Investigating the Effectiveness of Pair Programming in Industrial Domains: A Systematic Literature Review

Alex Pawelczyk
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Abstract

This research presents a systematic literature review (SLR) of empirical studies that focus on the effectiveness of pair programming (PP) in industrial domains. The primary objective of the SLR is to present the current research and evidence relative to the effectiveness of PP when applied in industry. The SLR also recognizes how PP effectiveness is measured, the degree of effectiveness that PP has in industrial domains, and factors that impact the effectiveness of PP. This study is based on a comprehensive review of a set of 33 research papers that have been extracted from five major online databases. The SLR highlights six measurements of PP effectiveness (e.g., software quality, knowledge transfer, productivity, effort, job satisfaction, and project cost) and five key factors that can influence PP effectiveness (e.g., expertise level, personality, task complexity, driver-navigator interactions, and work environment). The results suggest a general agreement that when applied appropriately, PP can increase software quality and knowledge transfer. Moreover, the evidence suggests that PP is most effective when used for high-complexity tasks and pairs are composed of individuals with complementary skills, personalities, and knowledge areas. Recommendations for practitioners include viewing PP as an investment into better software quality, using PP strategically, and adopting a team-oriented culture.

Keywords: pair programming, systematic literature review

ACM Reference Format:
Alex Pawelczyk. 2020. Investigating the Effectiveness of Pair Programming in Industrial Domains: A Systematic Literature Review. In *CS846: Advanced Topics in Software Engineering, Aug. 7, Waterloo, ON*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CS846: Advanced Topics in Software Engineering, Aug. 7, 2020, Waterloo, ON
© 2020 Association for Computing Machinery.
https://doi.org/10.1145/1122445.1122456

1 Introduction

Pair programming (PP) is one of the twelve core practices of eXtreme Programming (XP) [3] that is commonly applied or recommended for use in conjunction with many other Agile software development methods, including Test Driven Development, Scrum, Feature Driven Development, Crystal, Lean Software Development, and Dynamic Systems Development Method [1]. PP refers to two programmers working together at one computer and actively collaborating on the same design, algorithm, code, or test [66]. One of the programmers in the pair, called the *driver*, is responsible for controlling the keyboard and implementing the code or design. The other programmer in the pair, called the *navigator*, observes the work of the driver, looks for tactical or strategic defects, and offers ideas for solving a problem. Examples of tactical defects include syntax errors, typos, or calling the wrong function. Strategic defects (i.e., logic errors) occur when the driver is implementing code that will not accomplish the target objective. During a PP session, the driver and the navigator actively communicate and periodically switch their roles [66].

Over the years, PP has been an extensively researched topic in the software engineering community. Many people argue that PP can improve the software development process from a variety of different perspectives, and a large number of empirical studies have been conducted to determine if these claims are true. Some of the benefits of using PP in the software development process include:

- improving software quality [2, 6, 7, 10, 32, 33, 35, 42, 54, 55, 60, 62, 67]
- increasing worker productivity [25, 29, 30, 36, 67]
- shortening time to market [10, 17, 23, 39, 42, 43, 67]
- improving knowledge transfer and communication among teammates [10, 29, 32, 62, 66]
- increasing job satisfaction [58]
- easing the integration of new developers to the project and reducing training costs [20, 68]

However, not all studies report positive effects of PP. Experimental results from the Nawrocki and Wojciechowski study [38] show no positive effects of PP with respect to time taken to complete a task and no improved functional

correctness of the software compared with individual development. Consequently, the results suggest doubled costs without increased quality for the pairs compared with the individuals. Stephens et. al [57] claim that social dynamics, a lack of privacy, a lack of quiet thinking time, and ergonomic issues can lead to problems with PP. Other problems with PP arise from an increase in effort and mental exhaustiveness of the practice [2, 10, 38, 62, 66].

One of the problems with PP empirical studies stems from the difficulty of generalizing their results beyond their experimental environments and into professional software industry. For one, most of the experiments are conducted in educational settings where students are used as experimental subjects. The educational environment may not provide an accurate representation of the professional software industry where it is common for a team of developers to collaborate on various large-scale projects [12]. Moreover, many experiments are conducted within a small time-frame (e.g., a couple hours or a day) where subjects are asked to implement small programs or make small changes to existing programs. Since commercial applications may take multiple years to develop, generalizing the results from these experiments to the industrial domain is not feasible.

An essential component of PP research is gaining an understanding of its effects when applied to industrial settings. Managers of software development firms are hesitant to adopt new practices unless there is reliable evidence that points to the success of a particular technique. Thus, the objective of this paper is to present the current state of the art research and evidence relative to the effectiveness PP for developing commercial applications in industrial domains. The primary contribution of this paper is a systematic literature review (SLR) of empirical studies that focus on the use of PP in industrial settings and investigate the effectiveness of PP as it relates to solo programming (SP). Additionally, the SLR presents any conflicting findings from the analysis, identifies gaps in the existing body of knowledge, and discusses the implications of these results for managers and executives of software development companies.

The rest of the paper is organized as follows. Section 2 summarizes the results of past SLRs on PP. Section 3 outlines the review method of the SLR. Section 4 reports the results of the SLR based on a synthesis of evidence from a collection of empirical studies. Section 5 formulates advice for practitioners based on the key findings and implications of the SLR. Section 6 highlights the limitations of this work. Section 7 identifies areas for future work in PP research, and Section 8 presents conclusions from the review.

2 Related Work

Salleh et. al [48] conducted an SLR that presents evidence relevant to the effectiveness of PP as a pedagogical tool in higher education CS/SE courses. Along with investigating factors that impact the effectiveness of PP for educational purposes, the authors identify evidence regarding factors that impact pair compatibility and investigate which pairing configurations are considered as most effective. Salleh et. al also investigate how PP effectiveness and software quality were measured in the studies that they reviewed. After synthesizing evidence from 74 studies, the authors identified 14 compatibility factors that can affect the effectiveness of PP as a pedagogical tool. The results also show that student skill level is the factor that affects the effectiveness of PP the most. The most common metric for measuring PP effectiveness was time spent on programming, and the most commonly applied metrics for software quality were the number of passing test cases, academic performance, and expert opinion.

Dyba et. al [17] conducted an SLR to determine if existing empirical evidence substantiates the claims that PP is more beneficial than SP. They evaluated 15 studies that compared the effects of PP and SP, where four were conducted with professionals and 11 with students. Their results found a general agreement that PP leads to increased software quality, but there were contradictory results regarding time to market and effort. Out of eleven studies that investigated the effect of PP on time to market, two studies show a negative effect, while the remaining nine show a positive effect. Regarding effort, all but one study show a negative effect (i.e., working in pairs requires more person-hours to develop the same software).

After a thorough search, an SLR could not be found that solely focuses on professional software developers in industrial domains and investigates the effectiveness of PP as compared to SP in this setting. Managers and executives of software development firms are more likely to adopt the practice of PP if there is a strong body of evidence that suggests PP is more effective than current business practices, such as SP. Although studies conducted with students in educational domains provide valuable insights into the effectiveness of PP, their results may not generalize well to industrial domains. Thus, this work presents an SLR that examines the use of PP by professional software developers in industrial settings and discusses the implications of this evidence. The goal of this work is to summarize evidence from a multitude of studies and formulate advice for practitioners based on the evidence.

