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

Danielle L. Jones, Scott D. Fleming
Department of Computer Science
University of Memphis
Memphis, Tennessee 38152-3240
Email: {dljones9,Scott.Fleming}@memphis.edu

***Abstract*—Numerous studies have pointed to the considerable potential of pair programming, for example, for improving software quality. Using the technique, two programmers work together on a single computer, and take turns playing the role of driver, actively typing and controlling the mouse, and the role of navigator, attentively monitoring the driver's 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 qualitative study of seven pairs (14 senior undergraduate and graduate students) engaged in a debugging task. The study addressed open questions regarding partner teaching within pair programming, navigator contributions to tasks, and the impact of partner interruptions. Key findings included (1) that all pairs exhibited episodes of teaching, often covering practical development knowledge, such as how to use programming-tool features, (2) that navigators contributed numerous ideas to the task and the pairs acted upon the vast majority of those ideas without discussion, and (3) that pairs exhibited almost no indications that partner interruptions disrupted their flow.**

I. Introduction

Over 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 [30]. 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. Recent studies have found that pair programming can improve software quality [11], [15], [21], [29], [28], that pairs complete tasks faster [1], [8], [21], [29], [28], and 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) [15], [26], [29], [28].

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. However, many practitioners have expressed doubts about whether the practice is in fact more efficient than programming individually [3]. Moreover, some studies have contradicted the findings of benefit. For instance, one study indicated that pair programming has no positive effect on development time [20], and another found that pair productivity varied over projects [21].

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" [7]. 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 conducted a qualitative study of senior undergraduate and graduate students working in pairs on a debugging task. These pairs were working together for the first time. Williams et al. [29] argue that two programmers pairing for the first time go through an initial adjustment period, and eventually become *jelled*. Jelled pairs are significantly more productive that *pre-jelled* pairs (i.e., pairs who have not yet adjusted to working together). However, the pre-jelled state may also be important, for example, because each partner is learning about how the other works for the first time. Thus, the goal of our study was to refine our understanding of how pre-jelled pairs behave during pair programming and to reveal promising directions for future research.

To focus our investigation, we used the following research questions. Salinger et al. [25] recommend the use of such focusing perspectives to help save qualitative researchers from "drowning" in rich qualitative data.

- 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?

Regarding RQ1, pair programming has been touted as benefitting learning and spreading knowledge [8], [28]; however, no prior studies have looked at how pairs teach one another or at the types of knowledge they exchange. Students in one study reported having learned from their partners, but the study did not observe the learning firsthand [28]. 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 [24], 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 [5], [7]. Moreover, some programmers have indicated feeling more engaged in the task when they have control of the keyboard and mouse [7], and one study found that it was not uncommon for navigators to disengage from the task periodically [23]. 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 immersion in an activity [10]; 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 us: "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." Indeed, an in-depth study of two pair programmers found that pair communication was frequent, and that the navigator tended to dictate what the driver should do [31]. However, Belshee [4] argues the existence of *pair flow* in which partners jointly achieve a flow state. Thus, our study investigated the extent to which partner interactions disrupt concentration.

The remainder of the paper is organized as follows. Section II provides background on prior work regarding the benefits and moderating factors of pair programming. Section III describes our study method. Sections IV–VI report the results for each of our research questions along with discussion. Section VII concludes with discussion of future directions.

II. Background: Pair Programming

A. Potential Benefits

The literature suggests several key benefits of pair programming.

1) Product Quality: Studies have suggested that pairs produce higher quality software than solos. For example, a series of experiments, involving hundreds of undergraduate college students compared the work of student pairs versus solo programmers. Student pairs' projects passed significantly more automated tests than did solos' [29], and student pairs scored significantly higher on their projects than did solos [15]. A meta-analysis (by Dybå et al.) of four studies of professionals and 11 studies of students found general agreement among the studies that pair programming improves software quality over solo programming, with small to medium effect sizes [11]. But not all results regarding professionals have agreed: One study of 295 professional programmers found only a 7-percent (insignificant) improvement in the correctness of pair-produced programs over those of solos [1]. 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 [22].

2) Task Efficiency: Studies have also suggested that pairs produce this higher quality work more efficiently than 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. Moreover, the Dybå meta-analysis found that pair programming had a medium-sized overall reduction of the time to complete tasks, compared with individual programming [11].

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 [21], a result echoed by a subsequent study of 554 undergraduate college students [16]. 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 [2].

