Received February 15, 2017, accepted April 5, 2017, date of publication May 4, 2017, date of current version June 28, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2701339

An Empirical Study on the Impact of an IDE Tool Support in the Pair and Solo Programming

OMAR S. GÓMEZ[1], ANTONIO A. AGUILETA[2], RAÚL A. AGUILAR[2], JUAN P. UCÁN[2], RAÚL H. ROSERO[1], AND KAREN CORTES-VERDIN[3]
[1]Escuela Superior Politécnica de Chimborazo, Riobamba 060155, Ecuador
[2]Universidad Autónoma de Yucatán, Mérida 97000, Mexico
[3]Universidad Veracruzana, Xalapa 91190, Mexico

Corresponding author: Omar S. Gómez (ogomez@espoch.edu.ec)

ABSTRACT The adoption of Agile software development approaches has been widespread. One well-known Agile approach is extreme programming, which encompasses twelve practices of which pair programming is one of them. Although various aspects of pair programming have been studied, we have not found, under a traditional setting of pair programming, studies that examine the impact of using a tool support, such as an integrated development environment (IDE) or a simple text editor. In an attempt to obtain a better understanding of the impact of using an IDE in this field, we present the results of a controlled experiment that expose the influence on quality, measured as the number of defects injected per hour, and cost, measured as the time necessary to complete a programming assignment, of pair and solo programming with and without the use of an IDE. For quality, our findings suggest that the use of an IDE results in significantly higher defect injection rates (for both pairs and solos) when the programming assignment is not very complicated. Nevertheless, defect injection rates seem to decrease when pairs work on more complicated programming assignments irrespective of the tool support that they use. For cost, the programming assignment significantly affects the time needed to complete the assignment. In relation to the programming type, pairs and solos performed in a similar way with regards to quality and cost.

INDEX TERMS Pair programming, software quality and cost, integrated development environment, controlled experiment, software engineering.

I. INTRODUCTION

The adoption of agile approaches in the development and maintenance of software products has been becoming relevant in the software industry. The successful adoption of agile approaches has been reported in several countries [1]–[7].

One of the first and well-known agile approaches is eXtreme Programming, or XP [8], [9], which focuses on twelve practices for software development: planning games, short releases, system metaphors, simple design, testing, refactoring, pair programming, collective ownership, continuous integration, 40-hour-week, on-site customers, and coding standards.

Pair programming is a common practice used either, independently or as part of XP. In this practice, two programmers work together on the same task using a computer. One of the programmers (the driver) writes the code, and the other (the observer), actively reviews the work performed by the driver. Essentially, the observer reviews the work for possible defects, writes down notes, or defines strategies for solving any issue that can arise in the task that they are working on.

Various empirical studies that report beneficial effects of the use of this practice have been conducted [10]–[24]. Some beneficial effects reported in these studies are that pair programming helps to produce shorter programs and achieve better designs; programs contain fewer defects than do those written individually, and pairs usually require less time to complete a task than do programmers working individually.

Although different aspects of pair programming have been studied, we have not found studies that examine the impact of using a tool support such as an integrated development environment (IDE) under a traditional approach (programmers working in the same place with the same computer). With the objective of obtaining a better understanding of the influence of the use of an IDE within this practice, in this work, we present the results of an empirical study in the form of a controlled experiment that investigates the effect on quality and cost of pair and solo programming with and without the use of an integrated development environment (IDE).

The remainder of the document is organized as follows: In section II, we describe the related work; in section III, we present the experimental setting; in section IV, the analysis and results are presented; in section V, we discuss the findings; and finally, in section VI, the conclusions are drawn.

II. RELATED WORK

The use of a tool support for pair programming (PP) has been commonly studied in a distributed setting, wherein pairs are located at different geographical locations and are working in a synchronous or asynchronous manner [25]–[31].

For example, Winkler *et al.* [31] define a systematic tool evaluation approach for distributed PP, authors also report an initial tool survey of open source tools. Concerning the integrated development environments (IDEs), authors identify some that could be considered under a distributed PP approach.

Schümmer and Lukosch [30] present a plug-in for the Eclipse integrated development environment, which allows distributed PP capabilities such as shared editing, project synchronization, shared program and test execution, user management, built-in chat communication, and a shared whiteboard. Authors observed that role switches did not occur as often as expected from a traditional setting (non-distributed) of PP. Similarly, Ho *et al.* [29] also present a plug-in for the Eclipse IDE that allows users in different locations to share a workspace so that they may work as if they were using the same computer. Authors discuss its use under a distributed PP approach.

Atsuta and Matsuura [28] propose an XP support environment for conducting pair programming activities under a distributed approach. Some of the functions proposed for such an environment are: the role shift, states notifications, a chat, an editor synchronization and a white board.

Natsu *et al.* [27] present a synchronous source code editor that allows two distributed software engineers to write a program using pair programming. Their proposal implements characteristics of groupware systems such as communication mechanisms, collaboration awareness, concurrency control, and a radar view of the documents, among others. The authors reported a preliminary evaluation of its proposal.

Stotts *et al.* [26] report the results of two distributed pair programming cases studies, where participants used a set of available off-the-shelf applications for collaborative software development. Authors observed that pairs induce better teamwork and communication within a virtual distributed team.

Maurer [25] presents a process-support environment that helps software development teams to maintain XP practices in a distributed setting. Among the features that this environment supports are: project coordination, user stories, information routing, team comunication, and pair programming.

Summarizing these previous works, we observe that the assessment of different support tools is commonly carried out in the context of distributed pair programming research. However, under a traditional setting of pair programming wherein pairs work in the same location with the same computer simultaneously, we have not found studies that investigate the effects of using a tool support such as an IDE.

The experiment we report in this article has its origins in the work of [24]. Gómez [24] conducted a controlled experiment as part of a course on Design of Experiments (DoE) in Software Engineering (SE). The experiment was conducted at the Faculty of Mathematics of the Autonomous University of Yucatan (Mexico) in the Software Engineering degree program. In this experiment, a Latin square design [32] was used.

The main characteristics of this experimental design are that there are two blocking factors. The treatment is present at each level of the first blocking factor as well as at each level of the second blocking factor. This design is arranged with an equal number of rows (blocking factor one) and columns (blocking factor two). Treatments are represented by Latin symbols, where each symbol is present exactly once in each row and exactly once in each column. An example of the structure of this design is shown in Table 1.

TABLE 1. Example of a Latin square layout.

A	B	C
B	C	A
C	A	B

In this type of experimental design, blocking is used to systematically isolate the undesired source of variation when comparing the treatments. In the case of the experiment reported in [24], a Latin square design was used in an attempt to block the program being coded and the tool support, thus reducing the undesired source of variation between the treatments of interest, namely, the pair and solo programming approaches. The Latin square design structure used in [24] is shown in Table 2.

TABLE 2. Latin square experimental design used in [24].

Program / tool support	IDE	Text editor
calculator	Solo programming	Pair programming
encoder	Pair programming	Solo programming

The aspects studied in [24] were duration (cost) and effort of 7 pairs and 7 solos who coded two programs with and without the use of an IDE. The duration was measured as the time, in minutes, needed to code the programming assignment, whereas effort was measured as the amount of labor spent on coding the programming assignment (measured in person-minutes).

With respect to duration, Gómez [24] reported a significant (at $\alpha = 0.1$) decrease in time of 28% in favor of pairs and a medium effect size $d = 0.65$.

Regarding effort, Gómez [24] reported a significant (at $\alpha = 0.1$) decrease in effort of 30% in favor of solos and a medium effect size $d = 0.64$.

