Journal of Information Systems Education

Volume 31
Issue 1
Winter 2020

Experiences in Using a Multiparadigm and Multiprogramming Approach to Teach an Information Systems Course on Introduction to Programming

Juan Gutiérrez-Cárdenas

Recommended Citation: Gutiérrez-Cárdenas, J. (2020). Experiences in Using a Multiparadigm and Multiprogramming Approach to Teach an Information Systems Course on Introduction to Programming. *Journal of Information Systems Education,* 31(1), 72-82.

Article Link: http://jise.org/Volume31/n1/JISEv31n1p72.html

Initial Submission:	23 December 2018
Accepted:	24 May 2019
Abstract Posted Online:	12 September 2019
Published:	3 March 2020

ISSN: 2574-3872 (Online) 1055-3096 (Print)

Experiences in Using a Multiparadigm and Multiprogramming Approach to Teach an Information Systems Course on Introduction to Programming

Juan Gutiérrez-Cárdenas
Faculty of Engineering and Architecture
Universidad de Lima
Lima, 15023, Perú
jmgutier@ulima.edu.pe

ABSTRACT

In the current literature, there is limited evidence of the effects of teaching programming languages using two different paradigms concurrently. In this paper, we present our experience in using a multiparadigm and multiprogramming approach for an Introduction to Programming course. The multiparadigm element consisted of teaching the imperative and functional paradigms, while the multiprogramming element involved the Scheme and Python programming languages. For the multiparadigm part, the lectures were oriented to compare the similarities and differences between the functional and imperative approaches. For the multiprogramming part, we chose syntactically simple software tools that have a robust set of prebuilt functions and available libraries. After our experiments, we found that the students were strongly biased towards memorizing the syntax of these languages, jeopardizing their ability to learn to think algorithmically and logically in order to solve the given problems. We believe that teaching students using multiparadigm and multiprogramming techniques could be discouraging, especially for those students with no programming experience. In this research study, we present the results of applying this approach together with the achievements, failures, and trends of the students who were taught with this multipath system.

Keywords: IS education, Introductory programming, Multiparadigm, Multiprogramming

1. INTRODUCTION

Teaching different programming languages, irrespective of their multiparadigm features, is encouraged by the ACM Computing Curriculum (2013). This is part of a strategy to prepare students to adapt to situations that require the ability to self-teach new material and content. Wang (2002) proposed the above-mentioned approach as part of efforts to develop students' ability to self-learn, even after they have finished their undergraduate studies.

We based our proposal on the teaching of paradigms. A paradigm is a set of methods that could be effective in a certain problem domain (Bal, 1994). The programming languages are viewed only as tools for putting the students' ideas (paradigms) into practice. It is worth mentioning that following a paradigm approach will eventually help students cope with any new languages they encounter as part of the established paradigms. In addition, the paradigm approach will enable the students to solve tasks better by using the most appropriate paradigm for each specific scenario. Consequently, we aimed to give the students a broader view of what learning a language paradigm encompasses, independent of the syntaxes and add-on libraries of each programming language. Equally important, we also oriented our students to the idea that there is no programming language that is better or worse than another (ACM, 2013), but that each paradigm comprises a set of programming languages even though they use different rationale or logic. The paradigms chosen for our research were imperative and functional programming.

The imperative paradigm, with its structure of performing a set of operations over a flow of data, closely resembles a computer's internal representation of information and is one of the most widely understood paradigms (Zanev, 2011). In addition, this paradigm is better suited to the human brain's schema for performing tasks (Bal, 1994; Vujošević-Janičić and Tošić, 2008). According to Westbrook (1999), students who had previously taken an imperative approach were unable to recognize it as such in their courses. However, this is likely a direct consequence of how attention was focused solely on the software tool rather than on the underlying paradigm. With this in mind, we decided to teach an introductory programming course with a first part that was based on the imperative paradigm. Our approach focused on explaining the detail-oriented approach of the imperative paradigm and how it relates directly to the inner functioning of a computer (Bal, 1994; Zanev, 2011). In addition, we lectured on the imperative paradigm itself, independent of the programming language needed to teach its particularities.

The second part of our course was based on the functional paradigm. We taught this paradigm by succinctly describing

its features – its mathematical foundation, simplicity, and use of important features such as referential transparency (Luker, 1989) – and leading students away from the inner requirements of a computer's state flow (Vujošević-Janičić and Tošić, 2008). Equally important, we compared the characteristics of functional languages with the predominant characteristics of imperative languages, such as the side effects and instructions related to the data flow in a computer's architecture (Budd, 1995). It is worth noting that each paradigm was taught using a programming language suited for the characteristics of that paradigm, so the course taught multiple programming languages.

In addition, we found that the central problem with teaching multiple programming environments, unlike the multiparadigm approach, relates to the students' ability to cope with the syntax of each programming language. Therefore, the students found it difficult to abstract the main characteristics of the paradigms taught. During our course, students commented on and frequently complained about certain languages having a simpler syntax than others (e.g., reading or writing data from/to a text file). Nevertheless, the students understood the processes of both the functional and imperative paradigms, but they preferred the imperative paradigm due to the simpler grammar of the chosen language. Regarding this last point, we hypothesize that ease of understanding might depend heavily on the programming language selected. For example, it is generally easier for students to remember the *fopen* instruction in Python rather than Java's *BufferedReader* and *Exception* treatments, even though both programming languages perform the same task within the same paradigm. When deciding on the paradigm to use for this introductory programming course, a group of teachers voted for the imperative paradigm, while another group voted for the functional one. An idea raised by some teachers was to teach two paradigms while making comparisons of their similar features.

