AMERICAN JOURNAL OF

EDUCATION AND TECHNOLOGY (AJET)

ISSN: 2832-9481 (ONLINE)

VOLUME 1 ISSUE 2 (2022)

Indexed in

PUBLISHED BY: E-PALLI, DELAWARE, USA

American Journal of Education and Technologies (AJET)

Volume 1 Issue 2, Year 2022
ISSN: 2832-9481 (Online)
DOI: https://doi.org/10.54536/ajet.v1i2.612
https://journals.e-palli.com/home/index.php/ajet

Comparing Flowchart and Swim Lane Activity Diagram for Aiding Transitioning to Object-Oriented Implementation

A.Z. Umar[1*], M.M. Gumel[2], H.S. Tuge[1]

Article Information

Received: September 17, 2022

Accepted: September 28, 2022

Published: September 30, 2022

Keywords

Flowchart, Programming Pedagogy, Swim Lane Activity Diagram, Teaching OOP

ABSTRACT

Object Oriented Programming (OOP) paradigm is one of the programming styles that emerged in response to the challenge of designing complex software. However, students find it hard to conceptualize objects when they were already accustomed to non Object Oriented approach to programming. This paper hypothesizes that introducing Object Oriented (OO) notations to students during the design phase will smoothen their transition to Object Oriented Programming. To test the hypothesis, an experiment was conducted with the students of Al-Qalam University Katsina, Nigeria. The participating students were divided into two groups: (i) *Flowchart group* - representing the classical approach where flowcharts were used to design solutions. (ii) *Activity Diagram group* - which represents the control group in which swim lane activity diagram, as Object Oriented notation, was introduced to them at the design phase. Both groups were later introduced to Class Responsibility Collaborators (CRC) cards as an Object Oriented implementation model. The students were tested, four different times, on how well they converted flowcharts or activity diagrams, as the case may be, into Class Responsibility Collaborators cards, and their performances were recorded. The results were analyzed using Repeated Measure Analysis of Variance (ANOVA). Unexpectedly, the *Flowchart group* outperformed the *Activity Diagram group* but the results were not statistically significant. Similarly, there was no statistical difference between males' and females' performances.

INTRODUCTION

Computer programming is an important skill across diverse fields and disciplines and even more so for Computing Science students. As computers are becoming more pervasive, the skill is one of the necessary requirements for continuous automation of the modern world as well as for the maintenance of the already automated systems. For computing based disciplines such as Computer Science and Software Engineering, programming courses are taught at many levels - from the year of entry to the year of graduation. Despite the criticality of programming, acquiring the skills remains a challenge for many students(Mehmood *et al.*, 2020).

Although there are many reasons why students struggle with programming(Oroma *et al.*, 2012; Qian *et al.*, 2017; Sheard *et al.*, 2009) the following are the most relevant to this paper:

i. The cognitive intensity that is required to learn the low-level details of programming.

ii. Pedagogically, students have been literary dragged to it (the programming).

For the low level details inhibiting the comprehension of programming as outlined in (i) above, over the years programming languages have evolved in which the level of abstraction has been raised. In the process of abstraction evolution, many programming styles/paradigms emerged. One of the emerged paradigms is the Object Oriented Programming (OOP) paradigm developed out of the desire to manage complexity and improve productivity in software development. It is a programming style that uses real world "objects" to design computer programs. OOP reduces the conceptual gap from the design space to the implementation space(Bucci *et al.*, 2002; Evans, 2004) because the program is conceptualized as a collection of objects interacting to achieve a common goal(Hourani *et al.*, 2019). Similarly, programs developed using OOP are easier to maintain because objects are easier to trace and update (G. Antoniol *et al.*, 2001; Giulio Antoniol *et al.*, 2000; Bianchi *et al.*, 2000). Further, OOP provides additional support for code reuse through inheritance and additional support for flexibility through polymorphism (Daly *et al.*, 1996).

With OOP, professional software developers enjoy plethora of supports. For example, in testing, there exist unit testing frameworks (Daka & Fraser, 2014); in mapping objects to records in a relational database, there exist Object Relational Mapping frameworks (Torres *et al.*, 2017); in design, there exists a collection of design patterns to solve similar recurring problems (Gamma *et al.*, 1995); in source code organization, there exists guidelines and tool support for refactoring (Daughtry III & Kannampallil, 2005; Martin, 2018).