Because the program and the tool support were treated as blocking factors (obeying the Latin square design used), it was not possible to assess for possible interactions between the main treatment (type of programming) and the blocking factors (tool support and program being coded). The variability observed in the measurements related to the tool support and coded programs suggests the need for further investigation of a possible influence between these two factors.

III. EXPERIMENTAL SETTING

Considering the issues discussed in the previous section, we decided to conduct another controlled experiment, therein varying the experimental design as well as the effect operationalizations. According to [33], [34], the experiment reported here can be considered as an operational replication. The experimental design is varied with the aim of examining a possible influence based on how either the tool support used for coding or the type of program being coded (programming assignment) may affect the type of programming (pair or solo).

With respect to the effect operationalizations, one of the aspects studied in [24] was the time (cost) that pairs and solos spent coding the programming assignments. We keep this aspect; in addition, we examine another aspect related to the quality of the software products (programming assignments) produced by the pairs and solos.

Following the Goal-Question-Metric (GQM) approach [35], which facilitates the identification of the object of study, purpose, quality focus, perspective and context of an experiment, we define the experiment reported here as follows:

"Study pair and solo programming with the purpose of evaluating how quality and cost could be affected by the use of an IDE or a text editor as tool support along with the programs being coded. This study is conducted from the point of view of the researcher within an academic context. This context is composed of junior-year students enrolled in a course of DoE, where they will code, in pairs or individually, two programming assignments with different tool support."

In the previous experiment definition, the treatments: programming type (pairs and solos), tool support (IDE and simple text editor) and programs being coded (programs A and B) act as the independent variables, whereas the operationalizations of quality and cost act as the dependent variables. Based on our previous experiment definition, following we present the derived null hypotheses.

Concerning quality:

- $H0_a$: Participants working in pairs and individually develop software products with similar quality.
- $H0_b$: Coding through an IDE and a simple text editor yields software products with similar quality.
- $H0_c$: Implementing the specification of program A and program B yields a software product with similar quality.
- $H0_d$: Software product quality is not affected by the relationship between the type of programming (pair or solo) and the type of tool support used.
- $H0_e$: Software product quality is not affected by the relationship between the type of programming and the type of program being coded.
- $H0_f$: Software product quality is not affected by the relationship between the type of tool support used and the type of program being coded.
- $H0_g$: Software product quality is not affected by the relationship between the type of programming, type of tool support and type of program being coded.

Concerning cost:

- $H0_h$: Participants working in pairs and individually spend similar time (cost) coding the programming assignments.
- $H0_i$: Coding using an IDE and a simple text editor is performed in a similar amount of time (cost).
- $H0_j$: The implementation of the specification of program A and program B is performed in a similar amount of time (cost).
- $H0_k$: Cost is not affected by the relationship between the type of programming (pair or solo) and the type of tool support used.
- $H0_l$: Cost is not affected by the relationship between the type of programming and the type of program being coded.
- $H0_m$: Cost is not affected by the relationship between the type of tool support used and the type of program being coded.
- $H0_n$: Cost is not affected by the relationship between the type of programming, tool support and type of program being coded.

For each null hypothesis, we defined a nondirectional hypothesis as the alternative hypothesis. Thus, the alternative hypotheses state that at least one of the treatments in each category (programming type, tool support and programming assignment) is different with regards to either quality or cost. In the case of the relationships among treatments, the alternative hypotheses state that at least one of the treatments belonging to: the type of programming and the type of tool support used; the type of programming and the type of program being coded; the type of tool support used and the type of program being coded; the type of programming, tool support and type of program being coded is different with regards to either quality or cost.

A. EXPERIMENTAL DESIGN

The defined hypotheses for this experiment will be tested using different measurements collected from the participants of this experiment. The collected measurements belong to four groups: (1) Participants working in pairs on programs A and B with an IDE, (2) participants working in pairs on programs A and B with a simple text editor, (3) participants working individually on programs A and B with an IDE, and (4) participants working individually on programs A and B with a simple text editor. With the collected measurements, we will contrast them regarding the defined null hypotheses.

TABLE 3. Experimental design layout used for the experiment.

P. Type	T. Support	Exp. Unit	Session 1 (Program A)	Session 2 (Program B)
Pairs	IDE	eu_{a1}	Meas. 1	Meas. 2
		eu_{a2}	...	...
		...	...	...
	Text editor	eu_{b1}	Meas. 1	Meas. 2
		eu_{b2}	...	...
		...	...	...
Solos	IDE	eu_{c1}	Meas. 1	Meas. 2
		eu_{c2}	...	...
		...	...	...
	Text editor	eu_{d1}	Meas. 1	Meas. 2
		eu_{d2}	...	...
		...	...	...

With the goal of collecting the maximum number of measurements possible, we used a factorial design with repeated measurements [36]. A factorial design enables the study of several factors and the interactions among them. In this experiment, we applied a 2^2 factorial design and selected the main factors as the type of programming (pair and solo) and the tool support (IDE and text editor). This design is then repeated (maintained on the same experimental unit) for each programming assignment in separated sessions. The layout of this experimental design is shown in Table 3.

As showed in Table 3, two measurements (Meas. 1 and Meas. 2) are going to be collected for the experimental unit belonging to the four treatment combinations. Extracting repeated measurements of the same experimental unit (pairs or solos) provides an efficient use of resources compared to extracting measurements from different experimental units [36]. Another characteristic of this experimental design is the reduction of the variance of estimates, thereby allowing statistical inferences to be made with fewer participants [36].

B. PARTICIPANTS, TASKS AND OBJECTS

The sample for this experiment was conformed by junior-year students (i.e., in their third year) enrolled in a DoE course in a Software Engineering (SE) program at the Autonomous University of Yucatan (Mexico). This kind of sampling, also know as convenience sampling, is commonly used in SE experiments. It is a non-probability sampling technique where the researcher selects the participants because of they are conveniently accessible and proximate to the experiment.

The experiment reported here was conducted during the summer semester of 2013, and it was scheduled to end in November 2013. In this experiment, there were 24 students (8 pairs and 8 solos) in total who assisted and finished all the programming assignments (out of a total of 28 students). According to the Dreyfus and Dreyfus programming expertise classification [37], we categorized participants as advanced beginners; the participants have working knowledge of key aspects of Java programming practice. On average, they reported having 1.94 (SD=0.97) years of experience with the Java programming language and

TABLE 4. Characteristics of the experimental units.

Exp. unit	Composition	Id participant	Measurements performed
eu_1	Male, Male	22, 5	2
eu_2	Male, Male	27, 17	2
eu_3	Female, Female	6, 24	2
eu_4	Male, Male	25, 18	1
eu_5	Male, Male	14, 3	2
eu_6	Female, Female	28, 7	2
eu_7	Male, Male	12, 4	2
eu_8	Male, Male	8, 19	2
eu_9	Male, Male	9, 11	2
eu_{10}	Male	16	1
eu_{11}	Male	2	2
eu_{12}	Male	10	1
eu_{13}	Male	15	2
eu_{14}	Female	1	2
eu_{15}	Male	23	2
eu_{16}	Male	20	2
eu_{17}	Male	26	2
eu_{18}	Female	21	2
eu_{19}	Male	13	2

1.81 (SD=1.03) years of experience with the NetBeans IDE. Verbal consent was obtained from student participants, and they were informally advised about the data retained and that anonymity was fully ensured. No sensitive data were collected for the experiment.

