The Social Dynamics of Pair Programming

Jan Chong
Center for Work, Technology and Organization
Management Science and Engineering
Stanford University
jchong@cs.stanford.edu

Tom Hurlbutt
Stanford University HCI Group
Computer Science Department
Stanford University
hurlbutt@cs.stanford.edu

Abstract

This paper presents data from a four month ethnographic study of professional pair programmers from two software development teams. Contrary to the current conception of pair programmers, the pairs in this study did not hew to the separate roles of "driver" and "navigator". Instead, the observed programmers moved together through different phases of the task, considering and discussing issues at the same strategic "range" or level of abstraction and in largely the same role. This form of interaction was reinforced by frequent switches in keyboard control during pairing and the use of dual keyboards. The distribution of expertise among the members of a pair had a strong influence on the tenor of pair programming interaction. Keyboard control had a consistent secondary effect on decision-making within the pair. These findings have implications for software development managers and practitioners as well as for the design of software development tools.

1. Introduction

The practice of pair programming has begun to attract academic attention in recent years [1-5], as more and more commercial companies consider its use. Pair programming is perhaps the most unconventional practice promoted by eXtreme Programming (XP), one of the many agile programming methodologies that have recently become popular. With the complexity and size of modern software projects, most professional programmers do not work alone, but rather on a software development team. With the increasing need to coordinate work, programming work has had more and more of a social component. Programmers commonly turn to team members for technical knowledge, advice and programming help. Pair programming raises the level of collaboration by assigning joint responsibility for code design and implementation to a pair of programmers, who are then expected to work physically side-by-side on a shared machine.

The most common depiction of pair programming dynamics utilizes a driving metaphor to describe the division of labor in a programming pair. The two distinct roles are referred to as the "driver" and the "navigator." The driver controls the keyboard and is thought of as primarily being concerned with implementation, while the navigator thinks "strategically", evaluating implementation decisions and looking for logical pitfalls. This depiction is pervasive; programmers commonly draw upon it to describe their behavior when asked to explain the practice. But while these roles have been widely accepted in both the practitioner and academic literature, they have never been seriously questioned. This study presents a series of pair programming interactions drawn from a long term ethnographic study of two software development teams. These interactions suggest that the characterization of pair programmer roles as "driver" and "navigator" may not be accurate. As a result, the ways in which we currently train pair programmers may actually run counter to the ways that pairs work most naturally and effectively.

This mismatch between the dominant conceptualization of pair programming interactions and the observed interactions between professional pair programmers working *in situ* suggest that our understanding of pair programming as a practice is, at best, nascent. There is great enthusiasm for pair programming as a software development practice; a more thorough understanding of the dynamics that drive pair programming efficacy will allow us to better determine how to train, manage and support pair programmers.

2. Related Work

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

We begin with a brief review of pair programming characterizations in both the academic and practitioner literature. While pair programming as a concept has been traced back to the 1950s [6], the practice is perhaps most widely known in the context of XP. Beck's widely cited and widely read book [7], generally considered to be the first authoritative work on the principles and implementation of XP, includes pair programming as one of the methodology's twelve practices. In his discussion of pair programming, Beck writes:

> There are two roles in each pair. One partner, the one with the keyboard and the mouse, is thinking about the best way to implement this method right here. The other partner is thinking more strategically:
> - Is this whole approach going to work?
> - What are some other test cases that might not work yet?
> - Is there some way to simplify the whole system so the current problem just disappears?

While Beck's explanation of pair programming roles never uses the term "driver" or "navigator", it does describe two programmers thinking at distinctly different levels of abstraction. The programmer in control of the keyboard is assumed to be primarily concerned with the details of implementation; the other partner is assumed to consider broader, more strategic issues. Beck's definition of pair programming and its implication of the differential in levels of abstraction between the two programmers is echoed in many of the pair programming descriptions written by XP devotees. A blurb from adaptionsoft.com, for example, compares "a worm's eye view" (the driver) with a "bird's eye view" (the navigator) [8] .

Williams and Kessler [6] provide what is perhaps the most widely cited definition of pair programming roles. Here, they use the terms "driver" and "navigator" in their definition. According to Williams and Kessler, the driver is the programmer "typing at the computer or writing down a design", while the navigator "has many jobs, one of which is to observe the work of the driver, looking for tactical and strategic defects." They then go on to describe the navigator as a "strategic long-range thinker". This definition broadens somewhat the scope of the "navigator" role.

Few studies have focused specifically on the nature of the interactions between the programmers in a pair and, in general, the idea that pairs adopt these driver and navigator roles has gone unquestioned. Three exceptions are Chaparro *et al.* [9], Bryant [10] and Bryant [11]. Chaparro *et al.* noted that driver and navigator roles were difficult to identify in student pairs, but they speculated that perhaps professional programmers adhered more closely to the "driver" and "navigator" roles as compared to students who had only recently been introduced to the concept of pair programming. Bryant [10] analyzed patterns of pair activities and found that student pairs, rather than operating in "driver" or "navigator" roles, switched between "driver" activities and "navigator" activities somewhat erratically. When applied to professional programmers, Bryant found that pairs had the same behavioral profile regardless of which programmer was "driving" and which was "navigating". Methodological limitations, however, prevented her from conclusively determining whether this simply reflected seamless transitions by professional programmers between the two roles (i.e., drivers always act like drivers and navigators always act like navigators) or whether programmer behavior was simply role independent. In her later work on the expertise perception [11], Bryant is largely skeptical of the level of abstraction differential implied by the driver-navigator characterization, at one point questioning "how it would even be possible for two people working at different levels of abstraction to successfully sustain a conversation at all."

