COMPARATIVE ANALYSIS OF PROGRAMMING LANGUAGES - A STUDY

Onu F. U.¹, Asogwa S. C.², Ugwoke F. N.³
¹Computer Science Department, Ebonyi State University, Abakaliki – Nigeria
²Computer Science Department, Michael Okpara University of Agriculture, Unudike- Nigeria
³Computer Science Department, Michael Okpara University of Agriculture, Unudike- Nigeria
¹, ², ³ uche.fergus@gmail.com, ³asasogwa@gmail.com, ³ndidi.ugwoke@gmail.com

Abstract: The choice of programming language to use in developing applications for mission critical assignments such as the military, real-time application in industrial controls and health monitoring systems needs critical assessments before an appropriate choice can be made. The factors to consider in making this choice include; the memory requirements, reliability, speed, ease of coding, scalability and cost of coding and deployment. This paper makes attempt to analyze some programming languages features that can assist prospective and indeed all programmers in making the right decision in the choice of a programming platform to choose from. In this study, the functional, imperative, logic and the object oriented approaches was analyzes from the point of their features and applications.

Key Words - Levels of Abstraction, OOP Paradigms, Programming Languages

I INTRODUCTION

A computer program is a set of logical instructions written in a computer language for the hardware to execute. The programming language is a language with sets of rules or instruction formats through which the programmer writes these logical instructions. The computer programmer is therefore a person who understands the syntax, semantics and styles of a computer language and can use it to design and implements applications. There are thousands of programming languages in existence today, and new ones are waiting to be released sooner than later. Consequently, the choice of which programming language to learn or use in developing applications has become increasingly difficult to make. Whatever platform one uses determines how successful and reliable such application will be. The choice of languages to use in mission critical assignments such as the military and real-time application in industrial controls needs critical assessments before an appropriate choice can be made. The factors to consider in making this choice are includes; the memory requirements, reliability, speed, ease of coding, scalability and cost of coding and deployment. This paper makes attempt to analyze some programming languages features that can assist prospective programmers and also existing programmers in making the right decision in the choice o a programming platform to chose.

II TYPES OF PROGRAMMING LANGUAGES

The level of a programming language is a function of how close or far away the language is to the computer hardware. The ones that are close to the hardware (i.e the ones that the hardware can execute without the need of interpreters) are known as the machine and the low level programming languages. This means that the computer hardware does not require a lot of effort to understand and process the instructions. The high level languages on the other hand requires the interpreters and the compilers to execute the instructions written in a computer language.

III LEVELS OF ABSTRACTION

Another way to understand programming languages is in their level of abstraction; the machine language has no abstraction which means that the 0s and 1s are understood directly, it does not contain any code that is abstract in meaning to the hardware, while the low level and high level languages containing a lot of abstracts such as mnemonics, and English like statements that requires interpretation before it can be executed. See tables 1 and 2

IV FEATURES OF COMPILERS AND INTERPRETERS

When we design and implement an application on a high level programming language, the next thing to do is to use a tool (compiler) to convert this program into a form understandable by the hardware – the machine code. A compiler reads the whole source code and translates it into a complete machine code program to perform the required tasks which is output as a new file. Interpreter is a program that executes instructions written in a high-level language. An interpreter reads the source code one instruction or a line at a time, converts these lines into machine code and executes it. There are lots of functions of interpreters and the compilers. These functions are captured in the way they handle instructions, the memories and in the similarities and the differences between them as illustrated below.

Table 1. Differences Between Compilers and Interpreters

<table>
<thead>
<tr>
<th></th>
<th>Compilers</th>
<th>Interpreters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation of source programs</td>
<td>The whole program before execution</td>
<td>One line at a time when it is run</td>
</tr>
<tr>
<td>Frequency of translation</td>
<td>Each line is translated once</td>
<td>Has to be translated</td>
</tr>
</tbody>
</table>

www.iwcsit-apm.com

International Journal of Computer Science & IT
When a program is executed, it can be in the form of a source program or an object program. The source program is written in a high-level programming language and contains specific instructions for the computer to execute. The object program is the executable version of the source program that can be run on the computer. Every time a program is executed, it has to be saved in the computer's memory, which can be a slow process. However, if the program is saved in the computer's memory, it can be executed without the need for the source program.

**Object program**
- Can be saved for future execution without the source program.
- No object program is generated, so the source program must be present for execution.