In light of our findings, we decided to take a multiprogramming approach in our introductory programming course, using two programming languages. Thus, we chose Python because of its simplified syntax, ease of learning, and other characteristics suitable for beginner programmers (Grandell et al., 2006; Necaise, 2008; Mason, Cooper, and de Raadt, 2012). We also chose Scheme because of its ease of use and set of prebuilt functions. Therefore, we established both programming languages as implementation tools for lectures on the imperative and functional paradigms, respectively. It is worth mentioning that we could have used Python only, as it supports both imperative and functional paradigms. However, we opted to use two languages instead of only one with multiparadigm characteristics (Budd, 1955) to provide students with an opportunity to try a language that matched their individual styles. In addition, this would allow them to fulfill the requirements of lifelong learning (after graduation) and exposure to multiple languages, as stated in ACM (2013). There was an initial concern that since this was the first programming language experience in a university context for most students, they might spend more time memorizing the syntax of each language rather than focusing on the relevant characteristics of each paradigm. Thus, we chose to use a couple of light-syntax programming languages so that we could rely on the intuitiveness of both environments, relieving the students of the need to focus on learning the syntax of the tools. For example, students did not have to learn how to use static typing or 'for-iteration' statements relying on conditionals (such as in the C-family programming languages), which cause extra learning difficulties for students (Stefik and Siebert, 2013). Consequently, beginning students exposed to these light-syntax tools would be able to focus more on the logic and algorithmic parts. The integrated development environments (IDEs) used for our lectures were Racket (for Scheme) and WinPython (for Python). Both IDEs manage errors adequately and have automatic code-filling features that often help students implement their ideas. Regarding whether the teaching of Scheme could generate student enthusiasm for computer science topics later in their careers (Berman, 1994), we hypothesize that this mostly depends on how useful the students find the concepts learned in this course, regardless of the programming language studied. As we will show, the students' perceptions changed, preferring one paradigm over another, and therefore a different programming language to solve a particular computational problem. This preference was directed more by syntactic and language features than by algorithmic or paradigmatic ones.

In Peru, there is a noticeable tendency to mostly teach languages that are industry-oriented or 'hot' in the marketplace, but we believed we should focus more on environments that are easier to teach for pedagogical reasons (Mason, Cooper, and de Raadt, 2012). For example, a study in Australia (Mason and Cooper, 2014) by a group of universities identified the change from industry needs to pedagogical benefits. Even though there is no intention to diminish the importance of programming tools such as C or Java, we wanted (as previously stated) a more suitable environment. Consequently, we needed tools that would not immerse the students in a language's requirements and characteristics, but would guide their solution implementations by focusing more on problem-solving logic than on the programming tool. In summary, our choice of Scheme and Python was due to their simple syntax and the fact that the programming does not need to be typed. These benefits have been discussed previously in the existing literature on this topic (Bloch, 2000; Heliotis, 2011; Zanev, 2011; Agarwal, 2012). Furthermore, in our course, we also tried to debunk the myth that computer science relates only to programming (Denning, Tedre, and Yongpradit, 2017), a misconception that frequently leads to the lack of student applications to programs related to computer science (CS) or information systems (IS). The inclusion of programming in IS curricula not only helps undergraduate students acquire a new set of skills needed for their careers but also (through algorithmic and computational thinking) enables IS practitioners to identify processes that can benefit from computational techniques (Topi, 2008).

Numerous approaches encourage the teaching of multiparadigm schemas and, as such, different programming languages (ACM, 2013). These range from using different paradigms, such as object-oriented programming (Striegnitz and Davis, 2008), to using two different paradigms to solve parallel problems (Pankratius, 2012). Even though we agree that these schemas should be presented to undergraduate students, we hypothesized that positive results could be equally achieved by teaching two paradigms concurrently in an introduction to programming course. Based on our results,

we believed that it would be possible to teach multiple paradigms, but that this would also depend greatly on the students' background in some topics (e.g., functional programming requires a slightly stronger background in mathematics than imperative programming). Regarding the selection of a programming language, some authors proposed using a single programming language that has multiparadigm characteristics (Budd, 1995; Agarwal, 2012). However, we believed this approach would push students to mimic the characteristics of the studied paradigm that do not specifically belong to the selected language. For example, students might be encouraged to simulate tail recursion in Python without using the full set of functions for manipulating lists that functional languages have, or without trying to make concise compose functions. In conclusion, in this research study we will determine whether the students were able to absorb the material and understand each paradigm, or whether they got stuck on the specific learning challenges presented by each programming language.

2. CLASSROOM METHODOLOGY

The course was developed as an introductory course to programming for students who will major in information systems, but who mostly have little or no background in computer programming. The expected learning outcomes of the proposed course were the following:

- By the end of the course, the student will be able to use introductory concepts in algorithm analysis to develop solutions for simple computational problems.
- To implement their solutions, the students will be able to use a programming language based on an imperative or functional paradigm.

Our course is designed to encourage the analysis of solutions to problems that can be solved using a computer and to teach simple algorithms, data structures, and programming paradigms. Additionally, we expect that our course will inspire students to be curious about different paradigms and programming approaches. This exposure to different paradigms and programming characteristics will therefore enable students to adapt to different types of programming and paradigm types that they may encounter in the future. These objectives were aimed at achieving what ACM's computer science curriculum (ACM, 2013) calls a "commitment to life-long learning."

We decided to follow the structure of the topics presented in Felleisen et al. (2001) with some subtle variations so that the course would be interesting and useful for students planning careers in information systems, but not necessarily as developers, a track considered by (Topi, 2010).

The topics we taught included the following:

- Variables and expressions
- Functions and anonymous functions (optional lecturing of lambda calculus)
- Selection
- Recursion types
- Iteration as a contraposition to recursion
- Lists
- Data search: sequential and binary search algorithms
- Divide and conquer techniques
- File manipulation
- Programming language paradigms

We taught the functional and imperative paradigm approaches using Scheme and Python. Some topics such as tail recursion were oriented to just one paradigm since it is a technique that belongs to functional languages and has no equivalent in Python. Nevertheless, focusing on a single paradigm was infrequent during our course.

We pointed out the static and dynamic natures of each paradigm (Baber, 2011), explaining the functional paradigm as describing something without saying specifically how it works. For this paradigm, we tended to define and exemplify the cases using mathematical notation. In addition, we felt that we had an advantage by teaching our course in conjunction with an Introduction to Discrete Structures course. That course helped the students to connect information from the mathematical and the programming parts of the course. The description of the imperative paradigm was related to the dynamic nature of this model, in which each instruction is defined as a command (Baber, 2011). In addition, we emphasized that one characteristic of this paradigm involves the user's knowledge of the data flow at a machine level. On the other hand, we purposely skipped some topics in this paradigm because of the nature of the language taught. For example, we omitted the static typing and pointer operations present in some imperative languages, such as C.