In general, few researchers have studied the dynamics of pair interaction. Williams and Kessler [6] discuss the potential effects of expertise and personality type on pair interaction, but provide primarily anecdotal support. They argue that cross-pairing programmers of different levels of expertise could produce opportunities for learning and potentially improve code through the questioning of basic assumptions, but that these mismatches also had the potential to impede pair function. Experts paired with novices may grow tired of constantly having to "train" their partner; novices may not have sufficient knowledge or experience to give valuable input. Similarly, they note that novices paired with novices can be ineffective if neither programmer is sufficiently knowledgeable to contribute effectively to the process. They also raise personality as potentially confounding factor, noting that introverts may have difficulty fully contributing to the exchange and evaluation of ideas that lies at the heart of pair programming efficacy.

Empirical support for these arguments has been mixed. VanDeGrift [12] surveyed students enrolled in introductory programming courses and reported that the second largest complaint about pair programming was being forced to work with partners of different skill levels. Katira *et al*. [13] attempted to assess the effects of personality and skill on the compatibility of student pairs, but had mixed results. Sfetsos *et al*. [14] found that pair productivity was correlated to communication volume in mixed personality pairs (mixed in terms of Keirsey temperament), but that no such correlation was present for non-mixed pairs. Padberg and

IEEE COMPUTER SOCIETY

Muller [15] found a correlation between the comfort level of programmers within pairs (what they call the "feelgood" factor) and overall pair performance, but did not investigate the specific causes of pair comfort. We are not aware of any empirical studies of the effect of expertise.

This study seeks to examine pair interactions between professional programmers in a natural work environment; that is to say, in the context of a large team-based software project. We, in particular, seek to understand what factors may influence pair programming interactions and how they might do so.

3. Research Site

The results presented in this paper are based on a four month ethnographic study of pair programmers on two software development teams. The two teams were located in two different startup companies in the San Francisco Bear Area. Both teams were formed with eXtreme programming in mind as the development methodology of choice and therefore had a long history of pair programming.

Team A was formed in January of 2004. The team initially had six developers, but hired three additional programmers by the end of the observation period. With the exception of the new hires, the developers on the team were very familiar with the code base; four of the team members had been with the team since its inception. Pair assignments were negotiated on a daily basis, in a fairly ad hoc manner, although the team took pains to ensure diversity in pair partners over the course of a week of work. It was fairly common for neither member of the pair to be overly familiar with their assigned task for the day. The developers on Team A worked inside a large open bullpen, equipped with a set of shared workstations. Each station had dual flat screen monitors, a single machine, dual keyboards and dual mice.

Team B was formed in early 2002 and had been steadily growing in size since formation. When observations began, the team had nine programmers; the team hired an additional full-time programmer during the course of the observation period. Because the project code base for Team B was both older and more extensive than the code base for Team A, a thorough knowledge of the code's basic design and structure was less pervasive among the developers on Team B; four of the older team members were generally considered to be the most knowledgeable in this respect. Pairs on Team B were formed in an ad hoc manner each morning. Like Team A, members of Team B usually were not too knowledgeable about their tasks. However, due to the practice of "spiking", programmers sometimes had more knowledge with respect to a task than his or her partner. For difficult and complex tasks whose scope and cost were not apparent, the team sometimes assigned programmers to "spike" the task, or write up some rough, exploratory code, to gauge the complexity of the task. The programmers who spiked a particular task would not necessarily be subsequently assigned to implement the task, but in cases where one member of the pair had participated in the spike and the other had not, the spiking programmer would have substantially more knowledge of the task.

Programmers on Team B had personal desks inside a large open space. Each desk was equipped with a personal machine, monitor, a single keyboard and mouse. Equipment was non-standard across the team and the programmers frequently augmented their systems with additional equipment (most commonly monitors). When a pair formed at the beginning of the workday, the pair would negotiate for the role of the "driver." After designating this role, they then worked at the driver's desk for the duration of the task.

4. Methodology

We conducted ethnographic observations at both sites. We visited Team A from June 2005 to September 2005 and we visited Team B from May 2005 to August 2005. During the observation periods, we visited the teams on a weekly basis. In each observation session, we followed one pair of programmers for one and a half to three hours. We sat behind them as they worked, taking notes on their interactions and activities. Whenever possible, we recorded dialogue, which we transcribed and integrated with our notes to produce a detailed record of each session. We then reviewed our records to identify consistent and repeated patterns of behavior. We developed a coding scheme to help us categorize programmer behaviors, which we applied to all the observation data from Team A (to a statement level) and a selection of the data from Team B. All told, we had approximately forty hours of pair programming behavior to draw from in our analysis.

