Pair Programming: When and Why it Works

Jan Chong[1], Robert Plummer[2], Larry Leifer[3], Scott R. Klemmer[2], Ozgur Eris[3], and George Toye[3]

1 Stanford University, Department of Management Science and Engineering, Terman Engineering Center, 3rd Floor, Stanford CA 94305
jchong@cs.stanford.edu
2 Stanford University, Department of Computer Science, 353 Serra Mall, Stanford CA 94305
{plummer, srk}@cs.stanford.edu
3 Stanford University, Department of Mechanical Engineering, Building 530, 440 Escondido Mall, Stanford CA 94305
ozgur@stanford.edu; {toye, leifer}@cdr.stanford.edu

Pair programming is a software development technique where two programmers work together at a single PC. Over the past few years, pair programming has emerged as a promising method for creating higher-quality software in a time-efficient manner. It is a central aspect of many agile software development methods. While prior research has demonstrated the effectiveness of pair programming, there is still limited understanding as to *when* and *why* it is effective. Our research into the underlying reasons for success – and limitations of – pair programming employs a two-phase method. In the first phase, we are conducting ethnographic studies of software development teams in industry that currently employ pair programming. We will use the results of this phase of the research to drive the second phase of the research: a laboratory study of pair programming with professional developers as participants.

1 Introduction

This study is an investigation into the socio-cognitive factors of pair programming, a method of programming where two people work together shoulder-to-shoulder at a single computer. Studies of pair programming in university programming classes have shown that pair programming yields better design, more compact code, and fewer defects for roughly equivalent person-hours [1-5]. Studies have also noted that pair programmers exhibit greater confidence in their code and more enjoyment of the programming process [5-8]. Positive results with pair programming have lead to speculation that a collateral benefits of the practice may include improved morale and project knowledge shared efficiently across the development team in a manner that can be expected to improve productivity in subsequent development cycles [9].

While these results are compelling, the adoption of pair programming has faced resistance and skepticism from both managers and programmers. While this may simply be a result of either the novelty of the practice or skepticism of the larger methodological context (Extreme Programming/Agile methods) in which pair programming is often introduced, there is some evidence that pair programming may not necessarily be appropriate for everyone [10]. While prior research has demonstrated the effectiveness of pair programming, there is still limited understanding as to *when* and *why* it is effective. Additionally, the bulk of prior work has studied university students, and it is not clear to what extent these results transfer to professional programmers. Our study focuses on pair programming practices among software development professionals, both in a natural work setting and in the laboratory. Such an understanding would greatly aid managers and programmers in determining when and how to use pair programming to improve the software development process. Our research on pair programming fits into the broader picture of studying collaboration in software development.

2 Conceptual Framework

Pair programming, for the purposes of this study, describes a programming technique where all programming work is done by two programmers, working together at a single PC. Within the pair, work is split into two roles, known as the *driver* and the *navigator*. The driver is the person at the keyboard, responsible for the actual typing of the code being generated. The navigator is an active observer and monitor of the code being written. The driver and navigator collaborate on all aspects of the software development: design, coding, debugging, etc. They are in constant communication, asking and answering questions of each other. The two programmers may switch roles frequently in the course of a programming session.

The simplest view of why pair programming works is that two people make better design decisions than one. This view characterizes programming as a series of design decisions that are translated into code. The presence of a second individual distributes the cognitive task [11] of programming, aiding design discussion and error finding. More specifically, working in pairs has the following influences on decision-making:

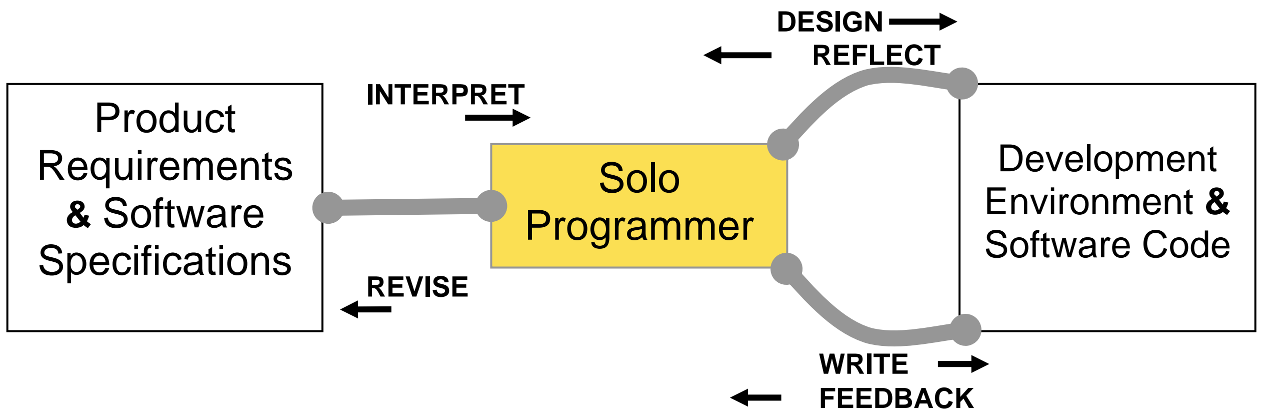
- Two individuals will have overlapping, but not identical, sets of information. When working together as a pair, sharing this increased pool of information can lead to better decision-making [12-14].
- Design collaboration affords a mutual apprenticeship, where through the collaboration each participant learns some of the technical skills and methods of their collaborator. This is one of the reasons why Beck [9] encourages pairs to rotate on a frequent basis.
- Collaborative design requires the negotiation of a shared understanding and mutual orientation. This negotiation process makes explicit the cognitive processes that are normally tacit when working individually [15].
- This negotiation process requires that programmers produce an account [16, 17] of goals, plans, decisions and actions. This appears to lead to a more thorough exploration of design options. This account production, verification, and affirmation leads to increased confidence by the programmers and vets flawed design ideas earlier.

Working in pairs also has influences when design decisions are translated into code. By monitoring the coding, the navigator can look for missed cases and typographical errors. The navigator can also think ahead of the code being typed at a given moment. One way of stating this is that the navigator can consider issues that have a longer time constant than those being addressed by the driver.

This social understanding of pair programming fits into the broader frame of studying small teams engaged in engineering design, and our research draws upon this background and literature. Tang has identified the importance of gesture and negotiation [18], and Brereton has demonstrated that in small design teams, rapid alternation between concrete and abstract issues yield the best products [19]. We postulate that this type of cognitive activity is characteristic of pair programming, where the driver operates in relatively more concrete thinking space by focusing on implementation, as opposed to the navigator, who deals with more abstract issues by focusing on higher level conceptual relationships and goals.

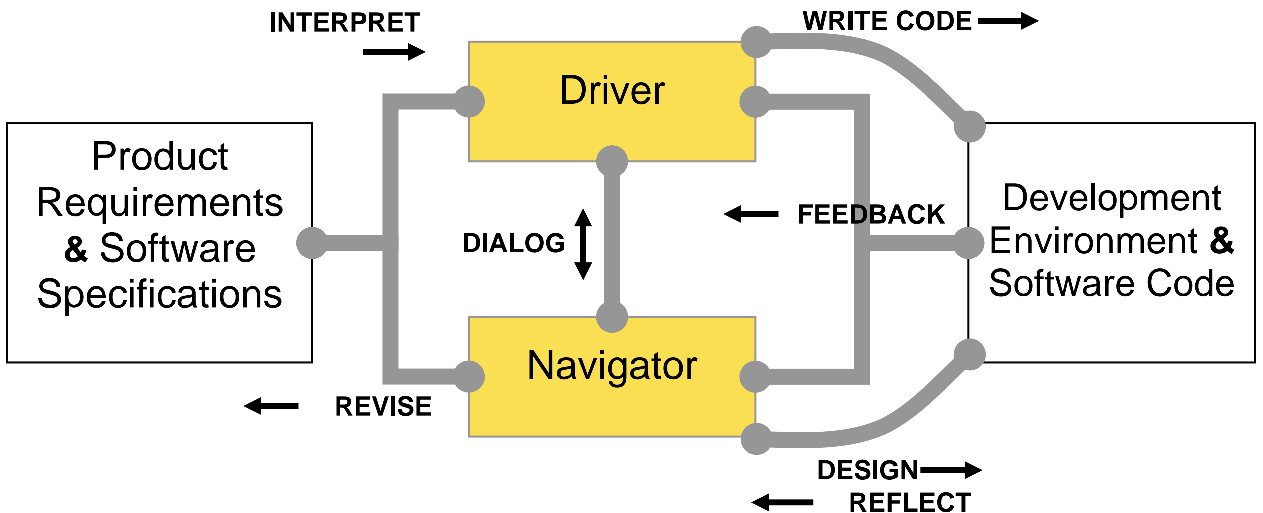
Observing engineering design teams has led us to believe strongly in the validity and utility of treating them as social entities whose learning activities are correlated with their design performance [20, 21]. Moreover, our observations resulted in a new insight, which was to recognize the importance of the role of coaching, including “self-coaching” within the team.

This research has also centered on the notion of information pathways. We would characterize solo programming as follows:



The solo programmer, in addition to interpreting and revising the specifications, must attend to the code at many levels, ranging from high-level design and design revisions to low-level entering of program statements and understanding of debugging results.

