International Journal of Software Engineering
and Knowledge Engineering
Vol. 16, No. 2 (2006) 293–313
© World Scientific Publishing Company

World Scientific
www.worldscientific.com

HOW DISTRIBUTION AFFECTS THE SUCCESS OF PAIR PROGRAMMING

GERARDO CANFORA*, ANIELLO CIMITILE†, GIUSEPPE ANTONIO DI LUCCA‡
and CORRADO AARON VISAGGIO§

RCOST–Research Centre on Software Technology,
University of Sannio, Palazzo ex Poste, Viale Traiano, 82100 Benevento, Italy
*canfora@unisannio.it
†cimitile@unisannio.it
‡dilucca@unisannio.it
§visaggio@unisannio.it

Received 25 April 2004
Revised 26 July 2005
Accepted 17 August 2005

Recent experiments demonstrated the effectiveness of pair programming in terms of quality and productivity. Growing interest towards global software development is fostering the design of suitable methods and tools for distributing software processes, at any level of detail, from entire subprocesses up to a single activity. Consequently, people placed in different locations could also share programming tasks and related practices, such as pair programming. Unfortunately, distribution might seriously compromise the success of pair programming, due to communication and collaboration issues. We have performed an experiment in order to investigate the impact of distribution on pair programming when performing maintenance tasks. An interesting conjecture stems from the experiment: under certain conditions, distributed pair's components tend to dismiss from each other, stopping the collaborative work. This can be a very expensive risk factor to keep into account when planning tasks of distributed pair programming.

Keywords: Pair programming; distributed software processes; experimental software engineering.

1. Introduction

Agile methods [1] are becoming popular in industrial settings, basically due to two reasons: they allow organisations to release working software in the early stages of development and to decrease software maintenance costs [3]. According to [14], agile methods lead to the reduction of risk exposure, but only with a certain profile of: requirements' stability, system's architecture, team size, and customer's knowledge, and affinity to active collaboration. Many industrial experiences of agile methods have recently been reported in the literature, identifying the most risky and promis-

ing practices, and suggesting recommendations about their successful implementation in real projects [10, 11, 15, 16].

Pair programming [2, 4] is one of the most attractive among the extreme programming practices [17] from both industry and research [5, 6] viewpoints. According to pair programming, two developers work side by side, collaborating on the same task, e.g. system design, algorithm development, code writing, or testing. One developer, named "driver", has the control of the keyboard and actively implements the task. The other developer, usually named "observer", looks at the driver's work and identifies tactical and strategic defects and issues. The observer may also perform complementary tasks, needed to successfully reach the goal of the pair. Such tasks detect flaws and mistakes in the document under development: Notable examples include searching for a better algorithm to use, optimizing parts of the code, and finding better functions or libraries to call. In addition to coding, pair programming can be applied to other phases of the process, such as design, testing, and debugging. From time to time, the developers switch their roles, so that both develop code equally. A detailed protocol to perform pair programming was neither developed nor required by professionals: the only recommendation is to perform continuous review, and the complementary tasks when needed. Conversely, it is highly recommended to leave the pair members free to set the frequency of role switching based on their needs, background, and working style.

Some authors suggested composing the pair with people presenting different levels of experience [23, 24] in order to have greater benefits. Pair programming advocates affirm that it increases software quality without impacting time to deliver [4, 5]; moreover the code produced by pairs show greater readability than the code produced by solo programmers [32]. At the best knowledge of the authors, there are no studies about the rationale for composing pairs in order to make pair programming successful. Social and methodological aspects must be taken into account, such as experience, selfishness, work's method, and leading capability. However, there are no explicit guidelines on such issues yet. A controlled experiment [6] demonstrated that pair programming can improve code quality and decrease development time. Further experiments with students [28, 29, 35] highlighted other benefits of pair programming concerning learning: the practice helps students to produce significantly better programs than the programs produced by individuals; students achieve higher levels of satisfaction and obtain higher grades. An experiment performed by Williams [30] produced the following outcomes: programs produced in pairs passed a greater number of test cases than programs produced by solo programmers; and, pair programming may increase the enjoyment in the work. Lui and Chan [31] conducted an experiment in order to understand when pair programming outperforms traditional solo programming when working on computer algorithms in terms of quality and productivity. Pair programming excels in procedural problems and deduction questions, which are key elements in programming algorithms. The authors conclude that pair programming achieves higher productivity when a pair writes a more challenging program that demands more time to be spent on design. In [34]

the authors analyse how pair programming can mitigate the drawbacks due to the involvement of new people in the late phases of the projects.

Recently, the distribution of software processes has become very widespread in industry and recommended practices are emerging [8, 9, 18]. These practices have been collected within a body of knowledge under the name of *Global Software Development* (GSD) [19, 20]. An economic motivation for GSD is that large organisations tend to acquire smaller companies, with the aim of achieving a competitive advantage by enforcing their workforce, or to penetrate new market segments. Instead of one single large organisation, a structure where many organisations are connected among themselves is becoming more and more widespread [27]. Such a configuration, named *net enterprises or virtual enterprises*, considers the interconnected organisations scattered in different places, but sharing processes at any level of detail, down to individual tasks. Consequently they also share the practices adopted to perform activities, like pair programming. In such cases, organisations can have the need for *distributing pair programming*. Distributed pair programming can be considered as a variation of pair programming where developers are geographically distributed and connected using technological means, rather than sitting in front of the same computer. Communication and collaboration issues become primarily relevant when distributing software processes [21]. Distance could have negative effects on communication-intensive tasks and on spontaneous conversation [22]. Baehti *et al.* [12] investigated the relationships between pair programming and distribution; the results indicate that distributed pair programming in virtual teams is a feasible way of developing object-oriented software. The results of the experiment indicate that software development involving distributed pair programming is comparable to that developed using collocated pair programming or virtual teams without distributed pair programming. Two metrics were used for this comparison: productivity (in terms of lines of code per hour) and quality (in terms of grades obtained). Collocated teams did not achieve statistically significantly better results than distributed teams. Hanks in [33] introduces a tool for supporting distributed pair programming: the case study demonstrates that the use of the tool can significantly improve the distributed work.