3 The Review Method

An SLR is a method for identifying, evaluating, and interpreting all available research relevant to a particular research question, topic area, or phenomenon of interest [28]. The primary motivations for conducting the SLR presented in this work are to summarize the existing evidence relative to the effectiveness of PP in industrial domains, identify any gaps in current research, suggest areas for further research, and provide a framework that appropriately positions new

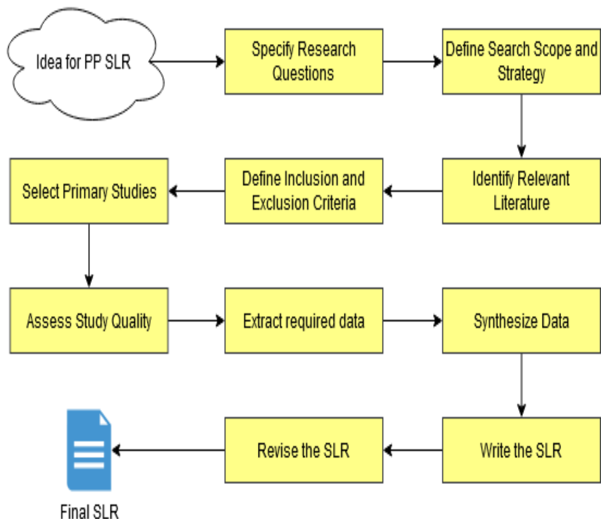

Figure 1. Overview of the SLR review protocol (adapted from [5, 65]).

research activities. To help fulfill these motivations, the SLR follows the guidelines that are defined by Kitchenham [28]. Figure 1 visualizes the steps involved in conducting the PP SLR.

3.1 Research Questions

Table 1 shows the *Population, Intervention, Comparison, Outcomes,* and *Context* (PICOC) structure of the research questions that guide this work. This SLR focuses on empirical studies that investigate the effectiveness of PP for developing commercial software applications in industrial domains.

The primary objective of this SLR is to investigate if PP performed with professional software developers in an industrial setting is more beneficial than SP. Contrary to educational domains where one of the motivations for using PP is to enhance the student learning experience, using PP in industry is driven by economic gains, such as faster time to market, lower development effort, and improved software quality [48]. Thus, the research questions that guide this SLR are as follows:

- **RQ1:** What evidence exists of PP studies conducted in industrial domains that investigate the effectiveness of PP for developing commercial software applications?
- **RQ2:** What methods are used to measure the effectiveness of PP when applied in industrial domains?
- **RQ3:** How effective is PP for developing commercial software applications in industrial domains?
- **RQ4:** What factors influence the effectiveness of PP in industrial domains?

RQ1 is motivated by the need to identify studies that report both successful and unsuccessful applications of PP in industrial settings. RQ2 assesses the metrics that are used in various studies to measure the effectiveness of PP. Different papers may have different methods for measuring PP effectiveness. Thus, RQ2 aims to eliminate ambiguity and specifies how these methods are measured. RQ3 is motivated by the need to report the degree to which PP has been effective during the development process of commercial software applications. This question aims to synthesize evidence from a number of studies that investigate the effectiveness of PP and determine if PP is suitable for industrial use. The motivation for RQ4 originates from the idea that the success of PP may depend on important factors, such as pair compatibility or the complexity of a given programming task. The aim of this question is to provide project managers with valuable insight regarding the most important factors for achieving effective PP.

Table 1. Summary of PICOC structure for research questions

Population	Professional software developers in industrial domains
Intervention	Pair programming
Comparison	Pair programming vs. solo programming
Outcomes	Effectiveness of pair programming
Context	Review(s) of any empirical studies on pair programming within industrial domains. No restrictions exist on the type of empirical study (e.g., case study) to be reviewed.

3.2 Identification of Relevant Literature

The aim of any SLR is to find as many primary studies that relate to the research questions as possible using an unbiased search strategy [28]. Khan et. al [27] recommend searching multiple databases to obtain as many citations as possible and to avoid publication bias. Using the keywords *"pair programming" OR "pair-programming" OR "collaborative programming"*, the first phase of identifying relevant literature involved searching the following five online databases: *ACM Digital Library*, *IEEE Xplore*, *ScienceDirect*, *Scopus*, and *SpringerLink*.

After searching the online databases and downloading relevant studies, the identification of relevant literature continued into the second phase. The *snowballing* technique was applied in this phase, meaning the references of all papers from the primary search phase were reviewed, and any references that were relevant to PP were added to the existing list of primary study candidates. This resulted in a pool of 405 primary study candidates.

3.3 Inclusion and Exclusion Criteria

Studies are eligible for inclusion if they present empirical data on PP and pass the minimum quality threshold (see Section 3.4). The primary inclusion criteria aims to select studies that investigate the effectiveness of PP as compared to SP when performed by professional software developers in industrial settings. Further inclusion criterion is composed of studies that investigate factors that affect the effectiveness of PP in industrial domains. This SLR includes both qualitative and quantitative research studies that were published between 1999-2020.

The primary exclusion criterion is composed of PP empirical studies that target the educational domain and use students as experimental subjects. Based on the exclusion criteria from [17, 48], studies are also excluded if they meet at least one of the following criteria:

- ***Criterion 1:*** PP studies targeted for CS/SE education. *Rationale:* These studies tend to use students as subjects, while the aim of the SLR is to analyze how PP is used by professional software developers in industrial domains.
- ***Criterion 2:*** PP studies conducted in experimental settings.
Rationale: Completing programming tasks during experiments typically takes a short amount of time, ranging anywhere from one hour to a few days. This is an unrealistic scenario in a real-world work environment, where projects are completed over a series of months, if not years. Moreover, it is difficult to match the complexity of an experimental programming task to that of an industrial-sized project.
- ***Criterion 3:*** Studies that do not include research questions or a research design.
Rationale: The aim of this SLR is to use empirically validated evidence, rather than reports on experiences or personal opinions.
- ***Criterion 4:*** Studies presenting claims by the author(s) with no supporting empirical data.
Rationale: Empirical data is important to obtain to draw meaningful conclusions from the results.
- ***Criterion 5:*** Studies that focus on the general application of Agile/XP.
Rationale: This SLR focuses on the specific practice of PP, and Agile/XP studies that do not thoroughly investigate the use of PP in industrial domains do not provide the necessary empirical data.
- ***Criterion 6:*** Studies that describe tools that can support PP.
Rationale: These studies mainly focus on the technical details of the software tool being proposed, while this SLR requires empirical data on the effectiveness of PP when applied in industry.
- ***Criterion 7:*** Studies on distributed PP.
Rationale: Although similar in nature, conducting remote PP sessions incorporates many factors that in-person PP sessions do not encounter.
- ***Criterion 8:*** Studies written in a language other than English or Polish. *Rationale:* The author is only capable of reading in English or Polish. However, this may result in relevant literature being omitted from the SLR (see Section 6).

