The Importance of Producing Shared Code through Pair Programming

Mehmet Celepkolu Kristy Elizabeth Boyer
Computer & Information Science & Engineering
University of Florida
Gainesville, Florida, USA 32611
mckolu@ufl.edu, keboyer@ufl.edu

ABSTRACT

Collaborative learning frameworks such as pair programming have been shown to be highly effective for computer science learning. Skeptics of this approach often refer to the risk of one student relying on a stronger partner to solve the problem. Lending weight to this skepticism, many theories emphasize the importance of learner autonomy. Therefore, it is reasonable to hypothesize that a hybrid pair programming paradigm—one in which partners work together side-by-side at two separate computers and produce their own versions of the code—may be even more effective than traditional pair programming. To investigate this hypothesis, we conducted a study in which 200 introductory programming students were paired and then placed in either a pair-programming condition (two students at one computer) or a hybrid condition (two students at two computers). The results show that traditional pair programming fostered comparable learning gains as measured on an individual post-test, and significantly higher student satisfaction, than the hybrid approach. These findings highlight the importance of not just collaborating, but working together on shared code, for novice computer science learners.

KEYWORDS

Collaborative Problem Solving, Pair Programming

ACM Reference format:
Mehmet Celepkolu and Kristy Elizabeth Boyer. 2018. The Importance of Producing Shared Code through Pair Programming. In *SIGCSE '18: 49th ACM Technical Symposium on Computer Science Education, Feb. 21–24, 2018, Baltimore, MD, USA.* ACM, NY, NY, USA, 6 pages. https://doi.org/10.1145/3159450.3159516

SIGCSE '18, February 21–24, 2018, Baltimore, MD, USA

ACM ISBN 978-1-4503-5103-4/18/02…$15.00
https://doi.org/10.1145/3159450.3159516

1 INTRODUCTION

Learning computer science is notoriously challenging [1]. While recent years have seen an increase in research on how to effectively support novices, the failure rate of introductory computer science courses is still high. Watson and Li [25] performed a systematic analysis of 161 CS1 courses in fifteen different countries and found that the worldwide pass rate is about 67.7% without any substantial difference between grade level, country, or class size. It is essential that we identify strategies that are effective for supporting students through this rigorous course.

Collaboration has been shown to be an effective approach that can increase performance [17] through which learners exchange ideas [10,11], improve critical thinking skills, and co-construct knowledge [9]. Even as collaborative learning for computer science is becoming increasingly ubiquitous, a concern among skeptics is the "free-rider" phenomenon, in which one student relies on a more knowledgeable classmate to solve a problem [7]. Although collaborative techniques such as pair programming have been overwhelmingly shown to improve outcomes for students overall [14,26], skeptics often extol the virtues of individual problem solving. Furthermore, many learning theories emphasize that in order to become responsible and autonomous learners, students need to take control of their own learning [2]. An active line of investigation in the computer-supported collaborative learning community examines the optimal balance of individual work and collaboration for supporting student learning. Previous studies have shown that a combination of collaborative learning and individual work produces better learning outcomes and enjoyment than collaborative learning or individual learning exclusively in CS [5,19].

We are faced with a pressing open question in computer science education: How can we best support students in learning introductory computer science through a combination of collaborative and individual problem solving? This paper investigates this broad question in the context of pair programming for CS1. Pair programming, in which two students solve a problem together at the same computer, has significant advantages over solo programming [13]. However, we hypothesized that a hybrid approach may be even more effective, in which students are still placed in pairs, but are seated at different computers and each student has autonomy

over his or her own version of the code. To investigate this hypothesis, we divided 200 introductory programming students into one of two different conditions. In the *Traditional* condition, pairs worked on the same code at one computer and were responsible for producing one common solution. In the *Hybrid* condition, pairs worked at side-by-side computers and each partner was responsible for producing their own code. In both conditions, students were encouraged to communicate often and to work closely together. The results showed that traditional pair programming provided comparable learning gains and significantly higher satisfaction compared to the hybrid approach.

