A Novel Approach for Collaborative Pair Programming

Sanjay Goel
Jaypee Institute of Information Technology, Noida, India
sanjay.goel@jiit.ac.in

Vanshi Kathuria
Accenture Technology Solutions, Hartford, CT, USA
vanshi.kathuria@accenture.com

Executive Summary

The majority of an engineer's time in the software industry is spent working with other programmers. Agile methods of software development like eXtreme Programming strongly rely upon practices like daily meetings and pair programming. Hence, the need to learn the skill of working collaboratively is of primary importance for software developers. During computing education, this may be particularly important for the stronger students as they may be the ones who least desire to work with other programmers. Further, programmers need to develop the ability to comprehend the programs developed by others and, also, to write programs that can be easily comprehended by other programmers**.** Increasing dependence on large amounts of Free and Open Source Software (FOSS) makes this even more crucial. Until a decade ago, one weakness in the typical undergraduate experience was the failure to train students to work with other programmers – in fact it was often considered a form of cheating. Over time, researchers have the need to find a way to allow students to work together within clearly defined boundaries that would be an acceptable practice.

A traditional form of pair programming based on the driver-navigator model has been successful in many introductory computer science courses. Its success is noticeable in better performance in computer science assignments, increased team work in and outside class, enhanced learning, and decreased frustration (Cliburn, 2003; Domino, Collins, & Hevner, 2007; McDowell, Werner, Bullock, & Fernald, 2002; Nagappan et al., 2003;Sfetsos, Stamelos, Angelis, & Deligiannis, 2009; Thomas, Ratcliffe, & Robertson, 2003; VanDeGrift, 2004; Williams & Kessler, 2003; Williams, Yang, Wiebe, Ferzli, & Miller, 2002). However, pair programming has its weaknesses too (Bevan, Werner, & McDowell, 2002, Cliburn, 2003; VanDeGrift, 2004). Most of the experiments have the shortcoming of not having been able to create a convincing need for collaboration. Such a situation results in students sitting together, sharing a machine, but not actually collaborating with each other

The objective of this paper is to propose a new framework for implementing pair programming in the classroom. Our model of collaborative pair programming is based on Dillenbourg's set of four conditions to set up an active collaborative context. We have transformed the traditional form of pair programming to suit this framework. Students in a pair first work independently and then work on a combined task that brings together their individual tasks. The framework has been experimented within a real classroom

Editor: Linda Knight

environment for the course 'Introduction to Computer Programming' offered to first year engineering students. All the inexperienced programmers were allowed to form pairs, while all the experienced students worked as solo programmers. This was done primarily to compare the performance of experienced versus inexperienced programmers at the end of the course. This comparison helped identify if such an implementation of collaborative pair programming proved advantageous for students.

The experiment was well monitored and regulated by four instructors and teaching assistants at any given time. Observations throughout the five months of the course and results from various examinations and feedback mechanisms showed that such a framework for collaborative pair programming enhanced problem solving skills, improved quality of work, and increased trust and teamwork. Final examination results for inexperienced programmers showed a marked increase in their performance when compared to results of experienced programmers. In conclusion, the new framework proved successful by helping inexperienced programmers perform at par with their experienced peers.

Introduction

Engineering education accreditation agencies of US, UK, Australia, Japan, and Singapore have transformed their accreditation process from the traditional resource-based approach to outcome-based approach. Competencies such as ability to apply knowledge, design skills, problem solving skills, technical competence, ability to work in multidisciplinary teams, communication skills, sensitivity towards global, societal, and environmental issues, sensitivity towards ethical and professional issues, and readiness for lifelong learning have been identified as necessary (Goel 2006a). Goel (2006b) has classified core engineering and general professional competencies with reference to the requirements of Indian IT industry. Problem solving ability has been rated as the pivotal skill. Nine critical competencies include a mix of engineering and general professional skills. These are analysis/methodological skills, basic engineering proficiency, development know-how, teamwork skills, English language skills, presentation skills, practical engineering experience, leadership skills, and communication skills. Consequently, it is mandatory that all courses in engineering curricula should consciously and directly contribute towards the development of these competencies.

Today, organizations have offices in multiple locations with teams interacting across various geographical sites, sometimes even in different time zones. Big software projects are divided across locations, and people across different sites are expected to combine their individual works to complete the project work. Further, programmers need to develop the ability to comprehend the programs developed by others and also to write programs that can be easily comprehended by other programmers**.** Increasing dependence on large amounts of Free and Open Source Software (FOSS) makes it even more crucial. However, a serious shortcoming of the typical undergraduate engineering education process is neglecting to train students to work with other programmers. Usually in a four-year engineering course, collaborative effort and teamwork is done only in the later years, i.e., in the third and fourth year. Even then the collaboration is driven more by division of labor due to size rather than complexity. Many a times, the assignment or project is designed after forming the groups in the classroom, when the teacher knows the strength of a particular group. Such approaches do not necessarily develop teamwork skills. Researchers have felt the need for a way to facilitate students to work together with clearly defined boundaries (Cliburn, 2003).

