4-18-2013

What Use Is a Backseat Driver? A Grounded Theory Investigation of Pair Programming

Danielle Latrice Jones

WHAT USE IS A BACKSEAT DRIVER?
A GROUNDED THEORY INVESTIGATION OF PAIR PROGRAMMING

by

Danielle Latrice Jones

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Major: Computer Science

The University of Memphis

May 2013

ACKNOWLEDGMENTS

We would like to thank our family for their love and support, our major advisor, Dr. Scott Fleming, for his encouragement, and our mentor, Keith Humphrey, for his guidance. Without all of you, we would not be where we are today.

ABSTRACT

Jones, Danielle Latrice. M.S. The University of Memphis. May 2013. What Use Is a Backseat Driver? A Grounded Theory Investigation of Pair Programming. Major Professor: Dr. Scott D. Fleming

Over the past fifteen years, numerous studies have pointed to the considerable potential of pair programming (e.g., improving software quality). Using the technique, two programmers work together on a single computer, and take turns driving, typing and controlling the mouse, and navigating, monitoring the work and offering suggestions. However, being a complex human activity, there are still many questions about pair programming and its moderating factors. In this paper, we report on a grounded theory study of seven pairs that addresses open questions regarding partner teaching, navigator contributions to tasks, the impact of partner interruptions, and navigator engagement in the task. Key findings of our study included (1) that all pairs exhibited episodes of teaching, (2) that navigators contributed numerous ideas to the task that were acted upon (most without discussion), (3) that pairs exhibited almost no indications of partner disruption to their flow, and (4) that navigators rarely disengaged.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In the past fifteen years, pair programming has demonstrated considerable promise as a technique for enhancing both software engineering education and practice. In pair programming, two programmers work together on a single computer (often sharing one keyboard and one mouse) collaboratively performing programming tasks [1]. At a given time, one of the programmers plays the role of *driver*, actively typing and controlling the mouse, and the other plays the role of *navigator*, attentively monitoring and checking the driver's work, offering suggestions, and asking clarifying questions. Numerous benefits have been ascribed to pair programming. For instance, Turing Award recipient Fred Brooks described an experience pair programming with a fellow graduate student: "[we] produced 1500 lines of defect-free code; it ran correctly on the first try" [2, p. 8]. Recent studies have found that pair programming can improve software quality [3, 4, 5, 6, 7, 8, 9, 10], that pairs complete tasks faster [11, 12, 5, 7, 8], that pairing leads to increased programming self-efficacy (i.e., the confidence a programmer has in his/her own ability to accomplish a programming task) [4, 13, 7, 8], and that programmers enjoy pair programming [4, 14, 15, 13, 16, 7, 9].

Despite this positive evidence, pair programming remains among the most controversial of development practices. For example, Extreme Programming (XP) advocates the practice of pair programming, with the rationale that the practice yields more well thought out code faster. [17]. However, many practitioners have expressed doubts about whether the practice is in fact more cost efficient than programming individually [18, 19, 20]. Moreover, some studies have contradicted the findings of benefit. For instance, one study indicated that pair programming has no positive effect on development time [21], and two more studies found that pair productivity varied over projects [22, 5].

This controversy no doubt arises because pair programming is a rich, complex human activity with many potential moderating factors, which are not well understood. As Chong and Hurlbutt put it, "our understanding of pair programming as a practice is, at best, nascent" [23]. The empirical evidence to date has tended to focus on byproducts and outcomes of pair programming, with relatively few studies directly examining the activity in detail.

To help fill this gap, we applied a grounded theory approach [24], conducting an observational study of senior undergraduate and graduate students working in pairs on a debugging task. We chose this population because they are at or near a stage where students commonly enter the workforce and may be exposed to pair programming for the first time. Williams et al. [7] argue that programmers go through an initial adjustment period from solo to pair programming, and once a pair has *jelled*, pair productivity increases significantly. However, the *pre-jelled* state may also be important, for example, because two individuals are learning how the other works for the first time. We focused on pairs who were working together for the first time, thus our sample consisted of pre-jelled pairs. The goal of the study was to refine our understanding of how pre-jelled pairs behave during pair programming and reveal promising directions for future research.

To guide our investigation, we focused on the following research questions.

RQ1: (a) to what extent do partners teach one another, and (b) what types of knowledge do they teach?

RQ2: (a) to what extent do navigators contribute ideas to the task at hand; (b) what types of ideas do they contribute; and (c) how do pairs respond to those ideas?

RQ3: (a) to what extent are interruptions by one partner that disrupt the other's flow an issue, and (b) what strategies do pairs use to mitigate interruptions?

RQ4: (a) to what extent do navigators disengage from the task during pair programming, and (b) what strategies do pairs use to facilitate engagement?

RQ5: After working with a partner for the first time, what impressions did participants have of (a) the benefits and (b) the problems with pair programming?

Regarding RQ1, pair programming has been touted as benefiting learning and spreading knowledge [12, 8]; however, no prior studies have looked at how pairs teach one another or the types of knowledge they exchange. Participants in one study reported having learned from their partners, but the study did not observe the learning firsthand [8]. Another study analyzed the communication among "side-by-side" programmers, who work independently on separate machines positioned next to each other, and found that participants exchanged project details and general knowledge [25], but the study did not investigate the standard pair programming technique. Additionally, prior studies have tended to emphasize the benefits of jelled pairs; however, the pre-jelled period may be particularly important for teaching because it is each partner's first exposure to how the other works.

Regarding RQ2, the thing that most separates pair programming from solo programming is the introduction of the navigator, but what does the navigator really contribute to tasks? Since the navigator generally lacks direct control of the activity, it stands to reason that the navigator's main contribution to the task will be ideas and suggestions. However, studies have found that the traditional characterization of the navigator as a strategist, thinking about the task at a high level of abstraction, is false, and that navigators approach tasks in a manner more similar to the drivers [26, 23]. Moreover, some programmers have indicated feeling more engaged in the task when they have control of the keyboard and mouse [23], and one study found that it was not uncommon for navigators to disengage from the task periodically [27]. But none of these prior studies has looked specifically at the types of ideas that the navigator offers and how pairs respond to those ideas.

Regarding RQ3, it has long been held that to maximize productivity it is important for a programmer to enter *flow*, which is a mental state marked by heightened concentration and full emersion in an activity [28]; but what impact does pairing have on flow? Flow is notoriously difficult to achieve in the presence of noise and distractions, and it is unclear the extent to which a partner might inhibit flow. For instance, as one anonymous developer told the authors: "The team I'm on right now is big on pair programming, and it's driving me *crazy*. Subjectively, I feel like having someone sitting over my shoulder interrupting all the time makes it very difficult to hold the pieces of a problem or design in my head." In contrast to this view, Belshee [29] argues the existence of *pair flow* in which partners jointly achieve a flow state; however, the evidence to date for pair flow is merely anecdotal. Thus, our study investigated the extent to which partner interactions disrupt concentration in pair programming.

Regarding RQ4, the driver is inherently engaged in the activity but the navigator may disengage from the task, for example, if his/her attention wanders. However, the navigator's ability to contribute to the task depends heavily his/her awareness of the activity. *Activity awareness* is knowledge of a collaborator's actions (what he/she is doing) and intentions (why he/she is doing it), and it is well known that activity awareness is important in computer-supported cooperative work [30, 31]. The literature contains conflicting evidence regarding navigator engagement in pair programming. In one study, Chong and Hurlbutt [23] found some pairs were so mutually engaged in the task that they did not need to speak in complete sentences to communicate. However in the same study, one participant felt more engaged when in control of the keyboard; but when he felt a switch was not imminent, he was less engaged. Another study by Plonka et al. [27] found that disengagement for short periods may not significantly impact pair effectiveness; however, navigator disengagement led to a loss of activity awareness in some cases. Given these conflicting findings, our study investigated the extent to which navigator disengagement impacts pre-jelled pairs.

Regarding RQ5, software professionals have expressed numerous concerns about pair programming. For example, Begel and Nagappan [18] surveyed 487 professional developers at a large software company, and those developers expressed numerous concerns, such as whether pair programming is cost effective, and whether personality clashes and disagreements are a common problem. However, most of the developers surveyed had never pair programmed before. Our study replicated the Begel study except that we focused on new pairs who have just worked together for the first time.

The remainder of the paper is organized as follows. Chapter 2 provides background on prior work regarding the benefits and moderating factors of pair programming. Chapter 3 describes our study method. Chapters 4–8 report the results associated with each of our research questions along with discussion. Chapter 9 concludes with discussion of future directions.

CHAPTER 2

BACKGROUND: PAIR PROGRAMMING

2.1 Potential Benefits

As mentioned in Chapter 1, the literature suggests several key benefits of pair programming. In this section, we describe the work regarding these benefits in more detail.

2.1.1 Product Quality

A number of studies have found that pairs produce higher quality software than solos. Even as early as the late 90's, it was noticed that pairs produce more readable and functional solutions than solos [5]. A subsequent series of experiments involving hundreds of undergraduate college students compared the work of student pairs versus solo programmers. In one of the experiments, pairs' class projects passed significantly more automated tests than did solos' [7], and in another of the experiments, student pairs scored significantly higher on their projects than did solos [4]. A meta-analysis (by Dybå et al.) of four studies of professionals and eleven studies of students found general agreement among the studies that pair programming improves software quality over solo programming, with small to medium effect sizes [3]. But not all results regarding professionals have agreed: One study of 295 professional programmers found only a seven percent (insignificant) improvement in the correctness of pair-produced programs over those of solos [11]. In contrast, a different study of 17 professional developers found that software produced with at least some pair programming had significantly lower defect densities than software produced by only solo programmers [32].

