Journal of Applied Technology and Innovation
vol. 1, no. 1, (2017), pp. 37-57

An Insight into Programming Paradigms and Their Programming Languages

M. Selvakumar Samuel
Faculty of Computing, Engineering & Technology
Asia Pacific University of Technology & Innovation
57000 Kuala Lumpur, Malaysia
dr.selvakumar@apu.edu.my

Abstract - A Programming Paradigm is the silent intelligence in any software design. Although many Programming Paradigms have evolved, only a few programming paradigms are actively used by the software industry. In addition, many hundreds of programming languages have been developed, but only a few are established and beneficial. The main aim of this paper is to provide an in-depth view into this area in order to give an opportunity for the Academia, Researchers, and the Software Industry to understand this domain in a different way. Basically, in this paper, a lot of relevant literatures have been reviewed and some useful facts, such as mainstream programming paradigms, suitable programming languages for the current software development scenario, weaknesses in the current research works in this domain, etc., have been derived as conclusions. The deduced facts would be beneficial for the education sector to decide the programming paradigms and programming languages to teach at this juncture, and as for the researchers, this paper would provide an alternative road map to conduct further research in this domain. Eventually, this work would benefit the software designers to choose appropriate programming paradigm concepts and their respective programming languages based on the deduced facts as the result of this study.

Index Terms **-** Programming paradigm; programming language design; software development

1. Introduction

A programming paradigm is the core and basis for any software and programming language design. There are many accepted definitions for the term "Programming Paradigm". According to Daniel [1], a programming paradigm is "a style of programming expressing the programmer's intent". Linda [2] said that, "it is an approach to solving programming problems" and Pamela [3] said that, "it is a way of thinking about computer systems".

In software development, there are many paradigms [4] that have been suggested and these paradigms keep evolving in order to suit the requirements of the software development of the respective times. However, only a few programming paradigms, such

as Imperative, Object-Oriented, Functional, Event Driven with Graphical User Interface (GUI), Logic, and Concurrent, are widely accepted or being used by the software industry. Amongst these few programming paradigms, the Object-Oriented programming paradigm can be considered as the dominant programming paradigm.

Each programming paradigm exclusively has its own approach, purpose, merits and demerits. Similarly, each programming paradigm has its own set of programming concepts. When designing software with the chosen programming paradigm environment, the respective programming paradigm concepts will be used. For example, when designing our solutions purely in the imperative way, Imperative programming paradigm concepts, such as control structures, input/output statements, assignment statements, etc., will be used.

A programming language is actually a collection of libraries or API's, which are based on a core programming paradigm and supports some other programming paradigms as well. For example, C++ is basically an Imperative programming language, but it supports Object-Oriented and others.

The main objective of this paper is to critically evaluate the programming paradigms adopted in programming languages with respect to the current software development context. The outcome of this paper is basically to provide an idea to the academicians to decide on the appropriate programming paradigms and programming languages to be covered based on the recent trends and current requirements. This work also attempts to provide a roadmap to the researchers working in this area and also to provide support for the software industry, especially in the stage of the software design phase of the software development life cycle.

The following contents evaluate the Programming Paradigms in terms of various works in literature whereby enabling the deduction of the relevant facts in order to achieve the above said objectives.

2. Most Influenced (Mainstream) Programming Paradigms

Every single programming language is based on one or more programming paradigms. Each programming paradigm consists of a set of programming concepts [5]. There are a huge number of programming languages, but only 27 different programming paradigms are being used [6]. Amongst these 27 programming paradigms, only a few are actively being used by the software designers.

The IEEE Spectrum [7] has ranked the programming languages based on 12 metrics across 10 reliable sources. Over 48 programming languages were analysed with a number of different dimensions. The results are summarised in Fig. 1 and Fig. 2.

Table 1 shows the list of the top programming languages and their respective programming paradigms. In this list, most of the languages are mainly realised from the programming paradigms, such as Object-Oriented, Imperative, Event-Driven with GUI, and Functional.

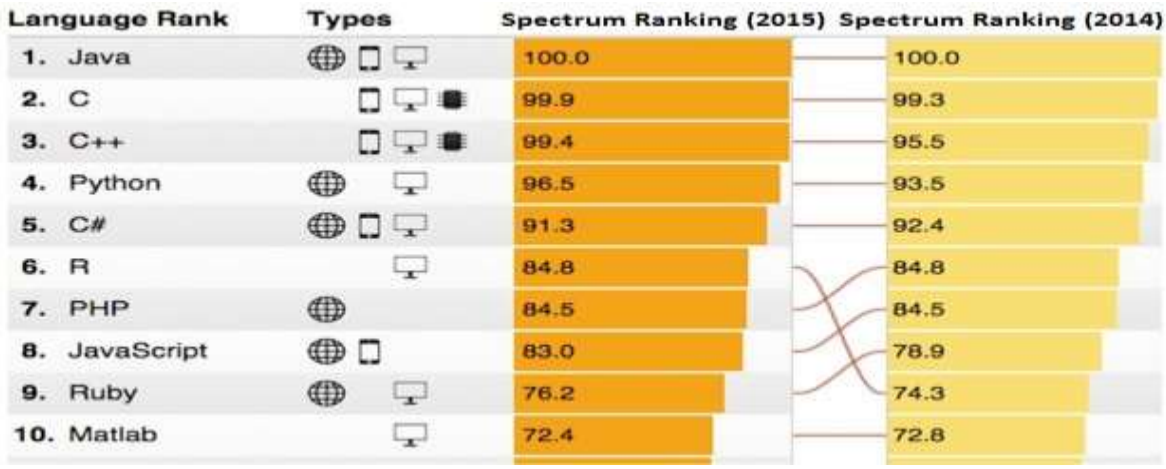

Fig. 1: Top 10 Programming Languages in 2014 and 2015 [7]

Fig. 2: Top 10 Programming Languages in 2016 [8]

Table 1: Top programming languages in 2016 and their respective programming paradigms

Number	Programming Language	Main Programming Paradigm(s)
1	Java	Object-Oriented, Imperative, Event-driven with GUI, Concurrent, Functional, Generic, & Reflection
2	C	Imperative (Procedural and Structural)
3	C++	Imperative, Object-Oriented
4	Python	Imperative, Object-Oriented, Functional, Event-Driven with GUI, Concurrent, Reflection, & Meta programming
5	C#	Object-Oriented, Imperative, Event-Driven with GUI, Functional, Concurrent, Generic, & Reflection
6	R	Functional, Object-Oriented, Event-Driven with GUI, Imperative, Reflective, & Array
7	PHP	Imperative, Object-Oriented, Event-Driven with GUI, Functional, & Reflection
8	Java Script	Scripting

9	Ruby	Functional, Object-Oriented, Imperative, Event-Driven with GUI, & Reflection
10	Go	Concurrent, Logic, Functional, & Object-Oriented.

