An Observational Study on the Maintainability Characteristics of the Procedural and Object-oriented Programming Paradigms

Wladymir Brborich
Departamento de Ciencias de la Computación
Universidad de las Fuerzas Armadas ESPE
Sangolquí, Ecuador
wabrborich@hotmail.com

Bryan Oscullo
Departamento de Ciencias de la Computación
Universidad de las Fuerzas Armadas ESPE
Sangolquí, Ecuador
bryanoscullo@gmail.com

Jorge Edison Lascano
Departamento de Ciencias de la Computación
Universidad de las Fuerzas Armadas ESPE
Sangolquí, Ecuador
jelascano@espe.edu.ec

Stephen Clyde
Computer Science Department
Utah State University
Logan, USA
Stephen.Clyde@usu.edu

Abstract—**This paper presents an observational study that takes an initial step towards answering the question of how the selection of a programming paradigm affects the maintainability of a software system. Answers to this question are important because they can provide insight on how the maintainability characteristics of different programming paradigms could be leveraged to improve software-engineering education. The observational study compares developers' effectiveness and speed performing pre-determined maintenance tasks on two equivalent variations of a web-based application: one based on Procedural Programming and one based on Object-Oriented Programming. Presented within this paper are the study's design, metrics measuring maintenance effectiveness and speed, and a statistical analysis of the results, which provides evidence that the maintainability characteristics of the two paradigms are different within the context of this research. In addition, as an observational study, some of its important contributions are lessons learned about the experiment design process and ideas for new hypotheses. These are presented in preparation for a future, broader research, and larger-scale experiment. Finally, we discuss some forward-thinking ideas about how differences in the maintenance characteristics of programming paradigms may influence on software-engineering education.**

Keywords—programming paradigms, object-oriented programming, procedural programming paradigm, maintainability, software-engineering education

I. Introduction

A programming paradigm is a way of thinking about, designing, and implementing software [1]. It is more general than a programming language because it does not define a specific set of symbols or grammar. Rather, it identifies core concepts or abstractions that a programmer might use when thinking about or working with software. As such, a paradigm can be characterized in terms of the principles, practices, and patterns that it espouses. It can also be characterized, or at least explained, in terms of the conceptual modeling languages that reflect its core concepts and that encourage developers to adhere to its principles. For example, Flow Charts and Control-Flow Diagrams can be used to teach someone to think according to the Procedural Programming (PP) paradigm [2], whereas UML Class, Interaction, and State Diagrams can help someone understand Object-Oriented Programming (OOP) [3].

Educators have long debated the choice of programming paradigms for computer-science curricula, directly in terms of which contains the most important concepts and indirectly in terms of the best programming languages for their classes. However, rarely has this debate included discussion points about how a given programming paradigm can affect the maintainability of software and therefore influence the education of software engineering students.

Maintainability, which is defined as the ease with which software can be modified to correct failures, increase performance, or adapt to different conditions [4], plays an important role in the success of software projects [5][6][7]. Furthermore, the ability of a development team to maintain software, effectively and efficiently, is essential for that team's long-term success [8].

Therefore, teaching students to perform maintenance tasks effectively and efficiently should be a critical component in a software-engineering educational program, which brings us back to the choice of programming paradigms. Which paradigm would be best for teaching students to create more maintainable software? Or, more importantly, how does a given paradigm impact what students learn about maintenance? Unfortunately, answers to these high-level questions depend on answers to more fundamental questions, like "how can we measure the maintainability of a software system?", "can we infer something about the maintainability characteristics of a programming paradigm from the maintainability of software systems built using that paradigm?", and "are there differences in maintainability characteristics among programming paradigms?"

Taking an initial step towards answering the last of these questions is the focus of this paper. Specifically, we present an observational study, conducted in the context of students at Universidad de las Fuerzas Armadas ESPE (UFA-ESPE), that compares the effectiveness and speed with which subjects (i.e., students) can complete a set of maintenance tasks on functionally equivalent programs developed with PP and OOP. Answers to other fundamental questions, although

978-1-7281-6807-4/20/$31.00 ©2020 IEEE

important, are beyond the scope of this paper. Nevertheless, some of the related work presented in Section II touches on the first question, namely, how to measure maintainability.

The rest of the paper will proceed as follows. Section II provides some additional background on programming paradigms, as well as maintainability. It also refers to sources that gave us a starting point for designing our experimental method, which Section III explains in detail. Section III also defines two maintainability metrics: one for effectiveness and one for speed. Section IV presents the results of the study, which indicate that the maintenance characteristics of PP and OOP are different. Although these results cannot be generalized to a broader context because of the limitations of the observational study, they are sufficiently interesting to warrant further investigation through a larger scale experiment. Section V discusses some of the unexpected aspects of the findings, as well as post-study insights about the nature of study, the impact of external variables, and ideas for improving the experimental method. Finally, Section VI summarizes the contributions of this paper and briefly discusses future work on addressing questions about the impact of the paradigm maintenance characteristics on software-engineering education.

II. Background

A. Programming Paradigms

Since Floyd first applied the term "paradigm" to programming in 1978 [1], many interesting and varied paradigms have emerged that allow developers to think about software in different ways. For example, with Logic or Constraint Programming [9], developers create programs as unordered collections of rules. With Functional Programming [10], developers construct systems using stateless functions and immutable data. With Dataflow Programming, programmers model systems as direct graphs where the nodes represent operations and the edges represent data flows [11].

The largest and most commonly used category of paradigms is Imperative Programming, which includes both PP and OOP – perhaps the two most widely used paradigms today [12]. Imperative Programming reflects the Von Neumann computer architecture, so programming languages that support paradigms in this category typically have three core features: state (i.e., stored data), instructions which manipulate that stored data, and the sequential execution of instructions [13][14].

With PP, developers organize programs around procedural abstractions, i.e., functions or subroutines that consist of small, reusable instruction blocks. PP encourages splitting large or complex chunks of code into smaller and more cohesive chunks [15][16].

OOP, on the other hand, encourages developers to organize programs around problem-domain abstractions, namely objects that have state and behavior [17]. With class-based OOP languages, like Java, C#, and C++, each object belongs to a class that defines the possible states and behaviors [13]. In contrast, object-based languages (also called prototype-based languages) allow programmers to work with objects that define properties directly or can inherit them from other objects (i.e., prototypes) [18].

