Investigating Learning in Novice, Pre-Professional Pair Programming Interactions

Adam Lupu

Northwestern University

Investigating Learning in Novice, Pre-Professional Pair Programming Interactions

Abstract

In this paper, I propose research that aims to help computer programming instructors achieve three outcomes: 1) to better prepare students for future employment as software developers, 2) to diversify both the current academic population and the existing professional population of computer programmers. 3) to improve individual student learning, performance, enrollment, confidence, and enjoyment. I begin with a motivation for the study of computer programming education and a specific practice known as pair programming. I then propose two research questions and the methods for studying them as well as the theoretical frameworks that will help guide my interpretation of the findings. I conclude by defining my research goals, my intended contribution to the field, and the future research this study could ignite.

Motivation

Computer programming is a highly sought after skill, and industry demand for software developers with extensive programming expertise continues to grow (Prabhakar, Litecky, and Arnett, 2005). Until recently, aspiring programmers looking to fill this demand had two options: learn on their own or study at a college or university. Learning on one's own can be quite challenging, but learning in a university can be worse. Many students in undergraduate computer programming classes suffer from poor grades, frustration, and mental fatigue (Braught, Wahls, and Eby, 2011; Murphy, Fitzgerald, Hanks and McCauley, 2010). Additionally, at least one third of students enrolled in a college-level introduction to programming course will either fail or drop out (Bennedsen and Caspersen, 2007). Unfortunately, students who continue and graduate with degrees in computer science often lack necessary skills for employment, including communication and collaboration (Begel and Simon, 2008).

Perhaps in response to these findings, a third educational option for novice programmers has emerged. Commercial organizations like Code Academy and General Assembly offer training programs that teach a modern programming language and professional software development skills to beginners. Organizations like RailsBridge seek to entice women, who in 2009 received only 18% of undergraduate

computer science degrees (NCWIT, 2011), into the software development industry. Yet, to date, there are no existing studies analyzing the learning that takes place in these organizations.

With computer programming education expanding beyond universities and do-it-yourself manuals, there is a growing opportunity for the development of more effective pedagogical practices. In this research proposal, I argue that a closer examination of one specific practice, known as *pair programming*, can help improve the educational opportunities that universities and commercial training programs provide. To make this argument, I will begin with a description of pair programming followed by examples of its many useful, but currently unexplained, benefits. I will conclude the argument with an explanation of how pair programming research can render learning processes visible through student interactions and the questions and methods needed to do so.

Pair programming involves two programmers working together to construct a single sample of computer code or perform a programming task (e.g. debugging). Paired programmers share a computer monitor that displays the code they are working on, and often a single keyboard for text entry. When pairing, it is common for programmers to take on one of two roles (Williams and Kessler, 2003). One programmer will serve as the "driver," entering code through a keyboard interface. The other will serve as the "navigator," observing, making suggestions, and searching for appropriate solutions on a second computer (Williams and Kessler, 2003). Pairs may switch roles often, and those that do typically use two keyboards (one for each programmer) in order to expedite this transition. Pair programming is often used in professional software development communities and appears in commercial training environments, as well as some undergraduate and high school classrooms (Williams, Wiebe, Yang, Ferzli, and Miller, 2002).

Existing literature on the pedagogical adoption of pair programming indicate several potential benefits (e.g. enhanced learning outcomes, improved student affect, improved student performance). However, why these benefits manifest in some classes and not in others remains unexplained. I will describe the potential benefits of using pair programming to train novice programmers, as well as some of

the potential drawbacks. Then I will argue why the results from previous studies merit a more deliberate investigation into how pair programmers learn.

One significant benefit to pair programming is that it is a common professional practice situated in the workplace (Chong et al., 2005). Therefore, exposure to pair programming in the classroom could familiarize students with a method they will be required to employ on the job. Similarly, pair programming could supplement existing curricula with additional learning opportunities that could support students in future careers. For example, students might learn methods for collaboratively solving problems, communicating their understanding verbally, and resolving disagreements with their peers. Students programming in isolation would be unable to achieve these learning outcomes.

Another benefit to pair programming appears to be improved student response to introductory computer programming classes. Students report less frustration, fewer feelings of isolation, and greater overall enjoyment in classes that employ pair programming compared to classes that do not (McDowel, Werner, Bullock, and Fernald, 2006). Improving student experiences in introductory classes by adopting pair programming also increases the likelihood that they will continue taking more advanced classes in computer programming (McDowel et al., 2006). Therefore, students who begin their education with pair programming are more likely to go on to learn advanced programming techniques than students who do not. Again, why this trend occurs is not yet well understood.