From a total of 356 credits, which is the minimum number of credits necessary to complete the SE degree curricula, at the time of the experiment, the participants had completed on average 213.40 (SD=49.92) credits of the SE degree (59.94% completion). Regarding the gender of the participants, there were 22 males and 6 females. Table 4 shows the gender distribution for each experimental unit, the identification number of each participant and the number of measurements performed. A total of 3 out of 19 experimental units did not attend all the planned experimental sessions (two sessions); therefore, only one measurement per aspect (quality and cost) was collected from these three experimental units.

Participants were randomized regarding to gender and allocated into four groups (treatment combinations) according to the 2^2 factorial design, thus having: (1) a group of pairs working with an IDE, (2) a group of pairs working with a text editor, (3) a group of solos working with an IDE, and (4) a group of solos working with a text editor. The treatment combinations were applied two times over the same experimental unit (the participants coded two programs in two different sessions), hence satisfying the factorial design with repeated measurements selected.

Although gender is a factor that can be analyzed [38], in this research we did not consider it as a study factor mainly because of the imbalance between the number of women and men participants, however we randomized participants according to gender with the aim of our findings can serve for quantitative synthesis [39] in works that seek to analyze gender in pair programming.

Before the experiment was conducted, we presented a talk to the students about eXtreme Programming with special focus on pair programming. In this talk, we explained the main concepts of this programming practice and how it can be applied. In another talk, we reinforced the concepts of pair programming and explained how to compile and run a Java program using only a text editor and the operating system console. Finally, we explained to students how to collect the measurements during the experimental sessions. The collection procedure consisted of writing down the amount of time that students spent writing a program (we asked them to record starting and ending times and compute the difference in minutes). In addition, we asked them to record only logical defects that they injected during coding.

We explained to students two basic types of defects that they can commit: syntax and logical defects. In a syntax defect, the program cannot be compiled; this is especially important for students working with a simple text editor because they do not receive syntax hints from the IDE. On the other hand, when a logical defect is committed, the program can be compiled and run, but it will not work properly according to its specification, i.e., it does not behave as intended. For this experiment we only focused on logical defects committed by the participants, leaving for future research the analysis of other types of defects, such as syntax defects. As tool support, pairs and solos utilized either the NetBeans IDE or a simple text editor (notepad, pico or nano) with the Java programming language. Printed forms were available for time and defect registration.

Prior to the experimental sessions, a training phase was conducted. The training phase enables additional control over experimental conditions, reducing undesired variations in the measurements. In two separate training sessions, participants coded two programs that were different to those employed in the experimental sessions. This training phase allowed the pairs to be immersed and to gain experience with the pair programming practice. The experience gained for pairs alleviates the issues discussed in pair programming experiments regarding the lack of training that pairs receive in comparison to solos programmers [11], [40].

We wrote the specification for two console programs that participants could code, compile, run and test during each experimental session. For the first program (identified as encoder, or program A), we asked participants to code a simple encoding-decoding program. Given a specified table that contains letter switches, the program must be able to encode or decode a line of text. The program receives two arguments: one that indicates whether to encode or decode the text and one that indicates the line of text (enclosed by quotation marks) to process.

For the second program (identified as calculator, or program B), we asked participants to write a simple calculator that evaluates expressions containing decimal numbers along with the operators addition ($+$), subtraction ($-$), multiplication ($\times$), and division ($\div$). If the expression is valid, the program prints the results on the screen; otherwise, the program prints a message indicating an invalid expression.

C. EXPERIMENT CONDUCT

Once the random assignment of participants to the treatment combinations was performed (this assignment was done prior to the training phase), the experimental units (pairs and solos) worked with the same treatment combination (type of programming and tool support assigned) during the training and experimental sessions, varying only the program being coded in each session.

The allotted time for each experimental session was 90 minutes. Both sessions were conducted in one of the computer classrooms of the university. Once most of the students were in the classroom, we started each session. In the first experimental session, we gave participants directions and projected the specification of the program to code (program A, encoder) onto the screen. In this session, two solo programmers spent more time coding than the planned time assigned to the session, namely, 107 and 147 minutes. Because the first of these two participants was nearly finished with the programming assignment, we decided to ask them to wait. We asked the second participant to pause their programming activities and restart them at home (while properly performing the measurement collection process).

The second experimental session was conducted in a similar manner as the previous session; we gave the participants directions and projected the second specification (program B, calculator) onto the screen. In this session, half of the experimental units (pairs and solos) finished the programming assignment on time. For the other half, we scheduled an extra session in the same classroom. In this extra session, two solos and one of the pairs did not finish the programming assignment; thus, we asked them to finish the assignment at home.

The programming assignment for each session was considered as finished or completed once we verified (in each experimental unit) the proper operation of the program against its specification.

D. METRICS

The metrics operationalized (dependent variables) for the effect constructs (quality and cost) were defined as: the number of defects injected per hour (product quality), and the elapsed time in minutes that the experimental units (pairs and solos) spent coding the programming assignment until it ran according to its specifications (cost).

The number of defects injected per hour is a common metric used in the software process arena [41], in this sense, we decided to use it in order to represent the quality construct. Following the suggestions in [42], we evaluated the product quality based on an external metric. It seems that the use of internal metrics for assessing product quality can lead to unreliable results [42], [43].

In the case of cost, we are interested in analyzing the elapsed time that each experimental unit (either pairs or solos) spend coding the assignments. Note that we differentiate cost

TABLE 5. Measurements collected for quality (defects injected per hour).

P. Type	T. Support	Exp. Unit	Session 1	Session 2
Pairs	IDE	eu_1	2.86	0.96
		eu_2	4.29	1.30
		eu_3	9.60	0.62
		eu_5	4.14	1.06
	Text editor	eu_6	0.92	0.87
		eu_7	3.43	1.68
		eu_8	1.33	0.85
		eu_9	4.39	1.15
Solos	IDE	eu_{11}	12.86	6.67
		eu_{13}	5.22	5.71
		eu_{14}	2.24	0.63
	Text editor	eu_{15}	2.14	0.67
		eu_{16}	1.25	1.24
		eu_{17}	2.86	1.80
		eu_{18}	0.41	0.83
		eu_{19}	1.00	0.67

TABLE 6. Measurements collected for cost (elapsed time in minutes).

P. Type	T. Support	Exp. Unit	Session 1	Session 2
Pairs	IDE	eu_1	42	125
		eu_2	14	92
		eu_3	25	97
		eu_5	58	340
	Text editor	eu_6	65	138
		eu_7	70	143
		eu_8	45	211
		eu_9	41	104
Solos	IDE	eu_{11}	28	36
		eu_{13}	23	63
		eu_{14}	107	479
	Text editor	eu_{15}	56	89
		eu_{16}	48	97
		eu_{17}	63	100
		eu_{18}	147	432
		eu_{19}	60	180

from effort, in the sense that effort is used to measure the amount of labor spent to perform a task. In the case of PP experiments, the effort is usually represented as the elapsed time multiplied by two [24], [44]. For both of the metrics we defined, we considered pairs and solos as a whole unit, i.e. the experimental unit.

After the experimental sessions were finished, we collected measurements from 28 participants; however, we had to withdraw the measurements of three experimental units (two solos and one pair) because the participants did not attend experimental session two. This is a restriction for the applied experimental design, it is necessary to collect all the repeated measurements from the experimental units [36]. Tables 5 and 6 show the measurements collected. Table 5 shows the measurements collected with respect to the number of defects injected per hour (product quality). Regarding the elapsed time in minutes required to code the programming assignments (cost), in the Table 6 we present the corresponding measurements collected.

