Agile Methods and Software Maintenance

by Dr. David F. Rico, PMP, CSM

Agile Methods only apply to the software "development" portion of the lifecycle and certainly don't apply to the software maintenance portion of the lifecycle, right? Gosh, how many times have we heard this cry? Every time a new software development paradigm emerges, software maintainers immediately emerge from the woodwork to complain that they're being ignored again. And, software maintainers have even gotten a little snobby too. A quick glance at the proceedings from some of the latest international software maintenance conferences reveals not a single paper on Agile Methods. Sort of a tit-for-tat, so to speak; you ignore me, I'll ignore you.

What's so special about software maintenance? Software maintainers even have their own standards for software maintenance (Institute for Electrical and Electronics Engineers, 1998). IEEE-STD-1219 is even replete with its own software lifecycle: (1) problem identification, (2) analysis, (3) design, (4) implementation, (5) system test, (6) acceptance test, and (7) delivery. Wait a minute, isn't that the waterfall lifecycle? Isn't, in fact, IEEE-STD-1219 just another Traditional Method? IEEE-STD-1219 even recommends the use of 41 different IEEE standards to document the software maintenance lifecycle! Huh?

Didn't other Traditional Methods like MIL-STD-1521B, DoD-STD-2167A, MIL-STD-498, ISO/IEC 12207, SW-CMM, CMMI, ISO 9001, and PMBoK do the same thing? That is, advocate a waterfall lifecycle and hundreds of software documents to promote software quality and maintenance at a cost of millions of dollars in documentation costs alone? Isn't this why the advocates of Agile Methods created a software development revolution in 2001 with the Agile Manifesto? That is, Agile Methods asked for: (1) individuals and interactions over processes and tools, (2) working software over comprehensive documentation, (3) customer collaboration over contract negotiation, and (4) responding to change over following a plan. But, now we've come full circle haven't we? That is, software maintainers say software developers don't address their needs and Agile Methods say software maintainers don't address their needs. It sounds a little like a chicken-and-egg thing going there, or a dog chasing its tail.

Furthermore, some people are attempting to superimpose a classical waterfall software lifecycle upon software source code created by Agile Methods such as Open Source Software Development (Koponen & Hotti, 2005). Their solution is to superimpose the waterfall lifecycle on Open Source Software: (1) process implementation, (2) problem and modification analysis, (3) modification implementation, (4) maintenance review/acceptance, (5) migration, and (6) retirement (Institute for Electrical and Electronics Engineers, 2004). How soon we've forgotten that Open Source Software is produced using: (1) individuals and interactions, (2) working software, (3) customer collaboration, and (4) responding to change. And, Open Source Software is replete "with user driven, just-in-time documentation" (Berglund & Priestley, 2001), "documentation in the form of code comments preferable by software maintainers" (de Souza, Anquetil, & de Oliveira, 2005, 2007), and "documentation that improves software quality" (Prechelt, Unger-Lamprecht, Philippsen, & Tichy, 2002). So, why would we want to "ruin" Open Source Software Development with waterfall-driven Traditional Methods?

In fact, the proponents of Agile Methods say the exact opposite. They say that Agile Methods are better suited for software maintenance than Traditional Methods. Why is that? Well, proponents of Agile Methods believe that software based on non-Traditional Methods promotes customer collaboration to elicit proper software maintenance needs, production of frequent software releases to deliver capability to maintenance customers sooner, collaboration within software development teams to leverage contextually rich maintenance communications, and the flexibility to respond to rapidly changing maintenance customer needs.

