IMPROVING STUDENT PERSISTENCE IN COMPUTER PROGRAMMING COURSES WITH PAIR-TESTING

Rajendran Swamidurai
Alabama State University
Department of Mathematics and Computer Science
rswamidurai@alasu.edu

Abstract

In recent years, enrollments in computing majors have started climbing after a precipitous decline that lasted several years. Though enrollments have started climbing, attrition in computing majors is still a problem. The attrition rates in computing courses are as high as 30% nationally and most of the attrition happens during freshman and sophomore years. Indeed, if left unaddressed, high attrition rates could easily negate the about said enrollment gains. Motivation and involvement are vital tools to address the above said attrition problem and retain students in computing majors. Collaborative learning tools such as pair programming are widely used to address the retention issue and motivate the student programmers to complete their jobs. Since pair programming is not beneficial and/or not possible to practice today's academic environment, we have used pair-testing as a collaborative learning tool to improve the students persistence in computer programming courses. It is expected that pair-testing, as a collaborative learning tool, will retain the benefits of pair programming while at the same time downplaying its drawbacks. In this paper, we describe the design and implementation of a pair-testing model for a computer programming course (Software Engineering I), and then present the results of an evaluation of the model.

1. Introduction

Employment in the computing profession is expected to increase by 32 percent over the next decade, making it one of the fastest-growing professions in the country [1]. In recent years, enrollments in computing majors have started climbing after a precipitous decline that lasted several years. Though enrollments have started climbing, attrition (the dropout and failure rates) in computing majors is still a problem. Various reports indicate that the attrition rates in computing courses are as high as 30% nationally [2, 3] and most of the attrition happens during freshman and sophomore years [2]. At Alabama State University (ASU), we have seen attrition rates much higher than the national trends. Our STEM (science, technology, engineering and mathematics) sophomore classes continued to have an average failure rate of about 30% to 35% and the failure rate of various computer programming courses are very high compared to non-programming courses for the computer science sophomores. Indeed, if left unaddressed, high attrition rates could easily negate the about said enrollment gains.

Motivation and involvement are vital tools to address the above said attrition problem and retain students in computing majors [2, 4]. Studies [5–8] show that collaborative student programmers' are not as easily frustrated and are motivated to finish their assignment. That is, these studies indicate that collaborative programming improves the student programmers' self-efficacy, one's

belief in his or her ability to succeed in a particular situation [9], and motivates them to complete their assignments. They also reveal that collaboration inside the classroom helps the students to produce better programming assignments and to complete the projects in shorter time.

Pair programming and peer-review are the two widely used collaborative pedagogies to teach computing courses. Peer-review is more adversarial process than collaborative process; but pair programming provides more social environment than competitive environment for students [10]. Though advocates of pair programming claim that pair programming provides the following educational benefits such as active learning and improved retention, program quality, and confidence in their work [6,7,11,12] it has many drawbacks [13]: 1) it is very difficult to execute or practice in today's academic environment. Because, the basic requirement of pair programming is that it requires that the two students be at the same place throughout their job. This is not realistic due to the nature of the current academic environment, in which students are matrixed concurrently to a number of courses and most of the computing courses are not taught in traditional environment; 2) the empirical evidence of the benefits of pair programming is mixed [6, 14–22]; and 3) pair programming is more effective if both members of the pair are novices to the task at hand. Novice-expert and expert-expert pairs have not been demonstrated to be effective [18, 23, 24]. Though one may argue that most of the computer science undergraduate students are novice, it may be true only in the traditional academic settings where the courses are offered in face-to face classroom; but today many academic institutions moving towards or believes that non-conventional teaching such as e-learning will be the future way to teach computing courses.

Since pair programming is not beneficial or not possible to practice in current academic environment, it is clear that we need a collaborative pedagogy that retains the benefits of pair programming at the same time minimize the pair-up time between the students. For this purpose, we tuned the pair-testing as a collaborative learning tool to address retention and improve performance in computer programming courses. Anecdotal evidences and our past experiences show that collaboration at the debugging (or testing) time helps the novice programmers to understand (and learn) the structure and operation of computer programs. Cañas et al. [25] study strengthens our claim; they showed that the use of trace facility allowed the students to learn the semantics of the program. That is, the tracing activity (such as debugging or testing) helping students to learn programming concepts (or helps the novice programmers to follow the flow of operations and actions of the program).

2. Pair-Testing

Pair-testing is a software testing technique in which pair of people, a driver and an observer, tests the software application sitting together in a single computer. The driver is responsible for testing the current module at hand and the observer is responsible for inspecting the testing process. Pair testing is an exploratory process, which helps both the team member to find out how the software actually works.