IV. ANALYSIS AND RESULTS

In this section, we present both descriptive and inferential statistics for the collected measurements. In addition, we present results from a qualitative analysis referring to a questionnaire that participants responded to on their pair programming experience. Tables 7 and 8 show the descriptive statistics for the programming type, tool support and programming assignment with respect to product quality and cost.

TABLE 7. Descriptive statistics with regard to number of defects injected per hour (quality) for programming type, tool support and program being coded.

Factor	Level	n	$\bar{x}$	SD	Min.	Max.
Prog. Type	Pair	16	2.47	2.34	0.62	9.60
	Solo	16	2.89	3.30	0.41	12.86
Tool	IDE	14	4.15	3.65	0.62	12.86
	Text editor	18	1.53	1.06	0.41	4.39
Program	Encoder	16	3.68	3.32	0.41	12.86
	Calculator	16	1.67	1.81	0.62	6.67

TABLE 8. Descriptive statistics with respect to elapsed time (cost, in minutes) for programming type, tool support and program being coded.

Factor	Level	n	$\bar{x}$	SD	Min.	Max.
Prog. Type	Pair	16	100.62	81.94	14	340
	Solo	16	125.50	135.75	23	479
Tool	IDE	14	109.21	134.60	14	479
	Text editor	18	116.06	92.73	41	432
Program	Encoder	16	55.75	33.10	14	147
	Calculator	16	170.38	131.79	36	479

Regarding product quality (Table 7), on average, both types of programming appear to produce similar defect injection rates. In the case of the tool support, participants who worked with an IDE seem to have higher defect injection rates. With respect to the program being coded, it seems that participants had higher defect injection rates when coding the encoder program.

In the case of cost (Table 8), solo programmers seem to spend more time coding than do pairs. Both groups seem to require a similar amount of time with both an IDE and a simple text editor. With respect to the time required to code both programs, the calculator program required three-times as much time than did the encoder program.

The descriptive statistics provide an overview of the collected measurements; however, at this point, we are unable to draw any conclusions with confidence with regard to possible differences between treatments. Once we have an overview of the data, we will proceed with the inferential statistics to test the previously stated null hypotheses. The statistical model employed according to the factorial design with repeated measurements is defined in equation (1).

$$\begin{aligned} y_{ijkl} &= \mu + \alpha_i + \beta_j + d_{ijl} + \gamma_k(\alpha\beta)_{ij} + (\alpha\gamma)_{ik} \\ &\quad + (\beta\gamma)_{jk} + (\alpha\beta\gamma)_{ijk} + \epsilon_{ijkl}, \end{aligned} \tag{1}$$

TABLE 9. ANOVA results for product quality.

Component	DFn	DFd	F	p	$p < 0.05$
Programming type	1	12	0.939677	0.351493	
Tool support	1	12	7.913204	0.015660	*
Program	1	12	13.491128	0.003189	*
Prog. type:Tool support	1	12	2.308808	0.154542	
Prog. type:Program	1	12	1.336983	0.270068	
Tool support:Program	1	12	4.264636	0.061216	.
Prog. type:Tool:Program	1	12	0.152236	0.703244	

TABLE 10. Pairwise comparisons between tool support and program.

Comparison	Estimate	Std. Error	z value	Pr(> |z|)	
Text editor.Calculator − IDE.Calculator = 0	-1.3362	1.1550	-1.157	0.64122	
IDE.Encoder − IDE.Calculator = 0	3.4642	0.8457	4.096	< 0.001	*
Text editor.Encoder − IDE.Calculator = 0	-0.4509	1.1550	-0.390	0.97862	
IDE.Encoder − Text editor.Calculator = 0	4.8004	1.1550	4.156	< 0.001	*
Text editor.Encoder − Text editor.Calculator = 0	0.8853	0.7459	1.187	0.62198	
Text editor.Encoder − IDE.Encoder = 0	-3.9151	1.1550	-3.390	0.00356	*

where μ is the general mean, α_i is the effect of the ith treatment (programming type), β_j is the effect of the jth treatment (tool support), d_{ijl} is the random experimental error for the experimental units (pairs and solos) within treatments (programming type and tool support) with variance σ_d^2, γ_k is the effect of the kth program, $(\alpha\beta)_{ij}$ is the interaction between the programming type and the tool support, $(\alpha\gamma)_{ik}$ is the interaction between the programming type and the program being coded, $(\beta\gamma)_{jk}$ is the interaction between the tool support and the program being coded, and $(\alpha\beta\gamma)_{ijk}$ is the interaction between the programming type, tool support and the programming assignment. Finally, ϵ_{ijkl} is the normally distributed random experimental error on repeated measurements with variance σ_e^2.

To assess each of the components of this statistical model, an analysis of variance (ANOVA) is applied. ANOVA relies on an analysis of the total variability of the collected measurements and the variability partition according to different components (in this case, factors and their interactions). ANOVA provides a statistical test of whether the means of several groups (of measurements) are all equal. The null hypothesis is that all groups are simply random samples of the same population. This implies that all treatments have the same effect (perhaps none). Rejecting the null hypothesis implies that different treatments result in different effects. In this experiment, we have four groups of measurements (programming type and tool support combinations), with two repeated measurements per group (the two programming assignments).

ANOVA was applied through the use of the R package ez [45], which implements a function for the analysis of factorial designs with repeated measurements. Table 9 shows the ANOVA results with respect to product quality.

As shown in Table 9, the tool support and program components exhibit a significant difference at an alpha level of 0.05, suggesting that defect injection rates are different for different types of tool support (IDE and simple text editor) and for different programming assignments (encoder and calculator). If we set an alpha level of 0.1, which represents a confidence level of 90%, the interaction between tool support and program exhibits a significant difference. Estimating the effect sizes for these components, a generalized η^2 [46] of 0.38, 0.23 and 0.08 was observed for tool support, program, and the interaction between tool support and program, respectively. According to [47], these effect sizes can be interpreted as large ($\eta^2 > 0.14$), large and medium ($\eta^2 > 0.06$), respectively.

Because a significant interaction was observed, we conducted a post-hoc test with multiple pairwise comparisons with the goal of examining differences in all the level combinations between tool support and program. Table 10 shows the post-hoc test results using Tukey's method [48].

According to the results in Table 10, three pairwise comparisons exhibit significant differences (at $\alpha = 0.01$). In the first one, a significant difference of 3.5 defects injected per hour is observed between the encoder and calculator programs, both programs coded with an IDE. In the second one, a significant difference of 4.8 defects injected per hour is observed between the encoder program coded with an IDE and the calculator program coded with a text editor. In the third one, a decrease of 3.9 defects injected per hour is observed when a text editor is used to code the encoder program. A visual representation of this interaction (tool support and program) is given in Figure 1.

As shown in Figure 1, the use of an IDE for the encoder program produces higher defect injection rates than does the use of a text editor. In the case of the calculator program, the use of an IDE appears to reduce the defect injection rates, but only for pairs. Another representation of the defect injection rates between programming type and tool support organized by programming assignment is shown in Figure 2.

According to this figure (Figure 2), solo programmers exhibit a similar pattern when coding both programs, and the use of a text editor seems to significantly reduce the defect

TABLE 11. ANOVA results for cost.