Collaborative learning focuses on the role of peer work for educational success. Vygotsky in his seminal social development theory proposed that social interaction plays a fundamental role in the process of cognitive development (Goldfarb, 2001). One of key assumptions of Brunner's model of constructivist learning is a social enterprise (Driscoll, 2004). Pask's Conversation theory

is founded on the idea that learning occurs through conversations with instructors or peers (Kutay, 2005). As per the social learning theory of Bandura, gaining insights into others' practices can be a valuable experience (Ladyshewsky & Ryan, 2002). Incorporating social learning theory into their experiment, these authors report that peer coaching provided the learners the benefits of enhanced knowledge, cognition, and metacognition. Lave and Wenger (1991) proposed situated learning theory on the central idea that learning involves a process of engagement in a 'community of practice.' From the perspective of this theory, learning is not seen as the acquisition of knowledge by individuals, but as a process of social participation (Smith, 2007). Dillenbourg (1999) elaborated on collaborative learning as follows:

> *"... a situation in which particular forms of interaction among people are expected to occur, which would trigger learning mechanisms, but there is no guarantee that the expected interactions will actually occur. Hence, a general concern is to develop ways to increase the probability that some types of interaction occur."*

Learning in a collaborative environment is a process that could be subject of two different perspectives: individual effort and social sharing of knowledge (Dillenbourg, 1999; Lipponen, 2002). However, sharing of knowledge is useful only if students are engaged in collaborative activities**.** Engagement in collaborative activities causes individuals to master something that they could not do before the collaboration (Lipponen, 2002).

In our study, we offered structured opportunities for collaboration to students of 'Introduction to Computer Programming' offered to first year students. While designing the lab exercises for our course we had two main purposes in mind. Firstly, the task should not be solved collaboratively just because it is complex, but it should be designed such that it demands collaboration. Secondly, we wanted to introduce teamwork in the early stages of engineering instead of the later semesters, as this not only helps in developing better technical skills but also helps in the development of numerous soft skills.

Literature Review

Group work has a very wide range of possible forms, ranging from simple task division between two members to intense community collaborations. At the simplest level, it may be in the form of coordination between members for handling simple situations with clear task boundaries requiring minimum intra-group communication. Alternatively, it may take the form of cooperation for handling complicated situations with well defined, but marginally overlapping subtask boundaries, and mild intra-group communication. Group work in its most sophisticated form requires collaboration for handling complex situations with evolving subtask boundaries and intense intra-group communication. The group size, distance, and professional as well as cultural diversity are other important aspects that bring further variations of these three forms of group work. Interestingly, software development involves all these forms of group work. However, sometimes the word collaboration is used to mean all these forms of group activities.

The nature of group work amongst software engineers is not limited to process-centred coordination. Whitehead (2007) observes that software engineering projects require many software engineers to collaboratively create a large number of artefacts incorporating code, requirement specifications, architecture descriptions, design models, test plans, etc. The distinctive feature of agile methods and culture is that they view and approach software development as a social activity. Shared development environment, constant immersion and engagement with the team, constant feedback, reviews, holistic awareness, continuous testing and integration, community of practitioners are some of the hallmarks of all agile methods (Whitworth & Biddle, 2007). Further, agile method like eXtreme Programming (XP) uses the practice of *pairing* not just for code development, but also for design, refactoring, as well as testing (Sharp & Robinson, 2005). Advocates

of pair programming believe that due to continuous discussion and review, a pair is actually more productive than two separate developers. It significantly reduces defect rate with which consequent implications for reliability, cost, and maintenance.

Brusilovsky, Kouchnirenko, Miller, and Tomek (1994), in a review of approaches and tools for teaching programming, noted that programming causes cognitive overload, and for this reason should be taught in small subsets. It is very essential to engage students in learning activities concerning simple authentic problems that are close to their experience, and show the usefulness of the programming process beyond the specific course to arouse their curiosity make them curious (Gogoulou, Gouli, Grigoriadou, & Samarakou, 2003).

A novel and well-appreciated approach in teaching programming has been identified in collaborative programming. A lot of research has been done over the past few years, and it has been seen that a pair or a group working together in solving a programming exercise minimizes the cognitive load (Williams, Wiebe, Yang, Ferzli, & Miller, 2003). Preston (2005) summarizes that collaborative learning research has already established two things: (1) the effectiveness of having students work together, and (2) the critical attributes common to successful collaborative learning approaches. One important attribute in the design of an exercise is that the solution should require co-authoring. Collaborative programming exposes code to constant review. The group methodology promises to facilitate collaboration, promote mentorship, and also enhance collective ownership of code.

The most common and widely used form of collaborative programming is pair programming. Pair programming is a situation in which two programmers work side-by-side, designing and coding, while working on the same algorithm. As Chaparro (2005) paraphrases (Cockburn & Williams, 2001; Williams & Kessler, 2003), a relevant aspect of pair programming is that it transforms what used to be an individual activity into a cooperative effort. Remote pair programming, or virtual pair programming, is another variation of traditional pair programming, where the two programmers are in different locations, working via a real-time editor or shared desktop. Typically there are two roles in pair programming: the driver who controls both the computer keyboard and the mouse, and the navigator who examines the driver's work, offering advice, suggestions and corrections to both design and code. Pair programming improves the quality of the software design, reduces the deficiencies of the code, enhances technical skills, improves team communication, and is considered to be more enjoyable for the participants (VanDeGrift, 2004). Studies that compared the performances of paired students and solo students showed that the former were more likely to hand in solutions for their assignments that were of higher quality.

