Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 2009 Proceedings

Americas Conference on Information Systems (AMCIS)

2009

The True Cost of Pair Programming: Development of a Comprehensive Model and Test

Wenying "Nan" Sun
University of Kansas, nansun@ku.edu

George M. Marakas
University of Kansas, gmarakas@ku.edu

The True Cost of Pair Programming: Development of a Comprehensive Model and Test

Wenying "Nan" Sun
University of Kansas
nansun@ku.edu

George M. Marakas
University of Kansas
gmarakas@ku.edu

ABSTRACT

This study aims to answer the following research question: compared to solo programming, is pair programming a more cost effective method for developing software? This paper proposes a research model based on theories and previous empirical studies. It discusses a multi-study approach to address the question in hand. The first study is a survey of practitioners in regards to their experience and perception of the cost of pair programming. Information acquired from the survey are then fed into simulation models as input parameters with the purpose to identify situations where pair programming is or is not more cost effective than solo programming.

Keywords

Pair programming, solo programming, cost, duration, effort, defect, knowledge transfer

INTRODUCTION

Pair programming is a programming method where two developers work side by side in front of one computer (Beck, 2000; Williams and Kessler, 2000; Arisholm, Gallis, Dybå, and Sjøberg, 2007). Pair programming differs from the traditional solo programming by having two developers work on the same programming task at the same time. Pair programming is one of the twelve principles Extreme Programming (XP) follows (Sommerville, 2007). XP, the brainchild of Kent Beck and colleagues Ron Jeffries and Ward Cunningham (Beck 2000), has emerged as the most successful and best-known agile method (Sommerville, 2007). Beck credits much of its success to the use of pair programming (Williams, Kessler, Cunningham and Jeffries, 2000).

There have been a considerable number of researches in pair programming. The majority of the researches study the impact of pair programming on program duration, effort, and quality. Although there are some mixed results from various studies, the general finding from this stream of research is pair programming requires more man hours but shortens duration and improves quality. While these studies enhance our understanding of the impact of pair programming on key project metrics, they do not address the combined effect of these metrics on the overall cost of a software development project. From this stream of research, one cannot answer the question: is pair programming a cost effective method for software development?

Two simulation studies attempted to address the economics of pair programming and yielded different conclusions. Erdogmus and Williams (2003) present a positive economic picture for pair programming in all situations while Padberg and Müller (2003) suggest the economic benefit of pair programming depends on factors such as market pressure. Both studies apply rigorous mathematical models but are restricted by a common limitation: the lack of reliable parameters for the simulation models. For example, Erdogmus and Williams' model relies on data on productivity, defect rate, and rework speed, and Padberg and Müller's model depends on data about pair speed advantage and pair defect advantage. However, empirical evidence of these data items is very limited.

As results from a non-systematic survey, Cusumano, MacCormack, Kemerer, and Crandall (2003) report 35.3 % of all projects (37 out of 104) they surveyed used pair programming. It seems, despite of the growing interest in agile methods, issues remain to inhibit the majority of the organizations from adopting pair programming. We believe the concern of increased overall project cost is a major obstacle. Compared to solo, does pair programming increase the overall project cost? How much does a project increase/decrease in overall cost with more man hours but improved quality? Since existing literature do not provide answers to these questions, corporate decision makers do not have guidelines to follow to resolve this bottom line issue. Without clear cost benefits, transition from solo to pair programming is difficult, if not possible, to argue for.

To fill the gap, this study aims to answer the following research question: compared to solo, is pair programming a more cost effective method for developing software? In particular, what are effects of system complexity, programmer expertise, and pair composition on the overall cost of pair programming?

This study makes several contributions. First, it synthesizes existing literature, and hopes to elevate the topic to a more comprehensive level of academic research. Second, by answering some crucial questions that are not addressed in the research to date, it provides practical guidance to the industry when one needs to decide which programming method to adopt in different project environments.

The paper is organized as follows. The first section provides literature review on pair programming. Next, we discuss the theoretical foundation. We then propose research hypotheses and research model. Finally, we present research methods.

LITERATURE REVIEW

A substantial number of studies have been conducted to examine the various effects of pair programming in different context: industrial setting, classroom setting, comparing pair to solo, comparing pair to other group development methods, distributed pair programming, pair composition, and the economic aspects of pair programming. A summary of the studies reviewed is presented in Tables 1 and 2 in Appendix A. Since the focus of this study is to examine pair programming using solo programming as the benchmark, literature that compare pair programming to other team development methods such as peer review and inspection are not reported.