Our research work aims at investigating the extent to which the distribution may deteriorate the recognised benefits of pair programming. The research goal is: to analyse the effectiveness and efficacy of distributed pair programming with the purpose of evaluating how distribution deteriorates benefits of the practice, from the viewpoint of the developer, in the context of a software maintenance student project. In order to reach the research goal, we have conducted an experiment at the University of Sannio, Italy. Afterwards, a replica at the University of Naples "Federico II", Italy, was done to confirm the findings of the first experiment. Both experiments have been accompanied by qualitative analysis accomplished with a questionnaire-guided discussion with experiment subjects.

The paper continues as follows. Section 2 shows the experiment setting; Sec. 3 presents the data gathered. Section 4 discusses the experiment's replica and its

findings. In Sec. 5 the outcomes of post-experiment qualitative assessment are illustrated. Finally, in Sec. 6 the experiment validity is treated and Sec. 7 draws the conclusions.

2. The First Experiment

This section describes the experiment made at the University of Sannio in terms of definition of the hypotheses and metrics, characterization of the context, and operation.

2.1. *Definition*

Two main concerns are critical for the distribution of pair programming tasks: collaboration and communication. If the technological platform does not address these two issues adequately, activities like reviewing, switching of roles, and decision-making can be obstructed up to deteriorating the practice itself. There is evidence that pair programming can decrease developing time and increase quality of work; it is reasonable to believe that distribution can cause the lost of such advantages. The research questions we investigated in the experiment are the following:

RQ_1 Are there significant differences in effort when the pair's components are distributed, with respect to co-located pair's components?

RQ_2 Are there significant differences in quality when the pair's components are distributed, with respect to co-located pair's components?

From here on, *distributed pair* indicates a pair whose components are distributed; *co-located pair* indicates a pair whose components are co-located.

The experiment investigated RQ_1 and RQ_2 for maintenance tasks. The null hypotheses were:

$H0RQ_1$: A significant difference in effort required for implementing modifications between distributed pair programming and co-located pair programming does not exist,

$$\mu_{\text{distr\_time}} = \mu_{\text{co-loc\_time}}$$

$H0RQ_2$: A significant difference between the quality of maintenance performed does not exist,

$$\mu_{\text{distr\_quality}} = \mu_{\text{co-loc\_quality}}$$

The alternative hypotheses were:

$H1RQ_1$: A significant difference in effort required for implementing modifications between distributed pair programming and co-located pair programming does exist,

$$\mu_{\text{distr\_time}} \neq \mu_{\text{co-loc\_time}}$$

H1RQ$_2$: A significant difference between quality of maintenance performed does exist,

$$\mu_{\text{distr\_quality}} \neq \mu_{\text{co-loc\_quality}} .$$

The following metrics were used to measure effort and quality:

(1) Effort spent; measured as the difference between the end time and the start time required to accomplish the maintenance tasks; ratio scale. Time was calculated by time sheet fulfilled by subjects.
(2) Quality of the maintenance realised; f_{qual}, an ordinal scale. The quality was evaluated on the basis of black box testing. Test cases were written and executed by experimenters and they were hidden from the subjects.

$f_{qual} = \sum_i (bin_i * over_i)$, where:

bin_i is
- 1 if the maintainers completed the maintenance request
- 0 otherwise

$over_i$ is
- 3 if the modified programs passed tests successfully (80% of tests)
- 2 if the modified programs passed tests with partial success ($< 80\%$ AND $> 20\%$ of tests)
- 1 if the modified programs did not pass tests ($< 20\%$).

$i = 1, 2, 3.$ For i maintenance requests.

Black box testing was used to evaluate quality mainly for two reasons. Firstly, the test driven development practice [3] is used in order to build working code, according to extreme programming; secondly, the purpose of each iteration in extreme programming is the production of a system's feature valuable to the customer.

2.2. *Characterization*

The experimental subjects were volunteer students of the Software Engineering II, a course of the fifth (and final) year of the *laurea* degree in Computer Engineering at the University of Sannio, Benevento. Before running the experiment, the subjects were trained on pair programming. Such training consisted of seminars about agile methods and extreme programming with special focus on pair programming practice, whose duration was 4 hrs. Afterwards, the students spent 2 hrs in the laboratory performing pair programming: they developed some Java programs. The students spent 2 hrs more in the laboratory in order to implement a training round. During that period students used the protocol outlined in Fig. 1 and the experiment technological platform described in Table 1. After the training round, they had the opportunity to enforce their knowledge about the experiment's tasks and execution by discussing their doubts with experimenters.

Driver	– Write the code – Listen to the observer's suggestions and ideas – Leave the keyboard to the observer when needed
Observer	– Continuously check the actions of the driver – Take the control of the keyboard, but only after common agreement with the driver – Achieve off-line tasks – Optimise parts of the code

Fig. 1. Pair programming rules.

Table 1. Experiment technological platform.

Tools	Function	Purpose	Motivation
VNC	Share the desktop: it lets the remote control of a PC.	Collaboration	The experimenters had experience in using it in previous projects; Open Source.
NetMeeting	Text chat.	Communication	Its usage was well known to all the experimental subjects.
JBuilder	IDE for Java Programs.	Programming	Subjects had experience in using it in previous projects.

Table 2. Experimental design.

