Analysis of the Interaction between Practices for Introducing XP Effectively

Osamu Kobayashi
Software Research Associates, Inc.
Osaka Honmachi-nishi Daiichi-seimei Bldg. 5F, 2-1-1 Awaza Nishi-ku, Osaka 550-0011 Japan
+81-6-6536-2331
o-kobaya@sra.co.jp

Mitsuyoshi Kawabata
Agileware

2-20-34-405 Hanaten-nishi, Johtoh-ku, Osaka 536-0011 Japan

+81-6-4258-2115
kawabata@agileware.jp

Makoto Sakai
SRA Key Technology Laboratory, Inc.
Marusho Bldg. 5F
3-12 Yotsuya, Shinjiku-ku, Tokyo 160-0004 Japan

+81-3-3357-9011
sakai@sra.co.jp

Eddy Parkinson
Department of Computer Science, Osaka University
1-3, Machikaneyama-cho, Osaka, 560-8531 Japan

+81 (0) 9036218029
eddy@ist.osaka-u.ac.jp

ABSTRACT

In this paper, we discuss interactions between XP (eXtreme Programming) practices. We discuss 2 case studies of introducing XP practices selectively from the 13 practices which are defined in XP, and we analyze how to select practices. Our analysis is based on interviews with developers. While it is difficult to introduce all the XP practices at once, our knowledge makes it easier to determine more effective combinations of practices.

Categories and Subject Descriptors

K.6.3 [**Software Management**]: Software process

General Terms

Management.

Keywords

extreme programming, software process, software engineering, metrics.

1. INTRODUCTION

XP (eXtreme programming), is an agile software development method which can quickly adapt to changes, it is gaining more attention [1]. Traditional methods based on waterfall model make large scale development easier by freezing specifications after defining them or strictly controlling changes to specifications.

But freezing specifications prevents us from adapting to changes in user requirements and environments. When it seems difficult to freeze specifications it is necessary to use methods such as prototyping. In recent years, computers have become a lot faster and their functionalities have become more diverse, more types of

users and user-interfaces have appeared. These trend to accelerate the possibility of changes to requirements. In circumstances such as these, agile software development methods are gaining more attention.

Among agile software development methods, XP characteristically has "four values" and also defines several practices. This makes it easier to understand its guidance and methods of development and has promoted the acceptance of XP in Japan. XP emphasizes four values (communication, simple design, feedback, courage). Communication makes it easier for the development team to head towards its goal. Simple design reduces defects and makes maintenance activities easier. Feedback from testing increases quality. Lastly, courage allows big problems to be tackled in a more fundamental manner. XP defines many practices that emphasize these values.

When introducing XP into our projects, we must choose which practices to use. Ideally we would introduce all XP practices at once, but in reality we often cannot use some practices because of constraints from customers or lack of resources. In some cases, because of the characteristics of the project, certain practices are neither necessary nor effective. Studies have reported many cases which used practices selectively. [6,8,10,12]. But these studies only report the effects of a single practice, there have not been sufficient examination of the effects of combining different practices. If we can improve our understanding of interactions between practices, we can better decide which practices to use in which projects, so we can introduce XP much more effectively.

In this paper, we report two case studies of software development projects using XP, and knowledge about interactions between practices gained from interviews with developers. We also consider methods of selective introduction of XP practices based on our analysis.

2. XP(eXtreme Programming)
2.1 Practices of XP

There are strong relationships between XP and its practices. Without practices it's not XP, and without practicing the practices of XP it cannot deliver benefits. Four values which XP emphasizes are not concrete practices, but are essential ideas

designed to lead projects to success. Practices are concretizations of these ideas as methods to be practiced in real projects. To solve problems of projects from all viewpoints, Kent Beck selected 12 practices [2], and Ron Jeffries developed them further and has 13 practices [7]. In this paper, we investigated effects of interactions between the following 13 practices.

Whole team

All people who take part in the project gather in one place to develop the system as a team.