Zuhud et al. [9] state the main programming paradigms as Functional, Imperative, Object-Oriented (OO), and Logic. Souza et al. [10] assert that, in today's current software development, the most widely used programming paradigms are Object-Oriented and Procedural. Similar to these researchers, many other researchers, academicians, books [11,12,13,14,15,16,17,18,19,20,21, &22], & literature works mainly discuss Imperative, Object-oriented, Functional, and Event-Driven with GUI programming. Event Driven with GUI is common in every current software. Particularly, every current software's interactive input and output designs are based on Event Driven with GUI approaches. Apart from this, many industrial, open source and academic software projects have been investigated. It is found that, these four mainstream programming paradigms are dominant in all software projects.

These findings fairly conclude that, these four programming paradigms are the most popular or mainstream programming paradigms in the industry, as well as in the academic domain. Amongst these, the Object-Oriented programming paradigm can be considered as the dominating paradigm; whilst the Functional programming paradigm is the emerging programming paradigm. Hence, in this research, Imperative, Object-Oriented, Functional, and Event Driven with GUI have been considered, and the research works which are related to these four programming paradigms and their concepts have been examined in the following sections.

As these four programming styles mainly dominate the software industry and the academic domain, the ideologies and the programming concepts behind these programming paradigms are obvious and very familiar. Hence, only a brief introduction of these four mainstream programming paradigms are stated as follows:

- Philip Roberts [23] states that in Imperative programming, the user instructs the computer as to what the user wants, and also instructs the computer as to how to get what the user wants. In other words, the user needs to define the details, step by step, for the computer in order to reach the goal. This type of programming is also known as algorithmic programming [24]. Procedural and Structural are the common Imperative style programming paradigms [25]. The Procedural programming paradigm is a traditional programming approach and it is the basis for the CPU's fetch-decode-execute cycle, as well. To produce the desired results, this programming paradigm has programs defined as a sequence of instructions which manipulates data to output the desired results [26].
- Object-Oriented programming is an engineering approach for building software systems which are based on concepts of classes and objects that are used for modelling the real-world entities, which changes the focus of attention from code to data. The general idea of object technology must be represented in the Object-Oriented programming languages so that complex problems can be solved in the same way as real-world situations.

- The idea of Functional programming was initially influenced by Alonzo Church's lambda calculus, which uses mathematical functions as the basic building blocks of the system [27]. Just as the name implies, a functional program is made up of functions which comprise the definition of the expressions and these expressions will be evaluated during program execution to yield the final results [28]. Hence, the main program itself is just like a single function which contains several operational computations. Programmers do not need to worry about its computational complexity as all of the complex operations will be laid in the back-end of the machine [29]. This makes functional languages more expressive.
- Event-Driven programming and graphical user interfaces (GUIs) are all interrelated [30]. The graphical interface objects or components on the forms, windows, or containers make up the view and control of an application, according to the Model-view-controller (MVC) framework [31]. Unarguably, GUI systems are Event-Driven oriented and connote that the system program does not flow sequentially from the start to the finish. "Event-Driven" literally means to be interrupted by an event or driven by an event; hence, waiting for something or an event to happen before the appointed response to that event occurs. GUI programming is among the trickiest programming paradigm shift [30].

Some other programming paradigms, such as Reflection, Concurrent, Generic, Array, Data Flow, Meta, and Constraint are also supported by these programming languages, but they do not have as rich a set as the programming concepts as the mainstream programming paradigms. Generally, these non-mainstream programming paradigms are either a technique to improve the program behaviour or a programming ability or a method to solve a specific problem, and they are always working very compatible with the mainstream programming paradigms.

In order to understand the nature of these non-mainstream programming paradigms, a brief introduction is stated as follows:

- DataFlow programming (DFP) internally represents applications as a directed graph, similar to a Dataflow diagram [32].
- Array programming mainly helps programmers in designing complex data analysis procedures [33].
- Reflection is the ability of a program to manipulate data as something representing the state of the program during its own execution.
- The kind of change that can be performed in the meta-programming approach is to modify the meta-interpreter prior to the execution of the program. Reflection, on the other hand, allows the program to change its behaviour whilst running, depending upon its current execution (such as the inputs and intermediate results) [34].
- Generic Programming is based on the principle that software can be decomposed into components which make only minimal assumptions about other components, allowing the maximum flexibility in the composition [35].

- The basic idea in constraint programming is that the user states the constraints, and a general-purpose constraint solver is used to solve them [36].
- A Concurrent program is the one defining actions that may be performed, simultaneously [37].

Amongst these non-mainstream programming paradigms, concurrent programming has relatively vast application areas.

3. Programming Paradigm Notions / Concepts

A programming paradigm is a generic way to design a program. Each programming paradigm has its own set of programming concepts. Class, Object, Inheritance, Data Hiding, etc. are called the programming notions or concepts of the Object-Oriented Programming Paradigm. In order to design a programming solution in the Object-Oriented way, the object-oriented programming concepts will be used.

Most of the concepts have their equivalents, and they produce the same results. For instance, Imperative programming iterative control structures are: for…loop, while…loop, and do…while…loop. These three control structures are alternatives to each other and they produce, generally, the same results.

Event-Driven with GUI programming is different from other programming paradigm concepts. As for the Event-Driven with GUI programming, GUI controls/elements are the key notions of event handling mechanisms in each development platform. Different elements of a graphical user interface exist on different platforms, including on modern day smartphone platforms, such as Android, IOS, and Windows. However, there is also a large overlap amongst the GUI elements that are common to all mobile operating systems (and even desktop operating systems) [38]. On a given User Interface (UI), some of these elements, generally or under certain circumstances, can be replaced with other similar elements. For example, the day of the week can be selected using a dropdown menu or radio buttons.

Application developers often apply these concepts without paying much attention to the impact they might have on the output of an application [39]. The concepts are the basic primitive elements used to construct the paradigms [40] and programming languages. In this section, some of the studies which predominantly evaluated the efficiency of the Programming Paradigms and their concepts are explored. Other related studies are discussed in the later sections.