Subjects	Round I		Round II	
Group A (8 units)	Co-located	P_1	Distributed	P_2
Group B (8 units)	Distributed	P_1	Co-located	P_2

We used a randomised design with one factor (placement of pair's components) and two treatments (co-located and distributed). 16 subjects took part in the experiment, forming 8 pairs organised in two groups (A and B) of four pairs. The experiment consisted of two rounds: during the first round pairs of group A were co-located and those of group B were distributed; they had to modify program P_1, according to three perfective maintenance requests. During the second round pairs of group A were distributed and those of group B were co-located; they had to modify program P_2, also in this case according to three perfective maintenance requests different from Round I. In both rounds pairs were formed randomly within the groups. The design of the experiment is illustrated in Table 2.

The distribution was achieved by placing the two components of each distributed pair in two different laboratories of University buildings. Both programs to be modified during the experiment were written in Java; Table 3 shows information about the two programs.

Table 3. Experiment's classes and related LOCs.

Program	LOCs	# Classes	Functional description
MultisalaMngmt (P_1)	171	3	A booking system for a cinema with many projection rooms.
ProjectMngmt (P_2)	95	4	A system for managing the attributes of a project's activities.

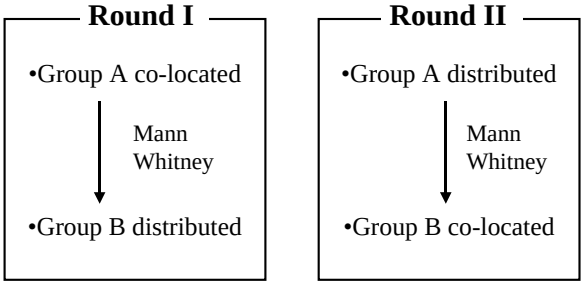

Fig. 2. Test used.

The students were provided with the following documentation:

(1) listings of the programs' code;
(2) textual description of maintenance tasks;
(3) time sheet to fill in;
(4) description of the correct execution of pair programming roles;
(5) questionnaire to be compiled at the end of the experiment.

2.3. *Operation*

The whole experimentation lasted 6 hours. The time was measured with a time sheet filled in by each pair participating in the experiment and was checked out by one of the experimenters. This helped enforce the reliability of results. The function f_{qual} was evaluated by the experimenters, executing the black box tests for the modified programs of each pair. As shown in Fig. 2, we used a Mann Whitney U test between the co-located and distributed pairs of the same round, because data was not normally distributed. This test evaluates the significance of differences in performance and quality when co-located pairs are different from the distributed ones.

3. Analysis of Data

In Table 4, the results of statistical tests on the effort and quality data are reported: both hypotheses were tested by fixing the p-level threshold value at 5%.

Table 4. Statistical test results.

	p-level	Description
Effort Round I	0.564	Mann Whitney test on effort data between Group A (co-located) and Group B (distributed) in Round I.
Effort Round II	1.000	Mann Whitney test on effort data between Group A (distributed) and Group B (co-located) in Round II.
Quality Round I	0.465	Mann Whitney test on quality data between Group A (co-located) and Group B (distributed) in Round I.
Quality Round II	*0.011*	Mann Whitney test on quality data between Group A (distributed) and Group B (co-located) in Round II.

Table 5. Descriptive statistics.

	Co-located					Distributed				
	Dev. Stand	Avrg	Max	Min	Moda	Dev. Stand	Avrg	Max	Min	Moda
Round I										
Quality	2.5	5.8	9	4	5	2.4	5	8	3	3
Effort	7.9	129	135	119	135	46	123	180	75	Na
Round II										
Quality	3	6	9	3	Na	2.5	6.3	9	3	Na
Effort	42.7	116	175	80	Na	52.7	107	155	54	Na

The differences of the response variables are not statistically significant; relying on the experiment outcomes, it cannot be claimed that pair programming efficiency is affected by distribution. Only the Round II quality's results are statistically significant, as the fourth row of Table 4 shows.

Information about descriptive statistics of the sample is provided in Table 5.

The meanings of the acronyms in Table 5 are: Dev. Stand.: standard deviation; Avrg: average; Max: maximum value; Min: minimum value; Moda: the most frequent value of the sample; Na: not available.

Figure 3 shows the box plots of effort in both rounds. The median values are very close; on the contrary, the 25 percentiles are significantly different. This value is an indicator of the performance of the fastest pairs: the best distributed pairs took a shorter time than the best co-located pairs. This suggests that the co-located pairs spent additional time, probably for discussing common policies and strategies to follow when accomplishing the task. We believe that the distributed pairs, after an initial period in which they attempted to collaborate and communicate, broke pair work, behaving as solo-programmers. The best times of distributed pairs, shown in Fig. 3, suggest that subjects did not negotiate strategies with the companion.

The worst times of distributed pairs (Fig. 3) are due to people trying to collaborate although they had problems with communication. In other words, the distributed pairs' components tend to *dismiss* from each other after an initial time

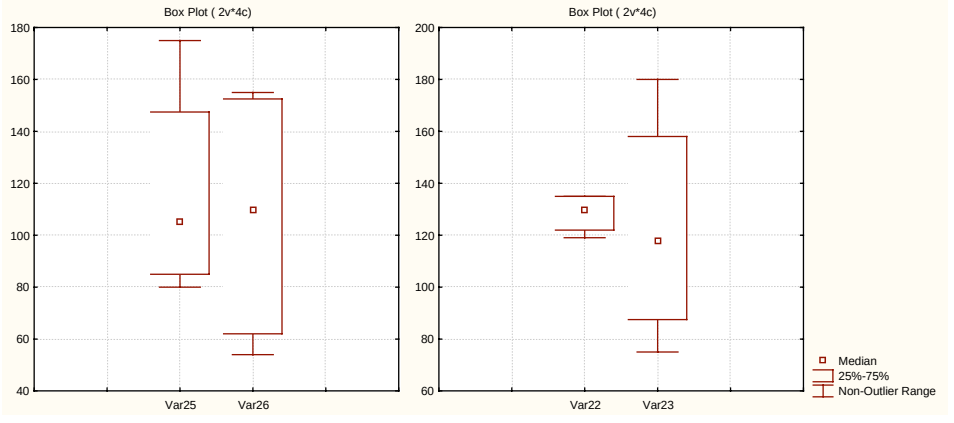

Fig. 3. Effort for co-located (Var 25) and distributed (Var 26) pairs in Round I; effort for co-located (Var 22) and distributed (Var 23) pairs in Round II.