4) Knowledge Transfer: A final potential benefit of pair programming that the literature suggests is knowledge transfer between programmers. For example, Cockburn and Williams offer this characterization: "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" [8]. There is some empirical support for this characterization: In one 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 [28]. 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 [7]. Another study in which 18 professional developers were interviewed found that although the developers believed that peer interaction, such as pair programming, is an effective way to discover new tools, such discovery happens infrequently [18]. However, none of these studies analyzed the types of knowledge being taught. In contrast, a qualitative study of three "side-by-side" programmers (students) observed instances of the programmers exchanging project-related and general knowledge [24]; however, the study did not investigate pairs engaged in the standard pair programming technique.

B. 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.

1) Pair Jelling: The literature contains some evidence that pairs may go through an adjustment period when they first work together [29]. After the pair has adjusted, or *jelled*, they perform tasks considerably more efficiently than before. We are unaware of any prior work that has specifically studied pre-jelled pairs, and we intentionally focused on new partners to better understand this essential part of the pairing process.

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 [14]. 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 [26]. In contrast, combinations of partner expertise have been indicators of pair performance. For example, several studies have offered evidence that individuals with similar expertise levels tend to make more successful pairs (e.g., [1], [27]). Moreover, one such study found that lower expertise pairs were generally as successful as higher expertise pairs on high complexity tasks [1].

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 [23].

4) Flow: The concept of *flow* has long been held as important to successful development [10]. 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 [4]. However, we could find no empirically grounded characterization of pair flow in the literature. Thus, it is an open question the extent to which partners interrupt each other's flow, and we address this question with our RQ3.

III. Method

A. Participants

Participants in our study comprised 14 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 10 were graduate students enrolled in a graduate-level software engineering course. Two of the 14 participants had pair programmed before in their undergraduate courses. Table I lists the participants' background information.

TABLE I. 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

B. Task and Environment

The primary pair programming task consisted of finding and fixing a bug in jEdit, a Java-based open source text editor. The defect came from an actual bug report (#2548764) and involved a problem with jEdit's text "folding" functionality. The jEdit code base comprised 96,713 source lines of code. None of the participants 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.

C. 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 (materials available at http://www.cs.memphis.edu/~sdf/studies/vlhcc2013/). 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 pairs were given a 15-minute pair-programming tutorial and practice pair-programming exercise. The participants then worked on the main task for 110 minutes. The task was sufficiently challenging that no participants finished in the allotted time.

D. Analysis

To analyze the data, we used qualitative methods from *grounded theory* [13] (esp. the Corbin and Strauss variant [9]). Central to our qualitative analysis was 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 capture the new concepts. 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.

IV. RQ1 Results: Partner Teaching

As Fig. 1 shows, all pairs exhibited teaching episodes (i.e., at least one bar per pair). 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.

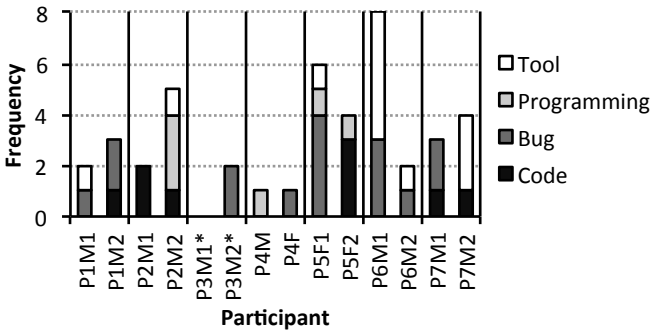


Fig. 1. Frequencies of partner-teaching episodes by topic. Vertical lines separate pairs. * indicates that some data was lost due to poor audio quality.

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.

A. General Development Knowledge

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), labeled as *Tool* and *Programming*, respectively, in Fig. 1.

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]

B. Project-Specific Knowledge

Participants also 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), labeled as *Bug* and *Code*, respectively, in Fig. 1.

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.

C. Discussion

Although pre-jelled pairs have generally been characterized as being less effective than jelled pairs in the literature 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. As shown by the Tool bars in Fig. 1, 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 [12]. Unfortunately, in practice, developers frequently take advantage of only a small subset of the features that IDEs have to offer [17]. 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

TABLE II. 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.

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 I).

V. RQ2 RESULTS: NAVIGATOR CONTRIBUTIONS

A. Preliminaries: Distribution of Navigator Role

Before we address navigator contributions, we first look at how the partners distributed the role of navigator. As Table II 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. 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.

B. Ideas Offered by the Navigator

As the second to last column of Table III 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 III 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. For example, P6M1 (as navigator) proposed the specific action of searching within a class for the word *delete* using Eclipse's Find utility:

TABLE III. 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.

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]

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.

