7 Pair Programming: Issues and Challenges

Kim Man Lui, Kyle Atikus Barnes, and Keith C.C. Chan

Abstract: Pair programming, two programmers collaborating on design, coding and testing, has been a controversial focus of interest as Agile Software Development continues to grow in popularity both among academics and practitioners. As a result of the many investigations into the effectiveness of pair programming in the last decade, many have come to realize that there are many hard-to-control factors in pair programming in particular and in empirical software engineering in general. Because of these factors, the results of many pair programming experiments are not easy to replicate and the relative productivity of pair and solo programming are still not fully understood. So far, it has been concluded by previous studies that pair programming productivity can vary, but few have shown how and why this is the case. In this chapter, we discuss a number of challenging factors in the adoption of pair programming and present an approach to deal with them. We discuss how different factors may affect our experimental outcomes and improve experiment design to reveal how and why pair programming can be made productive, at least, in controlled situations.

7.1 Introduction

Pair programming involves two programmers collaborating side-by-side while working on any type of programming task. These tasks include problem solving, system design, coding, and testing. Whether set of pair programmers or an individual is better for a specific programming task depends on several factors such as the capabilities of each programmer and the unique demand of each task. As the popularity of eXtreme Programming (XP) has increased (Kent 2000), the practice of pair programming has continued to draw much attention in the software community (Williams 2000).

Over the years, empirical studies in software engineering have consistently focused on how to determine which programmers are faster or more capable through assessment methods such as Program Aptitude Tests (PAT). Contrary to past programming studies, we use pair programming to better understand how the collaboration of programmers can build higher quality software in less elapsed time while also being adaptable to unpredictable changes in requirements and the challenges of personnel turnover. Therefore, we can expect the number of programmers with
pair programming skills to increase and the popularity of pair programming to continue its upward trend.

Many different variations of pair programming experiments have been reported but the results of these studies vary substantially (Williams 2000; Flor 1991; Nosek 1998; Nawrocki 2001; Hulkko 2005; Arisholm 2007; Ciołkowski 2002; Bellini 2005; Lui 2006; Lui 2008). This is mainly due to several consistent variables, which are difficult to control. The difficulties these variables imply extremely limit the ability of researchers to replicate previous experiments. Replication gives us the opportunity to justify pair programming and the chance to examine under what settings it could be adopted in the real world of software development. A recent meta-analysis by Hanney et al has recently been conducted, which reveals renewed perspective on many of these prior empirical studies (Hanney 2009).

Almost all empirical experiments consist of two core elements, subjects and the tasks they complete. The relationship between the elements is clear; the subjects perform the task so that we may observe the results. When we are interested in particular variables, we set up a control group and a treatment group. During the experiment one group is affected by the variables, which we are interested in, while the other group is unaffected in order to establish a baseline for comparison. After the experiment is conducted, statistics can be drawn from the collected data and used as a tool to understand our topic of interest.

When conducting a pair programming experiment, we must have two groups of subjects where one of the groups is composed of pairs. Each group is then presented with a number of tasks to complete. Human subjects can be assigned to groups using either random assignment (Nosek 1998; Nawrocki 2001; Bellini 2005) or a type of pre-test evaluation (Williams 2000; Arisholm 2007). Assigning subjects using either of these methods allows the results to be less affected by this variable. Most importantly, it establishes an initial basis for the results while maintaining consistency. After pairs have been formed, the experiment engages each set of subjects with a programming assignment. The tasks involved with these assignments may have different degrees of difficulty. When subjects are given tasks with varying degrees of difficulty; for example from simple to difficult or from difficult to simple, it must be noted that such factors may cause inconsistent results. The process of grouping people and task handling can become complicated; therefore replication becomes even more difficult.

The layout of this chapter is as follows: Section 7.2 discusses a classic Horse-Trading Problem. This exercise can be easily replicated to prove that collaborative problem solving makes a difference when comparing pairs and individuals. Section 7.3 reviews several initial studies that have impacted empirical research in pair programming. We go on to introduce other studies, which have resolved the problems of these initial studies, while introducing the new areas of research they have revealed. The section ends with a discussion on what outstanding issues should be explored in future pair programming experiments. Section 7.4 describes the process of Repeat Programming and how its experimental results can be used
to better understand what challenges lie ahead in the future of pair programming studies. Section 7.5 concludes our discussion.

7.2 Horse Trading Problem: Understanding Pair vs. Solo

Before introducing these empirical studies, we will begin with a simpler experiment that provides insight and understanding while justifying the legitimacy of the collaborative problem solving processes.

7.2.1 Human Subjects and a Simple Task

A well-known problem called the Horse-Trading Problem by Maier can help us explore collaborative problem solving. We have replicated this experiment; the following summarizes the preparation and results.

The Horse-Trading Problem is a simple question as seen in Figure 7.1.

