Pair Programming vs. Side-by-Side Programming*

Jerzy R. Nawrocki, Michał Jasiński, Łukasz Olek, and Barbara Lange

Poznan University of Technology, ul. Piotrowo 3a, 60-965 Poznan, Poland
{Jerzy.Nawrocki, Michal.Jasinski, Lukasz.Olek,
Barbara.Lange}@cs.put.poznan.pl
http://www.cs.put.poznan.pl

Abstract. In agile methodologies communication between programmers is very important. Some of them (e.g. XP or Crystal Clear) recommend pair programming. There are two styles of pair programming: XP-like and side-by-side (the latter comes from Crystal Clear). In the paper an experiment is described that aimed at comparison of those two styles. The subjects were 25 students of Computer Science of 4th and 5th year of study. They worked for 6 days at the university (in a controlled environment) programming web-based applications with Java, Eclipse, MySQL, and Tomcat. The results obtained indicate that side-by-side programming is a very interesting alternative to XP-like pair programming mainly due to less effort overhead (in the experiment the effort overhead for side-by-side programming was as small as 20%, while for XP it was about 50%).

1 Introduction

In classical approach to software development, a programming task (e.g. writing a software module to a given specification) is assigned to one programmer (see e.g. [9]). To assure quality all the production code should go through a peer-review process (it can be inspection, walkthrough, formal technical review etc. [16]).

Kent Beck, the creator of Extreme Programming (XP for short), has introduced to his methodology a different approach called pair programming [2]. In pair programming, as the name suggests, a task is assigned to a pair of programmers who are equipped with one computer. While one programmer is writing a piece of code, the other is watching, asking some questions, and proposing test cases (that provides so-called continuous review).

The efficiency of pair programming has been studied by many researchers. The first experiment concerning pair programming has been described by John Nosek [14]. He reported that pairs required about 30% less time than individuals but effort associated with pair programming was by 40% greater. Perhaps the most optimistic results have been obtained by Laurie Williams [21, 22]. According to her experiments the speedup accomplished by pair programming was at the level of 40% and the effort overhead was as small as 20%. The results of first experiments with pair

* This work has been financially supported by the State Committee for Scientific Research as a research grant 4 T11F 001 23 (years 2002-2005).

I. Richardson et al. (Eds.): EuroSPI 2005, LNCS 3792, pp. 28 – 38, 2005.

programming performed at the Poznan University of Technology [13] were more pessimistic. The speedup gained by pair programming was at the level of 20% and the effort was about 60% higher than for individuals.

Just recently another version of "programming in pairs" has been proposed by Alistair Cockburn [7]. It is called side-by-side programming (SbS). In SbS a task is assigned to a pair of programmers, and each programmer has his own computer. That allows them to split the task into subtasks and work on each subtask individually (similarly to the classical approach). The main difference between SbS and the classical approach is physical proximity of the pair members (which enhances communication) and unity of the goal (SbS programmers are working on the same task and they both are responsible for it). Unfortunately, so far there are no experiments comparing efficiency of SbS and pair programming.

The objective of the paper is to present an experiment aiming at comparison of SbS and pair programming. In Sec. 2 methodological aspects concerning programming experiments are discussed. Next the experiment is described (Sec. 3) and its results are presented (Sec. 4). An important issue concerning pair programming experiments is involvement of pair members in the development process (there is a danger that actually only one person will write and understand the code while the other person will be only a spectator). This aspect is discussed in Sec. 5. We have also asked our programmers a few questions concerning their impression and the results are presented in Sec. 6.

2 Methodological Aspects of Programming Experiments

In the software engineering community there is an increasing understanding that experimental research is needed in order to explain and improve software development processes [20, 1]. An experiment tests theoretical predictions against reality and is defined as a form of empirical study where the researcher has control over the independent variables being studied. It is carried out under controlled conditions in order to test a hypothesis against observation [1].