Component	DFn	DFd	F	p	$p < 0.05$
Programming type	1	12	0.293834	0.597695	
Tool support	1	12	0.007055	0.934445	
Program	1	12	15.348790	0.002043	*
Prog. type:Tool support	1	12	0.000292	0.986640	
Prog. type:Program	1	12	0.034953	0.854816	
Tool support:Program	1	12	0.346383	0.567080	
Prog. type:Tool:Program	1	12	0.000002	0.998689	

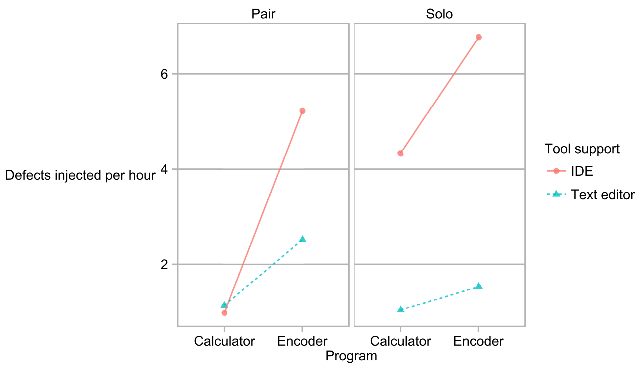

FIGURE 1. Interaction plot between tool support and program with regard to quality.

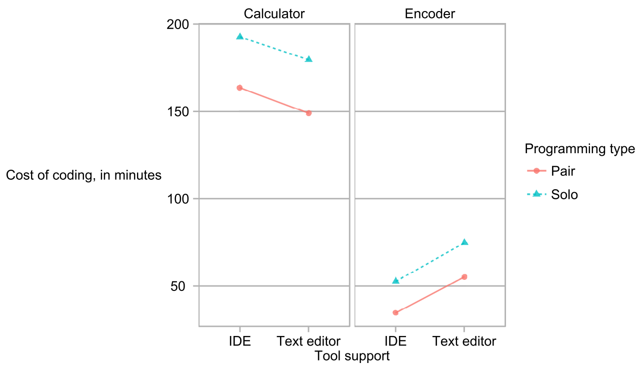

FIGURE 3. Interaction plot between programming type and tool support with respect to cost.

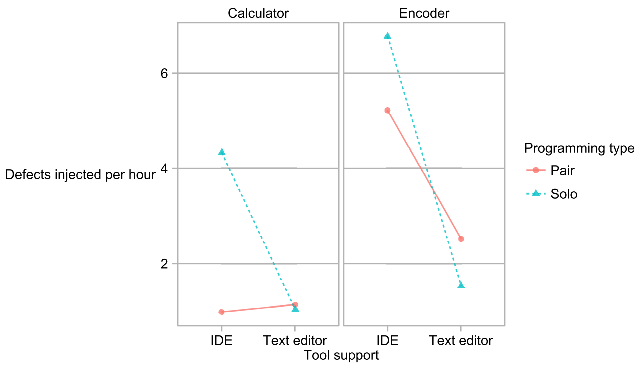

FIGURE 2. Interaction plot between programming type and tool support with respect to quality.

injection rates in comparison to the use of an IDE. This is partially supported for pairs working on the encoder program. This appears to be an interaction between programming type and tool support (although not significant), wherein pairs seem to inject fewer defects when using an IDE compared to solos. Conversely, solos seem to inject fewer defects when using a simple text editor compared to pairs. Regarding the program, a small number of defects were injected into the calculator program. In this program, pairs produce similar defect injection rates with either the use of an IDE or a text editor.

With regard to cost, measured as the time (in minutes) that participants spend coding the programming assignments, the analysis of variance is shown in Table 11. For this aspect, the program component exhibits a significant difference, therein suggesting a significant difference between the coded programs. This component shows an effect size η^2 of 0.28, which is interpreted as a large effect size [47].

This significant difference is shown in Figure 3; participants spent less time coding the encoder program than they did coding the calculator program. Pairs tend to require less time than do solos, although this difference is not significant. The use of an IDE seems to reduce the coding time for the encoder program; however, it increases when an IDE is used for coding the calculator program. This interaction is not significant, though.

To draw valid statistical conclusions, the employed statistical model was checked against the assumptions of normality, sphericity and randomness. One way of assessing normality is by examining whether the collected measurements conform to a normal distribution at each level of the within-subject factor. For the standardized residuals of each level of the within-subject factor, we used the JarqueâĂ"Bera test for normality [49], [50]. The null hypothesis of this test assumes that sample data originate from a normal distribution. Regarding quality, the standardized residuals for the encoder program show a p-value of 0.3202, and for the calculator program, a p-value of 0.0004 is found. Regarding cost, the standardized residuals for the encoder program show a p-value of 0.1133, and for the calculator program, a p-value of 0.1823 is found.

With regard to quality, the assumption of normality is violated for the calculator program. The observed outlier belongs to the experimental unit eu_{14}, which has the smallest defect injection rate in the programming type and tool support treatment combination. Although this experimental unit registered a higher than average number of defects detected (the average for this program was 3.18). This unit spent 479 minutes coding this program. In the case of the encoder program, the null hypothesis is accepted in favor of normality. Regarding cost, the null hypothesis is accepted in favor of normality for both programs.

The sphericity assumption implies that the variances of the differences between any two levels of within-subject factors (factors repeated over the same experimental unit) are similar. This assumption is always met when there are only two levels of within-subject factors [51], i.e., two repeated measurements, as in our case, where the two repeated measurements were for the encoder and calculator programs. With respect to the randomness assumption, the collected measurements were sampled randomly and independently of each other.

It is important to note that the previous assumptions are related to a univariate statistical model similar to the one described in equation 1. However, a repeated measurement model can be seen as a multivariate statistical model wherein the repeated measurements act as dependent (or response) variables. Under a multivariate statistical model, two common assumptions to assess are the multivariate normality and the homogeneity of covariance matrices [52].

A. QUALITATIVE RESULTS

After the experiment sessions were completed, participants working in pairs responded to a questionnaire regarding their perceptions of the use of pair programming. The questionnaire was responded to individually.

The first question referred to the degree of cohesion achieved (team jelling) during all experimental sessions. We defined a scale from 1 to 9, where 1 represents the lowest level of cohesion achieved and 9 the highest level of cohesion achieved. On average, the participants perceived a good level of cohesion with their respective partners (8.18, SD=0.73).

For the second question, we asked whether the cohesion was increased or decreased during the experimental sessions. With the exception of one respondent (6%), who answered to have the same level of cohesion during the sessions, the rest of respondents (94%) stated that the perceived cohesion increased during the sessions.

For the third question, we asked whether the participant would work on future assignments with the same partner given the achieved cohesion. 70% of the respondents would work on future assignments with the same partner, 12% would not and 18% perhaps would do it.

The fourth question was related to the role that participants were mainly involved in. 35% of the respondents stated the controller as being their main role, 41% acted as monitors and 24% responded as being equally involved in both roles.

Concerning whether participants changed role during the experiment (fifth question), 59% of the respondents answered that they did change roles, whereas 41% maintained the same role. Finally (sixth question), all respondents stated to have enjoyed the programming role that they were involved in.

V. DISCUSSION

Now that the analysis and results have been presented, in this section, we discuss them in relation to the hypotheses defined and with previous work.

In terms of quality, which is measured as the number of defects injected per hour, pairs and solos seem to develop software with a similar level of quality (failed to reject H_{0_a}). Coding using an IDE and a simple text editor yields software products with different levels of quality (H_{0_b} is rejected); the use of an IDE seems to generate higher defect injection rates than does the use of a simple text editor. The programming assignments being coded yield software products with different levels of quality (H_{0_c} is rejected); in this case, participants coding the program identified as encoder produced higher defect injection rates compared to those coding the calculator program. Software product quality is not affected by the relationship between the type of programming (pair or solo) and the type of tool support used (failed to reject H_{0_d}). Software product quality is not affected by the relationship between the type of programming and the type of program being coded (failed to reject H_{0_e}). Software product quality is affected by the relationship between the type of tool support used and the type of program being coded (H_{0_f} is rejected). Software product quality is not affected by the relationship between the type of programming, tool support and programming assignment (failed to reject H_{0_g}).