**Fig. 7.1.** Classic Horse-Trading Problem

In 2006, we taught a course on eXtreme programming. One of the lectures focused on pair programming; as an exercise, the students were asked to solve a Horse-Trading Problem. The problem was part of a class activity intended to show the students the validity of collaborative problem solving. This experiment was carried out on students for learning purposes; therefore the experiment was not strictly monitored. Students were asked to solve the problem either alone or in pairs, they were also allowed to form their own groups with as many members as they preferred (see Table 7.1).

**Table 7.1.** Group Distribution

<table>
<thead>
<tr>
<th>Group Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Members per Group</td>
<td>10</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Afterward, each group was handed a piece of paper with the problem printed on it. Groups were allowed to refer back to the question as needed throughout the problem solving process, and were able to use as much time as necessary to solve the
problem. Finally, each group wrote down their respective answers and submitted it to us. We immediately checked the answers and revealed the results of the exercise to the class for discussion.

Most of the groups were able to work out a solution in around three minutes. The time needed to determine a solution is not considered, as the difference in solution times may be large in terms of percentage, but overall the time is not significant as these differences are too short. More importantly, though, not all groups were able to correctly solve the problem by calculating the amount the man actually earns.

### 7.2.2 Results

Our observations revealed significant improvements in the average percentage of accuracy when comparing a group with only one member to a group with more than three members. Groups 1 and 2 offer a strong statistical basis as evidence and such results are consistent with sociological research findings (Maier 1969) (see Figure 7.2). However, when the group size consists of five or more, the correction percentage dropped slightly. This result could be related to ergonomics as the classroom seats arranged in fixed rows. A group of two may be able to communicate side-by-side effectively, but for groups larger than three a round table is needed to facilitate communication and collaboration.

![Collaborative Problem Solving Results](image)

**Fig. 7.2.** Collaborative Problem Solving Results
7.2.3 Discussion

Although the Horse-Trading Problem is simple, it brings to light several benefits. First, the Horse-Trading Problem experiment is easily replicated. This allows other researchers, like us, to quickly verify the results by repeating the experiment. Using such methods can steadily advance our knowledge, thus demonstrating that the previous works of others provides a strong foundation for the exploration of potential improvements. Finally, any such experiment should quickly resolve any debate concerning the validity of previous findings.

Second, the experiment stimulates us to consider an interesting quandary. Why is the Horse-Trading Problem easy to replicate, but a specific pair programming experiment not? Pair programming is simple in principle, but to create such an experiment is challenging. In order to complete the Horse-Trading Problem, one only needs an elementary education in basic reading and arithmetic; however, programming requires a unique skill set and experience which must be developed over time. A pair programming experiments requires that human subjects be classified according to their level of ability. A subject’s skills and experience in pair programming is a significant factor when considering the results of any such experiment. Additionally, the task to be completed in the Horse-Trading Problem is solid and limited to one possible answer. We know from programming that any programming assignment has unlimited solutions. If programming tasks in pair programming experiments could be limited to one specific activity, for example writing a program to compute the average of a set of numbers, then pair programming experiments by others could more easily be replicated. Therefore, when we consider the Horse-Trading Problem, we realize the task is specific. Solving the task requires no particular skill set and hence we need not be concerned with how the subjects pair. This is the reason why the Horse-Trading Problem is easy to replicate. However, how the subjects pair up and the methodologies used are significant considerations in any pair programming experiment.

The Horse-Trading Problem is not related to programming, but this experiment gives us insight and goals for empirical studies in pair programming. We should bear in mind that pair programming experiments should be designed so that they can be easily replicated and validated, even though such objectives may not be easily achieved in reality.

The Horse-Trading Problem is beneficial to people who are learning pair programming or pair programming coaches that are looking for a significant result, which clearly demonstrates the value of collaborative problem solving.

7.3 Pair Programming Studies

This section investigates five major pair programming experiments. Although three of these experiments were executed over a decade ago, it does not undermine their value. Our key interest is to understand the handling of variables in each ex-
periment, so that by the end of our discussion we can develop a list of variables, which will have the strongest affect on the results of future studies.

Before discussing these empirical experiments, we must point out that limited sample sizes may seem to be a critical factor. However, despite the sample size, we still may find that the statistics obtained are significant. Nevertheless, some may argue that the sample size may cause the results to be misleading. According to Miller (Miller 1986), the results of an experiment can be more informative with a small number of subjects rather than having a large set of statistical data from a larger group of subjects.

The key focus of our discussion is to determine the effect of pair programming combinations and the difficulty of the tasks to be completed.

