Integrating Pair Programming into a Software Development Process

Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
williams@csc.ncsu.edu

Abstract

Anecdotal and statistical evidence [1-3] indicates that pair programmers -- two programmers working side-by-side at one computer, collaborating on the same design, algorithm, code or test -- outperform individual programmers. One of the programmers, the driver, has control of the keyboard/mouse and actively implements the program. The other programmer, the observer, continuously observes the work of the driver to identify tactical (syntactic, spelling, etc.) defects, and also thinks strategically about the direction of the work. On demand, the two programmers can brainstorm any challenging problem. Because the two programmers periodically switch roles, they work together as equals to develop software. This practice of pair programming can be integrated into any software development process. As an example, this paper describes the changes that were made to the Personal Software Process (PSP) to leverage the power of two programmers working together, thereby formulating the Collaborative Software Process (CSP). The paper also discusses the expected results of incorporating pair programming into a software development process in which traditional, individual programming is currently used.

1. Introduction

The practice of pair programming is not new. In his 1995 book, "Constantine on Peopleware," Larry Constantine reported observing *Dynamic Duos* at Whitesmiths, Ltd. producing code faster and more bug-free than ever before [4]. That same year, Jim Coplien published the "Developing in Pairs" Organizational Pattern [5]. In 1998, Temple University Professor Nosek reported on his study of 15 full-time, experienced programmers working for 45 minutes on a challenging problem, important to their organization, in their own environment, and with their own equipment. Results showed that pair programming improved both their performance and their enjoyment of the problem solving process [1]. In 1999, a larger experiment at the University of Utah supported these findings [3, 6, 7]. (This experiment will be discussed in detail below.) Recently, industry use of pair programming is on the rise.

1.1 Industry Use of Pair Programming

While small groups of pair programmers can be found in many companies, both large and small, the largest known group of pair programmers is those that practice the eXtreme Programming (XP) methodology. In 1996, XP [8] started evolving. XP was developed initially by Smalltalk code developer and consultant Kent Beck with authors Ward Cunningham and Ron Jeffries. XP is a lightweight, yet disciplined, software development methodology. The methodology's success rate is so impressive that it has aroused the curiosity of many highly-respected software engineering researchers and consultants. Although departing significantly from traditional development practices, anecdotally, XP appears to be very effective.

Additionally, programmers report that developing with XP practices is much more exciting and enjoyable than with traditional processes.

XP's requirements gathering, resource allocation, and design practices are a radical departure from more traditional methodologies, such as PSP or the Rational Unified Process [9]. Customer requirements are written as fairly informal "User Story" cards, a rough estimate of required resources is assigned to the cards, these are assigned to a programming pair, and coding begins. With no *formal* design procedures or discussions on overall system planning or architecture, the pair determines which code in the ever-enlarging code base needs to be added or changed and then does it, without asking anyone "permission." This practice requires the use of "Collective Code Ownership" whereby any programming pair can modify or add to any code in the code base, regardless of the original programmer.

Programming pairs routinely "refactor" the code base by continuous change and enhancement. They view the code as *the* self-evolving design – they do not spend time on a design document and, therefore, have strict self-documenting code style and comment guidelines. XP also has particularly thorough testing procedures. Comprehensive test cases are written and automated prior to actual code changes. The results of running these automated new tests and previous, regression test cases determine if the change/enhancement to implement a User Story has been done correctly without harming the implementation of other User Stories.

XP credits much of their success to their use of pair programming by <u>all</u> their programmers, experts and novices alike. XP advocates pair programming with such conviction that even prototyping done solo is scrapped and re-written with a partner. Working in pairs, the engineers perform a continuous code review, noting that it is amazing how many obvious but unnoticed defects another person at your side notices. Additionally, because no formal architecture or design is produced, much of this "documentation" lives in the heads of the programmers. By continuously practicing pair programming and by rotating the partners that work together daily (or more often), the programmers are able to pass system structure and system knowledge around the team.