As the saying goes that a picture may be worth more than a thousand words, it was envisaged that diagraming will improve comprehension of computer programming (Smetsers-Weeda & Smetsers, 2017) as pictorial representation simultaneously raises the level of abstraction from low to high and can be used to improve over teaching pedagogy. Consequently, diagrams were also introduced to represent the design of a solution that will be implemented as a computer program. One of the diagrams used for the design is Flowchart. Similarly, one of the diagrams used for OO design is Activity Diagram and some of the notations used in documenting OO

[1] Department of Software Engineering and Cyber Security, Al-Qalam University Katsina, Nigeria
[2] Department of Computer Science and Information Technology, Al-Qalam University Katsina, Nigeria
[*] Corresponding author's e-mail: azumar@auk.edu.ng

implementation are Class Responsibility Collaborators (CRC) models. We provide the details of the flowchart, the activity diagram, and the CRC in the *Methodology section*.

Statement of the problem

In the introductory computing courses, students are expected to learn how to design solutions to computing problems using notations such as flowcharts and then subsequently learn how to translate the flowchart into an algorithm to be implemented using the procedural style of programming. Later, and in subsequent courses, students are introduced to the OOP programming style as another implementation alternative. However, since students were already accustomed to procedural implementation and knew that the approach 'works,' they struggled to conceive objects when introduced later as a viable alternative(Börstler *et al*., 2008; Cutts *et al*., 2019; Kölling, 1999). Students already learned how to design solutions to problems without the concept of objects in the scheme of things and were later required to transition to Object Oriented implementation. One of the reasons for the struggle might be that they do not have a clue about objects in the design space but have been asked to reflect them in the implementation space, which requires unlearning the procedural style of implementation.

Thus, this paper aims to compare the effect of learning flowcharts, as non OO design notation, and activity diagram (an OO based design notation) in transitioning to Object Oriented implementation.

Research hypotheses

Given the above, we envisage that the students may find the transition easier if OO-notations were also introduced to them while learning diagrammatic designs of solutions to computing problems (see Figure 1). As such, the paper hypothesized the following:

- **H1:** Introducing activity diagrams to students at the design stage will improve their ease of transition to Object Oriented implementation.
- **H2:** there is a difference between males and females in transitioning to Object Oriented design (OOD).

The rest of the paper is organized as follows: Section 2.0 presents the research works related to this paper. Section 3.0 explains the methodology adopted in this paper and begins with the clarification of the research constructs and ends with the detailed settings and execution of the study. Section 4.0 discusses the results obtained and highlights what could be threats to the validity of the research. Lastly, Section 5.0 concludes the paper.

LITERATURE REVIEW

Existing studies mostly focus on the performance of students in programming, generally, and not specifically on OOP. For instance, Olalekan *et al.,* (Akinola & Nosiru, 2014) investigated the effect of students' attitudes on ease of learning programming. They used students' attitudes such as regular attendance to lectures and interest in programming as some of the research constructs. They found that regular attendance at lectures was the most important factor, followed by the students' interest in programming. Other factors that were found to affect students' performance in learning programming were positive perception about the programming and the lecturers' attitudes toward the students. Similarly, Amnouychokanant *et al.,* (Amnouychokanant *et al*., 2021) assessed the effect of students' attitudes toward programming and its learning performance but with different sets of research constructs. Some of the research constructs used and found to be significant predictors of high-performance in programming were positive self-efficacy and creativity.

There has been a continuous search for effective techniques to teach OOP. For instance, Loksa *et al.,* (Loksa *et al*., 2016) proposed an approach aimed at teaching student problem solving skills. The approach put emphasis in creating an explicit mental model of the problem to be solved and depicting the coding as a mere translation of the mental model. Other techniques introduced include visualizing the progress in creating the solution as well as explicit support for promptings to reflect on their strategies to solve the problem. Similarly, (Bucci *et al*., 2002) reported how they worked in transitioning from the traditional imperative model to an Object Oriented model of learning programming for over ten years and concluded that teaching Object Oriented programming is not as simple or "natural" but difficult to convey to the students the advantages and methodologies associated with Object Oriented programming.

Still on the search for effective pedagogies to teach OOP,(Anfurrutia *et al*., 2017)Implemented Kolb's learning theory in visual programming environments in order to help students to become competent in Object Oriented programming. The authors analyzed the acceptance by the students as well as its effect on their motivation. Kolb's learning theory entails four cycles that learners must undergo to acquire knowledge. The cycles are: (i) carrying out a specific activity to have concrete experience (ii) reflecting on the experience from the carried out activity, (iii) conceptualizing the theoretical aspects of the activity, and (iv) applying the knowledge acquired in new scenarios or situations. The visual environments used were BlueJ and Greenfoot – another IDE for learning and teaching based on simulations or games. For the acceptance, students indicated that they would prefer using the tools even though females' responses were more negative than males'. As for motivation, the results were not as good as the authors' expected. However, the approach does not consider the problem analysis space.