3.1 Imperative Programming and its Concepts

The core concepts of the Imperative programming paradigm are Control Structures (looping constructs and iterations), Input /Output concepts, Error and exception handling, Procedural Abstraction, Expressions and assignment statements, and Library support for data structures [41]. Some relevant works have been identified in this area.

Oracle has studied String Builder with String Buffer as both are performing the same operations, but utilizing different volumes of resources. The result of this study

proved that String Builder consumed less resources than String Buffer [42, 43]. Likewise, they have studied ArrayList with VectorArray. They have found that the ArrayList performed better than VectorArray [44,45].

Most Imperative Programming languages support both recursion and iterations. Rubio-Sánchez [46] has said that, recursion is a key concept in computer science and mathematics, and it is a powerful problem-solving tool, which constitutes an attractive alternative to iteration, especially when problems can be solved using a divide and conquer approach. Imperative programming uses Ad-hoc recursion whilst functional programming has introduced the tail recursive approach. In languages that favour iterative looping constructs, there is usually significant time and space cost associated with the recursive programs due to the overhead required to manage the stack and the relative slowness of the function calls. Recursion and iteration are used to solve a task one at a time and finally, combine the results. Iteration emphasises repeating a task until it reaches its counter limit. Recursion emphasises breaking a larger problem into smaller pieces until the problem is solved. Looking at the efficiency point of view, recursion calls the same function over and over; whereas, iteration jumps to the beginning of the loop. The function call is normally more expensive than the jump.

Er [47] has evaluated the performance of recursive and iterative algorithms in terms of their time and space requirements. The evaluation was performed on the programs written for the famous game Towers of Hanoi. The result showed that the iterative approach outperformed the recursive one in both time and space performance. In a popular study, Schaeckeler & Shang [48] suggested that if a formal parameter or local variable is dead at all recursive calls, then it can be declared globally, so that only one instance exists independent of the call depth. The research also found that in 70% of popular recursive algorithms and in all our real-world benchmarks, it is possible to reduce the stack size by declaring formal parameters and local variables, globally. The stack size reduction starts to materialise for their benchmarks no later than in the fifth recursion on a 32-bit Intel Architecture.

Venkat Subramaniam [49] has conducted an experiment to perform a Factorial calculation on Java VM using Iteration, Recursion, and Tail Recursion with the Scala language. When computing the Factorial value of 10000 using Recursion, he got a Stack Overflow Error, but the Factorial value of 5 computed just fine. Furthermore, the iterative version of Factorial value of 10000 was able to be computed without any problem. He pointed out that the iterative version made use of a mutable local variable that the recursive solution nicely avoided. The best of both worlds could be achieved, if a recursive code using a compilation technique could transform the code and run as an iterative process.

In another study, Liu & Stoller [50] explain that transforming recursion into iteration eliminates the use of stack frames during the program execution. In a study by Schaeckeler & Shang [51], it was shown that compilers tried to reduce the code segment and neglected the stack segment; although, the stack can significantly grow during the execution of recursive functions.

Likewise, the study conducted by Chandran [52] to test the implementation of the Cyclomatic Complexity analysis feature in Coverlipse(an open-source coverage analysis tool), the researcher examined the program written in a *switch* statement and three other programs written in different constructs of *if* statements, namely, if-else statements, if statements, and an un-indented if-else statement. His research showed that these alternative styles of implementing a control flow program had resulted in different numbers of cyclomatic complexities, which meant that they had different levels of software complexity. This result further proves that both conceptual and syntactic level evaluations are feasible for performance evaluations.

Most of the researchers in the Imperative programming area are mainly focussing on the efficiency of the control structures and the recursive function call features.

3.2 Object-Oriented Programming (OOP) and its Concepts

The benefits of Object-Oriented programming include reduced complexity, object re-use, enhanced modularity, encapsulation, design benefits, and software maintenance. However, the OOP paradigm can increase power consumption and execution time, thereby hindering performance. Mattos, C.B. and Carro, L., [53] highlighted that Object-Oriented programming significantly increases dynamic memory use, thereby developing overhead in terms of memory, performance, and power. Due to various reasons, there are only a handful of documented bad experiences with object technology.

Boasson, M. [54] stated that companies are not very forthcoming with details of failed projects and secondly, it is difficult to get anything published that questions the usefulness of OOP. Researchers, such as [55, 56, 57, & 58], have demonstrated the consequences on the object-oriented system performance, in terms of execution time and memory overhead.

Kayun, C. and Chonawat, S. [59] have carried out a research which aims to find appropriate methods and recommendations of using some OOP concepts that consume less power by using real quantitative results. This research has focussed only on power consumption, but this is the only work targeted to identify the optimised concepts of Object-Oriented programming paradigms.

Several researchers have reported the issues and other concerns on the Object-Oriented programming paradigms, but very few works have been found related to the Object-Oriented programming notions. Some of the OOP concepts have alternative choices which can be used instead of the others, in some cases. Due to the differences in these concepts, some are lighter than others in terms of energy consumption. As such, developers need to choose the lightest and most appropriate concepts in a power conscious software development when there is an alternative.

3.3 Functional Programming and its Concepts

Some notable research works have been undertaken on Functional Programming and its concepts in terms of resource utilisation. Albert et al. [60] presented a framework to

define the upper boundary of the application's memory requirements so that the execution would not exceed the predefined memory limit. Their work was targeted at garbage collected languages which use mutable data, such as the Imperative and Object-Oriented programming languages. Meanwhile, Simoes et al. [61] have introduced a dynamic memory allocation framework that is able to determine the upper bounds of the memory requirements for lazily-evaluated higher-order functional programs.

Another resource utilisation approach was presented by Antoy & Jost [62], which is known as the target implementation design that aims at functional logic programs, such as Curry. These researchers implemented this design on a prototype known as Sprite that can perform functional computation in an effective manner. They, subsequently, compared the execution of this prototype with the Glasgow Haskell Compiler (GHC), and the results have indicated that this design is comparatively more efficient than GHC in terms of memory management and execution time.

Minutolo et al. [63] have proposed a lazy-evaluated pattern-matching algorithm to handle computational resources. They tested the algorithm in mobile devices, and the result proved that it is able to shorten the response time. This implies that the idea of applying Functional programming concepts to utilise resources is feasible, and the application can be further investigated by widening the scope to other available concepts so that more functional concepts will be useful for saving resources.