Experimentation in software engineering is difficult, because the process of developing software involves both technical aspects and human factors. Especially, psychological factors, which are less predictable, are a challenge for experimenters. For that reason, software engineering experimentalists should adopt well developed and elaborated methodological techniques used in the behavioural disciplines. Some issues concerning studying programmer behaviour experimentally has been discussed as long as 20 years ago [3, 19]. It was pointed out, that behavioural researchers in computer science must pay close attention to methodological issues. Some aspects of a programming experiment are presented in the next paragraphs.

Representative Sample

One of main problems in such empirical studies is *selection of appropriate subjects* for the experiment. To make a statement about the behaviour which will be true for the whole population, the experimenter must ensue that the subjects are representative. That means that subjects should be chosen randomly from the whole population of programmers. If this condition is not fulfilled, then such a type of research is called by behavioural scientists a *quasi-experiment*. Unfortunately, the results of a quasi-experiment cannot be extended to the whole population [4]. Every experiment we know about in the area of pair programming is in fact a quasi-experiment [21, 14]. First, because it is difficult to characterize the population of programmers. We do not know of any research that would aim at characterizing distribution of programmers' age, sex, experience in software development, technology used at work (including programming languages) etc. Secondly, conducting sensibly long experiments (a week or longer) on a large enough and diverse group of professionals (according to standards used in sociology and political sciences it should be hundreds of people) would be very expensive and difficult (we have observed a 'contest syndrome': it is easier to attract fast programmers than slow ones, so there is a danger that the sample will not be representative).

Due to these limits many experimentalists decide to conduct experimental research using Computer Science students as the subjects [13, 21]. Obviously, *students are not professionals*, but their way of thinking and their academic background is similar to professional programmers. Due to our observations most of programmers are young people (below 35). Moreover, many 4th and 5th year students have part-time jobs at software companies. Therefore, using Computer Science students, especially students of the 4th and 5th year, seems acceptable.

Another issue is *selection of programming assignments* for an experiment. Programming tasks should be a representative sample of some wider class of programs similar to those written by professionals at work. Especially, they should be of an appropriate level of difficulty and length to produce data with desirable statistical characteristics [3]. For instance, very short assignments or difficult algorithmic 'puzzles' used in the ACM programming contests (although very interesting) seem not representative.

Statistical Significance

An important methodological aspect of experimentation is statistical analysis of collected data. Usually a null hypothesis H_0 along with an alternative hypothesis H_1 is formed and an appropriate test statistic is computed. An obtained value of the test statistic is compared with a 'threshold' and H_0 is rejected or not. The probability of rejecting H_0 when H_0 is true is called a level of significance. The lower the level of significance is, the greater the confidence in statistical analysis. In behavioural sciences the standard value of level of significance is 0.05. In the case of programming studies the level of significance is sometimes as big as 0.20 [19]. A good approach is to present a *P-value* which is the smallest level of significance that would lead to rejection of the null hypothesis H_0 [12].

When performing statistical analysis, it is necessary to check if distribution of a dependent variable is *normal*. If the distribution is not normal another (much more difficult) statistical analysis should be conducted. Unfortunately, many reports from programming experiments do not mention this (see e.g. [21, 14]).

Fairness

Human factor in programming experiments is very important. Programmers can differ in programming speed as 1:10 or even more (see e.g. [8, 17]). When comparing two (or more) different programming techniques one has to ensure that average programming speed (and variance) in each group are similar. A good solution is to make a pre-test before an experiment, to estimate a programming speed for each subject. That is not easy and sometimes other simpler methods (e.g. questionnaires about years of experience [15]) are used.

Documentation of an Experiment

A scientific experiment should be replicable. Thus, a good documentation of the process is necessary. It should describe used materials, design, procedure and scoring [4]. In the case of research on programmer behaviour, a description of programming tasks and test cases should also be included. In addition, a measurement procedure (e.g. how the finish time is defined) should be presented. Unfortunately, some reports lack this information [11].

