International Scholarly Research Network
ISRN Software Engineering
Volume 2012, Article ID 516184, 11 pages
doi:10.5402/2012/516184

Research Article

Collaborative-Adversarial Pair Programming

Rajendran Swamidurai[1] and David A. Umphress[2]

[1] *Department of Mathematics and Computer Science, Alabama State University, AL 36104, USA*
[2] *Department of Computer Science and Software Engineering, Auburn University, AL 36849, USA*

Correspondence should be addressed to David A. Umphress, david.umphress@auburn.edu

Received 10 April 2012; Accepted 14 May 2012

Academic Editors: R. Bharadwaj and U. K. Wiil

This paper presents a study called *collaborative-adversarial pair (CAP)* programming which is an alternative to pair programming (PP). Its objective is to exploit the advantages of pair programming while at the same time downplaying its disadvantages. Unlike traditional pairs, where two people work together in all the phases of software development, CAPs start by designing together; splitting into independent test construction and code implementation roles; then joining again for testing. An empirical study was conducted in fall 2008 and in spring 2009 with twenty-six computer science and software engineering senior and graduate students at Auburn University. The subjects were randomly divided into two groups (CAP/experimental group and PP/control group). The subjects used Eclipse and JUnit to perform three programming tasks with different degrees of complexity. The results of this experiment point in favor of CAP development methodology and do not support the claim that pair programming in general reduces the software development duration, overall software development cost or increases the program quality or correctness.

1. Introduction

One of the popular, emerging, and most controversial topics in the area of software engineering in the recent years is pair programming. *Pair programming* (PP) is a way of inspecting code as it is being written. Its premise—that of two people, one computer—is that two people working together on the same task will likely produce better code than one person working individually. In pair programming, one person acts as the "driver" and the other person acts as the "navigator." The driver is responsible for typing code; the navigator is responsible for reviewing the code. In a sense, the driver addresses operational issues of implementation and the observer keeps in mind the strategic direction the code must take.

Although the history of pair programming stretches back to punched cards, it emerged as a viable approach to software development in the early 1990s when it was noted as one of the 12 key practices promoted by *extreme programming* (XP) [1]. In recent years, industry and academia have turned their attention and interest toward pair programming [2, 3], and it has been widely accepted as an alternative to traditional individual programming [4].

Advocates of pair programming claim that at the cost of only slightly increased personnel hours, pair programming offers many benefits over traditional individual programming, such as faster software development, higher quality code, reduced overall software development cost, increased productivity, better knowledge transfer, and increased job satisfaction [2].

Pair programming's detractors point to three significant disadvantages. First, it requires that the two developers be at the same place at the same time. This is frequently not realistic in busy organizations where developers may be matrixed concurrently to a number of projects. Second, it requires an enlightened management that believes that letting two people work on the same task will result in better software than if they worked separately. This is a significant obstacle since software products are measured more by tangible properties, such as the number of features implemented, than by intangible properties, such as the quality of the code. Third, the empirical evidence of the benefits of pair programming is mixed: some work supports the costs and benefits of pair programming [5–9], whereas other work shows no statistical difference between pair programming and solo programming [2, 10–13].

References [2, 14, 15] show that pair programming is more effective than traditional single-person development if both members of the pair are novices to the task at hand. Novice-expert and expert-expert pairs have not been demonstrated to be effective. Reference [16] notes that many developers are forced to abandon pair programming due to lack of resources (e.g., due to small team size), something that can actually harm a project by hindering the integration of new modules to the existing project.

This paper presents a study called the *collaborative-adversarial pair (CAP)* programming which is an alternative to pair programming. Developed at Auburn University several years ago as part of a commercial cell-phone development project, its objective is to exploit the advantages of pair programming while at the same time downplaying the disadvantages. Unlike traditional pairs, where two people work together in all the phases of software development, CAPs start by designing together; splitting into independent test construction and code implementation roles; then joining again for testing.

The study evaluated the following null hypotheses.

(i) *Duration:* the overall software development duration of CAP is equal or higher than PP in average.

(ii) *Overall software development cost:* the overall software development cost of CAP is equal or higher than PP in average.

(iii) *Development cost of the coding phase:* the cost of CAP coding phase is equal or higher than PP coding phase.

(iv) *Program correctness:* the number acceptance tests failed in CAP is equal or higher than the number of acceptance tests failed in PP.

Section 2 describes the CAP software development process. Section 3 explains the differences in CAP development methodology and pair programming development methodology in detail. Section 4 presents the formal experiment procedure. Section 5 discusses the statistical tests selected for the experimental evaluation and the results. Conclusions and discussions are presented in Section 6.

2. The CAP Process

The collaborative-adversarial pair (CAP) programming process employs a synchronize-and-stabilize approach to development in which features are grouped into prioritized feature sets then built in a series of software cycles, one set per cycle (Figure 1). Each cycle starts with the entire project team reviewing the features to be built. It is here that the customer requirements are translated into product requirements by converting user stories into "*developer stories*," which are essentially manageable units of work that map to user stories. Progress is tracked by two measures: the ratio of the number of users stories built to the total number of user stories and the ratio of the developer stories completed to the total number of developer stories to be built in the cycle. The first measure expresses progress to the customer; the second measure tracks internal progress.