3.4 Event-Driven with Graphical User Interface (GUI) Programming

The GUIs that are used in any software design, currently, emerged as a result of the field of HCI (Human Computer Interaction) and ID (Interaction Design) going through incremental and revolutionary changes coming from a myriad of different disciplines [64]. The user interfaces designed and employed in the major smartphone operating systems of today follow certain official guidelines developed by their vendors. These guidelines [65], [66], & [67] are a result of the HCI journey that has taken several years. But, the road of this evolution has always left performance considerations out of the interface development. Hence, these guidelines do not consider the design time performance consideration at the GUI level; although, this background has broadly described HCI, UX (User Experience), ID, and so forth.

The graphical user interface (GUI) of a piece of software is composed of a set of GUI elements, such as buttons, text boxes, etc., arranged in a meaningful manner that allows the user to interact with an electronic device [68]. Different elements of a graphical user interface exist on different software development platforms, including on the modern-day smartphone platforms, such as Andriod, iOS, and Windows Phone. However, there is also a large overlap amongst the GUI elements that is common to all mobile operating systems (and even desktop platforms) [69]. For example, the button is a common GUI element available in all smartphone operating systems and graphical desktop operating systems.

On a given UI, some of these elements, generally or under certain circumstances, can be replaced with other similar elements. For example, the day of the week can be

selected using a dropdown menu or radio buttons. This means that these GUI elements can be arranged in a myriad of ways to accomplish the same task, albeit some arrangements are more meaningful and easier to understand [70]. That is why user design guidelines have existed since the late 1980s, and are frequently used in software development to maximise the ease of use of a given GUI [71]. However, for a given screen of a software, the developer can come up with several arrangements of the GUI components, and all of those arrangements can completely abide by the usability guidelines. These screens can even make use of different GUI components from each other [72].

Given such circumstances, the selection of one meaningful arrangement of GUI out of many others has always either been irrational or just random. But, what if certain GUI elements, say a text box or label, etc., are faster than some other GUI elements, like a dropdown list? With such information at hand, a developer can make a more informed decision on which GUI elements can be swapped for others, when usability is not compromised, to make gains on performance. This is what this research aims to do.

It aims to create certain comparisons amongst the GUI elements that can be referred to by user interface developers or application developers concerned with system resources. In that, this research aims to open up the pathway for further explorations in incorporating performance factors in user interface designs.

4. An Evaluation of some key research works

Research in Programming Paradigms can be divided into four categories: They are the study of Programming Paradigm concepts in terms of efficiency (resource consumption), comparative analysis of the Programming Paradigms with the other Programming Paradigms, Evaluation of Programming Paradigms in view of certain types of software development (application domains), and the study of Multi Paradigm Programming. In the following sections, relevant literature to these four research categories is reviewed.

4.1 Basic inherent issues in Programming Paradigms

All programming paradigms have their own merits and demerits. In general, we can classify the Programming Paradigm issues into two major categories. The first category is the inappropriateness of the programming paradigms for certain types of software developments; in section 4.2 the relevant literature is reviewed. And, the second category is the resource consumption overhead; in section 4.3 the relevant literature is reviewed. As a solution for these inherent faults, researchers have recommended alternative programming paradigms and have also come up with new programming paradigms [73, 74, 75, 76, 77, 78, 79, 80, 81, & 82]. Apart from these solutions, the researchers can suggest ways to use the currently popular programming paradigm concepts, effectively, as these concepts are widely accepted by the software industry to produce the software, magnificently.

4.2 Evaluating programming paradigms in terms of an application domain

In this section, the research studies which have evaluated programming paradigms in terms of a particular application domain are explored. No programming paradigm is suitable for all types of application development, but many of the current applications have not been developed using the appropriate programming paradigms. Hence, those applications are not reliable and are facing performance issues and other overhead. Software Engineering Researchers have confirmed, the inability of programming paradigms to be used for certain types of development with their research results, some of those research studies are discussed briefly in this section.

Kim et al. [83] has stated that ubiquitous environments manage various types of data and dynamic changes of situations in the real world. Traditional programming paradigms have a lack of straightforward features for such management and are not suitable for ubiquitous applications. Wei et al. [84] has described that traditional programming paradigms can only support the design time environment, but for ubiquitous computing, it needs to support the mobility and dynamically changing environment; so, the programming paradigms should support both the design and running time environments. The traditional programming paradigms cannot adapt to the adversity, complexity, dynamics, and levity of ubiquitous computing environments.

The Object-Oriented paradigm also has many shortcomings when applied to pervasive systems [85]. Weis, Braker, & Brandle [85] have mentioned that a set of objects are deeply involved with each other because an object can hold the references of many other objects, and they also perform synchronous method invocation on each other. Due to this dependency, if an object vanishes for some reason, such as weak connectivity or power off, the whole system's behaviour will get unpredictable. Security related issues also arise for this kind of dependency in the Object-Oriented paradigm. As pervasive applications depend on changing environments, so it is very hard for the Object-Oriented program developers to automatically adopt the pervasive applications and maintain security mechanisms with the changing environment. In pervasive applications, devices are small and resource restricted, so traditional programming paradigms are not suitable for the efficiency of mobile devices in terms of bandwidth, memory, CPU usage, and resource optimisation.

Mattos, C.B. and Carro, L. [86] have studied the memory and power consumption of OOP in embedded system applications. Whilst developing embedded applications, performance, power, memory consumption, and other requirements must be considered. OOP tends to cause overhead in terms of these requirements and, unfortunately, the software industry has had to accept it due to the other benefits.

In general, researchers are targeting Object-Oriented programming paradigms in terms of various application domains. This might be due to its dominance in the software industry market. Most of the successfully running Object-Oriented software systems are large enterprise-level systems. Some of the other programming paradigms are significantly contributing to the software industry, but are mostly tied with the Object-

Oriented programming paradigms in multi-paradigm software systems. Therefore, the other paradigms are also the reason for this incompatibility issue. Hence, research in this area should cover all of the programming paradigms which are involved in designing a software solution that would improve the accuracy of the research results. The benefits of this multi-paradigm approach are briefly stated in Section 4.4.

4.3 Comparative analysis of the Programming Paradigms

In the Programming Paradigms area, some of the researchers comparatively analyse the efficiency of mainstream programming paradigms with each other. Generally, researchers have evaluated the Object-Oriented programming against other mainstream programming paradigms. In some of the studies, performance was measured in terms of processor time or CPU time, and in other research performance was measured in terms of memory usage or energy consumption.

Apart from this, some other researchers measured the efficiency of the programming paradigms based on how easy it was to implement a particular system using two different programming paradigms.