We covered certain functional topics, such as the use of functions and recursion, earlier than they are usually taught in imperative approaches. The use of mathematical expressions was a must. For example, when teaching how to solve a factorial we presented the students with the classic formula: $n! = n * (n-1)!$, and when presenting an example of how to create a program that sums a list of numbers we used the formula:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n-1} i + n$$

For the examples mentioned above, the imperative part consisted of how to solve both problems by using loops in Python. In general, the students responded well to this approach, but occasionally they leaned towards another approach relying on syntactic issues.

Let's look at another example used in teaching a file manipulation case. The student was presented first with a functional approach:

```
#lang scheme
(define (read file)
  (let ((port (open-input-file file)))
    (read-line port)
    (close-input-port port)
    'end))

(define (read-line port)
  (let ((lineR (read-line port)))
    (if (eof-object? lineR)
        'done
        (begin (display lineR)
               (read-line port)))))
```

This was followed by an imperative approach in Python:

```
def function():
    fileOpen=open("tale.txt","r")
    for line in fileOpen:
        lineRead=line.split()
        for w in lineRead:
            print w
```

Some students commented that the imperative version was more straightforward than the functional one but that they understood how the program behaves in its functional paradigm form. The students also felt that the Python syntax was simpler than the Scheme syntax in most of the programming situations that they encountered. To overcome this tendency to base a problem's solution on syntax only, we constantly encouraged students to look for simple ways to implement solutions based on the use of the correct paradigm and not on the particular features of a programming language.

The subjects of this experiment consisted of three undergraduate groups pursuing Information Systems degrees; each group had approximately 30 students. A different lecturer taught each group, but each strictly adhered to the same course outcomes, topics, and timelines for the material taught. After the first scored assessment, two teachers decided to withdraw from the experiment. The lecturers continued to adhere to the syllabus requirements, but only used one paradigm and one programming language. The main reason for this was that, on average, 40% of the students in each group were unable to cope with this multiparadigm and multiprogramming approach. This was reflected in the low grades achieved after the first practical test. Only one group of twenty-seven students remained a part of the experiment, and the research results described in this study reflect this proof-of-concept.

3. EVALUATION

The assessment consisted of four graded practical tests, a midterm exam, a final exam, and the presentation of a final project that incorporated the paradigms taught in the course. In the four practical tests, students were asked to solve three to five programming tasks using the concepts learned during the previous lectures. All evaluations were conducted using a computer in a lab, and we encouraged the students to submit their solutions even if the program could not be compiled. In this case, we awarded points for the logic presented in their draft solutions. The topics of the four practical tests are shown in Table 1.

Practical Test	Topics Assessed
Practical Test 1	Variables, Expressions, Data Flow: Sequentiality, Selection, Functions, Basic Algorithms
Practical Test 2	Recursion, Iteration, Introduction to Lists
Practical Test 3	Recursion in Lists: Graphs
Practical Test 4	Tail Recursion, Accumulators, Binary Search, File Manipulation

Table 1. Practical Test Topics

In some of the tests, students were asked to select and use one paradigm – imperative or functional – based on the methodology they thought would best serve the task at hand. The purpose of this was to assess the students' capacity to select and use the most appropriate paradigm and therefore the most appropriate technique for solving a specific problem (for example, using higher-order functions to exhaust a list that uses a map function instead of iterating over all the elements or being aware that iteration avoids the overhead caused by recursive functions). In other situations, students were asked to use a specific technique, such as recursion, so that we could measure the knowledge absorbed from specific parts of the course. After the midterm exam, students were generally free to choose from the paradigms taught and the available software tools, but they needed to consider the specificities of each programming language. This approach was not easy for students since most instinctively worried about syntax and why a program was not executing even though their logic seemed correct. To overcome this problem, we decided to grade their assignments more on the logic involved than on the execution of the program.

In a few of the assignments carried out in lab sessions, we found that a group of students tended to follow colleagues who were better at making their code executable even though their solutions may have been logically incorrect. This problem, also described in Bloch (2000), caused fewer logical ideas to be generated in the classroom as it fostered the feeling that a solution is right only if it executes in a computer. We addressed this situation by pointing out cases in which the computer execution of a program can lead to a wrong answer due to poor logical or algorithmic design. We also encouraged all of the students in the class to participate in the solution design phase. During the implementation part of the assignment, the teacher was more of a "syntax advisor," there to help correct coding mistakes.

Each test corresponded to a specific practice and measured the knowledge gained during different parts of the course. The following is an example of the second practical test:

Topic: Lists

1. *Consider any list, for example: [1,2,4,10,11,5]. Implement a program that prints the sum of the odd numbers in the list.*
2. *Code a solution that reverses a given list of numbers.*

Topic: Recursion vs. Iteration

1. *Imagine a triangle made up of blocks and arranged in rows; the first row has one triangle, the second row has two (making a total of three triangles), and so on. Implement a solution that will enable a user to count the total number of triangles by inputting a row number.*
2. *Assume that we have a word stored in a list; for example: [h,e,l,l,o]. Code a program that prints an asterisk (*) if it finds two identical characters next to each other. In our example, the program should print 'hel*lo'.*

In the above example, the students were required to show their knowledge of list-related topics such as basic data structures, recursion, and iteration. Regarding recursion and iteration, we wanted to measure how many students preferred iteration over recursion and to determine if recursion was indeed a more difficult technique for first-semester undergraduate students to understand. (For an analysis of how well recursion is understood by undergraduate students, refer to Lurlyn (2010)).

In addition, we informed our students that they could submit their solutions even if they did not execute as long as they believed their answers were logically and algorithmically correct. The intention was to encourage students to complete most of the tasks during the allotted time. Sometimes, teachers put programming examples in an addendum to help students grasp the syntactic constructors during a test. In this case, students had to submit a fully working version of the solution. As a way to contribute to the learning of our students, we also established the following mechanisms:

a) In addition to lectures, time was allotted in each session for students to work in pairs to solve a set of given exercises; afterward, we shared the correct answers with the classroom. In these sessions, the teacher also acted as a moderator.
b) A Moodle-based virtual education environment was used in which students could submit their questions.
c) There were also opportunities for students who demonstrated mastery of a topic to act as a teaching assistant, with time allocated to help students having difficulty in the course.