After the feature review, the team moves into collaborative-adversarial mode (Figure 2). The developers work together collaboratively to identify how to architect and design the features. They use this time to clarify requirements and discuss strategy. They then walk through their design with the overall project leader. After the design is approved, they move into their adversarial roles. One developer is assigned the responsibility of implementing the design, and the other developer is given the task of writing black-box test cases for the various components. The goal of the implementer is to build unbreakable code; the goal of the tester is to break the code. Note that the implementers are still responsible for writing unit-level white-box tests as part of their development efforts (Figure 3). Once both developers have completed their tasks, they run the code against the tests. Upon discovering problems, the pair resumes their adversarial positions: the tester verifies that the test cases are valid, and the implementer repairs the code and adds a corresponding regression unit test. In some cases, the test cases are not valid and are, themselves, fixed by the tester.

At the conclusion of the test phase, the team moves to a post mortem step. Here, the team (including the project manager) reviews the source code and the test cases. The purpose of the review is to (1) ensure the test cases are comprehensive and (2) identify portions of the code that are candidates for refactoring. The team does not walk through the code at a statement-by-statement level. This has been found to be so tedious that the participants quickly become numb to any problems. It is assumed that the majority of defects are caught in the black-box functional tests or in the white-box unit tests. Any gaps in test cases are captured as additional developer stories; refactoring tasks are done likewise. These developer stories receive a high enough priority that they are among the first tasks completed in the subsequent software development cycle.

A new development cycle begins again following the postmortem step.

3. Differences between CAP and PP

3.1. The Collaborative-Adversarial Roles. The most obvious difference between CAP and pair programming is the way in which the pairs enact roles in different phases of the software development. In pair programming, the pairs collaborate in all phases of the software development including analysis, design, implementation, and testing, whereas, in CAP, the pairs start by designing together; splitting into independent test construction and code implementation roles; then joining again for testing.

3.2. Unit Implementation/Coding. Like pair programming, the unit implementation procedure in CAP follows the test-driven development (TDD) approach with a small, but important, modification. Pair programming's TDD approach follows *test, code, and refactor* cycle in unit implementation. Since the unit in CAP is implemented by a single developer, the developer *self-inspects* the code after

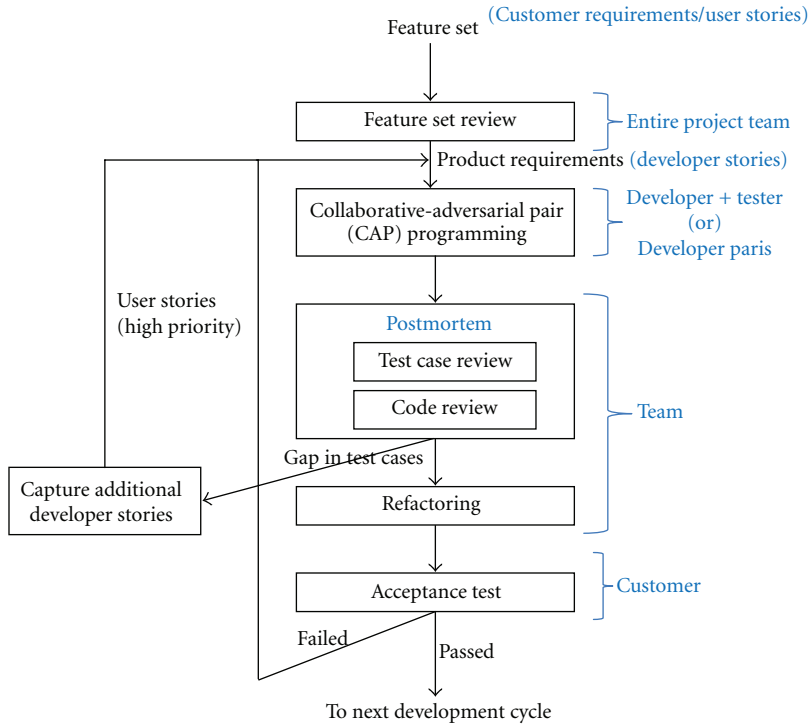

Figure 1: CAP development activity.

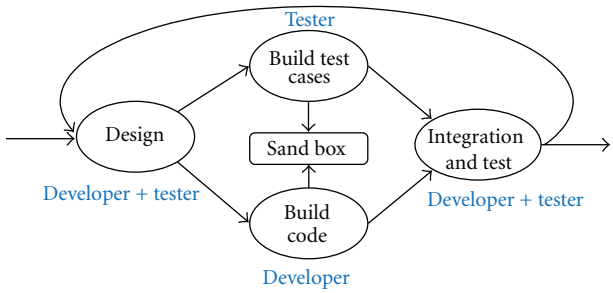

Figure 2: Collaborative-adversarial pairs (CAPs).

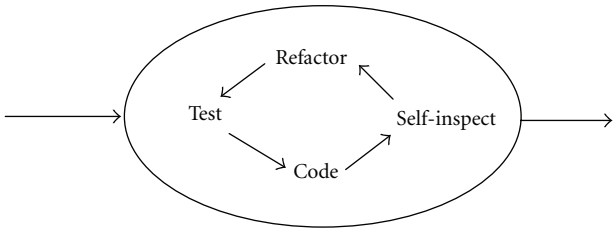

Figure 3: "Build code" activity in CAP.

Table 1: Unit implementation in PP and CAP.