Controlled Conditions

To ensure credibility of a programming experiment one has to carry it out in a controlled environment to minimise the influence of external factors that could impact the results. However, some researchers conduct experiments in an uncontrolled environment, e.g. they let the subjects to do their programming assignments at home.

3 Experiment Description

3.1 Subjects and Environment

The experiment was run at the Poznan University of Technology from February till April 2005. The subjects were 30 volunteers: students of master degree programs in Software Engineering, and in Database Systems (4th and 5th year of study). They had completed various programming courses (including Java and web-based applications) amounting to over 400 hours. They have also participated in a 1-year-long university project playing the roles of programmers.

The subjects worked in a few open-space laboratory rooms under supervision of research assistants. They were equipped with PCs (Pentium IV, 512 MB RAM). The programming environment consisted of Java J2SDK 1.4.2_06, Tomcat 5.0.18, Eclipse 3.1, MySQL 4.0.21 database server, and a CVS code repository server.

3.2 Process

The process consisted of five phases: Homework, Preparation, Selection, Warming-up and Run. Each time the subjects were to solve one or more programming assignments.

In the *Homework* phase the subjects were given a programming assignment concerning the technology and tools used in the subsequent phases (Java, Tomcat, Eclipse etc.). They worked individually at home. The aim of the phase was to allow them to learn (or re-call) the technology and tools.

During the *Preparation* phase the subjects worked individually at the university in the actual experimental environment. They were supervised by research assistants who also did the quality assurance: acceptance testing. The subjects worked until successful completion of the assignment (all the acceptance tests passed). There were no additional quality factors introduced, but the acceptance tests. The main aim was to make sure that the subjects know the tools and technology. Moreover, it gave the subjects a chance to get familiar with the process (acceptance testing, time measureement etc.). The preparation phase took one day. On the average the assignment completion time was 522minutes.

The *Selection* phase took another day. The subjects worked individually at the university on programming assignments. We have measured the time that elapsed from beginning of the experiment till successful completion of the task (all acceptance tests passed). In this case, average completion time was 370 minutes. We used the collected data to split the subjects into three groups: Pairs1, Pairs2, and Individuals. In the next phases of the experiment the Pairs1 and Pairs2 groups were to do pair programming (using different programming styles) and the Individuals group contained individual programmers serving as a reference group. It is well known that people differ in programming speed significantly (see e.g. [8, 17]). Therefore, to be able compare results concerning different programming styles we had to ensure that all the groups will be (on average) equally fast. Thus, the following problem arose: how to partition $5n$ subjects of known completion time into Pairs1 subset ($2n$ elements), Pairs2 subset ($2n$ elements) and Individuals subset (n elements) in such a way that average completion time of each subset is (almost) the same. Of the 30 initial volunteers 25 passed successfully the Selection phase and we split them into three abovementioned groups with n=5 (we have used three different heuristics and chosen the best result).

We wanted to compare XP-like pair programming (1 computer per pair) to side-by-side (SbS) programming (2 computers per pair). The aim of the *Warming-up* phase was to make sure that the subjects know how to do XP or SbS pair programming. The phase took two days. Each day started with a 20-minutes-long training based on a process miniature [6] (the subjects were solving programming puzzles). After the training session the subjects were developing a web application (that took the rest of the day). On the first day of Warming-up one group of pairs was following XP-like pair programming and the other one SbS-like pair programming. On the second day the groups switched. The average assignment completion time was 323 and 370 minutes respectively.

The most important was the *Run* phase. It took last two days. The aim was to collect data that would allow to compare XP-like and SbS-like pair programming from the point of view of development time and effort (meaning completion time for individuals and double completion time in case of pairs).The subjects worked like in the Warming-up phase but without training sessions. Again on the first day the Pairs1 group was doing XP-like pair programming and on the second day they switched to SbS (Pairs2 did the opposite). That way we have mitigated the risk of unbalanced groups of pairs (we did our best during the Selection phase to have balanced groups but development time is a random variable and you never can be sure). During the *Run* phase, the average assignment completion time was 335 minutes on the first, and 491 minutes on the second day of the experiment.