Indeed, in the design of tools for supporting 'collaborative' programming, less importance has been given to pair incompatibility and unequal contribution of participants (Chaparro, 2005). Also, in most of the experiments, less attention has been given to the nature and design of the exercises that students work upon. However, these problems have been a subject of study in the field of collaborative learning for a long time. As argued in Dillenbourg (1999), just putting people together does not mean that they will collaborate. There should be an identified need for collaboration, which may take place at different stages. The collaboration may be in the form of groups where the students act equivalently, or of role-playing where they act according to specific roles (e.g., pair programming).

Various methods have been implemented for forming the groups. As Cliburn (2003) highlights in his paper, McDowell et al. (2002) allowed students to choose their own partners. Nagappan et al. (2003) used a software program to make random partner assignments. Thomas et al. (2003) had students rate themselves on a scale of 1 to 9 in terms of their programming ability, and then assigned partners based on these ratings. In some projects, students who rated themselves highly (7-9) were paired with students who rated themselves poorly (1-3). In other projects, students were

paired with those who rated themselves similarly. Williams and Kessler (2003) also discussed forming pairs based on personalities (whether someone is an extrovert or introvert). In many studies (Nagappan et al., 2003; Thomas et al., 2003; Williams et al., 2002) programmers were assigned new partners throughout the semester. In others (McDowell et al., 2002), students kept the same partner throughout the term unless unforeseen circumstances occurred. Domino et al. (2007) reported better performance and satisfaction outcomes using face-to-face pair programming as compared to its virtual setting. They also found that limiting the extent of collaboration can be effective, especially when programmers are more experienced. Sfetsos et al. (2009) have shown better performance and collaboration-viability for pairs with heterogeneous personalities and temperaments.

Case Studies

In VanDeGrift's (2004) study, the context was a ten week introductory programming course for teaching the Java programming language. To introduce the pair programming concept, students read the paper, 'All I Really Need to Know about Pair Programming I Learned in Kindergarten' (Williams & Kessler, 2000a). Additionally, the instructors demonstrated pair programming, followed by a briefing session about the roles of drivers and navigators. Students completed three pair programming projects, with each project spanning two weeks. Each student wrote individual reports for each project. The survey results showed that students felt benefited from having a partner for programming projects, since partners helped answer questions, brought complementary sets of ideas and skills to the group, and helped with debugging and problem solving. Students felt that finding time for meeting with their partners posed the biggest burden.

Cliburn (2003) in his paper discusses the use of pair programming in an introductory programming course. Survey results showed that a majority of the students liked working in pairs on the projects. Most students thought pair programming improved their grade. An overwhelming majority of students thought that pair programming made them better at working with others. For the most part, students did feel that peer evaluations made them accountable to their partner.

McDowell et al. (2002) concluded that *"students who programmed in pairs produced better programs, completed the course at higher rates, and performed about as well on the final exam as students who programmed independently."* In their experiment, data was gathered from approximately six hundred students enrolled in four sections of an introductory programming course at the University of California. One of the two sections required students to complete programming assignments in pairs, while the other required students to write programs independently. Among all students who completed the course, students in the pairing class scored significantly higher on the programming assignments than those in the non-pairing class.

Further empirical evidence of the effectiveness of pair programming is provided by an experimental study conducted by Williams and Kessler at the University of Utah (Williams & Kessler, 2000b). In this study, forty-one upper level students enrolled in a course on web design were randomly assigned to complete four programming projects either independently or in pairs. During each programming cycle, the thirteen solo programmers completed one program, while the fourteen pairs completed two. Across all four cycles, on an average, the collaborators had 15% fewer defects in their programs than the solo programmers. Furthermore, collaborators spent, on average, only 15% more time completing two projects than the solo programmers spent completing one, suggesting that pair programming is 40-50% faster than programming alone. In addition to producing more bug free code, pair programming appears to enhance the programmers' enjoyment and confidence.

Students practicing collaborative programming, as well as professional pair programmers, were anonymously surveyed. Over 90% reported enjoying their jobs more when working in pairs, and

95% reported feeling more confident in their solutions (William & Kessler, 2000b). Nagappan et al. (2003) reported an improvement in pairs' grades on exams over students who programmed individually. Brereton, Turner, & Kaur (2009) report the results of a systematic literature review of ten empirical studies. They conclude that pair programming may improve undergraduate students' pass and retention rates on programming modules and is likely to improve their confidence in their work and attitude towards programming.

Limitations of Existing Work

The pair programming method as defined and practiced in available literature has a few limitations and weaknesses. In pair programming, the pair sits on the same computer all the time and takes turns to write the code. This technique has a major disadvantage when one of the students in a pair is dominant or a very high scoring student as compared to the other student. In such cases, the other student tends to become dormant, sometimes due to a complex, sometimes due to the fear of being wrong while writing code, and sometimes just to avoid work because it is being completed easily, thus becoming a social loafer. The stronger student does most of the work, diminishing the learning experience for the weaker partner. Lui and Chan (2006) reported a software process fusion (SPF) process in a real software production situation by alternating between the pair and solo programming. They found that the longer team members work alone, the more code patterns they develop for reuse later in pairs.