Unit implementation in PP	Unit implementation in CAP
(1) Write a test	(1) Write a test
(2) Run the test to ensure it fails	(2) Run the test to ensure it fails
(3) Implement the method/write code (refactor if necessary)	(3) Implement the method/write code (refactor if necessary)
(4) Run the test to ensure it passes	(4) Self-inspect the code to ensure it does the required job
(5) Refactor	(5) Run the test to ensure it passes
(6) Repeat	(6) Refactor
	(7) Repeat

writing it in order to assure correct functionality. Table 1 highlights the key differences in CAP and pair programming unit implementations.

3.3. Black-Box Test Cases. Another major change in CAP is the incorporation of black-box testing. In *functional testing (or behavioral testing)*, every program is viewed as a function that maps values from an input domain to an output range. Functional testing is also called *black-box* testing, because the content or implementation of the function is not known. Black-box testing is based on the external specifications (i.e., inputs and outputs) of the function and is usually data driven.

With black-box testing, test cases are developed from external descriptions of the software, such as specifications, requirements, and design. The functional test cases have the following two distinct advantages.

(1) They are independent from software implementation. Implementation changes do not affect the test cases and vice versa.

(2) They can be developed in parallel with the implementation, which, in turn, reduces the overall project development interval.

The functional test cases may suffer from the following two drawbacks.

(1) There may be a redundancy in the developed test cases.

(2) There can be a probability that portions of the software may be untested.

3.4. Code and Test Cases Reviews. In pair programming, a new development cycle begins again following the test phase, but in CAP at the conclusion of the test phase, the team moves to a postmortem step. Here, the team (including the project manager) reviews the source code and the test cases. The purpose of the review is to (1) ensure the test cases are comprehensive and (2) identify portions of the code that are candidates for refactoring. In addition to this, the code review also helps senior developers pass knowledge through a development team and helps more people understand parts of the larger software system [17]. The team does not walk through the code at a statement-by-statement level. This has been found to be so tedious that the participants quickly become numb to any problems. It is assumed that the majority of defects are caught in the black-box functional tests or in the white-box unit tests. Any gaps in test cases are captured as additional developer stories.

3.5. Refactoring. Refactoring is the process of changing software's internal structure, in order to improve design and readability and reduce bugs, without changing its observable behavior. Fowler [17] suggests that refactoring has to be done in three situations: when a new function is added to the software, when a bug is fixed, and as a result of a code review. The first two cases are being covered by the refactoring session of the unit implementation. Since CAP incorporates the code review session after integration and test, an additional refactoring phase is necessary.

Refactoring also helps developers to review someone else's code and helps the code review process to have more concrete results [17].

4. The Experiment

A formal experiment was held at Auburn University to validate the effectiveness of the collaborative-adversarial pair (CAP) programming during fall 2008 and spring 2009. The study was not part of any course conducted by the department. The experiment consists of three major programming exercises and took place from September to November 2008 and January to April 2009 in the Software Process lab. Both the experimental group (CAP group) and the control group (PP group) used Java and the Eclipse development environment.

4.1. Subjects. Twenty-six volunteer students from the Software Process class, a combined class of undergraduate seniors and graduate students, participated in the study. All participants had already taken software modeling and design (using UML) and computer programming courses such as C, C++, and Java. Out of twenty-six students, 14 students had 1 to 5 years of industrial programming experience, 11 students had no or less than one-year programming experience, and one student had more than 5 years programming experience. Seven students had prior pair programming experience.

4.2. Experimental Tasks. The subjects were asked to solve in Java three programming problems described as follows.

Problem 1. Write a program which reads a text file and displays the name of the file, the total number of occurrences of a user-input string, the total number of nonblank lines in the file, and count the number of lines of code according to the Line of Code (LOC) Counting Standard used in PSP, Personal Software Process [18]. You may assume that the source code adheres to a published LOC Coding Standard that will be given to you. This assignment should not determine if the coding standard has been followed. The program should be capable of sequentially processing multiple files by repeatedly prompting the user for file names until the user enters a file name of "stop". The program should issue the message, "I/O error," if the file is not found or if any other I/O error occurs.

Problem 2. Write a program to list information (name, number of methods, type, and LOC) of each proxy in a source file. The program should also produce an LOC count of the entire source file. Your program should accept as input the name of a file that contains source code. You are to read the file and count the number of lines of code according to our LOC Counting Standard. You may assume that the source code adheres to the published LOC Coding Standard given to you. This assignment should not determine if the coding standard has been followed. The exact format of the application-user interaction is up to you.

(i) A "proxy" is defined as a recognizable software component. Classes are typical proxies in object-oriented systems; subprograms are typical proxies in traditional functionally decomposed systems.

(ii) If you are using a functionally decomposed (meaning, non-OO) approach, the number of methods for each proxy will be "1". If you are using an OO approach, the number of methods will be a count of the methods associated with an object.

Problem 3. Write a program to calculate the planned number of lines of code given the estimated lines of code (using PSP's PROBE Estimation Script [18]). Your program should accept as input the name of a file. Each line of the file contains four pieces of information separated by a space: the name of a project and its estimated LOC, its planned LOC, and its actual LOC. Read this file and echo the data to the output device. Accept as input from the keyboard a number which

represents the estimated size of a new project. Output the calculations of each decision and the corresponding planned size, as well as the PROBE decision designation (A, B, or C) used to calculate planned size. For each decision, indicate why it is/is not valid. The exact format of the application-user interaction is up to you.