Programming paradigms are not independent of each other; ideas from one can be found in others. For example, the principle of striving for loosely coupled and highly cohesive chunks of code in PP, also exists in OOP, albeit with minor differences [19].

Furthermore, recent versions of many popular programming languages, have begun to support multiple programming paradigms and mixed-paradigm development. For example, with the introduction of Lambda functions, object-oriented languages such as Java, C#, and C++ now support a degree of Functional Programming [20].

B. Maintainability

M.A. Sicilia lists three factors that affect maintainability [21]: (1) good software documentation; (2) maintainability as an integral part of the development process; and (3) human comprehension of computer programs, i.e., comprehensive and readable software with appropriate documentation facilitates software´s maintenance, which must be present in each development-process stage so it will be less intrusive at the end of the process.

Although these factors are important for project teams to understand and monitor, they are difficult to measure quantitatively. So, they do not represent a suitable basis for comparing how two programming paradigms affect maintainability.

In fact, measuring maintainability is not a trivial task [22]. Maintainability metrics are measuring devices for gathering information about the ability to modify a software [4]. Information on maintainability of a software system is of great importance to the developer team, because it can help the team manage resources, schedule, set expectations, and determine a product's end of life.

Many experts have developed metrics for measuring code maintainability, documentation, and product as a whole [4][7][23]. In 2013, J. Saraiva conducted a systematic mapping study that identified 570 metrics related to software maintenance for OO only, and of these, 36 were found to be commonly used in academic research.

One notable source of software metrics is the software product quality model in ISO 25010 [5], which consists of eight quality characteristics, one of them being maintainability with its sub dimensions of modularity, analyzability, modifiability, testability and reusability. Of these, modifiability has the most direct relevance to this study, since it has the biggest impact on maintainability and development costs of a product (between 40% and 70% of the total cost) [24].

Modifiability can be measured in terms of effort supplied by individuals who complete maintenance tasks [6]. Fernández-Sáez et al. use effectiveness and efficiency metrics to measure modifiability in their experiment [25], which inspired the definitions of our maintenance effectiveness and speed metrics, see Section 3.

According to Cornelius [6], there are three 'intentions' for software maintenance: 1) perfective maintenance, which is related to system enhancements; 2) adaptive maintenance, which is related to changing the system to adapt to changes in requirements or its environment, and 3) corrective maintenance, which is related to error and failure corrections. For this research, we focus on adaptive and corrective maintenance.

C. Experimental Method

We use the experimental method described by Genero et al. [26]. The author describes a process to define, execute, gather, and interpret data from an experiment in a software engineering context. This experimental method consists of four sections: goal definition, planning (context selection, variable selection, hypotheses formulation, experiment design, instrumentation), operation (experiment preparation, execution), and analysis.

III. Method

A. Goal

As mentioned in Section I, the main goal of this observational study is to compare the maintainability of two variations of a program, each implemented using a different programming paradigm, namely PP and OOP. The context for study consists of software-engineering and systems-engineering students from UFA-ESPE.

B. Context Selection

Our experimental object is an existing and simple web application for managing a pizzeria. This application meets the following requirements: 1) users shall create, update, and read ingredients; 2) users shall create and read pizza sizes; and 3) users shall create and read pizza orders, as well as calculate their prices. The application's design includes a frontend and a backend and makes use of the Model-View-Controller (MVC) architectural pattern [27]. The frontend includes the View components programmed as web pages, and the backend includes both the Model and Controller components, with a RESTful Application Programming Interface (REST API). The frontend is implemented using HTML, CSS and JavaScript, while the backend is implemented two different ways:

B_{pp}: A variation of the backend implemented by following the PP paradigm, using Python and Flask

B_{oop}: A variation of the backend implemented by following the OOP paradigm, using Java and Spring Boot.

Each backend variation also includes a functionally equivalent set of test cases that provide reasonably thorough coverage using path and input-domain testing techniques [28]. To create maintenance tasks, we inject specific software errors in both B_{pp} and B_{oop}. No errors are injected into the frontend.

The subjects for this experiment are as follows: 32 third semester software-engineering students from two "Object-Oriented Programming" courses, 40 fifth semester systems-engineering students from two "Advanced Programming" courses, and 18 eighth semester systems-engineering students from a "Technical English" course.

All subjects participated voluntarily and had enough programming and software engineering knowledge to understand the program used in the experiments i.e., the web application. According to Genero's suggestion [26], the subjects were told that their performance during the experience would not be evaluated, but their knowledge of related lecture topics would be covered in a midterm exam. Therefore, threats to validity regarding fear of evaluation were lessened. The related lecture topics involved: training on programming paradigms, pertinent development tools and programming languages, good software-engineering practices, databases, JSON, HTTP, and REST.

Working with students presents several advantages [29]. For example, each group of students should have a relatively homogenous knowledge base, since their training and backgrounds are generally similar. Also, it is easier to monitor the experiment in an academic setting.

Due to the academic curriculum at UFA-ESPE, it is important to clarify that subjects from upper-division level classes are more likely to be biased towards OOP and Java.

C. Variable Selection

The independent variable is the selection of the programming paradigm, namely PP or OOP. The dependent variable is the maintainability measured using two metrics: *Maintenance Effectiveness* (ME) and *Maintenance Speed* (MS). To define these metrics formally, it is first necessary to define three concepts: maintenance tasks, test cases related to maintenance tasks, and correct or successful test cases for a version of the program.

A maintenance task, t, includes the description of a functional requirement of the system, its expected operation, and its current malfunction. In other words, t relates to a software error that the developer must fix. A suite of maintenance tasks T is a collection of t_i, where $0<i\leq n$ and n is the number of maintenance tasks. For the observational study to provide the best possible comparison between the two paradigms, each $t \in T$ must deal with a different kind of software error.