4. Findings

Observed pair behavior on both software development teams differed greatly from the driver and navigator roles described in the academic and practitioner literature. When the two programmers had equivalent expertise, they engaged jointly in programmer activities. When the distribution of task-relevant expertise differed, the programmer with more expertise dominated the interaction. We also found that keyboard control had a subtle, but consistent effect on decision-making. We discuss each of these findings in turn.

IEEE COMPUTER SOCIETY

5.1. The Driver/Navigator Myth

Aside from the task of typing, we found no consistent division of labor between the "driver" and the "navigator". Instead, the two programmers moved from task to task together, considering and discussing issues at the same strategic "range" or level of abstraction. In pairs where the level of expertise was roughly equal, the two programmers contributed to the discussion at roughly equal rates. The thought processes of the two programmers appeared to be tightly coupled, blending into a cohesive stream of discourse.

5.1.1. Pair Programmer Interaction. To illustrate typical pair interaction, we present an excerpt from a pair programming session from Team A. For reasons of space and conciseness, all of the excerpts used in this paper have been edited. In addition, all of the names have been replaced with pseudonyms. Here, the two programmers, Anthony and Ben, are implementing a new feature. In the portions of the session shown below, Anthony has control of the keyboard.

Anthony: Um, and we always expect an operation? [*He types*]
Ben: And there's always the action set.
Anthony: [*as if he had forgotten until Ben mentioned it*] Yes.
Ben: We'll just run it to see what comes out. Can't waste your brain cycles on these things. Okay, add one to the substring.
Anthony: To the substring. Well, if it's zero, you do want the zero case.
Ben: Oh, maybe we don't need the max thing, just add plus one to the substring and it always works?
Anthony: Oh, sneaky. [*He deletes the old code and implements this version instead*] That was sneaky.
Ben: It's a pure coincidence. Or somebody ten years ago had this case and said oh, minus one would be a good number.
Anthony: Right? So now we have this extraction?
Ben: So now keep the strings in a set.
Anthony: Right.
Ben: And don't go there if you've already been there.

Anthony and Ben's behavior at the beginning of the session looks reasonably consistent with traditional concepts of pair programming roles; Anthony is focused on implementation and Ben is offering suggestions and advice. When Anthony runs into a complication with his current implementation (the "zero case"), Ben is able to supply a cleaner solution. While they are both immersed in the technical detail of the implementation, Ben is perhaps the more tactical of the two. These roles will change as they finish their implementation and move to their next task.

Anthony turns to face Ben.

Anthony: Okay, so we can write the code in the way that we try to visit the place- we have a visit method that takes a link and it maybe does nothing, or do you want to go with if checks before that and doesn't visit?
Ben: Are you saying have a visit or not? Or are you talking about a line that should be in the method?
Anthony: No, we should have- we should have a visit method, you go there and it does all this checking?
Ben: A "go there if we haven't been there before" method. Is that what you mean?
Anthony: Right.
Ben: Yeah, I'm just wondering if visited is the best name. It's always hard, these things that are "do this unless it's in the cache" methods.
Anthony: Yeah. [*Anthony's hands are clasped. He stares straight ahead, thinking.*] So let's see, how to implement. How do we test that we've got actually invoke the, um...
Ben: Well, give it a list with a lot of links and see how many times it visits.
Anthony: We could have a hasVisited method.
Ben: We, uh, should probably have that too. Yeah.

Anthony begins to type. He creates a new method.

In this portion of the excerpt, the pair pauses. Anthony asks Ben a strategic question about the overall direction of their implementation; this causes them to begin discussing implementation options at this new level of abstraction. Note that it is Anthony who steers the discussion to a more strategic level, although he is technically acting as the "driver". Based on the descriptions of driver and navigator roles, we initially expected to find that the "navigator" would be chiefly responsible for sparking discussions from a more strategic or broader perspective, but in nearly all of the interactions between two programmers with equal expertise, these issues were raised by both the "driver" and the "navigator" at approximately equal rates.

In general, the dialogue between pairs was notable for the parity of contribution between the two programmers. Ideas and suggestions came from both parties, and programmers conscientiously solicited each other's input in the course of discussion. We did not see a sustained concern for the details of implementation by one programmer coupled with a more strategic world view from the other; rather the pairs moved together between the various levels of strategic thinking and implementation detail. Aside from the duties of keyboard input, the programmers in these pairs jointly took on the responsibilities of both "driver" and "navigator".

A substantial factor behind this pattern of behavior is that programming itself is not a continuous activity, but rather a sequence of fits and starts. Pairs would engage in short bursts of implementation and then, as Anthony and Ben did in the preceding excerpt, pause for reflection and design when they encountered unanticipated challenges or completed portions of functionality. When intended implementation was clearly un-

IEEE COMPUTER SOCIETY

derstood by both programmers, then the pair's behavior resembled the driver-navigator characterization. But, when the course of action was not quite as clear, for example during such activities as design, code comprehension and debugging, the pairs generally worked jointly, maintaining a steady exchange of ideas and feedback. Because communication within the pair occurred chiefly through conversation, whenever one programmer began to consider the problem from a new conceptual perspective, the other programmer, drawn in by the reciprocal nature of conversation, almost always did so as well.

