Pair Programming: What's in it for Me?

Andrew Begel
Microsoft Research
One Microsoft Way
Redmond, WA 98052
andrew.begel@microsoft.com

Nachiappan Nagappan
Microsoft Research
One Microsoft Way
Redmond, WA 98052
nachin@microsoft.com

ABSTRACT

Pair programming is a practice in which two programmers work collaboratively at one computer on the same design, algorithm, or code. Prior research on pair programming has primarily focused on its evaluation in academic settings. There has been limited evidence on the use, problems and benefits, partner selection, and the general perceptions towards pair programming in industrial settings. In this paper we report on a longitudinal evaluation of pair programming at Microsoft Corporation. We find from the results of a survey sent to a randomly selected 10% of engineers at Microsoft that 22% pair program or have pair programmed in the past. Using qualitative analysis, we performed a large-scale card sort to group the various benefits and problems of pair programming. The biggest perceived benefits of pair programming were the introduction of fewer bugs, spreading code understanding, and producing overall higher quality code. The top problems were cost-efficiency, (work time) scheduling problems, and personality conflicts. Most engineers preferred a partner who had complementary skills to their own, who was flexible and had good communication skills.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management – Programming teams.

General Terms

Measurement, Human Factors.

Keywords

Pair Programming, Survey experiments, Developers, Empirical studies.

1. INTRODUCTION

Pair Programming (PP) [19] is a software development practice that is gaining significant popularity in academia [13, 14, 21]. Pair programming refers to the practice whereby two programmers work together at one computer, collaborating on the same algorithm, code, or test. One member of the pair is the *driver*, who actively types at the computer, or records a design or architecture. The other plays the role of *navigator*. The navigator watches the work of the driver, attentively identifying defects and making suggestions. The two are also continuous brainstorming partners.

ESEM'08, October 9-10, 2008, Kaiserslautern, Germany.

Pair programming is part of the Extreme Programming methodology [3] that is gaining widespread use in industry. Most research on pair programming has been focused in an academic environment. There have been limited studies about pair programming in industry, and these only provide preliminary evidence of some empirical results related to quality and productivity. As researchers, we would like to understand how pair programming methodologies are used, what kinds of problems and benefits they are perceived to have, the types of partners people would like to work, and a general consensus on PP's usefulness in the software engineering professional community. We believe strongly in using quantitative and qualitative empirical methods to explore questions generated by these research topics. While there is much to be learned from looking at the software created by developers and from measuring developer productivity and software failure-proneness, we can gain great insight by asking software developers directly about their current development practices, and about their perceptions of the development processes that are spreading through the software engineering community.

We conducted a web-based survey of Microsoft developers, testers, and managers who are directly involved in the development of software. Our pair programming questionnaire was part of a larger survey on Agile software development [4]. The survey asked software engineers to respond to questions on perceived problems and benefits of pair programming, the type of partner they would like to work with, and their perceptions on whether pair programming takes more time, produces higher quality code, etc.

We find that 22% of participants have practiced pair programming, but only 3.5% do it in their current project. Most practitioners are more experienced than the average Microsoft employee practicing other Agile development methodologies. Around two-thirds like pair programming and believe it is a workable practice, but less than half would agree to that of their team's use of pair programming. Three-eighths of the respondents believe that pair programming takes more time than programming alone, but two-thirds believe that the quality of the resulting software is better.

Summarizing what we discovered about the perceived benefits of pair programming, two people working together on the same problem derive process improvements that result in better software. The problems perceived with pair programming are all forms of anxiety: individual anxiety working closely with someone else and an organizational anxiety over allocation of funds to pair programming. Many of the attributes of a good pair programming partner are similar to that of a good spouse. The partner should communicate well, complement the other's skills and personality, and in some areas be better than the other to stimulate

learning experiences as well as help solve problems together. A good pair programming team is fast, efficient, and effective because they have complementary skills, communicate well, are sensitive to the other's needs and personality, and work without antagonizing one another.

The rest of this paper is organized as follows. In Section 2, we discuss our contributions, and in Section 3 review the related research. Section 4 describes the survey methodology and illustrates the quantitative results. In Section 5, we discuss the benefits and problems of pair programming as perceived by engineers at Microsoft and the characteristics of a good pair programming partner based on our qualitative data card sort. Finally, Section 6 concludes with a review of our most important findings and their implications for future research.

2. CONTRIBUTIONS

In this paper our main contributions are

1. Quantitative data on the adoption of pair programming in a large software company.
2. The perceived benefits and problems of pair programming.
3. The characteristics an engineer looks for in an ideal pair programming partner and team.
4. The perception that pair programming produces higher quality code at the expense of more time compared with solo programming.

3. RELATED WORK

In our discussion of related work we classify the previous work in this area broadly as academic and industrial case studies.