2.1.2 Task Efficiency

Studies have also suggested that pairs produce this higher quality work more efficiently than do solo programmers. For example, one early study of 15 professional programmers found that pairs completed tasks faster that solos; however, the difference was not statistically significant [5]. Not all studies have shown positive results for task efficiency.

For example, one study of 21 students found no difference in the time it took to complete tasks between pair and solo programmers [21]. However, the Dybå meta-analysis found that pair programming had a medium-sized overall reduction of the time to complete tasks, compared with individual programming [3].

2.1.3 Self-Efficacy

A consistent result has been that pairs report higher confidence in their work than solos. A study of 15 professionals found that pairs were significantly more confident in their work than solos [5], a result echoed by a subsequent study of 554 undergraduate college students [33]. Confidence is particularly important because research has shown that individuals with high self-efficacy (a person's confidence in their ability to perform a particular task) tend to be more persistent and flexible in their problem solving, compared to individuals with low self-efficacy [34].

2.1.4 Knowledge Transfer

A final potential benefit of pair programming that the literature suggests is knowledge transfer between programmers. One study of 41 undergraduate programmers offered a characterization of the pairs' information exchange: "Knowledge is constantly being passed between partners, from tool usage tips (even the mouse), to programming language rules, design and programming idioms, and overall design skill" [12]; however, this characterization was only anecdotal. In another study of 20 undergraduate students, 84 percent of participants agreed subjectively with a statement that they had learned a topic better because they were working with a partner [8]. In the professional realm, an ethnographic study of two teams of professional developers observed instances where developers who knew more about a particular task brought developers with less knowledge up to speed, thus narrowing the gap in their knowledge [23]. However, neither of these studies analyzed the types of knowledge being taught. In contrast, a grounded theory study of three "side-by-side" programmers (students) observed instances of the

programmers exchanging project-related and general knowledge [25]; however, this study did not investigate pairs engaged in the standard pair programming technique.

2.2 Possible Moderating Factors

As pair programming is a complex human activity, it is perhaps unsurprising that the literature discusses a number of factors that may influence the potential benefits of pairing.

2.2.1 Pair Jelling

The literature contains some evidence that pairs may go through an adjustment period when they first work together [7]. After the pair has adjusted, or *jelled*, they perform tasks considerably more efficiently than before. Because pairs perform better when jelled, most prior research has tended to focus on jelled partners (or to not mention jelling at all). In our study, we intentionally focus on new partners to better understand the initial pairing process before jelling occurs.

2.2.2 Pair Composition

The literature contains numerous studies of how pair performance is influenced by various attributes of each partner. Surprisingly, personality traits have not been a strong indicator of pair performance. For instance, one study of 196 professional developers found that participants' personality-test results were not strong indicators of how pairs performed [35]. Similarly, another study of 218 undergraduate CS students found that differences in conscientiousness level did not significantly affect the academic performance of students who pair program [13]. In contrast, combinations of partner expertise have been indicators of pair performance. For example, several studies have put forth evidence that individuals with similar expertise levels tend to make more successful pairs (e.g., [11, 2]). Moreover, one such study found that lower expertise pairs were generally as successful as higher expertise pairs on high complexity tasks [11].

2.2.3 Engagement

Navigators' engagement in the task (i.e., the amount of attention they give it) has also been shown to impact pair performance. For example, a study of 31 professional

developers found that, although navigator disengagement was sometimes useful, there were also instances where such disengagement led the navigator to be unable to follow the driver's action and to be unable to contribute to the task at hand [27]. While another study of professional developers from 2 different companies found that, programmers were more engaged when in control of the keyboard; thus switching fostered engagement and effectiveness [23].

2.2.4 Flow

The concept of *flow* has long been held as important to successful development [28]. When a developer is in a flow state, he/she is fully immersed in his/her task, and achieves a state of heightened concentration and productivity. More recently, the concept of *pair flow* has been proposed wherein a pair of developers working together achieve the flow state [29]. However, we could find no empirically grounded characterization of pair flow in the literature. Chong and Siino [36] studied the types of external interruptions (such as phone calls) that disrupt pair programming sessions; however, they did not investigate how simply having one partner talking to the other might disrupt the other's concentration and flow. Thus, it is an open question the extent to which partners interrupt each other's flow, and we address this question with our RQ3.

CHAPTER 3

METHOD

3.1 Participants

Participants in our study comprised fourteen students (seven pairs) enrolled in CS courses at the University of Memphis. Four were senior undergraduate students enrolled in the CS capstone course. The other ten were graduate students enrolled in a graduate-level software engineering course. All participants had programmed in Java before (experience programming in any language: mean = 3.67 years, standard deviation = 2.11; Java experience: mean = 2.17 years, standard deviation = 1.47). Four participants had programmed professionally (median professional experience = 1.5 years). Two of the fourteen participants had pair programmed before in their undergraduate courses. Table 3.1 lists the participants' background information. We assigned each participant an identifier of the form P<pair number><gender><partner number>. For example, participant P1M1 was male 1 from pair 1. For pair 4, we omitted participant numbers because the partners were of different genders.

3.2 Task and Environment

The primary pair programming task consisted of finding and fixing a bug in jEdit [37], a Java-based open source text editor. The defect came from an actual bug report (#2548764) [38] and involved a problem with jEdit's text "folding" functionality. The jEdit code base comprised 96,713 source lines of code. In the background questionnaire, participants indicated that none were familiar with jEdit.

Pairs worked side by side at a workstation with a 24" wide-screen monitor, one keyboard, and one mouse. Their programming environment consisted mainly of the Eclipse IDE, although they could also browse the Web or use any tools commonly found on a Windows PC.

Table 3.1: Participant background information.) .

ID	Sex	Age	Major	Years of Programming Experience		
				Total	With Java	As professional
P1M1	M	20s	CS	8	3	NA
P1M2	M	20s	CS	4	3	NA
P2M1	M	40s	CS	4	4	0
P2M2	M	20s	CS	4	4	0
P3M1	M	20s	MIS	1	1	0
P3M2	M	20s	CS	4	3	1
P4M	M	20s	CS	6	1	2
P4F	F	20s	CS	1	1	1
P5F1	F	20s	CS	1	1	0
P5F2	F	20s	CS	3	2	NA
P6M1	M	20s	CE	3	1	0
P6M2	M	20s	ME	5	1	0
P7M1	M	40s	CS	2	2	0
P7M2	M	20s	CS	4	4	4

3.3 Procedure

We randomly partitioned the participants into pairs with the constraint that pairs had to have compatible schedules. Each pair participated in a session that was at most 2.5 hours in length and took place in a closed laboratory. For each session, we collected screen-capture video, video of the participants, and audio of their utterances.

At the beginning of a session, each participant filled out a background questionnaire. Next, the participants completed a 15-minute pair-programming tutorial and practice pair-programming exercise. The participants then worked on the main pair-programming task for 110 minutes. The task was sufficiently challenging that no participants finished in the allotted time. Lastly, the participants completed a post-questionnaire that asked for their opinions of pair programming and of their experiences during the session.

3.4 Analysis

To analyze the data, we used the *grounded theory* method [24]. Grounded theory is a systematic method for developing theories through the bottom-up analysis of data. We applied the method to our qualitative video data and questionnaire data.

Central to grounded theory is the *coding* of data. In coding, a researcher identifies points in the data where certain concepts/phenomena are apparent, and marks those points. In particular, the identification of key concepts/phenomena happens through an iterative *open coding* process. As the researcher immerses him/herself in the data, he/she identifies and codes concepts. *Analytic tools*, which consist of thinking techniques, support the coding process. For example, in *constant comparison*, with each piece of data that the researcher analyzes, he/she considers how that data is similar to or different from the other pieces of data seen so far. As new concepts emerge, the researcher revisits previously analyzed data and recodes those data to ensure that the new concepts are captured. For our video data, we used our research questions as a guide, and coded by watching and re-watching the videos, annotating them with the concepts we observed.

CHAPTER 4

RQ1 RESULTS AND DISCUSSION: PARTNER TEACHING

4.1 Results

As the far right column of Table 4.1 shows, all pairs exhibited teaching episodes. Each episode of teaching involved one partner instructing the other in, for example, how to do something or how something worked. Note that the audio quality for Pair 3 was poor, potentially making some of their teaching episodes inaudible.

Table 4.1: Frequencies of partner-teaching episodes.

Pair	General Development			Project-Specific			Grand total
	Programing	Tool	Pair total	Bug	Code	Pair total	
P1M1	0	1	1	1	0	4	5
P1M2	0	0		2	1		
P2M1	0	0	4	0	2	3	7
P2M2	3	1		0	1		
P3M1*	0	0	0	0	0	2	2
P3M2*	0	0		2	0		
P4M	1	0	1	0	0	1	2
P4F	0	0		1	0		
P5F1	1	1	3	4	0	7	10
P5F2	1	0		0	3		
P6M1	0	5	6	3	0	4	10
P6M2	0	1		1	0		
P7M1	0	0	3	2	1	4	7
P7M2	0	3		0	1		
Mean	0.4	0.9	2.6	1.1	0.6	3.6	6.1
Std. Dev.	0.9	1.5	2.1	1.3	0.9	1.9	3.3

* Some data lost due to poor audio quality.

