

Experimental Evaluation of Pair Programming¹

Jerzy Nawrocki, Adam Wojciechowski

Abstract

Pair programming is a kind of collaborative programming where two people are working simultaneously on the same programming task. It is one of the key practices of eXtreme Programming. In the paper we compare it with two variants of individual programming: one of them is based on Personal Software Process that has been proposed by W. Humphrey, and the other is a variant of eXtreme Programming tailored to individuals. Four experiments are described that has been performed at the Poznan University of Technology. During those experiments 21 students wrote 4 C/C++ programs ranging from 150 to 400 LOC. The obtained results are compared with the results of similar experiments described by J.T. Nosek and L. Williams et al.

1. Introduction

Pair programming is programming done by a pair of programmers. While one of them is creating a software artefact (e.g. a UML diagram or C++ code) the other is continuously assuring quality, i.e. he/she is watching, trying to understand, asking questions, looking for alternative approaches, helping to avoid defects. Two programmers are given one computer and they have to share it. After some time they are switching the roles: creator becomes quality assurer and vice versa. Since 1995 pair programming is gaining more and more popularity. It is one of the twelve practices of eXtreme Programming (XP for short) [1].

Despite its growing popularity, we still know very little about pair programming in sense of reliable experimental data. We know of only two experiments concerning pair programming. The first one was reported by John Nosek [5]. He describes an experiment in which 15 full-time system programmers from a program trading firm were asked to write a script that performs a database consistency check. The programmers were split into 5 pairs and a control group consisted of 5 individuals. Five measures have been considered. Four of them were subjective grades: two of them given by programmers (confidence about the solution and enjoyment of the process) and two other given by two independent graders (readability and functionality). The fifth measure was an objective one: it was time spent on writing the script. The average time for completion was 42 minutes for individuals and 30 minutes for pairs. So, pairs needed only 71% of the completion time necessary in case of individuals. However, in total, they consumed 43% more time than individuals. Unfortunately, that experiment concerned only one very short programming assignment (the allotted time was 45 minutes). This is in contrast with programming practice. For instance Beck suggests the following: *'if several tasks each take an hour, combine them to form a larger task'* [1, page 92]. So, tasks taking a few hours each would be much more representative.

The second experiment we know of has been conducted at the University of Utah [6]. Forty-two senior software engineering students have been split into 14 pairs and 14 individuals. They were asked to write 4 programming assignments in 6 weeks. The results of that experiment are even more optimistic than those received by Nosek. They report that *'the pairs completed their assignments 40% to 50% faster'* and total programmer hours to complete an assignment *'after the adjustment period (..) decreased dramatically to a*

¹ This work has been financially supported by KBN grant 8T11A01618.

minimum of 15% [6]. Unfortunately, they do not say what were the assignments about, what was the process used by the students, what were the sizes of solutions, how many hours the students spent on writing the programs, and last but not least, what was deviation in time, test cases passed, and size. Moreover, for one of the programs '*data entry problems prevented accurate recording of the completion times*'. This shows that further experiments are needed to investigate efficiency of pair programming.

In the paper four experiments are described that have been conducted in winter semester 1999/2000 at the Poznan University of Technology. The aim was to evaluate efficiency of pair programming. We decided to compare pair programming, which is a practice of XP [1], with individual programming in its rigorous version proposed by Humphrey and known as Personal Software Process (PSP) [3].

In the following sections we shortly describe PSP with focus on elements used in our experiments, and then we proceed to eXtreme Programming practices and pair programming. Next we present organization of the experiment and its results. The paper ends with conclusions and remarks concerning future research in the area.

2. Personal Software Process

Personal Software Process (PSP) and Capability Maturity Model (CMM) have been developed at the Software Engineering Institute, Carnegie-Mellon University. The aim of CMM and PSP is to improve software processes. While CMM aims at software organizations, PSP tries to help individual programmers. Both CMM and PSP define a few levels of maturity. For PSP there are the following levels of maturity (from the least to the most advanced):

PSP0: Development time and defects are recorded. Defects are categorized according to a defect type standard.

PSP0.1: Coding standard and size measurement are in use. Observed opportunities for improvement are put in a form of process improvement proposal.