Figure 1. Students engaged in pair programming in a CS1 lab.

2 BACKGROUND

This work builds upon a background of both theory and empirical results. This section first discusses the theoretical framing in terms of collaborative learning theory and shared goals. Then, it presents empirical results on the effectiveness of pair programming and techniques for pairing students.

2.1 Theoretical Framework

Pair programming is a form of collaborative learning. Collaborative learning theory holds that learning first occurs on a social level between two or more people in some environment or context, and then is internalized by each person on an individual level [24]. This theory is in stark contrast to other theories that focus solely on an individual student acquiring knowledge and skills and later putting those skills to work in richer contexts that involve collaboration. Collaborative learning has been shown to create a positive learning climate, promote social interaction, and foster critical thinking through peer interaction [12]. According to Roger and Johnson [20], collaborative learning is most valuable for complex or conceptual tasks that demand higher-level reasoning strategies and critical thinking. Computer science problem solving is a prime example of these sorts of tasks.

Successful collaboration is built on a relationship that aims to solve a problem, create a common outcome [23] or achieve shared goals [21]. Thus, collaboration is more than an interaction of two or more people; it is a "coordinated, synchronous activity that is the result of a continued attempt to construct and maintain a shared conception of a problem" [21].

2.2 Empirical Results on Pair Programming

Pair programming is widely used in computer science education, as it can encourage better code quality, greater enjoyment and better learning outcomes compared to individual programming. Previous studies have shown that students feel more productive in pair programming [18], are less likely to drop the course [14], and create more concise and higher-quality code [6]. A study conducted with 1200 students in two US universities showed that students who completed programming activities in pairs received equal or better exam scores, project scores and course grade, and were also better equipped to continue in a computer science-related major the next year [27]. Similarly, a study of 176 students who worked in pairs showed that they were more successful in completing later programming tasks individually. They also performed better on an exam and were more confident about their code [3]. This paper adds to the substantial body of prior work by investigating the impact of letting students collaborate with a partner while taking responsibility for their own separate version of the code.

3 METHODS

We conducted a quasi-experimental study with two conditions. In the *Traditional* pair programming condition, each student was paired with a partner seated at the same computer. In the *Hybrid* pair programming condition, students were paired but seated at neighboring computers. In both cases, students were encouraged to work closely together, but in the *Hybrid* condition each student turned in a separate program, while in the *Traditional* condition they turned in one shared solution. In addition to quantitative methods, we qualitatively examined students' reflection papers in which they reported their experiences of pair programming at the end of semester. Examination of the reflection papers allowed us to deeply explore the reasons for the lack of difference in learning outcomes and significantly different pair programming satisfaction between conditions.

3.1 Participants and Setting

The participants were students who were enrolled in CS1 (introductory computer programming for majors) in spring 2016 at a large public university in the southeastern United States. There were 375 students enrolled in the course. All 375 students attended weekly programming labs and completed the lab exercises described in this section. Of these enrolled students, 278 consented to have their data collected. No incentive was provided for opting in to data collection. Of these 278 consenting students, we collected data from 200 (the remaining 78 consenting students were either absent on the data collection day or they were paired with a non-consenting student). Of the 200 consenting students there were 43 women (21.5%) and 157 men (78.5%). Participants' mean age was 19.1 (range: 18-30) and there were 130 Freshmen (65%), 37 Sophomore (18.5%), 27 Junior (13.5%), 5 Senior (2.5%), and 1 graduate student (0.5%). Students described themselves as White (46%), Black (4%), Hispanic (17%), Asian (19%), Multiracial (13%), and Other (1%). The majority of the participants were from computing-related majors: Computer

Science (37%), Computer Engineering (29%), Other Engineering fields (15%) and Other (19%).

