The Collaborative Software Process

Laurie Williams
University of Utah
Computer Science Department
50 S. Central Campus Room 3190
Salt Lake City, UT 84112 USA
+1 801 585 3736
lwilliam@cs.utah.edu

Robert R. Kessler
University of Utah
Computer Science Department
50 S. Central Campus Room 3190
Salt Lake City, UT 84112 USA
+1 801 581 4653
kessler@cs.utah.edu

"None of us is as smart as all of us."[1]

ABSTRACT
Anecdotal and qualitative evidence from industry indicates that *two* programmers working side-by-side at *one* computer, collaborating on the same design, algorithm, code, or test, perform substantially better than the two working alone. Statistical evidence has shown that programmers perform better when following a defined, repeatable process such as the Personal Software Process (PSP). Bringing these two ideas together, the Collaborative Software Process (CSP) has been developed as the first author's doctoral dissertation. The CSP is a defined, repeatable process for two programmers working collaboratively. The CSP is an extension of the PSP, and it relies upon the foundation of the PSP. This paper discusses empirical evidence that demonstrates that software developers working collaboratively using the CSP produce products with higher quality more quickly than those working individually using the PSP.

An experiment was run in 1999 with approximately 40 senior Computer Science students at the University of Utah. All students learned both the CSP and the PSP. Two-thirds of the students worked in two-person collaborative teams using the CSP to develop their programming assignments. The other students worked independently using the PSP to develop the same assignments. The productivity, cycle time, and quality of the two groups have been compared. Empirical results point in favor of the collaborative teams using the CSP.

Keywords
Collaborative Software Process, CSP, Personal Software Process, PSP, collaborative programming, pair-programming

1 INTRODUCTION

The software industry has particular difficulty producing large software systems on schedule, of high quality and at a pre-established budget. Watts Humphrey, of the Software Engineering Institute (SEI), has approached this problem by defining the Personal Software Process (PSP)[2]. PSP requires each individual software engineer to follow a defined, disciplined software development process. PSP has met with great success in industry and is known for consistently and efficiently producing high-quality products. The value of PSP has been documented in several industrial case studies [3] and is also taught at many universities, including the University of Utah.

The PSP defines a framework of defined operations or sub-process and measurement and analysis techniques to help engineers understand their own skills and improve personal performance. Each sub-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). The measurement-based feedback of the PSP helps a programmer to measure, identify, and focus on areas in the development process providing maximum effectiveness and to adapt their *own personal* software process.

PSP has several strong philosophies. The first is that the longer a software defect remains in a product, the more costly it is to detect and remove. Therefore, thorough design and code reviews are performed for most efficient defect removal. The second philosophy is that defect

prevention is more efficient than defect removal. Careful designs are professed. Lastly, PSP rests on the notion that the best estimates, and therefore the best commitments, for schedule and defect rates can be made with a historical database of information. Data regarding how long previous products took to develop and their defect rates are kept in a database for use with the PROBE estimation sub-process. (PROBE stands for **PRO**xy-**B**ased **E**stimating). These processes and philosophies work together to produce excellent results. "SEI's data on 104 engineers shows that, on average, PSP training reduces size-estimating errors by 25.8 percent and time-estimating errors by 40 percent. Lines of code written per hour increases on average by 20.8 percent, and the portion of engineers' development time spent compiling is reduced by 81.7 percent. Testing time is reduced by 43.4 percent, total defects by 59.8 percent, and test defects by 73.2 percent [3]."

Though not as well documented, accepted, or proven, the Extreme Programming (XP) methodology, developed primarily by Smalltalk code developer and consultant Kent Beck with colleagues Ward Cunningham and Ron Jeffries also claims great success in software production. XP attributes great success to the use of "pair programming." Working in pairs, they perform continuous design and code reviews, noting that it is amazing how many obvious but unnoticed defects are noticed by another person at your side. This is, perhaps, the ultimate implementation of PSP's "defect prevention" and "efficient defect removal" philosophies.

XP also has particularly thorough testing procedures. Comprehensive test cases are written prior to actual code changes. The results of running these new tests and previous, regression test cases determine if the new code has been done correctly without harming the implementation of the base code.