The topics partners taught can be divided into two categories: general development knowledge and project-specific knowledge. General development knowledge is applicable in a wide variety of software development contexts, whereas project-specific knowledge tends to be applicable to only one particular project.

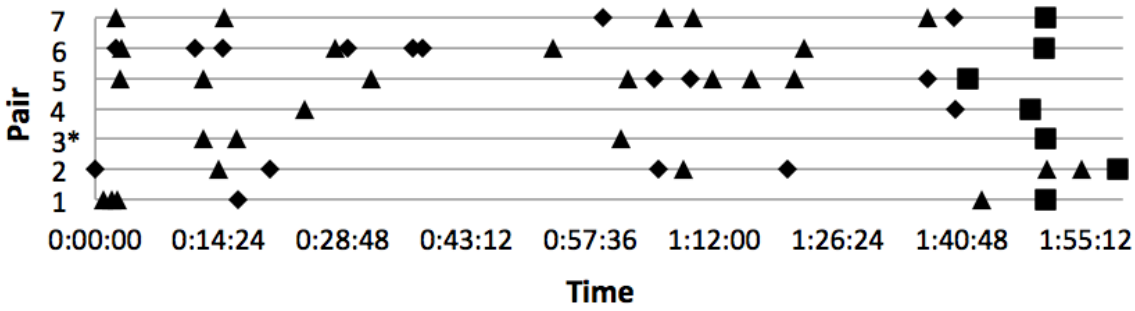


Figure 4.1: Timeline of teaching moments. ◇ indicates general knowledge; △ indicates context-specific knowledge; and □ marks the end of the session. * denotes that Pair 3 was difficult to hear/understand, and therefore, teaching moments may have been missed.

4.1.1 General Development Knowledge

As Table 4.1 shows, participants taught about two main types of general development knowledge: how to use development tools (in this case, features of Eclipse) and how to use the programming language (in this case, Java).

Tool knowledge taught included keyboard shortcuts, how to perform a code search on the entire project, and how to use the breakpoint debugger. For example, P6M1 (as driver) taught P6M2 (as navigator) about breakpoints:

> P6M1: If you want a breakpoint, you have to put– [points at screen] If you want a breakpoint here– [points at screen] So you have to put from here. [demonstrates placing and removing a breakpoint; then lets P6M2 try it] . . . This is the break point.

Programming knowledge included Java naming conventions, how to define an inner class, how to use try/catch blocks, and the concept of an "offset." For example, P5F2 (driver) explained to P5F1 (navigator) how to write a Java catch statement that catches all exceptions:

> P5F2: [starts writing a catch block and pauses on entering the type of exception to be caught]

P5F1: If in the catch block you write dots, it will catch every type of exception. I don't know if it's good in Java [it is not], but in C++, you don't have to explicitly mention it, that this kind of exception is–

P5F2: Um, you can just write "Exception," just the word, also. [referring to Java's Exception class]

4.1.2 Project-Specific Knowledge

As Table 4.1 also shows, participants taught about two main types of project-specific knowledge: how to reproduce the bug and the structure of the jEdit code (e.g., where a particular method was located in the code).

Many participants ran into trouble reproducing the bug, evidenced by the number of teaching moments related to the bug. Their difficulty may have arisen because reproducing the bug required precisely following a series of detailed steps, and it was easy to make subtle mistakes. For example, P5F1 corrected P5F2 several times on how to recreate bug, as in the following episode:

P5F2: [types text into jEdit]

P5F1: No, you have to fold that first.

P5F2: Huh?

P5F1: You have to fold it up.

P5F2: Oh yeah. [folds text input]

Code structure knowledge taught included where certain variables were defined, which methods perform a particular function, and how to call a particular method. For example, P5F2 (as navigator) explained that P5F1 had not given the number of parameters required by a method:

P5F2: It's not the right one. [takes over driving] . . . This method accepts two. [corrects the method call] It accepts two integers you just can't give it an array.

4.2 Discussion

Although pre-jelled pairs have generally been characterized as being less effective than jelled pairs in the literature [7] because pre-jelled pairs are less efficient than jelled ones, our results suggest that the pre-jelled period may be a particularly rich time for learning from a partner—all participants but one taught their partner something.

Participants' high number of teaching instances related to development tools are particularly encouraging. Recall (Table 4.1) that all but two pairs exhibited instances of such tool teaching. Modern IDEs contain scads of features, and leveraging these features effectively can help make developers significantly more productive [39]. Unfortunately, in practice, developers frequently take advantage of only a small subset of the features that IDEs have to offer [40]. The cause of this deficiency may be that such skills are typically not explicitly covered by computer science curricula, and developers are left to discover the skills on their own. Our results suggest that pair programming may help open developers up to new tools, helping to fill the education gap, and making them more productive individuals.

Pre-jelled pairs may see particularly strong gains in such tool skills. When a pair first works together, the repertoire of productivity tricks that each partner employs may be quickly revealed and shared. Our participants' high number of tool-teaching instances is consistent with this idea. It stands to reason that the longer two partners work together, there will be diminishing returns on such learning. Thus, a possible implication is that individuals in an organization should pair up—at least a few times—with as many others as possible to maximize dissemination of tool skills.

In addition to the above long-term productivity benefits of pair teaching, the bug-related teaching instances clearly benefited short-term productivity. In every case of teaching about the bug, one partner exhibited a misunderstanding about how to reproduce the bug. Given the subtlety of the steps required to reproduce the bug, an individual might waste considerable time figuring out his/her misunderstanding alone. Having a partner to

quickly identify and clear up the misunderstanding shortcuts this process and speeds up the task overall.

Also noteworthy is that within each pair the teaching instances did not all come from just one partner. With the exception of Pair 3 (for which some data was lost), all participants taught their partners about something. It seems that everyone had something to offer despite substantial differences in the programming experience of nearly all pairs (Table 3.1).

CHAPTER 5

RQ2 RESULTS AND DISCUSSION: NAVIGATOR CONTRIBUTIONS

5.1 Results

5.1.1 Preliminaries: Distribution of Navigator Role

Before we address navigator contributions, we first look at how the partners distributed the role of navigator. As Table 5.1 shows, all participants played the role of navigator at least once; however, the number of turns each partner took as navigator and the ratio of time each partner spent as navigator ranged widely. In our coding, the partner who had control of the keyboard and mouse was the driver, and the other partner was the navigator. Note that a technical failure caused the head/hands video for Pair 7 to be lost. Although it was often clear from the screen-capture video and audio which of the Pair 7 participants was navigator, there were some periods where it was unclear who was driving, and we excluded that data from our analysis.

Table 5.1: Time participants spent as navigator.

Participant	Turns as Navigator	Time as Navigator	Pct. Time
P1M1	16	0:35:58	30%
P1M2	16	1:23:37	70%
P2M1	1	0:11:49	12%
P2M2	2	1:24:14	88%
P3M1	11	0:34:49	32%
P3M2	12	1:14:39	68%
P4M	2	0:00:21	0%
P4F	3	1:39:09	100%
P5F1	9	1:17:41	81%
P5F2	8	0:18:13	19%
P6M1	26	0:43:11	39%
P6M2	25	1:06:17	61%
P7M1*	5	0:50:28	47%
P7M2*	4	0:56:09	53%

* Some data excluded due to missing video.

Table 5.2: Frequencies of ideas contributed as navigator.

Participant	Specific action	Goal/strategy	Total ideas	Ideas per hour
P1M1	3	2	5	8.6
P1M2	12	3	15	10.9
P2M1	0	0	0	0.0
P2M2	19	6	25	17.9
P3M1	2	1	3	5.3
P3M2	6	0	6	4.9
P4M*	-	-	-	-
P4F	18	9	27	16.4
P5F1	13	1	14	10.9
P5F2	2	0	2	6.7
P6M1	26	0	26	36.1
P6M2	2	0	2	1.8
P7M1	14	4	18	21.7
P7M2	14	0	14	15.1
Mean	10.1	2.0	12.1	12.0
Std. Dev.	8.2	2.8	9.8	9.7

* P4M played the role of navigator for less than 30 seconds.

5.1.2 Ideas Offered by the Navigator

As the second to last column of Table 5.2 shows, all participants but one contributed ideas for how to proceed with the task while playing the role of navigator. (And that one participant played the role of navigator for only about 12 minutes of his 1.5 hour session.) In our coding, a navigator contributed an idea if he/she verbally recommended or suggested some action or course of action to the driver.

Also apparent in Table 5.2 is that most ideas offered by navigators were specific actions for the driver to perform. With such ideas, it was always clear exactly what the driver should click, type, etc. The specific actions recommended by navigators can be grouped into six categories (defined in Table 5.3). For example, P6M1 (as navigator) proposed the specific action of searching within a class for the word *delete* using Eclipse's Find utility:

P6M2: [opens the class in the editor]

P6M1: Ctrl-F.

P6M2: [types Ctrl-F, which opens Eclipses Find utility]

P6M1: Just type “delete.”

P6M2: [types “edit”]

P6M1: No, use “delete” because the title itself is “jedit.”

P6M2: [types “delete” and executes the search]

Table 5.3: Types of specific actions proposed by navigators.

Type of action	Description
Edit code	Make a change to the source code.
Manipulate editor	Change what code is displayed in the editor (e.g., by scrolling or using other navigation features), or close or clean up tabs.
Manipulate program	Execute and/or interact with the program under development (jEdit) in a particular way.
View documentation	Go to a particular piece of documentation.
Search code	Use code search utilities in a particular way.
Manipulate debugger	Add a particular break point, step the debugger, show the current stack state, or show the console output.