Dingle, A. [87] has stated that, Object-Oriented programming is inefficient compared to Procedural (Imperative) programming. The overhead caused by OOP is the major factor which is unacceptable in embedded programming. Alexander, C. [88] aimed to evaluate the effect of OOP in comparison to the traditional Procedural programming style, on both power and performance in embedded processors. The research concluded that the memory consumption for both programming styles where almost the same, but OOP consumed more power and as such, OOP may not be suitable for adoption in power critical environments, such as embedded systems.

In line with the research of Alexander, C. [88], Sumil, D.; Jamwal, S. S.; and Devanand [89] also compared OOP against Procedural programming in embedded systems, and the research concluded that the speed of execution in Procedural programming is higher than in OOP by 8%.

Other related studies, such as Barnes and Hopkins [90], Da Penha et al. [91], Chatzigeorgiou [92], and Harrison *et al.* [93] have discussed the relationship between the programming paradigm and resource utilisation. Barnes and Hopkins [90] found that, Object-Oriented implementation took more memory and processor time compared to the implementation with the Imperative (structured) programming paradigm, but the Object-Oriented programming paradigm was easier to program and implement the system compared to implementing the system with the Imperative (Structured) programming paradigm.

Besides that, studies by Shin and Cury [94] and Ahearn *et al.* [95] cited in Barnes and Hopkins [90] also support this argument by claiming that, with the adoption of the Object-Oriented approach, the ease of modelling was achieved where pragmatic implementations were considered.

An experiment was conducted by Da Penha *et al.* [91] to investigate the performance of programming paradigms and languages using multi-threading on digital

image processing. The comparisons amongst the programming paradigms, such as the Object-Oriented and Procedural programming paradigms, were conducted with sequential and parallel image convolution implementation. Java and C++ programming languages were utilised to develop the programs. They found that the Object-Oriented programming paradigm took more time to respond than the Imperative (Procedural) programming paradigm.

According to Chatzigeorgiou [92], the performance of the Procedural programming paradigm was better as compared to the Object-Oriented programming paradigm, and Harrison et al. [93] have found that Object-Oriented yielded faster execution time and compilation time, and was easier to implement as compared to the Functional programming paradigm. The application domain used in this experiment was an image analysis algorithm. However, in a similar research by Pankratius et al. [26], they refuted the claim that the Functional style was bad in performance. The average run time of Scala was proved to be comparable to Java, and for the run-times with smaller values, Scala had actually performed better than in Java. At this point in time, these kinds of viewpoints may not be appropriate as the current development scenario is based on multi-paradigm programming languages. Pankratius et al.'s [96] research result against Chatzigeorgiou [92] was one of the valid supports for this argument.

4.4 Multi-Programming Paradigms

Contemporary software developments and development platforms support multi-paradigms. Basically, all mainstream programming languages support multi-paradigm development, although each language design is based on a specific programming paradigm as a core. For example, Java is an Object-Oriented Programming Language but it supports Imperative, Event-Driven with GUI programming, Functional programming and others as specified in Table 1.

Multi-paradigm programming, allows the programmer to design a system with a number of different paradigm principles. The efficiency of multi-paradigm based software system solutions is always better than the single-paradigm software systems. The use of multi-paradigm programming techniques, could lower implementation costs, and result in more reliable and efficient applications [97].

All the current software project designs use the programming concepts from various programming paradigms. In Fig. 3**,** a sample scenario is depicted. Likewise, all current operating systems support multi-paradigms by providing a wide set of tools and frameworks. With that, any research on programming paradigms should cover all of the mainstream programming paradigms which are used in the software design. However, previous research in this field had only addressed a particular programming paradigm or a programming concept [96, 98, 99, 100, 101, & 102].

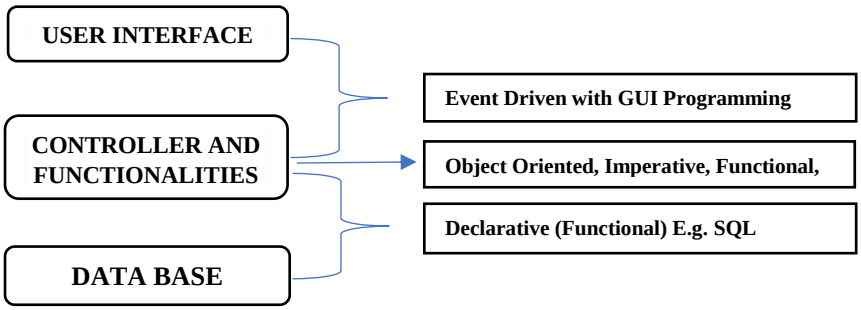

Fig. 3: Three-tier architecture with the application of Programming Paradigms

Researchers in this area, mainly focus on the multi- paradigm software design and the interaction between the different programming paradigms. In the following paragraphs, the related work of a few PhD theses are briefly summarised.

Coplien [103] has demonstrated in his research work that each multi-paradigm approach to software development has to deal with the issue of selecting an appropriate paradigm for a given application domain concept, at least to some extent. According to him, this process is seen as a mapping between the application (problem) and the solution domain. In line with this research, Valentino [104] proposed a new method of multi-paradigm software development with feature modelling in his PhD thesis, which improves the paradigm selection process to model both the application and solution domain. Its output is a set of paradigm instances annotated with the information about the corresponding application domain concepts and features.

Diomidis and Spinellis [105] have proposed the use of object-oriented design techniques as a method for encapsulating programming paradigms within multi-paradigm applications, and for abstracting common characteristics across paradigms. In another research, Coplien [103] has presented a broad design method called the multi-paradigm design. The main aim of this research [103] was to understand how to build systems that capture the structure of their domains in ways that support intentionality, software comprehension at the system level, and greater ease of evolution.

Basically, researchers in this area disregarded the resource utilisation perspectives of the multi-paradigm software design and developments. A rigorous research on the resource consumption aspects of programming paradigms, when they collaborate with other programming paradigms to achieve programming solutions, would help to achieve resource efficient software solutions.

5. Conclusion

The Programming Paradigm domain is the kernel in any software design. The Programming Languages and any software design cannot be developed without the role of Programming Paradigms. According to Campbell-Kelly [106], software possesses a great opportunity for innovation and increased efficiency. As such, further research in programming paradigms can be deemed a good way to mine this opportunity. Basically,

not much effort is being given to making software more efficient as the developers mainly concentrate on the achievement of the required solution.