Bringing the best of these two methodologies together, the Collaborative Software Process (CSP) has been developed and validated. The CSP is based on the PSP. It, too, consists of scripts, templates and forms. However, the content has been adapted to leverage the strengths and capabilities of a pair of collaborative programmers working together. Additionally, since the PSP was published in 1995, updated design and testing techniques have been incorporated into the CSP. The CSP gives a pair of programmers a framework of techniques and measurement-based feedback to improve their joint performance, and thereby the performance of their team.

This research has shown the evolutionary incorporation of CSP techniques into the software development process increasingly yields higher quality products and a more controlled process. These results are similar to those found with the PSP. The results further show that collaborative teams following the CSP outperform individuals following a PSP that has been adapted for the new design and testing techniques incorporated in the CSP.

2 CSP LEVELS

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 six levels to Level 2.1. 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 defined below:

CSP Level 0: Collaborative Baseline

At **Level 0.0**, 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.

However, one other important thing happens at this level, particularly if the engineers have never worked in pairs before – they jell as a team! We have all been conditioned to work individually – and switching to collaborative programming is certainly an adjustment. Many engineers venture into their first pair programming experience skeptical that they would actually benefit from collaborative work. They wonder about coordinating schedules, the added communication that will be required, about adjusting to the other's working habits, programming style, and ego, and about disagreeing on aspects of the implementation. (For guidelines on making an effective transition from solo to pair-programming, see [4].)

In industry, this adjustment period has historically taken hours or days, depending upon the individuals. In the university experiment described above, the students generally adjusted after the first assignment, though some reported an even shorter adjustment period. One important milestone is when the non-driver can resist the temptation to grab the mouse or keyboard from the driver! For the first assignment, the pairs finished in shorter elapsed time (because they worked in tandem) and had better quality, but they took, on average, 60% more programmer hours to complete the assignment when compared to the individuals. After the adjustment time, this 60% decreased dramatically to a statistically insignificant difference.

It doesn't take many victorious, clean compiles or declarations of "We just got through our test with no defects!" for the teams to celebrate their union – and to feel as one jelled, collaborative team.

At the CSP **Level 0.1**, several small process improvements are made. The engineers begin to follow a coding standard and to count and record their number of lines of code. After each program, they also reflect on their process – what went well, what didn't go so well – and record these

observations in the Process Improvement Proposal (PIP).

CSP Level 1: Collaborative Quality Management
Now that people learning the CSP have gotten used to working in pairs and taking some very basic measurements the attention is turned toward introducing particular activities to improve product quality in Level 1.

In **Level 1.0**, attention is focused on the first stages of the development process, analysis and design. Analysis is performed through the development of use cases [5] based on the customer requirements. The first step in developing the use cases is to identify the actors, the people or systems that are external to the system but act upon or with the system. Then, the use cases themselves can be discovered. A use case is a sequence of transactions performed by a system that yields a measurable result of values for a particular actor. A use case typically represents major functionality that is complete from beginning to end. In CSP, each use case is explored by completing a Use Cases Flow of Events template [6] shown below in Figure 1.

X Flow of Events for the <Name> Use Case
X.1 Preconditions X.2 Main Flow X.3 Sub-flows (if applicable) X.4 Alternative Flows

Figure 1: Use Case Flow of Events Template

Next, a CRC card exercise is performed to facilitate the process of identifying the system's objects and their public interfaces [7]. (CRC stands for **C**lasses, **R**esponsibilities, and **C**ollaborators) In the CRC Card exercise, index cards are used to identify classes, their responsibilities, and which other classes they must collaborate with to perform their services. A format of a typical CRC card is shown below in Figure 2. (Often the class attributes are written on the back of the card.)

The scenarios identified by the use cases are "role played" using the cards – to ensure that the classes perform the necessary services to complete each scenario. For example, a particular scenario is chosen from a use case (e.g. one particular flow through the use case flow of events). The engineers role play what the code would need to do in order for that scenario to complete successfully. Through this process, the engineers ensure that the classes have been well formulated and that they have the necessary behavior-responsibility (via their methods) and knowledge-responsibility (via their attributes) to handle the scenario. A representative number of scenarios are role-played. From the CRC card exercise the high-level/class design almost falls out – because the classes have been identified as well as the required methods and attributes.

Class Name	
Main Class Responsibility (one sentence)	
Responsibilities	Collaborators
. . .	. . .