### 7.3.1 Relative Effort Afforded By Pairs: REAP

There are two important methods that can be used to measure productivity achieved over a period of time when undergoing a pair programming experiment. We can consider the elapsed time to complete a task or the total effort divided by the time required for the programmers to complete the task. Lui and Chan (Lui 2006) developed a single measurement to include both of these methods. This measurement is known as Relative Effort Afforded by Pairs (REAP). In order to better understand the results of previous pair programming studies, each of their results have been converted into REAP equivalents.

\[
REAP = \frac{(f_{\text{inish\_time\_of\_pair}}) \times 2 - (f_{\text{inish\_time\_of\_individual}})}{f_{\text{inish\_time\_of\_individual}}} \times 100\%
\]

All together there are five possible results when computing REAP. Each result gives us a quantitative value to measure both productivity and time in a pair programming session.

(i) REAP less than 0: The total time / effort needed for pair programmers is less than the time needed for an individual programmer. Using pair programmers is less costly than individual programmers; therefore pair programming is actually more efficient than a single programmer.

(ii) REAP approximately 0: Pair programming halves the total time / effort required for an individual programmer. Opportunities and product life cycles have become much shorter over the past few years. In order to take advantage of premium pricing and the advantages of early movement in a market, such situations make it worthwhile for companies to focus more on short-term development costs (Kalantone 2000).

(iii) REAP greater than 0 but less than 100: Pair programmers require more total man-hours / effort, but the pair is still faster than an individual programmer. As the elapsed time for pairs is less than individual programmers, such situations can
be useful when time-to-market is a critical issue (Williams 2000; Nosek 1998). Division and reallocation of programming tasks involves planning and a significant amount of time. Pair programming allows for individual programmers to group their tasks and work in parallel. Overall, it provides an alternative method to speed up the development process.

(iv) REAP approximately 100: Programmers engaged in pair programming need about the same amount of time as an individual programmer. Pair programming, therefore, doubles the total-man hours / effort in comparison to individual programming.

(v) REAP greater than 100: Pair programmers need more time / effort than an individual programmer.

7.3.2 Nosek

In 1998, Nosek reported the first quantitative findings involving pair programming. His empirical experiment involved fifteen full-time system programmers. Five sets of pair programmers and five individual programmers were asked to write a UNIX script that performed a database consistency check (DBCC) on a Sybase database.

Throughout the course of the experiment, the subjects wrote a program that would start the consistency check, review the database log for errors, and finally send a warning e-mail if any errors were detected. After executing the DBCC command, the status of the database would be returned in a log file.

Nosek’s results in terms of REAP are shown in Table 7.2. In order to determine the quality of the software, two independent graders evaluated the readability and functionality of each pair’s or individual’s code. Readability was assigned a score between 0 and 2: 0 for an unreadable solution, 2 for a readable solution, and 1 for those cases in between. Functionality was assigned a score between 0 and 6. Those solutions with a score of 0 did not achieve the goal at all, while those receiving a score of 6 achieved the goal in its entirety. Although the two graders examined the code of all the subjects with an inter-grader reliability of 90%, it must be noted that human judgment was involved in verifying the quality of each piece of software.

Nosek’s experiment forms five sets of pair programmers, but how these pairs were made was not taken into consideration. As for the task, the process itself should be straightforward, especially for professional system programmers. In this case, we can suspect that the only area where the system programmers lacked experience was the use of DBCC commands. Nevertheless, it is impossible for us to understand how difficult this task was for the subjects. In short, the initial study reveals positive results, which should lead us to explore further issues in pair programming. Specifically, two variables need further consideration – the formation of pairs and the level of task difficulty.
7.3.3 Williams

In 2000, Williams reported on a pair programming experiment involving forty-one junior and senior university students. Each of the students was randomly assigned to work either as a pair programmer (total 14 pairs) or a single programmer (total 13 singles). The student subjects were taught how to use Active Server Page (ASP), and then asked to write web scripts with dynamic content. The project involved querying and updating a Microsoft Access Database. Williams’ students had little to no experience with ASP, however, they all had approximately three years of programming experience with C++. The application developed was similar to typical e-commerce web site solutions (Williams 2000).

Although students were assigned to be pair programmers randomly, Williams argued that the distribution of many different spreads is better than randomization when working with pairs in a pair programming experiment. Therefore, pairs were arranged according to their GPAs. The spread of distribution included high-high, high-average, high-low, average-average, average-low, and low-low pair groupings. The REAP was approximately 15% on average.

All the students, both pairs and singles, completed four assignments over a six-week period. The students recorded their time spent on the project using a web-based tool. It should be noted that the experiment was not monitored by any on-site research assistant, nor were the pairs monitored to confirm how closely they practiced pair programming. In order to test each assignment, a teaching assistant executed automated testing to analyze the program’s quality. In general, pair programmers passed more automated post-development test cases compared to single programmers.

After the first assignment, REAP was 60%. However, after the initial adjustment period, the total hours the pairs needed to complete the second and third assignments decreased substantially. REAP was then 15% on average (Williams 2000). Due to data entry problems, the completion times for the fourth assignment were not accurate. Thus, the results and time performance of the fourth assignment were not reported.