In addition to helping undergraduate computer science programs retain more students, pair programming may also serve to diversify the population of students who graduate with degrees in computer science. In the United States, women make up less than one fifth of undergraduate computer science classrooms (NCWIT, 2011). Carter (2006) found a larger proportion of women than men reject the study of computer science because they prefer a more "people-oriented" field of study (p. 30). When exposed to pair programming, however, women reported greater confidence and demonstrated a greater likelihood to take future computer programming classes (Braught, Wahls, and Eby, 2011). Braught, Eby, and Wahls (2008) also found that students with low standardized test scores in math show significant improvements in individual programming skill when enrolled in classes that use pair programming. Does

pairing with another programmer improve the learning outcomes for women and students with low math scores as well as their overall motivation and performance? The existing data points to this question but has not yet answered it.

Finally, pair programming may lead to improved individual performance. Students who train using pair programming typically obtain equivalent or higher grades (Braught, Wahls, and Eby, 2011) and demonstrate a greater aptitude for synthesizing and applying what they have learned to multiple problem types compared to students who program alone (Williams et al., 2002). However, like the benefits mentioned above, the causes for these learning and performance improvements are not well understood or explained. In addition, some studies indicate pair programming use can have negative consequences. For example, there is some evidence that pair programming leads to a reduction in the effort of at least one of the programmers in a pair (Hannay, Dyba, Arisholm, and Sjøberg, 2009). Paired programmers may also produce less efficient solutions to problems (Hannay et al., 2009), and student performance may be adversely affected if the relationship with their partner becomes contentious (Williams, Layman, Osborne, and Katira, 2006). Again, we are left to wonder why programming in pairs might lead some students to perform better while leading other students to perform worse.

I have described how the inclusion of pair programming in some learning environments results in significant affective and quantifiable benefits for several students, while its use in others can lead to less effort, poorer solutions, or reduced individual performance. Since existing empirical results are contradictory and unexplained, I call for a qualitative investigation into how pair programming affects student learning and why previous studies achieved these results. I propose the following research project to help explain previous findings as well as offer suggestions for computer programming instructors and training organizations to maximize the benefits of pair programming while minimizing the risks.

Research Questions and Methods

The main purpose of pair programming is the construction and testing of code. In order to investigate the learning that takes place during pair programming and identify explanations for

performance improvements, we must begin with an analysis of how programmers interact to achieve this specific purpose. Therefore, I propose:

Question 1: *How do the interactions between novice programmers lead to the construction and testing of code during pair programming sessions?*

To inform my data collection and analysis process in researching this question, I will draw on Jordan and Henderson's (1995) approach to interaction analysis. Interaction analysis is a research methodology that calls for the recording, repeated viewing, codification, and analysis of rapidly evolving interactions. This type of analysis is useful for studying pair programming interactions because it allows for pausing, replaying, and unpacking student problem solving techniques and learning processes for which researchers already know the outcome. Therefore, researchers can focus on interactions that led to a particular result or breakdown.

To help explain previous findings, I will focus on interactions that led to successful code constructions as well as those that led to a diminution of effort or to inefficient debugging solutions. I will also look for indications of students competently employing a technique or conceptual understanding previously unavailable to them. Understanding the role of pair programming in these moments of student transformation may help explain both overall student satisfaction and the appeal of pair programming to a more diverse student body. I will therefore look for evidence consisting of two parts: 1) an example of a student expressing an inability to comprehend or perform a specific task and 2) a following example of that student expressing verifiable comprehension or demonstrating competent performance of that task.

To acquire the data needed to study these interactions, I will create audio/video records of pair programming sessions between novice programmers at a commercial training organization. Screen capture software (e.g., *Screenflow*) allows for the simultaneous capture of the code students are working on and an audio/video record of their interactions. Using this type of software, I will record at least three pairs of programmers for one session per week for the entirety of the twelve-week course. This will result in a minimum of 36 hours of video data.

I will transcribe and code student interactions (employing a second coder for inter-rater reliability). To help identify moments where problems arise and how pairs interact to solve them, I will pay specific attention to when students stop typing, when they resume typing, and the verbal and gestural exchanges that occur between these two events. I will examine and attempt to explain how students decide who "drives" and who "navigates." I will also attempt to identify any problem solving and debugging interactions that are common across pairs.