According to Dillenbourg (1999), in order to maximize the likelihood of the specific forms of interaction as he had mentioned in his definition of collaborative learning, there are four conditions to accurately set a collaborative context:

1. Set up the initial conditions: This involves taking decisions about group formation. It is difficult to set up initial conditions that guarantee effective collaborative work. At this stage the faculty is required to take decision about the size, heterogeneity, geographical, and temporal placement of peers, i.e., face to face co-location, side by side co-location, geographically dispersed locations, collaboration technology (if any), selecting peers, etc.
2. Over-specify the collaboration contract with a scenario based on roles: The collaboration contract can be specified by setting up differences among learners either by triggering conflictual interactions or else because of peers' complementary knowledge through role playing or limiting their access to certain part of information.
3. Scaffold productive interactions by encompassing interaction in the medium: This encompasses specifying interaction rules in face to face or technology enabled collaboration.
4. Monitor and regulate the interactions: The teacher may decide to directly facilitate, monitor, and regulate the face-to-face or technology-supported collaboration among learners. Alternatively, a mechanism or a tool may be developed for self regulation by peer learners.

Pair programming as reported so far does not completely satisfy the above-mentioned conditions. In many case studies pair programming was practiced outside the class where students were required to complete their assignments and project work with their partner in whatever time they could spare. Such a situation did not fulfill the fourth condition, as the collaboration was not monitored and led to a major disadvantage where students complained that it was difficult for them to be free at the same time. In VanDeGrift's (2004) experiment, nearly half of the students who worked in pairs felt that it was extremely difficult to schedule time for meetings. At times, students have also complained of unreliable partners (Bevan et al., 2002) and the possibility of being paired with a parasite (Cliburn, 2003), suggesting that pair programming does not provide a

scenario based on well-defined roles. Although pair programming does specify that one student should act as the driver and the other as navigator, it does not assign specific tasks to individual students, thus leaving the students on their own to choose their tasks.

Methodology

After identifying limitations in the implementation of pair programming as it exists today, we transformed the concept of pair programming, in the light of Dillenbourg's framework, to make sure that both the students in a pair necessarily do an equal amount of work. Our research was aimed at investigating if our proposed model of collaborative pair programming can help inexperienced programmers to learn fast and reach at par with their experienced peers. As an outcome of this intervention, we wanted to increase students' motivation, give them the confidence of working individually and collaboratively, and also apply their knowledge to different situations. Therefore, we tried to set up conditions that were meaningful to the students, which were related to a goal that was challenging, and gave students the opportunity to express their beliefs and opinions. Table 1 shows Dillenbourg's four requirements for maximizing collaborative learning and how we implemented each in our study.

Table 1. Application of Dillenbourg's Principles

Dillenbourg's requirement	*Our implementation*
1. Set up the initial conditions.	Pairs of two students without any programming experience were formed by faculty in the beginning of the semester.
2. Over-specify the collaboration contract with a scenario based on roles	In each laboratory session, the members of pair were first required to individually complete two different programming tasks. On completion of both their individual tasks, they worked together to solve a more complex problem that was designed as an extension of both their individual problems (refer to Table 2).
3. Scaffold productive interactions by encompassing interaction in the medium	The pair members were not allowed to interact for completing their individual tasks. However, if one of the pair member completed his/her task much in advance, and the other member felt the need of peer's support even for completing his/her individual task, the laboratory instructor allowed them to do so by assigning a small penalty of marks to the second member.
4. Monitor and regulate the interactions	For every group of thirty students, at least two faculty members and one teaching assistant were available for doubt clearance and monitoring.

Our adapted implementation of collaborative pair programming was based on Dillenbourg's four conditions of collaborative learning as discussed in the previous section. Each combined task was designed based on the individual tasks and in a way such that it demands collaboration of earlier work. Each exercise or problem was designed in two parts. The first part required both students in a pair to work independently. Each of them was given a problem to solve that was of the same difficulty level and required the application of the same concepts. After solving their individual task, they were given a combined task. The difficulty of the combined task was higher compared to the individual tasks and was designed such that its solution required the students to combine the concepts, logic, and code of both the individual solutions. Each exercise fulfilled the purpose of making the students work both individually and in a team. Table 2 shows a few sample exercises.

There were a total of one hundred and seventy eight students enrolled in this course and they were divided into three batches for laboratory classes. The students were divided into two categories. The first category consisted of 66 students having prior programming experience during their K-12 education. This set of students was asked to work independently all throughout the semester. The other group consisted of 112 students who had no prior experience in programming. These students were asked to work in pairs. The students had the choice to choose their own permanent partner for the semester.