One such example was the use of Extreme Programming (XP), a popular Agile Method, for software maintenance instead of waterfall-based Traditional Methods (Poole & Huisman, 2001; Poole, Murphy, Huisman, & Higgins, 2001). In summary, use of Extreme Programming (XP) as a software maintenance model helped: (1) use refactoring to simplify and software source code by over 40%, (2) create and enforce coding guidelines proven to improve software quality instead of burdensome U.S. DoD standards, (3) fully automate software testing and daily-build processes, (4) reduce the Traditional Methods software maintenance staff size by 40%, and (5) simultaneously improve software quality by 67%. Here's a full list of the benefits of using Extreme Programming (XP) for software maintenance instead of Traditional Methods:

- Reduces code complexity by stripping out unused code.
- Implements patterns that make it easier to maintain, test, and understand the code.
- Reduces code size by over 40 percent.
- Promotes weekly presentations to merge strategies, patterns, and estimates.
- Institutes code reviews.
- Enforces adherence to source-code management guidelines.
- Promotes ownership of, and responsibility for code style.
- Promotes constant improvement of test coverage quality.
- Provides a proactive approach to problem solving.
- Fully automates the build and test process.
- Builds and unit tests the complete product set on a nightly basis.
- Reduces staff from 70 to 25 engineers (while increasing productivity 3 times).
- Provides a well-defined set of common rules that govern how to merge fixes and enhancements.
- Eliminates code complexity and stagnation.
- Results in a clean, well-structured code base that conforms to code standards.
- Provides processes to request, describe, prioritize, and implement incremental enhancements.
- Applies user stories to request bug fixes.
- Enables a 67% defect reduction.

Another study of Extreme Programming (XP) as a software maintenance process, showed that: (1) XP's coding standard, continuous testing, and collective code ownership were most valuable; (2) continuous testing, pair programming, and collective code ownership were the most culturally challenging practices to adopt, and (3) software maintenance productivity with XP increased over three times (Svensson & Host, 2005). In yet another study, the use of Extreme Programming (XP) showed that refactoring improved the quality and stability of two Java computer programs (Alshayeb & Li, 2005). As a side note, in a software maintenance study of a

200 million line of code C++ system: (1) there was less than a 4% probability that a one-line change introduced a fault in the code, (2) 10% of all changes made during maintenance were one-line changes, (3) 50% of the changes were small changes, and (4) deletions of up to 10 lines did not cause faults (Purushothaman & Perry, 2005). This study is significant, because the type of the changes it is citing is characteristic of refactoring found in Extreme Programming.

Open Source Software Development Methods are simultaneously Agile Methods and software maintenance processes. Therefore, they need no separate waterfall-driven Traditional Methods for software maintenance. One study showed that software quality was up to 20% higher in a study of six open source software products representing six million lines of code (Samoladas, Stamelos, Angelis, & Oikonomou, 2004). A study of two open source software products and four commercial projects showed that open source software projects had over five times fewer defects (Dinh-Trong & Bieman, 2005). In a study of 53 open source software products totaling 16 million lines of code, the proportion of major to minor contributors did not increase with size, complexity, or number of changes (Scotto, Sillitti, & Succi, 2007). Finally, in a study of 75 open source projects, the use of the Open Source Software Development paradigm not only increased software quality, but did not increase software development effort or cost (Capra, Francalanci, & Merlo, 2008).

One study of 130 software maintainers showed that the two documents used most often during software maintenance were the software source code itself and the comments they contained, and software architecture documents were considered the least important among software maintainers (de Souza, Anquetil, & de Oliveira, 2005, 2007). Another study of 96 programmers found embedded design patterns in software source code improve code quality and reduce software maintenance time, rather than those captured in separate documents (Prechelt, Unger-Lamprecht, Philippsen, & Tichy, 2002). Finally, a study of 78 software maintenance personnel indicated that the presence of UML documents did not improve the speed and productivity of software maintenance tasks and the costs of producing UML documents outweighed the benefits of producing them for software maintenance tasks (Arisholm, Briand, Hove, & Labiche, 2006).