of collaboration. It should be noticed, also, that the dispersion of values of distributed pairs' effort is broader than that of co-located ones in both box plots. The collaboration within the co-located pairs entails a levelling of the upper and lower values of the effort interval. This is due to a phenomenon of performances' compensation: when the driver slows down the rhythm of work, the observer keeps the control of the keyboard and continues the work. By assuming that the distributed pairs' data reflect the behaviour of a solo programmer, the graphs become meaningful. When slowed down, the solo programmer preferred to neglect the help of the observer rather than dealing with matters of communication and collaboration in order to maintain the pairing.

Round II suggests the breaking of the distributed pairs with even more evidence, as shown in Fig. 3: both the median and 25 percentile are greater for co-located pairs; and the dispersion of values is once again broader for distributed pairs. The more remarkable difference with the results of Round I is the interval in which the co-located values vary, which is tighter than in Round I. This difference is probably due to the fact that subjects learned to work better in pairs after Round I experience. This conclusion is confirmed also in Fig. 4, where the quality of co-located pairs is significantly better than that of distributed pairs in the second round.

In summary, the analysis of data from the first experiment suggests that without adequate means of communication and collaboration, the pairs tend to break down. This can be an important risk factor when implementing distributed pair programming. The dismissal occurs mainly under two conditions:

(1) The absence of an adequate communication support: the contemporary review is one of the aspects that make pair programming advantageous. Contemporary review requires fluent communication in order to be decisive for the effectiveness

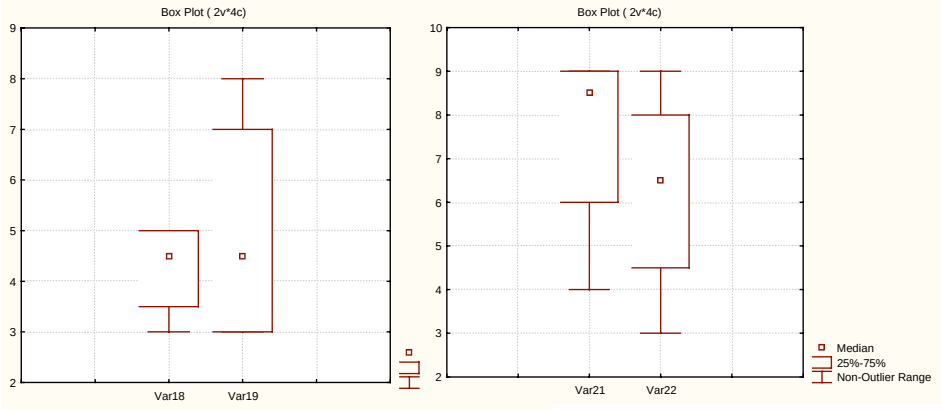

Fig. 4. Quality for co-located (Var 18) and distributed (Var 19) pairs in Round I; quality for co-located (Var 21) and distributed (Var 22) pairs in Round II.

of pair programming. A textual chat, for instance, is obstructive for the pair; the operations for using the chat disturb the continuity of work.

(2) The absence of an adequate collaboration support: switching roles avoid interrupting the rhythm of work. It requires that the observer can keep the control of the workstation whenever the driver cannot go on coding. Some desktop sharing tools suffer from technological limitations that make switching roles annoying.

The dismissal conjecture has been confirmed by subjects in a questionnaire-guided post-experiment assessment: while in the co-located round most of the pairs worked together for all the tasks, in the distributed round, after an initial time during which pairs tried to settle a common strategy of action, several among them tended to work as singleton developers. The initial roles became frozen, the switching was increasingly disregarded and finally only the driver developed the code whereas the observer looked at the companion working. Sometimes the observer attempted observations and suggestions that were neglected by the driver or often mismatched.

4. Experiment's Replica

The experiment's replica was aimed at testing the same hypotheses of the first experiment, while minimising the occurrence of the dismissal phenomenon. In order to limit the dismissal, we have followed two main policies. Firstly, we have performed a more intensive and focussed training to students: in addition to seminaries and lab exercises, students have been trained in working together and making faster decisions.

Secondly, the time for performing the tasks was sensitively reduced: it went from 180 minutes per round to 90 minutes per round. From the first experiment's assessment discussion we learnt that in the first period the distributed pairs strove

Table 6. Experimental design.

Subjects	Round I		Round II	
Group A (4 units)	Co-located	P_1	Distributed	P_2
Group B (4 units)	Distributed	P_1	Co-located	P_2

Table 7. Information on replica's programs.

Program	LOCs	# Classes	Description
AreaCalculating (P_1)	76	2	This program calculates the areas of plan geometry figures.
AverageNumber (P_2)	67	3	This program calculates some statistical values on a sample of numbers.

to work together. Then, given that the rhythm of work slowed down too much and that the communication and collaboration became too difficult to implement, they started to work alone. Our idea was to reduce the total amount of time available to the subjects, so as to gather data when distributed pairs were still trying to work together.

The definition of the replica is the same definition of the first experiment discussed in Sec. 2; therefore, only the characterization and operation will be discussed here.

4.1. *Characterization and operation*

The subjects were volunteer students of the Software Engineering Course in the fourth year of the *laurea* degree in Computer Engineering at the University of Naples "Federico II". They had to implement three maintenance requests on a C++ program. The first maintenance request was corrective, the other two were perfective ones.