Table 2: Sample Laboratory Assignment

Individual task 1	*Individual task 2*	*Combined task*
Write a program to make an n*n square using the "$" sign. Get the value of n from the user. Use an incrementing loop (i=0, i<n; i++).	Write a program to make an n*n square using the "$" sign. Get the value of n from the user. Use a decrementing loop (i=n;i>0;i--).	Combine the programs in such a way that exactly half the design is made by incrementing the counter value 'i' while the other half of the design is made by decrementing the counter 'i'.
Write a program to enter ten names and roll numbers and print them.	Write a program to enter ten names and ages and print them.	Write a program combining the two codes such that the output shows the name, roll number and age for all the common names.
Write a program to input two words and print the number of occurrences of each letter in each word.	Write a program to input two words and print all the common letters in both. Note that a letter will be common in both words if number of occurrences of that letter is same in both words.	Write a program that inputs two strings of any length and checks whether they are anagrams or not. Anagram is a word or phrase made from another by rearranging its letters (Ex.: *now* ➔*won*, *dread* ➔ *adder,* riot ➔ trio*)*. Also if they are not anagrams report the number of letters that are not same.
Write a function to count the number of characters in a given string. The string will be the input argument for the function. Answer has to be printed in the main function.	Write a function to count the number of words in a given string. The string will be the input argument for the function. Answer has to be printed in the main function.	Write a program such that for a given input string the main function calls the word_count function first and as soon as a word is identified it calls the char_count function with this word as input. In this way calculate the number of words and total number of characters in the given sentence.

Experienced programmers were required to solo-program the combined task directly. If at any point in time they found the problem difficult, they could first solve the two problems in the first part and then solve the combined task. Inexperienced students working in pairs had to solve the two problems in the first part individually and separately. Each student had to provide the solution for his or her sub-problem. After this, they had to collaboratively solve the combined task in the second part by combining the concepts they applied in their independent work.

The monitoring of this experiment was well regulated, with at least four lab instructors in each of the three lab sections of approximately sixty students. Each section had two hours of lab instruction every week. Five final year undergraduate students also assisted the first year students during the lab hours. We particularly made sure that students work sincerely for the entire duration of the lab and most of the exercises are solved during the lab hours in the presence of the instructors.

Results and Findings

Evaluation and grades for this course were based upon two minor examinations, major examination, a lab exam, and lab work assessment. To evaluate our experiment, we have analyzed the marks for minor examinations, major examination, and the lab exam. Further, at the end of the semester, an optional questionnaire was administered amongst the students in which they had to rate six parameters on a scale of 0-3, 0 being never and 3 being always. This questionnaire is adapted from a much larger questionnaire used to assess the degree of deep learning by the National Survey of Student Engagement (Kuh et al., 2005). Fifty-seven students responded. Some of them could not understand the questions clearly and asked for clarification. Table 3 shows the

average of each point for experienced-solo as well as inexperienced-pair programmers. The averages are calculated from the responses of seventeen solo programmers and forty paired programmers.

The students were also involved in discussions and feedback interviews where they were asked to share their views on the benefits they achieved while using the new method for solving their labs. These discussions took place during, as well as, after the course. Some of the students' comments are quoted here:

> Student 1: "I agree that when two people with no programming background work together they learning more easily."
>
> Student 2: "Working with a partner helped me. We could identify different ways of solving the same problem when we combined individual tasks."
>
> Student 3: "This new method made us think deeply and shaped our views towards a good approach to problem solving. Both partners had a feeling that they had the support of each other, and this added to the motivation level."
>
> Student 4: "Working with a partner really helped me. I had no programming background but I could ask my partners all doubts without any hesitation. Combining individual programs made us come across more mistakes."
>
> Student 5: "The new method was a life saver for students with no programming background. They were able to grasp much more. Two of my friends secured an 'A' even though they had no programming background."

Table 3: Survey Results: Comparison between paired and solo programmers.

S.No	How often did you have the opportunity:	A Average rating by inexperienced-paired programmers (0-3) 0: never; 3: always	B Average rating by experienced-solo programmers (0-3) 0: never; 3: always
1	to work with classmates outside of class to prepare solutions to exercises	1.85	1.29
2	to Analyze the basic elements of an idea, experience, or theory, such as examining a particular case or situation in depth and considering its components	1.90	2.06
3	to Synthesize and organize ideas, information, or experiences into new, more complex interpretations and relationships	1.90	1.59
4	to make judgments about the value of variables, arguments, or methods, such as examining how others gathered and interpreted data and assessing the soundness of their conclusions	1.90	2.06
5	to work on assignments that took more than an hour to complete	1.60	2.06
6	to better understand someone else's views by imagining how an issue looks from his or her perspective	1.80	1.59

For all the laboratory instructors and teaching assistants, the most common observations were to find paired programmers brainstorming far more than individual programmers, suggesting alternate implementations during evaluations, approaching instructors for doubts less often than individual programmers, and having more details like null checks and memory checks in their pro-

grams. It led the students to check their thinking and reason their decisions; they examined and discussed their ideas with others and evaluated other's statements and solutions. They modified their own programs to fit in their code in the new but similar situation presented by the combined task question and, at the same time, also acted as evaluators for their partner's programs. We believe that this experience trained them for reading and building upon others' code in future. The instructors also felt that student pairing also helped in improving the effectiveness of teacher student interaction in the labs.

