A study of Human Solutions in eXtreme Programming

Robert Gittins and Sian Hope
School of Informatics
University of Wales Bangor
{rgittins,sian}@informatics .bangor.ac.uk

Keywords: POP-VI.A. *exploratory* POP-V.B. *interviews*

Abstract

This paper develops the use of 'guided interviews' and questionnaires in an empirical study of a small software development company. In particular, the study identified a number of human issues in communications, technology, teamwork and political factors that significantly influenced the implementation and evolution of eXtreme programming into a software development team.

Introduction

The pressure to produce working software code 'to order' is driving Guru programmers to produce fast track solutions, which are indecipherable to other developers and unmaintainable in consequence.

This study looks at eXtreme Programming (XP) practices, as exercised by a small company implementing a relatively new methodology, and records attitudes, behaviour and change in applying human solutions to resolving the traditional problems of software development.

Answers to the software development dilemma are emerging from the study of implicit human influences, by the use of qualitative as opposed to quantitative methods. Recent thinking in software engineering takes heed of some long established methods such as ethnography and grounded theory, emanating from the Social Sciences and witnessed last year during a workshop at the International Conference on Software Engineering (ICSE2000).

Problems dominating staff development, productivity and efficiency are parts of a complex human dimension uncovered in a qualitative approach. The interpretation and development of XP's 'Rules and Practices' are reported, as well as the interlaced communication and human issues effecting the implementation of XP in a small business.

Related work

Previous qualitative research, e.g. Seaman (1990), Sharp et al. (1990) and Cockburn (2000), has concentrated on non-judgmental reporting, with the intent of provoking discussion within the culture being studied by providing observations and evidence, collaborators deciding for themselves whether any changes were required. In this study, the researcher is immersed for an extended period in the software developer team; thereby the active researcher becomes instrumental in the development and improvement of XP. Seaman (1990) describes an empirical study that addresses the issue of communication among members of a software development organization. Sharp et al. (1990) use combined ethnography and discourse analysis, to discover implicit assumptions, values and beliefs in a software management system. Cockburn and Williams (2000) investigate 'The cost benefits of pair programming'. Sharp et al. (2000) describe a 'cross-pollination' approach, to a deeper understanding of implicit values and beliefs.

XP developed recently from Beck (2000) and Beck and Fowler (2000) and more recently from Jeffries et al. (1974). Williams and Kessler (2000) study lone and paired programmers, and in Williams et al. (2000) they study the cost effectiveness of pairing programmers.

The Study

The focus of this paper is a small software company implementing XP, comprising a team of nine developers. The Company decided to implement XP in a progressive manner, conscious of minimising disruption to the business process. When research started, the Company had recently moved to larger offices, their involvement with XP consisted of some intermittent attempts at 'pairing' the developers. Their move presented opportunities for improving 'pairing' proficiency, and the selective adoption of XP practices.

Company policy was to commit themselves, and sufficient resources, wholeheartedly to the introduction of XP as quickly as possible. In the long term, it was envisaged that other departments would follow and that a period of educating staff to the concepts and practices of XP would be necessary to facilitate communication and help smooth the transition to accommodate that change.

Qualitative Research Work

This research adopts some of the techniques historically developed in the Social Sciences, ethnography, qualitative interviews and discourse analyses, an understanding of 'grounded theory' was particularly important. Grounded theory can provide help in situations where little is know about a topic or problem area, or to generate new ideas in settings that have become static or stale. Developed by Barney Glaser and Anselm Strauss (1967), grounded theory deals with the generation of theory from data. Researchers start with an area of interest, collect data, and allow relevant ideas to develop. Rigid pre-conceived ideas are seen to prevent the development of research. To capture relevant data, qualitative research techniques are employed that include the immersion of the researcher within the developer environment, qualitative data analyses, guided interviews, and questionnaires.

Qualitative Data: Qualitative evaluation allows the researcher to study selective issues in detail, without the pre-determined constraints of 'categorised' analyses. The researcher is instrumental in the gathering of data from open-ended questions. Direct quotations are the basic source of raw materials, revealing the respondent's depth of concern. This contrasts with the statistical features of quantitative methods, recognised by their encumbrance of predetermined procedures.