Four pairs have been involved in the experiment's replica, organised in two groups (A and B). The experiment design is shown in Table 6. The experiment consisted of two rounds. In Round I group A's pairs were co-located and group B's pairs were distributed and they had to implement three maintenance requests to the program *AreaCalculating*. In Round II group A's pairs were distributed and group B's pairs were co-located and they had to implement other three maintenance requests to the program *AverageNumber*. Information about programs is reported in Table 7.

Subjects received the documentation discussed in Sec. 2.2. The main differences between the experiment and the replica were:

(1) The time available to accomplish the overall tasks was reduced to 90 minutes, in order to avoid the dismissal phenomenon;
(2) The number of subjects decreased to 8;
(3) NetMeeting was used for desktop sharing instead of VNC;

Table 8. Mann-Whitney U tests on effort and quality data for the second experiment.

	p-level	Description
Effort	0.083	Mann Whitney tests on effort data between co-located and distributed pairs.
Quality	*0.043*	Mann Whitney tests on quality data between co-located and distributed pairs.

Table 9. Descriptive statistics of the replica's sample.

	Co-located					Distributed				
	Dev. Stand	Avrg	Max	Min	Moda	Dev. Stand	Avrg	Max	Min	Moda
Round I										
Quality	0.7	8.5	9	8	Na	1.4	7	8	6	Na
Effort	1.4	46	47	45	Na	22.6	66	82	50	Na
Round II										
Quality	0	9	9	9	9	1.4	7	8	6	Na
Effort	16.3	41.5	53	30	Na	19.1	61.5	75	48	Na

(4) Dev-C++ was used as IDE for C++ programs, because subjects had experience in using it from previous projects.

4.2. *Replica's results*

In Table 8, the Mann Whitney U test results on quality and effort data are reported, because data was not normally distributed. The hypotheses were tested by fixing the p-level threshold value at 5%.

There is empirical evidence that the quality of pair programming is affected by distribution ($p = 0.043 < 0.05$), as the second row of Table 8 shows, whereas there is no empirical evidence that distribution affects effort.

Descriptive statistics of the sample are provided in Table 9.

In Fig. 5 the box plots for effort and quality are illustrated. The box plots of effort show that the performance of distributed pairs is worse than the performance of co-located pairs. The worst co-located pairs reached the same level of the best distributed ones. This result seems to contradict the dismissal conjecture. Actually, the replica was planned in order to reduce the dismissal phenomenon within the pair. In fact, the available time for accomplishing the tasks was reduced: during the observation time, subjects of distributed pairs worked as pairs while dealing with collaboration and communication problems.

Based on such considerations, the overall degradation of performance in distribution seems to be due to technological issues. The dismissal phenomenon has not yet emerged in the replica, but the negative side effects of inadequate communication and collaboration had. Such consideration appears clear when analyzing

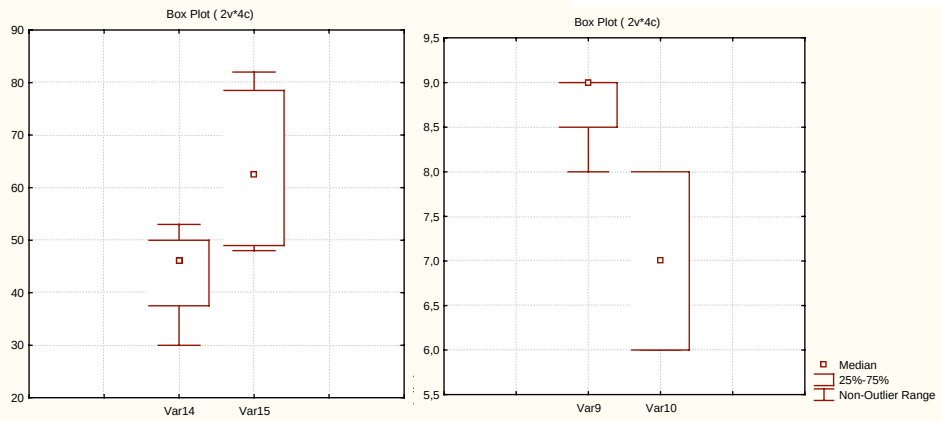

Fig. 5. Effort for co-located (Var 14) and distributed (Var 15) pairs; quality for co-located (Var 9) and distributed (Var 10) pairs.

quality data illustrated in Fig. 5. The median value is greater for co-located than for distributed pairs. Moreover, it should be observed that the worst quality level of co-located pairs is better than the best level of distributed ones.

The difference between the quality obtained from co-located and distributed pairs is greater than the one in the first experiment (see Fig. 4). We believe also that this was due to the reduction of available time. In the first experiment people tried to collaborate in the initial phase, but then started to work alone, ameliorating the quality of the work because they removed the continuous effort in establishing a pair fashion style of work.

During the replica, people worked as pairs while facing problems connected with the platform: the reviews were affected by such problems and consequently the quality was very low.

5. The Dismissal Phenomenon: Causes and Remedies

After the experiment and its replica, experimenters had a questionnaire-guided discussion with subjects, in order to accomplish a qualitative investigation on the experiment outcomes.

The most relevant result of the discussion was the confirmation of the dismissal conjecture, also issued in [26]. Together with the subjects, we strove to identify two main candidate causes for the phenomenon: the communication limit (we named it the **faulty phone cause**) and the divergence of approaches (we named it the **two-minds cause**). They are described in the following:

1. The **faulty phone cause**. As previously discussed, *communication* is a critical issue for implementing distributed pair programming successfully. Communication is important for performing contemporary review and decision-making within the pair. If the technology support does not satisfy completely the need

for comfortable communication, it could obstruct the driver while typing and the observer while inspecting the code. In our experience the text chat was not completely adequate to support distributed pair programming as it forced subjects to pay attention continuously to the chat window in order to get the companion's intervention, and so they had to take their eyes off the code frequently. After a while, people felt uncomfortable using this method of communication and switched to work alone on the code, ignoring messages from the pair's companion. This contributed to break distributed pair.

