Towards High Quality Software Development with Extreme Programming Methodology: Practices from Real Software Projects

Bin Xu
College of Computer Science & Information Engineering,
Zhejiang Gongshang University,
Hangzhou 310035, China
xubin@mail.zjgsu.edu.cn

***Abstract*—Software quality is essential to software development projects, from the aspects of the customer, sponsor, development team and quality assurance team. Though there are kinds of test tools which help us to find the defect more efficiently, many researchers agreed that it is not enough nor efficient to validate the software product after it is produced. At the other hand, some agile software processes such as Extreme Programming enable us to produce high quality product but with too much different requirement to the project management. This paper introduces Extreme Programming (XP) practices in some real software projects, which can be helpful for project stakeholders to manage the quality efficiently and with less risk.**

Keywords-quality assurance; Extreme Programming; software process tailoring

I. INTRODUCTION

Quality is no doubt the most important part in software projects. With the help from requirement analyst, the customer defines the requirement with the specification for functionality, performance, availability, portability, and so on. While the development teams are developing the system according to the requirement, programmers conduct frequently code review, unit test to make sure if the system matches the specification. The testers act as the customers to verify the final product match the requirement.

Therefore, quality becomes the core of software engineering domain, and it was focused by available software life cycle models and CMM/CMMI [8]. Though water fall model was normally used, people found the feedback between the involved groups is not enough, and water fall model is not efficient to deal with the frequently changing requirements from customer. The project risks cannot be easily migrated until the late of the project. As a result, software product is developed with the involving, incremental life cycle, agile methodologies or Rational Unified Process (RUP) [10]. Extreme Programming methodology (XP) [9] is one of the favorite life cycle models being used to realize frequently changing customer request.

With the experience in using XP in multiple software project [1, 2, 3, 6], the authors find the practices of XP may also facilitate other non-XP projects, and is helpful for them to produce high quality product. In this paper, the authors list how the best practice of XP benefit the product quality.

The rest of the paper is organized as following, Section 2 views Extreme Programming practices from the quality management perspective. Typically Extreme Programming practices are adopted in those agile projects, Section 3 presents how to enable Extreme Programming practices into non-Extreme Programming project. Section 4 introduces some cases studies and the summary is given in Section 5.

II. PRACTICE FROM QUALITY MANAGEMENT PERSPECTIVE

Extreme Programming has been proposed by Kent Beck and Ron Jeffries [9, 11, and 12]. There are totally 12 practices listed within XP, namely,

- The Planning Game
- Small release
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair Programming
- Collective ownership
- Continuous integration
- 40-hour week
- On-site customer
- Coding standards.

The core practice impact product quality is testing, and it is supported by other practices: on-site customer, simple design, small releases, continuous integration, pair programming, and refactoring [9].

A. Testing

Testing practice of XP is the fundamental in agile testing. Unlike the other testing, the testers in XP projects have the confidence that the product will work, and the program gets more and more confident. They perform the testing to ensure the product works. They never complain that the product is of bad quality, on the contrary, they work with the developers to

978-1-4244-4639-1/09/$25.00 ©2009 IEEE

find out and fix the possible defects so as to make the system work. Therefore, the communication between testers and developers is very good, as well as the feedback between them. In some projects, the authors observed that testers always worked as pairs with the developers. They enjoyed the quick defect finding and fixing without any communication delay.

The tests written in XP project ought to be isolated as much as possible and should be automatic. With isolated tests, the testers or developers may find the defects immediately and directly. They don't need to find the defects out from a pile of output messages.

B. *Simple design*

Beck in his book has listed 4 rules towards simple design:

1. Runs all the tests,
2. Has no duplicated logic,
3. States every intention important to the programmers,
4. Have the fewest possible classes and methods.

However, there is some difference voice towards this practice.

C. *On-site customer*

XP suggests a real, live user on the team, available to answer the questions, resolve disputes, and set small scale priorities all the time. The on-site customer may act as the bridge or agent between the other customers and programmers. The customer manages the project and leads it to the success.