For every maintenance task, t, there is a suite of test cases, Pu, that tests all the code affected by the software error associated with t, directly or indirectly. For an initial version of B_{pp} or B_{oop} and a given maintenance task t_i, some of the test cases in the associated Pu_i will fail while others will succeed. Those that fail are direct evidence of the software error associated with the maintenance task, t_i. If t_i is performed correctly, then all test cases in Pu_i should succeed. If t_i is performed incorrectly, any test case in Pu_i may fail, including those that originally succeeded. In other words, the test cases in Pu_i ensure that a subject makes an appropriate fix for the software error associated with t_i, without introducing other errors.

For any task t_i performed on either B_{pp} or B_{oop}, we will denote a set of the test cases Cu_i, where $Cu_i \subseteq Pu_i$, as those test cases related to t_i that execute correctly.

Equation (1) defines the ME metric, which is a representation of a developer's ability to perform a set T of maintenance tasks correctly. This metric is defined in the context of version of either a procedural or an object-oriented program, e.g. B_{pp} or B_{oop}.

$$ME = \frac{\sum_{i=1}^{n} Tc_i\left(\frac{Cu_i}{Pu_i}\right)}{n} \quad (1)$$

Tc_i represents whether a maintenance task, t_i, was completed (1) or not (0) , Cu_i represents the number of correct unit tests associated with t_i, and Pu_i is the number of unit tests associated with task t_i. We compute $\frac{Cu_i}{Pu_i}$ to measure the correctness ratio for t_i. In other words, ME is a weighted percentage of completed maintenance tasks that outputs a positive value between zero and one. If it is zero, the subject was not able to complete any of the maintenance tasks correctly, and if it is one, the subject was able to complete all maintenance tasks correctly.

Equation (2) defines the MS metric, which represents how fast a developer is able to perform a set of maintenance tasks correctly, for either B_{pp} or B_{oop}.

$$MS = \frac{\sum_{i=1}^{n} Tc_i\left(\frac{Cu_i}{Pu_i}\right)}{h} \quad (2)$$

Tc_i, Cu_i, and Pu_i represent the same as in (1). Also, h is the total time used to complete maintenance tasks, measured in hours. The MS metric outputs a decimal value greater than or equal to zero, measured in tasks per hour.

ME and MS are evaluated for each subject and each type of backend. Aggregations of the ME metrics for the two paradigms across all subjects are represented as ME_{pp} and ME_{oop}. Similarly, aggregations of the MS metrics are represented as MS_{pp} and MS_{oop}.

D. Hypotheses Formulation

For this experiment, we have formulated and tested the hypotheses listed in Table I.

TABLE I. NULL AND ALTERNATIVE HYPOTHESIS

Null hypotheses	Alternative hypotheses
$H_{0,1}: ME_{oop} = ME_{pp}$	$H_{1,1,1}: ME_{oop} \neq ME_{pp}$ $H_{1,1,2}: ME_{oop} > ME_{pp}$ $H_{1,1,3}: ME_{oop} < ME_{pp}$
$H_{0,2}: MS_{oop} = MS_{pp}$	$H_{1,2,1}: MS_{oop} \neq MS_{pp}$ $H_{1,2,2}: MS_{oop} > MS_{pp}$ $H_{1,2,3}: MS_{oop} < MS_{pp}$

The goal of our statistical analysis is to reject the null hypotheses and possibly accept the alternative ones.

E. Experimental Design

For this experiment, according to the values of the independent variable, we defined the following treatments:

- T1: Execute for B_{pp} tasks specified in the maintenance document.
- T2: Execute for B_{oop} tasks specified in the maintenance document.

For the subjects from Object-Oriented Programming (OOP) and Advanced Programming (AP) courses, we used intra-subject design [30]. The subjects from each class were divided into two subgroups of approximately the same size. The ones from OOP are labeled OOP1 and OOP2. Similarly, the ones from AP are labeled AP1 and AP2. We assigned each subgroup a different treatment order and both subgroups from each class performed both treatments.

The experiment involving the subjects in the "Technical English" (TE) group used inter-subject [30] design meaning that the group was randomly separated into two subgroups, TE1 and TE2, and each subgroup performed a different treatment.

Every subject executed the same set of maintenance tasks, but in a different order to minimize the effect that order may have on the experiment [26]. Table II shows the distribution of subjects and treatments (*subgroup – treatment*).

TABLE II. DISTRIBUTION OF SUBJECTS AND TREATMENTS FOR THE EXPERIMENT

Day 1	Day 2	Day 3	Day 4
	OOP1 – T1	OOP1 – T2	TE1 – T1
	OOP2 – T2	OOP2 – T1	TE2 – T2
AP1 – T1	AP1 – T2		
AP2 – T2	AP2 – T1		

F. Instrumentation

For our research, we developed the following experimental objects: a frontend and two variations of the same backend using PP and OOP, namely B_{pp} and B_{oop}. For each backend, we wrote a suite of unit tests, which allowed for the checking of the correctness of each maintainability task and the calculating of $\left(\frac{Cu_i}{Pu_i}\right)$ factor for each subject's version of a backend.

Next, we defined five maintenance tasks: four corrective maintenance tasks and one adaptive maintenance task, each representing a different kind of software error, including functionality, communication, and missing command errors. These software errors were then introduced into B_{pp} and B_{oop}.

Then, we created a maintenance document with the details about the maintenance tasks, where each task description contained the following: task name, maintenance type, requirement, error description, steps to reproduce the error using the frontend and the web services, correct output, actual output, and estimated time for maintenance.

For each subject group (i.e., course), we configured a GitHub repository with the maintenance document and the source code for the frontend and each backend.

Finally, we designed three surveys: pre training, post training, and post experiment, with test and Likert type questions to measure the level of agreement that the subjects have with certain affirmations. The objective of these surveys is to gather information about initial knowledge, training perception, task difficulty, and experiment perception.

The instrumentation is detailed in [31] and are available in GitHub repositories[1] [2] [3].

G. Experiment Preparation

Two weeks before the execution of the experiment, we conducted training sessions with each group of students, according to schedules in Tables III and IV. The training session had a time distribution according to each group's initial knowledge level and the schedule of its corresponding course. Before performing the training session, we asked the participants to complete the pre-training survey.