In terms of quality, pairs passed the test cases within the following range: 86.4-94.4%, whereas singles only passed the test cases 70.4-78.1% of the time (see Table 7.2).

Williams’ experiment is extensive and has taken into consideration several variables that were not considered in Nosek’s experiment. The pairing situation has been done randomly to determine singles and pairs, and further pretesting (GPA) was used to arrange pair formations.

Strictly speaking, one may argue that the experiment took place in too loosely of a controlled environment. As the students were not monitored, we cannot be sure that some of the students did not work on the project while at home. Another factor is the four assignments and their order of completion. We can assume that each task was given in order (1, 2, 3, 4), but we must consider the order of the tasks and their relative level of difficulty. Williams’ past work has also observed
the effects of pair learning and its significance. Therefore, one should not ignore
the effects concerning the order of tasks and their difficulties.

Another issue is re-submission, up until all test cases have passed. Williams’
experiment does not allow for the student to re-submit their work. Therefore, how
each test case was written to measure software quality would be a significant fac-
tor. We will come back to re-submission to facilitate quality measurement in pair
programming experiments in Section 7.3.4.

Finally, the combination of novice-novice programmers vs. expert-expert pro-
grammers has not been considered as all the students represented only novice pro-
grammers.

In summary, Williams resolved many of the problems introduced by Nosek’s
experiment. Furthermore, her work opened us up to explore the above mentioned
problems in pair programming experiments. Such factors must be given clear con-
sideration in future experiments. Topics such as the combination of tasks, the ef-
facts of novice-novice pairs vs. expert-expert pairs, and re-submission to insure
basic level of software quality are good areas for further exploration.

7.3.4 Nawrocki and Wojciechowski

In 2001, Nawrocki and Wojciechowski conducted an experiment on twenty-one
fourth-year university students. The students were divided into three groups and
asked to complete an assignment using one of three different methodologies. Methodology 1: five pairs using XP (referred to as pair XP); Methodology 2: five
singles using XP but without pair programming (referred to as single XP); and
Methodology 3: six singles using Personal Software Process (PSP). Nawrocki was
primarily interested in two specific comparisons involving the three methodolo-
gies: single XP vs. PSP and single XP vs. pair XP. However, in this chapter, we
are only interested in the comparison of pairs and singles using the same metho-
dology. Therefore, we have selected single XP vs. pair XP for our review.

All of the subjects had more than two years of formal study in C and C++ pro-
gramming. They were asked to write four programs, which would find the mean
and standard deviation of numerical data samples. Additionally, the program
would determine the linear regression parameters, count the number of lines out-
put in the program, and count the total lines of code (LOC) used to create the pro-
gram.

The results of the experiment revealed that singles and pairs needed around the
same amount of time to complete the first three assignments. Each assignment
needed 2.4, 1.3, and 2.4 hours respectively (Nawrocki 2001). Therefore, REAP is
equal to 100%. During the fourth assignment, pairs needed 3.5 hours and singles
needed 4.3 hours, which results in REAP equal to 63%. Although the improve-
ment seems to support pair programming, Nawrocki remarked that such improve-
ment was due only to the fact that the single programmers misunderstood the pro-
gram requirements. Additionally, the number of re-submissions was also counted,
as groups were required to re-submit their work until their program had no errors and passed acceptance testing.

The experiments of Nawrocki and Williams have similarities but several design factors differ. Group pairs have been formed using different methods and the degree of difficulty has been altered. The most interesting point is that Nawrocki’s results do not support pair programming, whereas Williams’ results do. It should be noted, however, that the subjects are monitored throughout the entire process in Nawrocki’s experiment, thus giving the data a higher probability of accuracy.

Another key factor is required re-submission. The need for re-submission greatly depends on how the test cases are written. We cannot say that all test cases are fairly used to measure the quality of a program, as a test cannot measure the absence of faults. For example, Program A is more robust but unfortunately neglects a few points addressed by the test cases, while Program B is less robust but just happens to better cover the test cases. Therefore, any successful program in an experiment depends on test cases for both functionality and exception handling. However, it should be noted that we should not regard this as a major factor as we believe the experimenter should design quality test cases while planning the experiment.

As long as a pair may re-submit, even if the program fails the first test case, the program can still be fully reviewed in order to determine its true quality.

7.3.5 Arisholm et al

In 2007, Arisholm et al conducted a pair programming experiment involving 295 subjects who were hired from 29 software companies in Norway, Sweden, and the UK. The subjects were grouped by their organization as well as their performance based upon a pre-test. The pre-test involved a maintenance task, which was simpler, but similar to the actual experimental task (Arisholm 2007). The experimental task was a change task focusing on seven class and 354 lines of code. Pairs were grouped according to their level of expertise (junior-junior, intermediate-intermediate, and senior-senior). Arisholm’s experiment was large scaled, well planned, and successfully implemented. The results in terms of REAP are show in Table 7.2.