The results of these studies undoubtedly provided valuable information to the overall picture of the efficacy of pair programming. However, a comprehensive review of the literature reveals several limitations. First, the majority of the studies were conducted using neither theory nor research framework. The framework suggested by Gallis, Arisholm, and Dybå (2003) and extended by Ally, Darroch, and Toleman (2005) has been largely ignored. Few empirical studies have been conducted to test and refine the framework. Actually, among the studies reviewed above, the only study that explicitly presented a research model is Arisholm, Gallis, Dybå, and Sjøberg (2007), which adopts certain components from the framework.

Second, most of the results have been obtained from experimental studies in university settings. Only a few empirical studies involved industrial practitioners. And even with those studies, since professional programmers worked on short tasks, the complexity of real world software development was not reflected. This might be the main reason why studies derive contradictory conclusions.

Third, most survey studies were conducted without academic rigor. None of the survey studies went through the usual cycle of reliability and validity check, which brings the subsequent results under question.

Finally, studies on the economic aspect of pair programming are limited. Only two economic models have been developed. The two models look into different factors, have simplified assumptions, use unsure parameters, and draw different conclusions. Five years have passed, no research has been done to synthesize or improve the models. Dawande, Johar, Kumar, and Mookerjee (2008) adopt genetic algorithm to develop models to compare the performances of pair, solo, and mixed development under two separate objectives: effort minimization and time minimization. While this study develops models to minimize effort or time, it does not address directly the economics of different programming methods.

In conclusion, our literature review echoes the comments several authors made in their studies. Gallis, Arisholm, and Dybå (2003) note that results from existing empirical work contradicted each other due to the lack of a theoretical framework to support the pair programming research. Hulkko and Abrahamsson (2005) state that the current body of knowledge in pair programming is scattered and unorganized. Parrish, Smith, Hale, and Hale (2004) suggest that more empirical evidence from real industry projects is needed.

THEORETICAL FOUDATION

Several theories in software engineering are available to explain why a pair outperforms a solo: egoless programming (Weinberg, 1971), surgical team (Brooks, 1975), dynamic duos (Constaine, 1995), and distributed

cognition (Flor and Hutchins, 1991; Williams and Kessler, 2001; Williams and Upchurch, 2001). The learning methods of self-discovery and co-discovery supported by Lim, Ward, and Benbasat (1997) present the same idea.

While these theories explain why pair programming is better than solo programming in terms of quality, they do not address whether pair programming is more cost-effective than solo programming. It is reasonable to assume that the natural goal of software development organizations is to be as profitable as possible while providing customers quality products as quickly and cheaply as possible. Therefore, the economic feasibility of pair programming is a key issue, for which we turn to economics of software engineering for guidance.

Several methods are present to measure the economic feasibility of a project. One is net present value (NPV) which measures the differences between the present value of benefits and the present value of costs. A couple of formulas have been suggested. One is used by Erdogmus (1999): *NPV = (Asset value – Operation cost)/(1 + Product risk)Development cost – Development cost +Flexibility value.* The other is applied by Padberg and Müller (2003):*NPV = (Asset value)/((1 + Discount rate)^Dev Time – Dev Cost)*

Another method to examine economic feasibility is breakeven analysis. Ergomus and Williams (2003) use the following formula to compare two development methods: *Breakeven Unit Value Ratio (BUVR) = BUV (solo) / BUV (pair).* BUV is the threshold value of V above which NPV is positive; V is measured in $/LOC and represents the fixed increase in earned value per each additional unit of output produced. They state that as the ratio increases, the advantage of pair over solo also increases.

Despite the differences in methods and detailed formulas, the underlying concept is the same, which is the economics of software engineering. As Sommerville (2007) states labor cost (the cost of paying software developers) is the dominant cost for most projects, the potential of incurring more total hours when two developers working on the same task together raises a legitimate concern, When the two developers negotiate to identify the best solution among the alternatives, the deadline is approaching. The decision of having the very best solution but missing the deadline or using a reasonable solution and staying within the time table is a common one facing all developers. These touchy issues suggest the balance of cost, quality, and timeline is fundamental to software development.