The CS1 lecture class was taught by a tenure-track faculty member, and met three times per week in a large lecture hall. Lecture classes included several active learning and live debugging activities per class. In addition to lectures, each student attended a mandatory two-hour lab each week facilitated by an undergraduate teaching assistant. In the lab, students pair programmed on the required assignment. There were 18 different lab sections each week, each with approximately 20 students. After students completed the lab assignment, post-quiz, and post-survey, they were free to leave. Students attended the mandatory lab activities as a part of their course requirement for 14 weeks. In the first week of the semester, students learned about the course logistics and filled out several surveys such as demographics, experience in programming, motivation and personality. Students were also informed about the motivation for, and implementation of, pair programming in the first week. After that, each student worked with either a randomly assigned or self-selected partner depending on the lab structure of each week for the rest of semester. Students were paired with a variety of different partners throughout the semester.

We conducted this study in week five of the course. Prior lab classes had met in weeks two, three, and four. The study was conducted as part of students' regular two-hour weekly lab. All students were paired with a pre-arranged partner by their teaching assistant. Each of the 18 different lab sections was randomly assigned to one of two conditions so that all students in a lab were in the same condition. 90 students were in the *Traditional* condition and 110 were in the *Hybrid* condition. In the *Traditional* condition, students began working on the assignment with their assigned partner to construct a mutual solution. In the *Hybrid* condition, each student was responsible for their own code but they were allowed to collaborate whenever they wanted.

3.2 Programming Task

Students were provided with a description of a problem to solve in their labs. Each week, the lab was accompanied by instructions on pair programming:

PAIR PROGRAMMING: You will complete this lab exercise using pair programming. Your TA will assign you to pairs. You must work with the partner your TA assigns. In pair programming, both programmers use the same computer. They take turns in two roles: driver and navigator. The driver controls the keyboard and mouse, and the navigator watches for errors and thinks about the "big picture" goal. Both driver and navigator should actively be talking to each other while they work, asking questions and saying what they are thinking. You should switch roles of driver and navigator approximately every 15 minutes.

In the data collection week, students in the *Traditional* pair programming condition followed the described instructions. In the *Hybrid* condition, lab teaching assistants announced the instructions that students would again work with an assigned partner and solve the same problem, but they would be working at two different computers. Also, they were told that they were responsible for their own code but were allowed to discuss the solution as much as they wanted.

The objective of the lab was for students to practice *while loops*. The programming task for the lab week was to write a program with two parts: Calculator and Sentence Separator. Sample outputs were provided for students for each part. In the calculator portion, students were asked to use a loop to create a four-function calculator that stops calculating after the result reaches a certain value. The program was required to give the user the option to multiply, divide, add, or subtract, and to use the last value calculated. In the Sentence Separator portion, students were asked to use the substring function for the String class to take in a sentence and print out each word one-by-one on separate lines. An excerpt from the lab description reads as follows:

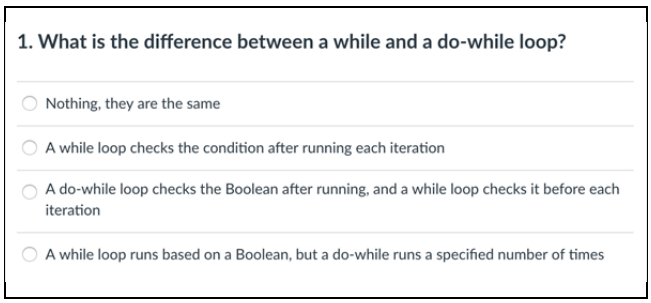
Consider a space to be the only separator of words. First, ask the user for a sentence, then create substrings until the sentence is left with only one word, and print out each substring and the last word. You may only use methods in the String class (there is a way to use Scanner to do this task but you must not use that approach). Hint: there is a nice solution to this problem using while or for loops. The for loop solution is slightly more elegant. You may use whichever one you choose.

3.3 Data

After they completed the assignment, or after the allocated time (two hours) had elapsed, students completed a post-test consisting of eight multiple choice questions. The post-test was constructed by the teaching staff and reviewed by the professor of the course prior to each week. These questions consisted of both conceptual and programming questions. Figure 2 shows two sample questions. Students also completed a ten-question post-survey on their learning experience, including one item that asked students to rate how often they talked to their partner about the problem.

