of various best practices. Our understanding was based on echoed assumptions of authors, but we couldn't recall justification for these positions. A search of the literature, unfortunately, provided no convincing support for the conventional wisdom. Given the lack of data, therefore, we thought that a survey of practitioners from a diverse group of small and large companies in defense, pharmaceutical, chemical, telecommunications, banking, and government industries (including several Fortune 500 companies) would be enlightening.

We built a Web-based survey instrument, but rather than enumerate the questions or survey mechanics, we refer the reader to that site.[1] Data was collected over a seven-week period during the spring of 2002. Of the 1,519 individuals who received both an e-mail invitation and a reminder, 194 completed the survey,[2] a response rate of approximately 13 percent.

The survey results convinced us that so-called conventional wisdom is akin to urban mythology. These myths persist because we want to believe them—and because no data exists to refute them.

Don't worry. We're not going to review the survey results here. These results can be found, without interpretation, in an article we previously published.[3] Instead, we want to opine on some of the more interesting responses and their implications. Be warned, however, we are about to enter a "no-spin zone," or more appropriately, a "no-myth zone."

MYTH I: THE DEMISE OF THE WATERFALL LIFE-CYCLE MODEL IS IMMINENT

The Waterfall process model (in which a software product is viewed as progressing linearly from conception, through requirements, design, code, and test) is a relic of yesteryear. Introduced (but not named) by Winston Royce[4] in 1970 when computer systems were monolithic, number-crunching entities with rudimentary front ends (by today's standards) and users' needs were filtered through the partisan minds of the computer illuminati building the systems.

OK, perhaps that's a little strong, but it's fair to say that most systems built in that era were spec'ed out by the programmers themselves—with little input from what we would now call stakeholders. In such an environment the Waterfall works. Requirements seldom change after specification because users are not involved in the development; they can't provide feedback about incorrect assumptions or missing features and needs. This era is over, though. Software systems are so much closer to the user that their voices cannot be ignored; they'll reject the system if it doesn't meet their needs.

This introduces a significant force for requirements change that the Linear Sequential Model (a cunning name change in an attempt to protect the guilty) cannot tolerate. This model of development assumes that requirements are set, stable, and fully evolved before analysis begins, because development progresses linearly through the phases from requirements through system deployment. A phase is revisited only if artifacts created in that phase fail inspection, review, or test. If you run into people who dispute this argument, remind them that water doesn't flow up a waterfall.

The modern reality of software development is that change is unavoidable and must therefore be explicitly accommodated in the life cycle. It is not an error that must be fixed; it's a natural aspect of system construction. This change is not isolated to requirements, but the

requirements example is the most immediate and most significant. The more we understand something, the more we realize the flaws in our initial assumptions and conceptions. If we cannot readily adapt our solutions to these changes, the costs of accommodating such requirements "errors" escalate exponentially.

To accommodate these issues, people have suggested a number of alternative process models. An early modification to the standard Waterfall introduced prototyping as a feedback and discovery mechanism so that initial misunderstandings and omissions could be identified early. Subsequent process models attempted to further mitigate such risks by breaking down projects into a series of "mini-Waterfalls" and iterating over the tasks, or delivering increments of the entire system in a sequence of releases eventually resulting in a complete capability.

It is both surprising and disappointing, then, that in a survey of almost 200 practitioners, accounting for several thousands of projects over the past five years, the dominant process model reported was the Waterfall, with more than a third claiming its use.[5] This result raises a question: Do practicing professionals know the Waterfall when they see it? Perhaps they are confusing it with other process models. This seems unlikely, but so does its dominance. It's more likely that in many circumstances, doing the wrong thing is easier than doing the right thing—and this is not a recipe for success.

The fact of the matter is that, despite much progress, the Waterfall model isn't quite dead yet. A lot of people identify it as their development method of choice. Either they're accurately describing the situation, which is bad, or they're confused, which isn't much better. In either case the death of the Waterfall model eludes us, alas.