Table 2. Quality criteria for selection of primary studies [16]

Questions
1. Is the paper based on empirical research, rather than a “lessons learned" report based on expert opinion?
2. Are the aims of the research clearly stated?
3. Was the research environment and context of the research adequately described?
4. Was the research design appropriate to address the aims of the research?
5. Was the recruitment strategy appropriate to the aims of the research?
6. Was there a control group with which to compare treatments?
7. Was the data collected in a way that addressed the research issue?
8. Was the data analysis sufficiently rigorous?
9. Has the relationship between researcher and participants been adequately considered?
10. Is there a clear statement of findings?
11. Is the study of value for research or practice?

3.4 Quality Assessment

At this point, the initial pool of 405 primary study candidates was reduced to 33 studies and the qualities of all the remaining studies were assessed based on the 11 criteria used by the Dybå and Dingsøyr SLR on Agile methods [16]. These criteria assess the reporting, rigor, credibility, and relevance of the remaining studies. Reporting is considered to be of high quality if the rationale, aims, and context of a study are clearly stated. Rigor refers to whether a thorough and appropriate approach was applied to the key research methods in a study. Studies are credible if the findings are well-presented and meaningful, and relevance describes the usefulness of findings to the software engineering research community and industry.

With regards to Table 2, questions 1-3 relate to the quality of the reporting of the rationale, aims, and context of a study. Since this SLR is based on evidence and data from empirical

studies, question 1 represents the minimum quality threshold that is used to exclude non-empirical studies. Questions 4-8 relate to the rigor of the research methods that were used to establish the trustworthiness of the findings. Questions 9 and 10 assess the credibility of study methodologies and help ensure that that the findings of different studies are valid and meaningful. Finally, question 11 assesses the relevance of a study to the software engineering research community and industry.

Evaluating each potential primary study based on these 11 criteria provides a measure of the extent to which the findings of a particular study can make a valuable contribution to this SLR. Each quality question is either answered as yes (1 point) or no (0 points), so the quality score of a study ranges between 0 (Very poor) to 11 (very good). However, any study that receives a no for criterion 1 (i.e., any non-empirical study) is excluded from the SLR.

3.5 Data Extraction

A predefined extraction form from the Dybå and Dingsøyr Agile SLR [16] serves as a guide for taking notes on relevant data from all primary studies. The data extraction form is an important tool that enables the recording of the full details of the studies under review and the details on how each study addresses the research questions of this SLR. Moreover, the data extraction form is a useful tool for organizing information from a large number of studies.

4 Results

This section reports the results of the SLR based on the research questions defined in Section 3.1.

4.1 Research Question 1

“What evidence exists of PP studies conducted in industrial domains that investigate the effectiveness of PP for developing commercial software applications?"

The SLR identified 33 studies conducted in industrial settings that investigate the use of PP by professional software developers for developing commercial software applications. Figure 2 shows that 13 of these studies use qualitative data analysis methods, 8 use quantitative data analysis methods, and 12 use a mix of both qualitative and quantitative methods.

4.2 Research Question 2

“What methods are used to measure the effectiveness of PP when applied in industrial domains?"

Many studies investigate the effectiveness of PP based on its impacts on software quality, productivity, knowledge transfer, effort, job satisfaction, or cost.

4.2.1 Measurements of software quality. Eleven studies measure the impact of PP on software quality [4, 13, 15, 24, 26, 47, 49, 50, 59, 61, 63]. Five of these studies use *defect density* (i.e., the ratio of total defects to total lines of code) as a quality metric [13, 15, 24, 59, 61]. Four studies use total defect count [47, 50, 61, 63]. Hulkko and Abrahamsson [26] measure software quality using three metrics: *density of coding standard deviation, comment ratio,* and *relative defect density*. Begel and Nagappan [4] measure code quality based on the code having fewer bugs and maintaining consistency with coding guidelines. Schmidt et. al [50] measure internal software quality by taking an average of the API quality, code modularity, and code understandability. One study collected data on software quality from surveys and interviews [49].

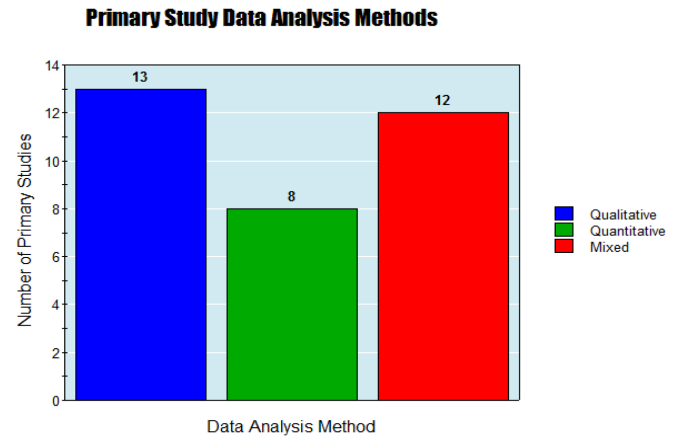

Figure 2. Overview of the primary study data analysis methods.

4.2.2 Measurements of knowledge transfer. The impact of PP on knowledge transfer is another method that ten studies use for measuring the effectiveness of PP [4, 15, 19, 21, 50, 59, 61, 63, 69, 70]. Surveys and questionnaires are the most common methods for obtaining data on knowledge transfer [4, 15, 50, 59, 61, 63]. A qualitative data analysis of full-length video recordings of industrial PP sessions is an approach used in two studies [69, 70]. Interviews of developers are applied by Gittins et. al [21] to gain insights into the effectiveness of PP for improving communication and spreading knowledge throughout a team. Fronza et al. [19] collect data throughout their study using *PROM* (Pro Metrics), an automated, non-intrusive tool for collecting and analyzing software process and product metrics. They focus on collecting data related to the amount of time that novices spend their time doing PP and draw conclusions about knowledge transfer from this data.

4.2.3 Measurements of productivity. Seven studies [26, 31, 34, 41, 52, 61, 70] evaluate the effectiveness of PP based on the impact that PP has on worker productivity. Sillitti et. al [52] measure productivity based on the amount of time that

developers working in a pair spend in directly productive activities. Productivity is defined by Vanhanen and Korpi [61] as lines of code per hour. Based on interviews of software professionals, Melo et. al [34] report that that workers have mixed definitions of productivity. Three interviewees mention timeliness as a criterion for productivity, three mention quantity, two mention quality, and one mentions customer satisfaction. Productivity is measured by Parrish et. al [41] as the average number of unadjusted function points per unit of time. Hulkko and Abrahamsson [26] calculate productivity as the ratio of produced logical code lines to spent effort. Zieris and Prechelt [70] employ the Strauss and Corbin approach of Grounded Theory Methodology (GTM). Mannaro et. al [31] use a survey to obtain developer feelings on whether they believe that PP speeds up the overall software development process.