Based on the above literature's knowledge at hand, the following thoughts can be deduced, whereby achieving the objectives of this paper. As per the objectives of this paper, the following content provides the concluding remarks for the academic and research community in this domain.

As for the computing education at this point of time, principally, the mainstream programming paradigms, such as Imperative (Structural and or Procedural), Object-Oriented, Functional, and Event-Driven with GUI programming, should be covered in the curriculum. Additionally, Logic Programming and Concurrent Programming could be covered as they have a notable application base in the current development scenario.

In another article, Selvakumar [107] has discussed a teaching method of the Programming Paradigms and their relevant programming languages. This article would help to teach the programming paradigms together with the respective programming languages for an application development domain. In general, the current curriculums focus on a single programming paradigm, actually, teaching multi-paradigms is appropriate for the computing education as current software designs cannot be accomplished with a single programming paradigm. For example, Imperative, Object-Oriented, and Event Driven with GUI programming can be covered in a single module and, in this case, web application development can be considered as an application domain.

In the case of Programming Languages, with the aid of one or two Programming Languages, all of these Programming Paradigm approaches can be delivered. However, in the application point of view, a Programming Language can be used for each application development, such as web application development, desktop application development, mobile application development, and the embedded application developments.

Fig. 1 and Fig. 2 can be referred to, in order to choose the appropriate programming languages to cover all of these mainstream programming paradigms and all types of application development domains. In some cases, software developers prefer different languages basically due to their interest, experience, support from the vendors, supporting tool availability, etc.; thus, the name of the programming languages are not suggested here. Therefore, in the computing curriculums, a total of four modules/subjects might be sufficient to cover the main-stream programming paradigms, their relevant programming languages and all of the four application development domains: the web, desktop, mobile, and embedded.

As for the research point of view, basically, all of the categories of research in this domain, which were described in section 4, should focus on the multi-paradigm aspects. Mostly, the previous similar research works had only addressed a particular programming paradigm or a programming concept. In the case of a research study on the efficiency of a programming design, not only should the study focus on the core programming paradigm, all other associated programming paradigms and their concepts should be considered in order to improve the accuracy of the research outcomes.

The researchers have suggested a new programming paradigm for the programming paradigm inherent problems as a solution; but instead, researchers have to identify a way to use the main-stream programming paradigms and their concepts, effectively, as most of the successfully running current projects are based on these programming paradigms.

Apart from this, researches on the resource utilisation aspects of programming paradigms, have to consider all of the other programming concepts from other programming paradigms which contribute to the software design. This would help to achieve the optimal software solutions. Finally, the researchers need to look at every possible aspect, layer, and architecture to get the software on par with the hardware in terms of efficiency.

Acknowledgments

Thanks to God for making this paper successful. Sincere thanks to our university management for the moral support and resources provided during the research.

References

[1] Daniel G. Bobrow, "If Prolog is the answer, what is the question", in FGCS,Tokyo, Japan, 1984, pp. 138-145.

[2] Linda Wieser Friedman, Comparative Programming Languages: Generalizing the Programming Function. Prentice Hall, 1991.

[3] Pamela Zave, "A compositional approach to multiparadigm programming," IEEE Softw., vol. 6, no. 5, pp. 15-25, Sept. 1989.

[4] Peter Van Ray and Seif Haridi, Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004.

[5] P.Van Roy and H. Seif , Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge, MA, 2004.

[6] P.Van Roy , The Principal Programming Paradigms, Poster version 1.08. [Online] Available from: www.info.ucl.ac.be/~pvr/paradigms.html, 2008.

[7] Stephen Cass, [Online] Available from: http://spectrum.ieee.org/computing/ soft ware/the-2015-top-ten- programming-languages.

[8] Stephen Cass, [Online] Available from: http://spectrum.ieee.org/computing/soft ware/the-2016-top-programming-languages

[9] Zuhud et al., "A preliminary analysis on the shift of programming paradigms," in ICT4M, Isesco Hay Rabat, Morocco, 2013, pp. 1-5.

[10] Souza et al., "An Experimental Study to Evaluate the Impact of the Programming Paradigm in the Testing Activity," Latin-american Center Informatics Stud. (CLEI) Electron. J., vol. 15, no. 1, pp.1-13, April 2012.

[11] Peter Van Roy and Seif Haridi, Concepts, Techniques, and Models of Computer programing. MIT press, 2004.

[12] Arvind Kumar Bansal, Introduction to Programming. CRC Press, 2013.
[13] Doris Appleby, Programming Languages: Paradigm and Practice. McGraw-Hill, 1991.
[14] David Watt, Programming Language Concepts Paradigms. Prentice Hall PTR, 1993.
[15] B. Allen Tucker and E. Robert Noonan, Programming Languages. 2nd ed. McGraw-Hill Eductaion, 2006.
[16] Gabbrielli et al., Programming Languages: Principles and Paradigms. New York: Springer-Verlag London, 2010.
[17] Peter Van-Roy and Seif Haridi, Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004.
[18] Doris Appleby and J. Julius Vandekopple, Programming Languages: Paradigm and Practice. 2nd ed. New York: Mcgraw-Hill, 1997.
[19] B. Allen Tucker and E. Robert Noonan, Programming Languages: Principles and Paradigms. 2nd ed. Boston: McGraw-Hill Education, 2007.
[20] Quintao Pereira et al, Programming Languages. Brazil: Springer-Maceio, 2014
[21] A. Raphael Finkel, Advanced Programming Language Design. Addison-Wesley, 1996.
[22] W.Robert Sebesta, Concepts of Programming Languages. 7th int. ed. Addison-Wesley, 2006.
[23] Philip Roberts. (2013, April 2). Imperative vs. Declarative. [Online]. Available :http://latentflip.com/imperative-vs-declarative/.
[24] Microsoft Developer Network. Functional Programming vs. Imperative Programming. [Online]. Available: http://msdn.microsoft.com/en-us/library/bb669144.aspx .
[25] Zuhud et al., "A preliminary analysis on the shift of programming paradigms," in ICT4M, Isesco Hay Rabat, Morocco, 2013, pp. 1-5.
[26] D. O. Da Penha et al, "Performance evaluation of programming paradigms and languages using multithreading on digital image processing," in Proc. 4th WSEAS Int. Conf. Appl. Math. and Comput. Sci., 2005, pp.3–8.
[27] R. Machado, "An Introduction to Lambda Calculus and Functional Programming" in IEEE Theoretical Comput. Sci. Conf., Rio Grande, 2013, pp. 26-33.
[28] P. Hudak, The Haskell School of Expression: Learning Functional Programming through Multimedia. Cambridge: Cambridge University Press, 2000.
[29] M. Fenwick et al., "An Open-Source Sandbox for Increasing the Accessibility of Functional Programming to the Bioinformatics and Scientific Communities," in 9th Int. Conf. Inform. Technology – New Generations, Las Vegas, 2012, pp. 89-94.
[30] F. Stephen, "Event-Driven Programming:Introduction, Tutorial, History," vol. II, no. 1, pp. 1-59, Aug. 2006.
[31] M. M. ELssaedi, "Event-Driven Programming: A new approach to Event-Driven Programming," vol. I, no. I, pp. 1-157, Aug. 2008.