In addition to the theories mentioned above, as discussed in the literature review section, Gallis, Arisholm, and Dybå (2003) suggests a comprehensive framework for pair programming research, and Ally, Darroch, and Toleman (2005) extends it by adding organizational variables such as organizational culture, team building and pair management . No empirical studies have been conducted to test the whole framework. Actually the majority of the pair programming research do not present explicit research models in their studies. Arisholm, Gallis, Dybå, and Sjøberg (2007) and Balijepally, Mahapatra, Nerur, and Price (2009) are two exceptions. Figures 1-3 are the framework and the proposed and tested research models.

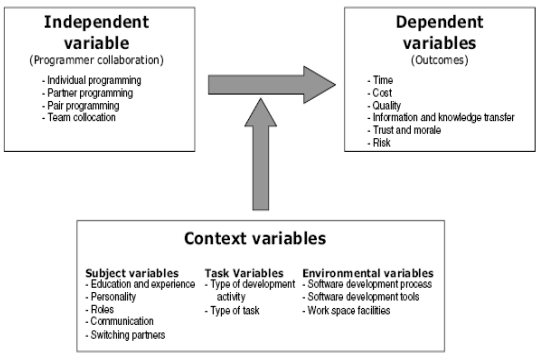

Figure 1. Research Framework for Pair Programming (Gallis et al., 2003)

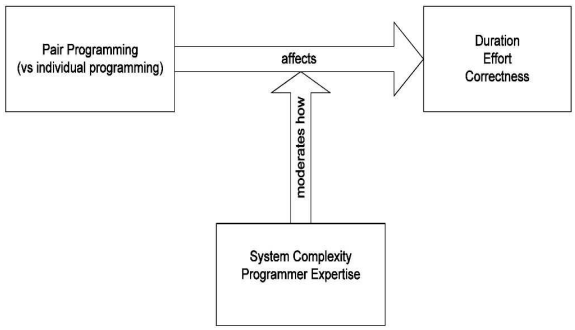

Figure 2. Research model (Arisholm et al., 2007)

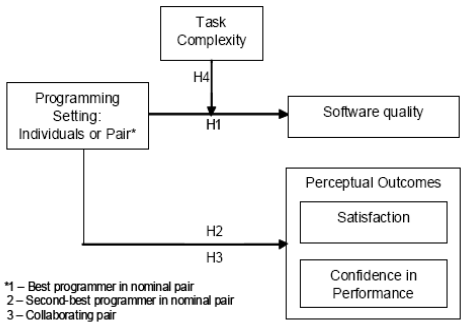

Figure 3. Research model (Balijepally et al., 2009)

RESEARCH HYPOTHESES

We present the following research model based on the pair programming research framework and findings from prior studies. The supporting arguments for the research model are as follows. The constructs in the dash box are derived from logic. As argued below, they do not require hypothesis testing. They are included to demonstrate how cost is considered in this research.