In another study on the pedagogical approach to teaching OOP, (Uysal, 2012) explored the effects of 'objects-first' and 'objects-late' methods of teaching Object Oriented Programming (OOP). The author experimented with two groups of students. The scope of the course was identical for the two groups but the structure of the contents differed in sequence. Both the participants in the two groups used BlueJ IDE to eliminate the possible

effects of different instructional tools. The objects first learners used all visual functionalities of BlueJ IDE while the objects late learners started with only the text-based interfaces of BlueJ and were instructed to use the visual support only in the last lectures. The study found that the learners instructed with the objects-first method achieved higher learning outcomes.

In a similar context of smooth transitioning from process engineering to process control engineering, (Vogel-Heuser *et al.*, 2003) explored the benefit of modeling. From the experiment they had conducted with students, it turned out that the groups that previously modeled the process had advantages in describing the several process steps and structuring the Programmable Logic Controller (PLC) program.

In another exploratory study, (Ivanović *et al.*, 2015) investigated different aspects of teaching introductory courses on Object Oriented Programming at three (3) universities in different European countries. They compared different aspects and experiences from Object Oriented programming courses that were taught in the three (3) institutions. They found that all the institutions use various forms of course delivery. This indicates that a generic strategy for teaching transition to OOP has not been found yet. However, in all three (3) institutions, technology-enhanced learning tools (TEL) played a central role in the OOP courses offered. In particular, BlueJ - an integrated development environment (IDE) for learning OOP with Java language designed for educational purposes- was found to be used in all three institutions.

METHODOLOGY

Figure 1 represents the conceptual model of this paper. In the Figure, flowcharting represents the art of non Object Oriented problem solving using flowcharts while Activity diagraming represents the art of Object Oriented problem solving using activity diagrams. Ease

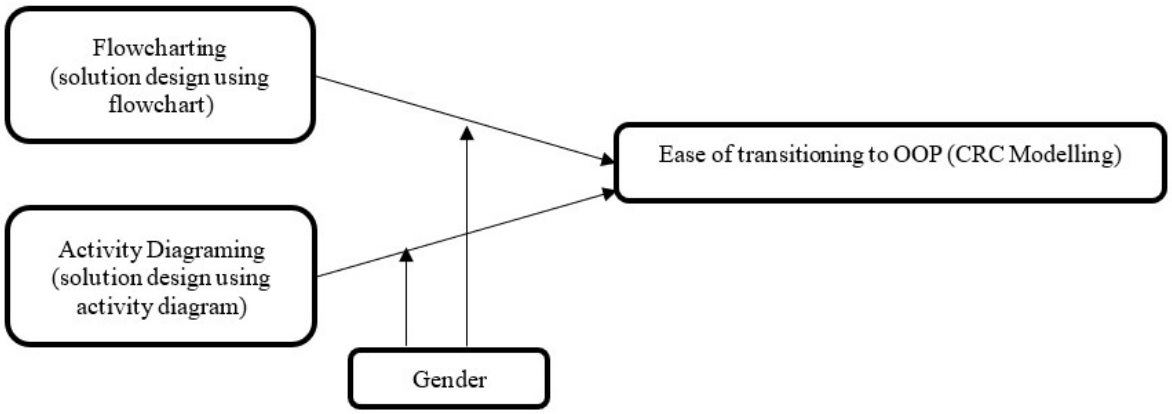

Figure 1: Conceptual model

of transition to Object Oriented programming is assessed based on how well the class responsibility collaborators (CRC) model is produced. We expect gender to play a moderating role.

To make the discussion in this section more concrete, we introduce a trivial problem of manual booking of an appointment with a dentist. In the manual process, a patient calls the dental clinic. The receptionist receives the call and guides the patients on the available slots. The patient selects one of the available slots, which the receptionist will reserve and informs the patient of the appointed schedule. For brevity, we assume that the patient has already registered with the clinic. In this section, we will discuss the background concepts with the solutions to the stated problem using a flowchart, an activity diagram, and class responsibility collaborators (CRC) cards.

Flowchart

A flowchart is a diagrammatic representation of steps to solving a given problem. Although it can also be used for other things such as the analysis of the problem or documentation of a process, we use a flowchart in this paper in the context of representing a solution to a computing problem. A flowchart is an approximate representation of an algorithm to solve a specific computing problem.