4 ANALYSIS and RESULTS

In this section, we present the results of learning outcomes and student satisfaction for the *Traditional* and *Hybrid* conditions. The goal of this study was to investigate whether a hybrid pair programming paradigm, in which partners work together side-by-side at two separate computers and produce their own versions of the code, produces better learning outcomes and satisfaction compared to traditional pair programming.



2. What is the value of "b" after the code has run?

```
int b = 4;
while (b < 100){
    b++;
    if (b == 65){
        break;
        b = 150;
    } else if (b == 32){
        continue;
    }
    }
```

Figure 2. Sample conceptual and programming post-test questions.

4.1 Interaction Time

Before comparing learning outcomes and satisfaction, we wished to gain insight into whether students indeed worked together closely in the *Hybrid* condition as they had done in the *Traditional* condition. After the session, we asked students the survey question, "*How often did you and your programming partner talk about the problem?*". This item was represented on a Likert scale of 1-5 (1 being the least and 5 being the most). Perhaps unsurprisingly, students in the *Hybrid* condition reported less time talking together (N=110, M=3.86, SD=0.66) than students in the *Traditional* condition (N=90, M=4.45, SD=1.1). This difference is statistically significant (Mann-Whitney U = 6372, p < 0.01).[2]

4.2 Learning Outcomes

To investigate whether the *Hybrid* approach supported more effective learning than the *Traditional* condition, we performed a top-level analysis comparing posttest scores from the two different conditions (Figure 3). There was no significant difference between the *Hybrid* condition (M=4.77, SD=1.85) and the *Traditional* condition (M=5.12, SD=1.97), $t(198)$ =1.29, p=0.2, d=0.18.

4.3 Satisfaction

We compared the two conditions, *Traditional* and *Hybrid*, based on the satisfaction post-survey questions:

- My programming partner was willing to discuss the details of the problem when needed.
- My programming partner was supportive.
- My programming partner did his/her fair share of the work.

We calculated the satisfaction score by summing the three questions' scores. Each survey item was represented on a Likert scale from 1-5 (1=strongly disagree; 5=strongly agree); therefore, the minimum possible summed score on the three items was 3 and the maximum possible score was 15. The mean score for all students was 12.95 and the median was 13. The results show that the students in the *Traditional* condition (M=13.36, SD=1.74) reported significantly higher satisfaction than the students in the *Hybrid* condition (M=12.63, SD=2.26). This difference is statistically significant (Mann-Whitney U=4081.5; Z=-2.198), p=0.028 (Figure 3. Table 1 shows the satisfaction and posttest scores for each condition).

Table 1. Students' satisfaction and individual posttest scores by collaboration condition.

	All Students *mean* (SD)	
	Traditional (N = 90)	***Hybrid*** (N = 110)
Learning Outcomes	5.12 (1.97)	4.77 (1.85)
Satisfaction	13.36 (1.74)	12.62 (2.26)

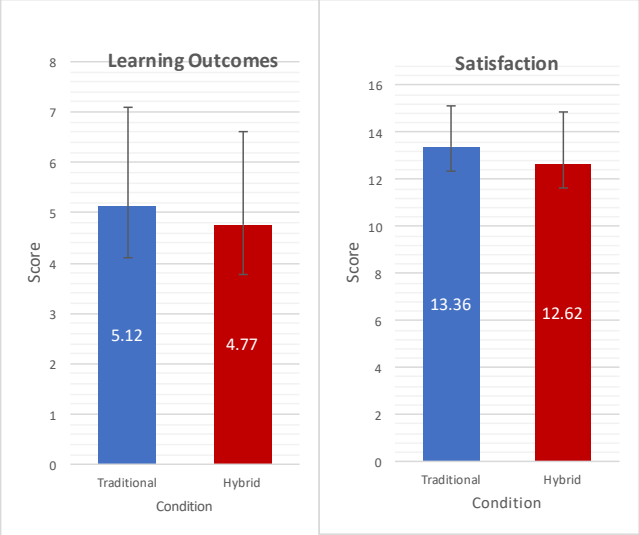


Figure 3. Students' individual posttest scores and satisfaction scores by collaboration condition.