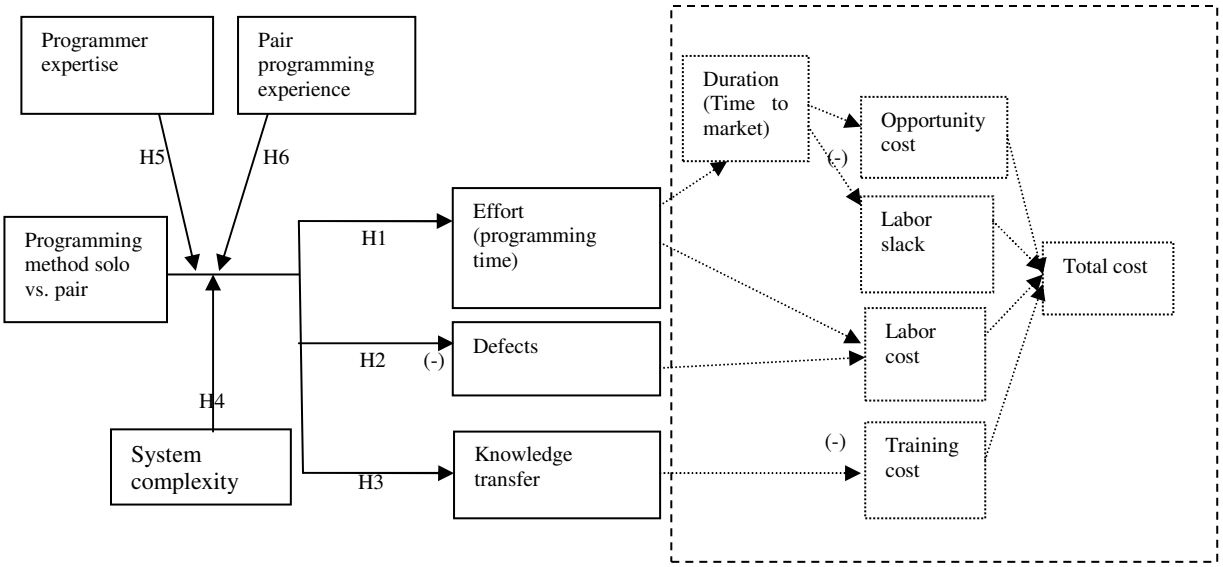

Figure 4. Research Model

Effort

In this study, effort is the total number of hours developers spend on the programming aspect of a project. In solo programming, for a particular programming task, the effort is the number of hours one developer spends on the task. In pair programming, since two developers work on the same task at the same time, the effort is 2 times the number of hours one developer spends on the task.

Studies generally find that pair programming incurs more programming hours than solo. Nosek (1997) reports 41% more effort compared to solo. Cockburn and Williams (2001) notes 15% more effort compared to solo. Nawrocki and Wojciechowski (2001) finds 100% more effort compared to solo. The meta analysis study conducted by Dybå, Arisholm, Sjøberg, Hannay, and Shull in 2007 reports a medium-sized negative effect due to pair programming (effect-size=-0.57). In the same vein, we hypothesize:

H1: Compared to solo programming, pair programming will increase the total number of programming hours in a project.

Defects

Defects are errors developers introduce to the programs. Defect is a primary dimension of project quality. In solo programming, the developer works by himself/herself to create the code and then relies on the compiler and debugging programs to identify and fix problems during unit testing, which is typically conducted by the developer who creates the program. In pair programming, the developers go through the same program creation and unit testing phase except that two developers are involved. During the program creation step, the navigator reviews the code as the driver types it in. This immediate review and feedback allows the developers to capture errors when the program is still in development stage. Furthermore, when there are errors in the initial program, during unit testing, besides tools such as compilers and debugging programs, there is an extra pair of eyes and an extra brain in pair programming situation to identify and fix problems before the program leaves the hands of the programmers and moves to a different project phase.

Numerous studies report that pair programming helps create programs with fewer defects. Nosek (1998) report pairs produce more functional solutions than solos. Jensen (2003) reports an error rate of 0.001 * normal with pair programming in his industry report. In Vanhanen and Korpi (2007), all developers reported that pair programming lowers the number of defects. Therefore, we hypothesize:

H2. Compared to solo programming, pair programming will have fewer defects in the final program.
H2a. Compared to solo programming, pair programming will enter fewer defects in the initial program.
H2b. Compared to solo programming, pair programming will identify and fix more defects during compilation and unit testing.

Knowledge transfer

Knowledge transfer is the communication of knowledge from a source so that it is learned and applied by a recipient (Ko, Kirsch, and King, 2005). In pair programming, the driver and the navigator communicate with each other constantly. Several aspects of knowledge transfer can take place. One is knowledge transfer in regards to programming syntax and logic. The other is the understanding of the program itself and its relationship to the overall system. The third is the approaches to solve problems in general.

Ally, Darroch, and Toleman (2005) and Vanhanen and Lassenius (2005) report that pair programming allows the improved knowledge transfer. All developers in Vanhanen and Korpi (2007) consider that pair programming increased their knowledge of the system. In the same vein, we hypothesize:

H3 Compared to solo programming, pair programming will result in higher knowledge transfer.

System complexity

We acknowledge there are many project related variables that need to be tested. For this study, we focus on system complexity, which, we believe, is the most salient factor that influences the outcome of a project.

Vanhanen and Korpi (2007) report effort depends on task complexity. For complex tasks, pair programming reduces the total effort but for simple tasks pair programming increases effort.

Arisholm, Gallis, Dybå, and Sjøberg (2007) suggest complexity should not be viewed as absolute terms; it is relative to the expertise of the developers. In their experiment, based on results from earlier experiments, they used application control style (delegated vs. centralized) to discriminate two levels of system complexity: low and high. They examine system complexity as a moderator and find the effect of pair programming on duration, effort and defects depends on system complexity.