For pair programmers, the situation is different:



In the pair programming software development paradigm, the driver and navigator act on the specifications in tandem and develop code. The two actors alternate roles frequently during the task. In this alternating dynamic, we postulate that the navigator's focus on higher level conceptual relationships and goals allow him/her to take on a coaching role where he/she observes the driver's interaction with the code, identifies needs and opportunities, and intervenes to supply needed information and/or strategy to arrive at the desired goal, while the driver attends to the immediate coding task at hand. Inclusion of the navigator (coach) introduces multiple feedback paths for knowledge creation and error correction. This view is consistent with our earlier statement that the navigator is performing tasks with a longer time constant than those of the driver.

In our research, we are exploring the extent to which findings in other areas of engineering design can be applied to software development. Through additional field studies, and then through laboratory

experiments, we hope to arrive at more thorough understanding of why and when pair programming is effective.

3 Methodology

Our study has two phases: an ethnographic field study and a controlled laboratory experiment. We will use the results of the field study to inform and refine our experimental design.

3.1 Ethnographic Field Study

The first phase of our study is an ethnographic field study. We will conduct weekly observations of software development teams using pair programming at two companies in the Silicon Valley area for eight to twelve weeks. To capture a range of organizational environments, we have selected one small company, Company A, and one large company, Company B. Company A is a four year old technology company with roughly 70 employees. Company A has one team of five to seven people in size which practices pair programming. Company B is a six year old technology company; it currently has approximately 3000 employees. Company B employs at least four development teams that pair program. Each team has three to four people.

Observations at both sites will be conducted during the same period by the same observer. Each observation session will run between three to four hours. The observer will make a full audio record of each session, while will then be transcribed and annotated with notes taken in the field. Using analysis techniques from qualitative research [22, 23], we use these observations to verify our conceptual model of pair programming and refine the set of measures for the experimental portion of the study.

3.2 Laboratory Experiment

We intend to conduct pair programming sessions with professional contract programmers, recording every aspect of their interaction with the development environment, product requirement documentation, language specifications and each other. All subjects will be required to have at least two years of industry experience; ideally they will be experienced pair programmers. The programmers will be given a complex programming task (as opposed to a set of academic problems or a programming exercise) involving the development of an actual application. We will observe 6 programming pairs doing tasks that require 4-5 hours of joint work. As a control, we will observe 6 solo programmers doing the same tasks.

We will use the design-activity observatory at the Stanford University Center for Design Research (CDR) [http://cdr.stanford.edu]. The corpus of data collected in this environment spans the full range of performance variables from individual keystrokes through to frame-by-frame video interaction analysis and automated indexing. This facility allows us to capture four simultaneous channels of digital audio-video and all computer and whiteboard interactions. The workstation configuration will consist of a modern PC, two large, flat-panel monitors, and the Eclipse development environment [http://www.eclipse.org].

In our experimental sessions, there are three primary aspects of pair programming that we plan to measure: speed of development, quality of software design, and defect rate. We will measure development speed by recording the overall completion time, and use a coding of how individuals and pairs spend their time. To assess quality of code design, we will use software metrics such as code size and have the code rated by a set of independent raters. We will also use a coded transcript of the sessions to measure design decision-making. To measure software defect rate, we will create a full suite of unit tests for each piece of software produced. Our coded transcript will be used to measure information about the interaction between programmers. We will review the recoded video and audio to track noun phrases, questioning behavior, interpersonal and human/computer interaction rates and gestures. For the control group, we plan to have

our subjects give self-reports of their thoughts and actions during the programming session. While this is not an ideal record of the solo programming process, the self report data will provide some basis for comparison with the pair programmers. Finally, we will ask all of our participants to rate the realism of experimental task and environment, to determine whether we succeed in producing a setting representative of that in which a professional programmer normally works.

4 Current Progress

We have conducted eight weeks of preliminary fieldwork at Company A, as part of an exploratory study [15] on Extreme Programming (XP). Company A had a formed a new project team in January of 2004 to develop a Java-based network administration tool. By June of 2004, when observations began, the team had 12 members. Team members worked in a shared bullpen-like space, programming in pairs on dual monitor, dual keyboard work stations. With the exception of the team coach, the team members nearly always worked in pairs. Each observation session focused on a single pair of programmers, usually which ever pair seemed most interesting. The observer sat behind the pair during the session, taking notes on the dialogue exchanged between the pair, what actions transpired and their interactions with the rest of the team. Periodically, the observer would prompt the programmers to give brief explanations of their actions. When possible, the observer made an audio recording of the pair programming session, which was subsequently transcribed. The audio transcript and the notes were then rewritten to produce a detailed record of the session's events.

