Publication IV

Jari Vanhanen and Harri Korpi. 2007. Experiences of using pair programming in an agile project. In: Ralph H. Sprague, Jr. (editor). Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS 2007). Waikoloa, Hawaii, USA. 3-6 January 2007. Los Alamitos, California, USA. IEEE Computer Society. 274b, 10 pages. ISBN 978-0-7695-2755-0.

Experiences of Using Pair Programming in an Agile Project

Jari Vanhanen
Helsinki University of Technology, SoberIT
jari.vanhanen@tkk.fi

Harri Korpi
Helsinki University of Technology, SoberIT
hhkorpi@cc.hut.fi

Abstract

The interest in pair programming (PP) has increased recently, e.g. by the popularization of agile software development. However, many practicalities of PP are poorly understood. We present experiences of using PP extensively in an industrial project. The fact that the team had a limited number of high-end workstations forced it in a positive way to quick deployment and rigorous use of PP. The developers liked PP and learned it easily. Initially, the pairs were not rotated frequently but adopting daily, random rotation improved the situation. Frequent rotation seemed to improve knowledge transfer. The driver/navigator roles were switched seldom, but still the partners communicated actively. The navigator rarely spotted defects during coding, but the released code contained almost no defects. Test-driven development and design in pairs possibly decreased defects. The developers considered that PP improved quality and knowledge transfer, and was better suited for complex tasks than for easy tasks.

1. Introduction

In pair programming (PP) two persons design, code and test software together at one computer. The driver controls the keyboard and the navigator observes the driver's work and thinks at a more strategic level. The persons should communicate actively and switch the roles periodically. [1]

PP seems to produce better designs with fewer defects in the code, in shorter elapsed time and more enjoyably than solo programming, and it also seems to benefit teamwork, knowledge transfer and learning [2, 3, 4, 5]. It seems that PP requires somewhat more development effort [2, 3, 4, 6, 7]. PP can be very intense and mentally exhaustive [1].

Anecdotes of developing software together dating back to the 1950s are reported in [1]. The first two experiments studying a similar practice calling it mere collaboration [8] or collaborative programming [3] were reported in the 1990s. Being one of the mandatory practices in the popular Extreme Programming (XP) software development approach [9] has made PP better known lately. XP's characterization of PP is similar to [1], but XP requires using it for all production code. However, PP can be used in a less disciplined way and in any development approach. According to a global survey PP was used in 35% of development projects [10] indicating a high interest in PP even though the research on PP is still inconclusive [11].

A PP research framework [11] proposes several context variables affecting the outcome of PP: education, experience and personality of developers, roles, communication, switching partners, type of development activity and task, development process and tools, and workspace facilities. Organizations adopting PP often have the possibility to control many of these variables, but they seem to understand poorly what would be a good context for PP. The difficulty can be seen in the practical questions we have faced when observing the adoption of PP in non-XP, industrial contexts:

- Which tasks are performed using PP?
  - … and which activities of the tasks (analysis, design, code, test)?
- Who proposes the use of PP for a task and when?
  - managers, developers or both?
- Who pairs with whom considering e.g. competencies, experience and personalities?
- How long does the same pair work together?
- If a pair does not do a whole task together:
  - How much do they work together?
  - Do they work separately with the same task?
  - How do they synchronize after separation?
  - How do they communicate during separation?
- What kind of infrastructure is good for PP?
- How does one behave during a PP session?
- How often are the roles switched?

XP gives some extreme answers, e.g., everyone should use PP for all development tasks from the start

Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS'07)
0-7695-2755-8/07 $20.00 © 2007 IEEE
1530-1605/07 $20.00 © 2007 IEEE

to the end. The guidelines and examples in [1] also mostly assume using PP in the XP way. However, as discussed in Section 2.1, it seems than even XP projects seldom apply such an extreme approach and therefore the answers from XP are not enough.

Generally, the answers are likely to depend on the goals set for the use of PP, e.g. ensuring high quality vs. mentoring a novice, and on the fixed context variables, e.g. the characteristics of the developers. Different answers lead to different flavors of PP.

Having better guidelines for answering this kind of questions especially in non-XP projects would be valuable. Detailed case studies about using different flavors of PP in different contexts would allow researchers and practitioners to gradually increase their understanding and create better guidelines. This paper attempts to provide detailed insights into applying PP in a small, agile team. We also report some data that has not usually been reported, such as the amount of knowledge transfer within the team and the amounts of time spent together by the different pairs.

The paper is structured as follows. Chapter 2 presents experiences of the practicalities of PP from literature. Chapter 3 describes the research methodology of our case study. Chapter 4 introduces the context of the case study. Chapters 5 and 6 present and discuss the results from the case study related to the practicalities and effects of PP. Chapter 7 concludes the paper.