Planning Game

Developers and customers make plans for software releases and iterations together, identifying each role clearly. Specifically, this involves the making of story cards from each user's point of view and splitting each story into task cards for individual developers. Based on these cards, they make plans that take into consideration the volume of work and the schedule.

Small Releases

By creating several small releases, usually each one or two months, developers get feedback from users. The development periods between releases are called "iterations".

Customer Tests

Users define test cases for system releases.

Simple Design

Developers implement just enough parts to satisfy requirements, and keep code simple. Always keep in mind the principle of YAGNI ("You aren't going to need it").

Pair Programming

Programmers always program in pairs sharing a computer.

Test Driven Developments

Programmers always create tests before implementing any functionality.

Refactoring

Make internal structures of programs better, without changing their behavior.

Continuous Integration

Integrate small units into the system which developers plan using a planning game and always keep the system running.

Collective Code Ownership

Code is owned not by some programmer but owned by the team collectively, and anyone can change any code at anytime.

Coding Standards

Make common rules to standardize coding styles in the team.

Metaphor

Using metaphor, developers in the team share understandings about how their programs work.

Sustainable Pace

Avoid overwork, work in sustainable pace.

If we cannot use some of the practices, then the gap should be filled using an alternative approach [12]. For example, pair programming has the effect of improving the quality of code, like code reviews, so if we cannot use pair programming for some reason, the practice can be replaced with code reviews. On the other hand, if only one quality control practice is used, for example pair programming, then code quality suffers. For effective code quality, we need other practices, like test driven development (TDD) as well. Without having tests and making sure the program passes them it is difficult to keep quality high. But even when TDD is used, developers who have little experience at writing tests can easily overlook important tests. We need reviews for tests too, and pair programming has a similar effect to reviews. As such, a lack of experience can be addressed by combining several practices. Thus, practices interact with each other, and this makes them more effective.

2.2 Case study of XP introduction

When introducing XP into projects, it is difficult to use all the XP practices. So it is important to choose appropriate practices. Several issues regarding XP introduction strategies and compromises have been published. Articles [6] and [12] show some ideas about how to use practices that to apply XP. In article [10], they apply XP practices in phases. In article [8], they show the effect of each practice and which practices were difficult to apply.

In [6], they report on experiences of XP introduction in a large organization in which software development processes were strictly defined. In this case study, they tried to apply XP to only one subsystem of a large safety critical system. They tried to use all the XP practices, but recognizing the characteristic of the system and policy of the organization, they used use case diagrams in planning games and prepared minimal necessary documents (but not only to conform to organization's policy. They prepared only useful ones.)

In [12], they report about a very small scientific research project which had only 2 developers. First, they show which factors make it difficult to apply XP to research projects, then they describe how to overcome these factors by changing some practices. For example, in planning game and simple design, they took the roles of customers themselves instead of real customers. They also show pair programming and collective code ownership increased productivity and readability of code.

In [10], they report about the application of XP when their project recovered from a big problem and moved to stable maintenance phase. In this case study, some members joined the project half way through to help it recover, then they introduced XP's practices in some phases after the first release, to create steady maintenance environment. Specifically, after creating a maintenance environment, they applied these practices;

Continuous Integration

(Modest) Refactoring

Simple Design

Coding standards

Standup meetings

After the project became stable, they examined some other practices.

In [8] there is a description of a case study of applying XP to a project involving the rewriting of a legacy system using new technology based on Java. In the report, they say that pair programming had the most impact and it helped when applying simple design, test driven development, and refactoring. Pair programming was especially important when used with test driven development. The report also describes applications and the effects of all practices. They mention communications between external teams, the whole team and defining the role of testers as difficult but important points.

Thus, when we introduce XP, which practices are used and how, has big influence on XP, because of this there is a need to analyze case studies carefully. It is important to consider interactions between practices, especially because practices are not independent of each other. In [6] and [12], each practice is used independently and interactions between practices are not examined. In [10], they show an example of introducing XP in phases, but they do not show interactions between practices clearly, so we cannot find in this case clues of how to introduce XP in different situations. In [8], they report that pair programming has a positive effect on applying other practices, but only a few of interactions between practices are considered. Thus, there has not been enough analysis of how to choose which practices to use when XP is introduced into projects.