Arisholm et al showed that pairs did not increase correctness or significantly reduce the total elapsed time. An important factor in this experiment was that the subjects were limited to one type of task: a change task. This is substantially different from experiments where subjects are asked to write programs from scratch (Williams 2000; Nosek 1998; Nawrocki 2001; Hulko 2005; Lui 2006). Maintenance tasks, however, are advantageous as they can easily be measured in a more objective way. For example, subjects can attempt to make a minimum number of changes to complete the task, and then the number of lines modified and added can be counted. When the experimental task is a development task, each group of subjects has the opportunity to create a different design or implementation of the
solution. Evaluation of design and implementation in comparison with other subjects is difficult and the process can be quite subjective. Therefore, the task is usually restricted to specific requirements to cover these issues and/or the final product must pass several pre-defined test cases.

The major difference between Arisholm and previous experiments (Williams 2000; Nosek 1998; Nawrocki 2001) is that the task involves a maintenance task rather than a development task. Nevertheless, we still cannot easily conclude that pair programming, as it has been originally addressed, is not only concerned with writing code but also with the program’s greater design. In Arisholm’s experiment, programmers need to understand the design before they can begin the maintenance task; therefore they may run the original program several times to determine what should be changed as it is impossible to test unwritten software. More importantly, Arisholm’s experiment is more realistic for the software industry as many programmers are responsible for maintaining code rather than writing original code (Lui 2008).

Over the duration of Arisholm’s experiment, many different Java development tools were used (i.e. JBuilder, Eclipse, IntelliJ, NetBeans, Emacs, and Javac). The decision to use certain tools was left up to the discretion of the subjects. The subjects’ familiarity with the tool and the tool’s ability to affect productivity or facilitate Java program maintenance was not measured. For example, Eclipse is a popular Integrated Development Environment (IDE) for Java while Emacs is multi-OS text editor. This factor increases the complexity of the pair programming experiment, as the use of a development tool or tools is an uncontrolled variable.

### 7.3.6 Lui, Chan, and Nosek

An individual’s performance on a programming aptitude test (PATs) has proven to be a strong indicator of their future programming performance (Bateman 1973; Denelsky 1974; Tukiainen 2002) therefore many organizations have adopted such tests to pre-screen potential programming candidates before their interviews.

Lui, Chan, and Nosek conducted two controlled experiments with full-time professional programmers. The subjects worked on increasingly complex programming aptitude tasks related to problem solving and algorithmic design. Each experiment revealed that pairs significantly outperformed individuals, thus providing solid evidence for the value of pairs in program design-related tasks.

These experiments brought up a number of questions: Is pair programming better for design tasks rather than implementation tasks, or vice-versa? Or is pair programming better for both? Design and implementation are two separate processes when considering traditional software development, but the nature of pair programming combines both design and implementation. Therefore, if pair programming is good for design, then implementing that design is not only easy, it is also a natural process. This is an important idea for the future exploration of pair programming.
Table 7.2. Summary of Pair Programming Experiments

<table>
<thead>
<tr>
<th>Subjects</th>
<th>Nosek</th>
<th>Williams</th>
<th>Nawrocki and Wjciechowski</th>
<th>Arisholm, et al.</th>
<th>Lui, Chan, and Nosek</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample Size</td>
<td>15</td>
<td>41</td>
<td>21</td>
<td>295</td>
<td>15</td>
</tr>
<tr>
<td>Same Organization</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Day Split</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Pre-Test</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Subjects already know algorithms needed to solve problems before the experiment</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Not Examined</td>
<td>No</td>
</tr>
<tr>
<td>Tasks</td>
<td>Simple</td>
<td>Moderate</td>
<td>Simple</td>
<td>Moderate</td>
<td>Programming</td>
</tr>
<tr>
<td>Development Task</td>
<td>Development Task</td>
<td>Development Task</td>
<td>Maintenance Task</td>
<td>Aptitude Test (PAT)</td>
<td></td>
</tr>
<tr>
<td>Quality assessment</td>
<td>Rated by two independent graders</td>
<td>Post-development test cases</td>
<td>Pass all predefined test cases</td>
<td>Correctness analysis by two independent senior consultants</td>
<td>Pass all PAT questions</td>
</tr>
<tr>
<td>REAP (%)</td>
<td>41.7</td>
<td>15</td>
<td>100</td>
<td>84</td>
<td>-1.8 ~ 21</td>
</tr>
<tr>
<td>General Conclusions</td>
<td>PP is productive</td>
<td>PP is productive</td>
<td>PP is not productive</td>
<td>PP is not productive</td>
<td>PP is productive</td>
</tr>
</tbody>
</table>

Lui and Chan have proposed an approach to better understand pair programming known as Repeat Programming. This information is not shown in Table 7.2 as this experiment will be discussed in Section 7.4.