3.3 Programming Assignments

More and more programmers are working on web applications. For that reason we have decided that our subjects will be working on Java-based web applications in JSP and Java servlet technology.

During the Preparation phase the subjects were to implement a password protected web site with security data stored on a database server. They had to implement the login procedure together with basic user management services (displaying list of users, adding and removing a user).

The Selection phase was based on two assignments. The first one was a simple document management application. Documents could be public or internal. Everybody could browse a public document, but only registered users could access internal documents (after passing the authorization control). The second assignment was a web application that would collect submitted abstracts and papers.

During the Warming-up and Run phases (4 days) the subjects incrementally developed a conference management system called Papers-Online (it was an extension of the second assignment from the Selection phase). Papers-Online has the following actors: authors, reviewers and a conference chairman. Authors can submit abstracts and papers. The chairman can register reviewers and assign papers to them. Reviewers can submit their reviews. Authors can view status of their papers. The chairman can set a conference program.

Each assignment contained use-cases [5] describing required functionality and a number of acceptance test-cases derived from them. More information can be found in [10].

4 Completion Time and Effort Analysis

Completion Time Analysis

The programming assignments were accompanied by acceptance tests. Each day we have measured *completion time* i.e. the time that elapsed from the beginning of programming session till successful completion (all the acceptance tests passed). We subtracted from that amount the duration of lunch break.

The Run phase took two days. As we have already mentioned, to mitigate the risk of unbalanced assignment of subjects to groups Pairs1 and Pairs2 (although we did it as carefully as possible – see Sec. 3.2) we decided that both Pairs1 and Pairs2 will do XP-like and SbS-like pair programming. A simple solution would be to have both groups do XP-like programming on the first day and SbS-like programming on the second day. But there would be a problem with a programming assignment. If the programming assignment was the same, then on the second day they would have just to repeat what they did on the first day and that would give fault results (SbS would seem more effective than it really is). If the programming assignments were different and, for instance, the first one was somehow easier than the second one, than XP would be privileged and one would come to false conclusions. To overcome that difficulty we decided that on the first day Pairs1 will be doing XP-like pair programming and Pairs2 will follow SbS. On the second day we gave them new assignments and Pairs1 was doing SbS while Pairs2 was following XP. Since the assignment used on the first day was different from that one used on the second day, we decided to use in our calculations *relative completion time*, i.e. the ratio of time used by pairs to the average time used by individuals (if the assignment used on the first day was by 30% more time consuming than the assignment used on the second day , than the average completion time for individuals was also by 30% greater for the first day than for the second day and that would compensate longer completion times for the first day).

The average values of relative completion times for SbS, XP, and individuals are shown in **Fig. 1** (obviously, the average relative completion time for individuals is 1). As the chart suggests, SbS is faster than XP and this observation is statistically significant with significance level equal to 0.15 (i.e. the probability of accepting the hypothesis that SbS is faster while it is not equals 0.15). Another observation saying that SbS and XP are faster than individual programming is statistically significant with significance level 0.05.

The above statistical analysis is based on the assumption that analyzed data are normally distributed (or reasonably close to normal distribution). Thus, one has to check if the collected data satisfy that condition. For that purpose we used the Shapiro-Wilk (SW) test [18]. The test confirmed that both raw completion times and relative completion times are normally distributed for all the programming styles, i.e. XP, SbS and individual (the confidence level is 0.05).

