Empirical Evaluation of Collaborative Support for Distributed Pair Programming

Jesus Favela[1], Hiroshi Natsu[1], Cynthia Pérez[1], Omar Robles[1], Alberto L. Morán[2], Raul Romero[1], Ana M. Martínez-Enríquez[3], and Dominique Decouchant[2]

[1] Departamento de Ciencias de la Computacion, CICESE, Ensenada, Mexico
{favela,hnatsu,orobles,cbperez,romero}@cicese.mx
[2] Laboratorire LSR, Grenoble, France
{Alberto.Moran,Dominique.Decouchant}@imag.fr
[3] Depto. de Ing. Electrica, CINVESTAV-IPN, D.F., México
ammartin@mail.cinvestav.mx

Abstract. Pair programming is an Extreme Programming (XP) practice where two programmers work on a single computer to produce an artifact. Empirical evaluations have provided evidence that this technique results in higher quality code in half the time it would take an individual programmer. Distributed pair programming could facilitate opportunistic pair programming sessions with colleagues working in remote sites. In this paper we present the preliminary results of the empirical evaluation of the COPPER collaborative editor, developed explicitly to support pair programming. The evaluation was performed on three different conditions: pairs working collocated on a single computer; distributed pairs working in application sharing mode; and distributed pairs using collaboration aware facilities. In all three cases the subjects used the COPPER collaborative editor. The results support our hypothesis that distributed pairs could find the same amount of errors as their collocated counterparts. However, no evidence was found that the pairs that used collaborative awareness services had better code comprehension, as we had also hypothesized.

1 Introduction

Pair Programming is a software development technique that is part of the Extreme Programming methodology. In pair programming two developers work side by side, on a single computer, to jointly produce an artifact (design, algorithm, code, etc.). It has been reported that this technique can be accounted for the development of higher quality software in half the time it required a single programmer [1,8].

The two programmers work as a unit, as a single mind responsible for all aspects of the artifact. One programmer, the driver, controls the pen, mouse, or keyboard to write the code. His colleague actively observes the work produced by the driver, looking for defects, alternatives and considering the implications of the strategy being followed. The pair changes roles periodically. Both participants are active throughout the process and share the responsibility for the work being produced [7].

The successful application of this technique requires the use of an appropriate workplace: "The programmers should be able to sit side by side and program, looking simultaneously at the computer screen, sharing the keyboard and the mouse" [7]. While they work together, the couple should be able to share the keyboard without

G.-J. de Vreede et al. (Eds.): CRIWG 2004, LNCS 3198, pp. 215–222, 2004.

changing seats. Extreme programmers need to be constantly in touch with their team and for this, their workspace has to be open and facilitate communication among peers as well as casual and informal encounters.

An important trend in software development has been the globalization of the software industry [2]. With increased frequency, software developers are required to work in groups that are geographically distributed. Under these circumstances the requirement for pair programmers to be in the same location seems an important limitation of this approach. For this reason we have developed the COPPER synchronous collaborative writing tool, designed specifically to support pair programming among distributed collaborators [5]. In this paper we report the preliminary results of an empirical evaluation performed to determine the feasibility of distributed pair programming and the services offered by COPPER in this regard.

2 The COPPER Pair Programming Editor

COOPER is a synchronous collaborative editor designed to support pair programming [5]. It is a based on a client-server architecture and composed of two main subsystems: the Collaborative Editor, and the User and Document Presence module (UD&P). On the client, these subsystems form an application used to write programs, access document services and communicate with peers.

The editor can be used disconnected from each other (in individual mode) or connected in pairs (in synchronous collaborative mode). Users can be either co-located or distributed on the Internet. The editor implements a turn-taking synchronous editor (see Figure 1). The user holding the floor (or editing right) uses the editing window (Figure 1F) to work on the currently loaded document.

Common document management and editing functionality is provided by means of the application‘s Menu bar and the toolbar (Figure 1A). Actions performed in the editor are propagated to the collaborator’s client. As an example of their use, consider the Open button (third button left to right) of the toolbar. When pressed, an Open dialog (Figure 1G) appears, providing access to documents stored in the Document server. This dialog presents an integrated file hierarchy with documents from the local machine and from other distributed WebDAV servers [6]. This allows for seamless navigation and document retrieval from the individual or collaborative work environment. Several documents can be edited at the same time, even if these documents come from different WebDAV servers.