5.1.2. Shared Context. Working collaboratively on the same task on the same machine meant that the pairs shared a substantial amount of visual and mental context. On occasion, generally after the pair had negotiated and then agreed to a specific course of action, the programmers sometimes slipped into a mode of behavior where they were exceptionally in sync with one another. Anthony and Ben fell into this mode a bit later in the same work session, as they finished up implementation work on a particular function:

Anthony: [*muttering as he types*] Visited... operations.
Ben: We don't need to check that it's there, just dump it in-
Anthony: Right. [*He deletes a line of code.*] I know it's going to fail now, because I don't strip the, um-
Ben: Right. [*Anthony runs the code and test exceptions show up on the screen*.]
Anthony: Good.

Here, Anthony and Ben are so tightly coupled that sentence completion is not required for effective communication. Ben begins to verbalize a train of thought, but Anthony cuts him off, already aware of how the thought ends. Similarly, Anthony never has to specify why he expects the test to fail, because both the reasoning and the expectation are clearly shared. This mode of interaction could not be sustained for very long, but was always recognizable from the incomplete verbal utterances between the two participants.

5.1.3. Keyboard Switching. On Team A, the tight coupling between programmers was reinforced by their tendency to switch keyboard and mouse control frequently. Team A's shared workstations were outfitted with dual keyboards and dual mice. Consequently, both pair programmers had ready physical access to a keyboard and mouse, although only one programmer could effectively use the devices at a time. Pairs on Team A developed a pervasive practice of rapidly switching control of machine input during programming sessions. The following excerpt demonstrates how fluid and how frequent these transitions could be. Casey and Dale will switch control of the keyboard three times within a two and a half minute period. When the episode begins, Dale is typing on the keyboard, while Casey watches:

Dale: Set...? SetConfig? Do we have this one?
Casey: Uh, not set. It's just config. [*Casey turns and puts her fingers on her keyboard.*] And it's as a string, which is important. Actually, what we need to do-
Dale: A list of strings? Not just one?
Casey: We need to do something like FetchAddress. Right? And send it to something like that name with a value in it. Does this take? [*She creates the constant*] Oh shoot-
Dale: If you don't mind, I would say, can I show you something? [H*e takes the mouse*] We can initialize the config here [*He points to a section of code using the mouse*] and add the values in each of the tests to see exactly what we're using.
Casey: That's fair, I would agree with you. But the only thing is it's used in like nine places.
Dale: The- We can move them up... So where are they used?
Casey: Well, so testFromServer, testFromClient both use them… So testFromUser is the only one that doesn't right now.
Dale: [*scrolling down in the file*] Let's keep it here, yeah.

Casey waits for her to finish and then begins to type.

Casey: So what I was going to do is at least make this one eleven something... [*She types*] Yeah. I think these were changed over yesterday and obviously this is a variable, but it's really a string, so... [*Casey turns a string variable into a constant.*] I'll rename it in a second.

By convention, the programmers refrained from typing when their partner was actively typing, but they frequently jumped in during pauses or periods of hesitation. In this excerpt, Dale types initially and then Casey switches in, beginning to type when Dale pauses. For that particular switch, Casey positions her hands on the keyboard well before she actually intends to type. This was not unusual among the programmers on Team A; both members of the pair would often be poised to use the keyboard at any given moment. Later in the excerpt, when Dale wishes to type, he interrupts Casey to ask permission to take control of the keyboard and mouse. Once permission is granted, the transition requires little additional effort; the second partner simply begins typing.

Switches occurred for several reasons. Sometimes it was simply easier for a programmer to execute an action him or herself, say typing in a line of code or locating a particular file, than it was to describe that action to their partner. In the course of completing their tasks, programmers often temporarily deferred control of the keyboard to their partners when they knew that their partner was more practiced in a particular subtask, such as using a particular feature or plug-in of the IDE. In some instances, it simply seemed that both pro-

IEEE COMPUTER SOCIETY

grammers were eager to type, with the non-keyboarded partner switching at pauses or during natural breaks in the task. Keyboard switches also occurred when one programmer in a pair was called away; generally the remaining partner took over keyboard control to continue working.

For the pairs on Team A, frequent shifts served to reinforce the tight coupling between the programmers. Faced with the constant prospect of switching roles with their partner, the programmers maintained a high level of mutual awareness of each other's actions. The effect of switching on the level of mutual knowledge was perhaps most evident towards the end of the observation period when, egged on by a newly hired team member, the developers on Team A experimented with using only one keyboard per pair. Below is a quote from Evan, comparing his experience with the two input configurations:

> *Evan*: When I have the second keyboard I am always thinking about what I want to type and when to jump in - more focused on the story [i.e., the task]. When I was drinking the coffee with the keyboard [*he laughs*], I was like, "Okay! You do the work! I am drinking coffee right now!"