4.2.4 Measurements of effort.

The amount of time and effort spent on developing software during PP sessions is used by five studies to determine PP effectiveness [15, 26, 59, 61, 63]. Four of these studies use surveys to gain insight into the amount of time and effort that is spent doing PP [15, 59, 61, 63]. Hulkko and Abrahamsson [26] use the quantitative metric of *pair programming effort percent*, which is the ratio of effort spent on PP activities to respective solo activities.

4.2.5 Measurements of job satisfaction.

Five studies incorporate job satisfaction as a measure of effectiveness [4, 21, 31, 61, 63]. Surveys of professional software developers are conducted in four of these studies [4, 31, 61, 63]. One study obtains data on job satisfaction via interviews [21].

4.2.6 Measurements of project cost.

Only two studies use development cost to measure PP effectiveness [4, 59]. Both of these studies use surveys to gather insight into the relationship between PP and development cost.

4.3 Research Question 3

“How effective is PP for developing commercial software applications in industrial settings?"

4.3.1 PP effects on software quality.

Ten out of eleven studies that investigate the impact of PP on software quality report that PP has a positive effect. Two of these studies report perceivable but small improvements in software quality [13, 63]. One study reports that 80% of the developers across four teams were more confident in the design and code that they generated while PP than when they work alone [15]. Questionnaire results from another study indicate that PP was the second most important practice (after test-driven development) for increasing the quality of the system and its design [61]. One study shows that software quality improved as engineers got positive reinforcement from each other, and software developed via PP had no patch rejections [47]. Interview results from another study cite an increase in code quality as one of the top three benefits of PP [49].

Sun et. al [59] survey software professionals that either have or do not have prior experience working with PP. Both groups believe that PP increases software quality, but there is a statistically significant difference ($p < 0.01$) in the magnitude of increase in software quality. Respondents with PP experience believe that PP increases quality by 26, 35, and 42 percent in low, medium, and high complexity tasks, respectively. Respondents without PP experience believe that PP increases software quality by 9, 18, and 29 percent.

Survey results from the Begel and Nagappan study [4] show that Microsoft professionals cite fewer bugs in the source code as the top benefit of PP, and higher quality code is the third most cited benefit of PP. Schmidt et. al [50] indicate that both low and high adopters of PP see a perceived improvement in all software quality aspects. Moreover, the high adopters of PP indicate a higher level of improvement in software quality compared to the low adopters. Fitzgerald and Hartnett [24] report that the required code quality level of the Intel Network Processor Division engineering team is achieved earlier when applying PP. On one project, the components that were developed via PP had the lowest defect density in the whole product (a factor of seven below the component with the highest density).

The study conducted by Hulkko and Abrahamsson [26] is the only primary study that does not report a positive effect of PP on software quality. The results of their multiple case study show conflicting findings with regards to software quality among different cases. One case reports a lower number of defects, while another case reports a higher number of defects. The authors conclude that PP does not provide as extensive quality benefits as suggested in the literature.

4.3.2 PP effects on knowledge transfer.

Ten studies report that PP can have a positive effect on knowledge transfer among a team [4, 15, 19, 21, 50, 59, 61, 63, 69, 70]. Three studies report that PP is particularly effective for facilitating knowledge transfer for inexperienced (i.e., novice) developers [15, 19, 63]. Two studies indicate that frequent rotation of partners can have a positive effect on knowledge transfer [21, 61]. Sun et. al [59] report that knowledge transfer is perceived to be the largest in high-complexity projects, junior-senior pairs, and pairs in which both developers have prior PP experience. The second most cited benefit of PP of the Begel and Nagappan study [4] is the spread of code understanding between members of a pair. Two separate studies by Zieris and Prechelt [69, 70] identify that not every episode of knowledge transfer goes frictionless, and there is a need to train developers to become better at PP to facilitate positive knowledge transfer. However, zero studies report that PP has an overall negative effect on knowledge transfer.

4.3.3 PP effects on productivity.

The results from seven studies that investigate the impact of PP on productivity are mixed. Sillitti et. al [52] show that PP helps developers eliminate distracting activities and focus on productive activities.

Vanhanen and Korpi [61] report that PP is considered the most important Agile practice that positively impacts the project productivity. Zieris and Prechelt [70] attempt to debunk the dichotomy that developers are either good enough to perform a productive PP session, or their skill levels are too far apart to be productive so they resort to a knowledge transfer session. The authors conclude that PP is both a productive and knowledge transfer technique. Survey results from the Mannaro et. al study [31] show that 72.7% of developers believe that PP speeds up the overall software development process.

Melo et. al [34] report that for complex tasks, some developers feel demotivated to work in pairs because they would like to have time to think about the problem alone before discussing it. This indicates that PP could have a negative impact on productivity. Parrish et. al [41] identify that pairs working together are not naturally more productive than solo developers. They conclude that pairs need the collaborative role-based protocol that PP provides to be productive. The multiple case study of Hulkko and Abrahamsson [26] reveals contradicting results, where two cases report an increase in productivity when doing PP, and one case shows an increase in productivity when conducting SP. Based on the empirical data, the authors conclude that a superior programming style for increasing productivity could not be detected.

4.3.4 PP effects on effort.

Drobka et. al [15] report that a negative effect of PP is that it requires large blocks of time. Vanhanen and Lassenius [63] conclude that in general, PP requires more effort than SP. They point out that although PP is exhaustive, the effort decreases to the level of SP over time. Sun et. al [59] and Vanhanen and Korpi [61] report that PP may lower the total effort for complex tasks, but for simple tasks, effort was considered higher with PP. Sun et. al also claim that a pair of junior developers increases effort, while a pair of senior developers decreases effort. Hulkko and Abrahamsson [26] report that the relative amount of effort spent on PP is highest in the beginning of a project and the defect correction phase of the project.

4.3.5 PP effects on job satisfaction.

PP had a positive effect on job satisfaction in three studies [31, 61, 63]. The tenth most cited benefit of PP in the Begel and Nagappan study [4] is an increase in morale. Gittins et. al [21] report mixed feelings on PP, where some developers experienced an increase in morale, but more experienced developers were concerned that PP undermined their status in the company.

4.3.6 PP effects on project cost.

Sun et. al [59] report that survey respondents with PP experience believe that PP reduces the overall project cost by 12%, but those without PP experience believe that PP increases the cost by 5%. The difference between the two groups is statistically significant ($p < 0.001$). Survey respondents from the Begel and Nagappan study [4] cite cost efficiency to be the number one problem with PP. One manager states, *"if I have a choice, I can employ one star programmer instead of two programmers who need to code in a pair."* Another survey respondent stated that PP, *"requires twice as many people,"* making it, *"difficult to justify the cost up front."*

4.4 Research Question 4

"What factors influence the effectiveness of PP in industrial settings?"