3. RELATIONSHIP BETWEEN PRACTICES BASED ON CASE STUDIES
3.1 Case 1: building online shopping site
3.1.1 An overview of the Project
In this case study, we introduced XP into the development of a system for analyzing workload of Nissen's online shopping site. (Table.1) This system analyzes the behavior of customers who visited the shopping site, such as when, on which site, what kind of goods, how much they bought.

Table 2 shows practices we used in this case. When the project began using XP, they had almost completed gathering requirements, so we applied XP from the next phase.

3.1.2 Details of the project
First and second iterations

In the first iteration, stories of the system were rather simple, so we omitted distinct design activity and began programming using TDD. But because we were not used to TDD and we gave short term schedules higher priority, we could not do enough refactoring. As a result, the program code became complex and we could not use it in the second iteration. We did a lot of refactoring in the second iteration throwing away almost all the code from the first iteration. This shows that even for simple stories it is necessary to do some modeling of functionalities. If we had done modeling like CRC sessions [4], we would not have

Table 1. An overview of the Project of case 1

Attribute	Value
Member	Manager, Developer x 3, User x 2
Term	5 months (XP: 3 months)
Language	C#, stored procedure
Number of release	3
Number of iteration	4

Table 2. Used Practices of case 1

Practice	Application
Whole team	Partial
Planning Game	Partial
Small Releases	Full
Customer Tests	Full
Simple Design	Full
Pair Programming	Full
Test Driven Developments	Full
Refactoring	Full
Continuous Integration	Full
Collective Code Ownership	Full
Coding Standards	Full
Metaphor	No
Sustainable Pace	Full

needed to do such a large amount of refatoring. And if we had done firm refactoring, program code would have been reusable and would have not been thrown away.

We always practiced pair programming, but could not keep going at a sustainable pace. Because of schedule delay we had to work at an unsustainable pace. After a week of overwork, both member of the programming pair began to arrive late for work, and said that they could not think well in the morning. It was apparent that overworking did not recover the delay, so we switch to using a sustainable pace of 40 hours a week. When we developed using a sustainable pace, productivity improved.

Third and fourth iterations

After doing agile designs by all members, the project progressed rather smoothly. There were three developers, so in the first two iterations, two of them did pair programming and the other worked alone. But in the third iteration it became apparent that solo programming introduced bugs and misunderstandings of specifications. So we did "triplet programming [11]" for rather

complex stories, and after this few misunderstandings were experienced.

Pair programming, test driven development and refactoring had a very good synergy allowing code to be kept simple. With simple code, we could implement stories for the third and fourth iterations faster than we had estimated, had recovered the schedule delay.

One of developers was less skilled at refactoring, but by doing pair programming with another developer who was good at refactoring allowed the developer to gain the required refactoring skills. On the other hand, the skilled developer sometimes refactored too much. He violated the principle of YAGNI and added too much flexibility. In such cases, the less skilled developer took an important role, creating a balance between the two extremes. Using pair programming was an effective method of doing test driven development. In test driven development, we can improve software quality by creating only test cases which might fail [3]. When developers programmed solely, they tend to miss some test cases or make too many test cases. In pair programming, they could always check the quality of test cases. Pair programming promoted the standardization of program code naturally without documented coding standards.

Test driven development enabled continuous integration. By providing test cases beforehand, we could find problems and analyze causes during integrations. And by refactoring we replaced conditional statements with polymorphism. This reduced the number of pre-conditions and test cases and kept the cost of test driven development low.

Our developer's impressions

- We should have used rapid modeling before the first iteration, because we had to do major refactoring after the first iteration. (XP suggests CRC sessions for modeling, but does not force it as a practice.)

- We need another new practice "modeling together".