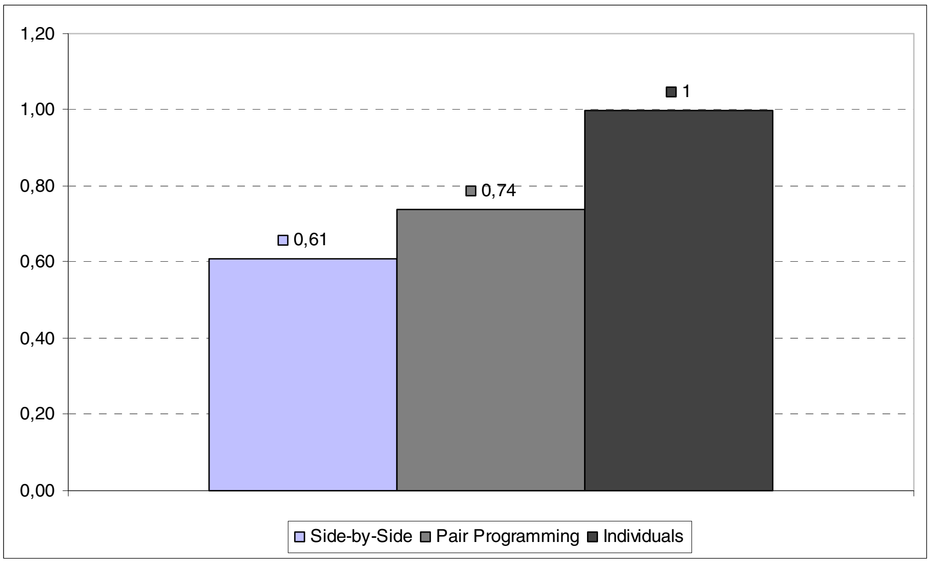

Fig. 1. Average relative completion time for individuals, XP pairs, and side-by-side pairs. Effort analysis.

For individuals *effort* equals completion time. In case of pairs the effort equals double completion time. For the sake of the reasons described in the first part of this section we are using *relative effort*, i.e. the ratio of effort for a given programming style to the average effort of individuals. The average relative effort for XP, SbS and

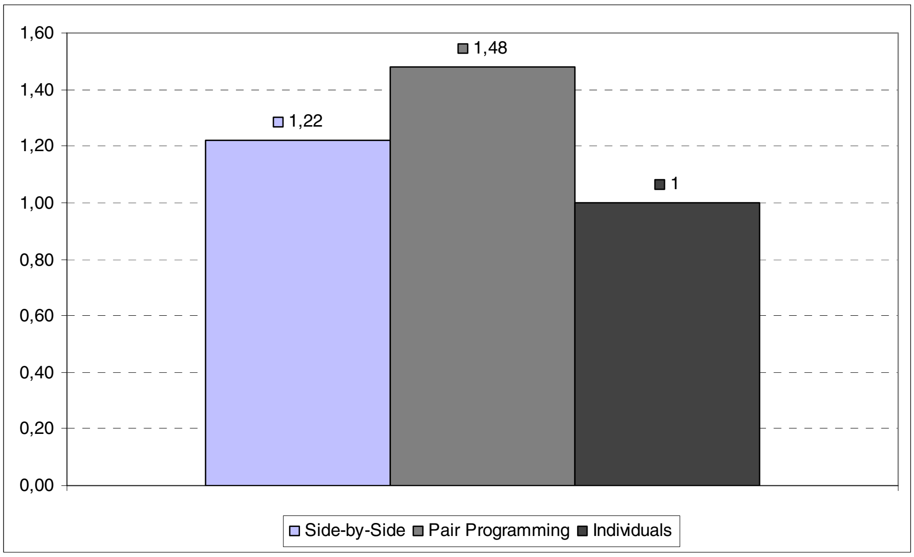

Fig. 2. Average relative effort for individual programmers, SbS and XP pairs

individuals is shown in Fig. 2. As the figure suggests, the effort for XP is greater than for SbS and this is statistically significant with significance level 0.15.