Because a significant interaction was observed (at $\alpha = 0.1$) between tool support and the programming assignment, interpretations should be based on interaction effects and not on main effects. A significant difference is observed for the programming assignment when it is coded with an IDE; the use of an IDE produced greater defect injection rates for the encoder program. Another significant difference is observed for the calculator program coded with a simple text editor and the encoder program coded with an IDE; the latter combination (IDE-encoder) yielded the highest defect injection rates. A third significant difference is observed for tool support and the encoder program. In this programming assignment, the use of a simple text editor produced lower defect detection rates.

Although on average the calculator program presents a lower cyclomatic complexity ($\overline{VG} = 6.93, SD = 7.79$) compared to the encoder program ($\overline{VG} = 16.25, SD = 19.01$), the calculator program is more complicated to code in the sense that it demands greater effort to be implemented. This programming assignment implies certain knowledge on how to address regular expressions.

According to our results, it seems that pairs, irrespective of the tool support that they use, tend to produce software products with better quality when they work on complex coding tasks (such as the calculator program used in our experiment); in this case, the tool support seems to have no effect (as shown in Figure 1). Our findings reinforce those reported in [18], [44], wherein has been observed beneficial effects when junior pairs work on more complex systems [44], and also has been observed that pair programming performs well when novice pairs encounters challenging programming problems. On the other hand, when coding tasks are less complicated (such as the encoder program used in our experiment), pairs and solos seem to produce software with better quality when a simple text editor is used.

In terms of cost, which is measured as the time required in minutes to complete the programming assignment, pairs and solos seem to require similar amounts of time to code the programming assignments (failed to reject H_{0_h}). The cost of coding using an IDE and that using a simple text editor are equivalent (failed to reject H_{0_i}). The programming assignments require different amounts of time to be coded (H_{0_j} is rejected); in this case, participants coding the encoder program required less time. Cost is not affected by the relationship between the type of programming (pair or solo) and the type of tool support used (failed to reject H_{0_k}). Cost is not affected by the relationship between the type of programming and the type of program being coded (failed to reject H_{0_l}). Cost is not affected by the relationship between the type of tool support used and the type of program being coded (failed to reject H_{0_m}). Cost is not affected by the relationship between the type of programming, tool support and the programming assignment (failed to reject H_{0_n}).

With respect to related work, our findings are compared only with regard to cost. This is because effort is not analyzed in the present work and because product quality is not analyzed in [24]. Regarding cost, our results suggest that pairs complete programming assignments in less time than do solos, but this difference is not significant. Conversely, the authors in [24] observed a significant difference (at $\alpha = 0.1$) in favor of pairs, whom also completed the assignment in less time than did solos. In this sense, our results are similar to those reported in [17], [44], and [53]–[58], wherein the authors did not observe significant differences when applying the pair programming practice.

In reference to the programming assignments, our findings reinforce those reported in [24]; the calculator program requires more time to code than does the encoder program. Regarding tool support, our findings are also congruous with those in [24]; the cost of coding is similar when using an IDE compared to when using a text editor.

Because tool support and program being coded where used as blocking factors in [24], it was not possible to assess possible interaction effects; however, this was alleviated with the experimental design used in this work. This design allowed the assessment of interactions between treatments of interest regarding quality and cost.

A. STUDY LIMITATIONS

Empirical studies are subject to undermining threats. Next, we describe strategies that we followed to minimize the threats to validity [59].

Regarding conclusion validity, the measurements collected during the experiment satisfy the principles of normality,[1] sphericity and randomness. However, although we checked for the statistical model assumptions, in this experiment we used a small sample which may have an impact on the conclusion validity.

With respect to internal validity, the participants were randomly assigned to treatments, which reduced learning effects. Boredom or fatigue was reduced by using alternating training and experimental sessions. The experimental units were placed in the same classroom, worked under the same conditions, and sat apart, with no interaction.

Concerning construct validity, cause constructs were operationalized in the same manner as in previous studies on the topic; effect constructs were operationalized in the same manner as in [41] (for quality) and as in [24] (for cost).

With respect to external validity, the use of students instead of practitioners might have compromised this type of validity. However, there is evidence that suggests that in some contexts, the results of empirical studies that employ students with enough technical skills are equivalent to the results of empirical studies that use practitioners [60], [61]. In the context of pair programming studies, there is evidence in favor of this claim: pair programming studies that employ practitioners [44], [54] report findings that are similar to studies employing students [53], [58]. In this respect, the participants in this experiment reported having almost two years of experience with the Java programming language and almost two years of experience with the NetBeans IDE.

Because there is scarce evidence on the use of an IDE within the pair programming practice (under a traditional approach), we decided first to run the experiment under an academic setting with undergraduate students. However our experiment can serve as a basis for conducting future replications in other academic or industrial settings.

VI. CONCLUSION

In this work, we presented a controlled experiment that assessed the quality and cost of pair and solo programming with and without the use of an integrated development environment (IDE). Although different aspects of pair programming have been studied, there is no work that examines the effects of using or not using a tool support such as an IDE under a traditional setting of PP, in which pairs work at the same location at the same time with the same computer.

In terms of quality, our results suggest that the use of an IDE produces higher defect injection rates (for both pairs and solos) when the programming assignment is not very complicated. Nevertheless, defect injection rates seem to decrease when pairs work on more complicated programming assignments, irrespective of the tool support that they use.

[1] Particularly, in the case of cost, normality is satisfied; for quality, normality is assumed only for the encoder program. In reference to the calculator program, a departure from normality was observed due to the presence of an outlier, as discussed in Section IV.

At first glance, instinct may suggest that the use of an IDE is more effective than the use of a text editor; however, the evidence presented here suggests the opposite. One possible reason for our findings is that when entry-level programmers use an IDE, it is common for programmers to write some lines of code and then compile and run it (as we observed in this experiment). It is possible that this mechanism interrupts the concentration flow required for coding. On the other hand, when a text editor is used for coding, programmers tend to prolong the compiling and running process; they have to use a console and manually perform these operations. It is possible that this mechanism encourages a longer concentration flow, which in our case produced lower defect injection rates. In an academic context, these findings suggest that students enrolled in programming courses should reinforce the use of a text editor for a certain period of time with the goal of obtaining greater concentration while coding.

In terms of cost, the programming assignment effects the time required to complete the assignment. Finally, regarding both studied aspects (quality and cost), pairs and solos behave in a similar manner. Future replications of this experiment need to be conducted to gain additional insight into the findings presented here.

REFERENCES

[1] M. M. M. Safwan, G. Thavarajah, N. Vijayarajah, K. Senduran, and C. D. Manawadu, "An empirical study of agile software development methodologies: A Sri Lankan perspective," *Int. J. Comput. Appl.*, vol. 84, no. 8, pp. 1–7, Dec. 2013.

[2] P. Rodríguez, J. Markkula, M. Oivo, and K. Turula, "Survey on agile and lean usage in finnish software industry," in *Proc. ACM IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Sep. 2012, pp. 139–148.