4.4.1 Expertise level.

Chong and Hurlbutt [9] report that expertise is an important factor that influences pair interactions, where pairing an inexperienced developer with an experienced one can lead to ineffective PP sessions. Drobka et. al [15] claim that pairing two inexperienced developers is not desirable because they may not understand development practices and how their tasks fit in the big picture. Eight studies report that pairing a novice developer with an expert can be useful for integrating novices into a team [11, 19, 22, 26, 40, 46, 50, 53]. Vanhanen and Lassenius [63] report that the most effective pairs consist of a senior and junior developer, or pairs should have knowledge areas that complement each other. Sun et. al [59] state that junior-senior pairs generate the most knowledge transfer, while junior-junior pairs generate the least. There is also more effort for junior-junior pairs in which neither has PP experience, and less effort in senior-senior pairs in which both have PP experience. Begel and Nagappan [4] report that the fifth most cited problem with PP is skill differences amongst a pair. Gittins et. al [21] report that PP works best when there is a harmony of skills and temperament between the paired developers, but PP can also be useful for mentoring novice developers.

4.4.2 Personality and social compatibility.

Personality traits within a pair are also reported as an important factor that determines the effectiveness of PP. Drobka et. al [15] indicate that a problem with PP is that managers must choose the pairs on their team carefully (e.g., PP can be ineffective with shy or overbearing personalities, or with two inexperienced developers). Based on a survey of 60 programmers, Chao and Atli [8] identify *open-minded* as the most important personality trait (75% of responses) followed by *creative*, *attentive*, *logical*, and *flexible*. Dick and Zarnett [14] claim that effective PP is difficult to achieve and requires a careful cultivation of personalities within the development team. The authors identify that effective communication, comfortableness working with one another, confidence in one's abilities, and the ability to compromise as the most essential personality traits for PP success.

Begel and Nagappan [4] cite personality clashes and ineffective communication as the third and tenth biggest problems with PP, respectively. Sfetsos et. al [51] report that other problems with PP stem from difficulties that some programmers have when working with others. Interviews from the

Gittins et. al study [21] reveal that social compatibility has an impact on the effectiveness of PP. One developer states, *"there are probably one or two people I prefer not to pair with, when you're having a conversation they're always trying to get one up on you."* Another developer states, *" ... the biggest problem is personality clashes. Generally most people here are very amenable. ... I have found myself preferring to partner with some people as opposed to others, just generally because we get on a bit better, and work in a similar way."*

4.4.3 Task complexity. Six studies report that the complexity of a given task can impact the effectiveness of PP [24, 26, 34, 59, 61, 63]. Vanhanen and Lassenius [63] identify that a large proportion of developers propose that PP should be used for complex tasks. Vanhanen and Korpi [61] delineate that PP is better suited for complex tasks than for easy tasks, in which PP can reduce effort for complex tasks, but increase effort for simple tasks. Melo et. al [34] report that PP can be ineffective on both complex and simple tasks. Conducting PP for simple tasks can be a waste of time, and for complex tasks, some feel demotivated to work in pairs because they would like to have time to think about the problem alone before discussing it.

Hulkko and Abrahamsson [26] report that developers feel that PP is more useful for demanding and complex tasks than for rote tasks. On the other hand, some developers feel that tasks which require a lot of logical thinking are best when done solo. Pair programming is beneficial when writing code which has many dependencies with other parts of the software. On the other hand, according to a team member in case two, PP for simple tasks helps a developer find mistakes that he or she has become blind to.

Results from the Sun et. al study [59] show that PP can increase effort in low-complexity tasks and decrease effort for high-complexity tasks. Moreover, PP lowers the defect rate in all types of projects, but as the project complexity increases, the defect rate decreases more. A similar pattern was found with the impact of PP on knowledge transfer, where as the complexity of a project increases, more knowledge transfer occurs. Fitzgerald and Hartnett [24] report that PP is unsuitable for simple or well understood problems, which could be fixed as quickly as a single developer could type. PP can also become frustrating when doing lots of small changes (e.g.. eliminating TO-DO's).

4.4.4 Driver-navigator interactions. The ways in which pairs interact within the driver-navigator roles is another factor that impacts the effectiveness of PP. Chong and Hurlbutt [9] identify that pairs appear to be most effective when both programmers take on driver and navigator responsibilities. Their observations show pairs engaging in a natural pattern of interaction, rather than having an explicit division of labor. Similar findings are reported Freudenberg et. al [18], in which rather than working at different levels of abstraction, the driver and navigator tend to talk in terms on the same levels of abstraction. The authors suggest that the driver and navigator should form a 'cognitive tag team' in which they work together in synchrony and frequently switch roles to alleviate the cognitive load of the driver.

Vanhanen and Korpi [61] note that frequent rotation of pairs can lead to an increase in knowledge transfer at the cost of a drop in productivity. Schmidt et. al [50] indicate that frequently switching roles allows developers to easily challenge each other on the code quality. O'Donnell and Richardson [40] report that frequent role switching is important to avoid having the navigator become bored and disengaged. Plonka et. al [45] identify that disengagement can negatively impact the effectiveness of PP. They suggest that in a novice-expert pair, the expert should encourage the novice to drive and provide sufficient explanation while working. Plonka et. al [44] also reports that contributions of both developers are a necessary precondition for achieving the benefits of PP, such as improved decision making.

4.4.5 Work environment. A given work environment is another factor that can impact PP effectiveness [4, 46, 56, 64]. Begel and Nagappan [4] report that scheduling time to work in pairs is the second most cited problem of PP. Vanhanen et. al [64] report problems with the resourcing of PP, where finding time amongst developers is not easy. Another problem stems from having a proper infrastructure to conduct PP, and implementing a dedicated PP room can solve this issue. Plonka and van der Linden [46] also claim that it is important to provide a suitable environment for PP (i.e., emphasis on collective code ownership and considering PP during the planning process). Socha and Sutanto [56] indicate that ad hoc meetings are frequent during the course of PP sessions, and social interactions of the PP practice extends outside of the pair. Their observations reveal that pairs do not create artificial boundaries around themselves and work in isolation. Instead, they intentionally configure their workspace and social conventions to enable peripheral awareness of what is happening nearby.

5 Advice for Practitioners

This section formulates insights from the SLR results in the form of explicit, procedural, and practical advice.

5.1 Investing in Better Software Quality

The results of this SLR show that an increase in software quality is one of the most reported benefits of applying PP in industrial domains. Most of the metrics that are used to measure software quality are related to the number of defects in the code that is produced when conducting PP. This suggests that the act of PP provides an inherent, full-time code review during the software development process. The main advantage of a continuous code review is that software defects are more likely to be found earlier in the software development process and be cheaper to fix. Although two

studies report that a negative aspect of PP is an increase in project cost, one can view this cost as an investment into an ongoing code review that occurs throughout the software development process. PP probably increases the cost of developing just the code, but reduces the overall cost by catching defects sooner when they are cheaper to fix. In the long run, PP is likely to pay off and save later debugging and repair costs (about 10 times what it costs to fix during the time you are programming).

The investment into better software quality may especially pay off for safety-critical systems. In an application domain where defects can result in the loss of human lives (e.g., autonomous vehicles), it is crucial to incorporate techniques that can help reduce or eliminate all bugs. For example, many studies report that formal methods are especially useful for increasing the in the software development process of safety-critical systems, where the end result is quality software of the highest integrity. The results of this SLR suggest that PP can serve as an alternative approach for producing quality software with minimal defects. Since PP is shown to have a positive effect on reducing bugs in the overall system, project managers should consider adopting PP for the development of safety-critical systems.