In addition to specific actions, navigators also proposed (albeit much less frequently) pursuing broad goals and strategies for which the specific actions were not specified. For instance, Pair 2 was inspecting some source code when P2M2 (as navigator) proposed they pursue the goal of figuring out how two methods work:

P2M2: Get-line-start-offset, or whatever. End-offset.

P2M1: Say that again.

P2M2: Those methods, get-line-start-offset, get-line-end-offset, we need to know what that’s doing.

5.1.3 Responses to Ideas

As Table 5.4 shows, pairs acted upon the ideas offered by navigators much more often than not. In our coding, pairs responded to navigator ideas in one of three ways: (1) acting upon the idea, (2) modifying/refining the idea, and (3) dismissing the idea. In our coding,

pairs acted upon an idea if they took action exactly as specified by the idea. For example, P4M (as driver) was working on reproducing the bug in jEdit, when P4F (as navigator) offered an idea:

P4M: [enters two lines of text into jEdit, the number of lines that the bug report specified was needed to reproduce the bug]

P4F: You can enter more lines so you can see.

P4M: [enters three more lines, as P4F suggested]

Pairs modified/refined an idea if they changed some aspect of the idea, and then took action consistent with the modified idea. For example, P1M2 (as driver) was annotating code with diagnostic print statements when P1M1 offered an idea to add an additional diagnostic if-statement:

P1M1: Right there. End-line minus start-line. [points at the screen] If that is becoming negative– Go ahead and try. . . . I was going to say, like, if less than 0, then 0, or something.

P1M2: [types "if", then deletes it; mumbles; types a diagnostic print statement that displays the number of lines, rather than an if-statement, as P2M1 suggested]

Pairs dismissed an idea if they dropped it without acting upon it. For example, P2M1 (driver) was scrolling in the console, looking at the thrown-exceptions output, when P2M2 (navigator) suggested navigating to and inspecting the method endCompoundEdit, which was referenced in the output:

P2M2: One thing I really don't want to look at is this method [endCompoundEdit], but–

P2M1: [laughs]

. . .

P2M2: Let's look at it one more time.

Table 5.4: Frequencies of actions and discussions in response to navigator ideas.

Pair	Resultant action			Resultant discussion	
	Acted upon	Modified, acted upon	Not acted upon	Discussed	Not discussed
P1M1	3	1	1	1	4
P1M2	13	1	1	1	14
P2M1	0	0	0	0	0
P2M2	23	0	2	2	23
P3M1	2	1	0	1	2
P3M2	6	0	0	0	6
P4M	-	-	-	-	-
P4F	15	0	12	2	25
P5F1	10	0	4	1	13
P5F2	2	0	0	0	2
P6M1	24	0	2	0	26
P6M2	1	0	1	0	2
P7M1	15	0	3	2	16
P7M2	9	1	4	4	10
Total	123	4	30	14	143

P2M1: [highlights the name of the method endCompoundEdit in stack trace, but does not move to open the method]

P2M2: [notices another method] Fire-transaction-complete. Want to look at that method?

P2M1: [opens method fileTransactionComplete in the editor, disregarding P2M2's previous suggestion about endCompoundEdit]

Also apparent in Table 5.4 is that pairs rarely discussed the ideas offered by the navigator. In our coding, pairs discussed an idea if the driver and navigator had a verbal exchange regarding the idea prior to acting upon, modifying, or dismissing the idea. A large majority of the time pairs simply acted upon navigator ideas without any discussion.

5.2 Discussion

Regarding the distribution of the navigator role among participants, Pair 6 stood out as resembling the professional pairs that Chong et al. [23] studied, with their rapid switching.

Chong et al. attributed this behavior to the partners working very closely together on the task. The Chong pairs worked especially closely when their course of action was somewhat unclear, such as during design and debugging activities. Given that our participants were engaged in a debugging task, it is perhaps not surprising that all our partners also worked very closely together.

In contrast to the Chong pairs, several of our pairs were considerably lopsided in their distribution of the driver/navigator roles. In particular, Pairs 2 and 4 switch roles little, with one partner acting predominantly as driver and the other as navigator. These pairs were clearly not behaving consistently with Williams and Kessler's advice that pairs switch roles frequently [1]. But is this lack of switching a sign of pair dysfunction? In the case of Pair 4, it may be, and we will elaborate more below.

The strong tendency of navigators to suggest specific actions (i.e., what to click or scroll) to the driver is a testament to how closely partners worked together. Chong et al. [23] also observed pairs (professionals) working so closely that the partners were practically finishing each other's sentences. Similar to the Chong pairs, our navigators were so engaged in the task and in tune with the context that they made most of their suggestions at the level of what to click next, rather than higher level strategies.

Our navigators' strong tendency to offer ideas for specific actions contrasts with prior findings [26, 25] about the level of abstraction of navigator discourse. In particular, Bryant et al. [26] studied the utterances of professional pairs and coded them based on five levels of abstraction (from lowest to highest). Their study found that navigator discourse was predominantly at a moderate level of abstraction, in which the program was discussed in terms of logical chunks and strategies. However, our navigators' specific-action suggestions were at a lower level of abstraction than logical chunks and strategies.

This difference may be because we looked only at utterances in which navigators offered ideas, but it may also be because of differences between the Bryant pairs and ours. For example, our pairs may have worked more closely together than the Bryant

professional programmers. A study of professional pairs by Plonka et al. [27] found that their navigators often had reason to disengage from the driver's activity, for example, because of interruptions or because they divided up work to be done in parallel with the driver. Our navigators generally did not exhibit such disengagement behavior.

The difference may also be because the Bryant pairs were professionals who had been pair programming for over six months. Thus, their pairs were likely already jelled, and as such, had developed their pair communication such that they could converse using higher levels of abstraction. In contrast, our pairs may not have developed the common vernacular necessary for easy communication at higher levels of abstraction.

Further indication of the closeness with which our navigators and drivers worked together was how pairs responded to navigator ideas: the vast majority of times, pairs acted upon such ideas without any discussion. Chong et al. [23] observed that when pairs were closely in sync, they had a shared context that reduced how much they needed to say to communicate a thought. Such a shared context among our pairs may have in many cases mitigated the need to discuss a navigator idea because the intent was evident to the driver. This shared context may also explain why pairs acted upon so many of the navigators' ideas: by being in tune with driver, the navigators were able to suggest ideas that were closely aligned with the drivers' goals and activities. Thus, drivers found many of the ideas apt and chose to act upon them.

Pair 4 was a notable exception to this trend, and their divergence may have been an indication of pair dysfunction. In particular, the pair dismissed a considerable number of P4F's ideas—44%, the highest dismissal rate of any navigator. This pair was also peculiar in that P4M drove the entire 1.5 hour session (save for 21 seconds). It is difficult to overlook the possible gender implications here because Pair 4 was the only mixed gender pair. A common pattern with the pair was for P4F to suggest an idea and for P4M to ignore her, offering no acknowledgment that she had spoken. To her credit, P4F stayed engaged in the task for the entire session, and was persistent, often voicing an idea several

times (and having it dismissed) before P4M finally acknowledged the idea and acted upon it. Williams and Kessler [2] argue (and we agree) that gender itself is a non-issue in pair programming; however, gender *chauvinism*, an attitude of superiority toward members of the opposite gender, can be an issue. At present, the literature contains relatively little empirical evidence about gender bias and compatibility in pairing, and it is an open question whether chauvinism played a role in P4M's behavior and the extent to which chauvinism is generally an issue in pair programming. In a rare exception, one study of student pairs found that mixed-gender pairs were less likely to report compatibility with their partners than same-gender pairs [41].

CHAPTER 6

RQ3 RESULTS AND DISCUSSION: PARTNER DISRUPTIONS OF FLOW

6.1 Results

In 14 hours of video, only one participant indicated that his partner had disrupted his concentration. We coded an episode as indicating an interruption if a participant gave a clear sign that his/her partner had disrupted his/her thinking. In the one such episode we coded, P7M1 (as driver) was trying to reproduce the bug while simultaneously monitoring jEdit's internal state in the debugger. This activity apparently required concentration because whenever the debugger hit a breakpoint, jEdit froze and become unresponsive. Because the jEdit window usually covered the debugger window, it was not always obvious why the application had frozen. As P7M1 was working through this activity, P7M2 (as navigator) interrupted him:

P7M1: [trying to reproduce the bug, flipping back and forth between jEdit and the debugger]

P7M2: You have to–

P7M1: It's defining it there.

P7M2: You just press F8.

P7M1: Yeah. Let me think for a second. You're kind of pushing me through here.

P7M2: OK.

Complying with P7M1's request, P7M2 waited for P7M1 to finish what he was doing before speaking again.

6.2 Discussion

The lack of partner interruptions in our pairs is encouraging given concerns about the potential interaction between flow and pairing. Our results are consistent with the idea that partner interruptions are infrequent, and therefore, may be relatively easy to manage. For

example, handling such situations in the manner of Pair 7—by simply asking the interrupting partner to hold his/her thought—may be sufficient.

There are several possible explanations for the lack of observed partner interruptions: First, partners may have tended to enter flow state, but not to interrupt each other's flow. Second, participants may have tended *not* to enter flow state in the first place. Third, participants may have been interrupted more than our results indicate, but tended not to give observable indications when it happened. In rest of this section, we discuss the first two of these possibilities in turn.