However, this last opportunity only resulted in a couple of extra sessions, since the designated students had other academic responsibilities that limited their ability to provide this mentorship.

4. RESULTS

4.1 Practical Tests

The practical tests assessed students' knowledge of the topics covered up to the test. For nearly all questions, students were free to choose the paradigm and implementation programming language. Additionally, an exam appendix was included that provided the syntax of some instructions along with small examples of code. The purpose of this aid was to assist students when they were unable to recall the syntax of the chosen language. It is worth noting that there were 27 registered students at the beginning of the course, but only 17 students completed all 4 evaluations required by this course.

4.1.1 Practical test 1. The questions in practical test 1 were related to implementing solutions for simple computational problems, such as calculating sales tax or the area of a geometric figure. The goal was to examine whether students successfully learned the concepts of modularization with functions, selection, and coding of small algorithms for solving computational problems.

The number of students who failed to attend this first examination was not high, but these students did subsequently drop the course.

4.1.2 Practical test 2. The topics in practical test 2 were related to list manipulation and the use of recursion or iteration in creating solutions for a given set of problems.

There was a dramatic drop in the number of students attending the course as 81% took practice test 1, but only 59% were present to take practice test 2. That said, the percentage of students who passed this exam was fairly high – almost 93%.

4.1.3 Practical test 3. Practical tests 3 and 4 were given after the midterm exam. Practical test 3 focused on evaluating skills in recursion and list manipulation using the functional paradigm. The first three questions were about performing operations on lists by traversing their elements. The last question required modification of a functional program that returned the number of vertices adjacent to a vertex given as the input to a function. A code sample to be used as a template was included in the test. Interestingly, we found that approximately 88% of the students who took this test were not able to interpret or modify a given program. Another noteworthy result was that 16% of the students used memorized solutions of problems studied in class. An example of one of the questions asked is the following:

Given a list with positive and negative numbers, develop a program that puts the positive numbers in one list and the negative numbers in another list. At the end, your program should return a join of both lists, for example:
>(splitLists '(1 -4 1 -6, 2 3) '() '())
(1 1 2 3 -4 -6)

The students mentioned that they tried to use a binary search program tree made in class because they noticed the similarity between (splitLists '(1 -4 1 -6, 2 3) '() '()) and the representation of a binary tree made in a functional language (search 9 '(5 (3 (1 () ()). We believe this is a common mistake made by undergraduate students and is one that we frequently see in our research. Consequently, the students strive to produce a running program instead of thinking of an algorithm that will solve the problem at hand.

In another question about reversing list elements, 27% of the students used the *reverse()* instruction even when they were prompted not to use it but were encouraged instead to develop a function that could emulate that task. In this problem, we saw the students' proclivity for using the features of a programming language which limits their logic and makes

them directly dependent on that language. In general, the most frequent errors in this test were:

4.1.4 Practical test 4. Practical test 4 had three questions: 1) traversing and performing operations on a list using iteration, recursion, or tail recursion; 2) transforming a binary search program created in functional mode into its imperative counterpart; and 3) counting the number of occurrences of a given word in a text file. For this last question, students could use any paradigm they thought best. Of the 17 students who took the test, only 35% managed to pass. At the end of the test, the students were asked to explain why they chose one paradigm and programming language over another. The results are shown in Table 2 for each question:

Trends Found	Percentage
Use of Scheme in a functional mode for traversing the list.	82%
Conversion of a given program from one paradigm to another.	35%
Read and manipulate the contents of a file successfully.	41%

Table 2. Programming/Paradigm Trends followed by Students during Practical Test 4

At this point, students reported that they were able to do recursion over iteration but that the requirements of the chosen functional programming language were cumbersome – e.g., the excessive number of parentheses (Vujošević-Janičić and Tošić, 2008; Zanev, 2011). Students manipulated the file using the imperative paradigm with Python, and the general comment was that the Scheme syntax was too complicated to remember for such a simple task. As such, we can conclude at this point that the students had a noticeable tendency to pay more attention to the syntax and extended functionality of a program. It is important to note that a student's focus on the syntax of a particular programming language will lead to an inability to cope with fundamental programming concepts (Mason and Cooper, 2012). Thus, it will be difficult for students to successfully implement the algorithmic parts, and this could lead to an increased failure rate in these introductory courses.

4.2 Midterm and Final Examinations

The midterm exam questions corresponded to topics evaluated in practical tests 1 and 2, and the final exam corresponded to topics evaluated in practical tests 3 and 4.

4.2.1 Midterm examination. This evaluation consisted of theoretical questions about general concepts and two practical programming questions related to: 1) selection and lists for calculating the wages of a group of workers under certain conditions, where the working hours of employees were provided in a list structure; and 2) solving an arithmetic series using recursion. A third question asked students to solve a very basic medical diagnosis system using only nested selection instructions. The trends we found are presented in Table 3 below.

Questions/Trends	Percentage
Question 1	
The student does not use lists when required.	68%
Use of iteration instead of lists.	9%
Question 2	
Proper use of recursion	27%
Question 3	
Use of 'cond' over 'if' instructions(*)	18%
Use of nested 'if' statements	9%
Other issues found in the mid-term exam	
Syntax issues	18%
Use of memorized patterns learned in class	4%
Use of constant values	9%
Use of Python in an imperative way	27%
Use of Scheme in most of the questions	45%

(*) 'Cond' is a type of selection instruction similar to 'if', in which a set of instructions to follow after a conditional are gathered together. Equivalents in other languages are the 'switch' or 'case' instruction.