5.2 Strategic Implementation of PP

PP is one of the core practices of XP, and Kent Beck suggests that PP should be used all the time and for every task [3]. However, the results of this SLR show that employing this ideology will result in ineffective PP. Rather than blindly applying PP to all stages of the software development process, project managers and developers must consider the appropriate scenarios where PP has the highest levels of effectiveness. Specifically, the complexity of a task, level of pair expertise, and social compatibility among a pair are three factors that need to be considered.

There is general agreement among the primary studies of this SLR that PP is better suited for complex tasks than for easy tasks. Using PP for complex tasks can increase knowledge transfer, reduce effort, and lower the number of defects in the software. On the other hand, conducting PP for simple tasks can be a waste of time and resources. Simple tasks are also likely to result in fewer bugs than complex tasks, so the continuous code review aspect of PP is especially beneficial for complex tasks.

The results of this SLR also indicate that the level of expertise of pair members has an impact on the effectiveness of PP. Depending on the needs of an organization, project managers should strongly consider this important factor. For example, if a high priority issue is to integrate new and inexperienced developers into a team, then a form of mentorship can be employed via junior–senior PP. To avoid disengagement and increase knowledge transfer, senior developers should clearly communicate their thoughts with their partner when driving. Seniors should also encourage novices to drive, and novices should be confident knowing that they have the security of a senior developer in the navigator role. However, in a scenario where there is time pressure to finish a task, pairing two senior developers together is an appropriate strategy.

When cultivating a PP team, managers should consider allowing the developers to choose their pairing partners. Certain people work together better than others, and the employees themselves are likely to have the best knowledge of who they work best with. Since programmer expertise is an important factor that needs to be considered depending on the needs of an organization, programmers should provide management with their top partner choices depending on skill level. For example, a senior developer would cite the top three senior and junior developers that they would want to work with (total list size of six). Providing developers with this choice will likely result in higher motivation to achieve proper results and increase job satisfaction. On the other hand, having a project manager choose partners or conducting random assignment may lead to ineffective PP.

5.3 Embracing a Team-Oriented Culture

Another important factor for achieving effective PP in industrial domains lies in the culture of a company. PP should be used in companies that embrace a team-oriented culture and collective code ownership. In a study conducted by Murphy et. al [37], Microsoft employees mention that they are evaluated based on their own accomplishments and that they are directly compared to coworkers in their own team. This leads to scenarios where engaging in PP can take time away from one's own contributions while increasing the value of another's. For PP to be effective, a company should focus developer efforts on achieving team-oriented goals, rather than individual ones. Moreover, it is important to emphasize collective code ownership and perform team evaluations instead of individual assessments.

6 Limitations

Several factors need to be considered when generalizing the results of this SLR. First, during the process of identifying relevant literature, primary studies were only obtained from electronic sources. Without searching non-electronic conference proceedings and journals, there is potential that relevant studies were omitted from this SLR. Moreover, only papers written in the languages of English or Polish were considered during the primary study selection process. Omitting papers written in other languages may have resulted in the discovery of relevant studies and key insights that could have been used in this SLR.

Another exclusion criteria poses a limitation because it excludes studies that use PP in experimental settings. Experimental studies are important because they are replicable and provide researchers with an opportunity to gain quantitative

results that are more statistically significant than industrial studies, which tend to only be one data point. Experiments also enable the use of a control group, whereas asking two teams to perform identical tasks in an industrial setting is impractical and expensive. Despite their valuable aspects, experimental studies were excluded because this SLR focuses on the effectiveness of PP for the developing commercial software systems. Compared to an experiment where subjects are asked to develop small-scale systems in a short period of time, an industrial domain features larger and more complex software systems that can take months, or even years, to develop. Since it is impractical, if not impossible, to develop an application of industrial size and complexity in an experiment, it was necessary to exclude these studies from the SLR.

Another issue stems from the fact that only one person (the author) was responsible for carrying out the entire review process. The author defined the review protocol and carried out the major tasks involved in each phase of the SLR, which may have unwittingly biased the results of the SLR. For example, having only one person to select primary studies based on a set of inclusion and exclusion criteria could have resulted in the inclusion of studies that should have been omitted (or vice versa). However, an effort was made to avoid bias by closely following the recommendations suggested in the SLR guidelines [28].

7 Future Work

Future work plans to recreate the work of Cockburn and Williams [10] and assess the modern-day costs and benefits of using PP in industry. Specifically, this work will compare the doubled cost of producing a line of code to the savings that results from discovering defects during PP sessions (as a result of the continuous code review), rather than being left to be discovered during testing (or later) and fixing when they are more expensive to fix. Cockburn and Williams report that the initial 15% increase in project development cost is repaid in shorter and less expensive testing, quality assurance, and field support [10]. The goal of the future research is to recreate the Cockburn and Williams study by using newer data and seeing if the costs and benefits from 2001 still hold up twenty years later. However, one key difference is that the previous study by Cockburn and Williams uses data from small-scale student experiments, but the new study will use data from both industry and experiments.

8 Conclusion

The results of this SLR suggest that PP can be successfully applied to the development process of commercial software applications. However, contrary to the mindset of XP where PP is blindly used for all development tasks, the results of this research suggest that PP effectiveness depends on the context of a given work scenario. This includes factors like organizational priorities, programming task complexity, social and technical pair compatibility, work environment, and company culture. As with many things in life, finding a proper balance between the use of PP and SP will lead to optimal development. Early evidence suggests that PP offers an extensive set of benefits in industry, and it is important for the software engineering community to continue investigating the costs and benefits of this practice.

Acknowledgments

I would like to express my very great appreciation to Dr. Daniel M. Berry for his valuable and constructive suggestions during the planning and development of this research work. I would also like to thank David Radke (PhD candidate, University of Waterloo) for his help during the revision process of this paper.

References

[1] Mustafa Ally, Fiona Darroch, and Mark Toleman. 2005. A Framework for Understanding the Factors Influencing Pair Programming Success. In *Extreme Programming and Agile Processes in Software Engineering*, Hubert Baumeister, Michele Marchesi, and Mike Holcombe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 82–91.

[2] E. Arisholm, H. Gallis, T. Dyba, and D. I. K. Sjoberg. 2007. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Transactions on Software Engineering* 33, 2 (2007), 65–86.

[3] Kent Beck. 1999. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Longman Publishing Co., Inc., USA.

[4] Andrew Begel and Nachiappan Nagappan. 2008. Pair Programming: What's in it for Me? 120–128. https://doi.org/10.1145/1414004.1414026

[5] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. 2007. Lessons from Applying the Systematic Literature Review Process within the Software Engineering Domain. *J. Syst. Softw.* 80, 4 (April 2007), 571–583. https://doi.org/10.1016/j.jss.2006.07.009

[6] Gerardo Canfora, Aniello Cimitile, Felix Garcia, Mario Piattini, and Corrado Aaron Visaggio. 2007. Abstract Evaluating performances of pair designing in industry. *Journal of Systems and Software* 80 (08 2007), 1317–1327. https://doi.org/10.1016/j.jss.2006.11.004

[7] Gerardo Canfora, Aniello Cimitile, Corrado Aaron Visaggio, Felix Garcia, and Mario Piattini. 2006. Performances of Pair Designing on Software Evolution: a controlled experiment. 197–205. https://doi.org/10.1109/CSMR.2006.40

[8] J. Chao and G. Atli. 2006. Critical personality traits in successful pair programming. In *AGILE 2006 (AGILE'06)*. 5 pp.–93.