(i) Your software should gracefully handle error conditions, such as nonexistent files and invalid input values.

(ii) The planned size should be up to the nearest multiple of 10.

4.3. Hypotheses. The following experimental hypotheses were tested.

$H0_1$ (Duration). The overall software development duration of CAP is equal or higher than PP in average.

Ha_1 (Duration). The overall software development duration of CAP is less than PP in average.

$H0_2$ (Time/$Cost_{Overall}$). The overall software development cost of CAP is equal or higher than PP in average.

Ha_2 (Time/$Cost_{Overall}$). The overall software development cost of CAP is less than PP in average.

$H0_3$ (Time/$Cost_{Coding}$). The cost of CAP coding phase is equal or higher than the cost of PP coding phase in average.

Ha_3 (Time/$Cost_{Coding}$). The cost of CAP coding phase is less than cost of PP coding phase in average.

$H0_4$ (Correctness). The number of acceptance tests failed in CAP is equal or higher than the number of acceptance tests failed in PP in average.

Ha_4 (Correctness). The number of acceptance tests failed in CAP is less than the number of acceptance tests failed in PP in average.

4.4. Duration. Duration (also known as delivery time) was defined as the elapsed time taken to complete the programming task and is calculated as follows:

$$\begin{aligned} \text{Duration}_{\text{PP}} &= \text{Time}_{\text{Design}} + \text{Time}_{\text{Coding}} + \text{Time}_{\text{Test}}, \\ \text{Duration}_{\text{CAP}} &= \text{Time}_{\text{Design}} \\ &\quad + \text{Max}\left(\text{Time}_{\text{Coding}}, \text{Time}_{\text{Test Case Development}}\right) \\ &\quad + \text{Time}_{\text{Test}}. \end{aligned} \tag{1}$$

4.5. Cost. To study the cost of overall software development, we compared the total development time, measured in minutes, of all the phases. Pair programming consisted of design, coding, and test phases; whereas CAP consisted of test case development phase in addition to the pair programming phases. The pair programming and CAP total software development costs were calculated as per the following formulas:

$$\begin{aligned} \text{Cost}_{\text{Total}}^{\text{PP}} &= 2 \cdot \left(\text{Time}_{\text{Design}} + \text{Time}_{\text{Coding}} + \text{Time}_{\text{Test}}\right), \\ \text{Cost}_{\text{Total}}^{\text{CAP}} &= 2 \cdot \left(\text{Time}_{\text{Design}} + \text{Time}_{\text{Test}}\right) + \text{Time}_{\text{Coding}} \\ &\quad + \text{Time}_{\text{Test Case Development}}. \end{aligned} \tag{2}$$

To study the cost of coding phase, we compared the coding phase of pair programming with coding phase of CAP measured in minutes. The pair programming and CAP total coding phase costs were calculated as per the following formulas

$$\begin{aligned} \text{Cost}_{\text{Code}}^{\text{PP}} &= 2 \cdot \left(\text{Time}_{\text{Coding}}\right), \\ \text{Cost}_{\text{Code}}^{\text{CAP}} &= \text{Time}_{\text{Coding}}. \end{aligned} \tag{3}$$

4.6. Program Correctness. To study the program correctness, the number of postdevelopment test cases passed by programs developed by pair programming group and CAP group was compared.

4.7. Subjects Assignment to the Experimental Groups. Before the actual control experiments begin, all the subjects were asked to (1) solve two programming problems individually in order to measure their programming skills and (2) complete a survey questionnaire which collected demographic information such as age, class level (senior/graduate), programming languages known, experience level, and pair programming experience.

From the pretest and survey, the following six metrics were collected:

(A). Java Programming Skills/Knowledge: 1-below average, 2-average, 3-good, 4-very good, 5-excellenet,

(B). Programming Analysis Skills (pretests results average): Analysis Score = (Test-1+Test-2)/200,

(C). Programming Experience: 1-less than one year, 3- 1 to 5 years, 5-more,than 5 years

(D). Knowledge of black-box testing: 1-Yes, 0- No,

(E). Knowledge of JUnit: 1-Yes, 0- No,

(F). Knowledge TDD (test-driven development): 1-Yes, 0-No.

Each subject was assigned a score based on these six parameters as follows:

$$\begin{aligned} \text{Score} = {} & \mathbf{A} * \mathbf{6} + \mathbf{B} * \mathbf{5} + \mathbf{C} * \mathbf{4} + \mathbf{D} * \mathbf{3} \\ & + \mathbf{E} * \mathbf{2} + \mathbf{F} * \mathbf{1}. \end{aligned} \tag{4}$$

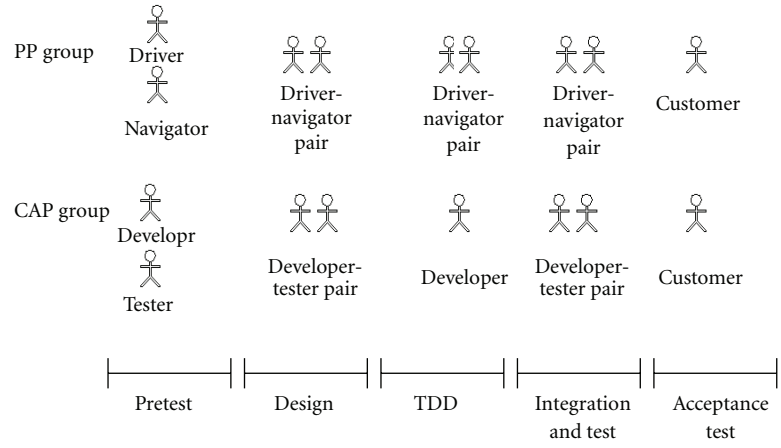

Figure 4: Experimental setup.