MYTH 2: WE THROW AWAY OUR FIRST ATTEMPT

Closely related to the choice of lifecycle model is the issue of prototyping. To many, the argument for prototyping is like the argument for motherhood and apple pie: Why wouldn't you want to explore the problem space with a rapidly constructed mock-up or skeleton? Everyone involved gets to try out their ideas and validate their understanding of the problem at hand. It also provides an ideal mechanism for customer discussion and feedback. So, we are in agreement that prototypes are great.

Well, actually, this is too simplistic. Although it's difficult to argue against prototyping per se, it's easy to argue against the uses of prototyping in practice. Developers, just like everyone else, hate to throw away the products of their labors. "I've built it once and everyone liked it, why do I have to build it all again?" is the common refrain. The obvious response, to us at least, is, "Is it as robust, maintainable, reliable, and, therefore, as fully tested as 'production-level code' (whatever that means)?"

In other engineering disciplines this isn't an issue since the prototypes couldn't be used in the final systems; they are manufactured. In software the manufacture process is a disk copy, and this has allowed prototypes to be used as final production systems.

We fail to see the advantage in this. The software industry has consistently failed to deliver robust, reliable, error-free systems, yet we continue to allow elements of solutions to persist that have not been subject to the rigors of production development; in prototypes it is common to defer structural and architectural concerns and to give scant consideration to fundamental practices such as exception handling.

There is an apropos phrase that should be applied here: "Throw the first one away." This advice isn't new; Fred Brooks wrote about it in *The Mythical Man-Month*.[6] Unfortunately, 20 years later, this is still not the dominant practice. Our survey asked respondents whether they performed prototyping, and if they did, whether they allowed those prototypes to evolve into production systems (evolutionary prototyping) or threw them away. The results from that survey question indicate that half of the time evolutionary prototyping is used. We think this is probably self-evident to many people given a little thought—ask yourself, do you or anyone on your team keep prototypes? Does that code (in original or evolved form) make it into final designs? Not only our survey, but also experience shows the disappointing reality.

Now, we're not suggesting that evolutionary prototyping cannot be used successfully. For example,

Do practicing professionals know the Waterfall when they see it?

relentless refactoring (design repair) can improve the quality of existing code. Also, situations in which few requirements are available benefit greatly from this.

Our fear is that evolutionary prototyping is being employed in situations other than those for which it was conceived, and is merely the official name given for poor development practices where initial attempts at development are kept rather than thrown away and restarted. The code might compile, it may run, and it may even pass tests, but there is more to software quality than these operational properties. We desire a host of other properties in our products. We need to develop robust, reliable, maintainable—and possibly reusable and portable systems—and these characteristics require more forethought and a wider perspective than is afforded during prototyping. The objective of prototyping is to explore an idea or technology, or to demonstrate a capability, feature, or interface—very different objectives from those described.

MYTH 3: THE INDUSTRY HAS RECOGNIZED THE VALUE OF BEST PRACTICES

The final myth we will examine here is that of methodology adoption. As professors of software engineering, we are sometimes criticized for having a tainted, academic view of the world of software development. We espouse the use of standard techniques and methodologies without consideration for tight deadlines, ill-informed managers, or a host of other real-world problems.

This is, of course, not true. We are well aware of such issues, an understanding borne of our own experiences in "industry." Our collective 25 years' experience in aerospace, enterprise systems, and application development—within both industry and academia—has exposed us to all these considerations: unrealistic expectations on budget and deadlines, irrational management and incompetent staff, moving targets of requirements, target platforms and technologies. In none of these situations has an ad hoc approach worked when attempted. So we realize that unless best practices are followed and promoted, the industry will always languish in crisis. It is simply indefensible to suggest that ad hoc, random practices will conquer the complexities of the problems we solve.