[32] Dataflow Programming Concept, Languages and Applications Tiago Boldt Sousa1,2 tiago.boldt@fe.up.pt 1 INESC TEC (formerly INESC Porto) 2 Faculty of Engineering, University of Porto Campus da FEUP Rua Dr. Roberto Frias, 378 4200 - 465 Porto, Portugal.

[33] Johan Montagnat et al., "A data-driven workflow language for grids based on array programming principles", ACM Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science, New York, USA, 2009.

[34] Reflection in logic, functional and object-oriented programming: a Short Comparative Study. Proc. of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI. pp. 29–38. August, Canada, 1995.

[35] M. H. Austern. Generic Programming and the STL Professional computing series. Addison-Wesley, 1999.

[36] Francesca Rossi, Peter van Beek, Toby Walsh, "Constraint Programming". [Online] Available from: https://cs.uwaterloo.ca/~vanbeek/Publications/kr_handbook06.pdf.

[37] David, W, "The concepts of concurrent programming", [Online] Available from ftp://ftp.sei.cmu.edu/pub/education/cm24.pdf.

[38] X. Meng, "Designing approach analysis on small-scale software performance testing tools," in Int. Conf. Electronic and Mechanical Engineering and Information Technology, 2011 © IEEE. doi: 10.1109/EMEIT.2011.6023983.

[39] L. Richie et al., "An object-oriented simulation–optimization interface," Comput. and Structures, vol. 81, no. 17, pp. 1689-1701.

[40] P.Van Roy and H. Seif , Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge, MA, 2004.

[41] A. B. Tucker and R. E. Noonan, Programming languages Principles and paradigms. New York: McGraw-Hill, 2007.

[42] Case Study: Oracle, "Class StringBuilder", Oracle, 2013.

[43] Case Study: Oracle, "Class StringBuffer", Oracle, 2013.

[44] Case Study: Oracle, "Class Vector<E>", Oracle, 2013.

[45] Case Study: Oracle, "Class ArrayList<E>", Oracle, 2013.

[46] M. Rubio-Sánchez, "Tail Recursive Programming by Applying Generalizatio," in Conf. Innovation and Technology Comput. Sci. Educ., 2010.

[47] M. C. ER, Performance Evaluations of Recursive and Iterative Algorithms for the Towers of Hanoi Problem. Computing, 1986.

[48] S. Schaeckeler and W. Shang, "Stack Size Reduction of Recursive Programs," in Inter. Conf. Compilers, Architecture and Synthesis for Embedded Syst., 2007.

[49] Venkat Subramaniam, 2012. Scala for the Intrigued - Recursions and Tail Call Optimization. Issue #35, May 2012. The Pragmatic Programmers.

[50] Y. A. Liu and S. D. Stoller, "From Recursion to Iteration: What are the Optimizations?," in Proc. ACM workshop on Partial Evaluation and Semantics-Based Program Manipulation, 2000.

[51] S. Schaeckeler and W. Shang, "Stack Size Reduction of Recursive Programs," in Inter. Conf. Compilers, Architecture and Synthesis for Embedded Syst., 2007.

[52] P. Chandran, "Updating and Extending the Open Source Java Coverage Tool, Coverlipse," M.S. thesis, Uni. Edinburgh, 2009.
[53] C. B. Mattos and L. Carro, "Making Object Oriented Efficient for Embedded System Applications," ACM Integrated Circuits and Systems Design, Florianopolis, 2005, pp. 104-109.
[54] M. Boasson, "Embedded Systems Unsuitable for Object Orientation," Reliable Software Technologies, vol. 2361, no. 1, 2002, pp. 1-12.
[55] S. W. Haney, "Is C++ Fast Enough for Scientific Computing?," Comput. Physics, vol. 8, 1994, pp. 690-694.
[56] A. D. Robinson, "C++ Gets Faster for Scientific Computing," Comput. Physics, vol. 10, 1996, pp. 458-462.
[57] B. Calder et al., "Quantifying Behavioral Differences Between C and C++ Programs," J. Programming Languages, vol. 2, 1994, pp. 313-351.
[58] A. D. Robinson, "The abstraction penalty for small objects in C++," in Parallel Object-Oriented Methods and Applications Conf., Santa Fe, New Mexico, 1996.
[59] Kayun, C., Chonawat, S., (2007a). Object-Oriented Programming Strategies in C# for Power Conscious System. World Academy of Science, Engineering and Technology, 10 (10), p587-592.
[60] E. Albert et al. (2010). Parametric Inference of Memory Requirements for Garbage Collected Languages. [Online] Available: http://dl.acm.org/citation.cfm?id=1806671.
[61] H. Simoes et al., "Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs," in Proc. 17th ACM Intern. Conf. Functional programming, Copenhagen, NY, 2012, pp. 165-176.
[62] S. Antoy and A. Jost. (2013). A Target Implementation for High-Performance Functional Programs [Online]. Available: http://web.cecs.pdx.edu/~antoy/homepage/publications/tfp13/paper.pdf.
[63] A. Minutolo et al., "A Lazy Evaluation Approach for Mobile Reasoning in DSSs," in 12th IEEE Intern. Symp. Computational Intell. and Inform., Budapest, 2011.
[64] J. Grudin, A Moving Target - The Evolution of Human-Computer Interaction. 2012.
[65] Microsoft Developers Network. (2015). Introduction to Universal Windows Platform (UWP) apps for designers - Windows app development.
[66] Google design guidelines. (2015). Introduction - Material design - Google design guidelines. [online] Available: http://www.google.com/design/spec/material-design/introduction.html.
[67] Apple iOS HCI Guidelines. (2015). iOS Human Interface Guidelines: Designing for iOS.
[68] C. Ghaoui, Encyclopedia of human computer interaction. Hershey PA: Idea Group, 2006.
[69] X. Meng, "Designing approach analysis on small-scale software performance testing tools," 2011 Intern. Conf. Electronic & Mechanical Eng. and Inform. Technology.