2. Related work

In this chapter we present and discuss experiences from literature related to the areas we discuss in our case study, i.e. adopting PP, pair formation and PP sessions. We searched carefully for scientific papers from IEEE Xplore, ACM Digital library and the INSPEC database using keywords “pair programming” or “collaborative programming” also going through the reference lists of the found papers. We excluded papers discussing PP used by students, because most of them discuss novices performing small development tasks, i.e. the context for PP is different from that of PP in industry. Many papers were about XP projects and in some of them PP was discussed only shortly among other practices. Probably due to the concise form of reporting, most papers discuss only some practicalities of PP. The published reports may be biased towards more positive experiences, because less successful adoptions may be less likely published.

2.1. Adopting pair programming

At wotif.com, a three person team first used all XP practices except PP. PP was not used because they expected it to require additional effort which was not included in the effort estimates for the project. However, the team started to think about adopting PP after they faced a critical problem, which might have been avoided with PP. Full PP use was still out of question due to the expected increase in effort. Even after this the developers did not really adopt PP. Only after the team coach persuaded the developers into interacting and the team lead started holding both partners responsible for code quality did the use of PP improve. The team used PP for all design work, but less for programming work. A pair met several times a day and the author updated the partner and they discussed problems. It is not explicitly mentioned in the paper, but it seems that only one person worked with the task when the pair was detached. They programmed together a few times per week the reason being that the partner had much knowledge about the task or the code was very subtle, complex or involved high risk. When the author finished a task the partner reviewed the work with the author. Code reviews worked well because the partner was familiar with the design and code. Developers enjoyed the flexibility of pairing. [12]

For a team developing firmware for Intel processors, increasing knowledge transfer between too specialized developers gave a reason for adopting PP. The team ended up using PP for detailed design and initial coding, but splitting when coding got tedious. They had problems in getting PP in frequent use due to the pressure from stringent deadlines. The developers felt it would be quicker to work alone in the area of their own expertise, i.e. PP could increase effort and lead to missing a deadline. In order to ensure some use of PP they started to require everyone to work one day per week with something else than their core expertise. [13]

In Guidant Corporation PP was added to a quite traditional development process. Rules for the practicalities of PP were established but the developers were allowed to choose between PP and their old way. However, as a reward of using PP, the code from a pair did not require a formal peer review. Five out of nine developers started using PP immediately and in a couple of months the rest of the team agreed to adopt PP after seeing the positive results. All tasks were jointly performed by a pair. However, the pair was free to choose its own style of working. Some worked very closely together, whereas others split up the work , did it separately, and then came together to share and review. [14]

At IBM in a team moving from a waterfall process to XP the degree of PP increased from roughly 11% to about 50% when PP was given as an alternative to formal inspections. Cultural resistance was mentioned as a reason preventing further increase of PP. [15]

In a ten-person XP team at Sabre the developers ended up using PP for about 50% of their time. Most developers saw no value in using PP for trivial tasks, but all considered it valuable for solving problems and overcoming technical difficulties. [15]

Secure Trading had a nine-person XP team. The developers spent about 30% of their time with a pair. After initial experiences 28% of the developers reported they prefer to work alone. About half of the developers thought that they could not work with everyone. [16]

In an organization the staff agreed with the given motivation for using PP but in practice they continued coding alone and then reviewed the changes with the partner before code check-in. After four months, things started to move towards real PP. The developers realized that it was more efficient to sit together all the time than to update the partner and review changes. The developers' personal experiences were the key to the move to systematic use of PP. [5]

At FJA Odateam an XP team used PP for difficult tasks and for teaching new people, but otherwise they tended to work alone. After the workspace was changed to an open office and ownership of workstations was dropped they started to use PP for all tasks. [7]

Aiken [17] presents experiences from three people who have used some PP or followed its use. According to them the organization's culture may have a huge impact on the success of PP. Reducing any potential fears of judgment is important, and the developers' learning styles and personal preferences should be respected. PP creates issues initially and there are decreases in productivity when people are adjusting to it.

The main findings are summarized in Table 1. The slow start for adopting PP was mentioned often. One reason was the expected increase in effort. Several special measures were used for increasing the use of PP. Reserving one day per week for PP ensures a certain amount of PP. The managers could persuade to its use or emphasize that both partners are responsible for the quality of a task. Others provided rewards for the use of PP in the form of avoiding formal reviews or inspections. Moving to an open office and dropping the ownership of work stations also improved the use of PP. Seeing the good results of PP was also a good motivator. Williams and Kessler [1] also report that most people resist transitioning to PP, but almost all who try it consider it better than working alone.

The amount of PP varied a lot between the cases. In two papers on XP context, figures of 30% and 50% for time spent with a pair were reported. This is much less that proposed by XP. In one XP team PP was reported to be used for all tasks.