Figure 2: CRC Card Format

At CSP **Level 1.1** – both design and code reviews are introduced. Even with the non-driver performing constant reviews, the pair still needs to step back from the computer and review their work against prescribed design and code review checklists. Realistically, even with collaborative pairs, some work will be done individually due to illness, time conflict, or by conscious choice. (For example many pair programmers have found that rote, routine coding is more effectively done alone.) Therefore, the CSP has two versions of the design review checklist and two versions of the code review checklist – one for individual work and one for collaborative work. Work done individually must be very carefully checked before being incorporated into the code base. Therefore, the checklists for individual work are more through. However, work performed by both partners does not require as thorough a formal review because the non-driver performs a constant review. The individual design review checklist is attached in Appendix A. Some items in the checklist were taken from [2] and [8]. Items in italics are not included in the collaborative design review checklist. Reviews of collaborative pairs focus overriding factors like "Does the design cover all items in the specification?" or "Did we completely implement our design?" Reviews of individual work also check for syntax and lower-level logic errors.

Level 1.1 also introduces testing techniques. Black box test cases are written early, in the design stage. The philosophy behind this is that if you diabolically think about "how can I break this code" and write test cases to see if you have or not, you will design and code in order to pass your own test cases. For each test case, the Test Case Template (See Figure 3 below) is completed. As is done in Extreme Programming, black and white box test cases are written and added to an automated regression test suite prior to writing the actual code. Once code is actually written, the test cases are run, to ensure that the new function works properly and that it hasn't broken anything else. Appendix B has a test coverage script that helps with the review of the test cases.

Test Case Template

1. Test Objective
2. Test Conditions/Description
3. Expected Result
4. Actual Result

Figure 3: Test Case Template

The last thing added to the Level 1.1 is measurements. Beginning with Level 0.0, engineers record data on the time they spend and the defects they remove. In the 1.1 level this data is turned into significantly more information in order to provide measurement-based feedback. The engineers can measure how effective their design and code reviews were via yield measurements, how long it took to find defects in the review stage, the compile stage, the test stage and much more. This information can provide critical feedback so they can effectively critique their own work and adjust their pair-process.

CSP Level 2: Collaborative Project Management

Most of the project management techniques of the PSP are unchanged in the CSP. They easily apply to collaborators as well as individuals.

In CSP **Level 2.0**, the PROBE method is used to estimate size and resource. An estimate of the number of lines of code per object estimate is made via projecting the classes that must be created and the quantity of methods each class needs (the techniques to do this were implemented in Level 1.0). Linear regression analysis is performed on the engineer's historical database of past projects – past estimates, actual size, and effort. Parameters from the linear regression analysis are used to project the estimated object lines of code to the estimated total lines of code and to estimated required resources.

In **Level 2.1**, the engineer implements task and schedule planning and tracking via the earned value method. "A particular task's earned value is based on the percentage of the total planed project effort that the task will take. As tasks are completed, the task's planned value becomes earned value for the project. The project's earned value then becomes an indicator of the percentage of completed work. When tracked week by week, the project's earned value can be compared to its planned value to determine status, to estimate rate of progress, and to project the completion date for the project. [9]"

3 DIFFERENCES BETWEEN CSP AND PSP

Obviously the largest difference between PSP and CSP is the incorporation of pair programming. 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.

Additionally, more recent Object-Oriented Analysis and Design techniques were incorporated into the CSP. Use Cases, CRC Cards and class design are introduced in Level 1.0.

Inspired by the automated testing techniques of XP, additional testing focus was incorporated into Level 1.1. Black box test cases are written during the design phase using a Test Case Template. White box and additional black box test cases are written prior to actually coding new functions, and these test cases are added to an automated regression test suite. Overall test coverage is checked against a Test Coverage Checklist (see Appendix B).

The chart below summarizes the differences between the software engineering techniques introduced in each level. Two major differences are noted. First, PSP levels 1 and 2 are swapped in the CSP. This was done to place additional focus on quality management early in the process, while accumulating more historical data that can be used when reaching CSP level 2. Additionally, cyclic development is encouraged through levels 1 and 2 in the CSP making the equivalent of PSP level 3 unnecessary.