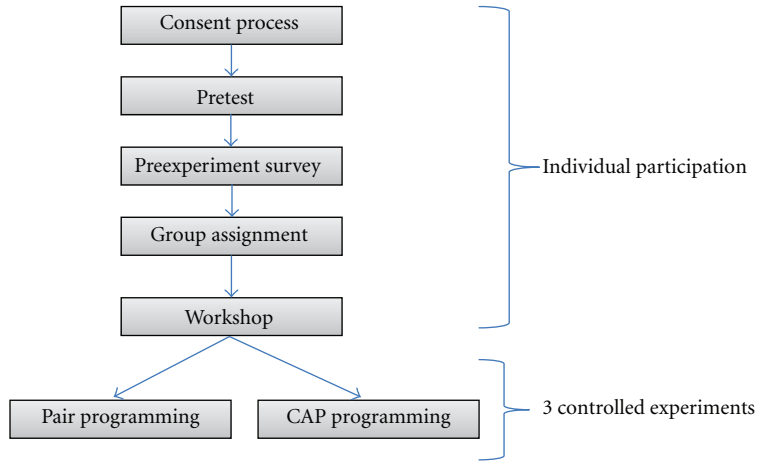

Figure 5: Experimental procedure.

Based on the score, the subjects were sorted and divided into groups of four. The subjects were randomly selected from each group and assigned to the two experimental groups: pair programming group and collaborative-adversarial pair (CAP) programming group.

4.8. Experiment Procedure

(1) *Consent process:* at the beginning of the course both in fall 2008 and in spring 2009, the IRB (Auburn University Institutional Review Board) approved informed consent for the project was handed out and students were given the chance to volunteer to participate. The researcher provided information to students about the project, handed out consent forms, answered any questions students raised by the students, and requested that the forms be returned the following class; so students had at least one intervening day to review all aspects of consent. The researcher returned the following class and answered the questions, if any, and collected the consent forms.

(2) *Pretest:* in the pretest, all the subjects were asked to solve two programming problems individually in order to measure their programming skills.

(3) *Preexperiment survey:* each subject was asked to complete a survey questionnaire which collected demographic information such as age, class level (senior/graduate), programming languages known, experience level, and pair programming experience.

(4) *Assigning the subjects to experimental groups:* based on the pretest's result and the survey, the subjects were divided into groups of four. The subjects were randomly selected from each group and assigned to the two experimental groups: pair programming group and collaborative-adversarial pair programming group.

(5) *Workshop:* before the actual control experiments started, there was a workshop for all the subjects. A lecture was arranged to explain the concepts of collaborative-adversarial pair programming,

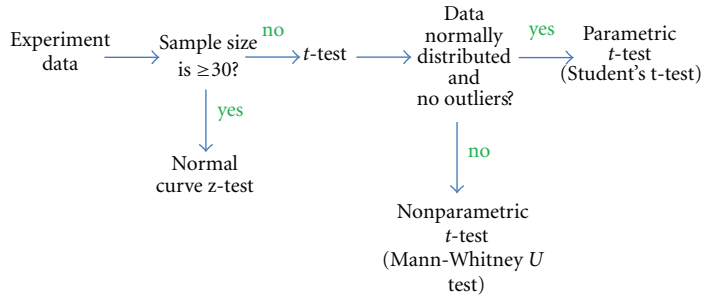

Figure 6: Overview of the evaluation process.

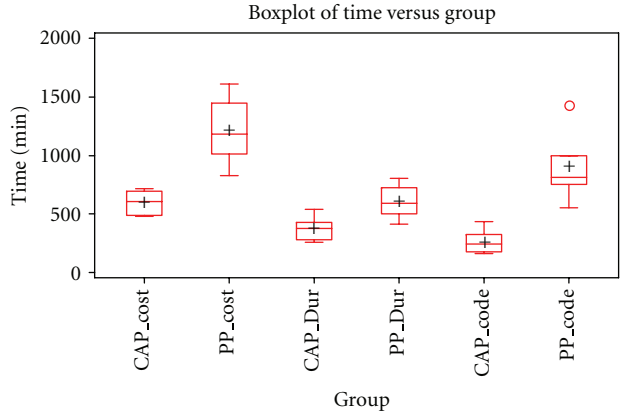

Figure 7: Box plot.

pair programming, and unit testing, and acceptance testing.

(6) *Control experiments:* three programming exercises were given to each experimental group. The subjects in both the pair programming group and the CAP group randomly paired up with a partner in their own group to do the exercises. The design of the experiments is shown in Figure 4.

Figure 5 summarizes the experimental procedure.