7.3.7 Outstanding Issues

There are many other studies that consider grouping and tasks in several different contexts. No matter the context, we must focus on subjects and tasks. Therefore, the fundamental question to answer is: *How should we group the subjects and set their respective tasks?* Once we have clearly answered this question, we will have...
a solid foundation on which to move forward. Let us consider the following four major factors:

Pairing of Programmers: The grouping of subjects into pairs and individuals is a non-trivial matter; however, how this is handled can greatly affect the outcome of the experiment. As discussed above, researchers have already recognized this fact. Therefore, they have grouped subjects according to their organization or class, and then allowed them to freely select their partners (Ciolekowsk 2002). Yet other researchers have used randomization to ensure a thorough distribution of programmer characteristics between groups of pair programmers and individuals (Nosek 1998; Nawrocki 2001; Bellini 2005). It should be noted that we are looking for general all purpose cases in empirical studies on pair programming. The determined general cases will be the best alternative in many, but clearly not all, situations. We are not seeking universal applicability; as to do so only undermines human nature and the essence of the software engineering industry.

Programming Tasks: Designing appropriate programming tasks can be more challenging than the pairing of programmers. For example, in experiments that evaluate the performance of pairs against individuals in non-programming related tasks, it is easier to control variables and reproduce the original experimental results (Forsyth 1999). In pair programming experiments, tasks are more complicated and entail the programmers to understand requirements, software design, coding, testing, and debugging. Still such tasks must focus either on Design Difficulty or Implementation Difficulty, and the results of any task must be measurable. Pair programming makes this difficult to achieve, as programming tasks can never be as specific as the horse-trading problem. The complexity of programming tasks is not the issue, rather there is not one single task that can be used in pair programming, which can objectively measure pair performance as well as the task’s relative complexity. Such factors are representative of the industry and its constant challenges.

Combination of Programming Tasks: Measuring the degree of difficulty for one programmer is already difficult. However, measuring the degree of difficulty for a group of subjects with varied skills and experience is even more challenging. Therefore, we must consider how to develop different programming assignments that can be distributed amongst a group of subjects that will lead to consistent results. Consider three tasks that are completed in succession (A -> B -> C), which steadily become more difficult. Now consider the same three tasks completed in reverse (C -> B -> A) where the tasks only become much easier. The combination of these tasks will inevitably have a large effect on the overall factor of pair learning.

Programming Hidden Variables between Programming Tasks: When discussing pair programming tasks we must determine a fair basis for comparison. Simply using the control group (i.e. individual programmers) as the baseline for comparison could be problematic and should be dealt with carefully. A more appropriate baseline includes the combination of both subjects and their tasks. In this case, all of the above-mentioned factors (i.e. 1 – 3) should be used to create
the baseline of the experiment; otherwise pair programming experiments cannot be accurately measured.

If a set of pair programmers is late in completing task A, then we can assume that they will be late for task B. This of course does not make much sense, the same as a set of pair programmers that are fast in completing task A must then be fast for task B. In order to justify such assumptions, we must clarify the correlation, if any, between tasks A and B. The key factor is the similarity of the tasks. Once the relationship between tasks A and B is established, this relationship then becomes the baseline.

In most cases there are distinct differences that exist or take place between tasks. These differences could involve a change of programming languages, the introduction of specialized domain information, or simply the fact that the set of pair programmers had a cup of coffee before taking on the next task. Each difference is a type of hidden variable, which must be clearly considered and in all possible cases controlled throughout the pair programming study. If such hidden variables are ignored, then the baseline can never be fixed.

Pair programming with partner rotation: Recent innovations in pair programming methodology consider many programmers pairing together and then rotating partners based upon a fixed constraint (i.e. time interval or completion of a number of classes). In such situations, the effect of pair jelling (Williams 2000) and the time needed to jell must not be discounted. Nevertheless, this issue creates yet another unique situation needing further study and investigation. However, before the effects of partner rotation can be fully considered, the four issues above must be understood. Thus, those issues are the focus of this chapter.

7.4 Repeat Programming

In 2005, we conducted an experiment known as “repeat programming.” In this experiment subjects were asked to write the same program several times. The experiment was not meant to simulate a real world situation; therefore it was a controlled lab experiment. In reality, a programmer will never rewrite a program exactly the same way. Actually, programmers actively solve problems of similar nature using design patterns they have developed from their own experiences. As repeat programming does not simulate an industrial environment, its purpose is to create a controlled environment to assess the productivity of pair programming versus solo programming.