PSP1: Software size is estimated and test reports are written.

PSP1.1: Task and schedule planning are in place.

PSP2: Code and design reviews are performed.

PSP2.1: Design templates are in use.

PSP3: Incremental approach is applied to software development.

Levels PSP0 and PSP0.1 are called Baseline Personal Process, BPP. In our experiments the programmers worked at the level of BPP.

In PSP an important role is assigned to strict software process definition which is a description of "*sequence of steps required to develop or maintain software*" [3]. A process can be split into a number of smaller processes called phases. Each process/phase is described by a script, which specifies not only the steps but also the entry criteria and exit ones. In his book Humphrey proposed to focus on three phases: design, code and testing. The reason for that choice is that according to Boehm those phases consume from 59 to 68 percent of development cost in most software organizations [2] as well as the fact that these are the parts of the process where a running product is created. Moreover, he introduced to the process so called 'postmortem' phase during which the programmer has to complete process logs and summary report. A typical process suggested by Humphrey for the BPP level is depicted in Figure 1 (that process has been used in our experiments).

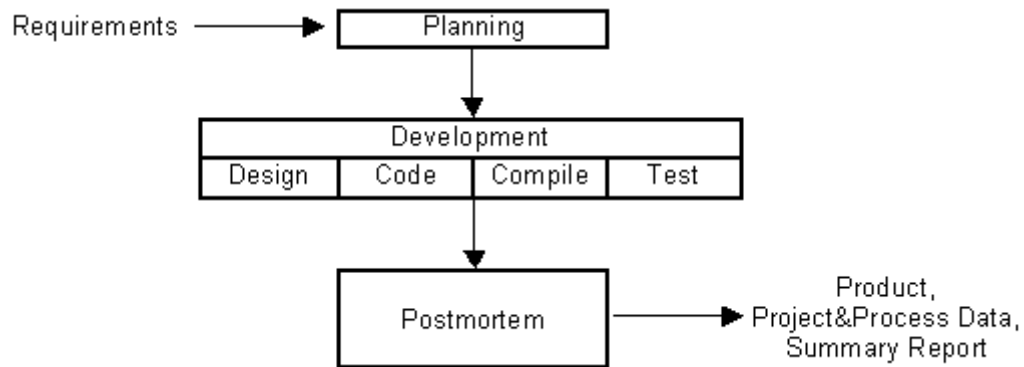


Figure 1. The PSP0 process

Specific to this approach is its reflective character, since the programmer does not use a compiler until the whole program is coded. Compilation starts from the moment when a compiler is used for the first time and it lasts till the first successful compilation. The testing phase comprises all the activities from the first successful compilation till the moment when acceptance criteria are met (in practice the program should pass all the tests – unfortunately, the scripts for BPP do not specify when the test cases should be created).

3. eXtreme Programming

eXtreme Programming [1, 4], XP for short, is a relatively new methodology of software development. It was proposed in response to problems concerning changing (dynamic) requirements and new technologies. Here are the practices of XP that have been used during our experiments:

- *Pair programming.* Pairs of programmers instead of individuals do coding. Each pair member is expected to feel an author and to fully understand the piece of code that is developed.
- *Experimentation and test-centred quality assurance.* Writing a piece of code should be preceded by preparing a set of test cases. If a defect is detected the programmer should write test cases that are able to detect the defect whenever it reappears. Each code must pass all the unit tests.
- *Simple solution.* Design should be as simple as possible and no functionality should be added early. Any optimisation that is not put in the requirements should be avoided.
- *Risk minimisation.* Spike solutions are recommended to reduce risk.
- *Keep moving.* Members of the programming teams should be moved around (we moved them before starting work on a new assignment).

Since the chosen programming assignments were small (150-400 LOC) and the requirements were stable

- we have not used user stories,
- there was no on-site customer and planning game,
- integration was very simplified,
- there was only one very short iteration lasting not longer than one day, and
- the students participating in the experiment did not use CRC cards.