Based on the results shown in Table 3 and the responses and statements of the students and instructors, some of the evident advantages of collaborative programming that we could bring out effectively in our course were:

1) Efficiency: According to Bevan et al. (2002), pairs spend less time working on assignments than individuals. In our experiment also inexperienced-pair programmers could produce code of the same quality in the same time as experienced-solo programmers. Although paired programmers had to write more code, (individual and combined tasks), they seldom took more than an hour to complete the task. This happened because when students attacked the combined task, the students working in pairs could work out the logic much easily and in less time as they had already grasped the concept while working on the individual tasks.

2) Trust and Teamwork: Collaborative programmers got to know their teammate really well, which helped to build trust and improve team work. Discussions with students brought forward the fact that paired programmers were more comfortable in clearing their doubts with their partners. When they worked collaboratively, they showed the confidence to state when something was right and the ability to admit when something was wrong. Another advantage that was found in the students' responses was that collaborative workers developed the tendency to work together even outside the class.

3) Problem solving skills: "*Four eyeballs are certainly better than two*" (Williams, Kessler, Cunningham, & Jeffries 2000). Collaborative programmers talked, discussed, and argued more than the individual programmers. They had the additional and increased opportunity to learn by watching how their partners approach a task, how they use programming language features, and how they use the development tools. They had the opportunity to better understand someone else's view by understanding how an issue looks from their partner's perspective. At such times, drawing from each person's unique talents and experience, a process known as 'pair brainstorming' occurs resulting in highly effective problem solving. The simple act of explaining an issue often leads to the solution faster.

4) Quality: Defects are prevented because knowledge is constantly shared between pairs (Jason, 2004). Though no specific measure were made about program defects, the lab instructors felt that compared to earlier batches when such a pairing was not tried out, the quality of the programs produced by novices improved significantly. In our experiment, the defect errors were reduced while solving the difficult task because the students had already encountered more than half the errors while solving the easier but similar exercises.

Along with the questionnaire, we also analyzed the marks of all the students enrolled in this course. Table 4 shows the average marks of experienced-solo, as well as inexperienced-paired programmers, in the two minor written examinations, lab examination, and the major written examination. In all the exams, the students were asked to write and analyze small programs. None of the examination questions was taken from the assignments or lab exercises. It also shows the percentage difference in the performance of individual and collaborative programmers. It can be seen that all the exams were designed such that they were found to be quite challenging even by experienced programmers.

Table 4: Examination results: Comparison between paired and solo programmers.

Evaluation Stage	Examination	Time of conduct of exam (after commencement of course)	A Average marks by inexperienced-paired programmers	B Average marks by experienced-solo programmers	Relative performance gap (B-A)/B
1.	Minor I: 20 marks	6 weeks	6.81	11.06	0.38
2.	Minor II: 20 marks	12 weeks	5.96	9.65	0.38
3.	Lab Exam: 40 marks	16 weeks	24.95	28.58	0.13
4	Major Exam: 50 marks	18 weeks	23.46	26.03	0.10

The trend, as seen in Table 4, clearly indicates that as the semester progressed, the inexperienced-paired programmers were coming at par with the experienced-solo programmers. The difference in the performance of students in the minor exams is due to the fact that in both the minor exams, a major portion of the paper was based on the topic last taught in class. Both the times, labs had not yet been conducted on those topics. Students with a programming background of K-12 days were already fluent in programming and, hence, managed to achieve a much better result. This also shows that for students having no programming background, the maximum learning experience came from the lab work. By the end of the semester, inexperienced-paired programmers reduced the relative performance gap from 40% to 10% and performed at the same level as the experienced-solo programmers during the final examinations.

The lab examinations were conducted towards the end of the semester. By then, the inexperienced programmers had improved significantly. After working collaboratively for about five months, students had become comfortable with lab exercises. When they were asked to individually solve difficult problems during the lab examinations, they analyzed the problem to manageable parts and synthesized an acceptable solution from these parts, just as they did while working in pairs. This accounted for the enhanced performance of the inexperienced programmers.

Conclusions and Future Work

We have used Dillenbourg's four conditions of collaborative learning to define a novel approach of collaborative pair programming. In our framework, the students are first required to complete different sub-problems and later integrate their solutions to solve a more complex problem. We have tested our approach in an introductory programming course for first year engineering students.

This approach has resulted in benefits such as enhancement of problem solving skills, efficiency, quality, trust, and teamwork skills. We have also observed that paired laboratory experience is especially advantageous to inexperienced programmers. A hidden advantage that was evident from the students' responses to the feedback sessions was that paired programmers were motivated to work collaboratively even outside the class, although this was not required as part of our experiment. Such results helped us to conclude that our aim of introducing collaborative work among programmers in the earliest stages of their undergraduate studies had been successful. As our experiment was well monitored, none of the students got neglected or overshadowed by dominant partners. Also, as each paired programmer had the chance to work individually before working with their partner, it gave them the opportunity to enhance their personal skills as well,

leaving little scope for dormancy. However, this study may have some biases as we did not compare the effect of collaborative pair programming vs. solo programming within novice programmers. We also did not compare the effects of our model with the traditional form of pair programming. This study also does not delve into detailed statistical analysis.