- We found that pair programming, test driven development, refactoring, continuous integration and coding standards are firmly interrelated.

- To practice pair programming, a steady sustainable pace is needed.

- Standup meetings [2] are very effective (Throughout our project, we had a standup meetings [2] everyday, checking the project's condition with story cards and task cards tapped on the wall, so we could grasp the progress and problems of the project without project management documents).

3.2 Case study 2. Customization of shrink-wrapped cost estimation software

3.2.1 An overview of the overview

This case study is about a project to customize shrink-wrapped cost estimation software. (Table 3) The size of the software itself was large, but only one-fifth of it required customization. Table 4 shows practices we applied to the project.

Table 3. An overview of the Project of case 2

Attribute	Value
Member	Manager, Requirements Developer x 2, Developer x 4, User x 1
Term	3 months (XP: 3 months)
Language	Java, Struts Framework
Number of release	2
Number of iteration	3

Table 4. Used Practices of case 2

Practice	Application
Whole team	Partial
Planning Game	Full
Small Releases	Full
Customer Tests	Full
Simple Design	No
Pair Programming	No
Test Driven Developments	No
Refactoring	Partial
Continuous Integration	Full
Collective Code Ownership	Full
Coding Standards	No
Metaphor	No
Sustainable Pace	No

Table 5. Number of tasks and work time in 1st iteration

Number of tasks in 1st iteration / in all iterations	98/192
Actual hours in 1st iteration / in all iteration	149h/374h

3.2.2 Details of the project
The first iteration.

We began to estimate requirements of customizations, because estimations of customization varies a great deal depending on the customized software, at first we analyzed the software itself and found that the style was not that of normal object-oriented code. There were large procedures that exceed 2000 lines, and it seemed to be inevitable that customizing them would affect existing functions within the software. So we decided to refactor the software first. We had to refactor a large volume of code including code that contained functions not allocated to our project. This made collective code ownership a necessary practice,

because without it we did not have the confidence needed to refactor. We made a rule for collective code ownership that we should synchronize the code at least once a day. We thought that if we did not synchronize, after about three days the number of conflicts would become excessively large and result in time being wasted. For classes which we shared on a regular bases we found we could avoid conflicts by synchronizing the code just before making changes.
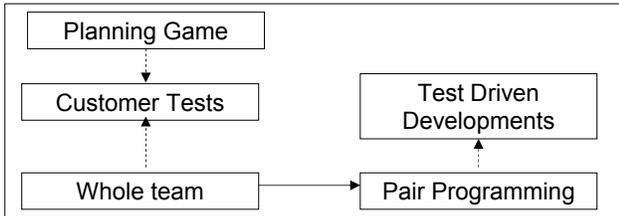

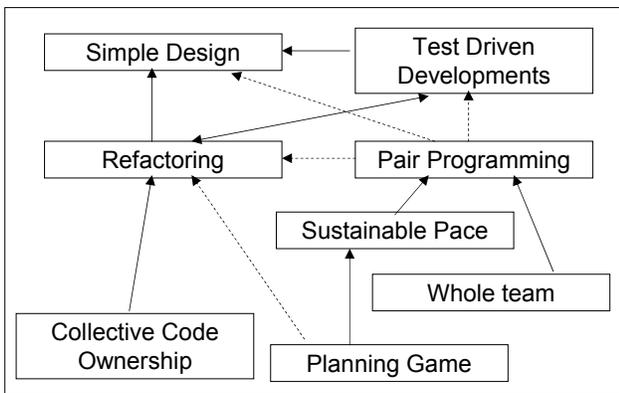**Figure 1. Communication reduced cost of documentation**
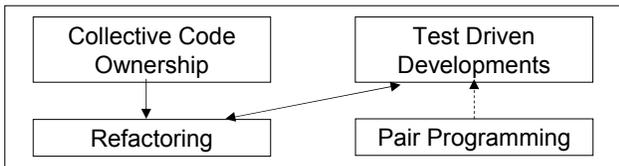