> *When an important new bit of information is learned by someone on the team, it is like putting a drop of dye in the water. Because of the pairs switching around all the time, the information rapidly diffuses throughout the team just as the dye spreads throughout the pool. Unlike the dye, however, the information becomes richer and more intense as it spreads and is enriched by the experience and insight of everyone on the team. [8]*

Additionally, pairs have been found to be more likely to actually perform some of the less popular practices of the methodology. "Under stress, people revert. They will skip writing tests. They will put off refactoring. They will avoid integrating. With your partner watching, though, chances are that even if you feel like blowing off one of these practices, your partner won't . . .the chances of ignoring your commitment to the rest of the team is much smaller in pairs than it is when you are working alone." [8] (This implicit pressure that pairs put on each other will be discussed more later in this paper.) Because XP is a **lightweight** yet **disciplined** methodology, pair programming seems the glue that holds it together.

1.2 Experimentation with Pair Programming in the Software Engineering Classroom

In 1999, a senior Software Engineering class at the University of Utah was structured as an experimental class. The experiment was designed to study the costs and benefits of pair programming. The experimental class consisted of 41 juniors and seniors. The students learned of the experiment during the first class. Generally, the students responded very favorably to being part of an experiment.

On the first day of class, the students were asked if they preferred to work collaboratively or individually, whom they wanted to work with, and whom they did not want to work with. Thirty-five of the 41 students (85%) indicated a preference for pair programming. (Later, many of the 85% admitted that they were initially reluctant, but curious, about pair programming.) Therefore, it must also be noted that the majority of those involved in the study were self-selected pair programmers. Further study is needed to examine the eventual satisfaction of programmers who are forced to pair program despite their adamant resistance.

The students were also classified as "High" (top 25%), "Average," or "Low" (bottom 25%) academic performers based on their GPA. (The GPA was not self-reported; academic records were reviewed.) Using this information, the twenty-eight students were then assigned to a collaborative group and thirteen to an individual group. The GPA was used to ensure the groups were academically equivalent. The students were assigned to one partner for the entire semester. The partner assignments ensured there was a sufficient spread of high-high, high-average, high-low, average-average, average-low, and low-low pair grouping. This was done in order to study the possible relationship between previous academic performance and successful collaboration. Of the fourteen collaborative pairs, thirteen pairs were mutually chosen in that each student had asked to work with their partner. The last pair was assigned because the students did not express a partner preference. (Note: the students were asked their partner preference because it is more difficult for students to meet to work on assignments than it is for programmers to work together in industry. They must balance the workload and timing of other classes and part-time jobs. It was much more likely they would make the time and effort to work with those they knew.)

Students in the collaborative group completed their assignments in pairs using the CSP (a variant of the PSP, which will be discussed below). Students in the individual group completed all assignments using a modified version of the PSP. The version of the PSP used by the students was modified from that defined in [6] in order to parallel the software development approaches defined in the CSP (i.e. object oriented analysis and design and testing techniques were incorporated). Therefore, the only difference between the individual and the collaborative groups was the use of pair programming. All students received instruction in effective pair programming and were given a paper [10] on strategies for successful collaboration. These helped prepare them for their collaborative experience.

A Windows NT data collection and analysis web application was used to accurately obtain data from and provide feedback to the students, as easily as possible for the students. Disney [11] stresses the importance of such a tool for accurate process data collection.

Significantly, in the experiment, the programs produced by the pairs had about 15% fewer defects [3] based on the automated test cases run by the teaching staff. (These results are statistically significant with $p < .001$.) After an initial adjustment period in the first program (the "jelling" assignment, which took approximately 10 hours), the pairs spent about 15% more working hours in total - or 42.5% fewer elapsed hours - completing their assignments compared to the individuals. The difference in time the individuals spent on the assignments vs. the time the pairs spent on the assignments was not statistically significant ($p > .38$). In summary, the pairs produced higher quality code in essentially the same amount of time as individuals.