3.1 ACADEMIC CASE STUDIES

Researchers at the University of California, Santa Cruz (UCSC) have reported positive results in studies involving pair programming with students [5, 12]. Their studies indicate that pair programming helped in increasing the retention rate of students who might have otherwise dropped out of the introductory programming course. They also indicate that pair programming students produce better quality code and perform comparably on exams with respect to solo programming students. Their research in pairing protocol leads them to recommend pairing students with others within the same section; pairing students with others with similar skill level; and ensuring that there is a coding standard. Additional research on a large sample of students (555 students) indicates that pairing bolsters the course completion and pass rates and leads to higher retention of students in a computer-related major. Furthermore, students show a positive attitude towards pair programming.

Similar research performed at North Carolina State University [15, 16] indicates that pair programming helps retain more students in the introductory computer science stream. Students in paired labs have a more positive attitude toward working in collaborative environments, and students who pair program in introductory classes do not perform adversely in future classes when they program individually.

Research results [8, 20] based on experiments held at the University of Utah in a senior-level software engineering course indicate that pair programmers produce higher quality code in about half the time when compared with solo programmers. But experiments conducted at the Poznan University of Technology, Poland [17] have obtained opposite results which indicate that pairs spend almost twice as much total programmer effort as solo programmers.

Thomas et al. [16] at the University of Wales show that students with least self-confidence enjoyed pair programming the most. In addition, most students with a higher skill level preferred not to pair with students of a lower skill level. The researchers discovered some initial evidence that students produce their best work when they are paired with a partner of equal skill and confidence level. Similarly, Katira et al. [10] examined compatibility among freshmen, advanced undergraduates, and graduate students. They found that the students' perception of their partner's skill level had a significant influence on compatibility. Graduate students worked well with partners of similar actual skill level. Similarly, first year undergraduates seem to work better with partners with different Myers-Brigg [6] personality types.

3.2 INDUSTRIAL CASE STUDIES

Chong and Hurlbutt [7] conducted an ethnographic observation of two teams of professional pair programmers over a period of four months. They found that professional programmers played different roles being engaged jointly in programmer activities than the traditional driver/navigator pair discussed in literature [19]. They identified expertise and keyboard control as important factors influencing pair programming interactions. In another industrial case study, Hulkko and Abrahamsson [9] observed mixed results regarding the perceived benefits of pair programming. Using empirical data from four projects written in C++ and Java ranging from 3.7 KLOC (thousand lines of code) to 7.7 KLOC, they observed a comparable defect density between solo and pair programming on one of their projects, and in another, a significantly lower defect density for the pair programmers. From the perspective of productivity (measured using KLOC per person hour), pair programming productivity in two of the projects was better than solo programming, but a third project was worse. All of these results indicate that pair programming may not consistently provide benefits to code quality or superior productivity when compared with solo programming.

In possibly one of the largest relevant studies [1], 295 software professionals of varying expertise from Norway, Sweden and UK participated in a controlled experiment on pair programming. The participants were divided into 98 pairs and 99 individuals. They performed several development and maintenance tasks on two Java systems with differing degrees of complexity. The results of this experiment do not support the hypothesis that pair programming reduces the time required to complete tasks correctly, nor does it increase the proportion of correct solutions. Conversely, they found an 84% increase in effort expended to complete the tasks correctly. Junior professional programmers enjoyed increased correctness when developing complex systems, whereas intermediate and senior professional programmers required less time to perform maintenance tasks correctly on simple systems.

From the above studies we observe that pair programming appears to be very different in academic than in industry. In academia, pair programming is used for education and has positive effects on student retention. In industry, there is little research on pair programming, and what little there is shows conflicting results. Surprisingly, we do not have a good qualitative assessment of how

professional programmers might explain these results. Our research aims to address some of these questions.

4. SURVEY METHOD

Our research was conducted using an anonymous web-based survey offered over a period of two weeks in October 2006 within Microsoft. As presented in our earlier paper [4], an invitation was sent by email to 2,821 recipients, randomly selected from a much larger pool of around 28,000 software developers, test developers, and program managers. We extracted the 28,000 engineers' email addresses into a spreadsheet, sorted them by job role, picked a random 10% of each role, and invited them to participate in our survey. The survey had several questions on the influence of pair programming on quality, productivity, and learning. We also requested free-form responses for the top benefits and problems of pair programming and for the characteristics of an ideal pair programming partner and team.

4.1 SURVEY DESIGN

Kitchenham and Pfleeger [18] discuss the design and construction of personal opinion surveys using the following steps: searching the relevant literature; construct an instrument; evaluate the instrument; document the instrument. In our survey, as suggested by Kitchenham and Pfleeger, we use formal notations, limit our respondents responses to numerical, Yes/No type, Likert-scale, and short free form answers. Respondents were anonymous, but could identify themselves (separate from their survey responses) to enter a drawing for a $250 reward. We followed Kitchenham and Pfleeger's advice on the need to understand whether the respondents had enough knowledge to answer the questions in an appropriate manner. For this, we restricted the people invited to participate in the survey to people in technical roles (no sales or marketing employees). Second, even if people had never pair programmed, they could skip the survey and still be included in the drawing, ensuring that no one felt compelled to take the survey for the chance to win the gift.