In addition, to calibrate the total time required to perform all maintenance tasks and fix any issue with the materials, we conducted a pilot experiment with students, who were not

[1] http://github.com/Wason1797/PROC-OOP-Experiment-OOP

[2] http://github.com/Wason1797/PROC-OOP-Experiment-PA

[3] http://github.com/Wason1797/PROC-OOP-Experiment-IT

TABLE III. DISTRIBUTION OF SUBJECTS AND TREATMENTS FOR TRAINING, WEEK 1.

Day 1	Day 2	Day 3	Day 4	Day 5
OOP1 – T1	OOP1 – T1	OOP1 – T1	OOP1 – T1	OOP1 – T1
OOP2 – T2	OOP2 – T2	OOP2 – T2	OOP2 – T2	OOP2 – T2
AP1 – T1		AP1 – T1		TE1 – T2
AP2 – T2		AP2 – T2		TE1 – T2

TABLE IV. DISTRIBUTION OF SUBJECTS AND TREATMENTS FOR TRAINING, WEEK 2.

Day 1	Day 2	Day 3	Day 4	Day 5
OOP1 – T2	OOP1 – T2	OOP1 – T2	OOP1 – T2	OOP1 – T2
OOP2 – T1	OOP2 – T1	OOP2 – T1	OOP2 – T1	OOP2 – T1
AP1 – T2		AP1 – T2		TE1 – T1
AP2 – T1		AP2 – T1		TE1 – T1

subjects but had approximately the same knowledge and skills as the subjects. Data from this plot are not included in the results reported here.

H. Execution

We started the experiment according to subjects' distribution detailed in Table II. The subjects were first given the necessary training and resources to perform the maintenance tasks, i.e., access to their group's GitHub repository. Every subject received the following instructions: 1) run the web application locally on an assigned development machine; 2) record the start time and begin working on a maintenance task according to its description; and 3) when a task is completed, inform one of the instructors of the start and end times for that maintenance task. Once the allotted time comes to an end, the subjects uploaded their changes to their own individual branches in group's repository.

To avoid possible bias in the results, the hypotheses under study were not reported to the subjects. The data was collected in a spread sheet so we can compute ME (1) and MS (2) for analysis with SPSS (statistical software). Incorrect tasks were not scored negatively. In case of detecting any abnormal behavior or unexpected situation involving a subject, we excluded all data from that subject from the final database.

IV. RESULTS

In this section, we present an analysis of the ME and MS data, and test the null hypotheses. In addition, we attempt to measure the effects of training and experience through an analysis of the surveyed data.

A. Descriptive Study

We studied the descriptive statistical data of ME and MS measures as follows: a) for both values of the independent variable, b) for the entire sample, and c) grouped by gender, age segment, course and treatment order.

Table V shows that subjects performing maintenance on B_{pp} were, on average, 8.33% more effective and approximately 1 task per hour faster than those performing maintenance on B_{oop}.

Table VI shows that female subject's average is higher for MS and ME in both paradigms than that of male subjects. Furthermore, both male and female subjects obtained higher

TABLE V. GENERAL DESCRIPTIVE STATISTICS

	N	Minimum	Maximum	Mean	Standard Deviation
ME_{pp}	81	0.00	1.00	***0.8228***	0.27444
ME_{oop}	81	0.00	1.00	0.7395	0.29397
MS_{pp} (task/hour)	81	0.00	12.86	***3.9191***	2.51813
MS_{oop} (task/hour)	81	0.00	10.00	2.8247	2.15826

TABLE VI. DESCRIPTIVE STATISTICS BY GENDER

Gender		ME_{oop}	ME_{pp}	MS_{oop} (task/hour)	MS_{pp} (task/hour)
Male	Mean	0.7261	0.814	2.7415	3.8384
	N	69	71	69	71
	Standard Deviation	0.2908	0.284	2.13229	2.51098
	Minimum	0.00	0.00	0.00	0.00
	Maximum	1.00	1.00	10.00	12.86
	Variance	0.085	0.081	4.547	6.305
Female	Mean	0.8167	0.885	3.3033	4.4925
	N	12	10	12	10
	Standard Deviation	0.3128	0.188	2.34034	2.62844
	Minimum	0.20	0.40	0.38	0.62
	Maximum	1.00	1.00	8.33	10.56
	Variance	0.098	0.036	5.477	6.909
Total	Mean	0.7395	0.822	2.8247	3.9191
	N	81	81	81	81
	Standard Deviation	0.2939	0.274	2.15826	2.51813
	Minimum	0.00	0.00	0.00	0.00
	Maximum	1.00	1.00	10.00	12.86
	Variance	0.086	0.075	4.658	6.341

measures using PP. Fig. 1 and Fig. 2 show the average ME and MS for male and female subjects.

Table VII shows that subjects between 18 and 21 years old present the highest averages in both paradigms followed in second place by subjects between 22 and 24 years old. Also, we can see that all age segments present higher averages for PP. Fig. 3 and Fig. 4 show how each age segment behaves in respect to ME and MS.

Table VIII shows that subjects registered in the Advanced Programming course present the highest measures in both paradigms. Furthermore, all courses average 5% to 17% higher effectiveness for PP and are faster while using PP. Fig.5 and Fig.6 show each subject group's averages in maintainability effectiveness and speed.

Table IX, Fig.7, and Fig. 8 show that subjects working on B_{oop} as their first treatment increase their averages in the second treatment. This is opposite to results for the subjects who performed their first treatment on B_{pp}. Nevertheless, subjects performing maintenance in B_{oop} as their second treatment had a lower average than those with PP as their first.

B. Influence of Programming Paradigm

To test the formulated hypotheses, we analyzed the effect of the independent variable using statistical tests depending on the data distribution. Table X shows the result of the Kolmogorov-Smirnov normality test for each measure.