2. An Example Integration: The Collaborative Software Process^{SM} (CSP^{SM})

As was stated earlier, the practice of pair programming can easily be incorporated into any software development process used in the classroom or in industry. As an example, this paper will explain specific changes that could be made to the Personal Software Process (PSP) [6] in order to leverage the power of two programmers working together. The author created the Collaborative Software Process (CSP) [7] in order to document these changes.

2.1 The Collaborative Software Process (CSP) Overview

The CSP is an extension of the PSP and it relies upon the foundation of the PSP. The CSP is a defined, repeatable process for two programmers working collaboratively. Each process has a set of scripts giving specific steps to follow and a set of templates or forms to fill out to ensure completeness and to collect data for measurement-based feedback. This measurement-based feedback allows the programmers to measure their work, analyze their problem areas, and set and make goals. For example, programmers record information about all the defects that they remove from their programs. They can use summarized feedback on their defect removal to become more aware of the types of defects they make to prevent repeating these kinds of defects. Additionally, they can examine trends in their defects per thousand lines of code (KLOC).

Like the PSP, the CSP follows an evolutionary improvement approach. A student or professional learning to fully integrate the CSP into their process begins at Level 0 and progresses to Level 2. Each level incorporates new skills and techniques into their process – skills and techniques that have proven to improve the quality of the software process and to improve the estimating accuracy of the engineer. These levels are very briefly defined below.

Level 0—Collaborative Baseline. At Level 0, the engineers use their "natural" process. The purpose of this level is to provide baseline measurements from which to compare results of future process improvements. Therefore, the only addition to their "natural" process is to record time and defect data about their development work.

Level 1—Collaborative Quality Management. At level 1, specific activities to improve the quality are added to the software engineer's "natural" process. Engineers perform a CRC card roleplay and use case analysis as input to a thorough (UML) high-level class diagram. Semiformal design and code reviews are introduced – though these reviews are greatly streamlined because pair programming is essentially a continuous code review. Black box and white box test cases are written early in the development process; the completeness of the set of test cases is checked against a test coverage checklist. The white box test cases are written and added to an automated regression test suite prior to writing the actual implementation code. Once code is actually written, the test cases are run, to ensure that the new function works properly and that it has not broken anything else.

Level 2—Collaborative Project Management. At level 2, project management activities are added to the process to aid the software engineer in making and keeping good commitments. Estimates are made by examining and extrapolating from actual completion statistics from past projects. Progress is tracked using earned value analysis to ensure the project is proceeding on schedule.

The basic evolution of the CSP is summarized in Table 1.

Table 1: CSP levels.

Level	Activity
0	Baseline / Current Process Coding Standard Size Measurement Process Improvement Plan
1	Analysis (Use Cases) Design Code Review Design Reviews Testing/Test Reports Measurements
2	Size Estimating Resource Estimating Task Planning Schedule Planning

As stated above, in the University of Utah experiment the pair programmers used the CSP in their development while the individual programmers used a modified version of the PSP (to mirror the organization and content of the CSP). The pair programmers outperformed the individual programmers.

2.2 Summary of Process Changes Related to Pair Programming

As was stated, the CSP relies on the foundation of the PSP. However, the CSP reorganizes and simplifies the PSP and places greater emphasis on testing and object-oriented design techniques. This paper, however, will discuss the changes made specifically for pair programming and will not further discuss other differences between the two processes.

Essentially every script, template, and form has been adjusted to incorporate the work of two and to specifically leverage the power of two working together. Forms are used to collect process information from the software engineers. The majority of the form changes were made to allow tracking and analysis (via reports) of when programmers work alone and when they work in pairs – and what specific development tasks they are performing during this time. Process scripts enumerate steps that should be performed during the development process. Script changes were made to give specific instructions on the tasks of the driver and of the observer and to document that the roles of driver and observer should periodically be switched. For example, the following is a list of tasks included all Development scripts:

- Implement the design.
- The driver implements the design by typing code via the keyboard.
- The observer watches the driver to ensure the code properly implements the design, identifying defects whenever necessary and giving suggestions for alternative implementations. The pair brainstorm on demand.
- Periodically, switch drivers.