**Similarities**
1. Both translate high-level language to machine code.
2. Both detect errors in the program and print error messages.
3. Both work out where to store the object program and its data in memory.

**Table 2. Levels of Abstraction of Programming Languages**

<table>
<thead>
<tr>
<th>Level</th>
<th>Instructions</th>
<th>Memory handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Language</td>
<td>Series of 0s and 1s, no abstraction</td>
<td>The programmer handles the memory management in coding time</td>
</tr>
<tr>
<td>Low level languages; machine code, assembly language</td>
<td>Simple machine-like instructions There is abstraction</td>
<td>Direct memory access and allocation. A lack of memory management support — programmers do that themselves</td>
</tr>
</tbody>
</table>

**High level languages**
- Expression and explicit flow of control.
- Abstraction exists
  - Names for almost everything: variables, types, subroutines, constants, modules
  - Complex expressions
  - Control structures (conditionals, switches, loops)
  - Composite types (arrays, structs)
  - Type declarations
  - Type checking
  - Easy ways to manage global, local and heap storage
  - Subroutines with their own private scope
  - Abstract data types, modules, packages, classes
  - Exceptions
  - Methods, Objects etc.

**V PROGRAMMING LANGUAGES**
There are vast numbers of programming languages in existence today and the number is still in the increase ranging from the machine language to the very high level languages. The application areas are equally vast. The features of the programming languages determine the areas where they are suitable for; stand alone systems, industrial applications, network and internet programming languages exist from where programmers can choose from. The following are the descriptions of the types of programming languages.

**VI MACHINE LANGUAGE**
This language is written in the lowest level of computer operations. Machine language is written as a series of 0s and 1s (the machine's native language) and so it is very difficult to write but very easy for the hardware to understand and execute. Most machine languages are provided by the hardware manufacturer and so it is hardware specific requiring the codes to be written in specific instruction format of the hardware.
VII ASSEMBLY LANGUAGE
Assembly language instructions are written in short English-like letters referred to as mnemonics codes. Its codes must be assembled into machine language before the hardware can execute it. Assembly language programs are machine dependent. It executes faster than the high level languages.

VIII HIGH LEVEL LANGUAGES
High-level programming languages allow the specification of a problem solution in terms closer to those used by human beings. The high level languages are machine independent and so it is more portable than the low level languages. It is easier to write because it is mainly written in English-like statements. Its disadvantage is that it requires more memory for its storage and operations. The high level language has a high level of abstractions making it impossible for hardware to execute them directly. Interpreters and compilers interpret and convert the high level languages to machine codes which it can execute directly. The high level languages can further be classified into some categories according to its paradigms, namely; procedural, functional and object oriented languages. A paradigm is a technique or ways of approaching a software implementation problem in order to find solutions.

Generally, a selected Programming Paradigm defines main property of software developed by means of a programming language supporting the paradigm of Which, scalability/modifiability, integrity, reusability, portability, performance, reliability and ease of use and creation are part of [1]

IX PROGRAMMING LANGUAGE PARADIGMS
A programming paradigm is a style or “way” of programming. [5]. A paradigm is not a type of programming language. It is simply an approach to programming that is suitable for solving problems through a programming language. Common programming paradigms can be classified as, procedural, logic, object oriented, etc.

X PROCEDURAL PARADIGMS
In procedural techniques, a related block of statements that performs a function is grouped together into one module known as procedure. For instance a module that performs a complete function such as computing the cumulative grade point of a student’s of a particular class can be called up from another procedure that requires same computation. It handles programs as a group of structured statements that states the steps a program must follow to come to a solution. Examples of languages that follow procedural paradigms are Ada, C, C++, Java, PHP, Python etc.

Program example ; the following C program shows how a function can be developed that returns an answer. This answer can then be called by any other procedure that requires computing a factorial

```c
Function factorial (value) {
    Var answer = 1;
    While (I <= value) {
        Answer = answer *1;
        I++ ;
    }
    Return (answer);
}
```

Programmers have dreamed/attemped of building systems from a library of reusable software components bound together with a little new code. Imperative (Procedural) Programming Paradigm is essentially based on concept of so-called “Functions” also known as “Modules”, “Procedures” or “Subroutines”. A function is a section of code that is parcelled off from the main program and hidden behind an interface [1]

XI LOGIC PROGRAMMING PARADIGMS
The unification of logic and functional programming, like the Holy Grail, is sought by countless people [2, 3]. Lawrence C. Paulson and Andrew W. Smith [6] posits that for logic programming, the realists and purists are far apart. Programming in a pure style is difficult. Existing Prolog systems do not even provide pure Prolog as a subset. They use depth-first search (which is incomplete) and they omit the occurs check (which can create circular data structures). Pure logic programs can be written by translating functional programs into clauses. But this is hardly logic programming: key aspects like backtracking are lost. Logic programming is far more ambitious than functional programming, which is why it has not reached a similar stage of maturity.