5 Familiarity-with-Code Analysis

In XP two programmers work all the time together: while one is writing code the other is doing continuous (on-the-fly) inspection. In SbS a pair has two computers and the partners can work on different tasks. Thus, a question arises how it does influence familiarity of the partners with all the code. To check it we decided to introduce a postmortem step which was performed at the end of each day. During postmortem step all the subjects were given a micro-assignment: they had to implement (individually) a small change request (the change request was the same for all the subjects). We measured the time required to complete the task. Relative completion time is the ratio of individual completion time (also for members of Pairs1 and Pairs2 – in the postmortem step they worked individually) to the average completion time for members of the Ind group. In Fig. 3 average value of relative completion time for XP, SbS and individual programming is presented (here "individual programming" refers to programming of the main assignment, not post-mortem). As the figure suggests, for SbS the code understanding is worse than for XP (in terms of average time required to implement a change). However, the smallest significance level at which one can assume this hypothesis is 0.25 (it is relatively high, so the hypothesis is rather weak). Figure 3 suggests that completion time for XP is greater than for individuals – that means that familiarity with code for XP is less than for individual programming. However, this hypothesis is not statistically significant.

We have also checked normality of the distribution of completion time for all the programmers performing postmortem step (we used the Shapiro-Wilk test). The completion time data are normally distributed with significance level equal to 0.10.

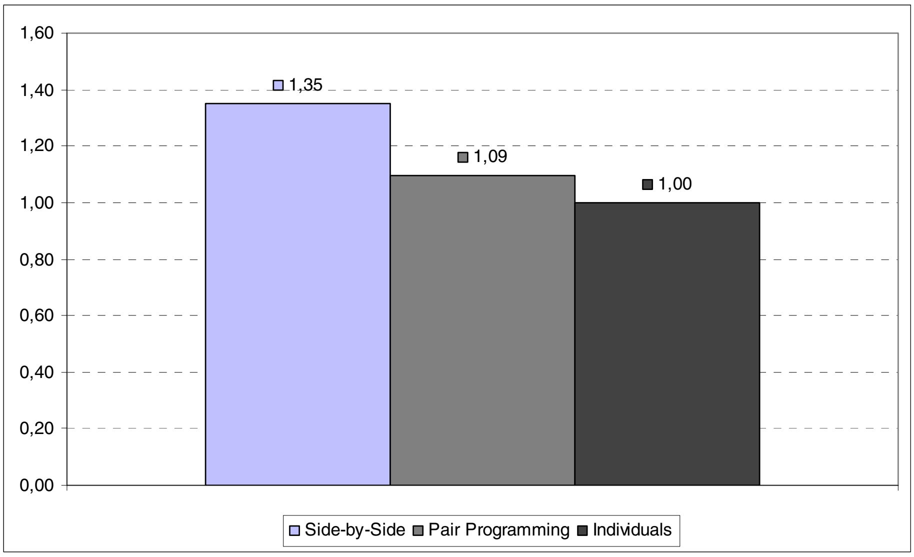

Fig. 3. Average relative completion time of the postmortem task (small change request) for individuals using different styles (XP, SbS or individual programming) for the main task

6 Participants Impression

After the experiment we have conducted a survey and asked the subjects a few questions about their impression on the programming styles. 55% of the subjects preferred collaborative programming (SbS or XP approach) to individual; while 40% had the opposite opinion (5% had mixed feelings). Of those 55% the Side-by-Side approach was preferred by 70% of the subjects, and XP by 30%.

The communication in SbS pairs was considered positive (very good or good enough) by 95% of the subjects.

48% of the subjects working in pairs were satisfied with their own code and 36% was unsatisfied. As regards partner's code, 45% were satisfied and another 45% had the opposite opinion. Since all the pair members were using both XP and SbS we do not know if this confidence (or lack of confidence) in code was greater for XP or for SbS.