Figure 2. Improvement of productivity

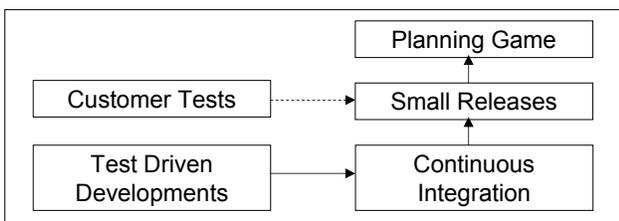**Figure 3. Improvement of the quality of program code**

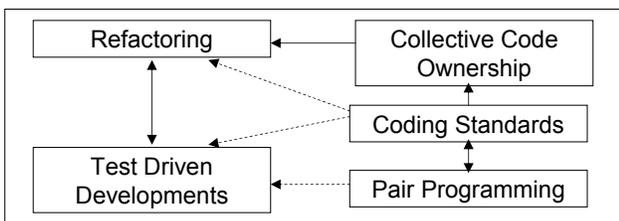**Figure 4. Improvement to the quality of requirements**

Figure 5. Improvements that simplify maintenance

The second and third iterations

Refactoring during the first iteration had a good effect. During the second and third iterations, there were additional changes to requirements of functions implemented during the first iteration, and feedback from end users about the first release of the software. Table 5 shows the number of and actual hours of work for tasks on the second and the third iteration, which were related to tasks of the first iteration. This shows that more than the half tasks of the second and third iterations ware related to the first iteration, but hours of worked on then was less than 40 %. Tasks which involved refactored code took less hours and refactoring during the first iteration took 22 hours, so we can say that refactoring had actually improved productivity.

Our developer's impressions.

- We needed collective code ownership to make refactoring successful. (To implement collective code ownership we used CVS as a version-control tool. One characteristic of CVS is that we can change the same program code simultaneously without locking it. In XP projects there is frequent refactoring, so this characteristic of CVS is highly suitable for XP.)

- When refactoring we made use of "Method extraction [5]" several times, this improved our productivity. (During the first iteration we made heavy use of refactorings, but this still totaled less than 50 % of code. This is because if we refactored excessively it would damage the framework of the software. When refactoring the code, we could foresee requirements of the second and the third iterations to some extent, so we refactored only code which seemed to be related to those requirements.)

3.3 Interactions between practices and their effects

We interviewed the developers of case study 1 and 2 about what we said in section 3.2, and we found that the introduction of XP had following 5 effects and there are dependencies between these practices of which some are strong and some are weak. So we defined a notation for these dependencies and interviewed them again. Figures 1 to 5 show the results of these interviews. In these figures, we categorized dependencies between practices according to their effects. We denote each practice as a rectangle and denote each dependency as an arrow. The meanings of different kinds of arrow and their directions are as follows,

<Relationships between practices>

Solid line arrows means "strong" dependency.

Dotted line arrows means "weak" dependency.

The practice which an arrow points to depends on the practice that the arrow comes out from.

(1) Communications reduced cost of documentations. (Figure 1.)

By using the practices of whole team, planning games, pair programming, customer tests and test driven development the three developers of case study 1 could share knowledge about requirements and design without documents.

To share knowledge with each other, developers have to change their partners in pair programming frequently. To achieve this all

Table 6. Interactions between practices and their effects

	Effects	communication	productivity	quality of code	quality of requirements	ease of maintenance	dependency	contribution	<->	←	→	<--	-->
Small Releases(SR)	1				v		1.5	1		CI	PG	CT	
Continuous Integration(CI)	1				v		1	1		TD	SR		
Customer Tests(CT)	2	v			v		1	0.5				WT PG	SR
Planning Game(PG)	3	v	v		v		1	1.5		SR	SP		CT
Simple Design(SD)	1		v				2.5	0		TD R		PP	
Sustainable Pace(SP)	1		v				1	1		PG	PP		
Coding Standards(CS)	1					v	1	2	PP		CC		R TD
Whole team(WT)	2	v	v				0	1.5			PP		CT
Refactoring(R)	3		v	v		v	3.5	2	TD	CC	SD	CS PP PG	
Collective Code Ownership(CC)	3		v	v		v	2.5	1		CS	R	CS	
Test Driven Developments(TD)	4	v	v	v		v	1.5	4	R		SD CC CI	PP	
Pair Programming(PP)	4	v	v	v		v	3	2.5	CS	SP WT			SD R TD

developers must gather on one site. In customer tests, users write many test documents, and these test cases show details of specifications, this means we can keep specification documents themselves simple.