[3] Y. Y. Yusuf and E. O. Adeleye, "A comparative study of lean and agile manufacturing with a related survey of current practices in the UK," *Int. J. Prod. Res.*, vol. 40, no. 17, pp. 4545–4562, 2002.

[4] H. Corbucci, A. Goldman, E. Katayama, F. Kon, C. Melo, and V. Santos, "Genesis and evolution of the agile movement in Brazil—Perspective from academia and industry," in *Proc. 25th Brazilian Symp. Softw. Eng. (SBES)*, Sep. 2011, pp. 98–107.

[5] O. Salo and P. Abrahamsson, "Agile methods in European embedded software development organisations: A survey on the actual use and usefulness of extreme programming and scrum," *IET Softw.*, vol. 2, no. 1, pp. 58–64, Feb. 2008.

[6] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Inf. Softw. Technol.*, vol. 50, nos. 9–10, pp. 833–859, Aug. 2008.

[7] A. Begel and N. Nagappan, "Usage and perceptions of agile software development in an industrial context: An exploratory study," in *Proc. 1st Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Sep. 2007, pp. 255–264.

[8] K. Beck, "Embracing change with extreme programming," *Computer*, vol. 32, no. 10, pp. 70–77, Oct. 1999.

[9] K. Beck, *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley, 2000.

[10] J. T. Nosek, "The case for collaborative programming," *Commun. ACM*, vol. 41, no. 3, pp. 105–108, Mar. 1998.

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

[12] J. E. Tomayko, "A comparison of pair programming to inspections for software defect reduction," *Comput. Sci. Edu.*, vol. 12, no. 3, pp. 213–222, 2002.

[13] C. McDowell, B. Hanks, and L. Werner, "Experimenting with pair programming in the classroom," in *Proc. 8th Annu. Conf. Innov. Technol. Comput. Sci. Edu. (ITiCSE)*, 2003, pp. 60–64.

[14] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald, "The impact of pair programming on student performance, perception and persistence," in *Proc. 25th Int. Conf. Softw. Eng. (ICSE)*, 2003, pp. 602–607.

[15] K. M. Lui and K. C. C. Chan, "When does a pair outperform two individuals?" in *Proc. 4th Int. Conf. Extreme Programm. Agile Process. Softw. Eng. (XP)*, 2003, pp. 225–233.

[16] G. Canfora, A. Cimitile, and C. A. Visaggio, "Empirical study on the productivity of the pair programming," in *Extreme Programming and Agile Processes in Software Engineering* (Lecture Notes in Computer Science), vol. 3556, H. Baumeister, M. Marchesi, and M. Holcombe, Eds. Berlin, Germany: Springer, 2005, pp. 92–99.

[17] M. M. Müller, "Two controlled experiments concerning the comparison of pair programming to peer review," *J. Syst. Softw.*, vol. 78, no. 2, pp. 166–179, 2005.

[18] K. M. Lui and K. C. C. Chan, "Pair programming productivity: Novice-novice vs. expert-expert," *Int. J. Human-Comput. Stud.*, vol. 64, no. 9, pp. 915–925, Sep. 2006.

[19] S. Xu and V. Rajlich, "Empirical validation of test-driven pair programming in game development," in *Proc. 5th IEEE/ACIS Int. Conf. Comput. Inf. Sci., 1st IEEE/ACIS Int. Workshop Compon.-Based Softw. Eng., Softw. Archit. Reuse (ICIS-COMSAR)*, Jul. 2006, pp. 500–505.

[20] M. A. Domino, R. W. Collins, and A. R. Hevner, "Controlled experimentation on adaptations of pair programming," *Inf. Technol. Manage.*, vol. 8, no. 4, pp. 297–312, Dec. 2007.

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

[22] T. Bipp, A. Lepper, and D. Schmedding, "Pair programming in software development teams—An empirical study of its benefits," *Inf. Softw. Technol.*, vol. 50, no. 3, pp. 231–240, 2008.

[23] K. M. Lui, K. C. C. Chan, and J. Nosek, "The effect of pairs in program design tasks," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 197–211, Mar. 2008.

[24] O. S. Gómez, J. L. Batún, and R. A. Aguilar, "Pair versus solo programming—An experience report from a course on design of experiments in software engineering," *Int. J. Comput. Sci. Issues*, vol. 10, no. 1, pp. 734–742, Jan. 2013.

[25] F. Maurer, "Supporting distributed extreme programming," in *Extreme Programming and Agile Methods—XP/Agile Universe* (Lecture Notes in Computer Science), vol. 2418, D. Wells and L. Williams, Eds. Berlin, Germany: Springer, 2002, pp. 13–22.

[26] D. Stotts, L. Williams, N. Nagappan, P. Baheti, D. Jen, and A. Jackson, "Virtual teaming: Experiments and experiences with distributed pair programming," in *Extreme Programming and Agile Methods—XP/Agile Universe* (Lecture Notes in Computer Science), vol. 2753, F. Maurer and D. Wells, Eds. Berlin, Germany: Springer, 2003, pp. 129–141.

[27] H. Natsu, J. Favela, A. L. Moran, D. Decouchant, and A. M. Martinez-Enriquez, "Distributed pair programming on the Web," in *Proc. 4th Mexican Int. Conf. Comput. Sci. (ENC)*, Sep. 2003, pp. 81–88.

[28] S. Atsuta and S. Matsuura, "eXtreme programming support tool in distributed environment," in *Proc. 28th Annu. Int. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 2, Sep. 2004, pp. 32–33.

[29] C.-W. Ho, S. Raha, E. Gehringer, and L. Williams, "Sangam: A distributed pair programming plug-in for Eclipse," in *Proc. OOPSLA Workshop Eclipse Technol. eXchange (eclipse)*, 2004, pp. 73–77.

[30] T. Schümmer and S. Lukosch, "Understanding tools and practices for distributed pair programming," *J. Universal Comput. Sci.*, vol. 15, no. 16, pp. 3101–3125, Oct. 2009.

[31] D. Winkler, S. Biffl, and A. Kaltenbach, "Evaluating tools that support pair programming in a distributed engineering environment," in *Proc. 14th Int. Conf. Eval. Assessment Softw. Eng. (EASE)*, 2010, pp. 54–63.

[32] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*. Norwell, MA, USA: Kluwer, 2001.

[33] O. S. Gómez, N. Juristo, and S. Vegas, "Understanding replication of experiments in software engineering: A classification," *Inf. Softw. Technol.*, vol. 56, no. 8, pp. 1033–1048, 2014.

[34] O. S. Gómez, "Tipología de Replicaciones para la Síntesis de Experimentos en Ingeniería del Software," Ph.D. dissertation, Facultad Informática, Univ. Politécnica Madrid, Madrid, Spain, May 2012.

[35] V. Basili, G. Caldiera, and H. Rombach, "Goal question metric paradigm," in *Encyclopedia of Software Engineering*. New York, NY, USA: Wiley, 1994, pp. 528–532.

[36] R. O. Kuehl, *Design of Experiments: Statistical Principles of Research Design and Analysis*, 2nd ed. Pacific Grove, CA, USA: Duxbury-Thomson Learning, 2000.

[37] H. L. Dreyfus, S. E. Dreyfus, and T. Athanasiou, *Mind Over Machine: The Power of Human Intuition and Expertise in the Era of the Computer*. New York, NY, USA: Blackwell, 1986.

[38] K. S. Choi, "A comparative analysis of different gender pair combinations in pair programming," *Behaviour Inf. Technol.*, vol. 34, no. 8, pp. 825–837, Aug. 2015.