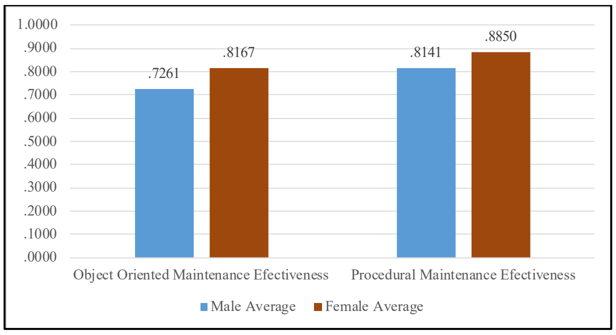

Fig. 1. Maintenance Effectiveness average by gender

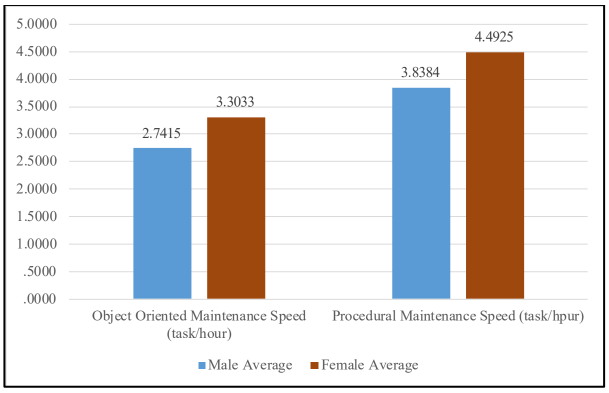

Fig. 1. Maintenance Speed average by gender

TABLE VII. DESCRIPTIVE STATISTICS BY AGE SEGMENT

Age Segment		ME_{oop}	ME_{pp}	MS_{oop} (task/ hour)	MS_{pp} (task/ hour)
18 to 21 years	Mean	0.7152	0.8182	2.7060	3.7739
	N	33	33	33	33
	Standard Deviation	0.2740	0.2671	2.28889	2.96031
	Minimum	0.20	0.00	0.35	0.00
	Maximum	1.00	1.00	10.00	12.86
22 to 24 years	Mean	0.8000	0.8234	2.9901	4.3005
	N	32	32	32	32
	Standard Deviation	0.27357	0.294	1.81876	2.41993
	Minimum	0.00	0.00	0.00	0.00
	Maximum	1.00	1.00	8.33	10.56
25 years and older	Mean	0.6500	0.8091	2.3168	3.4018
	N	10	11	10	11
	Standard Deviation	0.39791	0.30481	1.78751	1.71726
	Minimum	0.00	0.00	0.00	0.00
	Maximum	1.00	1.00	4.84	5.88
Total	Mean	0.7427	0.8191	2.7753	3.9417
	N	75	76	75	76
	Standard Deviation	0.29325	0.28047	2.02246	2.58125
	Minimum	0.00	0.00	0.00	0.00
	Maximum	1.00	1.00	10.00	12.86

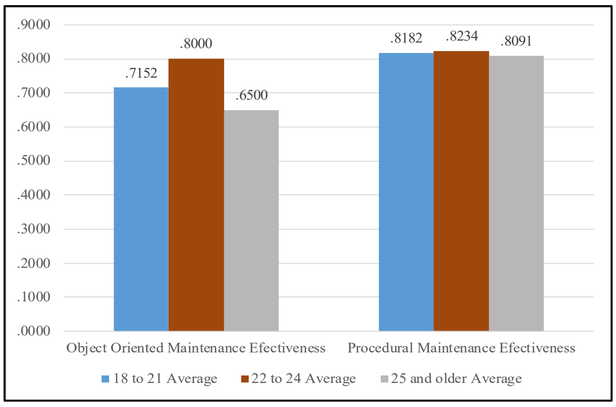

Fig. 2. Maintenance Efectiveness average by age

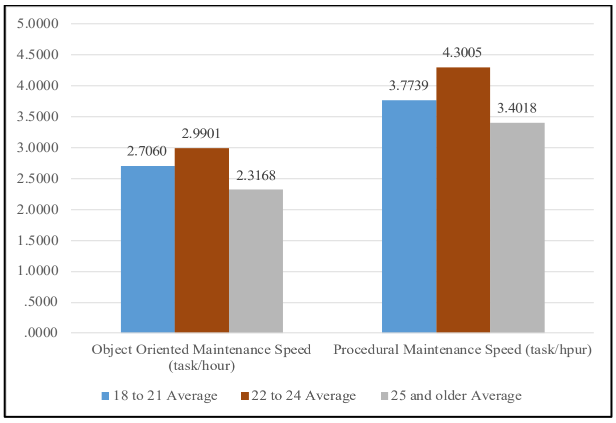

Fig. 3. Maintenance Speed average by age

TABLE VIII. DESCRIPTIVE STATISTICS BY COURSE

Course		ME_{oop}	ME_{pp}	MS_{oop} (task/ hour)	MS_{pp} (task/ hour)
Object Oriented Progra-mming	Mean	0.6500	0.7406	1.9161	2.9668
	N	32	32	32	32
	Standard Deviation	0.25400	0.30012	1.38193	2.53576
	Minimum	0.20	0.00	0.35	0.00
	Maximum	1.00	1.00	6.98	12.86
Advanced Progra-mming	Mean	0.8225	0.8788	3.4356	4.6378
	N	40	40	40	40
	Standard Deviation	0.30507	0.25290	2.11157	2.39712
	Minimum	0.00	0.00	0.00	0.00
	Maximum	1.00	1.00	10.00	12.00
Technical English	Mean	0.6889	0.8667	3.3406	4.1112
	N	9	9	9	9
	Standard Deviation	0.30185	0.21794	3.46075	2.05648
	Minimum	0.20	0.40	0.41	1.04
	Maximum	1.00	1.00	9.68	7.69
Total	Mean	0.7395	0.8228	2.8247	3.9191
	N	81	81	81	81
	Standard Deviation	0.29397	0.27444	2.15826	2.51813
	Minimum	0.00	0.00	0.00	0.00
	Maximum	1.00	1.00	10.00	12.86