Question 1 may help explain previous findings and unpack the interactions that lead to performance improvements. In order to construct useful recommendations to instructors and inform future curricula, we must also identify student learning processes, as well as how these processes influence and are influenced by the input from their programming partners. Therefore, I propose:

Question 2: *What learning outcomes do students experience and how are they influenced by interactions with their programming partners?*

To acquire the data necessary to identify these learning outcomes, I will use the audio/video record mentioned previously, as well as three additional methods of collection. First, I will conduct recorded clinical interviews with both students individually immediately following their pair programming sessions. I will begin each interview with a coding task similar to the one students just completed with a partner. This will help me observe and analyze the student's process when programming alone (for comparison with paired programming data) as well as gauge the success or failure of the pair programming session for that student. I will then prompt students to reflect on their paired programming session and identify what they learned as well as any specific moments they believe contributed to their understanding or ability.

Students in the training program I intend to study are also required to blog about their learning process throughout their participation in the course. I will collect these blog posts and analyze them for any narratives that might describe the learning outcomes that occurred during pair programming sessions. Additionally, I will monitor student interactions online as they communicate through class wikis, chat, and email. This data may give further insight into student learning processes, outcomes, and origins.

I will use the analysis from questions 1 and 2 to make suggestions for maximizing the benefits of pair programming and minimizing the drawbacks. I will pay specific attention to potential explanations for the empirical findings of previous studies. For example, to explore why previous studies found pair programming may lead to improved individual performance, I will identify specific moments where students exhibit improved performance and look for pair programming interactions that led to these transformations. To attempt to explain why students report less frustration, less isolation, and more enjoyment during pair programming, I will look for specific indications of supportive behavior, enthusiasm, and mutual satisfaction. I will then analyze the interactions and events that preceded these displays of gratification and support, and suggest scenarios that instructors can use to guide students toward these experiences. For interactions that lead to a breakdown in communication, a reduction of effort, or inefficient solutions, I will attempt to determine what led to these results and how they might be avoided.

To further inform my suggestions for pedagogical practices in pair programming, I have intentionally left my research questions broad in an effort to invite unexpected results. I believe qualitative research intended to promote future research and suggest new practices in a lightly studied field demands openness to such results. While maintaining this openness throughout my research, I will also draw on existing theoretical frameworks to help specify strategies for interpreting my findings.

First, I will look for indications of Vygotsky's (1978) theory of the "zone of proximal development" (ZPD). The ZPD defines an individual's learning potential as the distance between the understanding and performance the individual can achieve alone and the understanding and performance the individual can achieve through the guidance of an instructor or more capable peer. In analyzing pair programming, I could attempt to identify a student's ZPD by determining the level of comprehension and performance a student is capable of before pairing with another programmer and the difference in their comprehension and performance afterwards. I could then analyze what forms of guidance occurred during the pair programming session to determine how this transformation occurred. The ZPD may also help us identify learning outcomes that result when programmers of differing acumen are paired together.

The second theoretical framework, Roschelle's (1992) work on "convergent conceptual change" (CCC) considers how two or more people construct shared meaning through physical and verbal conversation. In pair programming, CCC could be used to analyze conversations between novice programmers, especially those that result in a shared understanding demonstrating greater expertise. For example, I could identify conversations that started with a disagreement between programmers on the nature of the assignment or on the methods for proceeding and ended with obvious concurrence between the pair. I could then analyze the conversation to determine what led to the successful formation of a common understanding and any learning outcomes that resulted.

The third framework, Hutchins' (1995) theory of "distributed cognition" describes how groups of people and the sensory representations they work with combine to make decisions and perform actions. In pair programming, two programmers work with a visual representation (their code) and an audio/visual representation (their reference computer or assigned prompts). The theory of distributed cognition could help identify how this system of four elements (two people, their code, and a reference computer) determines how to construct code and solves programming problems.

A fourth theoretical framework, Stevens and Hall's (1998) work on "disciplined perception" describes the process by which a more insightful individual directs the attention of a collaborator towards specific sections of a shared visual representation. Disciplined perception could help guide the analysis of pair programming during occurrences where one programmer with a better understanding of the problem directs the attention of his or her partner towards a specific section of code and its relation to a specific thought process. During a single pair programming session each student could demonstrate greater knowledge or more disciplined perception than his or her partner as the tasks and problems change. This alternating of expertise may even show up physically through a swapping of roles (i.e. driver and navigator exchange positions). Therefore, the study of pair programming may contribute to existing theory by presenting for analysis collaborative interactions where role switching and alternating expertise may affect learning.

Conclusion

This research project has three goals. It seeks to understand how and what is learned when novices program in pairs, to explain existing empirical results in pair programming research that are currently unexplained, and to offer guidelines for the design of learning environments that maximize the benefits and minimize the drawbacks in the educational use of pair programming. If this research is successful, it will lay the foundation for future design based research to optimize how instructors and curricula prepare students for future employment as software developers, promote diversity in programming classrooms and workplaces, and stimulate student learning, performance, enrollment, confidence, and enjoyment.