Because pair programming is essentially a continuous review, the most radical process changes are with the design review and code review procedures. These continuous reviews are often referred to as *pair-reviews*. The PSP documentation contains a suggested design review checklist and a suggested code review checklist for software engineers to use when they pause to review their own work. In addition to continuous pair-reviews, the CSP also advocates software engineers pause and review their work. (Other software development processes, such as XP, might feel the continuous pair-reviews are sufficient and no additional reviews are necessary. Further experimentation should focus on determining the need for additional reviews in addition to continuous pair-reviews.) However, the CSP has two design review checklists and two code review checklists. One version of each checklist is very similar to the PSP checklists and is for use when software engineers perform work by themselves. The CSP also has a more simplified version of each checklist for use when work has been done collaboratively. The second version of each checklist assumes that the pair has already removed lower-level tactical defects (i.e. syntax defects, loop structure. Instead, 'formal' pair-reviews focus on higher-level issues. Tables 2 and 3 below show the items that are contained in the CSP collaborative design and code review checklists. The CSP individual review checklists contain many more items.

Table 2: Collaborative Design Review Checklists

Purpose	**To guide you through an effective Design Review**
Completeness	Ensure that the requirements and specifications are completely and correctly covered by the design: • All specified outputs are produced • All needed inputs are furnished • All required includes are stated
Class Design	• All data members are private with public getters/setters where necessary and prudent • Data Connectedness: Can you traverse the network of collaborations between the classes to gather all the information you need to deliver the services based on a representative set of scenarios? • Abstraction: Does the name of each class convey its abstractions? Does the abstraction have a natural meaning and use in the domain? • Responsibility Alignment: Do the name, main responsibility statement data and functions in each class align?
Special Cases	Check all special cases: • Ensure proper operation with empty, full, minimum, maximum, negative, and zero values for all variables • Protect against out-of-limits, overflow, underflow conditions • Ensure "impossible" conditions are absolutely impossible • Handle all incorrect input conditions

Table 3: Collaborative Code Review Checklist

Purpose	**To guide you in conducting an effective code review**
Complete	Verify that the code is a complete and correct implementation of the design.
Standards	Ensure the code conforms to the coding standards

Inspections were introduced more than twenty years ago as a cost-effective means of detecting and removing defects from software. Results [12] from empirical studies consistently profess the effectiveness of reviews. Even still, most programmers do not find inspections enjoyable or satisfying. As a result, inspections are often not done if not mandated, and many inspections are held with unprepared inspectors.

Despite a consistent stream of positive findings over 20 years, industry adoption of inspection appears to remain quite low, although no definite data exists. For example, an informal USENET survey we conducted found that 80% of 90 respondents practiced inspection irregularly or not at all. [13]

With pair programming, this problem identification occurs on a minute-by-minute basis. These continual reviews not only outperform formal reviews in their defect removal speed, but they also eliminate the programmer's distaste for reviews.

3. Non-Process Related Changes/Results

So far, this paper has discussed explicit process changes to incorporate pair programming. However, the integration of pair programming will implicitly change aspects of the software engineering development environment. Through both these implicit and explicit changes, one could expect to yield superior results (higher quality software in about half the time). These implicit changes are discussed below.

3.1 Programmer Satisfaction

Many programmers are initially skeptical, even resistant, to programming with a partner. It takes the conditioned solitary programmer out of their "comfort zone." Shortly, however, most programmers grow to prefer pair programming. Ninety-two percent of the students in the University of Utah experiment said they were more confident in their projects when working with a partner; 96% of the students said they enjoyed the class work more when working with a partner.