Table 1. Experiences of adoption

Topic	Experiences
amount	• 50% of time (two different cases) [15] • 30% of time [16]
application	• all design and some programming [12] • detailed design and initial coding [13] • each task was assigned to a pair, but the degree of working together varied [14] • solving problems and technical difficulties, not for trivial tasks [15] • difficult tasks and teaching, later all tasks [7]
limiting factors	• expected effort increase [12] • no time for PP due to deadline pressure [13] • only peer reviews before check-in [5] • developers cannot work with everyone [16] • organization's culture [15, 17]

2.2. Pair formation

Two ways for pair formation are described in [1]. First, a short, daily meeting where the developers discuss their plans for the day and the possible problems they are having. During the meeting people can volunteer as partners to people who they think they can best help. If some developers are still without a pair after the meeting, a manager can form the most appropriate pairs out of them. Alternatively, a task owner can ask help from anyone and nobody can say no. The assumption seems to be that everyone works with a partner.

Pair rotation should be periodic [1]. It often occurs very casually the developers themselves knowing the optimal partners [1]. It works only if the tasks are broken into small, half day to one week chunks which are assigned to an owner, who can then recruit a partner for the task [1]. XP proposes rotation even every couple of hours switching at natural breaks in the work [9].

Forming pairs in a daily meeting is common [18, 19, 20]. The daily rotation in a ten-person team made everyone pair with everyone at least once during a three-week iteration [20]. In [14] the pairs were formed based on the skill-set required and rotated when new tasks arrived.

At Silver Platter Software a six-person XP team experimented with different attributes of PP. They found that rotating pairs very often, e.g. every 90 minutes, was most productive. Removing the person who had been working longer with the task was most productive. Initially most developers felt that switching was too frequent but after a few weeks they realized the effects on learning and got exited. The experimenters assume that the reasons for the surprising results were that 1) the developers worked with a "beginner's mind" all the time and 2) the most important information is usually passed during the first hour of pairing. [21]

Bryant [22] observed 14 one-hour PP sessions. She was surprised about the fluidity of pair rotation. Often a pair did not finish their task, but re-grouped easily when another pairing was more appropriate. Overhearing what other pairs were doing helped to realize when more appropriate pairs could be formed.

Chong [18] also found that the dialogue produced in the pairs made other developers aware of what the pairs were working with. This allowed a developer to join the pair when the pair was caught with a problem he could solve easily. Probably as a consequence of this the PP sessions were often interrupted when one or both persons turned their attention to help others.

Experiences of pair formation are summarized in Table 2. It seems that usually the pairs are formed casually, and in some cases pairs are rotated very frequently. There seemed to be no problems in forming and rotating pairs.

Table 2. Experiences of pair formation

Topic	Experiences
means and frequency	• in daily meetings [1, 18, 19, 20] • change the person who joined the task earlier every 1.5 hours [21] • when new tasks arrive [14] • overhearing allowed discovering situations when another pairing was more appropriate [18, 22]

There are context-specific benefits and drawbacks for different pairings with regard to e.g. skill levels and personality. Some pairings do not work even though PP works with most partners. Problems may occur, e.g. with a person who has excess ego or when pairing a novice with an expert having no mentoring attitude. [1]

PP experts warn about many possibly problematic pairings. A novice can slow down and frustrate an expert who may lower the self-esteem of the former; two experts can be inefficient, e.g. when the lack of involvement frustrates the navigator, or when there is constant 'clashing of the minds'; two novices have the risk of 'the blind leading the blind'. [23]

Jensen [24] reports than in his experiment the most troublesome pairings were those in which the partners had about the same capability level. On the other hand at Sabre large differences in expertise and age caused resistance for using PP [15].

Table 3. Experiences of pairing

Topic	Experiences
possibly challenging pairings	• similar expertise [23, 24] • large differences in expertise [15, 23], when the expert has no mentoring attitude [1] • large differences in age [15] • one of the persons having excess ego [1]

Experiences of pairings are summarized in Table 3. It seems that there may be some issues with several pairings, but the experiences are quite limited and to some degree contradictory.

2.3. Pair programming sessions

According to Williams and Kessler [1] switching roles periodically between the driver and navigator is very important. It activates a possibly passive navigator by letting him write.

When observing PP in four companies the researchers found that the roles were switched mostly when the driver slid the keyboard over to the navigator. The navigator seldom initiated control of the keyboard. [25]

In one XP team, switching did not happen. Even the frequent intervention by the team coach did not help. As a result, the driver's attention would drift away and also the knowledge transfer suffered. [19]

Bryant [22] observed the degree and type of interactions within different pairs. Pairs formed of more expert pair programmers (PPers) had 27% fewer interactions than pairs formed from novice PPers. She suggests that expert PPers might be more selective about their interactions and may have a better understanding of the role and knowledge of themselves and their partner. Bryant noticed that expert PPers spent time resolving differences in opinion, whereas novice PPers often trashed between different strategies, depending on who was driving. It seemed that all expert PPers behaved in a certain role (driver or navigator) in a similar way to each other. Novice PPers also changed their behavior when changing the role, but each novice PPer had his/her own style of behavior in a role. Therefore, Bryant proposed that novice PPers might learn PP from observing how expert PPers work in a PP session.