5. Results

5.1. Statistical Test Selection. Statistical tests are of two types: parametric and nonparametric. Each parametric test depends on several assumptions, such that the data must follow the normal distribution, the sample size should be within a specified range, and there should not be any outliers in the data. When its assumptions are met, a parametric test is more powerful than its corresponding nonparametric test. Nonparametric methods do not depend on the normality assumption, work quite well for small samples, and are robust to outliers. Student's t-test is suitable for smaller sample sizes (e.g., <30). The "normal curve z-test" is more suitable for larger samples (e.g., ≥30).

Therefore, it is clear that before we could finalize which statistical tests were most suitable to validate the CAP, we needed to analyze the data whether it satisfies the normality and no outlier properties or not.

We used a box plot to identify outliers, that is, data points which are numerically distant from the rest of the data. In a box plot, the outliers are indicated using circles. SAS's GLM procedure with BF (Brown and Forsythe's variation of Levene's test) was used to test the normality. We used Student's t-test to compare the CAP groups' means with the PP groups' means, if the data is normally distributed and contains no outliers; otherwise, we used the nonparametric t-test (Wilcoxon Mann-Whitney U test).

The overview of the evaluation process is shown in Figure 6.

5.2. Test for Normality. Table 2 shows the results of the normality tests. In Table 2, the P value of all tests is insignificant at 5% significant level ($P > 0.05$), which indicates that statistically there is no significant evidence to reject the normality; that is, the overall software development cost, duration, and coding phase cost data all follow normal distribution.

5.3. Test for Outliers. The box plots for the total software development cost, duration, and the cost of the coding phase are given in Figure 7. There are no circles in Figure 7 except PP's coding phase cost, which indicates that there are no outliers except PP's coding cost data.

5.4. Duration (Hypothesis 1). The pair programming groups took 605 minutes on average to solve all the three programming problems; whereas the CAP groups took only 379 minutes (37% less than PP groups) in average to solve all the three programming problems. The Student's t-test results for hypothesis 1 are shown in Table 3. Since $P < 0.05$ ($P = 0.0123$, two-sided t-value), we have sufficient statistical evidence to reject $H0_1$ in favor of Ha_1 and conclude that the overall software development duration of CAP is less than pair programming in average.

5.5. Cost (Hypothesis 2). The pair programming groups took 1209 minutes in average to solve all the three programming problems; whereas the CAP groups took only 599 minutes (50% less than PP groups) in average to solve all the three programming problems. The Student's t-test results

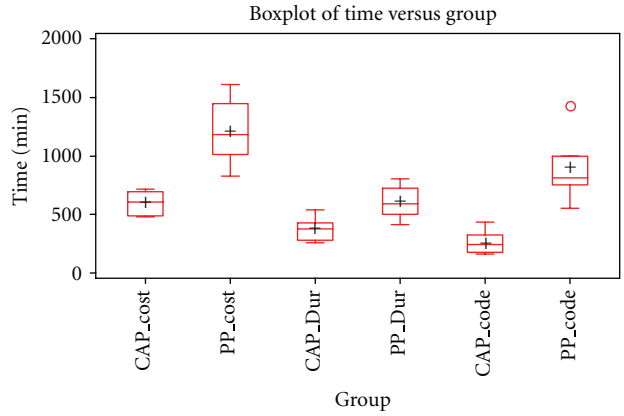

Figure 8: Average values.

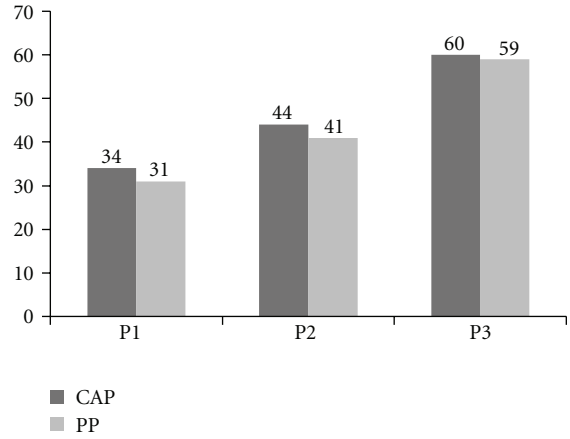

Figure 9: The number of test cases passed.

Table 2: Tests for normality.

(a) Cost

Test		*P* value
Shapiro-Wilk	Pr < W	0.9867
Kolmogorov-Smirnov	Pr > D	>0.1500
Cramer-von Mises	Pr > W-Sq	>0.2500
Anderson-Darling	Pr > A-Sq	>0.2500

(b) Duration

Test		*P* value
Shapiro-Wilk	Pr < W	0.6855
Kolmogorov-Smirnov	Pr > D	>0.1500
Cramer-von Mises	Pr > W-Sq	>0.2500
Anderson-Darling	Pr > A-Sq	>0.2500

(c) Coding cost

Test		*P* value
Shapiro-Wilk	Pr < W	0.1361
Kolmogorov-Smirnov	Pr > D	>0.1500
Cramer-von Mises	Pr > W-Sq	0.1418
Anderson-Darling	Pr > A-Sq	0.1169

Table 3: t-test results for CAP versus PP duration.