Robert Kowalski [7] in a study of logic programming stated that the driving force behind logic programming is the idea that a single formalism suffices for both logic and computation, and that logic subsumes computation. But logic, as this series of volumes proves, is a broad church, with many denominations and communities, coexisting in varying degrees of harmony. Computing is, similarly, made up of many competing approaches and divided into largely disjoint areas, such as programming, databases, and artificial intelligence. On the surface, it might seem that both logic and computing suffer from a similar lack of cohesion. But logic is in better shape, with well-understood relationships between different formalisms. For example, first-order logic extends propositional logic, higher-order logic extends first-order logic, and modal logic extends classical logic. In contrast, in Computing, there is hardly any relationship between, for example, Turing machines as a model of computation and relational algebra as a model of database queries. Logic programming aims to remedy this deficiency and to unify different areas of computing by exploiting the greater generality of logic. It does so by building upon and extending one of the simplest, yet most powerful logics imaginable, namely the logic of Horn clauses are sufficient for many applications in artificial intelligence. For example, and-or trees can be represented by ground Horn clauses. See figure 1.
Although Horn clauses are the underlying basis of logic programming (LP) and are theoretically sufficient for all programming and database applications, they are not adequate for artificial intelligence, most importantly because they fail to capture non-monotonic reasoning. For non-monotonic reasoning, it is necessary to extend Horn clauses to clauses of the form:

\[ A_0 \leftarrow A_1 \land \ldots \land A_n \land \text{not } B_1 \land \ldots \land \text{not } B_m \text{ when } n \geq 0 \text{ and } m \geq 0. \]

Each clause is an atomic formula, and “not” is read as “false”. Atomic formulas and their negations are also called literals. Here the \( A_i \) are positive literals, and the not \( B_i \) are negative literals. Sets of clauses in this form are called normal logic programs, or just logic programs for short. [7]

**XII OBJECT ORIENTED PARADIGM**

Object-oriented paradigm (OOP) is a way of programming the computer by handling every task as an object, providing the method (procedure) and the data or the attribute the methods act on to produce an a computer applications. According to[8], Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another. There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type. OOP has some other features such as: – Data abstraction, Encapsulation, Polymorphism and Inheritance. Examples of OOP includes; Java, C++, Python etc.

In paradigm of OOP, class defines a kind of a concept, and objects are instances of it. Each class consists of fields and methods, where fields define the structure of the class and methods define its behavior. In other words, fields define properties of the concept and methods are functions that give an opportunity to manipulate them. When the program creates an object as an instance of some class, this object has the same fields, as its class and each method of the class can be called for this object. In such a way, class implements the mechanism of encapsulation, because the object has the same structure and behavior, but it has its own values of the fields, which can differ from corresponding values of class’s fields and can be changed during program execution. [9].

**XIII APPLICATIONS AREAS OF OOP**

Object oriented paradigms have been employed in various facets of knowledge and applications. In most of the applications it has been proven to be an effective approach. The application areas of OOP are best captured in object oriented modeling. Object oriented modeling technique is as applied in case studies and applications analyzed below:

A. **Structuring an Educational Domain:** This is a work that describes an approach to modeling educational domains and reports on the results for object-orientation. It analyzes the dependency structure of object-oriented concepts and describes the implications that the high interrelatedness of concepts has on teaching introductory programming.

One of the strengths of the object-oriented mode of software development is to provide us with a set of powerful and expressive concepts, so powerful and expressive indeed that they can serve beyond their original target area - programming. These concepts, such as classes, message passing, single and multiple inheritance were initially programming concepts; but they are in fact useful for a far more general purpose: designing systems, modeling systems, and more generally thinking about systems.