When two partners work together closely on a task, they may be able to enter and maintain flow without interrupting each other. This idea is consistent with Belshee's notion of *pair flow* [29]; however, Belshee provided neither a detailed characterization of pair flow, nor empirical support for its existence. If the partner was intruding when suggesting ideas, then the lack of discussion supports validates its low disruptiveness [36]. In Chapter 5, our results suggested that the pairs were working together extremely closely. In doing so, a partner may be integrated into the task to the extent that interacting with him/her does not disrupt either partner's flow or take either one out of the task. The interruption we saw with Pair 7 seemed to be a case where one partner was engaged in a sub-activity that was particularly taxing on his cognitive resources. Although our participants encountered few such situations, it is an open question how much those situations present themselves in contexts other than debugging unfamiliar code (e.g., in design tasks). It is also possible that participants were able to listen selectively to their partners as a means of maintaining flow.

It is also possible that partners did not interrupt each other's flow because they tended not to enter flow state. Nakamura and Csikszentmihalyi [42] argue that a sense that one is engaging challenges at a level appropriate to one's capacities is necessary for achieving a flow state. Situations where a person feels inadequate for the task may lead to feelings of anxiety or apathy toward the task, feelings which inhibit flow. Several participants

indicated feeling daunted by the task, and thus, they may have had difficulty entering flow. For example, Pairs 1, 2, and 5 each expressed frustration with the task in the following three episodes:

P1M1: This is frustrating. So much code.

P2M2: The screen is getting blurry.

P2M1: [laughs] No, that is your eyes. [laughs more]

P2M2: Yeah.

P5F2: I'm tired of catching the same exceptions. [both partners laugh]

However, recall (Chapter 3) that studies have shown that pair programming tends to raise programmers' self-efficacy. The fact that often partners' expressions of frustration were responded to with humor and laughter by the pair may be indicative of how having a partner helps to ease anxieties. Thus, an interesting question is whether pairing actually promotes flow by reducing anxieties, which can inhibit flow.

CHAPTER 7

RQ4 RESULTS AND DISCUSSION: NAVIGATOR ENGAGEMENT

7.1 Results

Similar to our results for RQ3, there was only one episode in the over 10 hours of video where the navigator disengaged form the task. In our analysis, we coded only episodes where navigator disengagement was clearly apparent. Apparent disengagement was indicated by periods of distraction that were caused by internal factors (e.g., wandering attention) and/or external factors (e.g. ringing phone).

In the disengagement episode, P5F2 was clearly disengaged from the activity of the driver and was preoccupied. The episode lasted less than 5 minutes because P5F1 asked a direct question to P5F2 that required her to reengage in the activity:

P5F2: [sitting back; fidgeting with hands and hair; yawning]

P5F1: Sleepy?

P5F2: [nods head]

P5F1: Can we fold when we don't have [inaudible]?

P5F2: What? [sits forward]

P5F1: Can we fold when we don't have three lines written there? [begins executing jEdit] When we write some text then we can fold the lines.

7.2 Discussion

Engagement may have been a non-issue due to how closely the pairs worked together on the task. This idea is consistent with the Chong and Hurlbutt [23] finding that engaging was a natural pattern of interaction during pair programming. However, Plonka et al. [27] saw more harmful disengagement episodes than our results indicated. The difference may be because the Plonka pairs were professionals in a real-world work environment and under pressure to be productive. For example, their disengagement may have served to parallelize their work. Our pairs usually followed closely along with each other. This

behavior may be beneficial because it leads to mental models being constructed simultaneously and being highly similar due to the constant communication.

CHAPTER 8

RQ5 RESULTS AND DISCUSSION: PARTICIPANTS IMPRESSIONS

8.1 Results

Based on the post-questionnaire (Appendix A.6) responses, pair programming was well received by our participants. As Table 8.1 shows, their median responses to the Likert questions regarding potential benefits of pair programming were almost all positive. Only one participant, P4M, did not enjoy the experience, rating it a 2 (disagree). Additionally, participants generally agreed with the potential benefits of pair programming based on their experiences (Table 8.2).

Table 8.1: Participant responses to their impressions of pair programing (Likert scale from 1 - strongly disagree to 5 - strongly agree).

Impressions	Median Responses
Enjoyment of technique	4
Using pair programming professional setting	4
Interactions with partner	4
Pair Programming worked for me	4
Pair Programming worked for my partner	4
Confidence in solution	3

However, as Table 8.3 shows, participants had some reservations about pair programming. Our participants agreed with the Begel and Nagappan [18] professionals' opinion that inefficiency is the biggest concern. For example, P4M wrote "Actually, [I] don't think it is more efficient than I do it by myself [sic]." However, in contrast to the Begel findings, our participants were generally not concerned with the other potential problems (responding with neutral or disagree).

Table 8.2: Participant responses to potential benefits of pair programing (Likert scale from 1 - strongly disagree to 5 - strongly agree).

Potential benefits	Median response
Fewer bugs	4
Spreads code knowledge among	4
Higher quality code	4
Can learn from partner	4
Better design	4
Constant code review	4
Two heads are better than one	4
Improved creativity and brainstorming	4
Better testing and debugging	4
Improved morale	4

Table 8.3: Participant responses to potential problems of pair programing (Likert scale from 1 - strongly disagree to 5 - strongly agree).

Potential problem	Median response
Inefficient use of time	4
Differences in preferred programming style causes issues	3.5
Differences in personal style causes issues	3.5
Personality clashes	3
Communication issues between partners	3
Differences in skill level causes issues	3
One partner's interruptions distract both partners	3
Disagreements	2.5

8.2 Discussion

Regarding participants' concerns about efficiency, they may have found the practice more efficient if pair expertise had been taken into account in how participants were paired. (Recall, that pairs were randomly assigned as long as their schedules were complimentary.) For example, P4F commented that "...efficiency could be further improved if both programmers experiences/skill levels could be taken into consideration." Pair configuration has been reported as a factor impacting pair programming effectiveness [19, 41, 43, 2]. For example, a study by Katira et al. [41] showed participants were more compatible with partners of a similar skill level. Thus, the partner composition may have influenced their concerns about efficiency.

CHAPTER 9

CONCLUSION

In conclusion, our grounded theory study has shed new light on several aspects of pair programming: partner teaching, navigator contributions to the task, partner interruptions of flow, and navigator engagement. Key findings of our study included:

RQ1 (partner teaching):

- Partner teaching was common—all participants but one taught their partner something.
- Often the knowledge taught was general development knowledge regarding tools and programming, knowledge in which many developers have been found to have gaps [40].
- The most common knowledge taught was project-specific knowledge regarding how to reproduce the bug, the teaching of which served to clear up subtle misunderstandings that might otherwise have taken considerable time to resolve.

RQ2 (navigator contributions to the task):

- All navigators but one (who played the role for only 12 minutes) contributed ideas to the task, with an average rate of 12 ideas per hour.
- The vast majority of navigator ideas were specific actions for the driver to take, which suggests that navigators may have actually been more like "backseat drivers," working extremely closely with the driver and reasoning about the activity at essentially the same level.
- The vast majority of navigator ideas were acted upon without discussion, which further indicates the closeness with which drivers and navigators worked.

RQ3 (partner disruptions of flow):

- Among all the pairs, there was only one episode where a participant exhibited a clear instance of having his flow disrupted by his partner. By working very closely together, partners may have been so well integrated with each others activities that interruptions were not an issue.

RQ4 (navigator engagement):

- Among all the pairs, there was only one episode identified where a partner became disengaged. Disengagement may have been a non-issue because pairs were constantly communicating about the task.

RQ5 (participant impressions of pair programming):

- Participants overall agreed with the proposed benefits of pair programming.
- However, consistent with prior studies, they also expressed concerns about the efficiency of pair programming.

These findings point to several promising avenues for future research. Our partner-teaching findings suggest that "promiscuous pairing" (pairing with many partners) might increase developer expertise and productivity. It may also be that a few "trysts" with each partner is sufficient to get the main benefit. Others in the literature (e.g., [44]) have discussed the usefulness of promiscuous pairing (at any stage of jelling) for maintaining awareness of the state of the project. However, ours is the first work to suggest the importance of promiscuity among pre-jelled partners for spreading tool skills.

Our teaching findings also shed light on the myriad skills that are not well covered by CS curricula. For example, our pairs shared a substantial amount of tool knowledge in the limited time they were together. Such skills are important because, for example, many software development jobs require high proficiency in certain tools, such as IDEs. Others in the literature (e.g., [45]) have also come across important skills not well covered in CS education; however, their focus was on managerial skills, not technical ones.

Our navigator-contribution and activity awareness findings suggest that navigators follow closely what the driver is doing; however, this leaves open the potential problem of the navigator losing activity awareness of what the driver is doing. For example, the driver might be performing a sequence of actions quickly, and the navigator might lose the thread or not understand the rationale behind the driver's actions. Another study could verify that activity awareness is really a non-issue and even try to understand when and why activity awareness is lost.

Our findings regarding the absence of partner disruptions of flow may also have implications here. In particular, pairs in our study were mainly engaged in program comprehension and bug localization activities. Thus, they were likely coping with uncertainty as they worked—but what if they were engaged in activities for which the path was clearer? In such situations, the driver might roll ahead, applying greater concentration to get through the activity efficiently. Thus, the driver would be more susceptible to interruptions, and the navigator more likely to lose activity awareness. It is an open question whether there are actually different levels of flow, and although programmers in our study might have been in a flow state of sorts, it did not require the level of concentration of more cognitively taxing activities associated with a deeper level of flow.