Level	PSP	CSP
0.0	Baseline / Current Process	Baseline / Current Process
0.1	Coding Standard Size Measurement Process Improvement Plan	Coding Standard Size Measurement Process Improvement Plan
1.0	Size Estimating Test Reports	Analysis (Use Case) Design
1.1	Task Planning Schedule Planning	Code Review Design Reviews Test Reports Measurements
2.0	Code Review Design Review Measurements I	Size Estimating Resource Estimating
2.1	Design Templates Measurements II	Task Planning Schedule Planning
3.0	Cyclic Development	

Figure 4: Differences between PSP and CSP Levels

4 EMPIRICAL RESULTS

A formal experiment was run at the University of Utah to validate the effectiveness of the Collaborative Software Process. In the summer of 1999, a web programming class was taught to 20 undergraduates. The students formed ten pairs and worked collaboratively using the CSP for all assignments. The purpose of the class was to refine and

test the CSP before running a formal experiment.

The official experiment was run in the fall of 1999. The class consists of 42 juniors and seniors. (It is important to note that by the time these students participated in this class and this experiment, they have had significant programming experience in the form of internships and large class projects – such as writing compilers, portions of operating systems, and interpreters.) They learned both the PSP and the CSP and coded in C++, a language they had used for years. One third of the class worked individually while the rest worked in collaborative pairs. The individuals used the PSP; the pairs used the CSP. Both groups did the same programs so their results could be directly compared. The students completed five assignments over a period of seven weeks. The first and last assignments were pretest and posttest elements of the formal experiment in order to study the performance of an individual programmer versus the performance of the same individual as a collaborative programmer.

Product Results

An important measure is product quality. The results of the three (middle) programs done by individuals and collaborative teams are shown in the table below and have been reported in [10]. The pairs always passed more of the post-development test cases. Their results were also more consistent, while the individuals varied more about the mean. Individuals intermittently didn't hand in a program or handed it in late; pairs handed in their assignments on time. This result can be attributed to a positive form of "pair-pressure" the programmers put on each other. The programmers admit to working harder and smarter on programs because they do not want to let their partner down. Individuals do not have this form of pressure and, therefore, do not perform as consistently.

	Individuals	Collaborative Teams
Program 2	70%	86%
Program 3	78%	89%
Program 4	70%	83%
Overall	73%	86%

Figure 5: Percentage of Test Cases Passed

Another measure of process and product quality considers the number of defects found and removed during the development process as well as the post development defects. The CSP captures this data and it is shown below as a defects/KLOC measurement for all five programs written by the students. It is important to understand that through the introduction of CSP processes, such as design and code review – a downward trend in defects/KLOC is an indication that the new processes indeed improve process and product quality. (Obviously, all students found program 3 particularly difficult.) The collaborators consistently had less in-process and post-development defects/KLOC than the individuals. As noted below, the pairs consistently produced less code to accomplish the same task. Therefore, their effective defect density is even better than this graph would indicate.

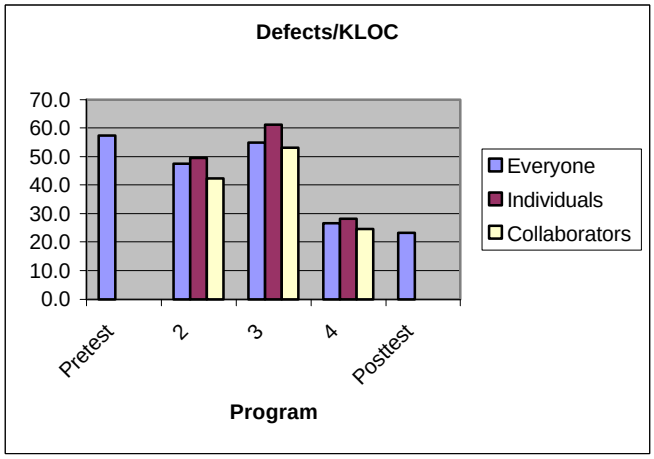

Figure 6: Defects/KLOC

Additionally, the pairs produced superior high-level project designs. The individuals were more likely to produce "blob class [11]" designs -- just to get the job done. The design from the collaborative teams exploited more of the benefits of object-oriented programming. Their classes demonstrated more encapsulation and had more classes with better class-responsibility alignment. The individuals tended to have fewer classes that had many responsibilities. The collaborative designs would, therefore, be easier to implement, enhance and maintain. A confirmation of their superior designs is that the pairs consistently write about 20% less code than the individuals to achieve the same result but with higher quality. This could not happen if not for better, well thought-out designs.