D. *Pair programming*

Pair programming is a important feature of XP, though there are many arguments from many software engineering sages towards it. Much research work has been conducted to evaluate this practice, which showed that the pair works is more efficient than being individual.

E. *Refactoring*

Duplicate code impairs the maintenance of the system. Refactoring is a way to keep the code in good form, remove the duplicate code, and make the code to be easily understood.

Since refactoring costs much in the projects, it can be performed on event when some new tools or technologies are introduced, or some new code has been implemented.

F. *Small release*

One weakness of water fall life cycle model is the risk cannot be settled down until the delivery. The practice "small release" shortens release cycle to speed the feedback from the customer so as to migrate the risks release by release.

One release should contain the most valuable business requirements and must make sense as a whole, which can then be valued by the customer. The normal size of a release is around a month or two.

G. *Continuous integration*

When finding defect or validating a code, the complexity of task increases by the size of the code. Continuous integration reduces the size of different parts between the conjoint integrations, and facilitates the defect finding and fixing.

Because the building requires the whole testing for the features, integration costs much time. Though it is suggested by Beck that daily building is the most period, it is performed normally weekly when the system is large.

III. ENABLING XP PRACTICES IN NON-XP PROJECTS

When adopting XP practices and activities in our case study, the authors always thought if these practice and activities can be used in non-XP projects and how. With the experience from these projects, the author stated their understanding and provided their proposal in the following sections.

A. *Test before coding*

Typically testing is performed after coding, the unit test will be conducted by programmers during the coding of the functionality. After then, integration testing is performed to ensure the new implemented or changed components that are fit in the whole system. Finally, system testing such as functional test, performance test, recovery test will be conducted to ensure that the final product match the need of customer. The integration test and system test are designed and performed by QA person.

Different from typically cases, XP methodology suggests "test before coding" which is a breakthrough in the field. The authors evaluated this proposal in a reengineering project and a new development project. The results were not the same. In the reengineering project, there was an original legacy system in production. The team built a comparison test case to test if the target part matched the requirement of the original part. The new development project involved a GUI interface to be developed. Without the pre-build GUI interface, the test case was not easy to be implemented before the coding. However, both projects found it useful to design the test cases before coding. For example, the requirement was analyzed and decomposed into testable parts. The vague requirement was identified and made clearly at the very beginning.

As a result, the authors argued that the team may design the test case before coding, and try to implement the test case if possible.

B. *Pair Programming*

There are lots arguments about "pair programming". This practice is blamed to be an ideal idea to improve a little quality using extra much more resource. In our case study, the authors found the dynamic and partly pair programming will gain more than working individual. The QA people and programmers sat together when there were critical defects or they needed immediate feedback. The customer sat with QA people or programmer to provide immediate explanation about the requirement or ensure that some key features had been completely understood. The programmers were not working as pairs all the time. They might code individual for some simple

part, and they would ask for the help from some other programmer if they need especial business knowledge or technical support, they worked as pair at the moment. The authors also found that the team might work in group to find a defect which was deeply hiding in the code or to fix a defect with appropriate way.

In other word, pair programming worked when the programmer, QA person need help from the team, or when they need immediate feedback. It was a consciousness of teamwork in our case study. The team with such consciousness was more willing to communicate with others and enjoyed the work.

Therefore, the authors would suggest the practitioners to value the "pair programming", and work in pair or group whenever they need. Pairing with the assignment of manager was not the better.

C. *Automatic test*

Automatic test is important for "continuous integration" practice in XP projects. As the test cases might be run multiple times daily, any manual test involving will rise the test duration and incur large resource cost to finish so many tests.

However, automatic test need more time and efforts to design, implement and maintain. Sometimes it is difficult to design automatic test when there is some third part systems involving. When the automatic test required too much time, such test could be run manually at the first and may be improved later. A practical method for the practitioners is to perform continuous integration with the entire automatic test, but leave the manual test to the daily or even weekly building. Of course as much as possible test cases should be made automatically.

Because the preparation time for test automation is up to 2/3 times longer than for manual testing, it was said that the benefit of the test automation only begins to occur after approximately the third time the tests have been run. When the requirement must be changed frequently, there is a high overhead to update the automated test scripts. To reduce such rework, the test case may be developed after certain feature is fixed to some degree [1].