Table 3. Programming and Paradigm Trends Found during the Midterm Examination

In this exam, students were prompted to use an imperative Python or functional Scheme. In Question 2, they were told to use recursion to solve the arithmetical summation series. Only 27% used recursion correctly, which is a poor result considering that most of the concepts using that technique had been taught (we had emphasized these concepts in the lectures on the recursion topic). We also found that 45% of the students in the class used Scheme with a functional approach, in contrast with the only 27% who attempted to use Python. The remaining percentage used a mix of imperative and functional approaches, not answering any question in a clear, algorithmic manner. Additionally, approximately 68% of the students did not remember how to extract the elements of a list for the first question so they processed the data as if the list elements had been entered one-by-one into the program. The functional way of processing a list – with the 'car' and 'cdr' instructions – along with the imperative way of extracting its elements using indexing seemed to have confused the students, which was reflected in the results obtained.

4.2.2 Final examination. The final exam had two questions for which students had to choose a paradigm to use (recursion or iteration). The topics covered were the use of selection, the use of lists as a data structure, and the design of bug-free programs. The first question involved adding or subtracting elements of a list according to a previously defined condition. The second question asked students to derive the sum of quantities sold by a seller and to eliminate any duplicate data that could appear on a list. We decreased the difficulty level of the questions and addressed only the minimum concepts students would need to know for the next course.

Table 4 shows the drawbacks we found in the final examination (for each question):

Drawback	Percentage
Choice of recursion over iteration	50%
Traversal of a list is done adequately	60%
Recursion is done adequately	46%
Use of constants	20%
Choose of imperative vs. functional paradigm	20%
Logic correct, but syntax problems found	53%
Use of unrelated functions	6%
Use of unnecessary/wordy instructions	73%

Table 4. Drawbacks and Trends Found during the Final Examination

The results revealed that 50% of the students used recursion over iteration, while only 10% chose iteration over recursion. The other 40% proposed solutions that failed to use recursion or iteration appropriately, revealing that there was confusion about the correct use of recursion and iteration. The problem of the use of constants is related to the students' inability to generalize their solutions for any input. Moreover, the students forced their solutions to work only with the data given to answer the questions on the exam. The use of additional or wordy instructions refers to things such as a student adding a zero value or trying to use additional lists for storing data. Regarding the number of students who successfully passed this exam, we can see that it is about the same as those that passed the midterm exam, but that there was a significant drop in the number of students who took the exam (81% of the class took the midterm exam while approximately 55% took the final exam). The overall results of both examinations are presented in Table 5.

	Midterm Exam	Final Exam
Students who took the exam	81%	55%
Students who passed the exam	55%	60%

Table 5. Overall Results for the Midterm and Final Examinations

At this point, we can discuss why a considerable number of students declined to use recursive patterns instead of their iterative counterparts. We hypothesize that the direct reason for this is because the functional paradigm was unclear, and more effort was required to recall the solutions and algorithms taught for solving a set of problems. We found that the students included in this study had (as we previously hypothesized) a marked tendency to focus on the syntactic issues of a programming language instead of first developing algorithmic solutions to the problems at hand.

4.3 General Results

The general results obtained are summarized in Figure 1.

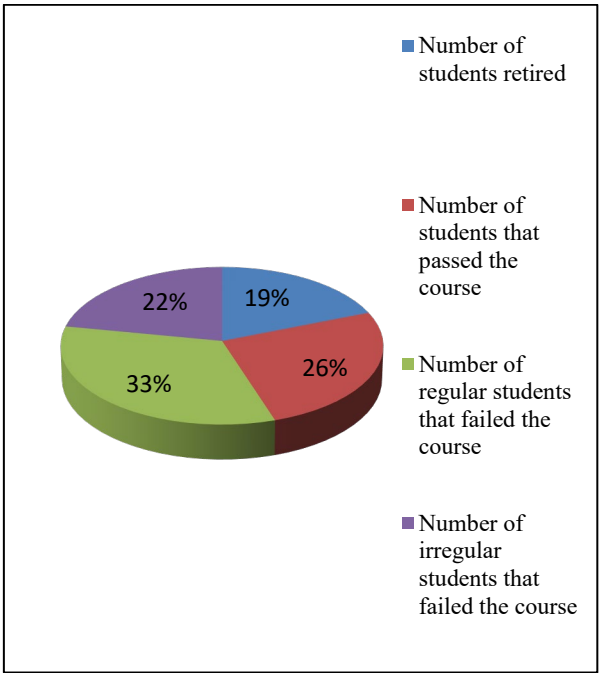


Figure 1. Percentage of Students that Passed/Failed the Course

For the general results presented above, we used what we call "raw" grades, meaning that we have not done things such as rounding up final grades or increasing grades for individual homework assignments since we discovered that these were done with the help of other students. We believe those types of situations would have biased the true grades achieved by students in the course.

Therefore, we have divided the number of students who failed the course into two categories: regular students (those who attended the course regularly with no significant absences) and irregular students (per the internal rules of our institution, those students who were absent for more than 28% of the term). Of the students who took the course more than once (25%), only one managed to pass. We hypothesize that whether a student has taken this course before has little significance because the previous course content focused on a Java-based gaming programming language. So the students who were retaking the course encountered significantly new content, such as the topics of the functional programming paradigm, data input, and file manipulation.

5. EVALUATION AND DISCUSSION

We hypothesized that with this new multiparadigm and multiprogramming approach, greater than half of the students enrolled in our course would fail due to the difficulties that we discussed earlier. We have applied a binomial test in R for the aforementioned number of students, as seen in Table 6.

H1 = More than 50% of students will fail the course	Exact Binomial Test
p-value	0.009579
Probability of success	74%

Table 6. Binomial Test

Thus, it can be seen that our alternative hypothesis is supported, enabling us to assume that using a multiprogramming and multiparadigm approach to teach beginning students could be very challenging for them.

To corroborate our assertions, we selected a parallel classroom that acted as a control group (we will call it Group B). We gave these students the same type of examinations, with the only difference that the teacher used only one paradigm and one programming tool in the course. We chose the imperative paradigm supported by the Python programming language for group B.

We evaluated both groups at the beginning of this experiment. The evaluation was aimed at measuring the students' initial knowledge of programming and algorithmic thinking. The results for both groups indicated that programming knowledge was lacking. Apart from a familiarity with very basic computing tools such as the use of a GUI operating system such as Windows, or the use of a word processor or spreadsheet program, no relevant programming skills were found among both groups. A previous study by Gutierrez and Sanders (2009) noted this same issue. In Peru, favoring the use of information technology tools in grades K-12 instead of strengthening logic or programming skills has made computer science in early education "sterile and uninteresting" (Gutiérrez and Sanders, 2009).