Many people's gut reaction is to reject the idea of pair-programming because they assume that there will be a 100% programmer-hour increase by putting two programmers on a job that one can do. After the initial adjustment period, discussed above, the total programmer hours spent on each assignment trended downward dramatically as shown below and reported in [10]. By program 4 in the formal experiment, the difference between the times for individuals and for the pairs was no longer statistically significant. Consider that if the individuals were required to spend the additional time to bring their code to the quality of the pairs -- the individuals would surely take even more time than the pairs.

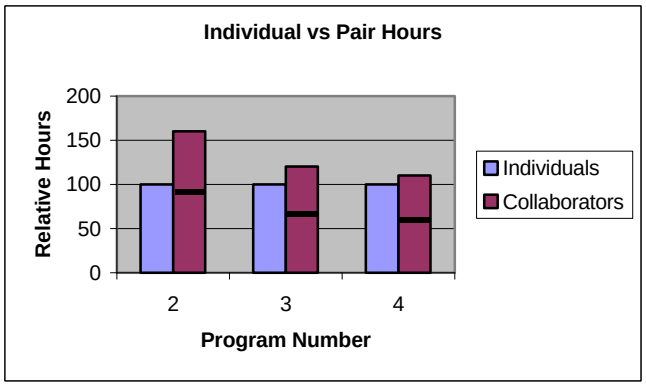

Figure 7: Individual vs Pair Hours

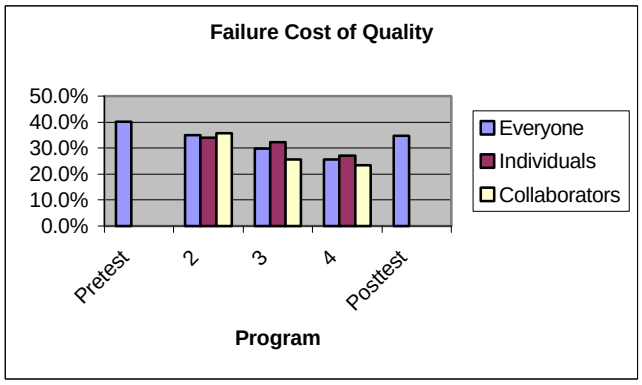

Figure 8: Failure Cost of Quality

Because the pairs worked in tandem, they were able to complete their assignments 40-50% more quickly. In today's competitive market -- where getting a quality product out as fast as possible is a competitive advantage or can even mean survival – pair-programming seems the way to go. Statistics [2] show that fixing defects after release to customers can cost hundreds more than finding and fixing them during the development process. [2] The minimal programmer-hour increase that might be experienced with CSP over PSP is offset by getting the product out faster, reducing maintenance expenses, and by improving product quality.

CSP measurements consider "failure cost" -- the percentage of time the programmers spend in the compile and test phases versus how much total time they spend on the project. We call the compile and test phases the "Chaos Zone" of the development cycle – because even a very small defect can cause a considerable and unpredictable amount of time to find and fix. Therefore, it is desirable to prevent defects and to remove any defects prior to the compile phase. A lower failure cost is an indication that the process is under control and is removing defects efficiently and effectively. Results, shown below, indicate that the introduction of CSP quality management techniques reduce the amount of time the programmers spend in this Chaos Zone. (The last program included the development of an automated regression tester. The students used this regression tester to test all of the old code in their code base – hence an increase in test time.) Note, the collaborators consistently spent less time in the Chaos Zone than the individuals.

Engineer Satisfaction

The incorporation of pair-programming into CSP improves engineers' job satisfaction and overall confidence while attaining the quality and cycle time results discussed above. Pair programmers were surveyed six times on whether they enjoyed their job more when pair programming. First, an anonymous survey of professional pair programmers was run on the Internet [12]. The summer class at the University of Utah was surveyed three times and the fall class was surveyed twice. Consistently, over 90% agreed that they enjoyed their job more when pair programming. The results are shown below.