(a) The t-test procedure: statistics

Variable group	Lower CL		Upper CL		Lower CL		Upper CL	
	N	Mean	Mean	Mean	Std Dev	Std Dev	Std Dev	Std Err
Time CAP	6	272.64	379.17	485.7	63.364	101.51	248.97	41.442
Time PP	6	447.21	604.5	761.79	93.558	149.88	367.6	61.189
Time Diff (1-2)		−390	−225.3	−60.67	89.438	128	224.64	73.902

(b) t-Tests

Variable	Method	Variances	DF	t value	Pr > $\lvert t \rvert$
Time	Pooled	Equal	10	−3.05	0.0123
Time	Satterthwaite	Unequal	8.79	−3.05	0.0142

(c) Equality of variances

Variable	Method	Num DF	Den DF	F value	Pr > F
Time	Folded F	5	5	2.18	0.4125

Table 4: t-test results for CAP versus PP cost.

(a) The t-test procedure: statistics

Variable group	Lower CL		Upper CL		Lower CL		Upper CL	
	N	Mean	Mean	Mean	Std Dev	Std Dev	Std Dev	Std Err
Time CAP	6	486.32	599	711.68	67.024	107.37	263.35	43.835
Time PP	6	894.42	1209	1523.6	187.12	299.76	735.21	122.38
Time Diff (1-2)		−899.6	−610	−320.4	157.32	225.15	395.13	129.99

(b) t-tests

Variable	Method	Variances	DF	t value	Pr > $\lvert t \rvert$
Time	Pooled	Equal	10	−4.69	0.0009
Time	Satterthwaite	Unequal	6.26	−4.69	0.0030

(c) Equality of variances

Variable	Method	Num DF	Den DF	F value	Pr > F
Time	Folded F	5	5	7.79	0.0417

Table 5: Wilcoxon Mann-Whitney U test results for CAP versus PP coding cost.

Wilcoxon two-sample test	
Statistic (S)	21.0000
Normal approximation	
Z	−2.8022
One-sided Pr < Z	0.0025
Two-sided Pr > $\lvert Z \rvert$	0.0051
t approximation	
One-sided Pr < Z	0.0086
Two-sided Pr > $\lvert Z \rvert$	0.0172
Exact test	
One-sided Pr ≤ S	0.0011
Two-sided Pr ≥ $\lvert S - \text{Mean} \rvert$	0.0022

for hypothesis 2 are shown in Table 4. Since $P < 0.05$ ($P = 0.0123$, two-sided t-value), we have sufficient statistical evidence to reject $H0_2$ in favor of Ha_2 and conclude that the overall software development cost of CAP is less than pair programming in average.

5.6. Coding Cost (Hypothesis 3). The pair programming groups took 892 minutes in average to code all the three programming problems; whereas the CAP groups took only 268 minutes (50% less than PP groups) in average to code all the three programming problems. The Wilcoxon Mann-Whitney U test results for hypothesis 3 are shown in Table 5. Since $P < 0.05$ ($P = 0.0172$, two-sided t-value), we have sufficient statistical evidence to reject $H0_3$ in favor of Ha_3 and conclude that the coding phase cost of CAP is less than pair programming in average.

Table 6: Experiment's data.

Group	Design	Code	Test cases	Test	Total	Duration	Cost	Coding cost
CAP1	19	177	129	68	393	221	480	177
CAP2	54	167	145	62	428	225	544	167
CAP3	36	244	178	101	559	313	696	244
CAP4	52	435	67	55	609	489	716	435
CAP5	30	255	67	144	496	342	670	255
CAP6	40	330	66	6	442	353	488	330
Average	39	268	109	73	488	324	599	268
PP1	37	377	0	0	414	414	828	754
PP2	49	279	0	177	505	505	1010	558
PP3	32	427	0	53	512	512	1024	854
PP4	51	710	0	41	802	802	1604	1420
PP5	105	500	0	65	670	670	1340	1000
PP6	62	384	0	278	724	724	1448	768
Average	56	446	0	102	605	605	1209	892

Table 7: The number of test cases passed.

Group	Problem 1	Problem 2	Problem 3	Total
PP1	5/6	6/8	10/10	21/24
PP2	4/6	8/8	9/10	21/24
PP3	4/6	3/8	10/10	17/24
PP4	6/6	8/8	10/10	24/24
PP5	6/6	8/8	10/10	24/24
PP6	6/6	8/8	10/10	24/24
Total	31/36	41/48	59/60	131/144
CAP1	5/6	8/8	10/10	23/24
CAP2	5/6	8/8	10/10	23/24
CAP3	6/6	4/8	10/10	20/24
CAP4	6/6	8/8	10/10	24/24
CAP5	6/6	8/8	10/10	24/24
CAP6	6/6	8/8	10/10	24/24
Total	34/36	44/48	60/60	138/144

Table 6 summarizes the experiment's data, and Figure 8 shows the average values taken by pair programming groups and CAP groups for the duration, cost, and coding cost, respectively.

5.7. Program Correctness (Hypothesis 4). The number of postdevelopment test cases passed by the pair programming group programs, and the CAP group programs are shown in Table 7 and Figure 9. The acceptance tests were conducted by a disinterested party. Specifically, a graduate teaching assistant for the introductory Java course was recruited to do this. The tester was not involved in any other way with the experiment. The total numbers of test cases passed by the pair programming groups were 31, 41, and 59 for Problem 1, Problem 2, and Problem 3, respectively, whereas, the total numbers of test cases passed by the CAP groups was 34, 44, and 60 for Problem 1, Problem 2, and Problem 3, respectively.