4. Description of the experiment

The aim of the experiment was to compare pair programming along with other practices of the XP (see Sec. 3) with the PSP approach described by Watts Humphrey (see Sec. 2). We have decided to use a set of programming assignments proposed by Humphrey [3] and measure both the development time and the number of defects. Experiment took place at the Poznan University of Technology, during laboratory classes at 4th year of studies (computer science major). The students have had similar experience in programming. They knew that the aim of the experiment was to evaluate the methodologies, not their skills.

XP-like pair programming and PSP are very different. They differ with regard to number of programmers and type of processes. Because of this we have decided to introduce XP-like software process that was targeted at individual programmers. All in all we had three software processes:

- Baseline Personal Process as introduced by Humphrey - referred to as **PSP**;
- XP-like pair programming – referred to as **XP2**;
- A version of XP for single programmers (the practices listed in Sec. 3 have been reduced to *Experimentation and test-centred quality assurance*, *Simple solution*, and *Risk minimisation*) – it will be named **XP1**.

A group of 21 students was divided into three sub-groups:

- PSP: 6 programmers,
- XP1: 5 programmers,
- XP2: 5 pairs of programmers (10 students; each pair was equipped this one computer).

We have assigned the students to the groups (PSP, XP1, and XP2) in such a way that for each group the average value of GPA (Grade Point Average) was almost the same. The students did all the work at the university when one of the authors was present, so we can trust their time records.

The students might choose their programming language between Pascal, C or C++. All of them used C or C++. Programs for the experiment were taken from [3] to make it easier to compare collected results to records presented in Humphrey's book, and to be sure that those programs suite well the methodology proposed in the PSP approach. The assignments were the following (this is a rough description – for details see [3]):

- **Program 1.** *Write a program to estimate the mean and standard deviation of a sample of n real numbers [3, Program 1A].*
- **Program 2.** *Write a program to calculate the linear regression parameters [3, Program 4A].*
- **Program 3.** *Write a program to count the logical lines in a program, omitting comments and blank lines [3, Program 2A].*
- **Program 4.** *Write a program to count the total program LOC, the total LOC in each object the program contains, and the number of methods in each object [3, Program 3A].*

Students using PSP approach were asked first to write their program on a sheet of paper, then to use a computer. Because they complained on loosing time while rewriting code, we allowed them, from the second program on, to use text editors, but we asked them to run a compiler not before the code is complete (such a strict approach is suggested by Humphrey).

The XP paradigm assumes that each of the two programmers working on a program is equally involved in the program creation. Thus, we had to prevent a danger that only one student would develop entire code while his partner might remain passive. To make sure that this will not happen we decided that once a program is finished and accepted we will inject three errors by performing the following modifications:

- Adding an assignment instruction.

- Removing an assignment instruction.
- Changing an instruction (not more than two lexical units could be changed).

After acceptance of a program the students were given a modified program and rough information about location of each injected error (an area of 20 lines of code was indicated). By comparing the fix time data for XP2 with that for PSP and XP1 we were able to check if the XP2 students have had the same knowledge about their programs as the PSP and XP1 students had.

During entire process of code development students had to track time and defects using special logs (time recording log, and defect recording log) [7], similar to those proposed by Humphrey. Information collected there includes time of starting and finishing of each phase as well as notes on all errors encountered in code, tests etc. We also wanted students to keep track how long they spent on uncovering and correcting their own errors as well as those injected by us after the program was accepted. Duration of each break in program development was subtracted from the time spend in particular phase of software process.

5. Results

Each of the experiment sessions gave us 16 logs concerning development time and 16 logs containing data about programming defects. Those records have been taken together for statistical analysis. The results are presented below.

First let us compare average value of total development time gained in XP1, XP2 and PSP groups (see Figure 2). From Figure 2 it follows that for the first three programs there is **almost no difference in development time between XP2 and XP1**. This suggests that **pair programming is rather expensive technology**. It is surprising, especially when you take into account results obtained by J.T. Nosek [5] and L. Williams et al. 7[6].

For Program 4 there is a significant difference between XP1 and XP2. This can be attributed to assignment interpretation problems one of the XP1 programmers have had (if we removed his data from the set, the average development time for XP1 would amount to 3 hours 40 minutes and this is just slightly above the average time for XP2).