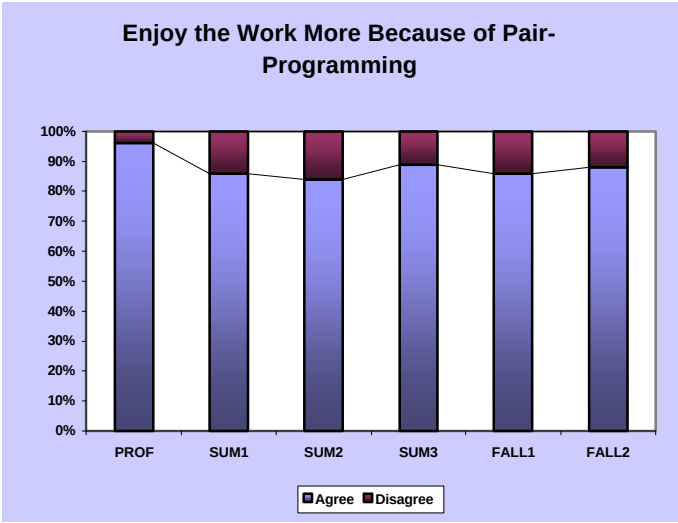

Figure 9: Pair Enjoyment

The groups were also surveyed on whether working collaboratively made them feel more confident about their work. These results are even more positive. (It is important to note that the fall class was surveyed for the first time before they were instructed on CSP and before they had ever pair-programmed.) The results are shown below.

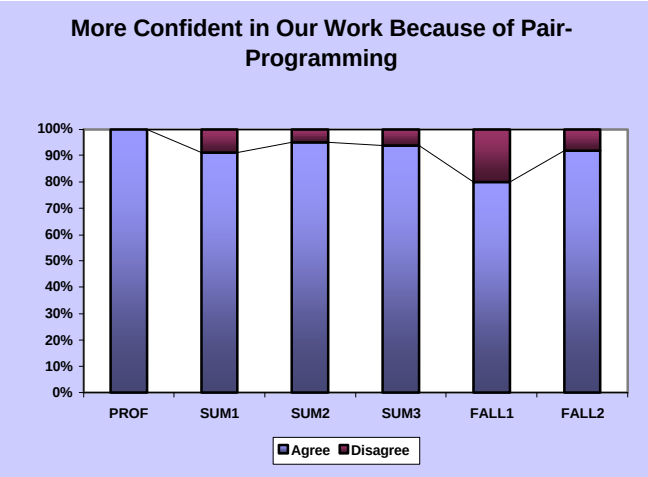

Figure 10: Pair Confidence

5 SUMMARY

Empirical evidence has shown that a pair of collaborative programmers using the CSP produce high quality products faster than individual programmers using the PSP. Additionally, the collaborative programmers find software development more enjoyable. There are many reasons why pair programming would be more satisfying than individual programming. First, humans (even programmers!) are social and enjoy the interaction. Matthias Felleisen of Rice University refers to solo programmers as "lonely macho warriors battling against a sea of bits and bytes." Not only are pair-programmers not lonely warriors, but the power of their two brains helps them think and reason in order to outsmart their "opponent." Additionally collaborative programmers always have someone to help him or her if they are confused or unknowing. They have a partner to talk to and to bounce ideas off of. The efficient and effective defect prevention and defect removal allows them to spend more time doing mentally stimulating design and less time doing frustrating debugging. The authors have pair-programmed and agree completely that pair programming is far more enjoyable than individual programming. There is a **shared** jubilation that is gained from the successful completion of a pair-programmed task. It is always more enjoyable to share a triumph with another . . . and to have someone to high five!

6 ACKNOWLEDGEMENT

A special thanks to the 62 students involved in the empirical study. Their enthusiasm and dedication has made this research very enjoyable! The authors would also like to thank Watts Humphrey for his careful review of this paper.

REFERENCES

1. Bennis, W., Biederman, Patricia Ward, *Organizing Genius: The Secrets of Creative Collaboration*. 1997: Addison-Wesley Publishing Company, Inc.
2. Humphrey, W.S., *A Discipline for Software Engineering*. SEI Series in Software Engineering, ed. P. Freeman, Musa, John. 1995: Addison Wesley Longman, Inc.
3. Ferguson, P., Humphrey, Watts S., Khajenoori, Soheil, Macke, Susan and Matvya, Annette, *Results of Applying the Personal Software Process*, in *Computer*. 1997. p. 24-31.
4. Williams, L., Kessler, R., *All I Ever Needed to Know About Pair Programming I Learned in Kindergarten*, to appear in *Communications of the ACM*.
5. Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*. 1992, Wokingham, England: Addison-Wesley.
6. Quatrani, T., *Visual Modeling with Rational Rose and UML*. Object Technology Series, ed. Booch, Jacobson, Rumbaugh. 1998, Reading, Massachusetts: Addison Wesley. pp. 29-32.
7. Bellin, D.a.S., Susan S., *The CRC Card Book*. Object Technology Series, ed. Booch, Jacobson, Rumbaugh. 1997, Reading, Massachusetts: Addison-Wesley.
8. Cockburn, A., *Object-Oriented Analysis and Design, Part 2*, in *C/C++ Users Journal*. 1998.
9. Hayes, W. and Over, J., *The Personal Software Process: An Empirical Study of the Impact of PSP on Individual Engineers*, . 1997, Software Engineering Institute: Pittsburgh, PA.
10. Williams, L., Kessler, R., Cunningham, W. and Jeffries, R., *Strengthening the Case for Pair-Programming*, in *IEEE Software*. submitted to IEEE Software.
11. Brown, W.J., Malveau, R. C., McCormick, H. W., Mowgray, T. J., *AntiPatterns*. 1998, New Yorki: Wiley Computer Publishing. 73-83.
12. Williams, L., *Pair Programming Questionnaire*, . 1999, http:\\limes.cs.utah.edu/questionnaire/questionnaire.htm.