A programmer comments,

"It is psychologically soothing to be sure that that no major mistakes had been made . . . I find it reassuring to know that [partner] is constantly reviewing my code while I drive. I can be sure I had done a good job if someone else I trust had been watching and approved."

As opposed to process changes such as implementing formal inspections, pair-programming is a process change that has been shown to improve quality and cycle time that programmers actually like to do. If programmers like to do it, it is far less likely that they will 'forget' to do pair programming in times of stress.

3.2 Pair-Pressure

Pair programmers put a positive form of "pair pressure" on each other. The programmers admit to working harder and smarter on programs because they do not want to let their partner down. Also, when they meet with their partner they both work very intensively because they are highly motivated to complete the task at hand during the session. "Two people working together in a pair treat their shared time as more valuable. They tend to cut phone calls short; they don't check e-mail messages or favorite Web pages; they don't waste each others time [2]." As each keeps his or her partner focused and on-task, productivity gains and quality improvements are realized.

With any software development process there is a constant struggle to get the software engineers to follow the prescribed process. As discussed above, another benefit of pair pressure is improved adherence to procedures and standards. Due to human nature, pairs put a positive form of pressure on each other to follow the prescribed process. As an extreme analogy, military research advocates that soldiers are in the presence of another during combat.

A tremendous volume of research indicates that the primary factor that motivates a soldier to do the things that no sane man wants to do in combat (that is, killing and dying) is not the force of self preservation but a powerful sense of accountability to his comrades on the battlefield. . . . Among men who are bonded together so intensely, there is a powerful process of peer pressure in which the individual cares so deeply about this comrades and what they think about him that he would rather die than let them down. [14]

We have observed this same type of bonding and behavior with pair programmers. Together, pairs more consistently adhere to process standard and, therefore, more produce with high quality results.

3.3 Problem Solving

Pairs seem better equipped to solve complex problems using techniques referred to as *pair-think* and *pair-relaying*. Pair-think refers to the pair's enhanced ability to generate and evaluate alternatives. Each member of the pair approaches the task with their unique background and problem solving abilities. As a result, together the pairs generate more alternative solutions. They then, through the process of negotiating which alternative to choose, are able to efficiently determine how to proceed with a hybrid 'optimal' solution based on their joint inputs.

With *pair relaying,* pairs consistently report that together they can evolve solutions to unruly or seemingly impossible problems. Pair relaying is a name for the effect of having two people working to resolve a problem together. Practitioners describe contributing their knowledge to the best of their abilities, in turn. They share their knowledge and energy in turn, chipping steadily away at the problem, evolving a solution to the problem. Through this, pairs report that in their problem solving, they do not spend excessive time lost in a particular problem or fix.

3.4 Pair-Learning

With pair programming, learning between pairs occurs in a dual-apprenticeship mode. The partners take turns being the teacher and the taught, from moment to moment. Knowledge is constantly being passed between partners, from tool usage tips (on even just using the mouse), to programming language rules, design and programming idioms, and overall design skill. Even unspoken skills and habits cross partners. [15]

Additionally, the continuous reviews of collaborative programming create a unique educational capability, whereby the pairs are endlessly learning from each other. "The process of analyzing and critiquing software artifacts produced by others is a potent method for learning about languages, application domains, and so forth."[13] Earlier, it was stated that the continuous reviews of collaborative programming were more effective than traditional review because of their optimum defect removal efficiency. To further this, the learning that transcends in these continual reviews prevents future defects from ever occurring – and defect prevention is more efficient than any form of defect removal. Says Capers Jones, chairman of Software Productivity Research,

It is an interesting fact that formal design and code inspections, which are currently the most effective defect removal technique, also have a major role in defect prevention. Programmers and designers who participate in reviews and inspections tend to avoid making the mistakes which were noted during the inspection sessions. [16]

Phillip M. Johnson, a professor at the University of Hawaii, refutes traditional inspections heuristic "Raise issues, don't resolve them." He speaks, instead, in favor of the educational opportunity that abounds in code inspections. "A strong argument can be made that overall software quality is affected far more profoundly by improvements to developer skills, which reduces future defect creation, than by simply removing defects from current individual documents [13]." The continuous reviews of collaborative programming, in which both partners ceaselessly work to identify and resolve problems, affords both optimum defect removal efficiency **and** the development of defect prevention skills.