The structure of this initial exam is presented below:

Question 1: Describe your previous experience with programming. What tools have you used before?
Question 2: What is a paradigm in programming?
Question 3: Solve the following problem using the basic constructs of data flow (the case was chosen at random).
Question 4: Solve the following problem using functions or by modularizing your proposed solution (the case also chosen at random).

We noticed some confusion with Question 1. Approximately 75% of the students in group A answered that they knew some programming tools, but they mentioned office utilities, such as Word or Excel, which are not real programming languages. In group B, only one student mentioned previous experience with Visual Basic, but he/she was not able to recall its syntax. The other 25% of group A, the students that were retaking the course, mentioned a Java tool for programming games that they used in previous semesters, but they failed to solve the programming problems in Questions 3 and 4. In both groups, the questions about programming were left blank; the same was observed with Question 2 regarding programming paradigms.

We compared the results from both groups and created the box and whisker plot presented in Figure 2.

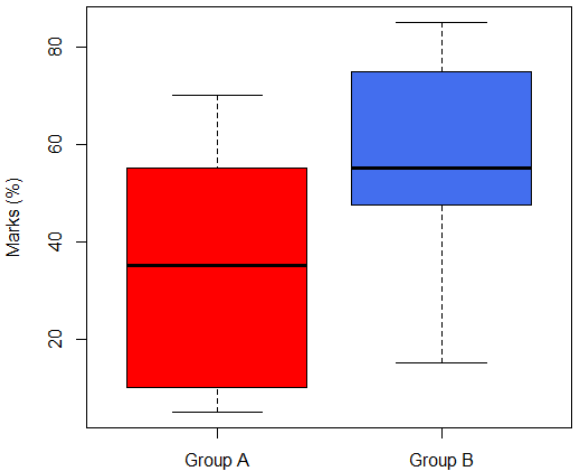


Figure 2. Box and Whisker Plot Showing the Quartile Differences between Group A (Multiparadigm Approach) and Group B (Single Paradigm Approach)

The marks in Figure 2 are in percentages; a mark above 55% indicates that the student passed the course. We can observe that the median for the multiparadigm group (red) is lower than the one for the single-paradigm group (blue). Additionally, the marks achieved by the latter group are significantly higher than the ones achieved by the students in Group A.

Using the data collected we applied a Wilcoxon test. We applied this nonparametric technique because our data did not follow a normal distribution. Our null and alternate hypotheses, followed by the results obtained in R, are presented in Table 7.

H1 = Samples come from different populations	Wilcoxon rank with sum correction
p-value	0.006177
W (sum of ranks)	122

Table 7. Wilcoxon Test

An analysis of the p-value in the above results confirms that we can reject the null hypothesis, so our data do come from different or unrelated populations. After this test, as an alternative to the Fisher exact test, we performed a Barnard test (code available at: https://www.r-statistics.com/wp-content/uploads/2010/02/Barnard.R.txt) since it is the recommended test for 2x2 tables (Mehta and Senchaudhuri, 2003). The results are presented in Table 8:

Contingency table	Multiparadigm approach	Single paradigm approach
Passed	26%	56%
Failed	74%	44%

2x2 matrix for Barnard's exact test: 100 28x19 tables were evaluated

	Barnard's Exact Test
Wald statistic	2.0084
Nuisance parameter	0.78782
p-values	1-tailed = 0.022846 2-tailed = 0.045693

Table 8. Barnard Test

Based on the results obtained from Barnard's exact test, we can conclude that there is a significant difference between the students who failed the course using the multiparadigm approach and those that failed the course using the single paradigm approach. We can also state that failure or success in this introductory course has a significant relationship with the paradigm used.

It is also worth noting that some additional factors could be the weak logical and mathematical preparation at the secondary school level, which imposes additional learning burdens for students trying to understand the concepts presented in this course. For example, functional programming requires a basic but solid mathematical foundation, while the imperative approach requires an algorithmic and detailed step-by-step logic that is not taught at all in pre-university education. This research only reinforces the findings by Mason, Cooper, and de Raadt (2012) that the learning of multiple programming languages could be a burden for novice students and those with weak backgrounds in this area. This situation could be aggravated by teaching very different programming paradigms at the same time.

Programming is a difficult task. According to du Boulay (1989), there is a set of identifiable areas of difficulty that seem relevant to our research. These include the difficulties students have with relating computer-performed operations to human-commanded actions, with the syntax and semantics of the various programming languages, and with learning the basic constructs to perform simple tasks.

A student's difficulty in relating how a computer works internally with how his program affects the computer's behavior depends heavily on the paradigm chosen. For example, the imperative paradigm follows a model closer to the Von Neumann model, in which the flow of control or change of states follows a step-by-step sequence. In contrast, a functional paradigm's method of control does not require the programmer to follow a change-of-state sequence because the level of transparency is higher. For example, consider how each paradigm deals with the use of variables. In an imperative paradigm, a variable represents a memory value and is subject to change depending on the program's current state. This characteristic allows the use of different types of variables, such as local and global ones. However, this characteristic does not apply in the functional paradigm, because a variable represents only a value, and a function (the main building block of this paradigm) will always return the same value no matter how many times it is called. This issue of the temporal value of a variable and how students have difficulties with this concept was also addressed by du Boulay (1989).

Another example would be how the changes in the values of variables (due to state changes) allow the construction of loops in the imperative paradigm, while in the functional paradigm one needs to resort to recursion to perform a similar task. These characteristics and more reveal the fundamentally opposite nature of the imperative and functional paradigms. Thus, we can conclude that if we apply a multiparadigm approach, we will oblige novice students to learn to abstract on different levels. One level must consider the changes in the internal state affecting the variables during program execution, and the other level must consider the construction of mathematical functions in which the internal state is irrelevant.