As in pair programming [26], the following are assumed in pair-testing as well:

- Paring is dynamic and the people have to pair with different people in the morning and evening sessions [26].
- A programmer can pair with anyone in the development team [26].
- The developers have to exchange their partners every day and some developers will exchange their partners more often depending upon the situation [27].
- Every member of the development team should try to partner with every member in the team and every programmer has to work in at least in two different pairs [28].
- A single developer owns the task at hand. The developer responsible for the task may partner with one person for one aspect of the task and someone else for another aspect of the task [27].

3. Implementing Pair-Testing in Software Engineering I

The Software Engineering I course at Alabama State University is typical in content, focusing on software lifecycle and associated tools and techniques. Software Engineering I is a 3 credit hour course and meets for 2 clock hours of lecture per week and 2 clock hours of lab per week. All students meet together for the same lecture, but they meet separately in small lab sessions. The course is normally offered every fall semester and each class has no more than 30 students.
A pilot model of the collaborative-adversarial pair learning was first implemented in the summer 2012 semester for object oriented programming course. There were four programming assignments in the semester and we asked the students to complete two of them using pair-testing. The pilot semester helped us to try things out and made many adjustments based on our initial assessments.

In fall 2013, we incorporated pair-testing in Software Engineering I course in a full-fledged manner. First, a lecture was arranged to explain the concepts of pair-testing approach. Then, a pair-testing practice session was conducted to enable the participants to what is expected of them. Twelve computer science seniors from Software Engineering I course participated in the study. The subjects were asked to solve two programming tasks of varying degrees of complexity using C++. The control experiments consisted of three phases: design, coding, system/acceptance test. The subjects were asked to complete all the programming exercises until they pass the acceptance test. If the program did not pass the test, the subjects need to fix the errors and repeat the test until the program passes the test.

The subjects were randomly divided in to two groups. For the first programming task, the group one were asked to do pair-testing (PT) and the group two were asked to do individual programming (IP). For the second programming task, we asked the first group to go for individual programming mode and the second group to pair-testing mode.

To study the impact of pair testing, we compared the pair-testing (PT) group with the individual programming (IP) group. The subjects in the PT group were asked to complete the design and coding alone and complete the test with a partner. The subjects in the IP group were asked to complete the entire programming task (design, code and test) alone.

To study the cost of overall software development, we compared the total development time, measured in minutes, of all the phases. The IP and PT total software development costs were calculated as per the following formulas:

- $Cost_{IP} = Time_{Design} + Time_{Coding} + Time_{Test}$
- $Cost_{PT} = Time_{Design} + Time_{Coding} + 2*(Time_{Test})$

The software delivery time is the sum of design, code and test times for both the experimental groups.

We tested the following hypothesis,

- **$H0_1$ ($Motivation_{PTvs.IP}$):** The number of programming assignments not completed by the PT group is equal or higher than IP group in average.
- **Ha_1 ($Motivation_{PTvs.IP}$):** The number of programming assignments not completed by the PT group is less than IP group in average.
- **$H0_2$ ($DeliveryTime_{PTvs.IP}$):** The software development duration or delivery time of PT is equal or higher than IP in average.
- **Ha_2 ($DeliveryTime_{PTvs.IP}$):** The software development duration or delivery time of PT is less than IP in average.
- **$H0_3$ ($COST_{PTvs.IP}$):** The overall software development cost of PT is equal or higher than IP in average.
- **Ha_3 ($COST_{PTvs.IP}$):** The overall software development cost of PT is less than IP in average.

4. Results

Table 1 below summarizes the control experiment data.

Table 1. Software Development Time (Measured in Minutes).

Pair-Testing Group		Individual Group	
Problem1	**Problem2**	**Problem1**	**Problem2**
27	21	20	Incomplete
29	16	34	16
21	38	43	Incomplete
23	30	65	Incomplete
32	21	67	Incomplete

4.1. Pair-Motivation

At the beginning of the experiment, we anticipated that both the control (individual) and experimental (pair-testing) group will complete the given two programming problems; but only the pair-testing group completed both the programming assignments. The total number of

programming assignments completed by the pair-testing groups and the individual groups are shown in Table 1. The total number of programming problems completed by the pair-testing groups was 10, whereas, the total number of programming problems completed by the individual groups was 6. Table 1 indicates that the number of programming problems completed by individual groups is less than the number of programming problems completed by pair-testing group. Hence, we accept the alternative hypothesis (Ha1) that the number of programming assignments completed by the pair-testing groups is less than the number of programming assignments completed by the individuals. Even though we did not enforce any time limit to complete the programming assignments, 4 out of 5 individuals completed only one programming problem. This clearly indicates that the extra brain (and additional pair of eyes) in the pair-testing phase helped and motivated the students to complete the programming assignments.