Repeat programming can be considered as a method for measuring programmer experience and how this experience is affected throughout the process of repeat programming. It should be noted, however, that the experience of solving a particular problem many times should not be misconstrued with “experienced programmers” (i.e. years of experience, skills, etc.).
7.4.1 Experiment

The above experiment was conducted in 2004 and used part-time masters students with full-time programming jobs as subjects. All together there were thirty-nine subjects who were taking the course, “Agile Software Development and eXtreme Programming”, as a requirement for their double Masters Degree jointly organized by the Hong Kong Polytechnic University (Hong Kong) and The Graduate School of Chinese Academy of Science (Beijing).

The students were divided into groups of threes. Subjects were placed with other programmers with “nearly-identical” abilities. This way any disparities in programming ability would be minimized. In order to determine each programmer’s ability, a pre-assessment test was taken by each programmer. The test consisted of fifty multiple-choice questions, which were taken from Munzert’s (1994) computer aptitude test.

The experiment incorporated three major phases:

1) Select capable and “nearly-identical” subjects to group (pre-assessment).
2) Familiarize subjects with the practice of Pair Programming.
3) Have subjects repeatedly write the same program.

As none of the subjects had any formal experience with pair programming, it was mandatory for groups to practice pair programming for at least four hours before being able to work separately. As a warm-up exercise, each group was asked to write Tower of Hanoi, a simple puzzle game. This pre-experiment was not recorded; its only intention was to ease the subjects into a pair-programming environment. Each group determined their own organization of pair and single programming amongst the three members. Thus, this minimized the impact on those group members who may have simply preferred solo programming or had any bias against pair programming. Furthermore, to minimize the possibility of conflict amongst group members, we suggested that in situations of disagreement, the final decision was to be made by the group member currently controlling the keyboard/mouse. All of the subjects fully understood that internal conflict and self-assertion would only reduce productivity and run counter to the objective of the experiment.

The task asked the subjects to write a FIFO warehouse application with the following requirements: in/out operations, reserved stock, bin management, and good return using SQL Server and ASP (or JSP if subjects were not as familiar with ASP). As the subjects were already full-time programmers, they were all familiar with popular programming languages and scripts like SQL and ASP (or JSP). The requirements of the tasks were determined during our initial study; however, we simplified the requirements so that the program could be completed in a shorter amount of time. More specifically, the program was to be completed over a weekend (approx. two days) as all the subjects had full-time jobs. The duration of the experiment was eight weeks, thus groups selectively chose four weekends over the
eight-week period to complete the program repeatedly during four separate sessions. During each session the subjects were responsible for keeping their own total elapsed time.

Fifty test cases were written and used to assess the quality of each group’s software. Many test cases were used in order to assess each program’s quality objectively, rather than subjectively using human judgment. Each group’s software had to pass all fifty test cases before the program was considered to be satisfactory. The test cases involved specific application requirements as well as exception handling. Such measurements were appropriate for two major reasons. First, quality could be objectively measured rather than by relying on human subjectivity. Secondly, from a client’s perspective, users would be more satisfied with software products that have undergone extensive testing and would regard them as higher quality. Traditionally, developers and customers tend to see software quality differently.

7.4.2 Result

After the first round of programming, the individuals completed the program in 635 minutes on average. Predictably, after the second round they accomplished the task much faster. Clearly, they were able to shorten their learning curves, especially in the areas of design and algorithm formation. Upon inspection of each version of their work, we found that design for the most part remained the same; however, syntax, naming standards, and ordering of statements were different in each version.

7.4.3 Discussion

The implementation of repeat programming provides substantial feedback to the design of the software as there is no up-front design phase. Therefore, in our experiment, the steepness of the slope reveals a stronger influence on design throughout the coding process, rather than coding solely for implementation. These results have a direct implication on the impact of pair design due to the difference in steepness between the two slopes. Therefore, we can conclude that pair programming is better for design; otherwise if there was no effect on design, then the steepness of slopes for pair and solo should be more similar. During the first and second iterations the focus for both pairs and solos is the design, whereas during the third and fourth iterations the focus is coding. Now, let us once again consider the four major factors and how they relate to repeat programming:
Summary of Repeat Programming Results

**Pairing of Programmers:** As described earlier, our goal in a pair programming experiment is to ensure at least one of two issues. Programmer ability amongst subjects must be either well distributed between all groups or each subject’s ability must be matched to their respective group members. Repeat programming groups programmers with similar abilities as determined by the pre-assessment test. Thus we avoid any issues that may arise due to ambiguous pairing and no extremes will be found in any pair combination.

**Programming Tasks:** In repeat programming we can focus less on the task and more on the interaction of pairs as the task never changes throughout the experiment. Programmers may be considered as novices with the program during the first iteration, but by the end of the fourth iteration, they all become experts.