Although Evan is not actively typing at the keyboard in either of the situations he describes, he clearly feels more engaged in the task when he has a keyboard available to him. When the prospect of switching roles is more remote, he maintains a much lower level of awareness regarding his partner's activities. Team A's foray into single keyboard use was brief, but for the few pairs we observed during this period, the non-keyboard controlling programmer appeared more prone to distraction.

Even with a single keyboard, the pairs on Team A attempted to continue their practice of rapid role switching. The increased effort required to physically shift the keyboard across the table to switch, combined with the programmer's inability to have the immediate keyboard access they were accustomed to, was frustrating to the pairs and caused them to quickly tire of the setup. By the end of the observation period, the bulk of the pairs had returned to dual keyboard setups.

Team B's technical setup was not conducive to control-switching and, in fact, very little switching occurred among the pairs on the team. Unlike Team A, Team B did not use shared workstations. Instead, when a pair formed, one member would be designated as the "driver" for the work session. The pair would then work on that programmer's machine located at that programmer's desk. Team B's machines were outfitted with a single keyboard and mouse. This meant that switching required more coordination and an explicit physical relocation of the keyboard. Switching was additionally impeded by custom machine configurations. A quote from Finn, a developer on Team B, is indicative of how different the practices on the two teams were:

> *Finn*: No, we try to rotate around. By rotating it around – so sometimes when I sit with Greg, you know, I can't type – he's an emacs user, I'm a vi user, I just can't put my fingers to the keyboard, I wouldn't know what to do.

On Team B, once a pair session began the programmer with keyboard control generally retained keyboard control for the entire session. The pairs on this team, not accustomed to rapid switching, did not find this unusual, but between the expertise differentials on the team (discussed in the next section) and the lack of control switching between paired programmers, maintaining active engagement in the task appeared more effortful for the programmers on Team B.

5.2 Expertise and Interaction

Due to team and project structure, Team A and Team B had very different distributions of expertise which lead to contrasting patterns of pair interaction across the two teams. When gaps in expertise were sufficiently large, the programmer with more expertise dominated the pair programming interaction. We use the term expertise here to refer to a combination of programmer skill and knowledge.

Team A had a very uniform distribution of expertise. The team was small and relatively young. The majority of the programmers had been on the team since the project began. Although sizeable, the developers considered the code base to be relatively easy to understand; one programmer described the bulk of the code as being "a variation on a theme". Indeed, when new programmers joined the team, they rapidly developed a proficient understanding of the code base. Team A did not appear to have any particular mechanism for training their newly hired developers, instead they simply began pairing with senior developers. During their first few days, the more senior programmer spent the bulk of these pair sessions explaining the basic structure of the code. Within one week, however, the new hires had become sufficiently proficient in their understanding of the code structure such that pairing proceeded normally. Since Team A did not regularly spike functionality prior to implementation, the developers usually had relatively equivalent levels of familiarity with their work tasks. The developers on Team A all had at least eight years of professional programming experience.

Team B had deeply entrenched differentials in expertise across the project; the more senior developers

IEEE COMPUTER SOCIETY

on the project were substantially more familiar with the code base than the newer team members. Team B was both older and larger than Team A. The team had started small, but gradually accumulated programmers over the course of the project's development. While Team B's code base was comparable in size to Team A's, it did not share the same ease of understandability. One of the senior programmers on the team noted that it had become difficult to "communicate the design of our system" to new team members. He felt that a good portion of the code recently added to the project had been written without "a conceptual knowledge of how the system works". In addition, Team B regularly spiked solutions, leading to potential gaps in task familiarity. The developers on Team B ranged from having three to over twenty years of professional experience.

The gaps in expertise between the programmers on Team B clearly influenced pair programming interactions on the team. On Team B, the member of the pair with greater expertise drove the bulk of the programming discussions. When compared to the pair programmers on Team A, the difference in the pattern of discourse was striking. In the following example, we see a senior programmer, Ilya, interact with a newer team member, Hugh. Hugh and Ilya are implementing a new function. Hugh controls the keyboard, but Ilya will direct the majority of the interaction during the session:

Ilya: So new has dollar sign myfield and a percent args. Yeah, we want a percent args too. But instead of percent args…have it be getNewArgs. [*Hugh types in these changes*] Just call it dollar sign myfield and percent args. Get rid of getNewArgs, just call it percent…comma percent args.
Hugh: This is…
Ilya: Actually, it's dollar sign value comma percent args.
Hugh: Put dollar sign class?
Ilya: [*As Hugh continues to type*] Yeah, then percent args. Next line, just do a return dollar sign value arrow super colon colon new and pass it percent args. And we want to do something in between those two lines, of course. Basically, you want a dollar sign my variable outside of the package scope right there.
Hugh: So…
Ilya: Parenthesis. So percent, getNewArgs… [*Hugh types.*] Exactly. So save off those two lines in the new method.
Hugh: Uh…
Ilya: Right…down, down, down, there we go.
Hugh: So we…
Ilya: So, percent getNewArgs equals percent args [*Hugh types this line to terminal.*] Uh, I think that's it.
Hugh: This?
Ilya: Yeah, that's all we want to do. Get rid of the blank line and close the new.