7 Conclusions

From the described experiment it follows that side-by-side programming (SbS) is an interesting alternative to XP-like pair programming. Completion time for SbS was at the level of 60% compared with individual programming, what means that the effort overhead for SbS is as small as 20% (in some earlier experiments the effort overhead associated with XP-like pair programming was as big as 60% and in this experiment it was at the level of 50%). However, the effort of individual code maintenance for SbS was about 20% greater than for XP what indicates that knowledge about code is spreading slower for SbS than for XP. As regards personal impression, only 55% of the subjects preferred pair programming (SbS or XP) to individual one. Among them 70% was for SbS and only 30% for XP-like pair programming.

In further studies we shall focus on guidelines for applying particular software development approach. The goal is to deliver a framework for project managers and software developers helping to choose the right team organization the right software project, depending on importance of such factors like: completion time, effort or solution's quality.

References

1. Basili, V. E., Lanubile, F.: Building Knowledge through Families of Experiments. IEEE Transactions on Software Engineering, Volume 25, No. 4 (1999) 456–473.
2. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional (1999).
3. Brooks, R. E.: Studying programmer behavior experimentally: the problems of proper methodology. Communications of the ACM, Volume 23, No. 4 (1980) 207–213.
4. Brzeziński, J.: Metodologia badań psychologicznych. Wydawnictwo Naukowe PWN (2004).
5. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley (2000).
6. Cockburn, A.: Agile Software Development. Addison-Wesley (2002).
7. Cockburn, A.: Crystal Clear. A Human-Powered Methodology for Small Teams. Addison-Wesley (2005).
8. Dickey, T. F. Programmer variability. Proceedings of the IEEE, Volume 69, No. 7 (1981) 844–845.
9. Humphrey, W.: A Discipline for Software Engineering. Addison-Wesley, Reading MA (1995).
10. Laboratory of Software Engineering. http://www.se.cs.put.poznan.pl/en/content/research/experiments/experiments.html (2005).
11. Lui, K. M., Chan, K. C. C.: When Does a Pair Outperform Two Individuals? Lecture Notes in Computer Science , Volume 2675 (2003) 225–233.
12. Montgomery, D. C.: Introduction to Statistical Quality Control. Third Edition. John Wiley & Sons, Inc. (1997).
13. Nawrocki, J., Wojciechowski A. : Experimental Evaluation of Pair Programming. In: Maxwell, K., Oligny, S., Kusters, R., van Veenedaal E. (eds.): Project Control: Satisfying the Customer. Proceedings of the 12th European Software Control and Metrics Conference. Shaker Publishing, London (2001) 269–276.
14. Nosek J. T.: The Case for Collaborative Programming. Communications of the ACM, Volume 41, No. 3 (1998) 105–108.
15. Padberg, F., Mueller, M.: An Empirical study about the Feelgood Factor in Pair Programming. In: Proceedings of the 10th International Symposium on Software Metrics METRICS 2004, IEEE Press (2004).
16. Pressman, R. S.: Software Engineering: A Practitioner's Approach. Fifth Edition. McGraw-Hill (2001).

17. Sackman, H., Erikson, W. J., Grant, E. E.: Exploratory Experimental Studies Comparing Online and Offline Programming Performance. Communications of ACM, Volume 11, No. 1 (1968) 3–11.
18. Shapiro, S.S., Wilk, M.B. An analysis of variance test for normality (complete samples). Biometrika. 52, 3 and 4 (1965) 591– 611.
19. Sheil, B. A.: The Psychological Study of Programming. ACM Computing Surveys, Volume 13, No. 1 (1981) 101–120.
20. Tichy, W.F.: Should Computer Scientists Experiment More? IEEE Computer, Volume 31, No. 5 (1998) 32–40.
21. Williams, L.: The Collaborative Software Process. PhD Dissertation at Department of Computer Science, University of Utah, Salt Lake City (2000).
22. Williams, L. at al.: Strengthening the Case for Pair Programming. IEEE Software, Volume 17, No. 4 (2000) 19–25.