(2) Improvement of productivity (Reduction of cost of changes by refactoring, Figure 2)

By using the practices of simple design, pair programming, test driven development, refactoring and collective code ownership, we could maximize the feedback of code implementation information and this minimized re-working and promoted reuse of components. This improved our productivity.

A sustainable development pace is needed for efficient pair programming. Sometimes we need to do large amount of refactoring to keep the design simple and well structured, because of this, the issue had to be considered during the planning games.

According to the policy of XP's simple design, we must keep the growing system simple. We achieved this by using test driven development and by refactoring to keep the code simple.

Sometimes refactoring reduced the productivity in the short term, but in the long run, it promoted reuse of components and improved productivity. In iterative development, we make changes and additions to existing code frequently, and it often takes a lot time to understand complex code. Changes and additions to simple code take much less time, refactoring generally improves productivity.

(3) Improvements in the quality of program code. (Figure 3)

By practicing test driven development, pair programming, refactoring, continuous integration and collective code ownership, we could keep the quality of the software satisfactory for users. In case study 1, after the first release, which consisted of 110 modules and 7,500 lines of code, only 3 defects were found.

In test driven development, programmers also create test cases, so it is important to review test cases during pair programming to reduce mistakes and help stop necessary test cases being missed. As a result pair programming is able to contribute to test driven development.

(4) Improvements in the quality of requirements (Figure 4.)

By practices of planning games, small releases and customer tests, we had many chances to reflect on users' requirements of systems in short iterations, this improved satisfactions of users.

In order to keep doing iterations with small releases, we had no time to do integration tests, so we need continuous integration. By doing customer tests in early iterations, we can get rapid feedbacks and find defects relating to the quality of requirements. These enabled us smooth out releases.

(5) Improvements of ease of maintenance (Figure 5.)

By practices of pair programming, collective code ownership, coding standards, test driven developments and refactoring, program code can be kept standardized and simplify maintenance.

Refactoring keep code readable and simple, but if everyone named their own program modules the chance of misunderstandings increases, so we needed coding standards. Collective code ownership also necessitates coding standards because we must change code written by others.

In pair programming, we write code adjusting partners according to what was needed. By continually changing partners, all development code gets standardized. By defining coding standards, we can avoid extra discussions about coding styles between drivers and navigators and do pair programming smoothly.

4. DISCUSSION

Table 6 shows interactions between practices and their effects which we discussed in section 3.3. The table contains "v" symbols and indicates which practices are related to which effects. The column titled "effects" contains the number of "v"s

for each practice. Columns whose titles contain arrow marks show dependencies between practices. The columns with arrows that point to the left contain the initials of practices. These initials are the practices that the left hand column practices depend on. The reverse is true for the columns with arrows that point right. Figures in the "dependency" column show the scores of "dependency" for each practice. We assigned 1 to each "strong" dependency, which is denoted with a solid line arrow and 0.5 for a "weak" one, which is denoted with a dotted line arrow, these are summed for each practice. Figures in the "contribution" column show scores of "contribution" for each practice. (We excluded practices which were not used in both case studies, because we could not gain knowledge about their interactions.)

Using this table, we can know which practices are necessary for which expected effect and also know the characteristics of each practice. For example, test driven development has a rather low score 1.5 for "dependency" and high score for "effects". This means that this practice is easy to introduce with a few other practices and also can be very effective. Both refactoring and pair programming have high score of 3 for dependency, so it might be difficult to introduce them alone, but refactoring has a high score for contribution and effects, so if we could introduce refactoring, the data suggests it would have a large positive effect.