Lister (2011) performed a set of experiments to demonstrate the evolution of novice students in the field of programming. His results enable us to analyze how the teaching of multiple paradigms and programming languages could be a difficult task for beginning students. By way of background, we should discuss a difference between the Piagetian classical model and the Neo-Piagetian model. The former is solely focused on the evolution of a child's intellectual development, while the latter contends that this evolution progresses throughout a person's life (Lister, 2011). These theories can also be considered in the context of computer programming: one can determine how novice learners begin to acquire more competences and observe how they evolve through the different stages of the Neo-Piagetian theory.

The Neo-Piagetian theory recognizes three stages that can also be applied to the field of programming (Lister, 2011; Teague and Lister, 2014). The first is called the sensorimotor stage, in which the student adheres to the syntax and basic constructs of a specific language. The second is the preoperational stage, in which the student is attached to values and still is not able to see relationships between the different parts of a program. The third stage is the concrete operational level, in which the student's level of abstraction enables the development of a sound solution to a given problem (Lister, 2011; Teague and Lister, 2014).

The sensorimotor stage, in which the student is getting used to the syntax of a program, could be more demanding if exposed to two programming languages simultaneously. We observed this difficulty at the beginning of the course all the way through to the final examination. In the final exam, approximately 53% of the class was still dealing with syntactic issues even though their logic showed signs of improvement. Another typical scenario was when the students began to get comfortable with the functional language's prefix notations and parenthesized expressions and then had to recall the infix notation and syntax of its imperative counterpart.

Also during the sensorimotor stage, the student should be able to trace portions of code, but the move from one paradigm to another did not enable this capability. For example, when teaching a simple case of adding numbers in a list or an array, the student was directed to remember two different data structures and how to access its elements (i.e., indexing with loops versus recursion over lists). These

different structures resulted in students being unable to use lists and recursion in a satisfactory manner during the mid-term examination. After recognizing this problem, the students put more effort into practicing this part, which is why they tended towards the functional paradigm instead of the imperative one in the final examination. We hypothesized that this could be because recursion required more time to conceptualize than the iteration of imperative languages.

It has been noted that, during the preoperational stage, students have difficulties in relating different parts of a program (Lister, 2011; Teague and Lister, 2014). In this stage, according to these research studies, students can only focus on one specific task or construct at a time. In a multiparadigm environment, the student has to address a problem by abstracting the characteristics of a couple of solutions (imperative and functional), making the task of focusing more challenging. We can see that for a student to engage at this level of abstraction would require having the capabilities of the concrete operational stage which, with their current skills, would be almost impossible to acquire considering the current learning stage. However, according to the same study, it would be possible for a student to achieve the concrete level by the 45th week of an introductory programming course. Since the preconcrete level could be reached by the end of the 11th week, we believe that it is possible for a student to at least achieve the preconcrete level in an introductory programming course. While our course had a duration of approximately 16 weeks, the inclusion of two distinct paradigms and the size of the classroom probably had an impact on our final results.

We believed that it took longer for students to obtain an understanding of the functional paradigm's particularities. For that reason, we omitted the teaching of intermediate-level topics, such as the use of higher-order functions or the importance of lambda functions and their relationship with lambda calculus. It is worth highlighting that the extended use of multiparadigm programming tools such as Python allows the teaching of concepts from either the imperative or functional paradigms concurrently. Nevertheless, we believe that the use of multiparadigm programming tools does not permit the student to acquire a grasp of all the features of a paradigm. Even though we debated the use of Python for the imperative paradigm (due to the absence of some pure-imperative items such as memory pointers, and because we were not going to teach program aliases and memory states), we decided to keep it for its syntactic simplicity. We chose Scheme for the functional paradigm because even though it is considered to be a functional programming language, it is regarded as an impure one. Other languages considered to be pure, such as Haskell, are slightly more difficult to learn for beginner students. In addition, it is worth noting that we had a course on Discrete Structures running concurrently with this first programming course. Therefore, some concepts already acquired by the students, such as functions and function composition, supported the students in understanding the ideas behind functional programming. We firmly believe that the discrete math part must be interwoven with programming courses when there are related topics, such as when teaching a subject such as functional programming.

6. CONCLUSIONS

We applied a multiparadigm (imperative and functional) and multilanguage (Python and Scheme) approach to our Introduction to Programming course for a group of students pursuing information systems degrees. We found that the students ultimately chose a given paradigm to solve a computational problem instead of using the most appropriate paradigm. The students in our study had a marked tendency to memorize the syntax of the given programming language instead of developing a logical solution. We believe a multiparadigm approach for beginner students in a computer science field such as information systems could be a heavy burden given that they enroll in these courses with little or no programming background. This conclusion is supported by the low percentage of students who passed the course compared to the numbers that failed or withdrew from the course. Therefore, we recommend that perhaps using a multiparadigm approach with only one programming language (that supports multiple paradigms) could produce better results which would leave other languages to be approached in future courses. This may better foster an algorithmic way of thinking in students and lead them away from focusing on the syntax of a particular software tool. The increased importance that is currently placed on multiparadigm programming tools and to impure programming languages has raised the relevant discussion of whether students today can actually recognize the differences between dissimilar paradigms. We believe that it would be an interesting future topic of research to identify how software tools that support mixed paradigms and that are almost fully transparent can impact the learning of new paradigms that could emerge in the area of programming languages.

7. REFERENCES

ACM/IEEE-CS Joint Task Force on Computing Curricula. (2013). *Computer Science Curricula 2013*. ACM Press and IEEE Computer Society Press.

Agarwal, K., Agarwal, A., & Fife, L. (2012). Python and Visual Logic©: A Good Combination for CS0. *Journal of Computing Sciences in Colleges*, 27(4), 22-27.

Baber, R. (2011). *The Language of Mathematics*. Hoboken, NJ: Wiley.

Bal, H. & Grune, D. (1994). *Programming Language Essentials (1st ed.)*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Berman, A. (1994). Does Scheme Enhance an Introductory Programming Course? Some Preliminary Empirical Results. *SIGPLAN Notices*, 29(2), 44-48.

Bloch, S. (2000). Scheme and Java in the First Year. *Journal of Computing Sciences in Colleges*, 15(5), 157-165.