A flowchart can be used to teach problem solving without getting deep into the low-level details of complete syntaxes of a programming language so that learning can be focused on the problem solving aspect. Flowchart-based programming environments are also used to entice students to programming with the big picture of the intended solution in their minds(Chen & Morris, 2005; Smetsers-Weeda & Smetsers, 2017).

The flowchart in Figure 2 represents the design of an automated system to solve the problem of manually booking of an appointment with a dentist as outlined above. As shown in the figure, after starting the system, it displays the list of available slots for patients to request an appointment with a dentist. The patient would then select a slot and request its booking. The system then checks to confirm if the slot has not been reserved for other patients since, due to time lag, another patient might have requested and booked the selected slot. If the slot has already been taken, the system returns the patient to the

list of available slots otherwise the system will reserve the slot and send a notification email to the patient. Lastly, the booking information shall be displayed to the patient.

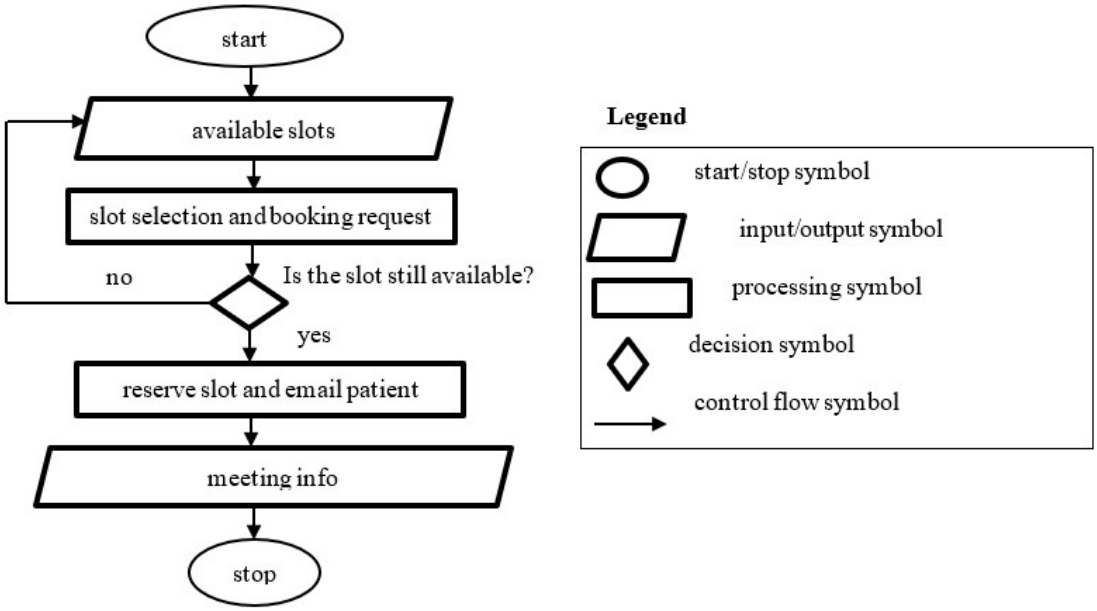

Figure 2: Flowchart illustration of a dentist booking system

It is similar to the flowchart as it can be used to diagrammatically represent a series of actions or flow of control to solve a given problem. Similarly, it can be used for other things such as modelling business processes or behavioural descriptions of a use case diagram(Jeyaraj & Sauter, 2014; Khaled AbdElazim *et al.*, 2020).

Swim lane activity is an activity diagram that is used to show which system actor is responsible for what, in addition to the representation of the series of actions or flow of control to solve the problem. Thus, the presence of objects as well as their high level responsibilities can be captured explicitly as system actors on the diagram. Swim lane activity diagrams are also powerful models in model-driven engineering (MDA) in the sense that they could also be transformed into other models or executable (Zhang *et al.*, 2012). The diagram may as well be recovered from Object Oriented source codes through reverse engineering(Martinez *et al.*, 2011).

Figure 3 is an activity diagram that represents the same solution depicted in Figure 2 as Flowchart. It is a swim lane activity diagram because the responsibilities are indicated under the System and Patient as the main actors.

Activity Diagram

The activity diagram is one of the behavioural diagrams of Unified Modelling Language (UML).