The Document server offers a centralized information repository, which provides document storage, editing access control, user authentication and permissions to avoid unauthorized accesses, and document presence extensions through an instant messaging client, which “listens” and informs the UD&P system the results of operations performed on the documents [4].

Each editor client “owns” the documents opened by its user, and it is the only one allowed to perform operations, such as *Save*, *Save As*, and *Close*, on these documents. When the connection is broken, each client keeps their “own” documents, and the user can continue working on them in individual mode. While in this mode, clients are ready to send or receive invitations to begin collaborating.

Floor management (or editing access control) is represented using a “traffic light” metaphor (Figure 1B), which is activated when working in collaborative mode. This component includes an action button to request and grant floor control.

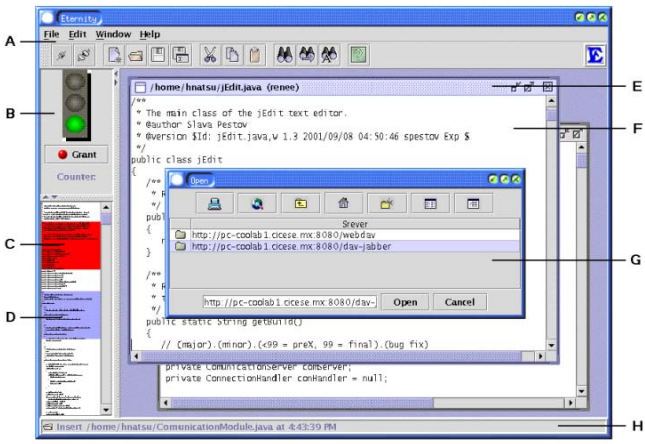

Fig. 1. The COPPER Pair Programming System.

Collaboration awareness is provided by means of a radar view (Figures 1C and 1D), editing window titles (Figure 1E), the status bar (Figure 1H), and the floor or editing control access component (described earlier). The radar view provides a general overview of the document being edited; it shows document changes in real time, and serves as a document navigation tool; selecting a particular line of the document in the radar view causes the current editing window to display the segment of the document where the selected line is present.

Editing window titles (Figure 1E) display the name and location of the document, as well as the identities of the owner of the document and of the collaborator (if present).

The status bar (Figure 1H) provides information on the last operation performed by any of the collaborators, as well as on the date and time it was performed.

The client-side module of the UD&P subsystem is used to send and receive messages to/from collaborators, manage the user's own presence and provide this presence information to other collaborators. Additionally, it extends this functionality to interact with the documents in a similar way as one would interact with users [4]: adding or deleting documents from the document (presence) list, and sending "group" messages to subscribed users of a document. Furthermore, subscribed users receive messages from the UD&P service whenever the document's availability and status information changes. Finally, this module implements the functionality offered by WebDAV, to perform basic editing operations on subscribed documents (e.g. locking a document to avoid users from concurrently modifying the same document).

3 Experimental Design and Procedure

We conducted an experiment to explore the potential and limitations of distributed pair programming. To guide our research we established two working hypothesis:

H1: Distributed pair programmers working remotely will find the same number of defects as collocated programmers in the same amount of time.

H2: Distributed pair programmers using COPPER will have better code comprehension as distributed pair programmers using application sharing in NetMeeting.

The first hypothesis aims at establishing that pair programming can be as successful when the group is distributed as when they are collocated, if appropriate technical support is provided. The second hypothesis is established from our believe that tools like NetMeeting, which offer limited collaboration awareness services are not adequate to support the intense level of interaction required for synchronous tasks such as software programming or design.

The subjects of the study were 12 graduate students in computer science and proficient in the Java programming language. The students were randomly grouped in 6 pairs. All subjects attended a one hour session that included an introductory lecture on pair programming, an explanation of the COPPER editor and time for hands-on experience with the system. In addition, the subjects completed a questionnaire which focused on their previous programming experience.

We used a within-subjects design; with all six groups asked two perform three different programming tasks in three different setups.

3.1 Experimental Setup

The modality in which the subjects performed the task was our independent variable. The experiment required both subjects to collaborate in performing three different programming tasks, each task was performed under a different condition.

Collocated Condition. In this condition, the two programmers were in the same office and shared a single computer running the COPPER editor in single-user mode. The programmers shared the display and keyboard as in traditional pair programming tasks.

NetMeeting Condition. In this condition, the programmers were in different offices, each of them with a workstation. The subjects had a voice connection through a telephone for the duration of the task. The telephone was left in speakerphone mode to free them from having to carry the handset. The subjects used the COPPER editor in single-user mode and shared the application through MS NetMeeting.