[9] J. Chong and T. Hurlbutt. 2007. The Social Dynamics of Pair Programming. In *29th International Conference on Software Engineering (ICSE'07)*. 354–363.

[10] Alistair Cockburn and Laurie Williams. 2001. *The Costs and Benefits of Pair Programming.* Addison-Wesley Longman Publishing Co., Inc., USA, 223–243.

[11] Irina Coman, Alberto Sillitti, and Giancarlo Succi. 2008. Investigating the Usefulness of Pair-Programming in a Mature Agile Team, Vol. 9. 127–136. https://doi.org/10.1007/978-3-540-68255-4_13

[12] E. di Bella, I. Fronza, N. Phaphoom, A. Sillitti, G. Succi, and J. Vlasenko. 2013. Pair Programming and Software Defects–A Large, Industrial Case Study. *IEEE Transactions on Software Engineering* 39, 7 (2013), 930–953.

[13] E. di Bella, I. Fronza, N. Phaphoom, A. Sillitti, G. Succi, and J. Vlasenko. 2013. Pair Programming and Software Defects–A Large, Industrial Case Study. *IEEE Transactions on Software Engineering* 39, 7 (2013), 930–953.

[14] Andrew Dick, Bryan Zarnett, Red Hook, Group Red, and Hook Group. 2002. Paired Programming and Personality Traits. (06 2002).

[15] J. Drobka, D. Noftz, and Rekha Raghu. 2004. Piloting XP on four mission-critical projects. *IEEE Software* 21, 6 (2004), 70–75.

[16] Tore Dybå and Torgeir Dingsøyr. 2008. Empirical Studies of Agile Software Development: A Systematic Review. *Inf. Softw. Technol.* 50, 9–10 (Aug. 2008), 833–859. https://doi.org/10.1016/j.infsof.2008.01.006

[17] T. Dybå, E. Arisholm, D. I. K. Sjoberg, J. E. Hannay, and F. Shull. 2007. Are Two Heads Better than One? On the Effectiveness of Pair Programming. *IEEE Software* 24, 6 (2007), 12–15.

[18] S. Freudenberg, P. Romero, and B. du Boulay. 2007. "Talking the talk": Is intermediate-level conversation the key to the pair programming success story?. In *Agile 2007 (AGILE 2007)*. 84–91.

[19] I. Fronza, A. Sillitti, and G. Succi. 2009. An interpretation of the results of the analysis of pair programming during novices integration in a team. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. 225–235.

[20] Ilenia Fronza, Alberto Sillitti, Giancarlo Succi, and Jelena Vlasenko. 2011. Analysing the Usage of Tools in Pair Programming Sessions. *Lecture Notes in Business Information Processing* 77, 1–11. https://doi.org/10.1007/978-3-642-20677-1_1

[21] Robert Gittins, Julian Bass, and Sian Hope. 2004. A Comparison of Software Development Process Experiences, Vol. 3092. 231–236. https://doi.org/10.1007/978-3-540-24853-8_30

[22] Peggy Gregory, Diane Strode, Raid Alqaisi, Helen Sharp, and Leonor Barroca. 2020. *Onboarding: How Newcomers Integrate into an Agile Project Team*. 20–36. https://doi.org/10.1007/978-3-030-49392-9_2

[23] J. E. Hannay, E. Arisholm, H. Engvik, and D. I. K. Sjoberg. 2010. Effects of Personality on Pair Programming. *IEEE Transactions on Software Engineering* 36, 1 (2010), 61–80.

[24] Gerard Hartnett and Brian Fitzgerald. 2005. A Study of the Use of Agile Methods within Intel. *International Federation for Information Processing Digital Library; Business Agility and Information Technology Diffusion;* 180. https://doi.org/10.1007/0-387-25590-7_12

[25] Sven Heiberg, Uuno Puus, Priit Salumaa, and Asko Seeba. 2003. Pair-Programming Effect on Developers Productivity. In *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering* (Genova, Italy) *(XP'03)*. Springer-Verlag, Berlin, Heidelberg, 215–224.

[26] H. Hulkko and P. Abrahamsson. 2005. A multiple case study on the impact of pair programming on product quality. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 495–504.

[27] Khalid S. Khan, Regina Kunz, Jos Kleijnen, and Gerd Antes. 2011. Systematic reviews to support evidence-based medicine.

[28] Barbara Kitchenham. 2004. Procedures for Performing Systematic Reviews. *Keele, UK, Keele Univ.* 33 (08 2004).

[29] Kim Man Lui and Keith C. C. Chan. 2003. When Does a Pair Outperform Two Individuals?. In *Extreme Programming and Agile Processes in Software Engineering*, Michele Marchesi and Giancarlo Succi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 225–233.

[30] K. M. Lui, K. C. C. Chan, and J. Nosek. 2008. The Effect of Pairs in Program Design Tasks. *IEEE Transactions on Software Engineering* 34, 2 (2008), 197–211.

[31] Katiuscia Mannaro, Marco Melis, and Michele Marchesi. 2004. Empirical Analysis on the Satisfaction of IT Employees Comparing XP Practices with Other Software Development Methodologies. 166–174. https://doi.org/10.1007/978-3-540-24853-8_19

[32] Charlie McDowell, Linda Werner, Heather Bullock, and Julian Fernald. 2002. The Effects of Pair-Programming on Performance in an Introductory Programming Course. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky) *(SIGCSE '02)*. Association for Computing Machinery, New York, NY, USA, 38–42. https://doi.org/10.1145/563340.563353

[33] Charlie McDowell, Linda Werner, Heather E. Bullock, and Julian Fernald. 2006. Pair Programming Improves Student Retention, Confidence, and Program Quality. *Commun. ACM* 49, 8 (Aug. 2006), 90–95. https://doi.org/10.1145/1145287.1145293

[34] Claudia Melo, Daniela S. Cruzes, Fabio Kon, and Reidar Conradi. 2011. Agile Team Perceptions of Productivity Factors. In *Proceedings of the 2011 Agile Conference (AGILE '11)*. IEEE Computer Society, USA, 57–66. https://doi.org/10.1109/AGILE.2011.35

[35] Raimund Moser, Marco Scotto, Alberto Sillitti, and Giancarlo Succi. 2007. Does XP Deliver Quality and Maintainable Code?, Vol. 4536. 105–114. https://doi.org/10.1007/978-3-540-73101-6_15

[36] M. Mueller. 2003. Are Reviews an Alternative to Pair Programming? *Empirical Software Engineering* 9 (2003), 335–351.