[70] J. Grudin, "Three Faces of Human-Computer Interaction," IEEE Annals Hist. Comput., vol. 27, no. 4, pp.46-62, 2005.
[71] D. Norman, The psychology of everyday things. New York: Basic Books, 1988.
[72] Thompson and John, Developer's Guide to UI. Oracle, 2012, pp.16-18.
[73] Torben et al., *Towards a Programming Paradigm for Pervasive Applications based on the Ambient Calculus*. Microsoft Research, Cambridge, 2011.
[74] Mbayashi, Y.; Ledgard, H.F.; Dept. of Comput. & Inf. Eng., Nippon Inst. of Technol., Saitama, Japan,An experiment on a new programming paradigm, IEEE, 2005.
[75] Jan Vitek, New Paradigms for Distributed Programming Jan Vitek ,Object Systems Group, University of Geneva, Geneva, Switzerland, 1997.
[76] Dennis Gannon, *Programming Paradigms for Scientific Problem Solving Environments*. Woco9, Prescott, 2006.
[77] L. Ling Shang, "Experiments on Programming paradigms for large scale scientific computing, grids, desktop grids and private clouds," M.S.thesis, 2010.
[78] Manuel Oriol, *The Internet Programming Paradigm.* CiteSeer 6M, 2011.
[79] M.Y.Kim et al., *A New Programming Language for Ubiquitous Applications*. 2013.
[80] S. A. Salustri and R. D. Venter, *A new programming paradigm for engineering design, engineering with Computers*. Springer Verlag, London limited, 1994.
[81] Johan Glimming, Thorsten Altenkirch and Patrik Jansson, What is the next Programming Paradigm?, Second International Software Technology Exchange Workshop, Kista, Sweden, 2012.
[82] H. Jin et al., "Testing new programming paradigms with NAS Parallel Benchmarks," NASA Technical Report Server, 2000 © NASA Ames Research Center; Moffett Field, CA United States. doi: 20000064623
[83] M. Y. Kim et al. A New Programming Language for Ubiquitous Applications [online]. Available: http://www.ics.uci.edu/~lopes/bspc05/papers/kim.pdf.
[84] W. Wei et al., "Allotropy Programming Paradigm for Ubiquitous Computing Environment," in Int. Conf. Convergence Inform. Technology, pp. 514-521, 2007.
[85] T. Weis et al. (2013) Towards a Programming Paradigm for Pervasive Applications based on the Ambient Calculus [online]. Available: http://wenku.baidu.com/view/f0cfddc72cc58bd63186bdac.html.
[86] C. B. Mattos and L. Carro, "Making Object Oriented Efficient for Embedded System Applications," ACM Integrated Circuits and Syst. Design, pp. 104-109, 2005.
[87] A. Dingle, Improving C++ Performance in Temporaries. Seattle Univ., WA, USA, 1998.
[88] C. Alexander, "Performance and power evaluation of C++ object-oriented programming in embedded processors," Inform. and Softw. Technology, vol. 45, no. 4, pp. 195-201, 2003.
[89] Sunil Dutt et al., "Object Oriented Vs Procedural Programming Embedded Systems," Intern. J. Comput. Sci. & Commun., vol. 1, no. 2, pp. 47-50, 2010.
[90] Barnes and Hopkins, The impact of programming paradigms on the efficiency of an individual-based simulation model. UK, science direct, 2001.

[91] D. O. Da Penha et al., "Performance evaluation of programming paradigms and languages using multithreading on digital image processing," in Proc. 4th WSEAS Intern. Conf. Applied Math. and Comput. Sci., pp.3–8, 2005.

[92] A. Chatzigeorgiou, "Performance and power evaluation of C++ object-oriented programming in embedded processors," Inform. and Softw. Technology, vol. 45, no. 4, pp.195-201, 2003.

[93] R. Harrison et al., "Comparing Programming Paradigms: an Evaluation of Functional and Object-Oriented Programs," Softw. Eng. J., vol. 11, no. 4, pp. 247-254, 1996.

[94] Y. J. Shin and P. Cury, "Exploring community dynamics through size dependent trophi interactions using a spatialized individual-based model," Aquati Living Resources, vol. 14, no. 2, 2001.

[95] S. C. Ahearn et al., "TIGMOD: an individual-based spatially explicit model for simulating tiger/human interaction in multiple use forests," Ecological Modelling, vol. 140, no. 81, 2001.

[96] V. Pankratius et al., "Combining Functional and Imperative Programming for Multicore Software: An Empirical Study Evaluating Scala and Java," in 34th Int. Conf. Softw. Engineering, New York, 2012, pp. 123-133.

[97] D. Diomidis Spinellis, Programming paradigm as Object classes: A structuring mechanism for Multi Paradigm Programming. 1993.

[98] NELSON, H. J., MONARCHI, D. E. & NELSON, K. M. Year. Evaluating emerging programming paradigms: an artifact-oriented approach. In: System Sciences, 1998. Proceedings of the Thirty-First Hawaii International Conference on, 6-9 Jan 1998 1998. 446-454 vol.6.

[99] S. Antoy and A. Jost, *A Target Implementation for High-Performance Functional Programs*. 2013.

[100] A. Minutolo et al., "A Lazy Evaluation Approach for Mobile Reasoning in DSSs," in 12th IEEE Int. Symp. Computational Intell. and Informatics, 2011.

[101] Sunil Dutt et al., "Object Oriented Vs Procedural Programming Embedded Systems," Int. J. Comput. Sci. & Commun., vol. 1, no. 2, pp. 47-50, 2010.

[102] W. D. Clinger, "Proper Tail Recursion and Space Efficiency," in Proc. ACM Conf. Programming Language Design and Implementation, pp. 174 – 185, 1998.

[103] O. James Coplien, Multi-Paradigm Design for C++. Addison-Wesley, 1999.

[104] Valentino Vranic, Multi Paradigm design with Feature modelling. 2004.

[105] D. Diomidis Spinellis, Programming paradigm as Object classes: A structuring mechanism for Multi Paradigm Programming. 1993.

[106] M. Campbell-Kelly, "The History of the History of Software", IEEE Annals Hist. Comput., vol. 29, no. 99, pp. x4, 2007.

[107] S. Selvakumar," Teaching Programming Subjects with Emphasis on Programming Paradigms", in icaet-14 © Atlantis Press. doi:10.2991/icaet-14.2014.22.