[39] L. V. Hedges and I. Olkin, *Statistical Methods for Meta-Analysis*. Orlando, FL, USA: Academic, 1985.

[40] J. Aranda. (Mar. 2007). *Pair Programming Evaluated*, accessed on Aug. 29, 2015. [Online]. Available: https://catenary.wordpress.com/2007/03/12/pair-programming-evaluated/

[41] W. S. Humphrey, *A Discipline for Software Engineering*. Boston, MA, USA: Addison-Wesley, 1995.

[42] N. Salleh, E. Mendes, and J. Grundy, "Empirical studies of pair programming for CS/SE teaching in higher education: A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 37, no. 4, pp. 509–525, Jul./Aug. 2011.

[43] J. Vanhanen and C. Lassenius, "Effects of pair programming at the development team level: An experiment," in *Proc. Int. Symp. Empirical Softw. Eng.*, Nov. 2005, p. 10.

[44] E. Arisholm, H. Gallis, T. Dybå, and D. I. Sjøberg, "Evaluating pair programming with respect to system complexity and programmer expertise," *IEEE Trans. Softw. Eng.*, vol. 33, no. 2, pp. 65–86, Feb. 2007.

[45] M. A. Lawrence. (2013). *ez: Easy Analysis and Visualization of Factorial Experiments, R Package Version 4.2-2*. [Online]. Available: http://CRAN.R-project.org/package=ez

[46] R. Bakeman, "Recommended effect size statistics for repeated measures designs," *Behavior Res. Methods*, vol. 37, no. 3, pp. 379–384, 2005.

[47] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Hillsdale, NJ, USA: Lawrence Erlbaum Associates, 1988.

[48] J. W. Tukey, "Comparing individual means in the analysis of variance," *Biometrics*, vol. 5, no. 2, pp. 99–114, 1949.

[49] A. K. Bera and C. M. Jarque, "Model specification tests: A simultaneous approach," *J. Econometrics*, vol. 20, no. 1, pp. 59–82, 1982.

[50] C. M. Jarque and A. K. Bera, "A test for normality of observations and regression residuals," *Int. Statist. Rev.*, vol. 55, no. 2, pp. 163–172, Aug. 1987.

[51] S. E. Maxwell, H. D. Delaney, and K. Kelley, *Designing Experiments and Analyzing Data: A Model Comparison Perspective*, 2nd ed. Orlando, FL, USA: Academic, May 2003.

[52] R. A. Johnson and D. W. Wichern, Eds., *Applied Multivariate Statistical Analysis*. Upper Saddle River, NJ, USA: Prentice-Hall, 1988.

[53] J. Nawrocki and A. Wojciechowski, "Experimental evaluation of pair programming," in *Proc. 12th Eur. Softw. Control Metrics Conf.*, London, U.K., Apr. 2001, pp. 269–276.

[54] M. Rostaher and M. Hericko, "Tracking test first pair programming—An experiment," in *Extreme Programming and Agile Methods—XP/Agile Universe* (Lecture Notes in Computer Science), vol. 2418, D. Wells and L. Williams, Eds. Berlin, Germany: Springer, 2002, pp. 174–184.

[55] S. Heiberg, U. Puus, P. Salumaa, and A. Seeba, "Pair-programming effect on developers productivity," in *Extreme Programming and Agile Processes in Software Engineering* (Lecture Notes in Computer Science), vol. 2675, M. Marchesi and G. Succi, Eds. Berlin, Germany: Springer, 2003, pp. 215–224.

[56] S. F. Freeman, B. K. Jaeger, and J. C. Brougham, "Pair programming: More learning and less anxiety in a first programming course," in *Proc. ASEE Ann. Conf.*, 2003, pp. 8885–8893.

[57] L. Madeyski, "Preliminary analysis of the effects of pair programming and test-driven development on the external code quality," in *Proc. Conf. Softw. Eng., Evol. Emerg. Technol.*, 2005, pp. 113–123.

[58] L. Madeyski, "The impact of pair programming and test-driven development on package dependencies in object-oriented design—An experiment," in *Product-Focused Software Process Improvement* (Lecture Notes in Computer Science), vol. 4034, J. Münch and M. Vierimaa, Eds. Berlin, Germany: Springer, 2006, pp. 278–289.

[59] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Designs for Research*. Boston, MA, USA: Houghton Mifflin, Jun. 1963.

[60] P. Runeson, "Using students as experiment subjects—An analysis on graduate and freshmen student data," in *Proc. 7th Int. Conf. Empirical Assess. Softw. Eng.*, 2003, pp. 95–102.

[61] M. Ciolkowski, D. Muthig, and J. Rech, "Using academic courses for empirical validation of software development processes," in *Proc. 30th EUROMICRO Conf.*, 2004, pp. 354–361.

OMAR S. GÓMEZ received the master's degree in software engineering from the Center for Research in Mathematics, CIMAT, Mexico, and the Ph.D. degree in software and systems engineering from the Technical University of Madrid, Spain. He is currently pursuing the Ph.D. degree with the University of Oulu, Finland. He was a SENESCYT-Prometeo Researcher (initiative of the Ecuadorian Government that seeks to strengthen research, academy, and knowledge transference) with the Polytechnic School of Chimborazo, Ecuador. He is currently a Computer Engineer with the University of Guadalajara, Mexico. He is also an Adjunct Associate Professor with the Escuela Superior Politécnica de Chimborazo, Ecuador. His research interest focuses on experimentation in software engineering as well as on issues related to quality and software design.

ANTONIO A. AGUILETA received the bachelor's degree in computer science from the Autonomous University of Yucatan, Mexico, and the master's degree in computing science from the Monterrey Institute of Technology and Higher Education, ITESM, Mexico. He is currently a full-time Professor and member of the Academic Group of Technologies for Training in Software Engineering with the Faculty of Mathematics, Universidad Autónoma de Yucatán. His research interests involve software engineering and educational informatics.

RAÚL A. AGUILAR received the bachelor's degree in computer science from the Autonomous University of Yucatan, Mexico, and the Ph.D. degree in computer science from the Technical University of Madrid, Spain. He is currently a full-time Professor with the Faculty of Mathematics, Universidad Autónoma de Yucatán. His research interests include software engineering and educational informatics.

JUAN P. UCÁN received the degree in computer science from the Faculty of Mathematics, Autonomous University of Yucatan, Mexico, the master's degree in computer systems, with specialization in software engineering, from the Technological Institute of Merida, Mexico, and the Ph.D. degree in computer systems from Southern University, Mexico. He is currently a full-time Professor with the Faculty of Mathematics and a member of the Academic Group of Technologies for Training in software engineering, Universidad Autónoma de Yucatán. His research interest focuses on topics related to software engineering, Web engineering, and educational informatics.

RAÚL H. ROSERO received the degree in computer engineering from the Central University of Ecuador, the Higher Diploma degree in software development process management from the Army University, Ecuador, and the master's degree in applied computer science from the Polytechnic School of Chimborazo, Equador. He is currently pursuing the Ph.D. degree in systems and computer engineering with the National University of San Marcos, Peru. His current research area is testing, Agile methods, and the verification and validation of software.

KAREN CORTES-VERDIN received the bachelor's degree in informatics from the University of Veracruz, Mexico, the M.Sc. degree in information systems engineering from the University of Manchester, Institute of Science and Technology, the master's degree in software engineering from the Center for Research in Mathematics, and the Ph.D. degree in computer science from the Center for Research in Mathematics. She is currently a Professor with Universidad Veracruzana, Mexico. Her research interests include software architecture, software design, software product lines. and software quality.