Qualitative Interviews: Patton (1990) suggests three basic approaches to collecting qualitative data through interviews that are open-ended. The three approaches are distinguished by the extent to which the questions are standardised and predetermined, each approach having strengths and weaknesses, dependant upon the purpose of the interview:

- '*Informal conversational'* interviews are a spontaneous flow of questions where the subject may not realise that the questions are being monitored.
- The '*General interview guide'* approach, adopted extensively for this study, predetermines a set of issues to be explored.
- The 'Standar*dised open-ended interview'* pursues the subject through a set of fixed questions that may be used on a number of occasions, with different subjects.

In a series of interviews, data was collected using 'Informal conversation' and verbatim transcripts taken from 'General guided interviews'.

Questionnaires: In an extensive questionnaire consideration was given to the 'Rules and Practices' of XP. Questions targeted the software development process, XP practices, and both managerial and behavioural effectiveness. Behavioural questions were based upon Herzberg (1974).

'Hygiene and Motivation Factors'. Ample provision was provided for open comments on each of the topics, and a developer floor plan provided for a respondent to suggest improvements to the work area. Repeating the questionnaire at three monthly intervals will help research and management by matching the maturing XP practices, as they progress, against developer responses.

XP Rules and practices

Pair Programming: Beck (2000), Beck and Fowler (2000). XP advances what has been reported for some time Cockburn (2000), Williams and Kessler (2000), Williams et al. (2000); Two programmers working together generate an increased volume of superior code, compared with the same two programmers working separately. The Company management discussed the implementation of 'pairing' with the development team, who unanimously agreed to 'buy-in' to the practice. The first questionnaire showed some of the team were unhappy with pairing. 28% of developers preferred to work independently, 57% didn't think they could work with everyone, and 57% stated that pair programmers should spend on average 50% of their time alone. XP practices recommend no more than 25 % of a conditional 40-hour week be paired. Two developers summed up the team's early attitude to pair programming:

"I feel that pair programming can be very taxing at times, although I can see the benefits of doing it some of the time."

"Not everyone makes an ideal pair. It only really works if the pair is reasonably evenly matched. If one person is quiet, and doesn't contribute, their presence is wasted. Also, if a person is really disorganised and doesn't work in a cooperative way, the frustration can (disturb) the other participant!"

Developers estimated that they spent approximately 30% of their time pairing, with partner changes occurring only upon task completion, changes being agreed and established ad hoc. Frequent partner swapping, and partner mixing, commands great merit in XP. Pairing practices matured with the introduction of a team 'Coach' and later a 'Tracker' as described by Beck (2000). Maintenance tasks were another problem which routinely disrupted pairing. Here control was reviewed and tasks better ordered to minimise this problem.

Two months after the first questionnaire, developers were still struggling with compatibility problems but with evidence of progress:

"Pair programming is aided when both parties are willing and able to share the keyboard. It becomes tedious for the person who lets another type when they are not up to speed, and it is equally tedious for the person who has to watch someone who is experienced on a given subject type away. When two (developers) have roughly the same understanding on a subject/language/method then pair programming works brilliantly. I feel in general pair programming is a lot better than developers coding alone."

Some developers preferred thinking time before starting to code and this was uncomfortable for them:

" I am just as happy working on my own as with others. I get on well with most people, but prefer to think a little more about tasks before going on to code than some do."

In time, the impact of pairing activity upon developers will translate into evidence, returned in the periodic questionnaire reviews and interviews, and in the timeliness and quality of code production.

Planning Games: Jeffries et al. (2000). Planning games were introduced soon after pairing practices were established. The 'customer' duly choose between having more stories, requiring more time; against a shorter release, with less scope. Customers are not permitted to estimate story or task duration in XP and developers are not permitted to choose story and task priority. Where a story is too complex or uncertain to estimate a 'Spike' is created. Spike solutions provide answers to complex and risky, stories. The Company succeeded well in developing Planning games, utilising 'Spike solutions' by logging a 'spike' as a fully referenced story, to quickly attack the problem, reducing a complex, inestimable, story to a simple and easily understood, group of stories. Results were very effective; 'spike solutions' proved easy to develop and derived estimates for completion proved consistently accurate. It was common practice to have the essential elements of both iteration

and release Planning games, combined into one meeting. This practice worked for them, in the context of the jobs they were planning.

Client On-site: Beck (2000). The Company rarely had this luxury. When required the role was undertaken by a 'Client's representative', co-opted from the Customer services department, who had worked closely with the client and was able to accept that responsibility. Developer Manager: "The inclusion of a representative from Customer services has proven to be hugely beneficial, providing immediate feedback of the systems successes and failures on a day-to-day basis."