Unfortunately, many still do try to defend such a position, suggesting that the techniques don't work, they take too long, or they stymie creativity. Whatever the reasoning, it is a frightening reality that in many development efforts no systematic approach to analysis and modeling is followed. This is clear from the responses to our survey. First of all, we were surprised to discover that object-oriented techniques were used only 30 percent of the time, especially given the exposure and seeming interest in object-oriented technologies and languages. That surprise pales, however, with the shock and disappointment we felt at finding that the most dominant practice was none at all—a practice (if it can be called such) reported by a full third of the survey participants.

It is considered trite to rant allegorically about the way other engineering disciplines cope with corresponding complexities and issues, and we realize the unique problems presented by software development. Remember, though, that we are not suggesting everyone follow a specific approach; we do not promote RUP (Rational Unified Process) for all projects, CMM (Capability Maturity Model) level 5 for all organizations, XP (extreme programming) for all teams, or object-orientation for all applications. Each problem, organization, and project has its own characteristics, requiring a range of techniques and strategies—but never none!

DEBUNKING MYTHS

We realize that the opinions we draw from our results are subjective and "localized." But combined with anecdotal real-world experience, we must draw the inevitable conclusion: All is not rosy in Programmingville, USA.

So what can you do to help debunk these myths? Better, how can we help eradicate these outmoded practices so that such myths will become unassailable facts? Fight complacency, for one. Seek to be an advocate against the minions of those succumbing to inertia, who refuse to change and refuse to adopt new methodologies. Point out those who cling to the archaic—for example, the old Waterfall model—or who refuse to adopt sound practices, such as throwaway prototypes. Question what appears to be the obvious.

The second thing that you can do is to become an agent of change. Work within your organizations to adopt appropriate methodologies. Remember that a one-size-fits-all approach might work for sock buying, but it won't work for software development: A range of solutions and techniques is required. Promote sound practices, especially with respect to your more senior colleagues, who may be defenders of the past. We almost want to say promote "best practices," but this is an overloaded term that probably captures unrealistic ideals. Perhaps we should be content with "decent practices." Fortunately, your newer colleagues probably have already bought into better practices, and the old ways are being unlearned by corporate inertia. Work to help them maintain their respect for the contemporary.

Finally, of course, the real enemy of ignorance is

enlightenment. Continue to learn and adapt practices to integrate the best of the past, present, and future. Q

REFERENCES

1. Software Requirements Practices Questionnaire: see http://www.personal.psu.edu/staff/c/j/cjn6/survey.html.
2. Neill, C. J., and Laplante, P. A. Requirements engineering: the state of the practice. *IEEE Software* 20, 6 (Nov./Dec. 2003), 40–45; http://csdl.computer.org/comp/mags/so/2003/06/s6040abs.htm.
3. See reference 2.
4. Royce, W. W. Managing the development of large software systems. *Proceedings of IEEE WESCON* (Nov. 1970). Reprinted in *Proceedings of the 9th International Conference on Software Engineering* (1987), 328-338.
5. See reference 2.
6. Brooks, F. *The Mythical Man-Month*, 2nd Edition. Addison-Wesley, New York: NY, 1995.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

PHILLIP A. LAPLANTE, Ph.D., is associate professor of software engineering at the Penn State Great Valley School of Graduate Studies. His research interests include realtime and embedded systems, image processing, and software requirements engineering. He has written numerous articles and 17 books, has cofounded *Real-Time Imaging*, and edits the CRC Press Series on image processing. Laplante received his B.S. in computer science, M.Eng. in electrical engineering, and Ph.D. in computer science from Stevens Institute of Technology—and an M.B.A. from the University of Colorado. He is a senior member of the IEEE, a member of ACM and the International Society for Optical Engineering (SPIE), and a registered professional engineer in Pennsylvania.

COLIN J. NEILL is assistant professor of software engineering at the University of Pennsylvania. His areas of expertise include object-oriented analysis and design, realtime systems design, and telecommunications. He has a B. Eng. in electrical engineering, M.S. in communications systems, and a Ph.D. in realtime systems design, all from the University of Wales in Swansea, U.K.