So, what have we demonstrated here? We've demonstrated that Agile Methods can be applied to software maintenance. We've also demonstrated that Agile Methods may be superior to software lifecycles based waterfall-driven Traditional Methods (even those advocated by contemporary software engineering standards). Furthermore, we've demonstrated that the use of Agile Methods may decrease software maintenance costs by over three times, improve productivity by over three times, and increase software quality by up to 67%. We've demonstrated that Agile Methods result in the type of software documentation most often used and preferred by software maintainers (which consequently improves software quality). And, we've demonstrated that traditional software documentation doesn't improve software quality as much as the type of software documentation produced by Agile Methods. For Open Source Software Development, another form of Agile Methods, we've shown that it is not necessary to overlay waterfall-driven Traditional Methods on top of them, because continuing to use them improves software quality by up to five times without increasing software development effort and cost. Furthermore, Open Source Software Development practices are scalable to Open Source Software Projects of all shapes and sizes without requiring the burdensome governance mechanisms associated with the use of waterfall-driven Traditional Methods.

REFERENCES

Alshayeb, M., & Li, W. (2005). An empirical study of system design instability metric and design evolution in an agile software process. *Journal of Systems and Software, 74*(3), 269–274.

Arisholm, E., Briand, L. C., Hove, S. E., & Labiche, Y. (2006). The impact of uml documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering, 32*(6), 365-381.

Berglund, E., & Priestley, M. (2001). Open-source documentation: In search of user-driven, just-in-time writing. *Proceedings of the 19th Annual International Conference on Computer Documentation, Sante Fe, New Mexico, USA*, 132-141.

Capra, E., Francalanci, C., & Merlo, F. (2008). An empirical study on the relationship among software design quality, development effort, and governance in open source projects. *IEEE Transactions on Software Engineering*, In-press.

de Souza, S. C., Anquetil, N., & de Oliveira, K. M. (2005). A study of the documentation essential to software maintenance. *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting and Designing for Pervasive Information (SIGDOC 2005), Coventry, UK*, 68-75.

de Souza, S. C., Anquetil, N., & de Oliveira, K. M. (2007). Which documentation for software maintenance? *Journal of the Brazilian Computer Society, 13*(2), 31-44.

Dinh-Trong, T. T., & Bieman, J. M. (2005). The freebsd project: A replication case study of open source development. *IEEE Transactions on Software Engineering, 31*(6), 481-494. Documentation, Sante Fe, New Mexico, USA, 132-141.

Institute for Electrical and Electronics Engineers. (1998). *IEEE standard for software maintenance* (IEEE-STD-1219). Piscataway, NJ: Author.

Institute for Electrical and Electronics Engineers. (2004). *IEEE guide to the software engineering body of knowledge* (2004 Edition). Los Alamitos, CA: Author.

Koponen, T., & Hotti, V. (2005). Open source software maintenance process framework. *ACM SIGSOFT Software Engineering Notes, 30*(4), 1-5.

Poole, C. J., & Huisman, J. W. (2001). Using extreme programming in a maintenance environment. *IEEE Software, 18*(6), 42-50.

Poole, C. J., Murphy, T., Huisman, J. W., & Higgins, A. (2001). Extreme maintenance. *Proceedings of the 17th IEEE International Conference on Software Maintenance (ICSM 2001), Florence, Italy*, 301-309.

Prechelt, L., Unger-Lamprecht, B., Philippsen, M., & Tichy, W. F. (2002). Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering, 28*(6), 595-606.

Purushothaman, R., & Perry, D. E. (2005). Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering, 31*(6), 511-526.

Samoladas, I., Stamelos, I., Angelis, L., & Oikonomou, A. (2004). Open source software development should strive for even greater code maintainability. *Communications of the ACM, 47*(10), 83-87.

Scotto, M., Sillitti, A., & Succi, G. (2007). An empirical analysis of the open source development process based on mining of source code repositories. *International Journal of Software Engineering and Knowledge Engineering, 17*(2), 231-247.

Svensson, H., & Host, M. (2005). Introducing an agile process in a software maintenance and evolution organization. *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR 2005), Manchester, UK*, 256-264.