4.2. Software Development (or Delivery) Time

The pair-testing groups took 21 minutes on average to solve the problem; whereas the individual programming groups took 36 minutes (42% more than pair-testing groups) in average to solve the programming problem. The t-test result for hypothesis 2 is shown in figure 1 and the box-plot is shown in figure 2. For a 95% confidence interval the P value is 0.1829. Since $P >= 0.05$, we do not have sufficient statistical evidence to reject $H0_2$ in favor of Ha_2. Therefore, we conclude that the overall software delivery time of pair-testing group is not less than individual programming in average.

The TTEST Procedure
Variable: time

group	N	Mean	Std Dev	Std Err	Minimum	Maximum
IP	5	35.6000	19.6799	8.8011	16.0000	65.0000
PT	5	21.2000	6.6106	2.9563	14.0000	29.0000
Diff (1-2)		14.4000	14.6799	9.2844		

Method	Variances	DF	t Value	Pr > |t|
Pooled	Equal	8	1.55	0.1595
Satterthwaite	Unequal	4.8913	1.55	0.1829

Fig. 1. t-Test Results for Pair Testing vs. Individuals Duration

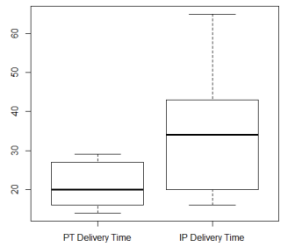

Fig. 2. Box plot for Pair Testing vs. Individuals Duration

4.3. Software Development Cost

In pair-testing experiment, the pair-testing groups took 26 minutes on average to solve the problem; whereas the individual programming groups took 36 minutes (28% more than pair-testing groups) in average to solve the programming problem. The t-test result for hypothesis 3 is shown in figure 3 and the box-plot is shown in figure 4. For a 95% confidence interval the P value is 0.3606. Since $P >= 0.05$, we do not have sufficient statistical evidence to reject $H0_3$ in favor of Ha_3. Therefore, we conclude that the overall software development cost of pair-testing is not less than individual programming in average.

The TTEST Procedure
Variable: time

group	N	Mean	Std Dev	Std Err	Minimum	Maximum
IP	5	35.6000	19.6799	8.8011	16.0000	65.0000
PT	5	26.4000	4.4497	1.9900	21.0000	32.0000
Diff (1-2)		9.2000	14.2671	9.0233		

Method	Variances	DF	t Value	Pr > |t|
Pooled	Equal	8	1.02	0.3378
Satterthwaite	Unequal	4.4079	1.02	0.3606

Fig. 3. t-Test Results for Pair Testing vs. Individuals Cost

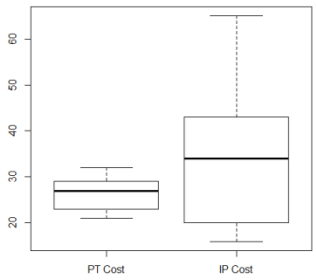

Fig. 4. Box plot for Pair Testing vs. Individuals Cost

5. Summary and Conclusion

We have adapted the pair-testing based learning model for a typical computer programming course and evaluated its effectiveness. In our model of pair-testing based learning, students get experience in critically analyzing each other's solutions in pair-testing, and reflecting on and learning from these collaborative sessions over the course of a semester. A formal evaluation in which the pair-testing model was compared to a traditional testing revealed that the pair-testing 1) motivated the students to complete the programming assignments, and 2) offered significant benefits to students in terms of both course content mastery and programming achievement. We are quite optimistic that the pair-testing based learning model will prove to be an effective approach to address the attrition issue found in computer programming courses and improve student persistence in computing majors.

The empirical evidence also shows that pairing on the test phase is beneficial and pair-testing helps the students to produce equal quality programs in faster time (42% less overall software development time than traditional individual programming) with cheaper cost (28% less overall software development cost than traditional individual programming).