We received 491 responses, of which 4 were invalid (two duplicates and two empty surveys), for an overall response rate of 17%. Response rate for developers was 18%, testers were 18%, and managers were 10%. The survey asked about the respondents' experiences and perceptions on Agile software development and pair programming. The Agile software developments results were discussed in an earlier paper [4] – this paper focuses on pair programming.

All of the free response answers were printed out on more than a thousand note cards. We sorted them to categorize the responses by thematic similarity (as illustrated in LaToza et al.'s study [11]). The themes that emerged during the sort were not chosen beforehand. Figure 1 shows the card sort with themes in colored cards. Respondents reported 435 benefits of pair programming and 350 problems. They also reported 447 attributes of a good pair programming partner, and 369 attributes of a good pair programming team.

Figure 1: Card sort

4.2 DEMOGRAPHICS

Among our overall sample of 487 respondents, 106 respondents have pair programmed in the past or are currently using pair programming (21.7 % of the overall respondents). This is the sample population we use for the analysis in this paper. 17 of these 106 respondents are using pair programming in their current development project. Of the 106 respondents, 68% are involved in development, 21% in testing, 8% in program management, and the remainder in documentation, research, etc. 81.1% of these respondents are individual contributors, 14.1% are managers and 4.8% are managers of managers. Respondents had an average of 10.6 years experience in the software profession (standard deviation was 7.4; median 9.0). They have worked on their current team for an average of 2.1 years (standard deviation 2.2, median 1.5). The respondents were spread across different geographical locations across the world. The work experience levels indicate that our respondent population is fairly experienced – more than our previous results on Agile software development [4].

Figure 2: Pair Programming Example at Microsoft

Figure 2 shows a typical example of pair programming for Microsoft. This archival picture shows a shared keyboard and large monitor arrangement between two developers, and is similar to the practice in academic case studies discussed in Section 2.

5. PERCEPTIONS OF PAIR PROGRAMMING

In this section we discuss the perceptions towards pair programming by engineers at Microsoft obtained directly through the survey and through the card sort.

5.1 INDIVIDUAL ATTITUDES TOWARDS PAIR PROGRAMMING

We asked respondents Likert-scale questions to learn whether they liked pair programming and how well it was working for them. Figure 3 shows the results. 64.4% said they believe that pair programming works well for themselves and 62.8% believe it works for their partner. A lower percentage (48.2%) say PP works for their team, and an even lower number (39.2%) say it is working for their larger organization. This reflects the grassroots nature of pair programming adoption at Microsoft. Individual teams are trying out pair programming, but are finding it difficult to get management buy-in to spread the practice.

Respondents were also asked three questions about the effects of pair programming on their work (Figure 4). First, addressing the perception that pair programming wastes two programmers to do the job of one, only 37.5% of respondents agree with the statement that pair programming takes more time to do the same work that one person could have done alone. Only 25.4%, however, believe that it does not. More than a third of respondents do not say either way. Almost two-thirds of respondents (64.5%) agree that pair programming results in fewer bugs in the code. We were worried that the bugs they left in the code would be harder to fix, but 35% of respondents disagree. Only 17.8% feel that pair programming leaves difficult bugs behind in the code.

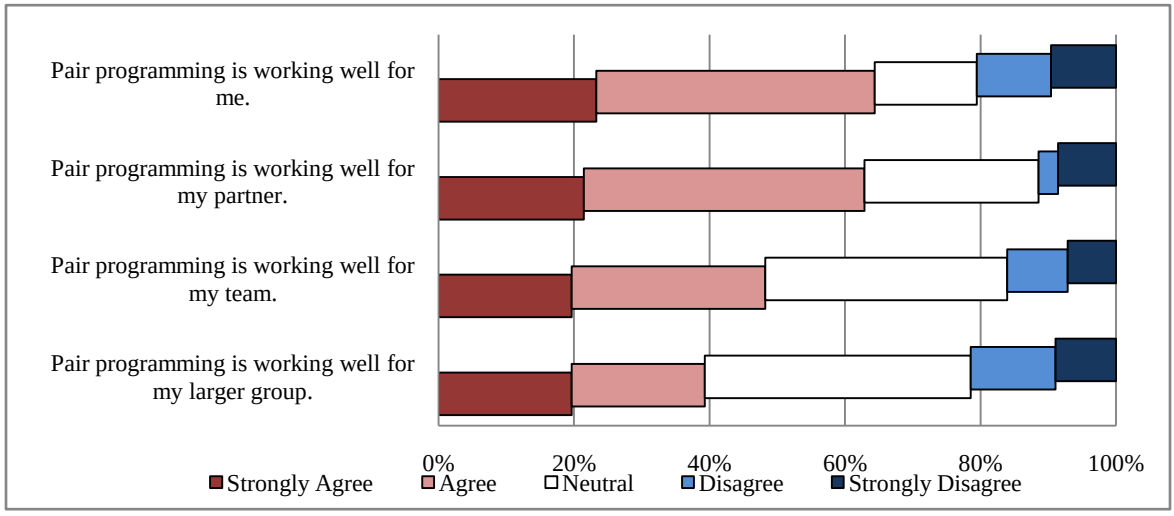

Figure 3: Individual attitudes towards pair programming

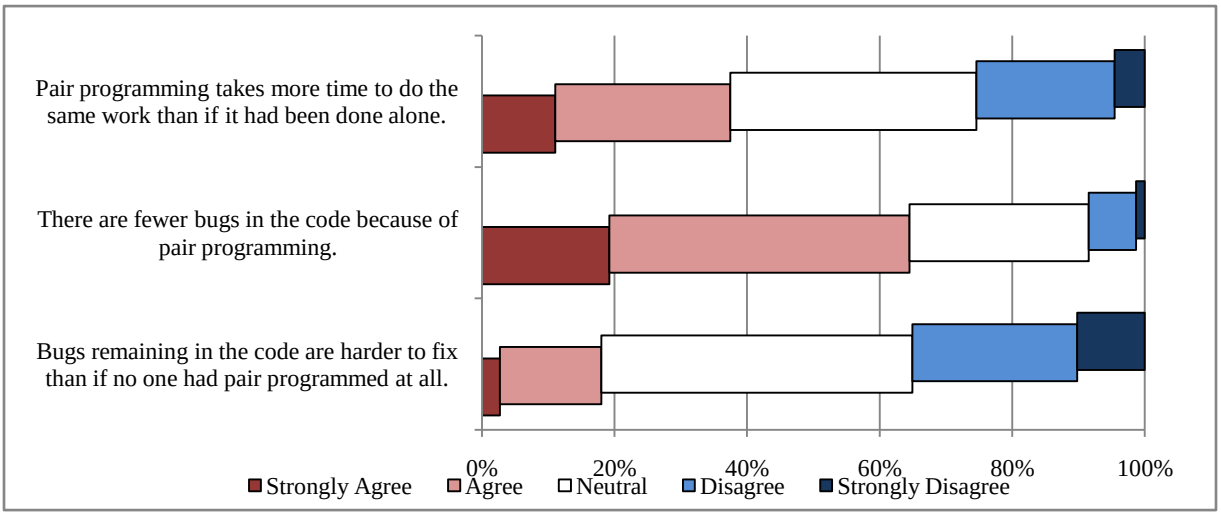

Figure 4: Effects of pair programming on work

5.2 PAIR PROGRAMMING BENEFITS

We asked all survey participants what they thought were the top three benefits and problems with pair programming. We also asked them what they thought were attributes of a good pair programming partner and a good pair programming team. Comments from the respondents are presented in italicized form to add more contextual information.

Table 1 presents the top 10 benefits of pair programming as perceived by the respondents, and the number who cited it as a benefit. The top benefit was fewer bugs in the source code. One person said "*it greatly reduces bug numbers.*" Simple bugs were found and fixed, as one respondent reported, "*there are fewer 'petty' bugs.*" In addition, respondents speculated that the longer bugs live in the code, the more difficult they are to fix. Using pair programming, "*bugs are spotted earlier*" in the development process, and "*may prevent bugs before [they are] deeply embedded.*"

Table 1: Pair Programming benefits

1. Fewer Bugs	66
2. Spreads Code Understanding	42
3. Higher Quality Code	48
4. Can Learn from Partner	42
5. Better Design	30
6. Constant Code Reviews	22
6. Two Heads are Better than One	22
8. Creativity and Brainstorming	17
9. Better Testing and Debugging	14
10. Improved Morale	13

The second most cited benefit indicated that pair programming helps to spread code understanding between the members of the pair. Pair programming provides "*shared equal deep knowledge of the product*," and is an efficient means to promote "*greater understanding of a larger codebase across the team.*" From a risk-aversion viewpoint, pair programming is a defense against employee attrition – "*There is never only one person in the team who knows all the code.*"

In third place was higher quality code. It provides "*improved software quality*," and "*higher quality code in terms of consistency with guidelines.*" The end result is a "*better quality product.*" Pair programming helps improve quality "*through more extensive review and collaboration.*"

Fourth was the ability to learn from a partner. The use of pair programming is a good way to "*quickly ramp-up new members*," and enables "*users to learn new techniques faster*." Partners like to teach as well, suggesting that "*mentoring and showing someone who is unfamiliar with the code is the best benefit.*" Pair programming has reciprocal benefits to both partners because "*everyone learns constantly from each other.*" "*That makes each member of the pair a stronger coder and employee.*"

The fifth benefit was the perception that software being built had a "*better architecture and implementation*" mainly due to "*adherence to good design and standards.*" Dissent is not only tolerated, but encouraged. Designs and architectures "*are challenged from the start, so designs are either great to start with, improved, or scrapped and replaced with a better design*." More cooks in the kitchen provide "*designs with the benefit of broader insight*" and "*differing opinions on their approach*."

Rounding out the top ten benefits of pair programming were constant code reviews, the notion that two heads were better than one, more creative brainstorming, better testing and debugging of the software, and improved morale. The Appendix at the end of the paper lists the other perceived benefits of pair programming we found in our survey.

5.3 PAIR PROGRAMMING PROBLEMS

Table 2 highlights the top ten problems with pair programming as perceived by the respondents and the number who cited it as a problem. The number one problem reported with pair programming is cost. Two people are being paid to the do the work of one. Pair programming "*requires 'twice as many' people*," making it "*difficult to justify the cost up front*." There was considerable "*skepticism that having two people working on one task is a good use of resources*" – the time taken to "*complete the project is not halved.*" Manager buy-in was also challenging, with one stating, "*if I have a choice, I can employ one star programmer instead of two programmers who need to code in a pair.*"

Table 2: Pair programming problems

1. Cost efficiency	79
2. Scheduling	31
3. Personality clash	25
4. Disagreements	24
5. Skill differences	22
6. Programming style differences	13
7. Hard to find a partner	12
8. Personal style differences	11
9. Distractions	10
10. Misanthropy	9
10. Bad Communication	9
10. Metrics/Hard to Reward Talent	9

The second problem is scheduling time to work in pairs. The two partners require "*equivalent schedules" and suffer "twice the scheduling complications*." "*Blocking out two calendars can be hard. Makes it almost impossible for a lead.*" Pairing "*reduces the freedom of work hours of individual contributors.*"

The third most cited problem is a clash of personalities. "*Personality differences are more disruptive to productivity than in solo programming.*" Similarly, "*if the pair is not on the same frequency, it is a nuisance*," and results "*in potential bad quality*." Finding compatible partners is a difficult process. It is "*hard to find pair programmers that have compatible personalities, value systems and lifestyles.*" "*Many partnerships fail due to personality conflicts,*" especially due to "*infighting, egos, and one person trying to be the superstar.*" Also, overfamiliarity can breed contempt – "*pairs get sick of each other.*"

The fourth problem is trouble resolving disagreements. Pairs find it "*hard to get to a consensus in ideas*." "*Sometimes we waste time on discussion*," where we surmise from the spectrum of responses that to many respondents, 'discussion' is synonymous with 'argument.'

The fifth problem is that engineers are worried that they will be paired with a partner who is not as smart or skilled as they are. They worry that "*it may slow down whiz kids,*" and "*tends to drag the faster/smarter/better person down*." The consequences for the partnership could be dire – "*if the partners' abilities are imba-*

lanced, it could be that one partner become obsolete in the process."

The rest of the top ten problems are anxiety over differences in programming style, difficulty finding a partner to program with, differences in personal style, and pair programming being very distracting and preventing one from getting work done. In a three-way tie for tenth, some respondents just do not like other people, they have trouble communicating with others, and they feel that management finds it difficult to properly attribute rewards to each member of a pair for the work that they do. In the Appendix, we list the remaining problems that respondents reported.

5.4 GOOD PAIR PROGRAMMING PARTNERS

Table 3 highlights the top ten attributes our respondents indicated were important to have in pair programming partner.

Table 3: Attributes of good pair programming partners

1. Complementary Skills	40
2. Flexibility	33
3. Good Communications	31
4. Smart	25
4. Personable	25
6. At Least the Same Skills as Me	21
7. Strong programmer	17
7. Better Skills than Me	17
7. Able to Focus	17
10. Knowledgeable	15

The top attribute of a good pair programming partner is that the person has complementary skills to your own. The diversity of thought process is the overriding sentiment in many responses. My partner "*usually looks at things from a different angle.*" He has a "*different background which provides a different perspective.*" An ideal partner would be "*able to think ... even [from] a different role (e.g. test and dev together).*" Development skills were also important. For example, they should have an "*overlapping but not identical skill set.*" It was useful that an ideal partner is "*experienced in areas that I am not,*" "*blocks on different things than I do*," and "*knows everything I don't know*." Engineers wished for a work partner in the truest sense of the term – "*someone who complements my thinking and skills in terms of technical and design skills*."

The second attribute is flexibility. An ideal partner is "*open minded*," and "*open to new ideas*." He is "*willing to cooperate and step away from the PC with me to work on design or other details.*" He "*understands that there is often more than one 'right way' and doesn't argue every point.*" A key characteristic of an ideal partner is that he is "*not stubborn.*" He has to be "*able to adapt to different working styles*."

Every study of pair programming has shown that it is a communications-intensive process. Thus, not surprisingly, the third most important attribute of a pair programming partner is good communications skills. Industrial developers qualify the communication to ask that their partner be a "*good listener*," "*articulate*," "*easy to discuss code with*," and "*very verbal, so I can understand the thought process*." He should "*enjoy debating and discussing code*," and should "*ask questions, [and] provide opinions*."

Tied for fourth is that the person is smart and personable. A good partner is "*mentally quick,*" "*technically skilled*," and "*intelligent*." He should have "*good interpersonal skills,*" be "*easy to work with,*" have a "*sense of humor,*" and be "*comfortable with people around them.*" Other mental qualities included being an "*analytical thinker,*" an "*excellent problem solver*," and someone who "*can think abstractly*." Demonstrating sensitivity in particular situations is quite important. He should be "*able to correct you without making you feel uncomfortable*," and definitely "*should not be a know-it-all or a person who always gets his/her way*."

The rest of the top ten attributes are that the person has at least the same level of skills as the other, that they are a strong programmer, that they have *better* skills than the other, that they can focus on the job at hand, and are generally knowledgeable. In the Appendix, we list the remaining attributes of a good pair programming partner that respondents reported.

5.5 GOOD PAIR PROGRAMMING TEAMS

Table 4 highlights the top ten attributes our respondents indicated were important to have in a successful pair programming team. Many of these were similar to that of a good pair programming partner, but were from a more summative point of view.

Table 4: Attributes of good pair programming teams

1. Good Communications	32
2. Complementary Skills	31
3. Compatible Personalities	25
4. Team Works Effectively	16
5. No Ego	15
6. Fast and Efficient	14
7. Flexibility	12
8. Common Goals	11
8. Good Quality	11
10. Same programming skills	10
10. Collaborative	10
10. Work Well Together	10

The number one attribute possessed by a good pair programming team is good communication skills. "*Communication, communication, communication*." Again, similar to a good partner, teams need "*compatible communication styles*," where each partner "*communicates often*," especially about design. The way partners interact face-to-face was crucial – they should have "*excellent communication (verbal as well as body language).*" Communication outside the team is important as well. They should be able to "*communicate effectively about what they did to others.*"

Number two is that they should have complementary skills. "*A little diversity here is good*." The team members should "*work off of each other's strengths and weaknesses.*" Considering different ways to tackle problems was very important – partners need "*complementary sets of knowledge [so] they don't get locked into the same approaches every time just due to familiarity.*"

Third is that they have compatible personalities. With "*cooperative personalities, they work well together, rather than trying to compete with one another.*" Also, "*neither pushes their opinion too much to the detriment of the partnership*." An ideal team "*consists of easy-going people who want to listen and share ideas with each other.*" "*Tolerance*" and "*mutual trust*" were vital traits

that both members should feel for one another, as well as "*personality types which aren't domineering.*"

The fourth attribute is that the team works effectively. The team should "*work well together.*" In order to do this, their "*minds [ought to] work in similar ways when solving problems (so they don't argue too much about how to do something).*" Demonstrating that they can produce results, a good team "*delivers quality code on time*," and "*shows [that they] have the collective skills required to achieve the job.*"

Rounding out the top five is that the team should leave their ego at the door. An ideal pair is "*ego-less*," where "*partners are not overly critical*," and are "*permissive to mistakes*." *Each should "respectfully disagree with each other*, " and "*not take criticism of their code as a personal attack against them.*" A team should exhibit a "*willingness to co-excel, as opposed to compete directly with their partner.*" "*No one [should] push their opinion too much to the detriment of the partnership.*"

The rest of the top ten attributes are that the team is fast and efficient, they are flexible, they share common goals, they produce good quality code, they have the same programming skills, are collaborative and that they work well as a pair. The remainder of the positive attributes of a good pair programming team can found in the Appendix.

6. THREATS TO VALIDITY

From the perspective of threats to validity we have the issue of content validity. *Content validity is a subjective assessment of how appropriate the instrument [survey document] is for a group of reviewers with knowledge about the subject matter* [18]. To assess this threat we did a pilot evaluation of our survey with people in different product groups familiar with Agile software development and pair programming. We observed them taking the survey while thinking aloud. What they said about each question and its response was used to remove any confusing elements or misunderstandings due to poor wording in the survey. These pilot participants were not part of the group to which the survey was sent.

From the researcher bias (and internal validity) point of view, the survey was conducted anonymously, by the two authors who do not belong to any Microsoft product division (all respondents belong to the product division). The benefits and problems were self-reported in free-form to remove any bias that could have been introduced by the authors asking the respondents to pick the benefits and problems of pair programming from a list. Furthermore, the authors have no influence on the use or perception of pair programming in Microsoft's product groups.

The main threat to external validity is that the results are from one organization. We cannot assume a priori that the results of our study generalize beyond the specific environment in which it was conducted. Researchers become more confident in a theory when similar findings emerge in different contexts [2]. Towards this end, we intend that our case study will be replicated in different software organizations.

7. CONCLUSION

In this paper we have reported on a large scale survey deployed at Microsoft Corporation to gain insights into perception towards pair programming in industry. Overall our findings indicate that a significant majority (64.4%) of our respondents believe that pair programming works well for them. But as things scale up to the team and organization level they feel pair programming does not work as well. A significant majority (65.4%) of our respondents also feel that pair programming produces higher quality code. Additionally our respondents are divided on whether pair programming take make time than solo programming – only 25.4%, a minority, believe that it does not. The primary perceived benefit of pair programming is better code quality with fewer bugs and the biggest perceived problems deal with cost efficient, scheduling issues and personality clashes with the partner. Most programmers would like to pair with someone who has complementary skills to their own, is flexible and has good communication skills. From the perspective of teams, good communication skills, compatible personality and complementary skills were most desirable.

Our study results show results which differ from prior empirical studies done in academia regarding the preference of pair programmers to work with partners having similar skill levels or knowledge, regarding perceptions towards time spent on pair programming, and regarding quality.

We have found several teams at Microsoft that currently use pair programming. Our next goal is to conduct a focused study with quantitative data from the team's product (like source code, defect measurements), qualitative data (surveys, interviews and ethnographic shadowing observations), and do an end-to-end observational study of a team using pair programming. To further build an empirical body of knowledge, we wish to collaborate with faculty and researchers in academia and industry to collect empirical data on the perceptions of pair programming in different environments.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous respondents to our survey at Microsoft. Researchers and faculty wanting to replicate the study are encouraged to contact the authors for a copy of the survey.

REFERENCES

[1] E. Arisholm, Gallis, H., Dyba, T., Sjoberg, D., "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise", *IEEE Transactions in Software Engineering*, 33(2), pp. 65 - 86, 2007.

[2] V. Basili, Shull, F.,Lanubile, F., "Building Knowledge through Families of Experiments", *IEEE Transactions on Software Engineering*, 25(4), pp. 456-473, 1999.

[3] K. Beck, *Extreme Programming Explained: Embrace Change*, Second ed. Reading, Mass.: Addison-Wesley, 2005.

[4] A. Begel, Nagappan, N., "Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study", Proceedings of Empirical Software Engineering and Measurement (ESEM), pp. 255-264, 2007.

[5] J. Bevan, L. Werner, and C. McDowell, "Guidelines for the User of Pair Programming in a Freshman Programming Class", Proceedings of Conference on Software Engineering Education and Training, Kentucky, pp. 100-107, 2002.

[6] C. Bishop-Clark and D. D. Wheeler, "The Myers-Briggs personality type and its relationship to computer programming", *Journal of Research on Computing in Education*, 26(3), pp. 358-370, Spring 1994.

[7] J. Chong, Hurlbutt, T., "The Social Dynamics of Pair Programming ", Proceedings of International Conference on Software Engineering, pp. 354-363, 2007.
[8] A. Cockburn and L. Williams, "The Costs and Benefits of Pair Programming," in *Extreme Programming Examined*, G. Succi and M. Marchesi, Eds. Boston, MA: Addison Wesley, 2001, pp. 223-248.
[9] H. Hulkko, Abrahamsson, P., "A multiple case study on the impact of pair programming on product quality", Proceedings of International Conference on Software Engineering, pp. 495 - 504, 2005.
[10] N. Katira, L. Williams, E. Wiebe, C. Miller, S. Balik, and E. Gehringer, "On Understanding Compatibility of Student Pair Programmers", Proceedings of ACM Technical Symposium on Computer Science Education (SIGCSE), Norfolk, VA, pp. 7-11, 2004.
[11] T. D. LaToza, Venolia, G., DeLine, R., "Maintaining mental models: a study of developer work habits", Proceedings of International Conference on Software Engineering, pp. 492-501, 2006.
[12] C. McDowell, L. Werner, H. Bullock, and J. Fernald, "The Effect of Pair Programming on Performance in an Introductory Programming Course", Proceedings of ACM Special Interest Group of Computer Science Educators, Covington, KY, pp. 38-42, 2002.
[13] C. McDowell, L. Werner, H. Bullock, and J. Fernald, "The Impact of Pair Programming on Student Performance of Computer Science Related Majors", Proceedings of International Conference on Software Engineering 2003, Portland, Oregon, pp. 2003.
[14] M. M. Müller and O. Hagner, "Experiment about Test-first Programming", Proceedings of Conference on Empirical Assessment in Software Engineering (EASE), pp. 2002.
[15] N. Nagappan, L. Williams, M. Ferzli, K. Yang, E. Wiebe, C. Miller, and S. Balik, "Improving the CS1 Experience with Pair Programming", Proceedings of SIGCSE 2003, pp. 2003.
[16] N. Nagappan, L. Williams, E. Wiebe, C. Miller, S. Balik, M. Ferzli, and J. Petlick, "Pair Learning: With an Eye Toward Future Success", Proceedings of Extreme Programming/Agile Universe, New Orleans, pp. 2003.
[17] J. Nawrocki and A. Wojciechowski, "Experimental Evaluation of Pair Programming", Proceedings of European Software Control and Metrics (ESCOM 2001), London, England, pp. 2001.
[18] F. Shull, Singer, J., Sjoberg, D.I. (Editors), *Guide to Advanced Empirical Software Engineering*. London: Springer, 2008.
[19] L. Williams and R. Kessler, *Pair Programming Illuminated*. Reading, Massachusetts: Addison Wesley, 2003.
[20] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair-Programming," in *IEEE Software*, 2000, pp. 19-25.
[21] L. Williams, C. McDowell, N. Nagappan, J. Fernald, and L. Werner, "Building Pair Programming Knowledge Through a Family of Experiments", Proceedings of International Symposium on Empirical Software Engineering (ISESE) 2003, Rome, Italy, pp. 143-152, 2003.

APPENDIX

In this appendix, we report on the benefits, problems, attributes of a pair programming partner and attributes of a pair programming team that were mentioned by at least two respondents in the survey.

After the top ten benefits for pair programming are another 18 benefits. Here they are in order of popularity: Better team work, improved productivity and efficiency, socialization, a single individual cannot block progress, maintainability of code, increased work focus, faster work pace, two people can work in parallel, shared code ownership, reliability, potential for self improvement, quick releases, better coding practices, shared work, better architecture, better customer support, high-level low-level balance, and fewer distractions.

The remaining problems that respondents reported with pair programming in order of popularity are that it is hard to adopt pair programming, there is limited office space, it is hard to think with another person working over your shoulder, it requires a commitment from the entire team, partners can be overcritical, it is difficult to get management to agree to let you use program in pairs, and you cannot easily do independent work while pairing. Continuing, some people are afraid of pair programming, they feel a lower sense of ownership over the code, they feel it is not part of the Microsoft culture, pairs can ignore their own interpersonal problems when programming, pairing suffers when there are too many changes in the product cycle, they compete with their partner, and they feel that people should work alone sometimes. The rest of the problems have two respondents associated with them: They worry about an unequal commitment level between the partner, long term pair compatibility, bad documentation, extra management overhead, a passive partner, less creativity, lack of tools, lack of privacy, they find it hard to pair when the team has an odd number of people, less work on upfront design, hard bugs are not found, it is stressful, hard to deal when a partner is absent, people do not know about pair programming, it can be difficult to share driving duties, they cannot multitask, they may need more than two people when working on a problem, they may need to rework some code written by the partner, it is too easy to stop pairing, it requires discipline, sometimes it does not work if the partner is not compatible, and it can deliver fewer features.

There are many other attributes that respondents attribute to a good pair programming partner: critical, objective, detail-oriented, easy to work with, no ego, patient, team player, good at reviewing designs, has the same work hours, fast thinker, committed to quality, likes to switch drivers, self-motivated, respectful, tolerant, good at planning, good problem solver, has acceptable personal hygiene, is a quick typist, thinks about multiple use cases, and is professional. A number of qualities were reported by only two respondents: willing to commit to pair programming, honesty, quick learner, pairing a developer with a program manager, pairing a developer with another developer, they like to swap pairs frequently, diverse, has the same design style, is productive and hard-working, is a good teacher, is creative and punctual.

The rest of the attributes that respondents perceive are important for a good pair programming team to have are shares knowledge, has a similar working style, willing to swap partners, learns from the other, committed to pair programming, critical, tolerant, enjoys pair programming, is a good team leader, conducts focused meetings, has equal knowledge as me, has good skills, conducts

good discussions, cares about metrics, is a problem solver, is a junior developer paired with a senior developer (and vice versa), is productive, has an open workspace, is patient, self-motivated, synergistic, feels strongly about code ownership, is a good match, disciplined, diverse, is on the same wavelength, and has experience. The remaining attributes were given by two respondents: has a customer focus, is willing to swap drivers, is dynamic, trustworthy, reliable, produces maintainable code, is engrained in the Microsoft culture, has matching passion, knows when pair programming is useful, and adheres to coding standards.