D. *Simple design*

It's difficult to have simple design while still matching the requirement. In our case study, the authors found the team kept in mind not to design beyond the requirement, however, they didn't care about whether the design is simple or not. The refactoring may be done if the design was found to be hard to understand.

Since there was no extra feature designed by the team, the side effect was released. It's easy for the testing and possible change in the future.

E. *On-site customer*

On-site customers have the ability to remove the gap between customer and programmers. They also help to build same vision between customer and programmers. The team having on site customer is most likely to be success.

However, having customer on site needs much financial support. When there is no on-site customer, the channel between businessmen and development team should also be made efficiently. The communication and coordination should be well arranged to achieve that [4, 5]. The authors provided some suggestions to facilitate the communication and reduce the gap between business side and technical side [2, 6].

When the customers couldn't be attached by development team, BA network maybe well established.

F. *Small release*

Small release is helpful to reduce some risks related with customer side. With 1~2 months' small release, the customer can monitor the project and provide the feedback in time. This practice may also be used in the cooperation between a new customer and a new software development team at the first time. The customer may find out the quality issue of the product or process at the very beginning, and manage the risks after that. The whole system may involve release by release.

IV. CASE STUDIES

A. *Process tailoring with Extreme Programming process*

A financial system was to be migrated from DEC-Alpha to AIX system. There was a 12 member development team in China, which included 6 reverse-engineering members and 6 forward-reengineering members. In USA, 4 members assumed onsite customer role for the requirement confirmation and code acceptance.

Besides communication tools such as email, instant message meeting, and teleconference which were used in the communication between the customers and the offshore develop team, XP practices were implemented to reduce the communication delay and improve communication quality as described in. The requirements were communicated in pieces, the development was conducted in small iteration, an organized knowledge database was set up on a website, accessible to developers and customers, to record all the knowledge generated, and a problem tracking tool were developed to record the communication as well. Only one person from Boston paid a two-week visit to kick off the project. The project was completed without major communication issues within 6 months as originally estimated, and with a cost-saving of at least 60% compared with doing the project entirely onshore.

Though this system migration project might not be viewed as an Extreme Programming project, the adopted Extreme Programming practices benefited the project very much.

B. *Quantitative Analysis of XP Practice*

In the first phase of the reengineering project, a representative and critical module was chosen which had 77.5KLOC source code and depended on a third party library.

Extreme Programming practices were adopted in the phase. Six analysts, who formed the reverse engineering group, were assigned to implement architectural spike and acquired the System Metaphor. Then, four more developers, who formed the

forward engineering group, joined the project. Reverse engineering group generated 22 stories with the basis of the System Metaphor and allocated these stories to forward engineering group. The reverse engineering and forward engineering were conducted in all these stories iteratively.

Develop groups worked out 50.2KLOC code in 7 weeks. Finally, they successfully got the executable application with the performance twice as that of the original one. The related project management data was collected and compared with other reengineering projects which were conducted in the company. With the adoption of XP, there were significant improvements in terms of productivity, quality, and time to market.

1) Shorter Duration of the Project: According to the plan, the project should be finished in 8 weeks. As there was some dependency among the tasks, it will be postponed at least 3 weeks if the reverse engineering and forward engineering were run sequentially through the traditional mode. Multi reverse engineering tasks could be finished at the same time, and the forward engineering tasks were generated accordingly. In such way, the stories could be performed in parallel, which enabled us to employ more programmers into the project than traditional methodology. Four more programmers were added for the forward engineering and develop groups were able to finish the project in only 7 weeks, 1 week ahead of the deadline.

2) High Quality of the Source Code: With the successful refactoring and simple design, the size of the source code dropped from 77.5KLOC to 50.2KLOC for approximately 1/3 and the throughput turned out to be doubled. In the whole process of the pilot project, the develop QA team found 7 bugs in total and the customer QA team found 2 bugs after the delivery. All these bugs were located very quickly and 7 of them were fixed in 30 minutes for each while a memory leakage bug took 2 hours to fix. The one left was a crash in the third party library, which was fixed in the patch given by the provider 4 days later. Develop groups removed almost 77% of the defects with the enhancement of testing and enjoyed the quick bug fixing with the simple design and team work. This project has notable low bug rate which is only 13% of that of the other project.