6. References

1. U.S. Bureau of Labor Statistics, *2010-2011 Occupational Handbook Published*, www.bls.org.
2. Beaubouef, T., & Mason, J., *Why the high attrition rate for computer science students: Some thoughts and observations*, ACM SIGCSE Bulletin, 2005, *37*(2), 103-106.
3. Guzdial, M. & Soloway, E., *Log on education: teaching the Nintendo generation to program*, Communications of the ACM, 2002, 45(4), 17-21.
4. Guzdial, M. , *Introduction to computing and programming in Python: A multimedia approach*, Upper Saddle River, NJ, Prentice Hall, 2004.
5. Bevan, J., Werner, L., McDowell, C., *Guidelines for the use of pair programming in a freshman programming class*, Proceedings of the 15th Conference on Software Engineering Education and Training, 100, 2002.
6. C. McDowell, L. Werner, H. Bullock, and J. Fernald, *The effects of pair-programming on performance in an introductory programming course*, in Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education, pp. 38–42, Cincinnati, Ky, USA, March 2002.
7. Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C., Balik, S., *Improving the CS1 experience with pair programming*, Proceedings of the 34th SIGCSE technical symposium on Computer science education, 359-362, 2003.
8. Williams, L., Yang, K., Wiebe, E., Ferzli, M., Miller, C., *Pair programming in an introductory computer science course: Initial results and recommendations,* presented at the OOPSLA Educator's Symposium, 2002.
9. Bandura, A., *Social Foundations of Thought and Action*. Prentice Hall, Englewood Cliffs, NJ, 1986.
10. Simons, K., & Klein, J., *The Impact of Scaffolding and Student Achievement Levels in a Problem-Based Learning Environment*. Instructional Science, 2007, 35, 41-72.
11. McDowell, C., Werner, L., Bullock, H.E. and Fernald, J., *Pair programming improves student retention, confidence, and program quality*, Communications of the ACM **49**(8), 90-95, 2006.
12. Mendes, E., Al-Fakhri, L. and Luxton-Reilly, A., *A replicated experiment of pair-programming in a 2nd year software development and design computer science course*, *ACM SIGCSE Bulletin* **38**(3): 108-112, 2006.
13. Rajendran Swamidurai and David A. Umphress, *Collaborative-Adversarial Pair Programming*, ISRN Software Engineering, vol. 2012, Article ID 516184, 11 pages, 2012. doi:10.5402/2012/516184.
14. J. D. Wilson, N. Hoskin, and J. T. Nosek, *The benefits of collaboration for student programmers*, in Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education, pp. 160–164, February 1993.
15. J. T. Nosek, *The case for collaborative programming,* Communications of the ACM, vol. 41, no. 3, pp. 105–108, 1998.
16. L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries, *Strengthening the case for pair programming*, IEEE Software, vol. 17, no. 4, pp. 19–25, 2000.
17. S. Xu and V. Rajlich, *Empirical validation of test-driven pair programming in game development*, in Proceedings of the 5th IEEE/ACIS International Conference on Computer and

Information Science (ICIS '06). In conjunction with 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (COMSAR '06), pp. 500–505, July 2006.
18. E. Arisholm, H. Gallis, T. Dybå, and D. I. K. Sjøberg, *Evaluating pair programming with respect to system complexity and programmer expertise*, IEEE Transactions on Software Engineering, vol. 33, no. 2, pp. 65–86, 2007.
19. J. Nawrocki and A. Wojciechowski, *Experimental Evaluation of pair programming*, in Proceedings of the European Software Control and Metrics Conference (ESCOM '01), pp. 269–276, ESCOM Press.
20. J. Vanhanen and C. Lassenius, *Effects of pair programming at the development team level: an experiment*, in Proceedings of the International Symposium on Empirical Software Engineering (ISESE '05), pp. 336–345, November 2005.
21. M. Rostaher and M. Hericko, *Tracking test first programming—an experiment, XP/Agile Universe*, LNCS, vol. 2418, pp. 174–184, 2002.
22. H. Hulkko and P. Abrahamsson, *A multiple case study on the impact of pair programming on product quality*, in Proceedings of the 27th International Conference on Software Engineering (ICSE '05), pp. 495–504, St. Louis, Mo, USA, May 2005.
23. D. Wells and T. Buckley, *The VCAPS project: an example of transitioning to XP*, in Extreme Programming Examined, chapter 23, pp. 399–421, Addison-Wesley.
24. K. M. Lui and K. C. C. Chan, *Pair programming productivity: Novice-novice vs. expert-expert*, International Journal of Human Computer Studies, vol. 64, no. 9, pp. 915–925, 2006.
25. Cañas, J.J., Bajo, M.T. & Gonzalvo, P., *Mental models and computer programming*, International Journal of Human-Computer Studies, 40(5), 795-811, 1994.
26. K. Beck, *Extreme Programming Explained: An Embrace Change*, Addison-Wesley, 2000.
27. William C. Wake, *Extreme Programming Explored*, Addison-Wesley, 2002.
28. Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2003.