Budd, T. & Pandey, R. (1995). Never Mind the Paradigm, What about Multiparadigm Languages? *SIGCSE Bulletin*, 27(2), 25-30.

du Boulay, B. (1989). Some Difficulties of Learning to Program. In E. Soloway & J. C. Spohrer (Eds.). *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum, 283–299.

Denning, P. J., Tedre, M., & Yongpradit, P. (2017). Misconceptions about Computer Science. *Communincations of the ACM*, 60(3), 31-33.

Felleisen, M., Findler, R., Flatt, M., & Krishnamurthi, S. (2001). *How to Design Programs: An Introduction to Programming and Computing.* Cambridge, MA: MIT Press.

Grandell, L., Peltomäki, M., Back, R.-J., & Salakoski, T. (2006). Why Complicate Things? Introducing Programming in High School Using Python. In *Proceedings of the 8th Australasian Conference on Computing Education*, Hobart, Australia, 71-80.

Gutiérrez, J. & Sanders, I. (2009). Computer Science Education in Perú: A New Kind of Monster?. *SIGCSE Bulletin*, 41(2), 86-89.

Heliotis, J. & Richard, Z. (2011). Moving away from Programming and towards Computer Science in the CS First Year. *Journal of Computing in Small Colleges*, 26(3), 115-125.

Lister, R. (2011). Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. *Thirteenth Australasian Computing Education Conference*, Perth, Australia, 9–18.

Luker, P. (1989). Never Mind the Language, What about the Paradigm? In *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education*, 252-256, ACM.

Lurlyn, T. & Sanders, I. (2010). Mental Models of Recursion: Investigating Students' Understanding of Recursion. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, 103-107, ACM.

Mason, R. & Cooper, G. (2012). Why the Bottom 10% Just Can't Do It – Mental Effort Measures and Implication for Introductory Programming Courses. In M. de Raadt and A. Carbone (eds.), *Australasian Computing Education Conference*, Melbourne, Australia, 187–196.

Mason, R. & Cooper, G. (2014). Introductory Programming Courses in Australia and New Zealand in 2013 – Trends and Reasons. In *Proceedings of the Sixteenth Australasian Computing Education Conference – Volume* 148, Darlinghurst, Australia, 139-147.

Mason, R., Cooper, G., & de Raadt, M. (2012). Trends in Introductory Programming Courses. In *Australian Universities – Languages, Environments and Pedagogy. Proceedings of the Fourteenth Australasian Computing Education Conference*, Melbourne, Australia.

Mehta, C. R. & Senchaudhuri, P. (2003). Conditional versus Unconditional Exact Tests for Comparing Two Binomials. Available at: http://www.nbi.dk/~petersen/Teaching/Stat2009/Barnard_ExactTest_TwoBinomials.pdf.

Necaise, R. D. (2008). Transitioning from Java to Python in CS2. *Journal of Computing Sciences in Colleges*, 24(2), 92-97.

Pankratius, V., Schmidt, F., & Garretón, G. (2012). Combining Functional and Imperative Programming for Multicore Software: An Empirical Study Evaluating Scala and Java. In *Proceedings of the 34th International Conference on Software Engineering*, 123-133, IEEE Press.

Stefik, A. & Siebert, S. (2013). An Empirical Investigation into Programming Language Syntax. *Transactions on Computing Education*, 13(4), Article 19.

Striegnitz, J. & Davis, K. (2008). Multiparadigm Programming in Object-Oriented Languages: Current Research. In Patrick Eugster (Ed.), *Object-Oriented Technology. ECOOP 2008 Workshop Reader*, *Lecture Notes in Computer Science, Vol. 5475*. Berlin: Springer-Verlag, 7-17.

Teague, D. & Lister, R. (2014). Programming: Reading, Writing and Reversing. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 285-290, ACM.

Topi, H. (2008). IS Education: The Role of Programming in Undergraduate IS Programs. *SIGCSE Bulletin*, 40(4), 15-16.

Topi, H., Valacich, J. S., Wright, R. T., Kaiser, K., Nunamker, J. F., Jr., Sipior, J. C., & de Vreede, G. (2010). IS 2010: Curriculum Guidelines for Undergraduate Degree Programs in Information Systems. *Communications of the Association for Information Systems*, (26)1, Article 18.

Wang, S. (2002). An Approach to Teaching Multiple Computer Languages. *Journal of Information Systems Education*, 12(4), 201-211.

Westbrook, D. S. (1999). A Multiparadigm Language Approach to Teaching Principles of Programming Languages. In *The 29th Annual Frontiers in Education Conference*, 10-13.

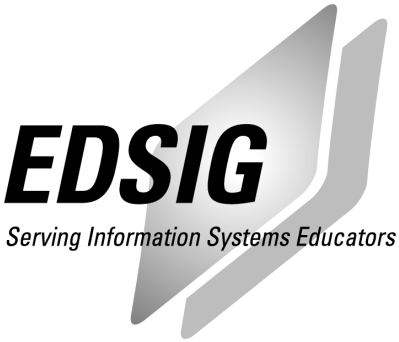
Vujošević-Janičić, M. & Tošić, D. (2008). The Role of Programming Paradigms in the First Programming Courses. *The Teaching of Mathematics*, XI(2), 63–83.

Zanev, V. A. (2011). Two-Language, Two-Paradigm Introductory Computing Curriculum Model and Its Implementation. *Serdica Journal of Computing*, 5, 129-152.

AUTHOR BIOGRAPHY

Juan Gutiérrez-Cárdenas is an assistant professor at the faculty of engineering and architecture of the Universidad de Lima. He has an M.Sc. in bioinformatics from the University of Helsinki and a Ph.D. from the University of South Africa. His research interests include machine learning applied to bioinformatics, computer science education, and computer security. Over the past 10 years, he has been actively involved in standardization, curricular guidelines, and accreditation of computer science and related careers from different universities in Perú.

Information Systems & Computing
Academic Professionals



STATEMENT OF PEER REVIEW INTEGRITY

All papers published in the Journal of Information Systems Education have undergone rigorous peer review. This includes an initial editor screening and double-blind refereeing by three or more expert referees.

ISSN 2574-3872