In addition to being more senior, Ilya had also "spiked" the code during the previous week. Consequently, he has a much greater familiarity with both the details of task implementation and the overall project code base than Hugh has. As this excerpt clearly demonstrates, Ilya dominates the interaction, determining how and what to implement while Hugh takes directives (to the keystroke) from him; Hugh primarily asks for minor clarifications. Hugh's level of participation here is actually unusually low (he will, in fact, begin to contribute somewhat more actively later in the session), but the structure of this exchange is consistent with the majority of the pair programming interactions on the team as a whole: the programmer with greater task knowledge or code base familiarity dominated. This occurred regardless of which programmer was at the keyboard. Although not evident from the excerpt shown above, like the pairs on Team A, the pairs on Team B still moved across levels of abstraction together. Unlike Team A, however, the majority of the shifts between levels were initiated by the programmer with greater expertise. When expertise between the programmers was more equal, their interactions had more of the parity that characterized pair interaction on Team A.

On Team A, when the programmers in a pair had a substantial difference in expertise, the developer with greater expertise reviewed the technical material in question with his or her partner until a sufficient amount of shared expertise had been established; they would then proceed to pair normally. Thus, it was rare for gaps in expertise to persist, but it could occur when a task was exceptionally long in duration. For tasks that spanned several days, one programmer was usually designated to see the entire task through, although his or her partner would change on a daily basis. Although we did not witness such persistent gaps, developers on Team A reported difficulty when introduced as the new partner on the second or third day, citing, for instance, time pressure as a barrier to establishing a sufficiently uniform level of expertise. One developer described his behavior in these pairs as largely "passive", noting that he did not want to impede overall implementation progress by forcing his partner to stop and explain the technical background required to thoroughly understand the task. This leads us to believe that in cases of exceptional expertise differentials, pairs on Team A may come to follow a similar pattern of behavior as the pairs observed on Team B did.

5.3. The Effect of Keyboard Control

Across both teams, when differences in levels of expertise were not an issue, control of the machine input had a consistent, albeit subtle, influence on pair

IEEE COMPUTER SOCIETY

interactions: the programmer that controlled machine input had a distinct advantage with respect to decision-making.

Barring issues of expertise, the pair programmers we observed constantly solicited and considered the input of their pair programming partner. However, when determining the ultimate course of action, the programmer controlling the machine input (generally, this meant control of the keyboard) had, in some sense, the final authority in decision making. Their partner could give suggestions, but fundamentally, the developer at the keyboard decided which suggestion to follow. The following excerpt illustrates this effect. Dale and Evan, programmers from Team A, are attempting to debug a function they have just written. As this exchange unfolds, Dale uses the keyboard and mouse, while Evan watches. Evan will propose a course of action (quickly move to the next breakpoint by pressing the F9 button). Dale will not agree:

Evan: Put a break point.
Dale: We have a breakpoint here. [*Dale hits run.*] It should come here. [*He hits run again, advancing to the next breakpoint.*] It does. [*Dale begins to step through the code line-by-line.*]
Evan: No, [press] F9.
Dale: [*mildly*] No, I want to go here.
Evan: But-
Dale: [*After a pause*] We are getting here. Ah, we are, but it's another one. I don't think it's… [*He hits F9 and the debugger stops on that line again.*] That's ours, now we're getting here, and address match list… okay, so that's where it is. This is… a… getPrefix for- we need another one.

Here we see that Dale's control of the keyboard and mouse enables him to essentially ignore Evan's proposed course of action, in spite of Evan's subsequent objection. This exchange is unusually direct, both in the language used by the two programmers and how the disagreement is resolved. When pairs disagreed, the programmers generally attempted to reach a consensus before acting. When the issue disagreed upon was relatively minor in scope or consequence, we often saw the programmer with input control simply implement the course of action they favored. Strong norms of mutual respect and politeness largely kept this behavior restricted to fairly inconsequential decisions, but since pairs were mainly peer-based, compliance with the negotiated decision was fundamentally voluntary. Rhetorically, once the action was completed, it usually became more effort for the other programmer to attempt to undo the action than it was to agree. Consequently, the non-typing member of the pair was, by default, at a slight disadvantage when it came to influencing the pair's ultimate course of action. On Team A, the practice of frequently switching keyboard control served to moderate this disadvantage. For Team B, these effects were largely obscured by the differentials in expertise (as discussion in section 5.2).

6. Discussion and Implications

Although commonly cited in descriptions of pair programmer behavior, even by the programmers themselves, the roles of "driver" and "navigator" as they are commonly defined in the practitioner and academic literature do not match the pair programming interactions observed in professional programmers. Pair programmers did not think on different levels of abstraction while working; instead they moved across these levels of abstraction together, considering and discussing issues at the same strategic "range". While the lack of driver/navigator division of labor was true for both teams, other patterns of pair programming behavior were linked to characteristics of the teams and pairs in which they occurred: the distribution of expertise across programmers on the team and frequency with which pairs alternated in keyboard control. The behaviors observed in this study have several implications for pair programming practice.