2. The **two-minds cause**. Each member of the pair brought a proper idea of strategies to meet the goals. Having the companion far from them discourages people to argue for their own ideas. The pair's components tend to assume an undisciplined behaviour and the roles are performed chaotically: the control of the machine is taken without the consensus of the companion and the reviews are neglected. In distributed pair programming, the collaborative work of the pair needs to be more disciplined than in co-located pair programming. It is necessary to train the subjects adequately.

The conducted assessment also suggested solutions in order to manage such issues adequately.

Behavioural protocol. Pair programming forces two people to share tools of work they use to consider strictly personal. This makes people resilient in assuming either an "observer" or "driver" behaviour completely, mainly because they do not know exactly which kinds of tasks are charged on the observer role and which ones on the driver role. Researchers and practitioners suggest to switch the two roles when necessary. The problem is that switching is less spontaneous in distributed settings if people are not adequately experienced with agile methods. In this case people tend to work mainly asynchronously on different tasks more than as a pair on the same task. People need adequate training to properly apply pair programming: the duties of the observer and those of the driver must be distinguished clearly. A behavioural protocol can be useful for people with scarce experience of pair programming.

Communication enabler. In order to support distributed pair work, communication means must implement a metaphor of the actual world. Effective platforms for distance communication must enable some sociological peculiar aspects of real life communication. In distributed pair programming, people need a communication means that owns at least two features: vocal communication and a blackboard. Vocal dialogue lets people communicate and keep on working on their task at the same time. Vocal dialogue helps people collaborate in a more realistic way than text-based chat or instant messaging. The latter devices force people to assume an unnatural behaviour and this obstructs the continuity of work. Defective communication is one of the candidate causes of pair dismissal (**faulty phone cause**). An interesting advise stemmed from subjects. Video-chat tools are neither required nor considered as useful. Blackboard can be exploited as a means to transmit graphs,

algorithm drafts, pictures as hints of design documentation's pieces.

Different experiences and capabilities. Our experimental subjects had a similar academic background. Differences among them consisted mainly of their academic curricula. Some of them had less experience as developers in firms, but not so much to determine a well defined gap with the others. The gaps in terms of capabilities within the pairs were too reduced for being useful but enough for leading the pair far away from co-ordination. The pair needs a component with more experience, who can act as a leader for the pair. Distribution magnifies critical situations: the leader figure becomes fundamental for successful task completion.

Change tracing and highlighting. The environment supporting distributed pair programming must keep track of the modifications realised by the two developers. It seems to be important for co-operative work to have an immediate idea of the place and author of a modification. Distribution emphasises because pair's components do not share physical space, e.g., they cannot use fingers to point the code under review. The platform can use different colours for referring to the different programmers. It should be useful to also keep track of the time of modifications. Overhead was due to the need for keeping in mind who has modified which part of code and when.

Awareness of the project. Distributed pairs tend to reduce discussions; as a consequence, the pair's components do not develop a common vision of the project: they consider different priorities for the goals of the project and different strategies to be adopted, different approaches to solve problems. Frequent rounds for knowledge leveraging must be properly planned during all phases of the project; they are recommended especially at the start up of the project. Some subjects proposed that one in the pair should have a greater awareness of the project than the other one. Such a person should assume mainly the role of the observer during the development phases.

6. Experimental Validity

This section presents a discussion of the threats considered as more relevant for the design of the experiment, referring to the classification in [13].

6.1. *Internal validity*

Maturation. The subjects were students not experienced with pair programming.

For both experiments, during the first round, the subjects acquired competence in the new practice, then exploited it in the second one. We used the Wilcoxon test between the rounds of the same group, in order to evaluate the significance of differences in performance and quality due to the maturation of groups between the two rounds.

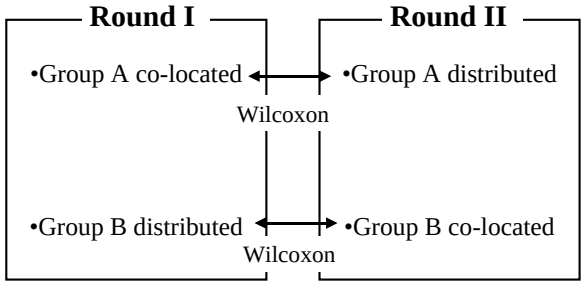

Fig. 6. Design of statistical test for maturity threat.

Table 10. Wilcoxon tests results in order to evaluate the maturity threat.

	p-level	Description
Effort Group A	0.465	Wilcoxon test on effort data of the Group A between Round I and II.
Effort Group B	0.715	Wilcoxon test on effort data of the Group B between Round I and II.
Quality Group A	0.345	Wilcoxon test on quality data of the Group A between Round I and II.
Quality Group B	0.969	Wilcoxon test on quality data of the Group B between Round I and II.

In Fig. 6 the design of tests are illustrated and in Table 10 the results are reported: there is no empirical evidence that maturation affects differences in each group's performance and quality between the two rounds ($p > 0.05$).

6.2. *Construct validity*

Mono Operation bias: The subjects were required to modify one program in each round, according to three specific maintenance requests. The difference of assignment in each round can affect the final results. In order to evaluate if such differences between rounds are statistically significant, we have used Wilcoxon tests. The tests were accomplished both for the first experiment and the replica (Fig. 7).

In Table 11 the tests results are reported: there is no empirical evidence that the assignment specifications affect quality and performances.

7. Conclusions

Several experiments have demonstrated the benefits of pair programming in terms of performances and quality. The distribution of software processes and teams is increasing within the industry. We have made an experiment and a replica in order to evaluate the impact of distribution on pair programming. Both the first experiment and the replica have produced empirical evidence that the quality of pair programming is affected by distribution.