COOPER Condition. The physical setup of this condition was similar to the NetMeeting condition. The only difference was that rather than using NetMeeting to share a single application, they used the COPPER editor in collaborative mode. In this way, they had access to the collaboration awareness features of COPPER, such as the radar view, and the use of the semaphore for floor control.

The programmers were videotaped in all three conditions. In addition, there was a researcher in each office who was present at the time of the experiment. This researcher kept control of the time and recorded the number of times the programmers exchanged control of the floor as well as the errors identified, corrected and introduced in the code. To track the progress of the programming team without looking over their shoulders, the researchers had a monitor in a separate desk connected to the programmer's workstation.

3.2 Procedure

The pairs had approximately 55min. to complete each task, which was divided in five phases. The tasks were performed one after the other with 5 to 10 minute rest periods between tasks for an approximate total duration of the experiment of 3 hours per couple. The tasks were executed as follows:

Phase 1. During 10 minutes the programmers were introduced to the programming task they had to perform and the condition in which they had to work. Each person was given an initial version of the program assigned for that task. Each program was injected with 10 errors of different types (syntax, assignment, logic, and interface). The subjects were told that the code included errors but not their type or number.

Phase 2. For the next 15 minutes the couple introduced the code into the editor. To keep this time constant for all programming tasks part of the code was already in the editing window, they were asked to type the remaining 60 lines of code. The pair was able to choose who will be the driver and who the observer. They could also interchange roles during the process. Although the objective of this phase was to introduce the code into the editor, we expected them to detect and correct some errors and become familiar with the code.

Phase 3. Once the code was introduced the subjects were given a maximum of 15 minutes to correct the program, that is, to find and correct errors. This included errors that were introduced by the researchers, and at times, errors introduced by the programmers during Phase 2.

Phase 4. In this phase the programmers were given a text describing a simple modification to be made to the code. They were given a maximum of 15 minutes to complete this phase.

Phase 5. Finally, the pair was asked to complete a survey with 12 Likert-scale assertions, which included topics such as their satisfaction with the modality in which they worked, the perception of the participation of their colleague, and their understanding of the programming task.

3.3 Experimental Tasks

The programming tasks were designed to be simple to assure that they could be done in the short amount of time that was provided and that the students would concentrate on correcting errors and delivering a high quality code. To avoid learning effects the pairs worked on the tasks in different order. Each group was asked to perform the following three programming tasks on a different condition.

LOC. The purpose of this program is to count the number of lines of source code in a file. The program takes a source file as input and returns the total number of lines contained in it. In the last phase the programmers were asked to modify the program to eliminate the number of lines with comments from the final count and also report the number of methods in the source code.

SORT. This program reads a file containing a list of numbers in random order and sorts them in ascending order using the selection sort algorithm. The modification required the students to use insertion sort and present the results in descending order.

N Figures. This is an object-oriented program that reads a file with data of geometric figures of different types and calculates their area, using the method appropriate to each type of figure. The extension requested was to add a new class with a different type of figure.

3.4 Measures

Since we kept constant the amount of time dedicated to the task, we concentrated on measuring the quality of the code since this is an area in which one would expect pair programming to be useful. Additionally we measured code comprehension, since we hypothesize that the help of the colleague would help a programmer understand the code more easily and this might be negatively affected by distance and positively affected by awareness tools.

To estimate code quality we measured the number of errors that were detected and corrected during the task. A researcher observed the programmers as they performed the task and recorded when and by whom errors were detected and corrected. She also indicated when new errors were introduced.

We measured code comprehension by whether or not the pair was able to successfully modify the code and through the questionnaire at the end of the task where we explicitly asked them if they understood the program and whether the help of his colleague helped in this regard.

4 Results and Discussion

4.1 Program Quality

Table 1 shows the number of errors detected by each team for each of the three programming tasks. The results are grouped by condition, with two teams per condition (A and B), per task. The total number of errors detected in the *Collocated* mode was 24, for an average number of errors found in each task of 4 out of the ten that were injected. The *NetMeeting* condition was the most productive in finding errors, with a total of 27. In the *COPPER* mode 22 errors were identified. Of the 73 errors detected, only 3 were not solved. Additionally, one error was solved without the programmers explicitly detecting it. Five groups detected between 11 and 16 errors in the three programming task, with one pair (No. 6), detecting only a total of 4 errors, clearly the team with the poorest performance.