Class Responsibility Collaboration (CRC)

Class Responsibilities Collaborators (CRC) was invented by Ward Cuningham and Kent Beck(Beck & Cunningham,

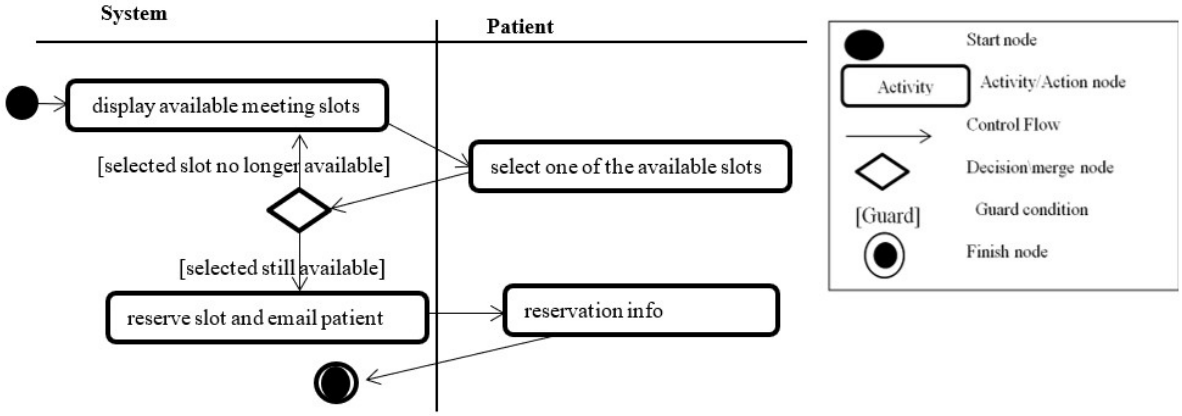

Figure 3: Swim lane activity diagram representing the design of a dentist booking system

1989; Cunningham & Beck, 1986) as an approach to discovering and documenting objects in OO design. CRC was initially designed to simplify learning OOP but has also been used in professional software development such as Agile's eXtreme Programming (XP)(Beck, 1999).

A class represents a template from which similar objects are created, responsibility is something that a class knows or does, and a collaborator is another class that a class interacts with to fulfill its responsibilities. CRC card is a 3x5 index card and is partitioned into three: the first

and the topmost portion is a row in which the name of the class being considered is represented; the second and the leftmost portion, represent the responsibilities of that class; the third portion represents the collaborators that the class will need to complete its responsibilities. Figure 4 presents the major CRC cards derivable from the swim lane activity diagram of Figure 3.

Patient	
Id	Dental Booking Manager
name	
previous appointments	
future appointments	
request for reservation	
request for cancellation	

Dental Booking Manager	
Available slots	Slot
Booked slots	BookedSlot
Process booking request	Patient
Process cancellation request	Dentist

BookedSlot	
Scheduled slot	Slot
Patient reserved for	Patient

Slot	
Slot id	Dental Booking Manager
Scheduled dentist	Dentist
Scheduled time	

Dentist	
id	
Name	
Rank	

Figure 4: CRC cards for objects implementing the dentist booking system in OOP

a class model instead but a study had found that students are more keen with CRC than the class diagram (Gray *et al*., 2003). Further, a correctly designed CRC can be implemented as an OOP program with relative ease.

Study Setting

An experiment was conducted with year one students of Al-Qalam University Katsina, Nigeria drew using a sample of convenience as only volunteers were recruited. The population of the study was Computer Science and Software Engineering students enrolled in the Introduction to Problem Solving course (module). The study recruited two groups of students: the *Flowchart group* consists of seventeen (17) students enrolled to study B.Sc. The Computer Science and *Activity Diagram group* comprises seventeen (17) students enrolled to study B.Sc. Software Engineering. All the students had taken Introduction to Computer Science in the previous semester. The distinct groups of the students were briefed about the motive of the experiment.

The students in the *Flowchart group* were taught flowcharts for a week and then preceded to learn Class Responsibility Collaboration (CRC) for another week. Students were then taught how to translate flowcharts to CRC cards for two weeks. Hence, the first group transitioned to Object Oriented implementation from procedural notations. The students in the *Activity Diagram group* were taught the swim-lane activity diagram for a week and then preceded to learn class responsibility collaboration (CRC) for another week. Students were then taught how to translate the swim lane activity diagram to CRC cards for two weeks. Therefore, the second group transitioned to Object Oriented implementation from Object Oriented design. A final-year undergraduate student taught both groups as part of her final year project.

The students were tested four (4) times within one week, each with a new problem. The reason for repeating the tests was to reduce the random noise and uncover the actual performance. In all the tests, the performances of the students were measured in terms of how well they translated the flowchart or activity diagram, as the case may be, to class responsibility collaborators (CRC) cards. That is, students in the different groups were asked the same question but with different notations to transitioning to CRC.

The experiment was treated as a 2 X 4 factorial design. Group was a between-subject factor with two levels (Flowchart vs Activity Diagram). The within-subject factor was the test which has four levels (test1, test2, test3, and test4). Data collected from the experiment were students' age, gender, and the scores of the tests.