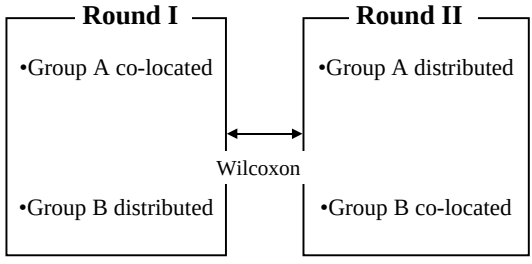

Fig. 7. Design of statistical test for mono-bias threat.

Table 11. Wilcoxon tests results in order to evaluate the mono-bias threat.

	p-level	Description
Effort first experiment	0.508	Wilcoxon test on effort data between Round I and II in the first experiment.
Quality first experiment	0.445	Wilcoxon test on quality data between Round I and II in the first experiment.
Effort replica	0.715	Wilcoxon test on effort data between Round I and II in the replica.
Quality replica	0.109	Wilcoxon test on quality data between Round I and II in the replica.

In the first experiment, the dismissal phenomenon emerged: if the technological platform does not support adequately communication and collaboration, the distributed pair working gets interrupted and just one of the pair's components keeps control of the workstation, neglecting the review and the switch requests from the remote companion. This entails the lost of benefits in terms of performance and quality, critical to pair programming.

Such phenomenon is a factor of risk and should be properly managed when planning the implementation of distributed pair programming within a process activity.

The replica was planned in order to minimise the pair dismissal phenomenon. The data collected from the replica reflected the actual behaviour of distributed pairs while facing communication and collaboration problems and striving to work together.

In the first experiment the dismissal phenomenon played a central role in the definition of the final results. In the replica, the dismissal was limited. The replica's results about quality make us believe that the support platform is the candidate factor for maintaining unchanged quality and performance when distributing pair programming.

Both the replica and the experiment offered the following main outcomes.

- ***Empirical evidence that the distribution affects pair programming quality.*** Some factors of distribution settings make the quality of pair program-

ming lower. The quality assessment of the experiment suggested that such factors have to be searched in the infrastructure of collaboration. Communication has to be fluent and neither obstructive for the driver nor the observer. On the contrary, reviewing code and discussing a common strategy require additional effort to be accomplished successfully.

- ***No empirical evidence that effort increases when distributing pair programming.*** Although the dismissal phenomenon favoured a higher expense of time in co-located tasks, the differences are not statistically significant. The qualitative analysis confirmed that the time can be reduced with distribution. The motivation for that is not encouraging: this is due to the breaking down of collaboration. Finally, it is only a waste of resources: two programmers are paid whereas only one works, without benefits of contemporary reviews.
- ***Some candidate factors determining the success of pair programming.*** We have identified them in the selection of an appropriate communication and collaboration support.

The experiment was executed in academic settings. Such kind of experiments helps to fix bugs within the experimental design, before executing it in industrial setting. As a matter of fact, the phenomenon of dismissal was noticed only after the first round of the experiment, and not foreseen during the design of the experiment.

Furthermore, experiments with students help to point out which are the likely findings that can be interesting for industry, in order to propose appealing investigations and gain maximum collaboration from professionals.

A strong limitation of the experiment is its size: the samples are small, the time for observation is short, the size of the problem is scarcely significant if compared with marketplace applications. Such limitations can be accepted by considering the experiment as a preliminary investigation on distributed pair programming. The aim is to define the most suitable design for executing the experiment in industrial setting.

From the experiments the following research questions emerged:

(1) ***Does an appropriate platform let the distributed pair programming remain beneficial as well as the co-located?*** From the post-experiment assessment discussion, one major reason for the dismissal of the pair and, consequently, of the deterioration of pair programming effectiveness, is the lack of an appropriate platform. Such platform should comprise at least: an audio channel as support for communication, a system to exchange/share images and drafts, a versioning control assisting continuous reviews. This suggests that an *ad-hoc* system for distributed pair programming would be helpful.

(2) ***What is the best combination of the pairs in terms of competence, experience, and character profile in distribution?*** It seems that knowledge and behavioural aspects of individuals are critical for the success of pair programming. All the subjects have highlighted that these aspects have a great impact on the practice. It should be very interesting to have empirical

evidence of such relationships. Moreover, it should be useful to understand how to properly manage such factors in forming the pair. If this issue is important for co-located pair programming, it becomes critical for distributed pair programming, where the implementation of the practice is obstructed by other kinds of problems. For instance, the difference of culture and habits can become further hurdles to the success of pair programming.

(3) ***Is distributed pair programming only a need or can it fit certain business targets better than co-located pair programming?*** Till now distribution is considered a need which arose from the widespread diffusion of pair programming and global software development. Moreover, maintaining the components physically detached can be beneficial for pair programming in specific contexts. The switching of role should happen in a more disciplined manner. The pair can exploit resources that are placed in two different organisations, and govern them directly. Pair programming can be used for merging people with very complementary competencies and located in two different places. Investigating when and how distribution can improve the practice of pair programming should present interesting findings.

Acknowledgements

We would like to thank the Software Engineering course students from Naples Federico II and Sannio Universities for having taken part in the experiments and for their useful observations in the assessment discussion. We would like to thank Orazio Abissinia, Teresa Daniela Mallardo, and Porfirio Tramontana for their useful support in preparing the experiment environment and for their assistance in running the experiment.

References

1. AA. VV, *Manifesto for Agile Software Development*, http://agilemanifesto.com (accessed on 27 June 2005).
2. L. Williams, *What is Pair Programming?*, http://pairprogramming.com (accessed on 27 June 2005).
3. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Reading, MA, 2000.
4. D. Wells, *Pair Programming*, http://www.extremeprogramming.org/rules/pair.html (accessed on 27 June 2005).
5. A. Cockburn and L. Williams, The costs and benefits of pair programming in *Extreme Programming Examined*, eds. G. Succi and M. Marchesi, Addison-Wesley-Longman, Boston, MA, 2001, pp. 223–243.
6. L. Williams, W. Cunningham, R. Jeffries, and R. R. Kessler, Straightening the case for pair programming, *IEEE Software* **17**(4) (2000) 19–25.
7. M. M. Muller and W. F. Tichy, Case study: Extreme Programming in a University Environment, in *Proc. 23rd International Conference on Software Engineering* (*ICSE 2001*), Toronto, Canada, IEEE CS Press, 2001, pp. 537–544.