4. Summary

Anecdotal evidence from industry and statistical evidence from academia support the use of pair programming for improved software quality and reduced cycle time. Yet, most documented processes are designed for individual programmers. The explicit changes to incorporate pair programming into these processes is quite straightforward, as described in the changes made to the PSP to yield the CSP. These changes involved: 1) updating process scripts to document the role of the driver and the observer; 2) adapting data collection forms and analysis reports and 3) altering design and code review procedures. Making these explicit changes to the process cause several implicit, but beneficial, changes to the development environment. Some examples of these are greater adherence to procedures, enhanced problem solving ability, and improved learning.

To augment the anecdotal and statistical support of pair programming, further research need be done. To date, only (strong) anecdotal evidence has been obtained from professional pair programmers in industry. Several studies are currently being planned, and we welcome any additional interested industries to partake in further research. Additionally, an experiment in Spring 2000 will study student programmers working in teams at North Carolina State University. Some teams will incorporate pair programming while others will work traditionally. The experiment will study the effect of pair programming on team coordination and communication.

References

[1] J. T. Nosek, "The Case for Collaborative Programming," in *Communications of the ACM*, vol. March 1998, 1998, pp. 105-108.

[2] Wiki, "Programming In Pairs," in *Portland Pattern Repository*, vol. June 29, 1999, 1999, pp. http://c2.com/cgi/wiki?ProgrammingInPairs.

[3] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair-Programming," in *IEEE Software*, vol. 17, 2000.

[4] L. L. Constantine, *Constantine on Peopleware*. Englewood Cliffs, NJ: Yourdon Press, 1995.

[5] J. O. Coplien, "A Development Process Generative Pattern Language," in *Pattern Languages of Program Design*, James O. Coplien and Douglas C. Schmidt, Ed. Reading, MA: Addison-Wesley, 1995, pp. 183-237.

[6] W. S. Humphrey, *A Discipline for Software Engineering*. Reading, Massachusetts: Addison Wesley Longman, Inc, 1995.

[7] L. A. Williams, "The Collaborative Software Process PhD Dissertation," in *Department of Computer Science*. Salt Lake City, UT: University of Utah, 2000.

[8] K. Beck, *Extreme Programming Explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.

[9] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Reading,

Massachusetts: Addison-Wesley, 1999.

[10] L. A. Williams and R. R. Kessler, "All I Ever Needed to Know About Pair Programming I Learned in Kindergarten," in *Communications of the ACM*, vol. 43, 2000.

[11] A. M. Disney, Johnson, Philip M., "Investigating Data Quality Problems in the PSP (Experience Paper)," presented at Foundations of Software Engineering, Lake Buena Vista, FL, 1998.

[12] M. E. Fagan, "Advances in software inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, pp. 182-211, 1976.

[13] P. M. Johnson, "Reengineering Inspection: The Future of Formal Technical Review," in *Communications of the ACM*, vol. 41, 1998, pp. 49-52.

[14] L. C. D. Grossman, *On Killing: The Psychological Cost of Learning to Kill in War and Society*. New York: Little, Brown and Company, 1995.

[15] A. Cockburn and L. Williams, "The Costs and Benefits of Pair Programming," presented at eXtreme Programming and Flexible Processes in Software Engineering -- XP2000, Cagliari, Sardinia, Italy, 2000.

[16] C. Jones, *Software Quality: Analysis and Guidelines for Success*. Boston, MA: International Thomson Computer Press, 1997.

[17] W. S. Humphrey, *Introduction to the Team Software Process*. Reading, Massachusetts: Addison Wesley, 2000.