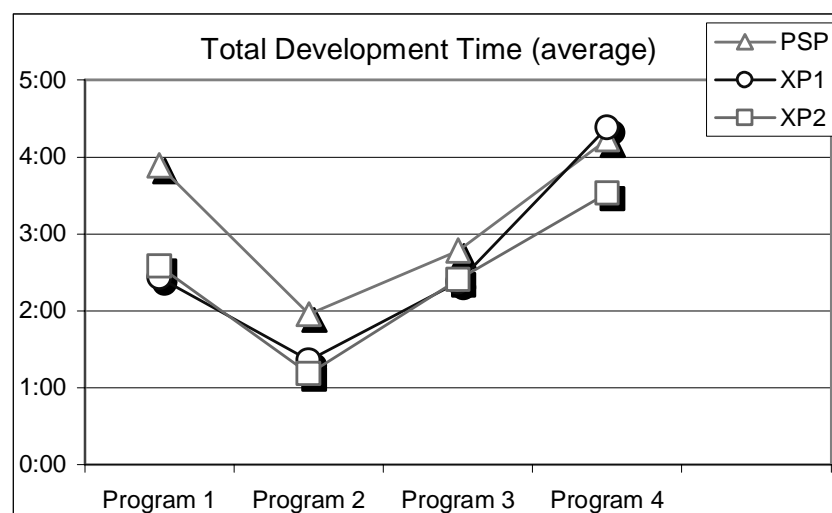


Figure 2. Comparison of average values of total development time.

Notice that the distance between average development time for PSP and XP2 is maximal for Program 1. It can be attributed to our early requirement that a program should be first written on a sheet of paper, then entered to the computer – that requirement was not in force

for the remaining programs. Since average program size was 150 LOC for Program 1, one should subtract about 30 minutes from the average time for Program 1 and PSP. Nevertheless PSP takes longer than XP2 and XP1. Especially important is the difference between PSP and XP1. It suggests that **experimentation and test-centered thinking reduces development time**, at least in case of small programs. In this context the strict waterfall approach presented in [3] (design – coding – testing) seems less efficient.

We have also analysed standard deviation of development times and program sizes in each of the groups (see Figure 3). From this analysis it follows that deviation both in development time and in program size for XP2 is smaller than for XP1 and PSP (the only exception is the development time for the first program). It indicates that **pair programming is more predictable than individual one**.

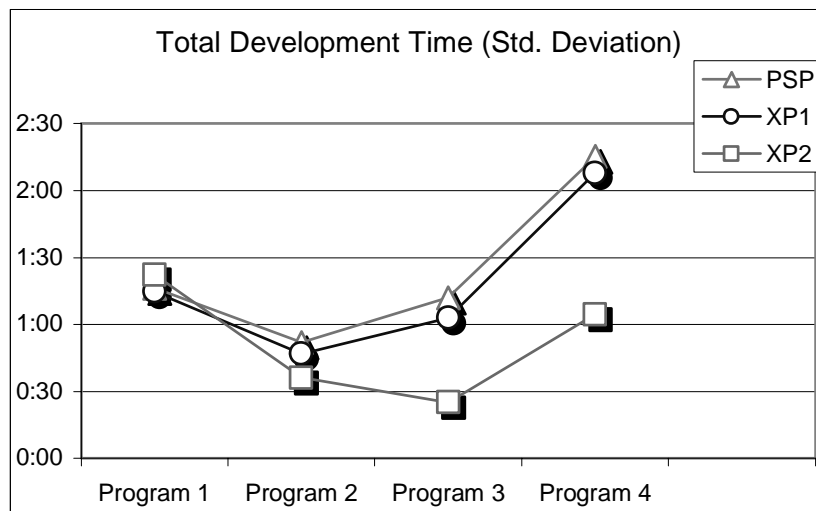


Figure 3. Standard deviation in development time.

Another interesting attribute of a software process is programming efficiency measured in lines of code (LOC) per hour per person. From Figure 4 it follows that **XP1 is the most efficient programming technology, while PSP and XP2 are more or less the same**. This can be attributed rather flexible process used by XP1 programmers and the requirement to write test cases before coding.