References

Arisholm, E., Gallis, H., Dyba, T., & Sjoberg, D. I. K. (2007). Evaluating pair programming with respect to system complexity and programmer expertise. *Software Engineering, IEEE Transactions on*, *33*(2), 65–86.

Begel, A., & Simon, B. (2008a). Struggles of new college graduates in their first software development job. *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, SIGCSE '08 (pp. 226–230). New York, NY, USA: ACM. doi:10.1145/1352135.1352218

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *SIGCSE Bull.* *39*(2), 32–36. doi:10.1145/1272848.1272879

Braught, G, Eby, L. M., & Wahls, T. (2008a). The effects of pair-programming on individual programming skill. *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, SIGCSE '08 (pp. 200–204). New York, NY, USA: ACM. doi:10.1145/1352135.1352207

Braught, G, Wahls, T., & Eby, L. M. (2011). The case for pair programming in the computer science classroom. *Trans. Comput. Educ.*, *11*(1), 2:1–2:21. doi:http://doi.acm.org/1921607.1921609

Carter, L. (2006). Why students with an apparent aptitude for computer science don't choose to major in computer science. *ACM SIGCSE Bulletin*, *38*(1), 27. doi:10.1145/1124706.1121352

Chong, J., Plummer, R., Leifer, L., Klemmer, S. R., Eris, O., & Toye, G. (2005). Pair programming: When and why it works. *Proc. The 17th Workshop of the Psychology of Programming Interest Group PPIG17* (pp. 43–48).

Dyba, T., Arisholm, E., Sjoberg, D. I. K., Hannay, J. E., & Shull, F. (2007). Are two heads better than one? On the effectiveness of pair programming. *Software, IEEE*, *24*(6), 12–15.

Hannay, Jo E., Dybå, T., Arisholm, E., & Sjøberg, D. I. K. (2009). The effectiveness of pair programming: A meta-analysis. *Information and Software Technology*, *51*(7), 1110-1122. doi:10.1016/j.infsof.2009.02.001

Hutchins, E. (1995). How a cockpit remembers its speeds. *Cognitive Science*, *19*(3), 265-288. doi:10.1016/0364-0213(95)90020-9

Jordan, B., & Henderson, A. (1995). Interaction analysis: Foundations and practice. *Journal of the Learning Sciences*, *4*, 39-103. doi:10.1207/s15327809jls0401_2

McDowell, C., Werner, L., Bullock, H. E., & Fernald, J. (2003). The impact of pair programming on student performance, perception and persistence. *Proceedings of the 25th international conference on Software engineering* (pp. 602–607).

McDowell, C., Werner, L., Bullock, H. E., & Fernald, J. (2006). Pair programming improves student retention, confidence, and program quality. *Communications of the ACM*, *49*(8), 90–95.

Murphy, L., Fitzgerald, S., Hanks, B., & McCauley, R. (2010a). Pair debugging: a transactive discourse analysis. *Proceedings of the Sixth international workshop on Computing education research*, ICER '10 (pp. 51–58). New York, NY, USA: ACM. doi:10.1145/1839594.1839604

NCWIT (2011) http://www.ncwit.org/about.factsheet.html retrieved on 12/2/11

Prabhakar, B., Litecky, C. R., & Arnett, K. (2005). IT skills in a tough job market. *Commun. ACM*, *48*(10), 91–94. doi:10.1145/1089107.1089110

Roschelle, J. (1992). Learning by collaborating: Convergent conceptual change. *Journal of the Learning Sciences*, *2*, 235-276. doi:10.1207/s15327809jls0203_1

Stevens, R., & Hall, R. (1998). Disciplined perception: Learning to see in technoscience. *Talking mathematics in school: Studies of teaching and learning*, 107-149.

Vygotsky, L. (1978). *Mind in society*. Harvard University Press.

Werner, L., & Denning, J. (2009). Pair programming in middle school: What does it look like? *Journal of Research on Technology in Education*, *42*(1), 29-49.

Williams, L., & Kessler, R. R. (2003). *Pair programming illuminated*. Addison-Wesley Professional.

Williams, L., Layman, L., Osborne, J., & Katira, N. (2006). Examining the compatibility of student pair programmers. *Agile Conference, 2006* (p. 10 pp.-420). Presented at the Agile Conference, 2006, IEEE. doi:10.1109/AGILE.2006.25

Williams, L., Wiebe, E., Yang, K., Ferzli, M., & Miller, C. (2002). In support of pair programming in the introductory computer science course. *Computer Science Education*, *12*(3), 197–212.

Wright, K. (2010). Capstone programming courses considered harmful. *Commun. ACM*, *53*(4), 124–127. doi:10.1145/1721654.1721689