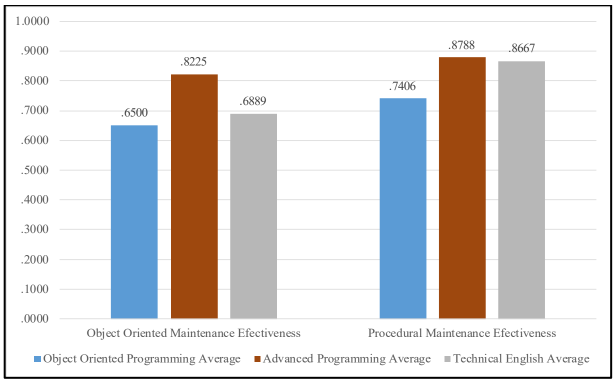

Fig. 5. Maintenance Efectiveness average by course

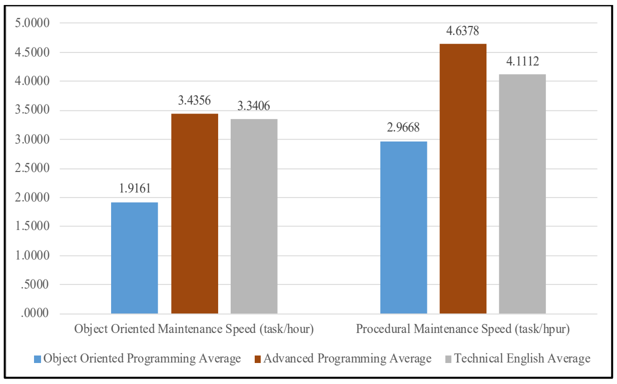

Fig. 6. Maintenance Speed average by course

TABLE IX. DESCRIPTIVE STATISTICS BY TREATMENT ORDER

Treatment order		ME_{oop}	ME_{pp}	MS_{oop} (task/hour)	MS_{pp} (task/hour)
Procedural – Object oriented	Mean	0.8086	0.8341	3.2818	3.4569
	N	35	44	35	44
	Standard Deviation	0.24299	0.27168	1.97910	1.83038
	Minimum	0.20	0.00	0.36	0.00
	Maximum	1.00	1.00	10.00	7.69
Object oriented - Procedural	Mean	0.6870	0.8095	2.4770	4.4688
	N	46	37	46	37
	Standard Deviation	0.32014	0.28083	2.24401	3.08507
	Minimum	0.00	0.00	0.00	0.00
	Maximum	1.00	1.00	9.68	12.86
Total	Mean	0.7395	0.8228	2.8247	3.9191
	N	81	81	81	81
	Standard Deviation	0.29397	0.27444	2.15826	2.51813
	Minimum	0.00	0.00	0.00	0.00
	Maximum	1.00	1.00	10.00	12.86

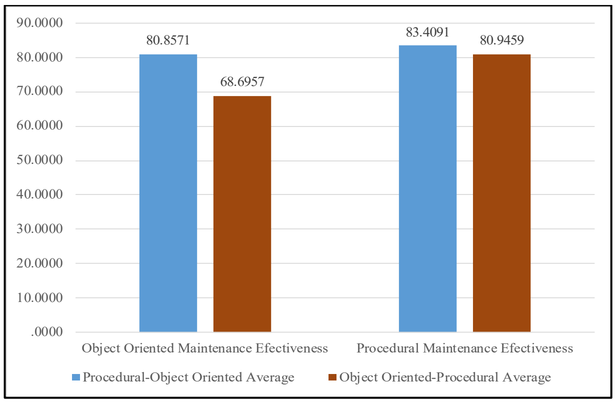

Fig. 7. Maintenance Efectiveness average by treatment order

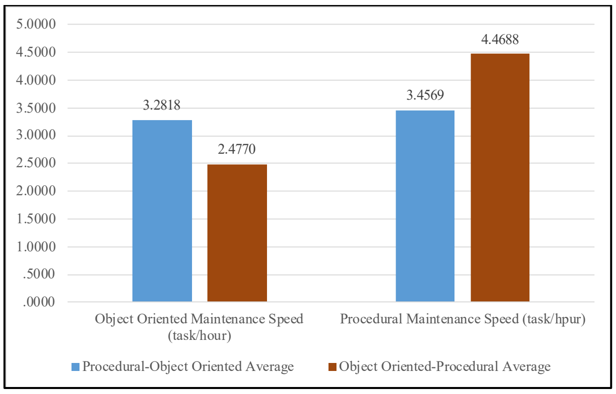

Fig. 4. Maintenance Speed average by treatment order

TABLE X. NORMALITY TEST OF A SAMPLE KOLMOGOROV-SMIRNOV

		ME_{oop}	ME_{pp}	MS_{oop} (task/hour)	MS_{pp} (task/hour)
N		81	81	81	81
Normal Parameters	Mean	0.8228	0.7395	3.9191	2.8247
	Standard Deviation	0.27444	0.29397	2.51813	2.15826
Extreme differences	Absolut	0.272	0.261	0.124	0.101
	Positive	0.259	0.188	0.124	0.093
	Negative	-0.272	-0.261	-0.060	-0.101
Kolmogorov-Smirnov Z		2.444	2.345	1.116	0.907
P-value		***0.000***	***0.000***	***0.166***	***0.383***

We used the Wilcoxon non-parametric test since the effectiveness data distribution is not normal. Table XI shows that we can reject $H_{0,1}$: $M_{oop} = ME_{pp}$, given a p-value lower than 0.05. Therefore, this initial study indicates that there is a difference in maintenance effectiveness when using PP or OOP, in favor of PP programming.

Because of data distribution for maintenance speed metric is normal (Table X), we used a T test to determine if there was a significant different in the rate for which maintenance tasks were completed for both B_{pp} and B_{oop}. Table XII shows that we can reject $H_{0,2}$: $MS_{oop} = MS_{pp}$, given a p-value lower than 0.05. Therefore, this initial study also indicates that there is a difference in maintenance speed when using PP or OOP, in favor of PP.

TABLE XI. WILCOXON STATISTICAL TEST FOR MAINTENANCE EFFECTIVENESS

	Object-oriented programming effectiveness – Procedural programming effectiveness
Z	-2.656
p-value	0.008

These results seem counterintuitive, given that most experts support the claim that OOP provides overall better maintainability characteristics [32] than PP. We will address some possible explanations in section V.

C. *Survey Results*

Figure 9 shows the average pre-training and post-training results of perceived knowledge (higher is better), grouped by gender, age segment, and course. Figure 10 shows the differences between the pre-training averages and post-training averages across the same categories.