Cockburn and Williams report that encouraging the developers to think aloud improved the interaction between the partners [5]. In another case the PP practice matured after introducing a team coach, whose task according to XP is to take care that the XP team uses certain practices [16].

In an XP team several developers mentioned that the only way to solve communication problems with the partner is to show more courage in criticizing the partner's work and more acceptance of the criticism. Frequent pair rotation is needed for increasing learning and the feeling of collective code ownership. [26]

At FJA Odateam in a PP experiment of about six hours, the developers were found to switch roles very often, from 6 to 42 times. The pairs formed of more experienced developers switched roles more often. [7]

Experiences of PP sessions are summarized in Table 4. There seemed to be problems with switching the

roles during a PP session in many cases. Both this and the lack of thinking aloud by the driver could lead to insufficient interaction between the partners.

Table 4. Experiences of PP sessions

Topic	Experiences
role switching	• periodic switching is very important [1] • mostly initiated by the driver [25] • did not happen and the navigator's attention would drift away [19] • roles were switched 6-42 times in different pairs during a six-hour PP session [7] • more experienced developers switched roles more often [7]
interaction	• pairs of more expert PPers had 27% fewer interactions than pairs of novice PPers [22] • overseeing needed for making PP work [5, 16] • solving communication problems by showing more courage in criticizing the partner's work and more acceptance of the criticism [26]

3. Research methodology

This study was a single case study [27]. The case was chosen based on its availability, i.e. a quite rare opportunity emerged to closely observe an industrial project using pair programming.

The research questions for the case study were:

1. How does the team apply PP during the project?
2. What are the effects of using PP?

Question 1 covers all the practicalities described in Chapter 1 and Question 2 evaluates the effects on quality, effort, knowledge transfer and work enjoyment.

We used several data collection methods. These included interviewing, inquiries, reporting requirements and observation, as shown in Table 5.

The first author observed the project from the outside, and the second author participated in it as a developer, who also took responsibility for disciplined collection of data from the team.

The developers tracked effort per task on a paper and the second author collected the data at least weekly. The customer reported to the team all defects he or the end-users found and the second author counted them.

After iterations I1, I2 and I3, each developer performed an evaluation for each module in the system with the question "How good is your knowledge about module X?" (Questionnaires KQ1-KQ3). This data allows evaluating knowledge transfer within the team.

After I4, the first author conducted a semi-structured team interview (TI4). It gave insights into the practicalities of PP and its effects. After the interview, each developer filled out a short questionnaire (IQ4) about the perceived effects of PP and how well they liked it.

After I4, the developers evaluated the difficulty of using certain practices on scale of "1=easy"–"5=difficult". They also ranked the practices based on their effect on quality, knowledge transfer, productivity etc. (questionnaire RQ4).

Table 5. Data collection instruments

What	When	How
effort	daily	individual time reporting, collected weekly
defects	daily	second author kept count
knowledge transfer	after I1-I3	questionnaire (KQ1-KQ3)
practicalities of PP	after I4	team interview (TI4)
effects and feelings of PP	after I4	questionnaire (IQ4)
importance and difficulty of practices	after I4	questionnaire (RQ4)

The first author analyzed all the data. The qualitative data from Interview TI4 and Questionnaire IQ4 was grouped and synthesized thematically, and finally the correctness of the interpretation of the data and missing details were discussed with a developer, who was not the second author.

4. Case study description

The observed project was carried out in a large telecommunications company in Finland. The goal of the project was to develop an internal reporting system for the company using Java technologies. Another goal was to pilot agile practices.

All four developers of the project team were males. They had not worked before with each other or in the case organization. Their willingness to use agile practices was ensured when recruiting the developers. The developers had 4-10 years of programming experience of which 1.5-4.0 years with the technologies used in the project. Three developers had not used PP before the project and one had used it for about a month. Before the project the attitudes of the developers towards PP varied from slightly negative to quite positive.

In practice all the developers were equal, even though one of them acted as a team leader who took care of, e.g. arranging the daily meetings. The second author was one of the developers and he observed the use of agile practices for his master's thesis.

The development was done in six consecutive iterations, usually lasting two weeks. They were preceded by a two-week pre-project iteration, where the team was given lectures about agile development covering also PP. In the pre-project iteration, the team decided about the process, selected the technologies, and experimented with both. The process was a collection of practices from several agile methodologies.

The team had a coach who helped in issues related to the work practices. In the beginning of the project the coach visited the team daily, e.g. by participating in the daily meeting. However, the developers themselves had the power to decide on the used practices. They actually had a reflection meeting after each iteration and thus continuously improved the used set of practices to better meet their needs.

The developers had an open workspace i.e. they shared the same room and had visual contact with each other. There were no private workstations available. The coach and a person acting in the customer role had their own rooms close to the team.