Appendix A: Individual Design Review Checklist

Purpose	To guide you through an effective Design Review
Completeness	Ensure that the requirements and specifications are completely and correctly covered by the design: o All specified outputs are produced o All needed inputs are furnished o All required includes are stated
Class Design	o All data members are private with public getters/setters where necessary and prudent o **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? o **Abstraction:** Does the name of each class convey its abstractions? Does the abstraction have a natural meaning and use in the domain? o **Responsibility Alignment:** Do the name, main responsibility statement data and functions in each class align?
Logic	o *All program sequences are in the proper order* o *Recursion unwinds properly and terminates* o *All loops are properly initiated, incremented and terminated*
Modularity	*Ensure the proper use of functions to modularize program steps.* o *Ease of reading* o *Repetitive steps*
Special Cases	Check all special cases: o Ensure proper operation with empty, full, minimum, maximum, negative, and zero values for all variables o Protect against out-of-limits, overflow, underflow conditions o Ensure "impossible" conditions are absolutely impossible o Handle all incorrect input conditions

Note: Items in italics are not included in the Collaborative Design Review Checklist because the non-driver should remove them during their continuous design review.

Appendix B: Test Case Coverage Checklist

Purpose	To guide you in reviewing the completeness of your test cases
Black Box Testing	
Complete	Does each requirement have it's own test case?
Equivalence Class Partitioning	Have you developed an equivalence class representing the set of valid or invalid input conditions for each test case: o If the input for the test case: o can be a range of values, try one valid input value and two different invalid input values o must be a specific value, try the valid input value and two different invalid input values o must be any of a set of values, try one valid value and one invalid value o is a boolean, try both true and false
Boundary Value Analysis	Have you performed boundary analysis on the input conditions for each test case If the input for the test case: o can be a range of values from a to b, try a, b, a-1, and b+1 (if integers -- otherwise slightly less than a an slightly more than b) o must be any of a set of values, test with the min of the set, the max of the set, the min-1 and the max+1 o is a boolean, try both true and false
Scenario Testing	Do you have test cases that run through a representative set of customer scenarios?
Data	Do you have test cases that check for the wrong kind of data -- for example a negative price?
White Box Testing	
Basis Path Testing	Has each line of code been executed with at least one test case? o Draw the flowgraph of a module. o Compute the minimum number of tests necessary to exercise each line of code by calculating the cyclomatic complexity V(G) using any one of the formulas below o V(G) = the number of regions in the graph OR o V(G) = E - N + 2 (where E= number of edges and N = number of nodes) OR o V(G) = P + 1 (where P = number of predicate nodes)
Loop Testing	Where n is the maximum number of allowable passes through the loop, write test cases to: o Skip the loop entirely o Make only one pass through the loop o Make two passes through the loop o Make m passes through the loop, where m < n o Make n-1, n, and n+1 passes through the loop

File Interface	Have you checked for: o proper file attributes o opening and closing files o eof handling
Error Handling	Have you tried error handling routines? o Are error descriptions meaningful? o Do error descriptions match the error conditions? o Are there any spelling mistakes in messages?
Object Oriented Testing	If you have a class hierarchy, have you tested each of the inherited methods in the context of each inherited class?
Misc	o Have you exercised any possible underflow or overflow conditions? o Has a list of common errors been used to write test cases to detect errors that have occurred frequently in the past o Do the test cases make hand checks easy?