3) High Productivity: The productivity of this project is 143LOC/per day, 1.5 times as other projects conducted before. Although there was 100 weeks as planned, the project was successfully finished with only 70 weeks. In the project, we did not find the pair could code in double speed as the individual. However, XP practices ensured the quality of the code, and saved much time in testing and bug fixing in this project.

4) Effective Collaboration: We didn't record the communication between forward engineering group and reverse engineering group due to its large volume, but we found that the communication between reverse engineering group and the remote customer was efficient and regular. Normally reverse engineering group got responses the next morning; occasionally they got the responses in 3 days or longer when the questions involved many departments of the customer. Dual-shore communication model reduced the communication delay between distributed teams to a tolerable level. It is proved that the communication model can be run efficiently in the whole pilot project.

V. SUMMARY

This paper viewed the XP practices from Quality Management perspective, and verified them in the case studies. With the experience of adopting XP in several distributed software development projects, the authors provided some proposal to modify the XP practice so as to use them in non-XP projects. Test before coding, pair programming, test automation, simple design, on-site customer and small release are suggested to be used in non-XP projects to improve their product quality.

The authors are now working on merging and tailoring the concurrent software engineering processes into a new process to be used in global distributed software engineering.

ACKNOWLEDGMENT

This research was financial supported by National Natural Science Foundation of China with No. 60873022 to Dr. Hua Hu, The Science and Technology Department of Zhejiang Province, China with No. 2008C11009 to Dr. Bin Xu and No. 2008C13082 to Dr. Xiaojun Li, and Education Department of Zhejiang Province with No. 20061085 to Dr. Bin Xu.

REFERENCES

[1] B. Xu, X. Yang, Z. He, S.R. Maddineni, "Extreme Programming in Reducing the Rework of Requirement Change", 2004 Canadian Conference on Electrical and Computer Engineering, Niagara Falls, Ontario,Canada, May 2004, pp. 1567-1570

[2] X. Yang, B. Xu, Z. He, S.R. Maddineni, "Extreme Programming in Global Software Development", 2004 Canadian Conference on Electrical and Computer Engineering, Niagara Falls, Ontario,Canada, May 2004, pp. 1845-1848

[3] B. Xu, X. Yang, Z. He, S.R. Maddineni, "Achieving High Quality in Outsourcing Reengineering Projects throughout Extreme Programming", Proceedings of International Conference on Systems, Man and Cybernetics, Hague, Netherlands, October 2004, pp. 2131-2136

[4] B. Xu, X. Yang, Z. He, A. MA, "Global Cooperative Design in Legacy System Reengineering Project", The Eighth International Conference on CSCW in Design, Xiamen, China, May 2004, pp. 483-486

[5] B. Xu, X. Yang, Z. He, A. Ma, "Enhancing Coordination in Global Cooperative Software Design", Proceedings of the 9th International Conference on CSCW in Design, Coventry, UK, May, 2005, pp.22-26

[6] B. Xu, "Extreme Programming for Distributed Legacy System Reengineering", The 29th Annual International Computer Software and Applications Conference, Edinburgh, Scotland, July, 2005, pp.160-165

[7] J.A. Whittaker. "What Is Software Testing? And Why Is It So Hard?," IEEE Software, vol. 17, no. 1, pp. 70-79, January/February 2000.

[8] M.C. Paulk, C.V. Weber, S.M. Garcia, M.B. Chrissis, and M.W. Bush. Key Practices of the Capability Maturity Model, Version 1.1, (CMU/SEI-93-TR-25, ADA 263432). Pittsburgh, PA:Software Engineering Institute, February 1993.

[9] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, Reading, Mass., 1999.

[10] P. Kruchten, The Rational Unified Process–An Introduction, Addison-Wesley-Longman, Reading, Ma, 1998.

[11] www.extremeprogramming.org

[12] www.xprogramming.com