Each task was usually self-assigned to a pair who worked together with it. If a pair separated the partners either continued with the task separately or sometimes one of them could start another task.

5. Experiences of pair programming

5.1. Adopting pair programming

The pre-project iteration consisted of eight hours of PP by each developer during four days. The developers considered this was sufficient to start doing PP efficiently. This is quite a short time considering that the developers had to learn pair programming and to get familiar with each other. Williams and Kessler [1] propose that it may take even some weeks before a pair gets into the flow of PP.

The developers adopted PP thoroughly from the start of the project, which contrasts heavily with the adoption problems described in Section 2.1. The team had only two high-end workstations and one (later two) low-end workstations. This practically forced them to use a lot of PP. The developers did not criticize the setting, probably because of their approval of experimentation with agile practices, but also because PP was soon accepted as a good practice.

The developers' opinions on the difficulty of using certain practices (Questionnaire RQ4) are shown in Table 6. Everyone considered using PP easy both absolutely and compared to the other evaluated practices.

Table 6. Difficulty of using certain practices

Practice	Answers
Pair programming	1, 1, 1, 2
Test-driven development	2, 2, 2, 3
Writing unit tests	2, 2, 3, 3
Working without real requirements	2, 2, 3, 3
Planning game	3, 3, 4, 5

Scale: 1=easy – 5=difficult

The degree of pair work in each iteration is shown in Table 7. 72% of the programming effort in the project and 52% of all effort was done in pairs. The use of PP was slightly lower in iterations I2, I5 and I6 because refactoring and fine-tuning activities took time away from developing new features. These activities were considered easier and therefore PP was used less. The team size also decreased to only two developers after I4. Iterations I1-I4 reflect the normal state of the project better and for them the amounts of pair work were 77% of the programming effort and 66% of all effort.

The amount of pair work was quite high compared to the experiences discussed in Section 2.1. The limited number of high-end work stations was certainly an explaining factor and can be considered as a good tactic to make developers use PP actively.

Table 7. Proportion of PP

	Pre	I1	I2	I3	I4	I5	I6	Σ
Persons	4	4	4	4	4	2	2	
Days	4	9	10	10	11	10	17	71
All effort	129h	233h	235h	264h	274h	147h	222h	1505h
Pair work	N/A	135h	117h	159h	140h	42h	37h	628h
	N/A	82%	58%	73%	57%	28%	16%	52%
Programming	32h	100h	127h	138h	107h	18h	60h	582h
Pair work	32h	91h	79h	109h	85h	12h	11h	419h
	100%	91%	63%	79%	79%	67%	18%	72%

Developers considered PP especially useful for complex tasks. Similar findings were reported in Section 2.1. When doing simple copy and paste coding the navigator soon lost his interest in the work. Some straightforward tasks were done alone if they were easy to split into two parts where the other part could be performed at the low-end workstation by the partner.

5.2. Pair formation

Pairs were formed in the daily meetings. First the formation of pairs was affected by e.g. trying to avoid pairing the two most experienced developers, and maybe also by the frequent habit of smoking by two of the developers. In the first iterations the same two people continued to pair the next day if the same task continued. After I1 the team considered that such an infrequent rotation was not sufficient for good knowledge transfer. Frequent rotation was emphasized more in I2, but it still did not happen very often. Therefore, starting in I3 the team formed the pairs by casting a lot each morning so that the pairs always rotated compared to the previous day. If a pair did not finish their task by the end of the previous day, one of them continued it

with a new partner. Regular rotation worked well and simplified pair formation, because in a four-person team rotating pairs after a task ends requires waiting until the other pair also finishes their task.

Figure 1 shows how much the different pairs worked together. In I2 two pairs were not used. In I1 and I3 the pairing was more balanced. The distribution is similar in I1 and I3 even though in I1 the same pairs could continue several days together, whereas in I3 the pairs were often rotated on a daily basis. We have not seen industrial data from other researchers about this topic.

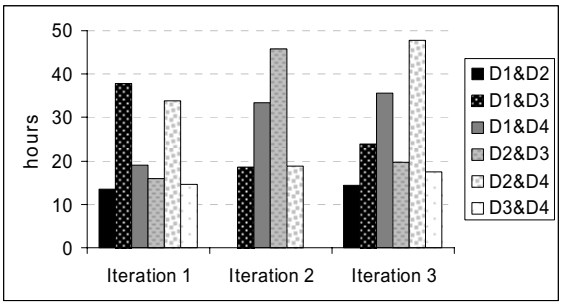

Figure 1. Working hours of the different pairs

5.3. Pair programming sessions

When starting a new task the pair made some specification work and consulted the customer for the details of the task. Then the pair did some design work and started programming. Sometimes the opinions on the amount of design required before coding differed and this could cause disagreement between the partners.