We then reviewed these transcripts using qualitative analysis techniques, beginning with multiple readings of the session records and open coding. As we refined the set of behaviors that we found interesting, we went back and recoded the data to rigorously identify any patterns of behavior that might be present. In current form, the data are coded for patterns of interaction between the programmers, shift in attentional focus (including interruptions), technical interdependence of the pair and the rest of the team, and asking questions of other team members. Our conceptual framework is largely based on this analysis.

We are currently in discussion with Company B for site access. We plan to return to Company A to conduct further observations of their team, focused primarily on pair programming. We will then analyze this field data to further refine and verify the components of our conceptual framework. Following Eris's approach [24], we will use the results of the fieldwork to identify salient aspects of pair programming for study in further detail in the laboratory.

We are also developing software tools that will aid in the analysis of the experimental sessions. This software will operate as a plug-in to the Eclipse IDE, and will log semantic operations such as class creation, method creation, and refactoring. It will then time-correlate each of these semantic operations with the video stream, and produce a visualization of activity over time. This logging can be used to generate automatically a set of measures for comparing development styles, and serves as an analysis baseline that can be extended by hand-coding.

References

1. Nicolescu, R. and R. Plummer, *A Pair Programming Experiment in a Large Computer Course.* Romanian Journal of Information Science and Technology, 2003. **6**(1-2): p. 199-216.
2. Williams, L., et al., *Strengthening the Case for Pair-Programming.* IEEE Software, 2000. **17**: p. 19-25.
3. Cockburn, A. and L. Williams. *The Costs and Benefits of Pair Programming*. in *First International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2000)*. 2001.
4. Williams, L. and R. Kessler, *Pair Programming Illuminated*. 2003, Boston, MA: Addison Wesley.
5. Hanks, B., *Empirical Studies of Pair Programming*, in *2nd International Workshop on Empirical Evaluation of Agile Processes (EEAP 2003)*. 2003.

6. Nosek, J.T., *The Case for Collaborative Programming.* Communications of the ACM, 1998. **41**(3): p. 105-108.
7. Hanks, B., et al. *Program Quality with Pair Programming in CS1*. in *9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. 2004.
8. McDowell, C., et al. *The Impact of Pair Programming on Student Performance, Perception and Persistance*. in *25th International Conference on Software Engineering*. 2003.
9. Beck, K., *Extreme Programming Explained: Embrace Change*. 2000: Addison-Wesley.
10. Dick, A.J. and B. Zarnett, *Paired Programming & Personality Traits*, in *XP2002*. 2002.
11. Hutchins, E., *Cognition in the Wild*. 1995, Cambridge, MA: The MIT Press. 381.
12. Moreland, R.L., L. Argote, and R. Krishnan, *Socially shared cognition at work: Transactive memory and group performance*, in *What's social about social cognition?* J.L. Nye and A.M. Brower, Editors. 1996, Sage: Thousand Oaks, CA. p. 57-85.
13. Laughlin, P.R., et al., *Generality of a theory of collective induction: face-to-face and computer-mediated interaction, among of potential information, and group versus member choice of evidence.* Organizational Behavior and Human Decision Processes, 1995. **63**: p. 98-111.
14. Wegner, D.M., *Transactive memory: A contemporary analysis of the group mind*, in *Theories of group behavior*, B. Mullen and G.R. Goethals, Editors. 1986, Springer-Verlag: New York. p. 185-205.
15. Chong, J., *Social Behaviors on XP Teams: A Comparative Study*, in *Agile 2005*. 2005: Denver, CO.
16. Dourish, P., *Where The Action Is: The Foundations of Embodied Interaction*. 2001, Cambridge, MA: The MIT Press.
17. Garfinkle, H., *Studies in Ethnomethodology*. 1967, Englewood Cliffs, NJ: Prentice Hall.
18. Tang, J.C., *Toward an understanding of the use of shared workspaces by design teams admin*. 1989, Stanford University.
19. Brereton, M.F., *The role of hardware in learning engineering fundamentals: An empirical study of engineering design and product analysis activity*. 1998, Stanford University.
20. Mabogunje, A.O., *Measuring conceptual design process performance in mechanical engineering: A question based approach*. 1995, Stanford University.
21. Eris, O., *Perceiving, comprehending and measuring design activity through the questions asked while designing*. 2002, Stanford University.
22. Strauss, A. and J. Corbin, *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. 1990, Newbury Park, CA: Sage Publications.
23. Glaser, B. and A.L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. 1967, London: Wiedenfeld and Nicholson.
24. Eris, O., *Effective Inquiry for Innovative Engineering Design*. 2004, Boston, MA: Kluwer Academic Publishers.