Further study of the above possibilities could yield considerable implications for the practice of pair programming, and could help software engineering practitioners and educators alike better leverage this promising technique.

REFERENCES

[1] L. A. Williams and R. R. Kessler, "All i really need to know about pair programming i learned in kindergarten," *Commun. ACM*, vol. 43, pp. 108–114, 2000.

[2] L. Williams and R. Kessler, *Pair Programming Illuminated*. Addison-Wesley, 2003.

[3] T. Dybå, E. Arisholm, D. I. K. Sjøberg, J. E. Hannay, and F. Shull, "Are two heads better than one? on the effectiveness of pair programming," *IEEE Softw.*, vol. 24, no. 6, pp. 12–15, 2007.

[4] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald, "The impact of pair programming on student performance, perception and persistence," in *Proc. 25th Int'l Conf. on Software Engineering* (ICSE '03). IEEE Computer Society, 2003, pp. 602–607.

[5] J. T. Nosek, "The case for collaborative programming," *Commun. ACM*, vol. 41, no. 3, pp. 105–108, 1998.

[6] L. Williams and H. Erdogmus, "On the economic feasibility of pair programming," in *Proc. Int'l Workshop on Economics-Driven Software Engineering Research* (EDSER '02), 2002.

[7] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the case for pair programming," *IEEE Softw.*, vol. 17, no. 4, pp. 19–25, 2000.

[8] L. Williams and R. R. Kessler, "The effects of "pair-pressure" and "pair-learning" on software engineering education," in *Proc. 13th Conf. Software Engineering Education and Training* (CSEET '00). IEEE Computer Society, 2000, pp. 59–65.

[9] L. Williams, C. McDowell, N. Nagappan, J. Fernald, and L. Werner, "Building pair programming knowledge through a family of experiments," in *Proc. 2003 Int'l Symp. Empirical Software Engineering* (ISESE '03). IEEE Computer Society, 2003, pp. 143–152.

[10] S. Xu and V. Rajlich, "Pair programming in graduate software engineering course projects," in *Proc. 35th Annual Conf. Frontiers in Education* (FIE '05), 2005, pp. F1G–7–F1G–12.

[11] E. Arisholm, H. Gallis, T. Dyba, and D. I.K. Sjoberg, "Evaluating pair programming with respect to system complexity and programmer expertise," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 65–86, 2007.

[12] A. Cockburn and L. Williams, "The costs and benefits of pair programming," in *Extreme Programming Examined*. Addison-Wesley, 2001.

[13] N. Salleh, E. Mendes, J. Grundy, and G. S. J. Burch, "An empirical study of the effects of conscientiousness in pair programming using the five-factor personality model," in *Proc. 32nd ACM/IEEE Int'l Conf. on Software Engineering* (ICSE '10). ACM, 2010, pp. 577–586.

[14] E. Mendes, L. B. Al-Fakhri, and A. Luxton-Reilly, "Investigating pair-programming in a 2nd-year software development and design computer science course," in *Proc. 10th Annu. SIGCSE Conf. Innovation and Technology in Computer Science Education* (ITiCSE '05). ACM, 2005, pp. 296–300.

[15] N. Nagappan, L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller, and S. Balik, "Improving the cs1 experience with pair programming," in *Proc. 34th SIGCSE Technical Symp. Computer Science Education* (SIGCSE '03). ACM, 2003, pp. 359–362.

[16] L. Thomas, M. Ratcliffe, and A. Robertson, "Code warriors and code-a-phobes: A study in attitude and pair programming," in *Proc. 34th SIGCSE Technical Symp. Computer Science Education* (SIGCSE '03). ACM, 2003, pp. 363–367.

[17] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[18] A. Begel and N. Nagappan, "Pair programming: What's in it for me?" in *Proc. 2nd ACM-IEEE Int'l Symp. Empirical Software Engineering and Measurement* (ESEM '08). ACM, 2008, pp. 120–128.

[19] J. E. Hannay, T. Dybå, E. Arisholm, and D. I. K. Sjøberg, "The effectiveness of pair programming: A meta-analysis," *Inf. Softw. Technol.*, vol. 51, no. 7, pp. 1110–1122, 2009.

[20] F. Padberg and M. Muller, "Analyzing the cost and benefit of pair programming," in *Software Metrics Symposium, 2003. Proceedings. Ninth International*, Sept. 2003, pp. 166–177.

[21] J. Nawrocki and A. Wojciechowski, "Experimental evaluation of pair programming," in *Proc. 12th European Software Control and Metrics Conference* (ESCOM '01), 2001.

[22] H. Hulkko and P. Abrahamsson, "A multiple case study on the impact of pair programming on product quality," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 495–504. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062545

[23] J. Chong and T. Hurlbutt, "The social dynamics of pair programming," in *Proc. 29th Int'l Conf. Software Engineering* (ICSE '07). IEEE Computer Society, 2007, pp. 354–363.

[24] J. Corbin and A. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, 3rd ed. Sage Publications, 2008.

[25] L. Prechelt, U. StŁrk, and S. Salinger, "Types of cooperation episodes in side-by-side programming," in *Proceedings of the 21st Annual Workshop of Psychology of Programming Interest Group* (PPIG '09), July 2009.

[26] S. Bryant, P. Romero, and B. du Boulay, "Pair programming and the mysterious role of the navigator," *International Journal of Human-Computer Studies*, vol. 66, no. 7, pp. 519–529, 2008.

[27] L. Plonka, H. Sharp, and J. v. d. Linden, "Disengagement in pair programming: Does it matter?" in *Proc. 2012 Int'l Conf. Software Eng.* (ICSE '12). IEEE Press, 2012, pp. 496–506.

[28] T. DeMarco and T. Lister, *Peopleware: Productive Projects and Teams*, 2nd ed. Dorset House, 1999.

[29] A. Belshee, "Promiscuous pairing and beginner's mind: Embrace inexperience," in *Proc. Agile Development Conference* (ADC '05). IEEE Computer Society, 2005, pp. 125–131.

[30] J. M. Carroll, D. C. Neale, P. L. Isenhour, M. B. Rosson, and D. S. McCrickard, "Notification and awareness: synchronizing task-oriented collaborative activity," *International Journal of Human-Computer Studies*, vol. 58, no. 5, pp. 605–632, 2003.

[31] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *Proc. 1992 ACM Conf. Computer-Supported Cooperative Work* (CSCW '92). ACM, 1992, pp. 107–114.

[32] N. Phaphoom, A. Sillitti, and G. Succi, "Pair programming and software defects - an industrial case study," in *Proc. 12th Int'l Conf. on Agile Processes in Software Eng. and Extreme Programming* (XP '11). Springer, 2011, pp. 208–222.

[33] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald, "Pair programming improves student retention, confidence, and program quality," *Commun. ACM*, vol. 49, no. 8, pp. 90–95, 2006.

[34] A. Bandura, *Social Foundations of Thought and Action*. Prentice Hall, 1986.

[35] J. E. Hannay, E. Arisholm, H. Engvik, and D. I. K. Sjoberg, "Effects of personality on pair programming," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 61–80, 2010.

[36] J. Chong and R. Siino, "Interruptions on software teams: a comparison of paired and solo programmers," in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, ser. CSCW '06. New York, NY, USA: ACM, 2006, pp. 29–38. [Online]. Available: http://doi.acm.org/10.1145/1180875.1180882

[37] "jEdit - Programmer's Text Editor," http://www.jedit.org/, accessed 2013-03-20.

[38] "jEdit bug report #2548764: Black hole bug is back," http://sourceforge.net/tracker/index.php?func=detail&aid=2548764&group_id=588&atid=100588, accessed 2013-03-20.

[39] N. Ford, *The Productive Programmer*. O'Reilly Media, 2008.

[40] E. Murphy-Hill, R. Jiresal, and G. C. Murphy, "Improving software developers' fluency by recommending development environment commands," in *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng.* (FSE '12). ACM, 2012, pp. 42:1–42:11.

[41] N. Katira, L. Williams, and J. Osborne, "Towards increasing the compatibility of student pair programmers," in *International Conference on Software Engineering 2005* (ICSE '05), 1521, p. 2005.

[42] J. Nakamura and M. Csikszentmihalyi, "The concept of flow," in *The Handbook of Positive Psychology*, 1st ed. Oxford University Press, February 2005, pp. 89–105.

[43] N. Salleh, E. Mendes, and J. Grundy, "Empirical studies of pair programming for cs/se teaching in higher education: A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 37, no. 4, pp. 509–525, 2011.

[44] J. Chong, "Social behaviors on xp and non-xp teams: A comparative study," in *Proc. Agile Development Conference (ADC '05)*, ser. ADC '05. IEEE Computer Society, 2005, pp. 39–48. [Online]. Available: http://dx.doi.org/10.1109/ADC.2005.40

[45] C. B. Simmons and L. L. Simmons, "Gaps in the computer science curriculum: an exploratory study of industry professionals," *Journal of Computing Sciences in Colleges*, vol. 25, no. 5, pp. 60–65, May 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1747137.1747147

APPENDIX A

STUDY DOCUMENTS AND MATERIALS

A.1 IRB Approval Letter

On the following pages, we include the IRB approval letter for the study.

THE UNIVERSITY OF MEMPHIS

Institutional Review Board

To: Danielle Jones

From: Chair, Institutional Review Board
For the Protection of Human Subjects
irb@memphis.edu

Subject: An Investigation of Pair Programming Activities (#2026) (#1016)

Approval Date: March 6, 2012

This is to notify you of the board approval of the above referenced protocol. This project was reviewed in accordance with all applicable statuses and regulations as well as ethical principles.