Move beyond the "Driver" and the "Navigator". The pair programmers in this study rarely hewed to the roles of "driver" and "navigator", yet these roles are so widely accepted in both the academic and practitioner conceptualization of pair programming that they are built into the tools we construct to support pair programming [16] and the materials through which we teach pair programming [6]. This characterization is so pervasive that even the programmers observed for this study sincerely described their own interactions in these terms, despite consistently deviating from these roles during their own pair work.

Our observations revealed pair programmers engaging in a natural pattern of interaction that, aside from designating primary responsibility for keyboard input, lacked an explicit division of labor. Instead, the pairs appeared to be most effective when both programmers took on driver and navigator responsibilities. This suggests that the driver/navigator characterization may not only be inaccurate, but that training pair programmers to work in these roles may actually inhibit more natural and more effective ways of working.

Help Programmers Stay Focused and Engaged. Pair programming is an intensive process that requires sustained energy and focus from both programmers to be effective. The variation in interactions between the programmers observed for this study demonstrated that the right tools and work practices can help both programmers maintain active involvement in the programming. In this study, programmers felt more en-

IEEE COMPUTER SOCIETY

gaged in their tasks when they either had keyboard control or keyboard control was imminent. The data suggest that equipping pair programmers with dual keyboards to facilitate the rapid switching of keyboard control can be a simple way to foster engagement. Practices that require regular shifts in keyboard responsibility (such as ping pong pairing) should also be helpful in this regard.

In general, the tools we develop to support collocated pair programming and distributed pair programming should take care to support programmer engagement. This problem may be exacerbated for distributed pairs, where programmers may lack the immediate social and physical cues of their partners to help maintain interest and focus. Tools for distributed pair programming therefore should attempt to minimize barriers to transitions in keyboard control and maximize shared visual and mental context.

Consider differentials in programmer knowledge. Expertise emerges as a particularly important factor influencing pair interactions, a finding which affirms both the arguments made by Williams and Kessler [6] and informal reports that individuals dislike pairing with someone with lower expertise [12, 17]. Rhetorically, when the difference in expertise is large, the programmer with less expertise has difficulty assessing the technical arguments put forth by the "expert". The pair programming literature suggests that one possible role for the "novice" is to question assumptions by requesting reviews of the code logic [6], but in a time pressured work environment this does not appear to be realistic. In the pairs observed for this study, the less knowledgeable programmer instead reported a tendency to become "passive", disengaging from the task so as not to impede his or her partner's ability to make timely forward progress on the task. This passivity also reduces any benefits that these programmers might receive from exposure to the production or alteration of unfamiliar areas of the code base. In a professional environment, pair programming may simply not be an effective way to negotiate large differentials in programmer knowledge. Developers should consider carefully the ramifications of expertise when forming pairs.

In our observations, pairing less knowledgeable programmers with more knowledge programmers did seem to be effective when the less knowledgeable programmer was new to the team and code base. Both Team A and Team B hired new programmers during the observation period and these programmers did not, as the regular programmers did, shy away from asking for clarifications and explanations of code they did not understand. Unfortunately, our observations at both teams ended shortly after the programmers were hired, so we were unable to determine how or when this behavior changed. These programmers, at least during their first week with the team, appeared to feel much more latitude in interrupting task progress to request explanations, likely on account of their positions as new hires.

Given the size and state of modern code bases, uneven distribution of expertise among a team's programmers is likely to be common. Team B has begun giving a weekly series of team wide talks on the structure of their code base in an attempt to reduce the disparity in expertise in programmers, but several of the developers on the team feel that, for their particular code base, specialization of expertise is inevitable. In academic environments, the effect of expertise may be less pronounced due to a smaller general disparity in expertise across students in a course. For the gap in expertise to be equivalent, one group of students would have to have spent the better part of two years reviewing the course material relative to their peers.

Avoid pair rotation late in a task. Although both teams felt that short tasks (ideally one day in duration or less) were the ideal, in practice tasks sometimes spanned more than one day. Team A held to a consistent system of rotating pair partners daily regardless of the length of the task. This was not problematic when the tasks where short, but for multi-day tasks, this led to significant differences in the level of task knowledge between the paired programmers, inhibiting the ability of the newer programmers to contribute effectively. While, in general, programming partner rotation appeared to be effective in ensuring increased dispersion of code knowledge across the team, rotating late in the task may break up an effectively functioning pair and introduce a new programmer in a disadvantaged position.

6.2. Future Directions

This study highlights several factors that influence pair programmer interactions, but we have really only begun to explore the dynamics of pair programming.