Communication was continuous with no silent moments. If a person was already familiar with the problem he acted as the driver and explained continuously what he did. The developers felt it impossible to act as the navigator without knowing what the code did.

PP could last the whole day, interrupted only by lunch or other breaks every now and then. The roles were switched 2-3 times a day, typically after the lunch break or when the driver took a personal break and temporarily left the workstation.

The use of PP got more loose later in the project. For example, the developers did not necessarily sit at the same workstation all the time when programming. Especially when very simple things were programmed the partner sometimes went to do other work or took a break. Choosing to work alone with simple code has been reported also by others [12, 13].

5.4. Dependencies with other practices

The developers considered that PP supported some practices and was supported by some other practices. The findings were similar to those proposed in XP.

The developers reported that as a pair they were more disciplined in using at least "test-driven development", "coding standard" and "integrate often" practices because the navigator noticed the deviations from their use. The navigator also helped decide when it was truly appropriate not to use a practice instead of the decision being made based on the laziness of the driver to follow a practice.

PP supported also collective code ownership because at least two persons participated in each task. Two persons were also better able to solve problems related to writing testable code and designing good unit tests. Test-driven development supported PP by increasing communication of ideas between the partners.

PP increased collective task ownership through increased collective code ownership. This together with an information radiator, i.e. post-it notes moved on an office wall according to their progress, created a shared feeling of achievement when a task was finished.

The importance of the daily meetings was quite low due to knowledge transfer through PP. However, the daily meetings were needed for forming the pairs.

6. Effects of pair programming

6.1. Quality

The number of defects found from each iteration release is shown in Table 8. The numbers include defects found by the customer in the acceptance testing after the iteration and by the end users in production use. The very low numbers seem to be reliable, because after 1.5 years in active production use only five new defects have been found. Over 200 000 items are daily updated in the database, a few users generate small reports daily, and an operator generates critical reports distributed to dozens of managers every other week.

Table 8. Defects found after development

Variable	I1	I2	I3	I4	I5	I6
Defects	1	4	3	4	3	2
Defects/KLOC	0.5	1.1	0.6	0.5	0.4	0.2

The developers considered that PP improved the understandability of design. When the navigator could no more understand the written code he forced the driver to stop writing. Thus, most code was written so that at least two persons were able to understand it. The developers also considered that they understood well even code that was not familiar to them if the code had been written by a pair.

In Questionnaire IQ4 all developers reported that PP lowers the number of defects and the measured defect counts also showed a very low value. However, in

Team Interview (TI4), the developers were somewhat uncertain about the role of PP in decreasing the number of defects because the navigator did not spot many defects during the programming. It may be that PP prevents the bugs before they are even written and therefore the navigator cannot point them out. This could happen because the pair brainstorms the design and writes unit tests together and thus thinks about the solution more thoroughly before writing the actual code.

The developers considered PP as the second most important practice after test-driven development for increasing the quality of the system and its design.

The improvements in quality are parallel with the results of the experiments made by others [2, 3, 4].

6.2. Effort

The developers considered the effect of PP on the development effort to be dependent on the type of task. For complex tasks, the use of PP might lower the total effort. The number of complex tasks was quite small, but they were big tasks requiring about 50% of the total project effort. The effort was considered higher for simple tasks than when using solo programming. Generally, PP was considered the most important practice affecting the project productivity positively. Others have reported that PP increases the development effort somewhat [2, 3, 4, 6, 7], but they have discussed this aspect in the context of individual tasks or small projects.

Productivity as LOC/hour for each iteration is shown in Table 9. Productivity was highest in I1 and I4. In I2, I5 and I6 the lower productivity is probably explained by the focus on refactoring and maintenance. A possible explanation for the lower productivity in I3 compared to I1 and I4 is discussed in the next section.

Table 9. Productivity

	Pre	I1	I2	I3	I4	I5	I6	Σ
LOC increase	517	2198	1411	1859	2290	248	700	8706
LOC/h	4.0	9.4	6.0	7.1	8.4	1.7	3.2	5.8
LOC/progr. h	16.2	21.5	11.2	13.4	21.3	13.8	11.6	14.9

6.3. Knowledge transfer

Each developer evaluated his knowledge of each module after iterations 1-3. The number of modules increased from 14 to 17 from I1 to I3.

The average of the evaluations of a module characterizes the team's knowledge of the module. The average of these characterizes the team's overall knowledge of the whole system (the first row in Table 10). The changes in the team's overall knowledge between the iterations were small. It indicates effective learning because the same knowledge was preserved even though the system grew and became more complex.

We assume that the knowledge transfer within the team is high if the differences (standard deviation) between the developers' knowledge of each module are small. The average of the differences from all the modules characterizes the overall differences (the last row in Table 10) and thus the degree of knowledge transfer. The overall differences decreased considerably in I3 indicating high knowledge transfer. The detailed data reveals that most of the decrease in the differences is explained by increases in the low values of knowledge and only little by decreases in the high values.