In [10], the researchers describe a modeling approach and the supporting tool for modeling educational domains through their main concepts and the relations between these concepts, and its application to the educational domain of introductory programming. In the analysis of the dependency structure of the tools they used (Eiffel and java) the researchers maximized educational domain and identifies structural dependencies between concepts and extends the idea of TRUC (testable, reusable units of cognition) by addition of two additional types of knowledge units. Their final model then used three types of knowledge units notion trunks and clusters which defines several types of relationship between them. To capture the dependency structure of the knowledge units, the truc framework defines two types of relations between notions. This relation is comparable to inheritance mechanism in object- oriented systems.

In a research to use a meta model approach for testing OOP software, the researchers tested the existing methods of testing OOP software such as specification based and program based and found out that current approach uses extra languages and the mechanism used do not maximize the advantages in OOP. The therefore proposed a meta model that offers a new approach that makes it possible to write unit tests inside the classes to be tested. It facilitates the programmer to write and maintain a software unit. The proposed metamodell contains test properties for the class properties, for the classes and for the packages. [11]

Researchers have presented an approach to teaching object- oriented programming with games. This approach may be characterized by (1) object-orientation first, programming second (2) graphically rich examples (3)

---

Figure 1. And-and or tree and Corresponding Propositional Horn Clause Program [7].
game as the trigger. The proposed approach provides a positive and supportive atmosphere in which students can learn the principles of object-oriented programming. It keeps students interested and reinforces those programming principles.[12]

B. Computational Models

Characteristic features of object oriented programming, such as polymorphism, inheritance, and dynamic binding, can be incorporated into the model in a natural way. Programs are embedded in a complete lattice of specifications, allowing for the development of a refinement calculus.[13]

Researchers in [14] believe that parallel programming can benefit from object-orientation in the same way as traditional serial programming does. For example, object-oriented languages can provide better reuse of parallel software through the mechanisms of inheritance and delegation. Object-orientation can also help with portability of parallel applications because OOP languages support high level of abstraction through separation of services form implementation. Parallel applications can be consistently and naturally developed through object-oriented analysis, design, and programming. The rapid development of ATM networks and other fast connections opens opportunities to integrate existing workstations into relatively cheap distributed computing resources. Nowadays, diverse parallel processing platforms become increasingly available to application programmers. The expanding parallel applications cover not only traditional areas, such as scientific computations, but also new domains, such as, for example, multimedia.

C. OOP Paradigms in Process Control

We usually think of software as algorithmic: we compute outputs (or execute continuous systems) solely on the basis of the inputs. This normal model does not allow for external perturbations; if non-input values of a computation change spontaneously, this is regarded as a hardware error. The normal software model corresponds to an open loop system; in most cases it is entirely appropriate. However, when the operating conditions of a software system are not completely predictable—especially when the software is operating a physical system—the purely algorithmic model breaks down.

We can now establish a paradigm for software that controls continuous processes. The elements of this pattern incorporate the essential parts of a process control loop, and the methodology requires explicit identification of these parts: (i)Computational elements: separate the process of interest from the control policy. Process definition, including mechanisms for manipulating some process variables. Control algorithm to decide how to manipulate process variables, including a model for how the process variables reflect the true state. Data elements: continuously updated process variables and sensors that collect them. Process variables, including designated input, controlled, and manipulated variables and knowledge of which can be sensed. Set point, or reference value for controlled variable. Sensors to obtain values of process variables pertinent to control. The control loop paradigm establishes the relation that the control algorithm exercises surveillance: it collects information about the actual and intended states of the process and tunes the process variables to drive the actual state toward the intended state.[15]

XIV CONCLUSIONS AND FUTURE WORK

There are numerous platforms through which instructions can be written for the computer hardware to execute. The existence of several of these platforms often creates uncertainty about the best of these platforms to choose from. Therefore there is a need to present the features and various paradigms that exists so that the prospective programmer can make an informed choice. In this paper, we have studied and presented the various approach to programming the computer hardware. We have also given some useful examples and models that will be of great guide to the programmer.

Software for the internet differs from the traditional software in terms of form, structure and behavior, consequently its paradigms should differs in terms of trust, situation and autonomy. In our future research we will present frameworks through which the object oriented paradigms can be a contributory factor in the development of the internetwork – a software paradigm for internet computing and the internet of things (IOTs).

REFERENCES


[12] Lu Yan “Teaching Object-Oriented Programming with Games” 2009 Sixth International Conference on Information Technology:

