Pair Programming Benefits: Claims vs. Evidence

Mårten Jakobsson
Helsinki University of Technology
mjakobss@cc.hut.fi

***Abstract*— Pair programming, also known as collaborative programming, is the agile software development practice, where two programmers jointly develop software side by side on one workstation. Many claims have been made as to the benefits of pair programming, but there is also some resistance, especially from people that have never tried pair programming.**

This paper tries to give a clear and objective picture of pair programming and it's uses, by gathering the claims made about pair programming and matching the claims with evidence found through empirical studies. Information about possible drawbacks with pair programming and situations when pair programming should not be used is also gathered from the empirical studies.

People deciding whether or not to incorporate pair programming into their current software development process can use the results of this study.

***Index Terms*— Pair programming, benefits, drawbacks, claims, evidence.**

1. Introduction

1.1. Background and motivation

PAIR programming is one of the twelve practices of Extreme Programming (XP) and due to the attention XP has gained in the recent years, pair programming has also become quite familiar as a concept.

Although pair programming is mostly known from XP, it is a practice that can be incorporated into any software process and it is therefore interesting to investigate pair programming on its own (Williams, 2001).

Pair programming is performed by two developers sitting side by side at one workstation, writing code jointly, continuously collaborating on the same design, algorithm, code or test. One of the developers is the driver, controlling the keyboard and mouse and writing the code. He is concerned about writing the current method or piece of code correctly, both logically and syntactically. The other developer is the navigator, continuously and actively observing the work done by the driver. Besides watching for possible errors done by the driver, he considers the current work from a more abstract and higher-level viewpoint, planning and thinking about overall design impacts and decisions.

The roles of the driver and navigator should be changed regularly. Both participants are considered equal, regardless of experience level, and they share the ownership of the work product.

Anecdotal evidence regarding pair programming strongly suggests that pair programming is effective and outperforms traditional software development in solo. Pair programmers have been claimed to complete their tasks faster than solo programmers, while generating almost error free code. The claims go even further, saying that the pair programmers enjoy their work more and that pair programming also increases programmer satisfaction and confidence, among other things.

There has been an increasing interest in pair programming during the last few years. Studies have been performed to investigate different aspects of pair programming and the amount of empirical data is increasing all the time.

Still, people that haven't tried the practice are skeptical, since they have a hard time acknowledging that two developers working as a pair can be more productive than two developers working alone. They might feel that the pair would have to complete the assigned tasks in half the time it would take a single developer, in order for the pair programming practice to be beneficial.

Programming has also traditionally been seen as a solo performance and it can sometimes be hard for programmers to give up ownership for their code and share the glory of a successful implementation. Most developers could probably function well as pair programmers, but it requires an open mind and a different mindset, compared to solo development. The pair has to communicate actively and express their ideas and thoughts continuously, which is quite different from the traditional programming style.

In order for pair programming to gain public acceptance, people have to be made aware of the benefits of pair programming and empirical evidence is needed to back up the claims made.

This paper gathers claims made about pair programming and it's benefits. The claims are matched to empirical evidence and possible discrepancies are reported. Problems found during the use the pair programming practice are also registered and reported.

People deciding whether or not to incorporate pair programming into their current software development process can use the results of this study as a base for their decision-making.

1.2. Research problem

The research problem of this study is to find, analyze and organize the claims made about pair programming and to investigate if these claims are supported by empirical evidence. Additionally this study will take notice of problems found when using pair programming and present them to the reader in a compiled form.

The research problem is represented by the two research questions I will try to answer:

- What are the claimed benefits of pair programming and how do they compare to empirical evidence?
- What drawbacks have been found with pair programming?

1.3. Objectives

The objectives of this study are

- to identify the claims made about pair programming,
- to investigate if there is empirical evidence to back up the claims,
- to critically evaluate the literature to find problems or limits in their conclusions and in the way they where performed,
- to suggest areas that need further investigation.

The results can be used both by managers and developers that don't have previous experiences of pair programming, as a base to found their decisions about pair programming on.

1.4. Scope

The scope of this study is limited by the focus on only one agile practice, namely pair programming. I will not include the investigation of distributed pair programming in this study, since the amount of material on the subject is quite limited, the tools needed would have to be investigated more thoroughly and the common use of pair programming involves co-location of the developers.

The literature search was limited to articles written in English from the year 1994 until present time. Since the topic has been investigated actively during this time period this limitation is necessary in order to get up to date and relevant information.

1.5. Method

The study was performed as a literature study, with a classification of the material into to basic groups; material containing claims made about pair programming in previous studies and in other literature, as well as material containing evidence in the form of results from empirical studies. Additionally some literature was used for background, definitions and similar purposes.

Most of the literature used was articles found by searching the following databases: ABI Inform, Proquest Direct, ACM Digital Library, Computer and Information Systems Abstracts, IEEE/IEE Electronic Library, INSPEC.

The queries used when searching the databases were the following: *pair programming, quality* and combinations of these. Based on the results from these queries, other articles were found and used.

1.6. Structure

The structure of this paper is the following: In the next chapter the results of this study are presented in two sections. The first section contains the claimed benefits of pair programming compiled into seven main areas of improvement. The second section lists the empirical studies found and explains the idea of each study and it's results. In chapter three I discuss the results of this investigation, namely how the claims and the empirical evidence match up and potential future investigation possibilities. Last but not least is a chapter with the summary and conclusions of this study and its results.

2. Results

In this chapter the results of this study are presented. The chapter is divided into two sections. The first section contains the claims made about pair programming and the second section contains empirical evidence regarding pair programming.

2.1. Claims

This section identifies the claimed benefits of pair programming made in different literature. The claims have been compiled into seven main areas of improvement that are described in more detail in separate subsections.

Anecdotal evidence suggests that pair programming is more effective than traditional practices. One of the first and most known uses was on Chrysler's Comprehensive Compensation (C3) project, where a failing project was put back on track and completed successfully. Most of the errors that made it through testing after pair programming was incorporated, originated from someone programming alone. (Haungs, 2001)

Up front it might seem as a waste of resources to pair developers two and two for doing development work. Pair programming has nonetheless been used successfully in different contexts and the reason for adopting pair programming is to take advantage of the claimed benefits.

The claimed benefits of pair programming made in different literature have been compiled into seven main areas of improvement, shown in table I. The claimed areas of improvement have one column each. The sources from which the claims have been gathered are presented one at each row. If a source supports a given claim, a *x* is printed in the corresponding column for that source. Otherwise the space is left blank. The different areas are explained in more detail in the following subsections.

2.1.1. Quality: Both the quality of the design and the quality of the code is improved. Since the code is continuously reviewed, the defect rate is kept at a minimum (Cockburn and Williams, 2000; Williams et al., 2000; Williams and Kessler, 2000; Jensen, 2003). Because two minds are always working at the problem at hand, a larger number of alternative solutions will be explored, than a single programmer alone might do and by collaborating the pair might even come up with solutions that neither of them would have found working individually (van Deursen, 2001; Williams and Kessler, 2000). If several solutions to a problem are found, one can assume that the best solution will be selected and this will in turn increase the quality of the product.

2.1.2. Cycle time: Having two minds searching for solutions to a problem also speeds up the process of finding a solution (van Deursen, 2001; Williams and Kessler, 2000). Pair programmers almost always outperform solo programmers. The performance gain can be up to 40% to 50% at best. (Nosek, 1998; Williams et al., 2000) The total amount of time required is still larger than for a solo developer. The two developers would have to work twice as fast as a single developer, in order to spend the same amount of time in total.

TABLE I
CLAIMED AREAS OF IMPROVEMENT WHEN USING PAIR PROGRAMMING

	Quality	Cycle time	Satisfaction and confidence	Motivation, morale and discipline	Trust and teamwork	Knowledge transfer and management	Learning, training and mentoring
Williams and Upchurch (2001b)	x		x		x		x
Williams and Kessler (2002)	x	x		x	x	x	x
Williams (2003)	x	x	x		x	x	x
Marchesi et al. (2002)	x	x	x		x	x	x
Auer and Miller (2002)	x	x		x	x	x	x
Martin Lippert (2002)	x	x				x	x
Williams (2003)	x	x	x		x	x	x
Cunningham and Cunningham (2004)	x			x	x	x	x

2.1.3. Satisfaction and confidence: Solving problems faster and more successfully will give the developers satisfaction and they enjoy the work more. Since the code produced has been reviewed and accepted by two developers, the pair programmers have more confidence in their solutions than solo programmers have. (Williams and Kessler, 2000)

2.1.4. Motivation, morale and discipline: Working side by side with a partner makes the sessions more intense, since the developers don't want to waste each other's time with phone calls, reading email, surfing the web or other unproductive pastimes (Williams and Kessler, 2000; Jensen, 2003). Pair programming will encourage people to follow guidelines and not neglect other tasks, especially under pressure (Beck, 2000; Auer and Miller, 2002). Pair programmers are happier programmers and this helps job retention because employees who are having fun are less likely to leave (Williams and Kessler, 2002).

2.1.5. Trust and teamwork: Most people that have tried pair programming feel that working in pairs is more enjoyable than working alone. The people also learn to work together and communicate better, which is good for the team spirit. (Williams and Upchurch, 2001b,a) Since the pairing is rotated, everybody on the team get to know each other.

2.1.6. Knowledge transfer and management: Since a pair of two developers writes all code, none of the project team members have critical information regarding the project and thereby pose a threat to the project (Williams et al., 2000; Cockburn and Williams, 2000; Srinivasa and Ganesan, 2002). New developers and project members can be made productive faster, by pairing them with more experienced programmers (Cockburn and Williams, 2000; Srinivasa and Ganesan, 2002; Williams et al., 2003).

2.1.7. Learning, training and mentoring: When working with a pair, knowledge is continuously transferred, both more fundamental information related to good coding practices and design methodologies, as well as small tips and tricks, that usually cannot be taught through formal training methods (Srinivasa and Ganesan, 2002; Canfora et al., 2003; Wood and Kleb, 2003).

As can be seen from table I, two claims are supported by all sources presented, namely the claim regarding improvements in quality as well as the claim regarding learning, training and mentoring. The claim concerning trust and teamwork, the claim concerning knowledge transfer and management and the claim concerning cycle time have all been supported by most of the presented sources. The claim about increased satisfaction and confidence and the claim about increased motivation, morale and discipline have only gotten limited support.

2.2. Empirical Evidence

This section contains the empirical evidence found regarding pair programming. Each study is presented in detail to show the areas covered, the results found and the impact and reliability of the study in the context of this investigation.

The last few years a number of experiments and investigations focusing on pair programming have been performed. During the literature search eleven papers where found that reported about empirical studies. Of these five where done in purely educational environments, based on university experiments, four were performed in purely industrial environments with professional developers and two combined results from both educational and industrial environments.

A summary of the papers is given in table II. The table presents the main characteristics of the studies. These characteristics are then used to calculate the importance of each of the studies for the purpose of this investigation.

The different characteristics and their influence on this study will be discussed in the following.

The studies belong to one of two types, namely empirical or case study. The empirical studies are more scientific in the way they are performed. One of the goals, or the main goal, is to investigate pair programming and this goal along with the used

TABLE II

EMPIRICAL STUDIES INVESTIGATING PAIR PROGRAMMING

Study	Type	Participants	Environment	Assignment	Other	Importance
Nosek (1998)	Empirical	15 (5 pairs, 5 solo)	Industrial	Challenging problem, important to their organization	Maximum time allowed to solve the problem was 45 minutes	High
Williams et al. (2000)	Empirical	41 (14 pairs, 13 solo)	Educational and industrial	School projects	Undergraduate students. An online survey of professionals incorporated	High
Cockburn and Williams (2000)	Empirical	98 (33 pairs, 32 solo)	Educational and industrial	School projects in the educational studies and a challenging problem, important to their organization in the industrial case	Data from three empirical studies are analyzed, two of which are Nosek (1998) and Williams et al. (2000)	High
Gehringer (2003)	Empirical	96 (at most about 30 pairs, the rest solo)	Educational	Simulation of three microprocessor parts	Senior/masters-level course	High
Jensen (2003)	Case study	10 (5 pairs)	Industrial	Multitasking real-time system executive		Medium
Lui and Chan (2003)	Empirical	15 (5 pairs, 5 solo)	Industrial	Solutions to deduction problems and procedural algorithms		High
McDowell et al. (2003)	Empirical	555 (202 pairs, 148 solo, 3 unaccounted)	Educational	School projects	Introductory course	Low
McDowell et al. (2003)	Empirical	216 (58 pairs, 100 solo)	Educational	School projects	Data from three studies presented, with seniors, juniors and sophomores	Low
Nagappan et al. (2003)	Empirical	495 (162 in pairs, 171 solo)	Educational	School projects	Freshmen and sophomores	Low
Williams et al. (2003)	Empirical	1215 (401.5 pairs[a], 412 solo)	Educational	School projects	Introductory courses	Low
Wood and Kleb (2003)	Empirical	2 (1 pair)	Industrial	Software test bed for evaluating the performance of a numerical scheme to solve a model advection-diffusion problem	XP was evaluated, with pair programming reviewed	Medium

[a]There where no actual half pairs. Some ”pairs” consisted of three people in order to manage uneven number of pairers.

metrics are defined before the study is performed. The study should be repeatable. The case studies are usually papers from industrial settings, where the achievements are analyzed and reported on afterwards. The amount of background information is detailed enough to give the reader a good picture of the environment, when evaluating the credibility and consistency of the results.

The amount of participants is presented, since this affects the reliability of the results. The larger the amount of participants is, the more credible the results, from a statistical point of view.

The environment in which the study is performed is industrial, educational or a combination of both. An industrial study is performed in an industrial environment with professional developers. An educational study on the other hand is performed in a university setting with students as participants. The studies that are both industrial and educational have some parts of the study performed in one environment and other parts performed in the other. The division into industrial and educational environments is important when assessing the applicability of the results in other contexts. The results from for example an educational study examining freshmen or sophomores cannot be directly translated into an industrial environment. In the educational setting the experience level of the students vary, since some courses are introductory and others more advanced, and this is also taken into account when evaluating the studies.

The assignments performed in the studies are given in order to clarify the scale and importance of the work produced. Additionally other information relevant to the assessment of the study is given in a separate column.

The importance of a study from the viewpoint of this investigation is given on the scale high, medium and low. The value is calculated as follows. All studies start out with the rating medium. If the study is an empirical study it moves up to high. Case studies don't change. If the environment is industrial, the rating moves up a notch from the current position. If the environment is educational and freshmen or sophomores have been studied, the rating is decreased, since many of the lessons learned in that context, will not hold in the general case. The last factor influencing the rating is the number of participants. The possible values are divided into three intervals that span from two to 25, 26 - 100 and more than 101. The basic idea is that the more participants the better and therefore belonging the first group will decrease the importance rating and belonging to the last group will increase the rating. Due to the importance of the applicability of the results, however, the environment rating is also considered, so that if the environment caused a decrease in the importance rating, then a large amount of participants will also decrease the rating one step instead of increasing the rating.

One of the first empirical studies in an industry setting was reported by Nosek (1998). The participants were 15 full-time system programmers. Five developers worked solo and 10 developers worked in 5 pairs. All of them worked on the same problem, a challenging problem important to their organization. The work was performed in their own environments and with their own equipment. The maximum time allowed for solving the problem was 45 minutes. The results of the study show that the pairs produced more readable and functional solutions and that the programmers working in pairs had greater confidence in their work and enjoyed the process more. It also showed that more experienced programmers performed better, than less experienced programmers. Although the average completion time for pair programming groups was smaller than for individual developers, the result was not statistically significant. It was stated that pair programming is beneficial for speeding up the development rate and decreasing time to market, as well as for improving the quality of the software.

The empirical paper by Williams et al. (2000) explains an experiment in a university setting with 41 participating students (13 individuals and 14 pairs) working on the same problem. The investigation is complemented with an online survey of professional developers. The results show that the pairs passed more test cases than the individual developers and that the difference is statistically significant. The pairs completed their programs 40% to 50% faster than the individuals. A vast majority of the students and the participants in the online survey stated that they enjoyed working in pairs more than working alone and that they were more confident in their solutions. There are a few things in the execution of this experiment that decreases the reliability of the results. During the university experiment, the division into pairers and solo developers was done based on an opinion survey at the beginning of the course. In an industry environment this is not possible however, since everybody either follows the practice or nobody does. The problematic people are seldom the ones that are interested and motivated to make the pair programming practice work and therefore the results of this study might not be accurate, when considering developers in general. Since the participants in the university experiment were students, all the results are probably not directly transferable to an industrial environment.

The empirical paper by Cockburn and Williams (2000) investigated further the results from Nosek (1998), Williams et al. (2000) and one other study in order to further explain why pair programming is beneficial. The first study was performed in and industrial setting, as noted earlier and the two other studies were performed in educational environments. The total amount of participants was 98, divided into 33 pairs and 32 solo developers. The results show that many mistakes are noticed and corrected during the coding sessions. The defect rate is therefore very low and the number of defects found during quality assurance or in the field is kept small. It was shown that the designs used by pair programmers are better and the resulting code is shorter. The reported that problems are solved faster when using pair programming and that knowledge about the system and software development in general is transferred more effectively and as a consequence multiple people understand each piece of the system. The also reported that pair programmers learn to work together and communicate better, which is good for the team spirit and that pair programmers enjoy their work more. The paper also points out that the required increase in total time due to these benefits was only 15%, not 100% as might be expected and that the extra cost should be repaid in shorter and less

expensive testing, quality assurance and field support. Since this paper was based partly on the results by Williams et al. (2000), the critique for that paper also applies here.

The empirical paper by Gehringer (2003) investigates the use of pair programming in a course that used, but didn't teach, programming skills. The course contained three projects for creating simulations of three microprocessor parts. All 96 students of the class were surveyed after the end of the semester and 59 of them responded. The results show that a majority of the students that had pair programmed liked the pair programming experience and that the cooperation and communication with the other team member was good. The participants complained, however, that scheduling was a problem in the university environment. A large amount of students paired for the first project, but the amount decreased for the following two projects. Besides the scheduling problems, it was believed that the students didn't perceive the benefits of pair programming to be large enough once the computing environment was familiar. The last two projects were also easier than the first one, which was believed to have influenced the decision. Due to the educational environment the study was performed in, the results cannot be directly transferred to an industrial environment. Especially the scheduling problems should not occur in a professional setting, where people work full-time at the same location.

In the case study by Jensen (2003), industry experiences regarding pair programming are reported. The project team consisted of 10 programmers (5 pairs) with expertise in different areas and one manager. The team developed a 50000-line multitasking real-time system from scratch. The results from the experiment are compared to historical data from the company and show that the productivity measured as lines / person-month increased 127% and the error rate dropped to one thousandth of the normal value, which is quite remarkable. During the project they experienced that it was counter-productive to pair developers with the same experience level, since they didn't function smoothly together. Pairing developers with different experience levels usually worked better and they experienced the benefits of on-the-job training. The pairs were put in two person cubicles and they found that this limited the communication between pairs. The study also discussed the importance of having open-minded managers that are supportive of the pair programming process.

The empirical study by Lui and Chan (2003) investigates when a pair of programmers can outperform two individuals. Fifteen professional programmers were tested. The division between pairers and non-pairers changed during the test, but both the number of pairs and solo developers was always five. The assignments consisted of finding solutions to deduction problems and procedural algorithms. The results show that the pairs excel in procedural problems and deduction questions and are therefore more useful when writing more challenging programs with critical design issues. They also found that the pairs, due to better problem learning and exploring of larger solution spaces, solved new problems faster. This benefit is not as noticeable when the pair solves problems they are familiar with.

The empirical paper by McDowell et al. (2003) investigates the impact of pair programming on student performance, perception and persistence in an introductory computer science course. The study had a total of 555 participants. The number of pairs was 202 and the number of solo developers was 148. The status of three students was not known. The course was performed in four sections and had a compulsory programming project. The results show that a significantly larger amount of pair programmers passed the course and that the pair programmers achieved higher grades for their projects, than individual programmers. The pairs also got more satisfaction from their work, they showed a higher confidence in their work and they enjoyed the work more than individuals. The results also show that pair programming did not negatively affect the grades in the final exams. A larger percentage of individually working students also dropped out of the course than paired students. The participants were allowed to suggest whom they wanted to pair with at the start of the course. During the course the pairs were not rotated, with the exception of a few cases where schedule changes and drops made reassignments necessary. Regarding the reliability if the results, the pairing scheme may have had an effect on the results, since people that know each other from before a more likely to get along. Because this was an introductory course, the participants were novice programmers and this limits the applicability of these results in other contexts quite a lot.

The empirical study by McDowell et al. (2003) combined the results from three studies at UCSC with students participating in computer science courses containing a programming project. The total number of participants was 216, divided into 58 pairs and 100 solo developers. The participants ranged from sophomores to juniors and seniors. The results show that pair programmers get higher scores for their projects on average, but the score in the final exam was not significantly affected by the use of pair programming. The study did however not show that the pair programmers completed the assignments faster than solo programmers. Accurate time tracking was not required and this is believed to be one of the reasons for this finding. They also found that students that claimed to be strong programmers liked pair programming the least. Since the study was performed in an educational environment, partly using quite inexperienced programmers, the results cannot be directly translated to an industrial context.

The efficacy of pair programming in several introductory computer science courses was investigated in the empirical paper by Nagappan et al. (2003). The number of participants was 495, divided into 162 pairs and 171 individuals. The results show that pair programming students performed better in many projects and that pair programming reduced the burden of the class and that the attitude towards collaboration was more positive among the pairs than among the solo students. Pair programming was not deterrent to student performance and it helped the retention of students in computer science. There were some incompatible pairs. The pairs also had to monitored in order to avoid one dominating and the other carrying the whole burden. Since this was an introductory course, the participants were novice programmers and this limits the applicability of these results in other contexts quite a lot.

In the study by Williams et al. (2003) the required time and cost of using pair programming was studied by investigating empirical data from several courses at UCSC and NCSU. The total number of participants was 1215 for these studies. The number of solo developers was 412 and the rest worked in pairs, theoretically forming 401.5. There were no actual half pairs of course, but sometimes the number of pairers was uneven. Letting some groups contain three students solved this problem. The results show that a greater percentage of pair programmers pass the course than solo programmers and the pairs produced better project results. The pairs performed at least equally well as solo students on the exams. The paired students had a more positive attitude towards pair programming and they stated afterwards that they enjoyed the pair programming experience. It was also shown that using pair programming in a course did not affect performance in further courses, but pair programming students were more likely to pursue a computer science related major. Since these courses were introductory courses, the participants were novice programmers and this once again limits the applicability of these results in other contexts.

In the paper by Wood and Kleb (2003), Extreme Programming is explored for scientific research in an industrial environment. The purpose of the project was to deliver a software test bed for evaluating the performance of a numerical scheme to solve a model advection-diffusion problem. The team size was only two, so pair programming seemed be a bad fit at first, but as it turned out, it worked well. The results show that the performance of the team was good, and the produced code was clean. The size of the program was smaller than for previous similar projects and the programming sessions were more intense, due to pair pressure. They also noticed some cross-fertilization of tips and tricks. They implemented functionality at the historical rate, but also supplied an equal amount of supporting test, which had not been done before. The reliability of the results is quite poor. The number of developers was small and since the study explored the use of Extreme Programming, some of the benefits reported may also partly be due to the other XP practices and this has to be regarded.

When studying and experimenting with pair programming, the results of solo developers working in isolation are quite often compared to the results of pairs. In an industrial environment, however, the developers usually have colleagues and other experts they can turn to, if they have problems (Matt Stephens, 2003). Therefore forcing developers to work totally isolated is not representative of the way work usually is done and will therefore affect the correctness of the results. All the studies presented above that compare solo developers with pairs suffer from this problem.

3. Discussion

In this chapter the claims are matched up with the empirical evidence, to see which claims are backed up by evidence and which areas need more investigation. Additionally drawbacks and possible future investigation areas will be discussed.

In table III all empirical studies are presented, one at each row. The claims made about pair programming are placed as columns and if a study verifies the given claim, an *x* is printed in the corresponding column for that study. If a claim is not supported an *o* is printed and if the claim was not investigated the cell is left empty.

As can be seen in the table, all claims have been verified by some study. The claim concerning quality has been verified by almost all studies and can be considered a true benefit of pair programming. The study by Gehringer (2003) could not statistically verify improvements in quality, but this was believed to stem from the overall high level of the results in the study. Higher product quality is one of the most important benefits of pair programming. Even though pair programming usually requires more resources in total, the time and energy saved by minimizing the error correction efforts and by simplifying further development, will probably make up for the additional cost (Nosek, 1998). This is especially true in mission-critical systems where human lives or large businesses are dependent on the software. It is unclear, however, if this benefit could be achieved by a less personnel intensive approach such as pair inspections (Müller and Tichy, 2001).

The claim regarding improvements in satisfaction and confidence is also verified by most of the studies, especially the important ones, so this claim can also be considered a true benefit of pair programming. Even though an employee's satisfaction and confidence don't directly affect a company, the improvements are beneficial in the long run, since they will increase self-esteem and motivation.

The claim regarding improvements in cycle time has been verified by several important studies, but not by all. The study Nosek (1998) could not verify this claim. The average completion times for pair programming groups where smaller, but the difference was not statistically significant. The study McDowell et al. (2003) could not either verify the claim. The results indicated that pair programmers spent less time on the project development, but the difference was not statistically significant, but since accurate time tracking was not required, the resulting times are not reliable and cannot be taken into account. All in all it seems like pair programming can speed up development in some cases. Still the total amount of time required is always larger than the time required by a single developer and this cost has to be offset by the other advantages, for the practice to be profitable. If the schedule is tight and the time to market critical for the success of the product, this additional cost can well be worthwhile (Nosek, 1998). The big differences in measured speed gains for pair programmers in the empirical studies are a cause of concern, however, and suggest that there are more factors at play. It has been shown that pair programming is more efficient when facing new and complex problems and that the advantage is smaller fore simple repetitive work (Lui and Chan, 2003). It has also been noticed that simple programming tasks that don't require mind crunching are better done alone than in pairs (Müller and Tichy, 2001; Williams and Kessler, 2000). These are interesting discoveries and areas that should be investigated further, in order to find out what kind of work should be done in pairs and what kind of work is more effectively done alone.

The claim regarding learning, training and mentoring has also been verified by several important studies and can be

TABLE III
CLAIMED AREAS OF IMPROVEMENT SUPPORTED BY THE EMPIRICAL STUDIES

	Importance	Quality	Cycle time	Satisfaction and confidence	Motivation, morale and discipline	Trust and teamwork	Knowledge transfer and management	Learning, training and mentoring
Nosek (1998)	High	x	o	x				
Cockburn and Williams (2000)	High	x	x	x		x	x	x
Williams et al. (2000)	High	x	x	x	x			
Gehringer (2003)	High	o		x		x	x	
Jensen (2003)	Medium	x	x		x			
Lui and Chan (2003)	High	x	x				x	x
McDowell et al. (2003)	Low	x		x				
McDowell et al. (2003)	Low	x	o					
Nagappan et al. (2003)	Low	x						x
Williams et al. (2003)	Low	x		x				
Wood and Kleb (2003)	Medium	x	x	x	x		x	x

considered valid in most cases. Still, is can also be irritating for the mentor to get interrupted all the time and this has to be considered.

The claim regarding improvements in knowledge transfer and management has also been verified by several important studies, mostly industrial. The reason why this claim hasn't been verified in many educational environments is probably that it cannot easily be done in that context. One or two students do the school projects once and the projects have a very short lifespan. Therefore the knowledge transfer and management aspect is not needed. All the industrial studies presented that verify this claim have had only a limited amount of participants, so it is unclear how well this benefit will scale for larger numbers of developers. From a professional point of view, the knowledge sharing and transferring from one project team member to another is important, in order for the team to not be too dependent on a single developer. Therefore this area deserves more investigation.

The claim regarding motivation, morale and discipline as well as the claim regarding trust and teamwork have only been verified by a couple of studies. The claims have not been investigated very actively and the reason for this is unclear. It might be that the claims are considered self-evident or thay they are considered to be of lesser importance at this time.

Only a few areas of concern were reported in the studies. In one study it was reported that in a university setting it could be hard to schedule the pair programming sessions, when the team members didn't always have compatible timetables (Gehringer, 2003). In another case the pairs had to be monitored to avoid one dominating the other and to make sure the development burden was distributed evenly (Nagappan et al., 2003). Two studies reported that there had been some incompatible pairs that had to be dealt with (Jensen, 2003; Nagappan et al., 2003). In one of the studies the pair programmers were placed in two person cubicles and it was noted that this limited the communication between pairs. As an afterthought they realized that an open space would probably have been even more productive. (Jensen, 2003)

The amount of problems reported in the studies is small. The reason for this may be that the pair programming practice works very well. It may also be that the authors are reluctant to bring forth negative aspects too strongly, since this could affect the general opinion about pair programming negatively. Still the drawbacks of the practice are worth noticing. It is not acceptable to have people suffer due to bad personal chemistry and forcing people to work in a way that doesn't suit them.

On the whole, the results thus far look promising and pair programming will probably be beneficial to many projects.

Many of the studies examined where performed in a university setting. Although the results from university experiments are valuable, they can't always be transferred to industry settings as is. The surrounding factors, background and experience of the participants and so on usually differ somewhat.

Many studies that investigate the performance of pair programming conclude that pair programming requires more developer resources, but that this is compensated due to the smaller defect rate and the better design accomplished (Nosek, 1998; Williams et al., 2000; Cockburn and Williams, 2000). They ask "Can an a partner ever be of any value?", when the proper questions should be "Is a partner of sufficient value to justify the cost?" and "Does the partner cause additional problems?" (Matt Stephens, 2003). At the moment there are no studies that have shown the true value of these code improvements, or if they could be achieved with smaller resource requirements or inconveniences. In order to make a good decision found on solid evidence regarding the use of pair

programming, the true value of the pair programming benefits will be needed. This includes the total cost of developing a software product using pair programming and the total cost of developing an equivalent software product using only solo programmers. This calculation should also include the cost caused by bug fixes and other alterations needed, in order to get the software product to the same quality level as the one developed with pair programming. Errors in the developed systems will have to be corrected at one point or another anyway.

It has also been shown that the variations in productivity between the most productive and the least productive programmers can be as large as 200:1 (Bryan, 1994). Additionally the cognitive ability of the individual team members, faithfulness to the methodology, different types of conflicts and the way the conflicts are handled are all significant to development success (Domino et al., 2003). More thorough investigations into how these and other factors impact the productivity of pair programming pairs is needed in order to take full advantage of the pair programming practice.

Pair programming does bring some benefits, as have been shown. The successfull implementation of the practice many times depends on the people involved and their mindset. A good side of the practice is that you don't have to convert everybody at once, but you can try it out on a few developers or use it only for certain challenging tasks. If the practice works out well, the usage level can be increased. Developers don't have to change their whole way of doing work, since pair programming can be incorporated into any current software development process (Williams, 2001).

4. Summary and Conclusions

This paper has identified the claims made about pair programming. It has found and critically evaluated empirical studies regarding pair programming and it has matched the claims to the empirical data in order to investigate if the claims are backed up. It has also pointed out areas that need more investigation.

In a time when time to market is getting more and more important, without sacrificing quality, effective ways of producing high-quality software are needed. The pair programming practice claims to deliver these benefits and more. Several studies have been performed to investigate and measure the benefits of pair programming and today there is empirical evidence to back up at least the most important claims of higher quality in shorter time.

This paper has shown that the quality of the final product will be better when using pair programming and that the programmers will have more confidence in their solution and feel a higher level of satisfaction. It has also been shown that the cycle time will decrease in most cases, decreasing time to market.

Based on the results found, pair programming can be highly recommended for increasing the overall productivity in projects, where high quality and time to market is critical for the success of the project. The pair programming practice can also be used in other circumstances with good results, but the advantages might not necessarily be as big. Even though this study only focuses on pair programming, without considering other software development practices, the recommendation is valid, since pair programming can be incorporated into any current software development process.

There are however still a number of areas that should be investigated further, especially in industry settings:

- How effective is the knowledge transfer when pair programming?
- Does the benefits of pair programming scale to larger project teams?
- Is pair programming the most beneficial practice for all kinds of development work?
- How does different pairing strategies impact the productivity of a pair programming pair? What kind of persons cannot and should not participate in pair programming?
- Could the main benefit of pair programming, higher quality, be achieved by a less personnel intensive approach such as pair inspections?
- What are the real costs of producing software products with equal quality using traditional software development practices compared to using pair programming? This investigation should include the costs caused by bug fixes and other alterations to the system in order to get products of equal quality.

References

Auer, K. and R. Miller (2002). *Extreme programming applied: playing to win*. Addison-Wesley Longman Publishing Co., Inc.

Beck, K. (2000). *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc.

Bryan, G. E. (1994). Not all programmers are created equal. In *Aerospace Applications Conference, 1994. Proceedings., 1994 IEEE*, pp. 55–62. GEB Software, Pacific Palisades, CA, USA: IEEE.

Canfora, G., A. Cimitile, and C. A. Visaggio (2003). Lessons learned about distributed pair programming: what are the knowledge needs to address? In *Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Los Alamitos, CA, USA, pp. 314–319. Res. Centre on Software Technol., Sannio Univ., Benevento, Italy: IEEE Comput. Soc.

Cockburn, A. and L. A. Williams (2000). The costs and benefits of pair programming. In *Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2000)*, Cagliari, Sardinia, Italy. Addison-Wesley.

Cunningham, W. and K. Cunningham (2004). Pair programming benefits. http://www.c2.com/cgi/wiki?PairProgrammingBenefits.

Domino, M. A., R. W. Collins, A. R. Hevner, and C. F. Cohen (2003). Conflict in collaborative software development. In *Proceedings of the 2003 SIGMIS conference on Freedom in Philadelphia: leveraging differences and diversity in the IT workforce*, Philadelphia, Pennsylvania, pp. 44–51. University of South Florida, Tampa, FL: ACM.

Gehringer, E. F. (2003). A pair-programming experiment in a non-programming course. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Anaheim, CA, USA, pp. 187–190. North Carolina State University, Raleigh, NC: ACM.

Haungs, J. (2001). Pair programming on the C3 project. *Computer 34*(2), 118–119.

Jensen, R. W. (2003). A pair programming experience. *CrossTalk, The Journal of Defense Software Engineering 16*(3), 22–24.

Lui, K. M. and K. C. Chan (2003). When does a pair outperform two individuals? In *Lecture Notes in Computer Science*, Volume 2675, pp. 225–233. Springer-Verlag Heidelberg.

Marchesi, M., G. Succi, D. Wells, and L. Williams (2002). *Extreme Programming Perspectives*. Addison Wesley.

Martin Lippert, Stefan Roock, H. W. (2002). *eXtreme Programming in Action: Practical Experiences from Real World Projects*. John Wiley & Sons.

Matt Stephens, D. R. (2003). *Extreme Programming Refactored: The Case Against XP*. Apress.

McDowell, C., B. Hanks, and L. Werner (2003). Experimenting with pair programming in the classroom. In *ACM SIGCSE Bulletin , Proceedings of the 8th annual conference on Innovation and technology in computer science education*, Volume 35.3, Thessaloniki, Greece, pp. 60–64. University of California, Santa Cruz, CA: ACM.

McDowell, C., L. Werner, H. E. Bullock, and J. Fernald (2003). The impact of pair programming on student performance, perception and persistence. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, Piscataway, NJ, pp. 602–607. Comput. Sci. Dept., California Univ., Santa Cruz, CA, USA: IEEE Computer Society.

Müller, M. M. and W. F. Tichy (2001). Case study: Extreme programming in a university environment. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001.*, Los Alamitos, CA, USA, pp. 537–544. Dept. of Comput. Sci., Karlsruhe Univ., Germany: IEEE Computer Society.

Nagappan, N., L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller, and S. Balik (2003). Improving the CS1 experience with pair programming. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, Volume 35.1, Reno, Navada, USA, pp. 359–362. North Carolina State University, Raleigh, NC: ACM.

Nosek, J. T. (1998). The case for collaborative programming. *Communications of the ACM 41*(3), 105–108.

Srinivasa, G. and P. Ganesan (2002). Pair programming: addressing key process areas of the people-cmm. In D. Wells and L. A. Williams (Eds.), *Extreme Programming and Agile Methods - XP/Agile Universe 2002. Second XP Universe and First Agile Universe Conference. Proceedings Lecture Notes in Computer Science*, Volume 2418, Berlin, Germany, pp. 221–230. Dept. of Comput. Sci., North Carolina State Univ., Raleigh, NC, USA: Springer-Verlag.

van Deursen, A. (2001). Program comprehension risks and opportunities in extreme programming. In E. Burd, P. H. Aiken, and R. Koschke (Eds.), *Proceedings Eighth Working Conference on Reverse Engineering*, Los Alamitos, CA, USA, pp. 176–185. CWI, Amsterdam, Netherlands: IEEE Computer Society.

Williams, L. (2003). XP practices ... or best practices? An examination of the XP practices. "http://sunset.usc.edu/events/2003/March_2003/Agile_Methods_Perspective_ARR_Laurie_Williams.pdf".

Williams, L., R. R. Kessler, W. Cunningham, and R. Jeffries (2000). Strengthening the case for pair programming. *IEEE Software 17*(4), 19–25.

Williams, L. A. (2001). Integrating pair programming into a software development process. In D. Ramsey, P. Bourque, and R. Dupuis (Eds.), *Proceedings 14th Conference on Software Engineering Education and Training 'In search of a software engineering profession" Cat. No.PR01059*, Los Alamitos, CA, USA, pp. 27–36. Dept. of Comput. Sci., North Carolina State Univ., Raleigh, NC, USA: IEEE Computer Society.

Williams, L. A. and R. R. Kessler (2000). All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM 43*(5), 108–114.

Williams, L. A. and R. R. Kessler (2002). *Pair Programming Illuminated*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Williams, L. A., C. McDowell, N. Nagappan, J. Fernald, and L. Werner (2003). Building pair programming knowledge through a family of experiments. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, Los Alamitos, CA, USA, pp. 143–152. Dept. of Comput. Sci., North Carolina State Univ., Raleigh, NC, USA: IEEE Computer Society.

Williams, L. A., A. Shukla, and A. Anton (2003). Pair programming and the factors affecting brooks' law. Technical Report TR-2003-04, Department of Computer Science, North Carolina State University.

Williams, L. A. and R. L. Upchurch (2001a). Extreme programming for software engineering education? In *31st Annual Frontiers in Education Conference. Impact on Engineering and Science Education. Conference Proceedings. Cat. No.01CH37193.*, Volume 1, Piscataway, NJ, USA, pp. 12–17. Dept. of Comput. Sci., North Carolina State Univ., Raleigh, NC, USA: IEEE.

Williams, L. A. and R. L. Upchurch (2001b). In support of student pair programming. In *Proceeding of the Thirty-second SIGCSE Technical Symposium on Computer Sciense Education (SIGCSE-01)*, Volume 33.1 of *ACM Sigcse Bulletin*, New York, pp. 327–331. Dept. of Comput. Sci., North Carolina State Univ., Raleigh, NC, USA: ACM.

Wood, W. A. and W. L. Kleb (2003). Exploring xp for scientific research. *IEEE Software 20*(3), 30–36.