8. C. Ebert and P. de Neve, Surviving global software development, *IEEE Software* **18**(2) (2001) 62–69.
9. E. Carmel and R. Agarwal, Tactical approaches for alleviating distance in global software development, *IEEE Software* **18**(2) (2001) 22–29.
10. M. C. Paulk, Extreme programming from a CMM perspective, *IEEE Software* **18**(6) (2001) 19–26.
11. J. Grenning, Launching extreme programming at a process-intensive company, *IEEE Software* **18**(6) (2001) 27–33.
12. P. Baheti, E. Gehringer, and D. Stotts, Exploring the efficacy of distributed pair programming, in *Proc. Extreme Programming and Agile Methods — XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference*, Chicago, IL, LNCS, Springer-Verlag, 2002, pp. 208–220.
13. C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, Boston, 1999.
14. B. Boehm, Get ready for Agile Methods, with care, *Computer* **32**(1) (2002) 64–69.
15. M. Lippert, P. Becker-Pechau, H. Greitling, J. Koch, A. Kornstadt, S. Roock, A. Schmolitzky, H. Wolf, and H. Zullinghoven, Developing complex projects using XP with extension, *Computer* **36**(6) (2003) 67–73.
16. O. Murru, R. Deias, and G. Mugheddu, Assessing XP at a European internet company, *IEEE Software* **20**(3) (2003) 37–43.
17. D. Wells, *Extreme Programming: A gentle introduction*, http://www.extremeprogramming.org (accessed on 27 June 2005).
18. H. Miller and J. Sanders, Scoping the Global Market: Size is just part of the story, *IEEE IT Pro* **1**(3) (1999) 49–54.
19. P. Fuglewicz, Global Software Development: Attainable challenge or the holy trail?, *Cutter It Journal* **12**(3) (1999) 22–26.
20. J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter, An empirical study on Global Software Development: Distance and speed, in *Proc. 23rd Int. Conf. on Software Engineering* (*ICSE 2001*), Toronto, Canada, IEEE CS Press, 2001, pp. 81–90.
21. D. Chaffey, *Groupware, Workflow and Intranets*, Digital Press, Boston, MA, 1998.
22. J. D. Herbsleb, and R. E. Grinter, Architecture, coordination, and distance: Conway's law and beyond, *IEEE Software* **16**(5) (1999) 63–70.
23. R. W. Jensen, A pair programming experience, *CrossTalk*, March 2003, http://www.stsc.hill.af.mil/crosstalk/2003/03/jensen.html (accessed on 14 July 2005).
24. AA. VV., *Pair Programming is Done by Peers*, http://www.c2.com/cgi/wiki?PairProgrammingIsDoneByPeers (accessed on 27 June 2005).
25. W. J. Orlikowski, Knowing in practice: Enacting a collective capability in distributed organizing, *Organization Science* **13**(3) (2002) 249–273.
26. G. Canfora, A. Cimitile, and C. A. Visaggio, Lessons learned about distributed pair programming: What are the knowledge needs to address? in *Proc. IEEE Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises* (*WETICE-2003*), Linz, Austria, IEEE CS Press, 2003, pp. 314–319.
27. N. Nohria and R. Eccles, *Networks and Organizations: Structure, Form, and Action*, Harvard Business School Press, Cambridge, MA, 1992.
28. C. McDowell, B. Hanks, and L. Werner, Experimenting with pair programming in the classroom, in *Proc. 8th Annual Conf. on Innovation and Technology in Computer Science and Education 2003*, Thessaloniki, Greece, ACM Press, 2002, pp. 60–64.

29. E. F. Gehringer, A pair-programming experiment in a non-programming course, in *Proc. Companion of the 18th Annual ACM SIGPLAN Conference on Object Oriented 2003*, Anaheim, CA, ACM Press, 2003, pp. 187–190.
30. L. Williams, C. McDowell, N. Nagappan, J. Fernald, and L. Werner, Building pair programming knowledge through a family of experiments, in *Proc. 2003 Int. Symp. on Empirical Software Engineering* (*ISESE 2003*), Rome, Italy, IEEE CS Press, 2003, pp. 143–152.
31. K. M. Lui and K. C. C. Chan, When does a pair outperform two individuals?, in *Proc. Fourth Int. Conf. on eXtreme Programming and Agile Processes in Software Engineering* (*XP 2003*), Genova, Italy, LNCS, Springer-Verlag, 2003, pp. 225–233.
32. J. T. Nosek, The case of collaborative programming, *Communications of ACM* **41**(3) (1998) 105–108.
33. B. F. Hanks, Distributed pair programming: An empirical study, in *Proc. Extreme Programming and Agile Methods — XP/Agile Universe 2004: 4th Conference on Extreme Programming and Agile Methods*, Calgary, Canada, LNCS, Springer-Verlag, 2004, p. 81–91.
34. L. Williams, A. Sukhia, and A. J. Anton, An initial exploration of the relationship between pair programming and Brooks' law, in *Proc. Agile Development Conference* (*ADC'04*), Salt Lake City, Utah, IEEE CS Press, 2004, pp. 11–20.
35. M. M. Muller and F. Padberg, An empirical study about the feelgood factor in pair programming, in *Proc. 10th Int. Symp. on Software Metrics* (*METRICS'04*), Chicago, Illinois, IEEE CS Press, 2004, pp. 151–158.