RESULTS

In the two groups, there were a total of twenty-one (21) male students and thirteen (13) female students. However, five (5) female and three (3) male performance records were removed before the analysis because they were absent in some of the tests. Therefore, assessments of only twenty-four (24) students were analysed. Since the study was repeated four (4) times, we still had 96 data points that were subjected to the analysis.

The scores for the individual students ranged from 50 to 100 with the overall (grand) average score as 82.01. The means scores of the four different tests range from 78.85 to 84.39 but the difference was not statistically significant $F(1, 3)=1.424$, $P=0.242$. However, the means scores for all the four tests were significantly higher than the expected mean score of 50 to 55 obtained from historical

scores of computing courses (t1(25)=14.61, p=0.000; t2(25)=15.22, p=0.000; t3(25)=17.87, p=0.000; t4(25) =13.86, p=0.000).

In the first, second, and fourth tests, the means for the males were 85.61, 80.83, and 83.06 respectively. The means for females were 81.63, 74.38, and 80.63 respectively. Thus, the means for the male were higher. In contrast, in the third test, the means for females were slightly higher which was 83.13 against males' 82.22. However, there was no significant interaction between gender and the performance F(1,3) = 0.531, p=0.663.

In the first, second, and third tests, the mean for the *Flowchart group* were 86.0, 84.55, and 85.46 respectively, while for the *Activity Diagram group* the means were 83.2, 74.67, and 80.33 respectively. Thus, the means for the *Flowchart group* were higher than the expected outcome. In the fourth test, the means were comparably very close: for the *Flowchart Group* 82.27 while for the *Activity Diagram group* 82.33.

In addition to the quantitative results analyzed above, the second author also profiled students' mistakes. In the *Flowchart group*, students found it difficult to distinguish between input/output and process symbol as both were used interchangeably. They were also not bothered to put the stop symbols at the end of the flowchart. Lastly, they use directional arrows inappropriately. For the *Activity Diagram group*, Students struggle with specifying the different actors in the different columns of the diagram. They mistook the end node and the end node. Lastly, they often place a process in a column it does not belong to. Nonetheless, the students in both groups were making similar mistakes in the CRC such as skipping some of the responsibilities, interchanging the position of the responsibilities and collaborators, and not writing the class name.

DISCUSSION

It might be observed that the average score of the participants students was remarkably high considering the rate of failure in computer programming and problem solving related ourse(Omeh & Olelewe, 2021); It might be the case that only high-performing students volunteered to participate in the study. Similar to the studies in (Amnouychokanant *et al.*, 2021; Omeh & Olelewe, 2021), no significant interaction between gender and the performance. Thus, we reject the hypothesis that there is a difference between males and females in transitioning to Object Oriented design (OOD). The findings contrast with the results obtained in(Anfurrutia *et al.*, 2017) where males' and females' preferences differed significantly.

Similar to the results obtained in (VVilner *et al.*, 2007), there was no significant interaction between the different groups and the performance (F(1, 3) = 1.160, p=0.331). Consequently, we reject the hypothesis that introducing activity diagrams to students at the design stage will improve their ease of transition to Object Oriented implementation. It likely that the sample used in the study was not large enough. Thus, further study should be conducted with large and (semi)randomised samples.

Threat to Validity

It may be argued that the activity diagram may not be the right notation to test the ease of transitioning from procedural style as previous studies mostly used Integrated Development Environments (IDEs) (Anfurrutia *et al.*, 2017; Ivanović *et al.*, 2015; Uysal, 2012). However, using IDEs assumed that students already know how to design solutions before their implementation using a specific programming environment. Our study focused on problem-solving without the cumbersome aspect of learning syntaxes of any programming language. In addition, modeling using, diagrammatic notation, helps in bringing out the big picture of the intended solution(Vogel-Heuser *et al.*, 2003).

There was a chance of sampling error as the students' average scores were significantly high. Nevertheless, since the individual scores ranged from 50 to 100 and there was a total of 96 data points, any variations between the groups that were not down to chance should have been detected. However, further study with large and (semi) randomised samples may give better insight. Similarly, the results may not be generalizable as the participated students were mainly novices. Nonetheless, different backgrounds do not necessarily matter(VVilner *et al.*, 2007).