[37] Brendan Murphy, Thomas Zimmermann, Laurie Williams, Nachiappan Nagappan, and Andrew Begel. 2013. Have Agile Techniques been the Silver Bullet for Software Development at Microsoft. *International Symposium on Empirical Software Engineering and Measurement*. https://doi.org/10.1109/ESEM.2013.21

[38] Jerzy Nawrocki and Adam Wojciechowski. 2001. Experimental Evaluation of Pair Programming. *Proceedings of the 12th European Software Control and Metrics Conference* (08 2001).

[39] John T. Nosek. 1998. The Case for Collaborative Programming. *Commun. ACM* 41, 3 (March 1998), 105–108. https://doi.org/10.1145/272287.272333

[40] Michael O'Donnell and Ita Richardson. 2008. Problems Encountered When Implementing Agile Methods in a Very Small Company. *Software Process Improvement*, 13–24. https://doi.org/10.1007/978-3-540-85936-9_2

[41] A. Parrish, R. Smith, D. Hale, and J. Hale. 2004. A field study of developer pairs: productivity impacts and implications. *IEEE Software* 21, 5 (2004), 76–79.

[42] Monvorath Phongpaibul and Barry Boehm. 2006. An Empirical Comparison between Pair Development and Software Inspection in Thailand. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering* (Rio de Janeiro, Brazil) *(ISESE '06)*. Association for Computing Machinery, New York, NY, USA, 85–94. https://doi.org/10.1145/1159733.1159749

[43] M. Phongpaibul and B. Boehm. 2007. A Replicate Empirical Comparison between Pair Development and Software Development with Inspection. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. 265–274.

[44] L. Plonka, J. Segal, H. Sharp, and J. v. d. Linden. 2012. Investigating Equity of Participation in Pair Programming. In *2012 Agile India*. 20–29.

[45] Laura Plonka, Helen Sharp, and Janet van der Linden. 2012. Disengagement in Pair Programming: Does It Matter?. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 496–506.

[46] L. Plonka and J. van der Linden. 2012. Why developers don't pair more often. In *2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE)*. 123–125.

[47] C. Poole and J. W. Huisman. 2001. Using extreme programming in a maintenance environment. *IEEE Software* 18, 6 (2001), 42–50.

[48] N. Salleh, E. Mendes, and J. Grundy. 2011. Empirical Studies of Pair Programming for CS/SE Teaching in Higher Education: A Systematic Literature Review. *IEEE Transactions on Software Engineering* 37, 4 (2011), 509–525.

[49] C. Schindler. 2008. Agile Software Development Methods and Practices in Austrian IT-Industry: Results of an Empirical Study. In *2008 International Conference on Computational Intelligence for Modelling*

Control Automation. 321–326.

[50] Christoph Tobias Schmidt, Srinivasa Ganesha Venkatesha, and Juergen Heymann. 2014. Empirical Insights into the Perceived Benefits of Agile Software Engineering Practices: A Case Study from SAP. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 84–92. https://doi.org/10.1145/2591062.2591189

[51] Panagiotis Sfetsos, Lefteris Angelis, and Ioannis Stamelos. 2006. Investigating the extreme programming system–An empirical study. *Empirical Software Engineering* 11 (06 2006), 269–301. https://doi.org/10.1007/s10664-006-6404-6

[52] A. Sillitti, G. Succi, and J. Vlasenko. 2012. Understanding the impact of Pair Programming on developers attention: A case study on a large industrial experimentation. In *2012 34th International Conference on Software Engineering (ICSE)*. 1094–1101.

[53] Alexandre Silva, Fabio Kon, and Cicero Torteli. 2005. XP South of the Equator: An eXPerience Implementing XP in Brazil, Vol. 3556. 10–18. https://doi.org/10.1007/11499053_2

[54] R. Sison. 2008. Investigating Pair Programming in a Software Engineering Course in an Asian Setting. In *2008 15th Asia-Pacific Software Engineering Conference*. 325–331.

[55] R. Sison. 2009. Investigating the Effect of Pair Programming and Software Size on Software Quality and Programmer Productivity. In *2009 16th Asia-Pacific Software Engineering Conference*. 187–193.

[56] D. Socha and K. Sutanto. 2015. The "Pair" as a Problematic Unit of Analysis for Pair Programming. In *2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering*. 64–70.

[57] Matt Stephens and Doug Rosenberg. 2003. *Pair Programming (Dear Uncle Joe, My Pair Programmer Has Halitosis)*. 135–160. https://doi.org/10.1007/978-1-4302-0810-5_6

[58] Giancarlo Succi, Witold Pedrycz, Michele Marchesi, and Laurie Williams. 2002. Preliminary Analysis of the Effects of Pair Programming on Job Satisfaction. (08 2002).

[59] Wenying Sun, George Marakas, and Miguel Aguirre-Urreta. 2015. Effectiveness Of Pair Programming: Perceptions Of Software Professionals. *IEEE Software* 33 (01 2015), 1–1. https://doi.org/10.1109/MS.2015.106

[60] J. Vanhanen and H. Korpi. 2007. Experiences of Using Pair Programming in an Agile Project. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. 274b–274b.

[61] J. Vanhanen and H. Korpi. 2007. Experiences of Using Pair Programming in an Agile Project. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. 274b–274b.

[62] Jari Vanhanen and Casper Lassenius. 2005. Effects of pair programming at the development team level: An experiment. 10 pp. https://doi.org/10.1109/ISESE.2005.1541842

[63] Jari Vanhanen and Casper Lassenius. 2007. Perceived Effects of Pair Programming in an Industrial Context. *Conference Proceedings of the EUROMICRO*, 211 – 218. https://doi.org/10.1109/EUROMICRO.2007.47

[64] J. Vanhanen, C. Lassenius, and M. V. Mantyla. 2007. Issues and Tactics when Adopting Pair Programming: A Longitudinal Case Study. In *International Conference on Software Engineering Advances (ICSEA 2007)*. 70–70.

[65] Danny Weyns, M. Usman Iftikhar, Didac Gil de la Iglesia, and Tanvir Ahmad. 2012. A Survey of Formal Methods in Self-Adaptive Systems. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering* (Montreal, Quebec, Canada) *(C3S2E '12)*. Association for Computing Machinery, New York, NY, USA, 67–79. https://doi.org/10.1145/2347583.2347592

[66] L. Williams and R.R. Kessler. 2003. *Pair Programming Illuminated*. Addison-Wesley Longman Publishing Co., Inc. https://books.google.com/books?id=LRQhdlrKNE8C

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

[68] L. Williams, A. Shukla, and A. I. Anton. 2004. An initial exploration of the relationship between pair programming and Brooks' law. In *Agile Development Conference*. 11–20.

[69] Franz Zieris and Lutz Prechelt. 2014. On Knowledge Transfer Skill in Pair Programming. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Torino, Italy) *(ESEM '14)*. Association for Computing Machinery, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/2652524.2652529

[70] F. Zieris and L. Prechelt. 2016. Observations on Knowledge Transfer of Professional Software Developers during Pair Programming. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 242–250.