From Figure 9 and Figure 10, we observe that on average, subjects increased perceived knowledge in all categories. Female subjects have, on average, greater values before and after training than male students. Moreover, subjects from the Object-Oriented Programming class have the highest difference between pre and post training, while subjects in Technical English have the least difference. Also, we noticed that Technical English subjects have the greatest perceived-knowledge averages before training.

V. DISCUSSION

The primary results of this observational study are twofold. First, in the context of sample population and experimental object, there is a difference in clear maintenance characteristics of PP and OOP. Second, the experimental method proved to be sufficient and could serve as a value starting point for future experimental designs. Moreover, in the process of completing this study and analyzing the data, we identified several potential enhancements that could benefit a larger, broader-reaching experiments. As such, these insights are valuable contributions of this paper.

The statistical data presented in section IV might lead one to conclude that PP has better maintainability characteristics than OOP. However, ranking the maintainability characteristics of the paradigms was not a goal of this study. Also, the apparent superiority of PP over OOP is contrary to what some experts would predict, since the aim of OOP is to manage complexity and thus improve maintainability, by

TABLE XII. PAIRED SAMPLES TEST

Paired Differences	Procedural programming speed (task/hour) – Object-oriented programming speed (task-hour)
Mean	1.13486
Standard Deviation	2.21360
Standard Error Mean	0.26088
Lower Difference	0.61469
Upper Difference	1.65503
t	4.350
df	71
p-value	0.000

encapsulating data with the operations that affect such data and by keeping those abstractions as close to real-world concepts as possible [32]. There are several reasons why this study produced the results that it did and that can provide insights into how the experimental method could be improved.

First, this study was intentionally limited to a manageable population, consisting of students at UFA-ESPE. Ideally, a larger experiment would include more subjects and subject groups that span a broader range of backgrounds and skill sets. In fact, with sufficient funding, it would be valuable to include subjects who are working in a variety of software industries, such as: web development, mobile-app development, cloud-based service, and embedded systems.

Second, the web application used as the experimental object for this study is representative of only one type of software and system architecture. Ideally, a broader-reaching experiment should either use a system that represents a wider range of software designs or multiple experimental objects that do the same.

Third, regardless of what kind of experimental objects are used, each functionally equivalent variation should be verified and validated by external reviewers, who can ensure that it adheres to the principles and best practices of the paradigm that it is implemented with, and that it makes appropriate use of that paradigm documented patterns. Although the B_{pp} and B_{oop} variations of the experimental object used in this study were tested for functional correctness and equivalence, they were not reviewed by external experts in PP and OOP to ensure that they were of equal quality with respect to their paradigms.

Forth, the set of maintenance tasks needs to be larger and more representative of common problems, including errors

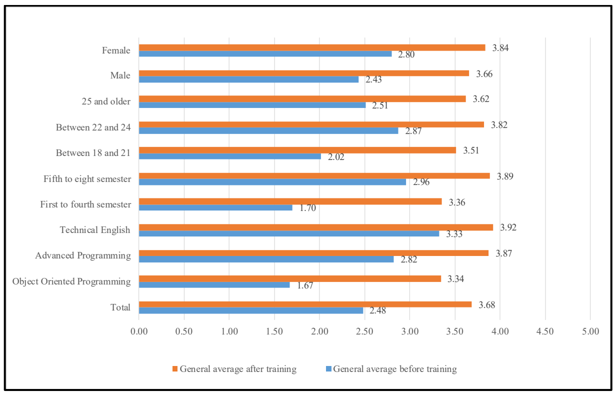

Fig. 8. General averages before and after training by gender, age, and course.

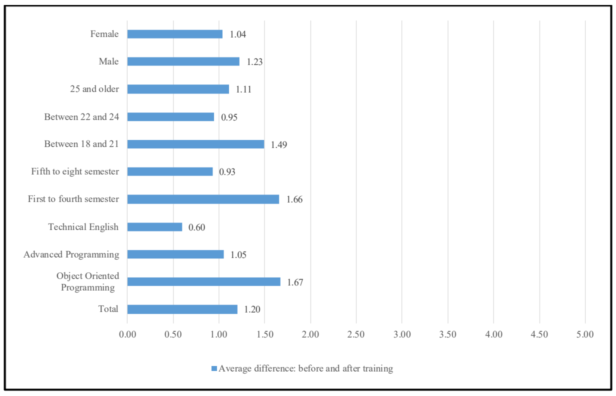

Fig. 9. Average differences before and after training

that were original made in requirement definitions, specifications, and designs. Also, the maintenance tasks need to be representative of problems whose solutions have different degrees of difficulty.

Because of the experience level of our subjects (some of them did not have any knowledge about web programming and rest APIs), our maintenance tasks were educational examples with low level difficulty. They are not representative of real maintenance tasks that might occur in actual production software. Furthermore, the proposed tasks may not reflect the strengths of each programming paradigm.

Summarizing, the subjects' background and skills, the type of software used as the experimental object, the quality of the variations of experimental object, and the nature of the maintenance task may be considered threats to the external validity. This means that we cannot generalize the results of this initial study to a broader context. Nevertheless, there is enough evidence regarding differences among maintainability characteristics of different programming paradigms, to warrant further the research in an attempt to build an accurate classification of the strengths and weaknesses of each one.

VI. Conclusions and Future Work

Software maintenance is a human-based activity, so it is important to focus on the performance of developers. In this regard, students are being trained to become future professionals in the software industry. Therefore, it is important to include in their education, tools that allow them to be valuable assets to the industry. The results of this research provide preliminary evidence that PP may still be a valuable paradigm to include in computer science education, as PP outperformed OOP with respect to ME and MS, in the context of this study.

With the results of this study, which include preliminary evidence that there is a difference in the maintainability characteristics of PP and OOP, we encourage further investigation into the maintainability characteristics of programming paradigms, and in particular with respect to how software-engineering educational programs can leverage these differences. Other important contributions include a prototype of an experimental method that can be adapted for future experiments and insights about how to strengthen those experiments with more diverse experimental objects and maintenance tasks.

With adequate funding, it would be possible to design and conduct meaningful experiments in software maintenance for broader audiences within both academic and industry contexts, for additional programming paradigms, and for different types of software. Furthermore, hypotheses like those presented here need to be tested, not only for the maintainability characteristics of a programming paradigm, but also for the whole software development lifecycle. Such research would provide extremely valuable data that could guide software-engineering education for years to come.