Approval of this project is given with the following obligations:

1. At the end of one year from the approval date, an approved renewal must be in effect to continue the project. If approval is not obtained, the human consent form is no longer valid and accrual of new subjects must stop.
2. When the project is finished or terminated, the attached form must be completed and sent to the board.
3. No change may be made in the approved protocol without board approval, except where necessary to eliminate apparent immediate hazards or threats to subjects. Such changes must be reported promptly to the board to obtain approval.
4. The stamped, approved human subjects consent form must be used unless your consent is electronic. Electronic consents may not be used after the approval expires. Photocopies of the form may be made.

This approval expires one year from the date above, and must be renewed prior to that date if the study is ongoing.

Chair, Institutional Review Board
The University of Memphis

Cc: Dr. Scott Fleming

A.2 IRB Informed Consent

On the following pages, we include the IRB-approved informed consent document for the study.

THE UNIVERSITY OF MEMPHIS
Dreamers. Thinkers. Doers.

Institutional Review Board

315 Administration Bldg.
Memphis, TN 38152-3370
Office: 901.678.2533
Fax: 901.678.2199

Informed Consent

Principal Investigator: Danielle L. Jones
Study Title: An Investigation of Pair Programming Activities
Institution: Computer Science Department, University of Memphis

Name of participant: _____
Age: _____

The following information is provided to inform you about the research project and your participation in it. Please read this form carefully and feel free to ask any questions you may have about this study and the information given below. You will be given an opportunity to ask questions, and your questions will be answered. Also, you will be given a copy of this consent form.

Your participation in this research study is voluntary. You are also free to withdraw from this study at any time. In the event new information becomes available that may affect the risks or benefits associated with this research study or your willingness to participate in it, you will be notified so that you can make an informed decision whether or not to continue your participation in this study.

For additional information about giving consent or your rights as a participant in this study, please feel free to contact the IRB at 901-678-2533 or email irb@memphis.edu.

1. **Purpose of the study:**
   You are being asked to participate in a research study because we would like to observe experienced developers engaged in pair programming to gain a better understanding of the practice (e.g., its benefits and liabilities).

2. **Description of procedures to be followed and approximate duration of the study:**
   As a participant in the study, you will take part in a session that lasts no longer than 2.5 hours. For the majority of the session, you will engage in a pair programming task with another participant. Prior to the task, there will be a short tutorial on pair programming. You will also be asked to fill out a questionnaire and take part in a short interview during the session.
   You will be videotaped throughout the session. After the session, the video data will be stored on a computer in an office on campus. To protect the data, the computer will be password protected, and the office will be kept locked. When our analysis of the data is complete, the data will be destroyed. As an additional constraint, the data will be destroyed within two years of when it was collected.

3. **Expected costs:**
   2.5 hours of your time.

4. **Description of the discomforts, inconveniences, and/or risks that can be reasonably expected as a result of participation in this study:**
   There is only a minimal psychological and social risk stemming from judgment of the your performance on the task. To further minimize this risk, lab access during each session will be restricted to only the investigators and participants. Moreover, all data will be kept confidential within limits of law, and any data shared with the research community (e.g., via publications) will be anonymized such that all information that might identify you is removed.

5. **Compensation in case of study-related injury:**
   U of M does not have a fund set aside for compensation in the case of study related injury.

THE UNIVERSITY OF MEMPHIS
Dreamers. Thinkers. Doers.

Institutional Review Board

315 Administration Bldg.
Memphis, TN 38152-3370
Office: 901.678.2533
Fax: 901.678.2199

6. **Anticipated benefits from this study:**
   a) The potential benefits to science and humankind that may result from this study are advancements in knowledge on software engineering. In particular, the study will contribute to the body of evidence on the efficacy and applicability of pair programming for software engineering.
   b) The potential benefit to you from this study is gaining experience with a popular software engineering technique, pair programming, thus furthering your knowledge of the field of software engineering.

7. **Alternative treatments available:**
   No alternative treatments available.

8. **Compensation for participation:**
   1.5% will be added to your final percentage grade for the class. Example: 86% + 1.5% = 87.5%. The extra credit will be awarded upon completion of the given task, questionnaire, and interview. For grading purposes, the investigator will provide a list of participants' names to the course instructor. Information about your performance in the study will be kept confidential from the instructor; however, the investigator may share aggregate and/or anonymized study data with instructor.
   Instead of participating in this study, you may alternatively earn the 1.5 percentage points of extra credit by writing a short essay. The essay should take an in-depth look at one of the methods/techniques covered in the course. It should describe the strengths/weaknesses of the technique, define the scope of applicability, and thoroughly back-up your position from the literature (books or research papers). For sources, you should look to recent (within last 10 years) conferences in software engineering (e.g., ICSE, ASE, and FSE) or publication by the ACM or IEEE. In terms of format: The paper should be no less than 2 pages in the IEEE conference-proceedings format (10-point, Times Roman font, two columns), and you must cite at least 3 references.

9. **Circumstances under which the Principal Investigator may withdraw you from study participation:**
   No foreseeable circumstances in which the investigator will withdraw you.

10. **What happens if you choose to withdraw from study participation:**
    Upon your notice of withdrawal, the session will end, and you will leave. The investigators will retain any data collected. However, upon your request, the data will be destroyed.

11. **Contact Information.** If you should have any questions about this research study or possible injury, please feel free to contact **Danielle L. Jones** at **901-356-4490** or my Faculty Advisor**, Dr. Scott Fleming** at **901-678-3142,** questions regarding the research subjects' rights, the Chair of the Institutional Review Board for the Protection of Human Subjects should be contacted at 678-2533.

12. **Confidentiality.** All efforts, within the limits allowed by law, will be made to keep the personal information in your research record private but total privacy cannot be promised. Your information may be shared with U of M or the government, such as the University of Memphis University Institutional Review Board, Federal Government Office for Human Research Protections, *if* you or someone else is in danger or if we are required to do so by law.

THE UNIVERSITY OF MEMPHIS
Dreamers. Thinkers. Doers.

Institutional Review Board

315 Administration Bldg.
Memphis, TN 38152-3370
Office: 901.678.2533
Fax: 901.678.2199

13. STATEMENT BY PERSON AGREEING TO PARTICIPATE IN THIS STUDY

☐ **I have read this informed consent document and the material contained in it has been explained to me verbally. I understand each part of the document, all my questions have been answered, and I freely and voluntarily choose to participate in this study.**

_____ _____
Date Signature of patient/Research Participant

_____
Printed Name of Patient/Research Participant

Consent obtained by:

_____ _____
Date Signature

_____
Printed Name and Title

A.3 Study Session Procedure

On the following pages, we include the procedure used for the study sessions.

Materials

- Workstation with web browsers (e.g. FireFox, IE, Chrome), Eclipse, video camera (e.g. video from HP HD-3100 widescreen webcam and sound from Logitech c200 webcam with built in microphone), video capture software (e.g Camtasia Studio 7), and miscellaneous software (Adobe Reader, Adobe Flash, Java SDK).
- User account for each pair with clean installs of Chrome, Eclipse (including documentation), and task project files
- 24" flat screen monitor
- 2 sets of pen/pencil and paper
- 2 copies of pre- and post-questionnaire

Setup

1. Prepare browsers: install Chrome, bookmark Java IDE (http://docs.oracle.com/javase/7/docs/api/) and jEdit homepage (http://jedit.org/) in all browsers
2. Prepare Eclipse: Create workspace on desktop. Import a clean copy of practice and main project from shared directory. Import both tasks into Eclipse. Compile and run.
3. Create a backup copy of initial project on local machine so participants can get back to the original source
4. Prepare Camtasia: start Camtasia Recorder; adjust screen capture area and microphone volume, set auto-save for every 10min
5. If necessary, adjust screen resolution for video recording
6. Check that webcam can capture both participants
7. Check that microphone selected is Logitech c200
8. Turn off screensaver and power saving if possible

Pair Programming Session

1. Ask participants to turn off all electronics
2. Collect signed consent forms and pre-questionnaire
3. Start video recording
4. Read script (given next)

Script

<BEGIN: Establish context. The following information is given in the consent form, and otherwise likely to have been told to the participants already. Read as much or as little as you deem appropriate given the circumstances.>

As a participant in this study, you will be performing a programming task in a Java program using the pair programming technique. As you work, I will ask you both the be as verbal as

possible and use the paper/pen provided as needed. Please speak only in English.

Throughout this study, I will be recording what you say and do with the webcam. Please do not touch this instrument. Additionally, I will be capturing video of the computer screen as you work, and the computer will log everything you do with the interface.

<END: Establish context.>

I will start by familiarizing you with the procedure of pair programming. In particular, I am going to ask you to solve a practice problem using with you each taking turns being the driver and the navigator.

I will assign the roles in this practice session and initiate the switching. There are some guidelines you should be considering when pair programming such as:

Guidelines to Pair Programming

- Play your role:
    - Driver controls keyboard and records design decisions
    - Navigator continually analyzes design and reviews code
- Switch roles periodically
- Both partners own everything
- Keep each other on-task
- Be open to your partner's ideas

For 5 minutes, you will pair programming on the practice task. The task description has been provided for you.
<Assign roles and wait until have read task assignment sheet>
<Wait 5 minutes>

Stop. Are you comfortable following the guidelines? Do you have any questions or comments? Good. Moving on.