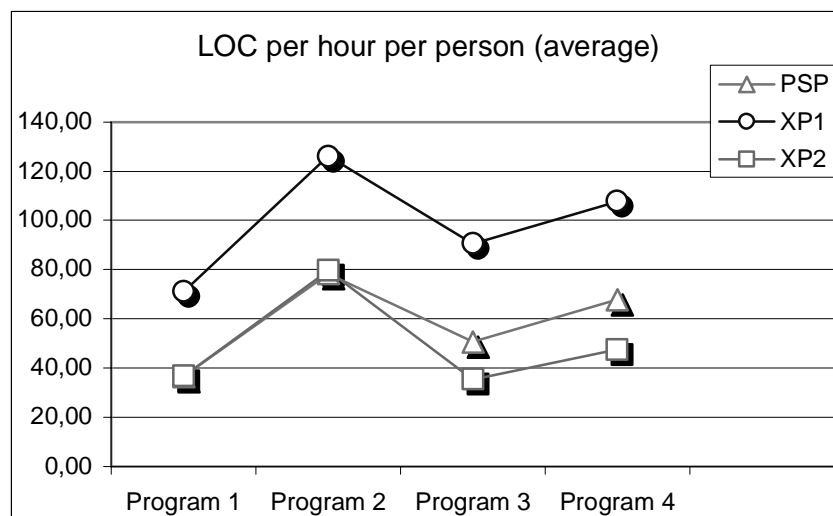


Figure 4. Programming efficiency (LOC per hour per person).

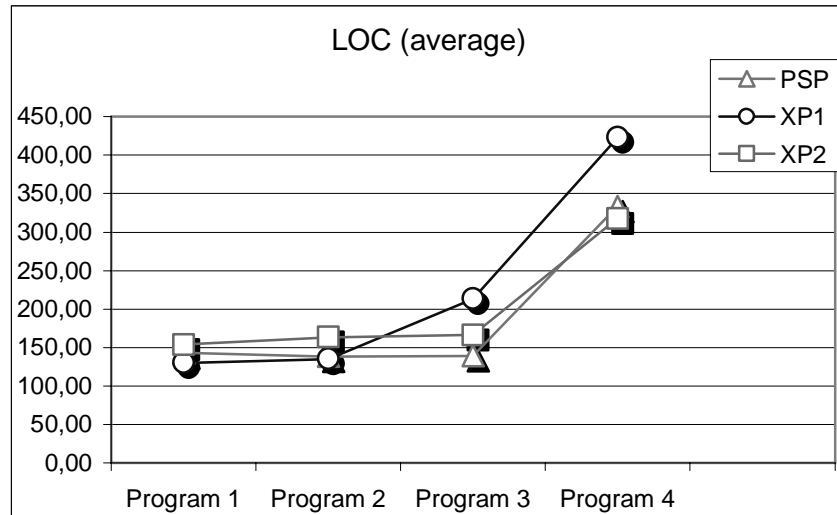


Figure 5. Average program size (LOC).

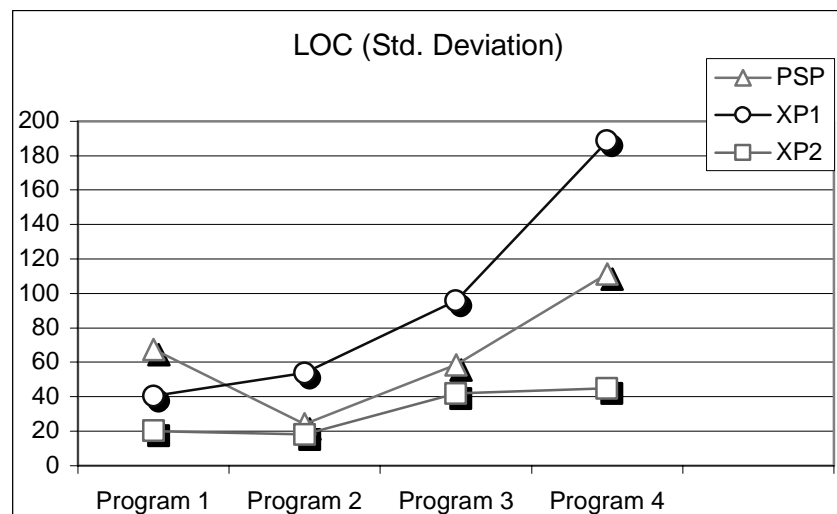


Figure 6. Standard deviation in code size.