5 DISCUSSION

The goal of the study reported here has been to explore whether giving students more control of their code was a beneficial hybrid approach to pair programming. Although the learning gains were not statistically different, student satisfaction in the traditional pair programming condition was better, and students reported engaging more actively in conversation with their partners.

One phenomenon that is known to be influential in collaboration, and which likely differed markedly when students

[2] In these comparisons as well as others reported later, the data were not normally distributed but skewed, and therefore we utilized the Mann-Whitney U test to compare sample means.

were seated at different computers rather than the same one, is *shared goals*. Previous studies have shown that success of collaborative activities depends on having a shared intention [23], and constructing a shared conception of a problem [21]. However, students in the *Hybrid* condition may have shared goals to a lesser extent, as they implemented their solutions individually. Without a shared goal, students may have had less motivation to help one another, and certainly less of a shared mental model for doing so. To provide further insight into students' experience in pair programming in this class, we extracted excerpts from the reflection essays that students submitted at the end of the semester. These essays reflect on the entire semester, including all weeks of pair programming in the labs, not only on the week in which the *Hybrid* vs. *Traditional* condition was tested.

One student pointed out some important strengths of working collaboratively, including the importance of a shared goal:

> *"The practice of pair programming was an excellent way to go about completing the labs. This allowed us to interact with our classmates and help each other learn… It is nice to have a class in which we get to practice working together, building upon the strengths of each person, to achieve a common goal."*

Another student expressed that the immediate assistance from a partner helps the problem-solving process go more smoothly. When working at two different computers, the immediacy of help would be reduced since the partners were not continually attending to the same version of code.

> *"The pair programming, to me, was very helpful. Frequently, my partner and I would exchange information the opposite person wasn't familiar with. To elaborate, if I didn't know something, my partner would let me know how to do it the right way, and vise-versa."*

Further positive feedback on pair programming includes the benefit of working together even if they did not know their partners beforehand. Many students' reflections centered around the benefits of pair programming in terms of receiving help from other people:

> *"…pair programming assisted me greatly in figuring out how to solve problems… I can honestly say I looked forward to lab every week, even if we were required to have a random partner. It was extremely instructive…"*
>
> *"… with a partner, putting our mind together makes things a lot less stressful and I feel like I learn better."*
>
> *"Pair programming also helped me find people in the class that I could talk to and get help from."*

Prior studies suggest that asking questions and indicating confusion are beneficial for student learning [22]. In the *Hybrid* condition, even though students were allowed to collaborate on the problem solution, seeking help and asking questions would have required slightly more overhead work to transition the partner's attention onto a student's own code since the partner had his or her own separate code. Some students may not have asked for help as readily [4], and we know that the context can affect question-asking behaviors [8]. Pair programming at the same computer, creating shared code, naturally open doors for collaboration and may facilitate question asking. Reflections from students support the idea that having a partner encouraged students to ask for help and have less frustration:

> *"Pair programming helped me get through the lab and learn from my partner instead of just sitting there frustrated and staring at my code, too afraid to ask for help."*
>
> *"I would question them* (partners) *as to how they thought of their code, and why it works. Hearing their explanation helped me implement those techniques into future programming assignments that I would later work on my own."*

On the other hand, computer programming has long been taught and seen as an individual activity in many contexts [8, 21], and indeed, some students express a preference for working individually. These students also sometimes believe that working alone is more beneficial for them compared to pair programming.

> *"The part that I did not enjoy from lab was that it was pair programming. I work best when it comes to programming by working on my own, messing with different options and asking for help when I get stuck."*
>
> *"My learning during lab was optimized when my partner and I both coded our separate programs on our own laptops, while still asking each other for help if either of us got stuck. "*
>
> *"I think a much better solution to the process would be for everybody to write their own code, but to have a partner to whom you could ask questions to. "*

It is important to take this student feedback into account regarding negative perceptions of pair programming. While the evidence is compelling that working together on a shared version of the code holds benefit, we need to consider approaches that mitigate the limitations of pair programming—both practical and perceived—in order to improve the student experience.