With the support of some colleagues, the first author has also applied this model in another course – Object-Oriented Programming. In our department, we now intend to apply the model to many other courses in future, especially the courses on Data Structures, Database Management, and Web Technology Laboratory. Some of the areas of improvement which would be addressed in these future studies are a quantifiable measure of code quality, are analysis of the statistical significance levels of the survey results. Another variation to this implementation would to allow a few novices to work as individual programmers and compare the growth of the paired novices with unpaired novices. Similarly, we need to extend the study to pairing amongst the experienced programmer. We plan to concentrate on these areas in our future studies.

Acknowledgements:

The authors are highly thankful to the cooperation and support extended by Prakash Kumar, Divakar Yadav, Manish Thakur, Aarti Gupta, Uma Moorthy, Siddhartha Singh, Badal Vishal, and Deepak in conducting the laboratory classes. The first author is indebted to Dr. Mukul K. Sinha for his insightful views and enriching discussions. The encouragement provided by Prof. J. P. Gupta has been the source of strength for bringing innovative teaching practices at JIIT. The inputs given by reviewers, editors, and Harkesh Singh Dagar also need a special mention.

References

Bevan, J., Werner, L., & McDowell, C. (2002). Guidelines for the use of pair programming in a freshman programming class. *Conference on Software Engineering Education and Training* (pp. 100-107). KY: IEEE Computer Society.

Brereton, P., Turner, M., & Kaur, R. (2009). Pair programming as a teaching tool: A student review of empirical studies. *Proceedings of 22nd Conference on Software Engineering Education and Training* (pp. 240-247). IEEE.

Brusilovsky, P., Kouchnirenko, A., Miller, P., & Tomek, I. (1994). Teaching programming to novices: A review of approaches and tools. In T. Ottman & I. Tomek (Eds.), *Proceedings of ED-MEDIA'94 - World Conference on Educational Multimedia and Hypermedia* (pp. 103-110). Vancouver, BC.

Chaparro, E. A. (2005). An intelligent cognitive tool to foster collaboration in distributed pair programming. *8th Human Centered Technology Postgraduate Workshop*, University of Sussex. Retrieved from http://hct.fcs.sussex.ac.uk/Submissions/22.pdf

Cliburn, D. C. (2003). Experiences with pair programming at a small college. *Journal of Computing Sciences in Colleges*, *19*(1), 20-29.

Cockburn, A., & Williams, L. (2001). The cost and benefits of pair programming. In G. Succi & M. Marchesi (Eds.), *Extreme programming examined* (p. 223-243). Addison-Wesley Longman. Retrieved from http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF

Dillenbourg, P. (1999). What do you mean by collaborative learning? In P. Dillenbourg (Ed.), *Collaborative learning: Cognitive and computational approaches* (pp. 1-19). Retrieved from http://tecfa.unige.ch/tecfa/publicat/dil-papers-2/Dil.7.1.14.pdf.

Domino, M. A., Collins, R. W., & Hevner, A. R. (2007). Controlled experimentation on adaptations of pair programming. *Information Technology and Management, 8*(4), 297-312.

Driscoll, M. P. (2004). *Psychology of learning for instruction* (pp. 227-245). Allyn & Bacon.

Goel, S. (2006a). Competency focused engineering education with reference to IT related disciplines: Is the Indian system ready for transformation? *Journal of Information Technology Education, 5,* 27-52. Retrieved from http://www.jite.org/documents/Vol5/V5p027-052Goel88.pdf

Goel, S. (2006b). Investigations on required core competencies for engineering graduates with reference to the Indian IT industry. *European Journal of Engineering Education*, *31*(5), 607-617.

Gogoulou, A., Gouli, E., Grigoriadou, M., & Samarakou, M. (2003). Exploratory + collaborative learning in programming: A framework for the design of learning activities. *Proceedings of the 3rd IEEE International Conference on Advanced Learning Technologies*, pp. 350-351. Retrieved from http://ieeexplore.ieee.org/iel5/8621/27318/01215118.pdf

Goldfarb, M. E. (2001). The educational theory of Lev Semenovich Vygotsky (1896 - 1934). Edward G. Rozycki & M. F. Goldfarb and Associates. Retrieved from http://www.newfoundations.com/GALLERY/Vygotsky.html

Kuh, G. D. et al. (2005). *Exploring different dimensions of student engagement: 2005 annual survey results*. National Survey of Student Engagement, Indiana University, USA. Retrieved from http://nsse.iub.edu/pdf/NSSE2005_annual_report.pdf

Kutay, C. (2005). Implementation patterns for supporting learning and group interactions. PhD thesis, University of New South Wales, p. 19. Retrieved from http://www.library.unsw.edu.au/~thesis/adt-NUN/uploads/approved/adt-NUN20060823.125823/public/02whole.pdf

Jason, A. (2004). Technical and human perspectives on pair programming. *ACM SIGSOFT Software Engineering Notes, 25*(5), 1-14.

Ladyshewsky, R. K. & Ryan, J. (2002). Reciprocal peer coaching as a strategy for the development of leadership and management competency. *Teaching and Learning Forum,* Edith Cowan University, Australia. Retrieved from http://www.ecu.edu.au/conferences/tlf/2002/pub/docs/Ladyshewsky.pdf

Lave, J., & Wenger, E. (1991). *Situated learning: Legitimate peripheral participation*. Cambridge, UK: Cambridge University Press.