Table 1. Errors detected by each programming team.

Condition		N Figures		LOC		SORT		Total
		Errors detected	Team	Errors detected	Team	Errors detected	Team	
Collocated	A	2	(5)	5	(1)	6	(3)	24
	B	5	(2)	4	(4)	2	(6)	
NetMeeting	A	4	(3)	5	(5)	7	(4)	27
	B	2	(6)	3	(2)	4	(1)	
COPPER	A	5	(1)	0	(6)	3	(2)	22
	B	5	(4)	4	(3)	5	(5)	

These results provide some evidence in support of our first hypothesis. That is, that the number of errors detected is not reduced when the pairs are working in remote locations. In fact, the results were slightly better for the groups that were distributed and used NetMeeting to share the code while the tasks performed with COPPER in distributed mode resulted in a slightly lower number of errors found.

4.2 Code Comprehension

In table 2 we present the results to five of the questions, related to code comprehension, from the survey completed after each task. The survey used a seven-point Likert scale with anchors ranging from strongly disagree (1) to strongly agree (7). Although the programmers were in general able to understand the code, the responses to questions 1, 4, and 5, the ones more directly related to the understanding of the code, indicate that code comprehension was slightly better when the authors were collocated. When comparing the two distributed conditions NetMeeting seemed to work slightly better than COPPER.

Table 2. Perception of code comprehension.

Question	Collocated	NetMeeting	COPPER
1. I understood the program	6	5.58	5.5
2. I found the program to be complex	2.75	2.92	2.75
3. Working with someone helped me find more defects	6.42	6.17	6.25
4. Working with someone helped me to modify the program	6.17	6	5.67
5. Working with someone helped me understand the program	6.33	6.08	5.83

With respect to the actual completion of the tasks assigned to the programmers, we have that 4 groups completed the modification when using NetMeeting, while 5 groups successfully modified the program using COPPER. Results that give a slight edge to the COPPER mode.

The results from the questionnaires and the completion of the task do not provide evidence that the awareness features incorporated in COPPER helped pairs understand the code more than NetMeeting did. Although one more group completed the task on time, the subjects had the perception of having better tackled the task when working with NetMeeting.

5 Conclusions

Extreme programming techniques, and in particular pair programming, are gaining considerable attention given their advantage at handling software development projects with vague or changing requirements. As the software industry continues to grow and their practice becomes global, distributed teams will require appropriate tools to support their software development practices. The development of such tools needs to be supported by empirical research aimed at establishing the necessary services required to support the intensive nature of the collaboration required during this practice.

Towards this end we have presented results of an evaluation conducted with 6 groups that were asked to complete three programming task in different conditions: collocated, distributed and with limited collaboration awareness, and distributed and several collaboration services. Our results seem to support our hypothesis that distributed groups could be as effective in finding programming errors as collocated ones. On the other hand, the additional awareness provided by the COPPER tool didn't seem to facilitate the programmer's understanding of the code over what simple application sharing can accomplish. Analysis of the videos will be conducted to better understand the advantages and disadvantages of each condition.

References

1. Cockburn A. and Williams L., The Cost and Benefits of Pair Programming. Addison Wesley. (2001)
2. Herbsleb J.D. and D. Moitra., Global Software Development. IEEE Software. 18(2), (2001), 16-20
3. Kircher M., Jain P., Corsaro A., and Levine D., Distributed Extreme Programming. Extreme Programming and Flexible Processes in Software Engineering, Italy, May, (2001)
4. Morán, L., Favela, J., Martínez, A., y Decouchant, D., Document Presence Notification Services for Collaborative Writing. In Proc. of CRIWG'2001. IEEE Computer Press. Darmstadt, Germany, Sept. 6-8, (2001), 125-133
5. Natsu, H., Favela, J., Moran, A.L., Decouchant, D., and Martinez, A.M., Distributed Pair Programming in the Web. In Proc. ENC'03, IEEE Comp Society, Mexico, 2003, 81-88.
6. Whitehead E. J., Collaborative Authoring on the Web: Introducing WebDAV. Bulletin of the American Society for Information Science, 25(1), (1998), 25-29
7. Williams L. and Kessler R., All I Really Need to Know about Pair Programming I Learned in Kindergarten. CACM, 43(5), (2000), 109-114
8. Williams L., Kessler R., Cunningham W., Jeffries R., Strengthening the Case for Pair Programming. IEEE Software, 17(4), (2000), 19-25