For another 5 minutes, you will switch roles and continue working on the same task.
<Assign roles>
<Wait 5 minutes>

Stop. Are you comfortable following the guidelines? Do you have any questions or comments? Good.
<STOP VIDEO RECORDING **F10** AND SAVE>
<RESTART VIDEO RECORDING>
Your task will be to fix a Java application called jEdit.

jEdit is a mature programmer's text editor with hundreds (counting the time developing plugins) of person-years of development behind it.

Two applications are minimized at the bottom of the Desktop. <*Point to apps in the taskbar.*>

One of the apps is a FireFox browser with two open tabs. The first tab is open to the jEdit homepage; and the second tab is open the Java™ Platform, Standard Edition 7 API Specification. These pages have been also been bookmarked in all the browsers in case you need to get back to them.

The other app is Eclipse, which is open to the jEdit project. <*Point to project*>

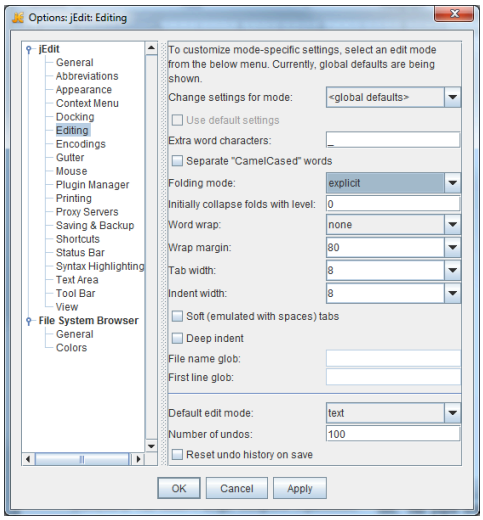
There is also pen and paper here for you to use if you want <*Point.*>

Now, your task is to pair program in attending to fix the bug to be described without introducing any new bugs.

I will step you through replicating the bug. In jEdit, it is possible to "fold" arbitrary blocks text.

To fold blocks of text

1. Enter the global options pane of jEdit
2. Select the "explicit" option for folding mode under “Editing." This is located under the Editing sub-item in the options.



To replicate the bug

1. Select <any block of text needs to be surrounded by "{{{" and "}}}" and nothing outside of the brackets on same line>
2. Select the "Fold lines" command from the "Folding" menu option.
3. Select the "Edit" menu option, then "Text", then "Delete lines"
4. Type in the area that the text used to be.

Typing in the area where the text used to be caused the deleted text to reappear, the cursor disappears and eventually leads to jEdit throwing an exception.

If you complete the task, let me know. You will have 1:50 to work.
Do you have any questions? <*Respond to questions*>

Please begin pair programming now.

<*After 0:50 has elapsed*>
<STOP VIDEO RECORDING **F10** AND SAVE>
Stop. I will save the first part of the session video now.
<RESTART VIDEO RECORDING>
Please restart pair programming.

<*After 1:00 has elapsed*>
Stop. Thank you.
<STOP VIDEO RECORDING **F10** AND SAVE>
<RESTART VIDEO RECORDING>

5. Collect post-questionnaire
6. Do exit interview (or give paper)

<STOP VIDEO RECORDING **F10** AND SAVE>

7. Move saved videos to <Pair Session #> folder on server

A.4 Participant-Recruitment Email

On the following pages, we include the participant-recruitment email for the study.

Hi Folks,

As I discussed in class yesterday, there is currently a special opportunity for you earn _3_ Above and Beyond Points.

You may choose one of two possible options (details below): (1) participate in a study OR (2) write an essay.

INSTRUCTIONS:

If you wish to take advantage of this opportunity, email me your choice (Option 1 or Option 2).

Additionally, if you wish to participate in the study, fill out the attached XLSX file, and email it to Danielle Jones <dljones@memphis.edu>. For the scheduling portion of the form, please provide as much availability as possible, because coordinating participant schedules may be a challenge.

OPTION #1: Participate in Research Study

Wanted: Java programmers to participate in a study of the practice of pair programming.

Study procedure: As a participant, you will take part in a study session that may last up to 2.5 hours. Your main task during the session will be to work with another participant on debugging a Java program using pair programming.

Scheduling: We will schedule sessions based on your availability.

OPTION #2: Essay

The essay should take an in-depth look at one of the methods/techniques covered in the course. It should describe the strengths/weaknesses of the technique, define the scope of applicability, and thoroughly back-up your position from the literature (books or research papers). For sources, you should look to recent (within last 10 years) conferences in software engineering (e.g., ICSE, ASE, and FSE) or publication by the ACM or IEEE. In terms of format: The paper should be no less than 2 pages in the IEEE conference-proceedings format (10-point, Times Roman font, two columns), and you must cite at least 3 references.

Let me know if you have any questions/concerns.

A.5 Background Questionnaire

On the following pages, we include the background questionnaire that participants filled out.

Observation Study of Pair Programming
Pre-Session Questionnaire

Demographics (check ☐ only one)

Sex:

- ☐ Male
- ☐ Female
- ☐ Decline to disclose

Age range:

- ☐ 18–19
- ☐ 20s
- ☐ 30s
- ☐ 40s
- ☐ 50 or over
- ☐ Decline to disclose

Current Field of Study / Major:

- ☐ Computer Science
- ☐ Computer Engineering
- ☐ Electric Engineering
- ☐ Mechanical Engineering
- ☐ Mathematics
- ☐ Decline to disclose
- ☐ Other:

  _____

Education (highest degree or level completed):

- ☐ High school graduate - high school diploma or the equivalent (for example: GED)
- ☐ Some college credit, but less than 1 year
- ☐ 1 or more years of college, no degree
- ☐ Associate degree (for example: AA, AS)
- ☐ Bachelor's degree (for example: BA, AB, BS)
- ☐ Master's degree (for example: MA, MS, MEng, MEd, MSW, MBA)
- ☐ Professional degree (for example: MD, DDS, DVM, LLB, JD)
- ☐ Doctorate degree (for example: PhD, EdD)
- ☐ Decline to disclose

Is English your primary language?

- ☐ Yes
- ☐ No, indicate your primary language :

  _____

Experience

Years of….	
Programming experience	
Professional programming experience	
Using Java	
Working in Java in professional capacity	
Using IDE tools (NetBeans, Eclispe, BlueJay)	
Working with IDE tool in professional capacity	

Have you ever worked on subject software before?

- ☐ No
- ☐ Yes, describe context and nature of the work.

_____

_____

_____

_____

Have you ever pair programmed?

- ☐ No
- ☐ Yes, how many sessions? Explain the context of the work (e.g. school, work, etc.):

_____

_____

_____

_____

A.6 Post-Questionnaire

On the following pages, we include the post-questionnaire that participants filled out at the end of their sessions.

Observation Study of Pair Programming
Post-Session Questionnaire

In your opinion…

	Very High	High	Neutral	Low	Very Low
Confidence in solution	☐	☐	☐	☐	☐
Enjoyment of technique	☐	☐	☐	☐	☐
Impressions…	**Very Satisfied**	**Satisfied**	**Neutral**	**Dissatisfied**	**Very Dissatisfied**
… of interaction with partner	☐	☐	☐	☐	☐
Pair Programming worked…	**Very Well**	**Well**	**Neutral**	**Poorly**	**Very Poorly**
… for me	☐	☐	☐	☐	☐
… for my partner	☐	☐	☐	☐	☐
These are significant benefits in pair programming.	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
Fewer bugs	☐	☐	☐	☐	☐
Spreads code knowledge among	☐	☐	☐	☐	☐
Higher quality code	☐	☐	☐	☐	☐
Can learn from partner	☐	☐	☐	☐	☐
Better design	☐	☐	☐	☐	☐
Constant code review	☐	☐	☐	☐	☐
Two heads are better than one	☐	☐	☐	☐	☐
Improved creativity and brainstorming	☐	☐	☐	☐	☐
Better testing and debugging	☐	☐	☐	☐	☐
Improved morale	☐	☐	☐	☐	☐
These are significant problems in pair programming.	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
Inefficient use of time	☐	☐	☐	☐	☐
Personality clashes	☐	☐	☐	☐	☐
Disagreements	☐	☐	☐	☐	☐
Differences in skill level causes issues	☐	☐	☐	☐	☐
Differences in preferred programming style causes issues	☐	☐	☐	☐	☐
Differences in personal style causes issues	☐	☐	☐	☐	☐
One partner's interruptions distract both partners	☐	☐	☐	☐	☐
Communication issues between partners	☐	☐	☐	☐	☐

Wrap-up

I would use pair programming in a professional setting.

- ☐ Highly Likely
- ☐ Likely
- ☐ Neutral
- ☐ Unlikely
- ☐ Highly Unlikely

Did you ever become impatient/have problems with some/all of pair programming guidelines? Explain.

When navigator, did you guide and identify tactical and strategic deficiencies? Explain.

When navigator, were you thinking at the same level of abstraction as the driver? Explain.

When navigator, did you feel like you were making a meaningful contribution to the work? Explain.

Please give feedback, comments, or any reflections about your pair programming experience.

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

A.7 Pair Programming Guidelines

On the following pages, we include the pair programming guidelines participants were asked to follow.

Guidelines to Pair Programming

- Play your role:
  - Driver controls keyboard and records design decisions
  - Navigator continually analyzes design and reviews code
- Switch roles periodically
- Both partners own everything
- Keep each other on-task
- Be open to your partner’s ideas