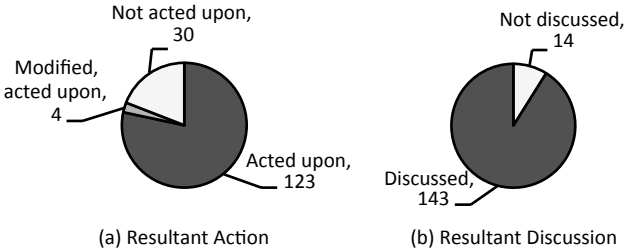
C. Responses to Ideas

As Fig. 2a shows, pairs acted upon the ideas offered by navigators much more often than not. (See also Table IV for results broken down by pair.) 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. 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.



(a) Resultant Action (b) Resultant Discussion

Fig. 2. Frequencies of responses (by type) to navigator ideas over all pairs.

TABLE IV. FREQ. OF RESPONSES TO EACH NAVIGATOR'S IDEAS.

Pair	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

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.
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 Fig. 2b 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.

D. Discussion

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. [7] also observed pairs (professionals) working very closely together—so close 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 about the level of abstraction of navigator discourse. In particular, Bryant et al. [5] 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. [23] 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 6 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. [7] 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.

As shown in Table IV, 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 [27] 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.

VI. RQ3 Results: Partner Disruptions of Flow

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 clear why jEdit 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.

A. 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* [4]; however, Belshee provided neither a detailed characterization of pair flow, nor empirical support for its existence. In Section V, 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 subactivity 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 (e.g., in design tasks).

It is also possible that partners did not interrupt each other's flow because they tended not to enter flow state. Nakamura and Csikszentmihalyi [19] 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 (Section III) 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.

VII. Conclusion and Future Work

In conclusion, our qualitative study has shed new light on several aspects of pair programming: partner teaching, navigator contributions to the task, and partner interruptions of flow. 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 about tools and programming, knowledge in which many developers have been found to have gaps [17].
- The most common knowledge taught was project-specific knowledge about 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.

These findings point to several promising avenues for future research. Our partner-teaching findings point to the idea that "promiscuous pairing" (pairing with many partners) may increase developer expertise and productivity. Furthermore, a few "trysts" with each partner may be sufficient to get the main benefit. Others in the literature (e.g., [6]) 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 navigator-contribution 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.

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.

Finally, our study emphasized debugging, and it is an open question to what extent our findings generalize to other types of programming tasks. 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] 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.

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

[3] 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.

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

[5] S. Bryant, P. Romero, and B. du Boulay, "Pair programming and the mysterious role of the navigator," *Int. J. Hum.-Comput. Stud.*, vol. 66, no. 7, pp. 519–529, 2008.

[6] J. Chong, "Social behaviors on XP and non-XP teams: A comparative study," in *Proc. Agile Development Conf.* (ADC '05). IEEE, 2005, pp. 39–48.

[7] 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.

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

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

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

[11] 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.

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

[13] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine, 1967.

[14] 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.

[15] 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.

[16] ——, "Pair programming improves student retention, confidence, and program quality," *Commun. ACM*, vol. 49, no. 8, pp. 90–95, 2006.

[17] 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.

[18] E. Murphy-Hill and G. C. Murphy, "Peer interaction effectively, yet infrequently, enables programmers to discover new tools," in *Proc. ACM 2011 Conf. Computer Supported Cooperative Work* (CSCW '11). ACM, 2011, pp. 405–414.

[19] J. Nakamura and M. Csikszentmihalyi, "The concept of flow," in *The Handbook of Positive Psychology*. Oxford Univ. P., 2005, pp. 89–105.

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

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

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

[23] 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.

[24] L. Prechelt, U. Stärk, and S. Salinger, "Types of cooperation episodes in side-by-side programming," in *Proc. 21st Annu. Workshop Psychology of Programming Interest Group* (PPIG '09), 2009.

[25] S. Salinger, L. Plonka, and L. Prechelt, "A coding scheme development methodology using grounded theory for qualitative analysis of pair programming," in *Proc. 19th Annual Workshop of the Psychology of Programming Interest Group* (PPIG '07), 2007, pp. 144–157.

[26] 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.

[27] L. Williams and R. Kessler, *Pair Programming Illuminated*. Addison, 2003.

[28] ——, "The effects of "pair-pressure" and "pair-learning" on software engineering education," in *Proc. 13th Conf. Software Eng. Education and Training* (CSEET '00). IEEE, 2000, pp. 59–65.

[29] 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.

[30] 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.

[31] M. Zarb, J. Hughes, and J. Richards, "Analysing communication trends in pair programming videos using grounded theory," in *Proc. 26th BCS Conf. Human Computer Interaction* (HCI '12), 2012.