An inherent limit of naturalistic observation is a relative lack of control over that which one observes. In this study, all the pair programming observed occurred in the context of eXtreme programming. We have tried to delineate the specific practice of pair programming from the overarching methodology, but we cannot definitively exclude the effect of XP on the behavior of these programmers. A study of pair programmers working in a non-XP environment would add greatly to our understanding of how both the factors discussed here and other, heretofore unconsidered factors impact programmer performance.

This study sought to explore how professional pair programmers interact, taking those interactions where both programmers were engaged in the programming

IEEE COMPUTER SOCIETY

task to be the ideal. We found that a substantial knowledge differential between paired programmers interfered with the active exchange of ideas and feedback during programming sessions. For pair programming to be an effective mechanism for knowledge exchange in professional environments, either the programmers must already share a substantial amount of knowledge or the less knowledgeable programmer must feel free to ask questions, even at the expense of working more productively. This suggests that knowledge transfer through pair programming will be more effective at certain times (e.g., when developers first join a project or at lulls in the development timeline) and for certain forms of knowledge (e.g., design patterns, tool features, language features). Knowledge transfer through pair programming bears further study and evaluation, particularly for programmers joining new software development projects.

Finally, this study demonstrates that the professional programming environment differs in important ways from the academic programming environment. Student studies are extremely valuable, but we cannot assume that they will generalize completely to the behavior of programmers in a professional environment.

7. Conclusions

This paper presents a descriptive ethnographic study of professional pair programming behavior on two software development teams. The study finds that pair programmers behave in ways inconsistent with the driver/navigator division of labor that is described in the pair programming literature. We identify expertise and keyboard control as important factors influencing pair programming interactions and make several recommendations for software development practice based on these observations.

9. Acknowledgments

We are grateful to Özgür Eris, Scott Klemmer, Larry Leifer, Robert Plummer, and George Toye for their insightful discussions, to Diane Bailey for her guidance during the formative stages of this work, to Jim Herbsleb for his thoughtful comments and feedback, and to the programmers on both teams for their patience during our observations. This work was partially sponsored by a gift from Microsoft Research.

9. References

[1] C. McDowell, L. Werner, H. Bullock, and J. Fernald, "The effects of pair-programming on performance in an introductory programming course," in *Proc. SIGCSE*, Northern Kentucky, 2002, pp. 38-42.

[2] H. Hulkko and P. Abrahamsson, "A multiple case study on the impact of pair programming on product quality," in *Proc. ICSE*, St. Louis, Missouri, 2005, pp. 495-504.

[3] J. T. Nosek, "The case for collaborative programming," *Communications of the ACM,* vol. 41, pp. 105-108, 1998.

[4] J. Nawrocki and A. Wojciechowski, "Experimental evaluation of pair programming," in *Proc. ESCOM*, London, UK, 2001.

[5] L. Williams, "The Collaborative Software Process," in *Computer Science*. vol. Ph.D. thesis Salt Lake City, UT: University of Utah, 2000.

[6] L. Williams and R. Kessler, *Pair Programming Illuminated*. Boston, MA: Addison-Wesley, 2003.

[7] K. Beck, *Extreme Programming Explained*. Boston, MA: Addison-Wesley, 2000.

[8] adaptionsoft.com, "XP Practices: Pair Programming." vol. 2005, 2005.

[9] E. A. Chaparro, A. Yuksel, P. Romero, and S. Bryant, "Factors affecting the perceived effectiveness of pair programming in higher education," in *Proc. PPIG*, Brighton, UK, 2005, pp. 5-18.

[10] S. Bryant, "Double trouble: Mixing qualitative and quantitative methods in the study of eXtreme programmers," in *Proc. VL/HCC*, Rome, Italy, 2004, pp. 55-61.

[11] S. Bryant, "Rating expertise in collaborative software development," in *Proc. PPIG*, Brighton, UK, 2005, pp. 19-29.

[12] T. VanDeGrift, "Coupling pair programming and writing: Learning about students perceptions and processes," in *Proc. SIGCSE*, Norfolk, Virginia, 2004, pp. 2-6.

[13] N. Katira, L. Williams, E. Wiebe, C. Miller, S. Balik, and E. Gehringer, "On understanding compatibility of student pair programmers," in *Proc. SIGCSE*, Norfolk, Virginia, 2004, pp. 7-11.

[14] P. Sfetsos, I. Stamelos, L. Angelis, and I. S. Deligiannis, "Investigating the Impact of Personality Types on Communication and Collaboration-Viability in Pair Programming," in *XP/Agile 7*, 2006, pp. 43-52.

[15] F. Padberg and M. Muller, "An Empirical Study about the Feelgood Factor in Pair Programming," *METRICS,* vol. 10, pp. 151-158, 2004.

[16] C.-W. Ho, S. Raha, E. Gehringer, and L. Williams, "Sangam: a distributed pair programming plug-in for eclipse," in *Proc. OOPSLA workshop on eclipse technology eXchange*, Vancouver, BC, Canada, 2005, pp. 73-77.

[17] R. Gittins and S. Hope, "A study of human solutions in eXtreme programming," in *Proc. PPIG*, Bournemouth, UK, 2001, pp. 41-51.

IEEE COMPUTER SOCIETY