CONCLUSION

When learners were already accustomed to the procedural implementation style, they may struggle to conceive objects in object oriented programming despite the advantage of the Object Oriented style over the procedural style. This paper conceived that introducing Object Oriented modeling at the solution design phase may help ease the transition to Object Oriented programming. Thus, the paper experimented to test the effect of object oriented modeling in easing the transition to Object Oriented style. The results show that introducing the Object Oriented modeling will not achieve that desired effect as expected. The paper suggests further studies with large and randomise sample.

REFERENCES

Akinola, O. S., & Nosiru, K. A. (2014). Factors Influencing Students' Performance in Computer Programming: A fuzzy Set Operations Approach. *International Journal of Advances in Engineering & Technology, 7*(4), 1141–1149. http://www.e-ijaet.org/media/3I22-IJAET0721391_v7_iss4_1141-1149.pdf

Amnouychokanant, V., Boonlue, S., Chuathong, S., & Thamwipat, K. (2021). A Study of First-Year Students' Attitudes toward Programming in the Innovation in Educational Technology Course. *Education Research International, 2021*. https://doi.org/10.1155/2021/9105342

Anfurrutia, F. I., Alvarez, A., Larranaga, M., & Lopez-Gil, J. M. (2017). Visual Programming Environments

for Object-Oriented Programming: Acceptance and Effects on Student Motivation. *IEEE Revista Iberoamericana de Tecnologias Del Aprendizaje, 12*(3), 124–131. https://doi.org/10.1109/RITA.2017.2735478

Antoniol, G., Canfora, G., Casazza, G., & De Lucia, A. (2001). Maintaining traceability links during object-oriented software evolution. Software: *Practice and Experience, 31*(4), 331–355. https://doi.org/10.1002/SPE.374

Antoniol, Giulio, Caprile, B., Potrich, A., & Tonella, P. (2000). Design-code traceability for object-oriented systems. *Annals of Software Engineering 2000 9:1, 9*(1), 35–58. https://doi.org/10.1023/A:1018916522804

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*, addison-wesley professional.

Beck, K., & Cunningham, W. (1989). A laboratory For Teaching Object-Oriented Thinking. *ACM SIGPLAN Notices, 24*(10), 1–6. https://doi.org/10.1145/74878.74879

Bianchi, A., Fasolino, A. R., & Visaggio, G. (2000). An exploratory case study of the maintenance effectiveness of traceability models. *Proceedings - IEEE Workshop on Program Comprehension, 2000-January,* 149–158. https://doi.org/10.1109/WPC.2000.852489

Börstler, J., Nordström, M., Kallin Westin, L., Moström, J. E., & Eliasson, J. (2008). Transitioning to OOP/Java — A Never Ending Story. Lecture Notes in Computer Science *(Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 4821 LNCS,* 80–97. https://doi.org/10.1007/978-3-540-77934-6_8

Bucci, P., Heym, W., Long, T. J., & Weide, B. W. (2002). Algorithms and object-oriented programming: Bridging the gap. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education),* 302–306. https://doi.org/10.1145/563517.563459

Chen, S., & Morris, S. (2005). Iconic programming for flowcharts, java, turing, etc. *ACM SIGCSE Bulletin, 37*(3), 104–107. https://doi.org/10.1145/1151954.1067477

Cunningham, W., & Beck, K. (1986). A diagram for object-oriented programs. *ACM SIGPLAN Notices, 21*(11), 361–367. https://doi.org/10.1145/960112.28734

Cutts, Q., Barr, M., Bikanga Ada, M., Donaldson, P., Draper, S., Parkinson, J., Singer, J., & Sundin, L. (2019). Experience report: Thinkathon - Countering an "I got it working" mentality with pencil-and-paper exercises. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 203–209. https://doi.org/10.1145/3304221.3319785

Daka, E., & Fraser, G. (2014). A survey on unit testing practices and problems. *Proceedings-International Symposium on Software Reliability Engineering, ISSRE,* 201–211. https://doi.org/10.1109/ISSRE.2014.11

Daly, J., Brooks, A., Miller, J., Roper, M., & Wood, M. (1996). An empirical study evaluating depth of inheritance on the maintainability of object-oriented software. In: Empirical Studies of Programmers: Sixth Workshop. *Core.Ac.Uk.* https://core.ac.uk/download/pdf/9015758.pdf

Daughtry III, J. M., & Kannampallil, T. G. (2005). Refactoring to Patterns. In The *Journal of Object Technology, 4*,(4). https://doi.org/10.5381/jot.2005.4.4.r2

Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., Johnson, R. E., & Vlissides, J. (1995). Design patterns: elements of reusable object-oriented software. *In Pearson Deutschland GmbH.*

Gray, K. A., Guzdial, M., & Rugaber, S. (2003). Extending CRC cards into a complete design process. *ACM SIGCSE Bulletin, 35*(3), 226. https://doi.org/10.1145/961576.961582