Table 7 indicates that the number of acceptance tests failed in CAP is less than the number of acceptance tests failed in pair programming. Hence, we accept the alternative hypothesis that the number of acceptance tests failed in CAP is less than the number of acceptance tests failed in pair programming in average.

A summary of the four control experiments and their results are given in Table 8.

6. Conclusions

In this paper, we experimented with a new agile software development methodology called collaborative-adversarial pair programming. We see CAP as an alternative to traditional pair programming in situations where pair programming is not beneficial or is not possible to practice. The CAP was evaluated against traditional pair programming in terms of productivity and program correctness. The empirical evidence shows that traditional pair programming is an expensive technology and may not live up to the quality claims.

The empirical evidence shows that better or equal quality programs can be produced with a much cheaper cost (50% less overall software development time than traditional pair programming) using the CAP programming technique. The empirical evidence also shows that better or equal quality code can be produced with a much cheaper cost (70% less than traditional pair programming) using CAP programming technique.

It is expected that CAP will retain the advantages of pair programming while at the same time downplaying the disadvantages. In CAP, units are implemented by single developers (whereas two developers are developing a unit in pair programming) and functional test cases can be developed in parallel with unit implementation, which, in turn,

Table 8: Summary of control experiments and their results.

Null hypothesis	Data properties	Statistical test	Result	Reject?
$H0_1$ (Duration)	Normal Equal variance No outliers	Student's t-test	$P < 0.0001$	Yes
$H0_2$ ($Cost_{Overall}$)	Normal Equal variance No outliers	Student's t-test	$P < 0.0001$	Yes
$H0_3$ ($Cost_{Coding}$)	Normal Equal variance Outliers	Mann-Whitney U test	$P < 0.0001$	Yes
$H0_4$ (Correctness)	Not applicable	None	Number of acceptance test cases failed in CAP is less than PP	Yes

reduces the overall project development interval. The CAP testing procedure judiciously combines the functional (black box) and structural (white box) testing, which provides the software with the confidence of functional testing and the measurement of structural testing. CAP allows us to confidently test and add purchased or contracted software modules to the existing software. And finally the functional test cases in the CAP allow us to change the implementation (if needed) without changing the test cases and vice versa.

Acknowledgments

The authors gratefully acknowledge Dr. William "Amos" Confer and Bradley Dennis for their original work in helping develop CAP, as well as Dr. Richard Chapman for employing CAP in his senior design course.

References

[1] K. Beck, Extreme Programming Explained: An Embrace Change, Addison-Wesley, 2000.

[2] E. Arisholm, H. Gallis, T. Dybå, and D. I. K. Sjøberg, "Evaluating pair programming with respect to system complexity and programmer expertise," *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 65–86, 2007.

[3] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. A. Visaggio, "Evaluating performances of pair designing in industry," *Journal of Systems and Software*, vol. 80, no. 8, pp. 1317–1327, 2007.

[4] M. M. Müller, "Two controlled experiments concerning the comparison of pair programming to peer review," *Journal of Systems and Software*, vol. 78, no. 2, pp. 166–179, 2005.

[5] J. D. Wilson, N. Hoskin, and J. T. Nosek, "The benefits of collaboration for student programmers," in *Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education*, pp. 160–164, February 1993.

[6] J. T. Nosek, "The case for collaborative programming," *Communications of the ACM*, vol. 41, no. 3, pp. 105–108, 1998.

[7] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the case for pair programming," *IEEE Software*, vol. 17, no. 4, pp. 19–25, 2000.

[8] C. McDowell, L. Werner, H. Bullock, and J. Fernald, "The effects of pair-programming on performance in an introductory programming course," in *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, pp. 38–42, Cincinnati, Ky, USA, March 2002.

[9] S. Xu and V. Rajlich, "Empirical validation of test-driven pair programming in game development," in *Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science (ICIS '06). In conjunction with 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (COMSAR '06)*, pp. 500–505, July 2006.

[10] J. Nawrocki and A. Wojciechowski, "Experimental Evaluation of pair programming," in *Proceedings of the European Software Control and Metrics Conference (ESCOM '01)*, pp. 269–276, ESCOM Press.

[11] J. Vanhanen and C. Lassenius, "Effects of pair programming at the development team level: an experiment," in *Proceedings of the International Symposium on Empirical Software Engineering (ISESE '05)*, pp. 336–345, November 2005.

[12] M. Rostaher and M. Hericko, "Tracking test first programming—an experiment, XP/Agile Universe," *LNCS*, vol. 2418, pp. 174–184, 2002.

[13] H. Hulkko and P. Abrahamsson, "A multiple case study on the impact of pair programming on product quality," in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pp. 495–504, St. Louis, Mo, USA, May 2005.

[14] D. Wells and T. Buckley, ". The VCAPS project: an example of transitioning to XP.," in *Extreme Programming Examined*, chapter 23, pp. 399–421, Addison-Wesley.

[15] K. M. Lui and K. C. C. Chan, "Pair programming productivity: Novice-novice vs. expert-expert," *International Journal of Human Computer Studies*, vol. 64, no. 9, pp. 915–925, 2006.

[16] K. Boutin, "Introducing extreme programming in a research and development laboratory," in *Extreme Programming Examined*, chapter 25, pp. 433–448, Addison-Wesley.

[17] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[18] W. S. Humphrey, *PSP(sm): A Self-Improvement Process for Software Engineers*, Addison-Wesley, 2005.