Communication: Beck (2000). A great deal of attention is necessary in providing an XP environment in keeping with the practices to support XP. Key factors in communication are the use of white boards, positioning and sharing of desk facilities to facilitate pair programmers, 'stand-up' meetings, developers 'buying-in to the concepts of the 'rules and practices' of XP, and collective code ownership. Interviews and questionnaires revealed many areas of concern amongst developers. For example 86% of developers disagreed that meetings were well organized; "Agreements at meetings are not set in concrete" and, "Confidence is lost with meeting procedures, when agreed action or tasks are later allowed to be interpreted freely by different parties"; management were quick to address these concerns by concentrating on the development of XP story card practices. Developers were encouraged to agree, and finalise with the client, the task description and duration estimates at timely Planning Game meetings. Story cards were fully referenced and signed, the card becoming the responsibility of the initiating developer until completion. Only the responsible creator of a Card was authorized to amend it.

The use and placement of white boards is said to be an essential means of good communication in XP practices, Beck (2000). Mobile whiteboards were introduced by The Company, soon after pair programming practices gained momentum, and used to record the stories agreed at Planning Game meetings. At one point, story cards were physically stuck to the boards in prioritised order with adjacent notes written on the board. This proved unpopular; and developed into cards being retained but not stuck on the white board. Stories were written on the boards, referenced stories contained ownership, estimation, iteration and priority details, and displayed in columned format. On completion, the owner added the actual task duration, providing feedback to future Planning Games.

Stand-up meetings promote communication throughout the team. The Company introduced this practice from day one. At ten o'clock every morning, a meeting allowed everyone to briefly state (standing promotes brevity) their work for the day and discuss problems arising from the previous days activity. Anyone was free to comment, offer advice or volunteer co-operation. The benefits of adopting stand-up meetings were far-reaching and seen by developers and management as an effective way to broadcast activities, share knowledge and encourage collaboration amongst and between team members and management. The Company meetings tended to degrade when reports migrated predominantly to topics of yesterday's activity, rather than those planned for the day. This activity persists and may remain or need to be resolved and modified as their particular brand of XP develops.

Simple Design: Beck (2000) summarises simple design by 'Say everything once and only once'. However a comment by one developer interviewed revealed a common concern, "Sometimes, it is a bit too simplistic, and issues seem to be avoided". XP states that it is important to produce a simple system quickly, and that 'Small Releases' are necessary to gain feedback from the client. The Company didn't see themselves in a position to implement this practice so early in their XP programme. XP allows companies to cherry-pick those practices they regard suitable for implementation, in the order they see fit.

Tests: Unit tests are written in XP before coding. This gives an early and clear understanding of what the program must do. Time is saved both at the start of coding, and again at the end of development. Latent resistance to early unit testing is manifest, when the perceived closeness of a deadline looms. This activity is perhaps the hardest to implement and requires commitment from developers. An early questionnaire revealed that 71% of The Company developers regarded unit-testing practices in general to be 'very poor'. Developer Manager on early introduction of unit

testing: " If you already have a large complex system, it is difficult to determine to what extent testing infrastructure is to be retrospectively applied. This is the most difficult aspect in our experience. From scratch it is much easier to make stories and code testable."

Refactoring: Fowler (1999). 'The process of improving the code's structure while preserving its function.' The use and reuse of old code is deemed costly, often because developers are afraid they will break the software. XP indicates that refactoring throughout the project life cycle saves time and improves quality. Refactoring reinforces simplicity by its action in keeping code clean and reducing complexity. The Company had not developed refactoring activities in line with XP at that time. Many developers expressed concern with refactoring, more commonly reported by traditional companies: " … with more people, we could spend more time refactoring and improving the quality of our existing code base." The questionnaire revealed that 45% of developers considered refactoring sporadic or very poor.

Collective Code Ownership: Beck (2000), Beck and Fowler (2000). This concept states "Every programmer improves any code anywhere in the system at any time if they see the opportunity." Collective code ownership has many merits; It prevents complex code entering the system, developed from the practice that anyone can look at code and simplify it. It may sound contentious, but XP Test procedures should prevent poor code entering the system. Collective Code Ownership also spreads knowledge of the system around the team. The Company experienced growing pains in developing this principle, revealed by the comments of two developers:

"I have conflicting interests in collective code ownership. I think it is very good when it works, but there are times when some code I have written seems to just get worse when others have been working on it."