Collective code ownership has a score of 3 for effect and a score of 1 for contribution. Pair programming has a score of 4 for effect and a score of 2.5 for contribution. Both of these practices have high effects and low contribution. This might mean that these practices have many merits by themselves and these merits are independent on other practices.

On the other side, the scores for effect in table 6 are determined by counting relationships to five categories of effects which were gathered from interviews with developers, so we must be careful how the scores are treated. Introducing practices which have high effect scores is in theory an efficient way or introducing XP, but it does not necessarily mean that in practice this will be effective for realizing all four XP values. Especially if we pay attentions to practices which improve quality, these have different characteristics from the practices that have other improvement effects. The table shows that when we try to improve quality there is no simultaneous improvement of other effects, so scores for effects are not so high. But when quality is a major business goal, practices that improve quality need to be applied.

5. CONCLUSIONS

We described two case studies that apply practices of XP to real projects and summarized interactions between the practices based on interviews with developers. And we considered the effect of selective application of XP practices. Using the knowledge gained, it is possible to make more informed choice when selecting combinations of XP practices.

So far, when we introduced XP, we had to apply practices based on knowledge of what look to be independent pros and cons of each practice. In such cases, the application of practices can fail or becomes less effective than expected, because of unforeseen interactions between practices. By modeling these interactions between practices based on our experiences, we can foresee the effects of practices and prepare for problems. This might make the introduction of XP safer even when we cannot apply all of it's practices.

For example, if improvement of productivity has the highest priority, we can introduce XP effectively. First we can consider 8 practices which relate to the improvement of productivity in table 6 and choose ones which can be applied. Then we can consider other practices on which the chosen ones depend and alternatives for ones which we cannot apply. In such a way, we can use table 6 when we examine effective selections and alternatives of practices considering purposes and constraints of our projects.

We know that we did not use all practices under the various possible situations, and we did not experience all possible effects of interactions between practices. And because there can be many projects which have characteristics that are different from our projects, we don't say that our knowledge can be generally applied. We are going to apply XP to various projects using the knowledge gained. We would understand the effect of interactions and find effective ways to choose practices more accurately. Our goals are to accumulate case studies and patterns of XP introductions.

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] Beck, K., Boehm, B., Agility through discipline: A debate, *IEEE Computer, pp.44-46, June, 2003*.

[2] Beck, K., *eXtreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[3] Beck, K., *Test-Driven Development: By Example,* Addison-Wesley, 2003.

[4] Bellin, D., Simone, S. S., *The CRC Card Book*, Addison-Wesley, 1997.

[5] F., Martin, Kent, B., Brant, J., Opdyke, W., Roberts, D., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[6] Grenning, J., Launching eXtreme Programming at a Process-Intensive Company, *IEEE Software, Vol.18, No.6, pp.27-33, 2001*.

[7] Jeffries, R., What is eXtreme Programming?, *http://www.xprogramming.com/xpmag/whatisxp.htm*, 2001

[8] Rasmusson, J., Introducing XP into Greenfield Projects: Lessons Learnd, *IEEE Software, Vol.20, No.3, pp.21-28, 2003*.

[9] Sakai, M., Matsumoto, K., Torii, K., A new framework for improving software development process on small computer systems, *International Journal of Software Engineering and Knowledge Engineering, vol.7, no.2, pp.171-184, 1997*.

[10] Schuh, P., Recovery, Redemption, and eXtreme Programming, *IEEE Software, Vol.18, No.6, pp.34-41, 2001*.

[11] Williams, L., Kessler, R., *Pair Programming Illuminated*, Addison-Wesley, 2002.

[12] William A. Wood, William L. K., Exploring XP for Scientific Research, *IEEE Software, Vol.20, No.3, pp.30-36, 2003*.

[13] Japan XP User Group, Japan XP User Group, *http://www.xpjug.org/*, 2001.