6 LIMITATIONS AND THREATS TO VALIDITY

This study was conducted in the context of a CS1 course, and was one of a series of exploratory studies conducted that semester. A more rigorously controlled experiment in the future will shed more light on the phenomena surrounding collaboration and autonomy in pair programming. An example of an experimental control that was missing from this study is that students had already been pair programming for three weeks; thus, students in the hybrid condition experienced a change from "business as usual" while students in the traditional pair programming condition did not. Providing students equal exposure to these approaches over a longer period of time will support deeper investigation of the research question.

This study was not conducted in a controlled environment, but an actual lab for a CS1 class. Along with that rich context came numerous complicating factors that had to be dealt with in the study. For example, in this naturalistic classroom environment, some students interacted with classmates from other dyads, and also sought help from the teaching assistant. We did not intervene in these cases. An additional limitation concerns diversity. The study was conducted at a large research university whose student population is predominantly white and whose computer science students are overwhelmingly male. The demographics of the students who participated in this study

were representative of the class (in fact, about 70% of students in the class of more than 350 consented to participate in data collection). It is crucial for further work to examine pair programming dialogues in other contexts and with more diverse student populations. Finally, student preference regarding whether to work in pairs or not is likely influential in their success, but we did not explicitly measure this preference.

7 CONCLUSION AND FUTURE WORK

The SIGCSE community has long studied how best to support students in collaborative problem solving. In this study, we examined the learning outcomes and satisfaction difference between *Traditional* and *Hybrid* pair programming, in which students either shared a computer with a partner and created a shared solution, or worked on two different computers but helped each other when needed. The findings indicate that when students create a shared solution during pair programming activities, they show comparable learning gains and significantly more satisfaction. Moreover, when students are responsible for their own code, they tend to talk with each other less. Qualitative examination of the students' own reflections suggests that some students do believe working individually will enable them to learn more, but this study provides even more evidence that pair programming is a highly effective learning method for CS.

There are many promising areas for future work. This paper has reported on results of one modification of pair programming: when students take responsibility for their own code individually. There are many other scaffolding approaches that should be studied, including peer teaching, code dividing, error hunting, and completing each other's partially complete code. Finally, we have utilized post-test scores, self-reported satisfaction from a post survey, and students' reflection on pair programming to extract quotes that yield insight into the quantitative finding; however, deeper process-oriented analyses through recorded pair programming videos can reveal many other important factors that affect the learning outcome and satisfaction of students. Gathering these results and moving towards developing adaptive techniques that consider these findings is a promising research direction for computer science education.

8 ACKNOWLEDGMENTS

The authors wish to thank the members of the LearnDialogue group at the University of Florida for their helpful input. This work is supported in part by Google through a CS Capacity Research Award and by the National Science Foundation through grant CNS-1622438. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the authors, and do not necessarily represent the official views, opinions, or policy of the National Science Foundation.

REFERENCES

[1] Jens Bennedsen and Michael E. Caspersen. 2007. Failure rates in introductory programming. *ACM SIGCSE Bulletin* 39, 2: 32.

[2] Aaron E. Black and Edward L. Deci. 2000. The effects of instructors' autonomy support and students' autonomous motivation on learning organic chemistry: A self-determination theory perspective. *Science education*: 740-756.

[3] G. Braught, T. Wahls, and L. M. Eby. 2011. The Case for Pair Programming in the Computer Science Classroom. *ACM Transactions on Computing Education* 11, 1: 1–21.

[4] John T. Bruer. 1993. *Schools for thought: A science of learning in the classroom.* MIT Press.

[5] M. Celepkolu, J. B. Wiggins, K. E. Boyer, and K. Mcmullen. 2017. Think First: Fostering Substantive Contributions in Collaborative Problem-Solving Dialogues. In *Proceedings of the International Conference on Computer-Supported Collaborative Learning*: 295-302.