Hourani, H., Wasmi, H., & Alrawashdeh, T. (2019). A code complexity model of object oriented programming (OOP). *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology, JEEIT 2019 - Proceedings*, 560–564. https://doi.org/10.1109/JEEIT.2019.8717448

Ivanović, M., Xinogalos, S., Pitner, T., & Savić, M. (2015). Different aspects of delivering programming courses-Multinational experiences. *ACM International Conference Proceeding Series, 02-04-September-2015*. https://doi.org/10.1145/2801081.2801085

Jeyaraj, A., & Sauter, V. L. (2014). Validation of Business Process Models Using Swimlane Diagrams. *Journal of Information Technology Management, 25*(4), 27–37.

Khaled AbdElazim, D., Moawad, R., Elfakharany, E., Widasuria Abu Bakar, N., Musa, S., & Hadi Mohamad, A. (2020). A Mini Comparative Study of Requirements Modelling Diagrams towards Swimlane: Evidence of Enterprise Resource Planning System. *Journal of Physics: Conference Series, 1529*(5), 052054. https://doi.org/10.1088/1742-6596/1529/5/052054

Kölling, M. (1999). The problem of teaching object-oriented programming. *Engineering, 11*(9), 6–12.

Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J., & Burnett, M. M. (2016). Programming, problem solving, and self-awareness: Effects of explicit guidance. *Dl.Acm.Org,* 1449–1461. https://doi.org/10.1145/2858036.2858252

Martin, F. (2018). Refactoring: improving the design of existing code. In Addison-Wesley Professional.

Martinez, L., Pereira, C., & Favre, L. (2011). Recovering Activity Diagrams from Object Oriented Code : an MDA-based Approach. *Proc. Intl. Conf. on Software Engineering Research and Practice.*

Mehmood, E., Abid, A., Farooq, M. S., & Nawaz, N. A. (2020). Curriculum, Teaching and Learning, and Assessments for Introductory Programming Course. *IEEE Access, 8*, 125961–125981. https://doi.org/10.1109/ACCESS.2020.3008321

Omeh, C. B., & Olelewe, C. J. (2021). Assessing the Effectiveness of Innovative Pedagogy and Lecture Method on Students Academic Achievement

and Retention in Computer Programming. *Education Research International,* 2021. https://doi.org/10.1155/2021/5611033

Oroma, J. O., Wanga, H. P., Ngumbuke, F., & Wanga, H. (2012). C*hallenges of teaching and learning computer programming in developing countries: lessons from Tumaini University*. https://doi.org/10.13140/2.1.3836.6407

Qian, Y., Lehman, J., Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education, 18*(1). https://doi.org/10.1145/3077618

Rentsch, T. (1982). Object oriented programming. *ACM SIGPLAN Notices, 17*(9), 51–57. https://doi.org/10.1145/947955.947961

Sheard, J., Simon, Hamilton, M., & Lönnberg, J. (2009). Analysis of research into the teaching and learning of programming. *ICER'09 - Proceedings of the 2009 ACM Workshop on International Computing Education Research,* 93–104. https://doi.org/10.1145/1584322.1584334

Smetsers-Weeda, R., & Smetsers, S. (2017). Problem solving and algorithmic development with flowcharts. *ACM International Conference Proceeding Series,* 25–34. https://doi.org/10.1145/3137065.3137080

Torres, A., Galante, R., Pimenta, M. S., & Martins, A. J. B. (2017). Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and Software Technology, 82*, 1–18. https://doi.org/10.1016/J.INFSOF.2016.09.009

Uysal, M. P. (2012). The effects of objects-first and objects-late methods on achievements of OOP learners. *Journal of Software Engineering and Applications, 5*(10), 816. https://www.scirp.org/html/23962.html?pagespeed=noscript

Vogel-Heuser, B., Friedrich, D., & Bristol, E. H. (2003). Evaluation of Modeling Notations for Basic Software Engineering in Process Control. *IECON Proceedings (Industrial Electronics Conference), 3*, 2209–2214. https://doi.org/10.1109/IECON.2003.1280586

VVilner, T., Zur, E., & Gal-Ezer, J. (2007). Fundamental concepts of CS1: procedural vs. object oriented paradigm-a case study. *ACM SIGCSE Bulletin, 39*(3), 171–175.

Zhang, W., Wang, Z., Zhao, W., Yang, Y., & Xin, X. (2012). *Generating Executable Capability Models for Requirements Validation*. https://doi.org/10.4304/jsw.7.9.2046-2052