Lipponen, L. (2002). Exploring foundations of computer-supported collaborative learning. *Proceedings of the Computer-supported Collaborative Learning 2002 Conference.* Retrieved from http://www.helsinki.fi/science/networkedlearning/texts/lipponen2002.pdf

Lui, K. M., & Chan, K. C. C. (2006). Software process fusion: Uniting pair programming and solo programming processes. In Q. Wang et al. (Eds.), *Proceedings of SPW/ProSim 2006, LNCS 3966*. (pp. 115 – 123). Springer-Verlag Berlin Heidelberg.

McDowell, C., Werner, L., Bullock, H., & Fernald, J. (2002). The effects of pair programming on performance in an introductory programming course. *Proceedings of the Thirty-Third Technical Symposium on Computer Science Education (SIGCSE 2002),* pp. 38-42. ACM Press.

Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C., & Balik, S. (2003). Improving the CS1 experience with pair programming. *Proceedings of the Thirty-Fourth Technical Symposium on Computer Science Education (SIGCSE 2003),* pp. 259-362. ACM Press.

Preston, D. (2005). Pair programming as a model of collaborative learning: A review of the research. *Consortium for Computing Sciences in Colleges*, pp. 39-45.

Sfetsos, P., Stamelos, I., Angelis, L., & Deligiannis, I. (2009). An experimental investigation of personality types impact on pair effectiveness in pair programming. *Empirical Software Engineering*, *14*(2), 187-226.

Sharp, H., & Robinson, H. (2005). Some social factors of software engineering: The maverick, community and technical practices. *Proceedings of the 2005 workshop on Human and social factors of software engineering, International Conference on Software Engineering,* pp.1-6. ACM.

Smith, M. K. (2007). *Learning theory*. Infed. Retrieved from http://www.infed.org/biblio/b-learn.htm

Thomas, L., Ratcliffe, M., & Robertson, A. (2003). Code warriors and code-a-phobes: A study in attitude and pair programming. *Proceedings of the Thirty-Fourth Technical Symposium on Computer Science Education (SIGCSE 2003),* pp. 363-367. ACM Press.

VanDeGrift, T. (2004). Coupling pair programming and writing: Learning about students' perceptions and processes. *Proceedings of the Thirty-Fifth Technical Symposium on Computer Science Education (SIGCSE 2004)* pp. 2-6. ACM Press.

Williams, L. A., & Kessler, R. R. (2000a). All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM, 43*(5), 108-114.

Williams, L., & Kessler, R. (2000b). Experimenting with industry's 'pair programming' model in the computer science classroom. *Journal on SW Engineering Education.* Retrieved from http://collaboration.csc.ncsu.edu/laurie/Papers/CSED.pdf

Williams, L., & Kessler, R. (2003). *Pair programming illustrated*. Boston, MA: Addison Wesley.

Williams, L., Kessler, R.R., Cunningham, W., & Jeffries, R. (2000). Strengthening the case for pair programming. *IEEE Software*, *17*(4), 19-25.

Williams L., Wiebe E., Yang K., Ferzli M., & Miller C. (2003). In support of pair programming in the introductory computer science courses. *Computer Science Education, 12*(3), 197 - 212.

Williams, L., Yang, K., Wiebe, E., Ferzli, M., & Miller, C. (2002). Pair programming in an introductory computer science course: Initial results and recommendations. *OOPSLA Educator's Symposium*, Seattle, WA. Retrieved from http://collaboration.csc.ncsu.edu/laurie/Papers/EdSym_PL_0318.pdf

Whitehead, J. (2007). Collaboration in software engineering: A roadmap. *Future of Software Engineering, (FOSE'07)*, IEEE Computer Society, pp. 214-225.

Whitworth, E., & Biddle, R. (2007). The social nature of agile teams. *Agile 2007 Conference, IEEE Computer Society,* pp. 26-36.

Biographies

Sanjay Goel is Associate Professor and Head of the Department of Computer Science & Engineering/Information Technology at Jaypee Institute of Information Technology (JIIT), Noida, India. In this capacity, he has created and delivered many novel courses for computing students. He earned hisM.Tech. (Computer Technology) from IIT Delhi, and his B.E (Hons.) (Electrical and Electronics Engg.) and M.Sc. (Hons.) (Physics) from BITS, Pilani. He has 24 years of mixed experience in teaching, software development, and interactive multimedia content creation. Prior to joining JIIT in 2002, he was Director (Multimedia) at Indira Gandhi National Centre for the Arts (IGNCA). He has been a faculty member at Delhi University for more than a decade, and has also worked at National Informatics Centre, New Delhi. He enjoys inter-disciplinary work and his professional interests include Computing Education, Cultural Informatics, Multimedia Computing and Applications, and Software Design.

Vanshi Kathuria is an Analyst Programmer with Accenture Technology Solutions, USA. She earned her B.Tech. (Computer Science) from Jaypee Institute of Information Technology (JIIT), Noida, India. She has 4 years of experience in software development, application maintenance, and project delivery and management. She has worked for global clients in the Life Sciences & Healthcare and Consumer Goods domains and specializes in Java programming and Content Management Systems. Her professional interests include Interface Designs, Web Services and Multi-tier System Architectures.