[6] Alistair Cockburn and Laurie Williams. 2000. The Costs and Benefits of Pair Programming. *Extreme Programming and Flexible Processes in Software Engineering XP2000*: 223-247.

[7] Gerald Giraud, Craig Enders. 2000. The Effects of Repeated Cooperative Testing in an Introductory Statistics Course.

[8] M. Guzdial, P. Ludovice, M. Realff, T. Morley, and K. Carroll. 2002. When collaboration doesn't work. *Proceedings of the International Conference of the Learning Sciences*: 125–130.

[9] R. G. Hausmann, M. T. Chi, and M. Roy. 2004. Learning from collaborative problem solving: An analysis of three hypothesized Mechanisms. *Proceedings of the Cognitive Science Society* 26, 26.

[10] S. D. Johnson and S. P. Chung. 1999. The Effect of Thinking Aloud Pair Problem Solving (TAPPS) on the Troubleshooting Ability of Aviation Technician Students. *Journal of Industrial Teacher Education* 37, 1.

[11] Michael E. Lantz. 2010. The use of "Clickers" in the classroom: Teaching innovation or merely an amusing novelty? *Computers in Human Behavior* 26, 4: 556–561.

[12] Lin Lin. 2015. Exploring Collaborative Learning: Theoretical and Conceptual Perspectives. In *Investigating Chinese HE EFL Classrooms*. Springer Berlin Heidelberg, 11–28.

[13] Charlie Mcdowell, Brian Hanks, and Linda Werner. 2003. Experimenting with Pair Programming in the Classroom. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, 60–64.

[14] Charlie McDowell, Linda Werner, Heather E. Bullock, and Julian Fernald. 2006. Pair programming improves student retention, confidence, and program quality. *Communications of the ACM* 49, 8: 90–95.

[15] Charlie McDowell, Linda Werner, H eather E. Bullock, and Julian. Fernald. 2002. The effects of pair-programming on performance in an introductory programming course. *ACM SIGCSE Bulletin* 34, 1: 38.

[16] Charlie McDowell, Linda Werner, Heather E. Bullock, and Julian Fernald. 2002. The effects of pair-programming on performance in an introductory programming course. *ACM SIGCSE Bulletin* 34, 1: 38.

[17] J. R. Mergendoller, N. L. Maxwell, and Y. Bellisimo. 2000. Comparing Problem-Based Learning and Traditional Instruction in High School Economics. *The Journal of Educational Research* 93, 6: 374–382.

[18] N. Nagappan, L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller, and S. Balik. 2003. Improving the CS1 experience with pair programming. *ACM SIGCSE Bulletin* 35, 1: 359.

[19] J. K. Olsen, Nikol Rummel, and Vincent Aleven. 2017. Learning Alone or Together? A Combination Can Be Best! In *Proceedings of the International Conference on Computer-Supported Collaborative Learning*, 95–102.

[20] Roger T, and David W. Johnson. 1994. An Overview of Cooperative Learning. *Creativity and collaborative learning.*

[21] Jeremy Roschelle and Stephanie D. Teasley. 1995. The Construction of Shared Knowledge in Collaborative Problem Solving. In *Computer Supported Collaborative Learning*. 69–97.

[22] Richard A. Schmuck and Patricia A. Schmuck. 1975. *Group Processes in the Classroom.*

[23] Michael Schrage. 1995. *No More Teams!: Mastering the Dynamics of Creative Collaboration.*

[24] Lev S. Vygotsky. 1978. Interaction between learning and development. In *Mind in Society*. 78–91.

[25] Christopher Watson and Frederick W.B. Li. 2014. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*, 39–44.

[26] Laurie Williams, Robert R. Kessler, and Ward Cunningham. 2000. Strengthening the case for pair programming. *IEEE.*

[27] Laurie Williams, Charlie McDowell, Nachiappan Nagappan, Julian Fernald, and Linda Werner. 2003. Building pair programming knowledge through a family of experiments. In *Proceedings - 2003 International Symposium on Empirical Software Engineering, ISESE 2003*, 143–152.