We believe a general measure of system complexity is the number of modules and their interdependence. As the number of modules grows, more communications among the modules are required, thus increasing system complexity. This argument is in line with what is suggested by several studies in project complexity (Banker, Davis, and Slaughter, 1998; Kemerer, 1995; Espinosa, Slaughter, Kraut, and Herbsleb, 2007).

Literature suggests complex tasks benefit more from discussions among the group members on alternative solutions (Robbins, 2000). However, as complexity increases, more and longer time is needed to deliver a solution (Arisholm et al., 2007). We also argue the amount of knowledge transferred varies as the complexity of the project changes.

We hypothesize:

H4a. System complexity moderates the effect of programming method on effort.
H4b. System complexity moderates the effect of programming method on defects.
H4c. System complexity moderates the effect of programming method on knowledge transfer.

Programmer expertise

Several studies report the effect of pair programming depends on the programmer expertise. Nosek (1998) finds programmers with more years of experience perform better than programmers with fewer years of experience. In his study, experience and problem scores were highly correlated for both the control/solo (73.5%) and the experimental/pair (87.2%) groups.

Jensen (2003) and Toll III, Lee, and Ahlswede (2007) suggest pair programming works best when the pairs are of slightly different skill level.

Lui and Chan (2006) aim to understand when a pair outforms a solo. In their repeat program experiment, they have subjects repeatedly write the same program four times and measure their corresponding productivities. They conclude novice-novice pairs against novice solos are more productive than expert-expert pairs against expert solos. They posit the factor of familiarity affects productivity in pair programming, meaning the productivity of pair programming diminishes when pairs keep solving the same problem. They suggest a pair is more productive when they are new to a programming problem.

Chong and Hurlbutt (2007) suggest the ramification of expertise should be carefully considered when forming pairs. Through their enthnographic observations, they note pairing less knowledgeable programmer with more knowledgeable programmer seem to be effective when the less knowledgeable one is new to the team and the code base.

Arisholm, Gallis, Dybå, and Sjøberg (2007) test programmer expertise as a moderator in their study. In their experiment, pairs consisted of two individuals with a similar level of programmer expertise. Programmer expertise was measured by two indicators. One was programmer category (junior, intermediate, and senior) as determined by the project managers. The other attempted to measure programming skill more directly on the basis of the results of the pretest programming task. They find the effect of pair programming on duration and defects but not effort depend on expertise.

In this study, we adopt the terms of junior and senior to represent the different levels of expertise in a project. A senior developer is one who has high expertise in both the project domain and the project tasks. A junior has low expertise in either or both domain and tasks. We argue the effect of pair programming varies when pairs of different compositions - junior-junior, senior-senior, and junior-senior are compared to junior/senior solos.

We hypothesize:

H5a. Programmer expertise moderates the effect of programming method on effort.
H5b.Programmer expertise moderates the effect of programming method on defects.
H5c.Programmer expertise moderates the effect of programming method on knowledge transfer.

Pair programming experience

Bryant (2004) finds there are different interaction frequencies and interaction types between novice and expert pair programmers. His study reports expert pair programmers are more selective about their interactions and averaged 27% fewer interactions per hour and novice pair programmers suggested and counter-suggested more frequently (82.5 times per session) than experts (42.5 times per session). The concept of pair jelling was discussed in Williams, Kessler, Cunningham, and Jeffries (2000). The authors suggest that programmers go through an initial adjustment period in the transition from solitary to collaborative programming. This adjustment period varies from hours to days. The authors report that in their student experiments, after the adjustment in the first assignment, the paired students performed much better in the subsequent tasks. For example, on average, pairs took 60% more programmer hours to complete the assignment, but after the adjustment period, this 60% decreased to 15%. We

argue prior pair programming experience allows the programmers to get adjusted to each other and become productive quickly. We hypothesize:

H6a. Pair programming experience moderates the effect of programming method on effort.
H6b.Pair programming experience moderates the effect of programming method on defects.
H6c.Pair programming experience moderates the effect of programming method on knowledge transfer.

Duration

Project duration is the time elapsed from the start to the delivery of a project. Several studies support the notion that pair programming allows the faster delivery of a project compared to solo.

Lui and Chan (2003) suggest pairs were faster than solos. Reifer (2002) reports a time-to-market compression - 25 to 50 percent less time compared to projects that did not use agile methods. Williams, Kessler, Cunningham, and Jeffries (2000) and Williams and Kessler (2001) both find that pairs produced programs with shorter cycle. Dybå, Arisholm, Sjøberg, Hannay, and Shull (2007) note a medium-sized overall reduction of the project duration (effect-size = 0.40) in their meta-analysis.