VII. Acknoledgement

We thank Ing. Wladymir Ladislao Brborich Peñaherrera for assistance with the statistical analysis and data processing.

References

[1] R. W. Floyd, "The paradigms of programming," *Commun. ACM*, vol. 22, no. 8, pp. 455–460, 1979.

[2] S. Xinogalos, "Using flowchart-based programming environments for simplifying programming and software engineering processes," *IEEE Glob. Eng. Educ. Conf. EDUCON*, pp. 1313–1322, 2013.

[3] R. Agarwal and A. Sinha, "Object-oriented modeling with UML: a study of developers' perceptions," *Commun. ACM*, vol. 46, no. 9, p. 248, 2003.

[4] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability - A preliminary report," *QUATIC 2007 - 6th Int. Conf. Qual. Inf. Commun. Technol.*, pp. 30–39, 2007.

[5] ISO, "ISO/IEC 25010," 2011.

[6] B. J. Cornelius, M. Munro, and D. J. Robson, "Approach to software maintenance education," *Softw. Eng. J.*, vol. 4, no. 4, pp. 233–236, 1989.

[7] S. Lai, "A Maintainability Enhancement Procedure for Reducing Agile Software Development Risk," *Int. J. Softw. Eng. Appl.*, vol. 6, no. 4, pp. 29–40, 2015.

[8] M. Pizka and F. Deißenböck, "How to effectively define and measure maintainability," *Proc. 4th Softw. Meas. Eur. Forum*, pp. 93–101, 2007.

[9] P. Norvig, *Paradigms of Artificial Intelligence Programming*. 1992.

[10] A. Alexander, *Functional Programming Simplified*. 2017.

[11] T. B. Sousa, "Dataflow Programming: Concept, Languages and Applications," *7th Dr. Symp. Informatics Eng.*, vol. 7, no. January 2012, p. 13, 2012.

[12] M. Laine, D. Shestakov, E. Litvinova, and P. Vuorimaa, "Toward unified web application development," *IT Prof.*, vol. 13, no. 5, pp. 30–36, 2011.

[13] J. & D. Tošić, Milena Vujošević, "The Role of Programming Paradigms in the First Programming Courses," *Teach. Math. 2008, Vol. XI, 2, pp. 63–83*, vol. XI, pp. 63–83, 2008.

[14] S. Frame and J. W. Coffey, "A Comparison of Functional and Imperative Programming Techniques for Mathematical Software Development," *WMSCI 2012 - 16th World Multi-Conference Syst. Cybern. Informatics, Proc.*, vol. 2, no. 2, pp. 1–4, 2012.

[15] P. Asagba and E. Ogheneovo, "A Comparative Analysis of Structured and Object-Oriented Programming Methods," *J. Appl. Sci. Environ. Manag.*, vol. 11, no. 4, 2011.

[16] S. Wiedenbeck, V. Ramalingam, S. Sarasamma, and C. L. Corritore, "Comparison of the comprehension of object-oriented and procedural programs by novice programmers," *Interact. Comput.*, vol. 11, no. 3, pp. 255–282, 1999.

[17] J. Frederick P. Brooks, "No Silver Bullet —Essence and Accident in Software Engineering," *Univ. North Carolina Chapel Hill*, 1986.

[18] C. Dony, J. Malenfant, and D. Bardou, "Classifying Prototype-based Programming Languages," *Prototype-Based Object-Oriented Program. Concepts, Lang. Appl.*, pp. 17–45, 1999.

[19] P. Van Roy, "Programming Paradigms for Dummies: What Every Programmer Should Know," *New Comput. Paradig. Comput. Music*, pp. 9–47, 2009.

[20] P. Narbel, "Functional programming at work in object-oriented programming," *J. Object Technol.*, vol. 8, no. 6, pp. 181–209, 2009.

[21] J. C. Chen and S. J. Huang, "An empirical analysis of the impact of software development problem factors on software maintainability," *J. Syst. Softw.*, vol. 82, no. 6, pp. 981–992, 2009.

[22] I. Sommerville, *Sommerville-Software Engineering*. 2009.

[23] J. Saraiva, "A roadmap for software maintainability measurement,"

Proc. - Int. Conf. Softw. Eng., pp. 1453–1455, 2013.

[24] A. Al-Badareen, M. Selamat, M. A. Jabar, J. Din, and S. Turaev, "The impact of software quality on maintenance process," *Int. J. Comput.*, vol. 5, no. 2, pp. 183–190, 2011.

[25] A. M. Fernández-Sáez, M. Genero, M. R. V. Chaudron, D. Caivano, and I. Ramos, "Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance?: A family of experiments," *Information and Software Technology*, vol. 57, no. 1. pp. 644–663, 2015.

[26] M. Genero, J. Cruz-Lemus, and M. Piattini, *Métodos de investigación en ingeniería del software*. 2014.

[27] A. Leff and J. T. Rayfield, "Web-application development using the Model/View/Controller design pattern," *Proc. - 5th IEEE Int. Enterp. Distrib. Object Comput. Conf.*, vol. 2001-Janua, no. January, pp. 118–127, 2001.

[28] R. Zhao, M. R. Lyu, and Y. Min, "A new software testing approach based on domain analysis of specifications and programs," *Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE*, vol. 2003-Janua, no. December, pp. 60–70, 2003.

[29] M. Host, B. Regnell, and C. Wohlin, "Using Students as Subjects - A Comparative Study of Students and Professionals in Lead-Time Impact Assessment," vol. 5, no. 3, 2000.

[30] S. L. Pfleeger, "Experimental design and analysis in software engineering," *Ann. Softw. Eng.*, vol. 1, no. 1, pp. 219–253, 1995.

[31] W. Brborich and B. Oscullo, "Estudio comparativo entre paradigmas de programación y su influencia en un proyecto de desarrollo de software," Universidad de las Fuerzas Armadas ESE, 2020.

[32] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, and K. Houston, *Object-Oriented Analysis and Design with Applications*, 3rd ed. Pearson Education, Inc., 2007.