"I like the idea of collective code ownership, but in practice I feel that I own, am responsible for, some bits of code."

From this traditional perspective, it will be important to record how attitudes change, as XP practices mature.

Metaphor: A metaphor in XP is a simple shared story to encompass and explain what the application is 'like', communicating a mental image, so that everyone involved can grasp the essence of the project in a term universally understood. This may seem to be a relatively easy, or lightweight, activity to adopt. However, the value of this practice was not immediately evident to developers, early difficulties developing and applying suitable metaphors were experienced and this practice was reluctantly abandoned for future consideration.

Starting from Scratch

Traditional companies adopting XP have many difficulties to overcome. During the course of the research a number of companies were visited and short interviews with management recorded. Like many companies who have developed over the last ten years, a common thread was found that characterised there predicament. The main features showed traditional teams of developers, comfortably established, working in small offices and in prohibitively closed environments. Management were very aware that legacy software in circulation was in the 'ownership' of one or two heroic developers, at the cutting edge of their business. Some teams were badly under-performing and in many cases consultants had been approached to resolve their problems. Management were often reluctant to allow access for research teams to visit developer offices. Tensions often ran high. In these companies, 'Risks' Beck (2000) are high, quality is compromised, communication difficult, and control largely ineffective. There are other considerations when starting from scratch; The study Company developer-manager reflects upon implementing XP projects:

" One of the key 'discoveries' has been the relative ease to which XP has been employed on an all-new project, and the difficulty in applying XP retrospectively on an established system."

Conclusions

XP unlike most traditional methodologies, concerns itself with complex human issues and intangibles such a pair relationships, code ownership, and values and principles. A combination of qualitative and quantitative methods has helped identify solutions in applying XP practices in a small software development company. How particularly one company interpreted and developed their brand of XP, moulded from their successes and failures. Successes in such areas as the use and development of 'spike solutions', and Customer role-play within 'Planning Game' activity, and from failures, as in developer reluctance to 'buying-in' to 'collective code ownership', and the difficulties of implementing the practice of 'simple design', and in the use of 'metaphors'. Partial success was seen in 'Pair programming', that having posed early problems, showed improvement in maturity. Future work will monitor the complex factors in the development of XP within small and growing companies at various levels of maturity. By acknowledging the characteristic unsharp boundaries of qualitative data sets, future work will investigate the use of fuzzy logic for data analyses.

Acknowledgements

This paper acknowledges the funding and support of the EPSRC (Award No. 99300131).

References

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison Wesley.

Beck, K. and Fowler, M. (2000). *Planning Extreme Programming*. Addison Wesley.

Cockburn, A. and Williams, L. (2000)

The cost benefits of pair programming. Downloadable from http://members.aol.com/humansandt/papers/pairprogrammingcostbene/pairprogrammingcostbene.htm

Fowler, M. (1999). *Refactoring: Improving the design of existing code.* Addison Wesley. July 1999.

Glaser, B. G., and Strauss, A. L. (1967). *The Discovery of Grounded Theory: Strategies of Qualitative Research*. Chicago: Aldine Publications.

Herzberg, F. (1974) *Work and the Nature of Man*. Granada Publications Ltd.

Jeffries, R. Anderson, A. and Hendrickson, C. (2000). *Extreme Programming Installed*. Addison Wesley

Patton, M. Q. (1990*). Qualitative Evaluation and Research Methods.* (2nd Ed.). SAGE Publications

Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, Vol. 25 (4):557 - 572, July/August 1999.

Sharp, H., Woodman, M., Hovenden, F. & Robinson, H. (1999). The role of 'culture' in successful software process improvement. *EUROMICRO* Vol.2, p17. IEEE Computer Society.

Sharp, H., Robinson, H., and Woodman, M. (2000). Software engineering: community and culture. *IEEE Software*, Vol. 17, No.1, Jan /Feb

Williams, L. A. and Kessler, R. R. (2000). All I really wanted to know about pair programming I learned in kindergarten. *Communications of the ACM*. May 2000 Vol.43, No5.

Williams, L. A., Kessler, R. R., Cunningham, W., and Jeffries, R. (2000). Strengthening the case for pair programming. *IEEE Software*, Vol. 17, No. 4: July/August,2000,pp19-25.