In I3 the higher frequency of rotating pairs made the developers work with more modules in I3 but also spend less time with each individual module. Therefore, the frequent rotation probably contributed both to the increases in the low values and decreases in the high values of knowledge. Table 9 showed much lower productivity in I3 than in I4. It may be that the frequent rotation first decreased the productivity as the developers worked more with modules unfamiliar to them and spent time learning new things. The benefits of learning were probably realized in I4.

Table 10. Team's knowledge of the system

Statistical value	I1	I2	I3
Average of averages of each module	3.82	3.74	3.91
Average of std.deviations of each module	1.04	0.96	0.69

Scale: 0=never heard – 5=like my own pockets

On a personal level there were large increases in knowledge for developers D1 and D2 in I3 (Table 11). D3 spent much less time for development than the others during I3, which explains his decreased knowledge.

Table 11. Personal knowledge of the system

Developer	I1	I2	I3
D1	3.71	3.53	4.12
D2	3.71	3.65	4.41
D3	4.00	3.88	3.24
D4	3.86	3.88	3.88

Scale: 0=never heard – 5=like my own pockets

All developers considered that PP increased their knowledge of the system more than solo programming. Two developers considered that PP helped them learn the development tools better, but the other two found no difference in this knowledge transfer aspect. The developers ranked PP as the most important practice for increasing team communication. The next most important practices were the open workspace and daily meetings.

6.4. Work enjoyment

The developers considered the team spirit very good. Some believed that co-location was a sufficient factor for this, but some considered PP to be more important. All agreed that PP promoted the formation of good team spirit in the beginning of the project. The information radiator and the open workspace were also mentioned as contributors to good team spirit. Two of the developers liked PP more than solo programming and two found no difference.

7. Discussion and conclusions

7.1. Lessons learned

Our findings are discussed below. They increase the body of knowledge of PP, help others in industry to apply PP, and help the research community to build improved guidelines for PP.

Learning PP took place quickly and developers considered its use very easy. Adoption seemed to be much easier than reported by other researchers. Having a smaller number of high-end workstations than there were developers certainly contributed to the quick adoption and rigorous use of PP. Surprisingly, the developers did not criticize the lack of own computers.

It seems that PP was better suited for difficult tasks. The developers avoided its use for trivial tasks if it was possible to split the task in two parts, i.e. one for each partner. The development effort was considered lower for PP than for solo programming with complex tasks but for easy tasks the situation was reverse. Similar findings have been reported by others [13, 15].

Initially pair rotation occurred only after a task was finished. In order to improve knowledge transfer the team started to actively rotate pairs each morning even if their tasks were not finished. The data about the developers' knowledge of the modules indicates that this change may really have increased the knowledge transfer within the team. However, there was a drop in productivity after the change, but the productivity rose again in the next iteration.

The driver/navigator roles were switched only 2-3 times a day. This may passivize the navigator [1, 19], but in our case study the partners maintained active communication. The use of PP became slightly more relaxed later in the project and the pairs could split up when the driver did some easy programming.

PP and especially the presence of the navigator increased discipline in using many other practices, such as test-driven development, coding standard and frequent integration. On the other hand, collaborative task ownership, test-driven development and daily meetings supported the use of PP.

The developers considered PP to be a contributing factor for the low defect counts in the system. This effect of PP has been identified also by others [2, 3, 4]. However, contrary to the literature, the navigator seldom found defects during the programming, meaning that some other aspect of PP, such as designing and test-driven development together, probably helped to avoid injecting defects.

The team spirit was very high, at least partially thanks to PP. Nobody was against PP and half of the developers liked it even more than solo programming.

7.2. Limitations

There were some factors that should be considered when generalizing the results of this study. All the developers were recruited based on their interest in using agile practices including PP, which can cause a positive bias towards the use of PP. The project started from scratch regarding the development process and practices. There was no old way of doing things that could have e.g. slowed down the adoption of PP. Because the project acted as a pilot for testing new practices, reflection of the practices and process measurement were more disciplined activities than in a typical project. The developed system was quite small and simple.

Much of the data is based on the opinions of the developers because only some things, such as defects, could be measured objectively. The reliability of the data was improved by having a researcher participate in the project. This gave a detailed insight into the project. In addition, the first author interviewed the team, did the data analysis and also discussed all results and conclusions afterwards with one more developer in order to ensure correct interpretation.

The participating researcher may have affected the team. However, he was a developer just as the others, who additionally collected experiences of all practices used in the project. He did not have a bias towards any particular practice, and therefore the effect of his participation to the team's use of PP should be negligible.

Measuring the knowledge of the system modules by asking about it of the developers and analyzing the knowledge transfer based on this data contains reliability problems. However, it provides at least some indicative data on a topic that is difficult to measure

Measuring productivity by LOC/hour may be misleading e.g. when doing lots of refactoring or copy and paste coding. However, if these kinds of issues are taken into account, its use to at least compare the changes in productivity between the iterations of the same project is justified.