While previous literature usually develop hypotheses to test the effects of programming method on duration, we argue duration can be derived from effort when all others being equal. For instance, if the efforts in solo and pair programming are 140 and 210 hours respectively, then assuming working seven hours a day, solo programming will finish the project in 20 days (140/7) while pair will finish it in 15 days (210/2/7). No additional data collection is necessary.

Cost

According to Sommerville (2007), project cost is primarily the costs of the labors involved. These direct labor costs are the costs of paying software developers for the hours spent on a project.

For each task, the direct labor cost is the number of hours developers spend on the task times their corresponding pay rate. For the overall project, the direct labor cost is the sum of the costs for all tasks involved. Since the direct labor cost is a function of hours and pay rate, assuming constant pay rates, the more hours developers spend on the project, the higher the direct labor cost will be.

In our study, since the topic is pair programming, we are concerned about labor cost on programming tasks and directly related activities such as testing, quality assurance, and rework. As developers spend more time on the initial programming, the labor cost increases. Once the initial programming is done, testing and quality assurance take place with the sole purpose of detecting problems in the program. As defects are uncovered, developers will incur rework time to fix the problems. Therefore, labor cost increases as developers spend more programming hours on the project, and labor cost decreases as there are fewer defects in the system.

The other category of cost is training cost. Trainings are mechanisms to educate the developers so they have the skills and information to successfully complete the project. Trainings are essential when developers do not have the technical skills to implement the project, do not have a good understanding of the system, and/ or do not have good problem solving skills. We do not expect knowledge transfer between the pairs to replace all trainings. However, we argue through watching, discussing, and learning from each other knowledge transfer between the pairs help developers acquire technical skills, problem solving skills, and understandings of the modules and the overall system, thus, reducing training cost.

Opportunity cost is defined as costs associated with opportunities an organization loses for undertaking the project. An example of opportunity cost is the profit the organization could have made by working on a different project. With a short-term project, organizations can jump into other business opportunities rather quickly, thus minimizing opportunity cost. When a project goes on forever, an organization's resources are tied by the project, therefore does not have extra resources available to undertake other projects. As a result, opportunity cost increases.

Labor slack occurs when developers are not working on a project. That happens when a task/project is finished, but the other task/project is not scheduled to start. Some labor slacks are designed by organizations for purposes of re-training and/or re-energizing the workforce, while others are neither scheduled nor anticipated. We argue as pair

programmers finish the task/project sooner than solos, it is more likely for the pair programmers to have downtime between tasks/projects.

Other costs include costs of hardware, software and computer networks needed to support the project, costs of office space, utilities, central facilities, and employee benefits like pensions and health insurance (Sommerville, 2007). These overhead costs tend to be fixed. In this study, we consider total cost of a development project as a function of labor cost, training cost, opportunity cost, and labor slack. The total cost changes as any of the four costs changes.

METHODOLOGY

Two studies will be conducted. The first study is a survey with two objectives: 1). Gather information on practitioners' perception on the cost of pair programming and to provide an initial validation of the research model; 2). Acquire data on project and developer characteristics so we can use those data as parameters for the simulation models, which is the second study.

Due to its originality, except for the few knowledge transfer related questions we are able to tailor from Ko, Kirsch, and King (2005), all the other questions are created from scratch. For each construct in the research model, multiple questions are designed to seek good understanding of the construct from different angles. The questionnaires are designed online using Survey Monkey. Edits are placed to ensure answers' completeness. The following is an example of the type of questions asked. In this example, answers to all questions will be used to test hypothesis 1 in the research model and the answer to the last question is a parameter for the simulation models. The complete questionnaire is available at www.business.ku.edu/pair.

Compared to a solo programmer, two programmers working as a pair will
Spend more time weighing different alternatives.
Double the total programming hours.
Please estimate the percentage difference in programming effort between solo programming and pair programming.

Table 1. Sample Survey questions

By using Straub's (Straub, 1989; Boudreau, Gefen, and Straub, 2001) list of questions, the survey instrument will be validated in the following areas: content validity, construct validity, and reliability. To be more specific, we will seek expert opinions and conduct a pilot study before the survey is officially fielded; confirmatory factor analysis will be conducted to ensure the instrument has convergent and discriminant validity; reliability coefficients (Kline, 2005) will be generated to ensure the instrument's reliability. The survey respondents will be practitioners who use and do not use the pair programming method.