Data concerning program size for each program are presented in Figure 5. For the first two programs the average program size was almost the same for each of the approaches, but for Programs 3 and 4 XP1 provided longer programs (on average) than PSP and XP2. From Figure 6 it follows that standard deviation of program size for XP2 was the smallest one. It means that pair programming leads to more stable solutions. Notice that XP1 is (except Program 1) least predictable in sense of program size.

Another interesting indicator of software process efficiency is the number of errors uncovered during acceptance testing. In Figure 7 the number of re-submissions of corrected programs is depicted. In ideal case that number should be equal to zero. From the figure it follows that all the groups have had similar level of acceptance errors and XP2 is slightly better.

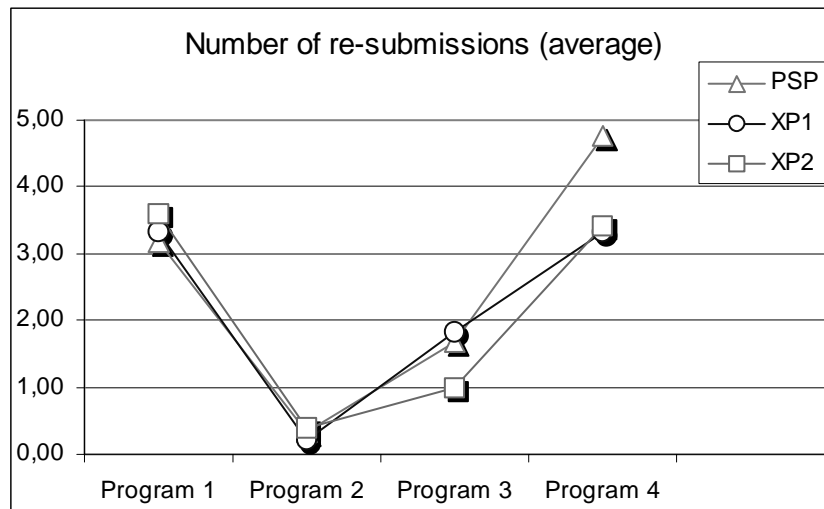


Figure 7. The number of re-submissions of corrected programs.

6. Conclusions

In the paper we have compared XP-like pair programming with individual programming based on PSP and XP practices. From our research it follows that:

- XP-like pair programming appears less efficient than it is reported by J.T. Nosek [5] and L. Williams et al. [6]. Only in one case (Program 4) reduction of average development time for XP2 (to 77% of the time needed by XP1) was similar to that reported by J. T. Nosek (reduction to 71%) but still it is far from the level of 50% mentioned by L. Williams et al. Moreover, in case of Program 4 one of the students misunderstood the assignment and this distorted the results.
- PSP seems less efficient than XP1 (see Fig. 2). Maybe it is too restrictive? However, its average code size for programs 3 and 4 was considerably smaller than in case of XP1 (see Fig. 5).
- Pair programming is more predictable than individual one with regard to development time and program size (i.e. standard deviation is the lowest one – see Fig. 3 and Fig. 6).
- Experimentation and test-oriented thinking reduces development time.
- The number of re-submissions indicates that the amount of rework for XP2 is slightly smaller than for XP1 or PSP.

However, one should take into account that the research was restricted to relatively small programs (150 – 400 LOC). Although our assignments were a few times longer than one used by J.T. Nosek, we think that further experiments are needed oriented towards larger programs.

7. References

- [1] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Boston, 2000.
- [2] B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, 1981.
- [3] W.S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, Reading, 1995.
- [4] <http://www.extremeprogramming.org>, January 2000.
- [5] J.T. Nosek, The cast for collaborative programming, *Communications of the ACM*, vol. 41 (1998), No. 3, 105-108.
- [6] L. Williams et al., Strengthening the case for pair programming, *IEEE Software*, vol. 17 (2000), No. 4, 19-25.
- [7] http://www.cs.put.poznan.pl/awojciechowski/research/pair_programming/, February 2001.