**Combination of Programming Tasks:** In repeat programming, the level of task difficulty steadily becomes easier as the subjects continually repeat their design. Therefore in the experiment the affects of pair programming are minimized as the order of tasks does not matter.

**Programming Hidden Variables between Programming Tasks:** In repeat programming, a programmer’s capability is not as important as the fact that the program is repeated numerous times, which enables the programmer to inactively learn the program while focusing on collaboration. Thus, as we are able to control the hidden variables we can easily see the differences between pairs and solos without depending on the expertise of the programmers. Most importantly, we must establish a fair baseline. Initially we use pair vs. solo as a baseline; this is

<table>
<thead>
<tr>
<th></th>
<th>1st Round</th>
<th>2nd Round</th>
<th>3rd Round</th>
<th>4th Round</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solo (mins)</td>
<td>635</td>
<td>407</td>
<td>334</td>
<td>285</td>
</tr>
<tr>
<td>Pairs (mins)</td>
<td>410</td>
<td>320</td>
<td>281</td>
<td>272</td>
</tr>
<tr>
<td>REAP</td>
<td>29%</td>
<td>57%</td>
<td>69%</td>
<td>91%</td>
</tr>
</tbody>
</table>

**Fig. 7.3.** Summary of Repeat Programming Results
good, but not solid enough to measure the effects of the consecutive iterations. Repeat programming establishes a standardized baseline after the first iteration, thus the slope drops considerably as we move to the second iteration. As the iterations continue we can clearly see the slope steadily decrease, thus further enhancing the baseline. This provides us a clearer framework for future pair programming experiments.

### 7.5 Conclusion

So far, pair programming experiments have yet to deal with all four of the factors mentioned above. In order to move forward in empirical pair programming studies we must make it our goal to resolve these factors and the greater issues they introduce. The Horse-Trading Problem we first introduced can be used as a contrast with collaborative programming situations, but the Horse-Trading Problem does not deal with all the factors that emerge during pair programming experiments. Repeat programming, on the other hand, considers the four main factors and gives us better insight on how we can adjust our baseline for future contributions in empirical pair programming experiments.

Repeat programming gives us insight to explore these four factors; unfortunately though, it will be better to design an experiment that mimics a real world situation while taking all four factors into consideration. Pair programmers, coaches, and researchers alike will be confident of our understanding in order to explore more issues, including extensions of pair programming experiments such as Side-by-Side Programming and Software Process Fusion.

Finally, we are not saying that pair programming and its applicability should limit its focus only to these four factors. However, we should pay close attention to these four factors because of their potential to affect our experimental results. Empirical studies such as Arisholm et al, which consider new approaches such as maintenance tasks in pair programming, and meta-analytical studies like those of Hanney et al are key to understanding the effects of these four major factors (Arisholm 2007; Hanney 2009).

### References


**Author Biographies**

Kim Man Lui has worked in a number of IT positions in the commercial sector from system engineer, analyst programmer, system analyst, project leader and IT manager. Dr. Lui was a certified Sybase Database Administrator in 1996, a certified Oracle Database Administrator in 1999 and a SUN certified Java Programmer in 2002. Dr. Lui is an author of three books: two books written in Chinese on CMM and Agile Software Development in Beijing SPI and *Software Development Rhythms*, Wiley, 2008. The book is being translated in Chinese and is scheduled to be published by Publishing House of Electronics Industry.

Kyle Atikus Barnes is a professional software engineer working in both the United States and Asia. Mr. Barnes completed his Bachelor of Arts Degree in Business Administration and International Studies from Queens University of Charlotte, North Carolina (2005), and recently graduated with distinction from Hong Kong Polytechnic University (2010) with a Master of Science Degree in Information Technology focusing on Agile Software Development. Mr. Barnes has worked in numerous industries including Travel, Logistics, and Education. Currently, Mr. Barnes works in implementing agile practices and methodology as Owner / Manager of Seattle Software Solutions, LLP.

Prof. Keith Chan received a B.Math. (Hons.) degree in Computer Science and Statistics in 1984 and a M.A.Sc. and Ph.D. degree in Systems Design Engineering from the University of Waterloo, Ontario, Canada in 1985 and 1989 respectively. He had worked as a senior analyst at the IBM Canada Laboratory, Toronto, Canada and as an Associate Professor at the Department of Electrical and Computer Engineering at Ryerson University, Toronto, Ontario, Canada. He joined the Hong Kong Polytechnic University (PolyU) in 1994 and is currently a Professor and Head of the Department of Computing. He is a Guest Professor of the Graduate School, the Chinese Academy of Sciences (CAS), Beijing and is also the Co-Director of the CAS-PolyU Joint Software Engineering Laboratory. Prof. Chan’s research interests are in software engineering and data mining. Prof. Chan is active
in consultancy and has served as consultant to government agencies as well as large and small to medium enterprises in Hong Kong, China, Singapore, Malaysia and Canada.