In regards to sample size, some rough guidelines are offered by Kline (2005) for studies using SEM analysis. He suggests less than 100 would be considered as small, 100-200 would be a medium sample size, and greater than 200 cases would be considered large. Besides general guidelines, numerous approaches have emerged to conduct power analysis in the SEM framework, among which MacCallum, Browne, and Sugawara (1996) have been considered the "work" on SEM power analysis. Instead of focusing on parameter testing, their method concentrates on model testing and ignores the number of factors that affect power. They use the root-mean-square error of approximation (RMSEA) and the test of poor fit. Their method answers two questions: What is the power to reject an incorrect model (H0: RMSEA >= 0.05)? What is the power to detect a reasonably correct model (H0: RMSEA <= 0.05)? Their analysis results are in line with the general guideline. 100-200 cases are a reasonable sample size.

The other method is a series of simulation studies. Its objective is to use responses from the survey as input parameters and to determine in what situations pair programming is more cost effective than solo programming. We will examine the project and developer characteristics and incorporate these data into simulation models. We will vary parameter values during the simulation study to investigate the effects of programming methods, system complexity, programmer expertise, and pair composition on the overall cost of pair programming. Table 2 below describes the scenarios we will test in the simulation models.

Method	Pair		Solo
Expertise	Low	Low	Low
	Low	High	Low
	High	High	Low
	Low	Low	High
	Low	High	High
	High	High	High
Pair Programming Experience	No	No	N/A
	Yes	No	N/A
	Yes	Yes	N/A
System Complexity	Low, Medium, High		
Duration/Effort/Defect/Knowledge	Distribution Based		

Table 2. Simulation Scenarios

EXPECTED CONTRIBUTIONS

We believe this study can provide several theoretical and practical implications for researchers as well as practitioners. From a theoretical perspective, this study is expected to be the first one to conduct a systematic survey in pair programming with several under-studied constructs: prior pair programming experience and knowledge transfer. All the cost-related constructs are new. We hope the questionnaire we develop and validate will provide basis for future research in this area.

From the practical perspective, the study will provide guidelines to the industry when one needs to decide which programming method to adopt in different project environments. The purpose of the simulation study is to use the various ranges of parameter values given by the practitioners on the survey and to identify three situations and their associated project and developer characteristics: situation 1 - pair programming is more cost effective than solo programming, situation 2 – the two methods are pretty much the same, and situation 3 – solo programming is more suitable than pair programming. Such findings are expected to provide suggestions to organizations in regards to which programming method to use given their unique project and developer characteristics.

REFERENCES (THIS IS A PARTIAL LIST – THE COMPLETE LIST IS AVAILABLE UPON REQUEST)

1. Arisholm, E., Gallis, H., Dybå, T., and Sjøberg, D.I.K. (2007) Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise, *IEEE Transactions on Software Engineering, 33, 2,* 65-86.
2. Balijepally, Mahapatra, Nerur, and Price (2009) Are Two Heads Better than One for Software Development? The Productivity Paradox of Pair Programming, *MIS Quarterly*, 33, 1, 91-118.
3. Cusumano, M., MacCormack, A., Kemerer, C.F., and Crandall, B. (2003) Software Development Worldwide: The State of the Practice, *IEEE Software,* 20, 6, pp 28-34.
4. Dawande, Johar, Kumar, and Mookerjee (2008) A Comparison of Pair Versus Solo Programming
5. Under Different Objectives: An Analytical Approach, *Information Systems Research*, 19, 1, 71-92.
6. Dybå, T., Arisholm, E., Sjøberg, D.I.K., Hannay, J.E., and Shull, F. (2007) Are Two Heads Better than One? On the Effectiveness of Pair Programming, *IEEE Software*, 24, 6, 12-15.
7. Erdogmus, H., and Williams, L. (2003) The Economics of Software Development by Pair Programmers, *Engineering Economist*, 48, 4, 283-319.
8. Gallis, H., Arisholm, E., and Dybå, T. (2003) An Initial Framework for Research on Pair Programming, *Empirical Software Engineering*, 132-142.
9. Padberg, F., and Müller, M.M. (2003) Analyzing the cost and benefit of pair programming, *Software Metrics Symposium*, 166-177.
10. Sommerville, I. (2007) Software Engineering, Addison-Wesley.
11. Williams, L., Kessler, R.R., Cunningham, W., and Jeffries, R. (2000) Strengthening the Case for Pair Programming, *IEEE Software*, 17, 4, 19-25.