7.3. Future Work

One must be careful when generalizing the findings of a single case study to other context. The research community needs to do new, detailed case studies and also experiment the use of PP in industry in many different contexts in order to provide better guidelines for organizations interested in adopting PP.

8. Acknowledgements

This study was part of the ITEA-AGILE research project. We would like to thank Nokia Technology Platforms for making the case project possible, and the coach Tuomo Kähkönen, the customer Jari E. Määttä, and the development team (Lasse Moisio and Timo Tenkanen from Affecto, and Seppo Sahi and Harri Korpi from HUT) for their participation in the study.

References

[1] L. Williams and R. Kessler, Pair Programming Illuminated, Addison-Wesley, 2002.

[2] L. Williams, The Collaborative Software Process, Ph.D. dissertation, University of Utah, 2000.

[3] J. Nosek, "The Case for Collaborative Programming", Communications of the ACM, 41(3), 1998, pp. 105-108.

[4] J. Vanhanen and C. Lassenius, Effects of Pair Programming at the Development Team Level: An Experiment, In Proceedings of the International Symposium on Empirical Software Engineering (ISESE2005), 2005.

[5] A. Cockburn and L. Williams, The Costs and Benefits of Pair Programming, In Extreme programming examined, Addison-Wesley, 2001, pp. 223-243.

[6] J. Nawrocki and A. Wojciechowski, Experimental Evaluation of Pair Programming, In Proceedings of the 12th European Software Control and Metrics Conference, 2001, pp. 269-276.

[7] M. Rostaher and M. Hericko, Tracking Test First Pair Programming – An Experiment, Extreme Programming and Agile Methods - XP/Agile Universe 2002, pp. 174-184.

[8] J. Wilson, N. Hoskin, and J. Nosek, The Benefits of Collaboration for Student Programmers, In Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education, 1993, pp. 160-164.

[9] K. Beck, Extreme Programming Explained, Addison-Wesley, 2000.

[10] M. Cusumano, A. MacCormack, C.F. Kemerer and B. Crandall, "Software Development Worldwide: The State of the Practice", IEEE Software, 20(6), 2003, pp. 28-34.

[11] H. Gallis, E. Arisholm, and T. Dybå, An Initial Framework for Research on Pair Programming, In Proceedings of the International Symposium on Empirical Software Engineering (ISESE2003), 2003.

[12] G. Luck, Subclassing XP: Breaking its rules the right way, In Proceedings of the Agile Development Conference (ADC'04), 2004.

[13] B. Greene, Agile Methods Applied to Embedded Firmware Development, In Proceedings of the Agile Development Conference (ADC'04), 2004.

[14] A. Pandey, N. Kameli and A. Eapen, Application of Tightly Coupled Engineering Team for Development of Test Automation Software – A Real World Experience, In Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC'03), 2003.

[15] L. Williams, Extreme Programming Practices: What's on Top?, Agile Project Management, Executive Report, 12(5), Cutter Consortium, 2004.

[16] R. Gittins, S. Hope, and I Williams, Qualitative Studies of XP in a Medium Sized Business, In Proceedings of the XP 2001 Conference, 2001.

[17] J. Aiken, "Technical and Human Perspectives on Pair Programming", ACM SIGSOFT Software Engineering Notes 29(5), 2004.

[18] J. Chong, Social Behaviors on XP and non-XP teams: A Comparative Study, In Proceedings of the Agile Development Conference (ADC'05), 2005.

[19] A.J. Dick and B. Zarnett, Paired Programming & Personality Traits, In Proceedings of the XP 2002 Conference, 2002.

[20] H. Sharp and H. Robinson, "An Ethnographic Study of XP Practice", Empirical Software Engineering, 9(1-2), 2004, pp. 353-375.

[21] A. Belshee, Promiscuous Pairing and Beginner's Mind: Embrace Inexperience, In Proceedings of the Agile Development Conference (ADC'05), 2005.

[22] S. Bryant, Double trouble: Mixing qualitative and quantitative methods in the study of eXtreme Programmers, In Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing, 2004, pp. 55-61.

[23] M. Ally, F. Darroch, and M. Toleman, A Framework for Understanding the Factors Influencing Pair Programming success, In Proceedings of the XP 2005 Conference, 2005.

[24] R. Jensen, "A Pair Programming Experience", CrossTalk, 16(3), 2003, pp. 22-24.

[25] S. Bryant, P. Romero, and B. du Boulay, Pair programming and the re-appropriation of individual tools for collaborative programming, In Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work, 2005.

[26] B. Tessem, Experiences in Learning XP Practices: A qualitative Study, In Proceedings of the XP 2003 Conference, 2003.

[27] R.K. Yin, Case Study Research: Design and Methods, 2nd ed., Sage Publications, 1994.