APPENDIX
Literature review summaries

Study	# of Subjects	Subject Type	Statistical Testing	Dependent variables									
				Higher Quality	Higher Productivity	Less Effort/ Total Time	Shorter Duration	Higher Confidence	More Enjoyment of the Process	Better Knowledge Transfer/learning	Higher Course Pass Rate	Compatible Exam Score	Higher Retention
Arisholm et al. (2007)	298	I	Y	P		N	P						
Canfora et al. (2007)	18	I	Y	P		P							
Hulkko and Abrahamson (2005)	20	I	dc	N	N								
Lui and Chan (2003)	15	I	dc	Y		Y	Y						
Nosek (1998)	15	I	Y	Y		N		Y	Y				
Parrish et al. (2004)*	48 Modules	I	Y		N								
Rostaher and Hericko (2002)	16	I	Y			N	N						
Al-Kilidar et al. (2005)	121	S	Y	P									
Balijepally et al. (2009)	120	S	Y	P				P	Y				
Canfora et al. (2005)		S	Y			Y							
Ciolkowski and Schlemmer (2002)	55	S	Y	N		N			Y				
Gehringer (2003)	96	S	Y	P					Y				
Hanks et al. (2004)	150	S	Y	P				Y	Y				
McDowell et al. (2002)	600	S	Y	Y							Y	Y	
McDowell et al. (2003)	288	S	Y	Y		N						Y	
McDowell et al. (2003, 2006)	554	S	Y	Y				Y	Y		Y	Y	Y
Madeyski (2006)	188	S	Y	N									
Mendes et al. (2005)	300	S	Y	Y					Y		Y	Y	
Müller (2006)	18	S	Y	N		N							
Nagappan et al. (2003a-b)	495	S	Y	Y		Y			Y		Y	Y	
Nawrocki and Wojciechowski (2001)	15	S	dc		N	N							
Vanhanen and Lassenius (2005)	20	S	Y	N	N				P	Y			
Williams et al. (2000) Williams and Kessler (2001 Exp #2)	41	S	Y	Y		P	Y	Y	Y		Y	Y	
Williams and Kessler (2000b, 2001 Exp1)	20	S	dc	Y				Y	Y				
Williams et al. (2003)	1200	S	Y	P					Y			Y	Y

Table 3. Summary of Pair vs. Solo Experimental Studies

(S=Student I=Industrial Practitioner Y=Yes, N=No, dc=Descriptive Statistics Only, P=Partial Support)
*Instead of solo vs. pair, this study compared high vs. low concurrency as described in the text.

Study	Subject Type	# of Subjects	Stat Test	Benefits								Issues			
				Quality	Duration	Productivity	Confidence	Enjoyment/ Satisfaction	Knowledge Transfer/ Learning	Team Work	Problem Solving	Compatible Exam Score/ Retention	Schedule Conflict	Pair not compatible	Unequal Workload
Succi et al. (2002)	I	108	Y					Y							
Vanhanen and Korpi (2007)	I	4 cases	Y	P		D		P	Y	Y					
Vanhanen and Lassenius (2007)	I	28	Y	Y	Y			Y	Y	Y					
Williams and Kessler (2000)	I	Not reported					Y	Y							
Cliburn (2003)	S	17	N					Y				Y	Y	Y	
Declue (2003)	S	24	N	Y							Y		Y	Y	
Janes et al (2003)	S	15	Y						Y						
Müller & Tichy (2001)	S	12	N					Y	Y						Y
Sanders (2001)	S	60	N							Y	Y	Y	Y	Y	Y
Srikanth et al. (2004)	S	287	dc	Y						Y			Y	Y	Y
VanDeGrift (2004)	S	546	dc			Y		Y			Y		Y	Y	Y
Williams (1999)	S	20	dc	Y		Y	Y	Y	Y						
Xu and Rajlich (2006)	S	12	dc	Y			Y				Y		Y	Y	

Table 4. Summary of Survey Studies

(S=Student I=Industrial Practitioner Y=Yes, N=No, dc=Descriptive Statistics Only, P=Partial Support)