COMPARATIVE ANALYSIS OF OBJECT-ORIENTED SOFTWARE DEVELOPMENT AND CLASSICAL PARADIGM

Eke B. O. and Iwumune Chinwe G.
Department of Computer Science
University of Port Harcourt, Rivers State, Nigeria

ABSTRACT
Increasing need for better and more efficient software have continued to put pressure on developers to know which is the best paradigm to apply in developing their system. In this paper, we will perform a comparative analysis of the object-oriented paradigm with other classical paradigm to explore their features, strength and weaknesses and discover what makes a given paradigm suitable in the light of modern day technological development. The comparison covers instruction path length, memory resources, instruction fetch, reusability and binary size of the paradigm using simple code samples. This will form a guide for developers in selection of suitable paradigm for specific system development need.

INTRODUCTION
Software refers to all the programs in a given electronic machine that drive the action of the machine. Programs are instructions given to the computers or electronic microprocessors to carry out a particular action. Software comprises of the systems software and the application software. The systems software are built into the computer by the manufacturers while the application software are designed to solve specific problems of the users. Programming a computer to solve a problem provides a unique experience and challenges (Burrows, 2003). A programmer should be able to imagine, design and construct a functional computer-based solution to a given problem. The challenges involve developing working codes required by most application. Worst still, the need of users in modern application have continued to increase dramatically. The question that borders the mind remains what paradigm will be the best both in theory and in practice in helping developers to cope with the huge challenges involved in the modern application development. Some of these challenges include:

Legacy Software Reuse: There are many critical and important software that still perform important functions today which were written many years ago with classical tools and programming languages. If the system is working why not reuse it, why must we change it remains a paradigm challenge. If we must reuse it then it simply implies that the paradigm used in developing them has received a tacit approval.

Heterogeneity of Software Landscape: In today's information technology world, there are different types of computers with various kinds of software support systems. There is the challenge of building dependable software which is flexible and operable across several platforms.

Speed of Deliver: The challenges of getting software product to the market on time is inevitable in satisfying the customers desire. Speed is inevitable in order to meet the challenges of business delivery today and for large and complex systems without having to compromise system quality. Thus a pragmatic paradigm that is responsive to the dynamic demands of today's business is needed to keep ahead of competition.

Cost of Software: Customers will like to pay less for a system that is slightly lower in quality; how much more for a product that is of equal quality to an alternative one which is much more expensive. Lower cost can be gained when system is reused instead of reinventing the wheel for every software developed. Components for some common features can be reused thereby cutting the cost of development and making the system relatively cheap. A paradigm that supports software reuse will therefore be much more desirable than those once that do not support it to the fullest.

As a result of the challenges highlighted, we need to examine the paradigm more critically to clearly recommend the one that is most adequate for certain class of application and other activities within the software engineering environment.
Comparative Analysis of Object-Oriented Software Development and Classical Paradigm


Guy Lewis Steele, Jr. (1977) "Debunking the 'Expensive Procedure Call' Myth, or, Procedure Call Implementations Considered Harmful, or, Lambda: The Ultimate GOTO". MIT AI Lab. AI Lab Memo AIM-443. October 1977.


Pantharcout2011 N.11. Sun, E-otalk fobercon7011traboo.eeni
SOFTWARE ENGINEERING PARADIGMS

The word paradigm has been used in science to describe distinct concepts. It comes from Greek (Paralleigma), "pattern, example, sample" from the verb (parade ikmani), "exhibit, represent, expose" and giat from (no, "beside", Iseyio and (ekiknomit.) "to show, to point out" (Wiki). This word has come to refer very often to a thought pattern in any scientific discipline or other epistemological context (These:1006). The Merriam-Webster Online dictionary defines this usage as "a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and the experiments performed in support of them are formulated; broadly: a philosophical or theoretical framework of any kind." (Marriam. 2011)

SCIENTIFIC PARADIGM

The historian of science Thomas Kuhn gave paradigm its contemporary meaning when he adopted the word to refer to the set of practices that define a scientific discipline at any particular period of time. Kuhn defines a scientific paradigm as: "universally recognized scientific achievements that for a time, provide model problems and solutions for a community of researchers", i.e.,

- what is to be observed and scrutinized
- the kind of questions that are supposed to be asked and probed for answers in relation to this subject
- how these questions are to be structured
- how the results of scientific investigations should be interpreted

Thus an additional component of Kuhn's definition of paradigm is:
- how is an experiment to be conducted, and what equipment is available to conduct the experiment. Thus, within normal science, the paradigm is the set of exemplary experiments that are likely to be copied or emulated.

SOFTWARE PARADIGM

Software engineering paradigm is the set of software engineering practices that define a development practice that provide model software development problems and solutions for a group of developers at any particular period of time. The software engineering practice must be acceptable universally by developers who hold the philosophy outlined by the model. Since there can be many models and many development philosophies as there are many groups of developers with varying experience it can easily be deduced that there can also be at least two paradigm in existence between a change over time. This change over time is usually a time between a paradigm in use and a new one that have been developed. In figure 1, different paradigms and their change over time where clearly illustrated. The change over time was indicated to be in the center of two main paradigm. In the period of the first change over in figure 1, paradigm 1 and paradigm 2 are practiced at a ratio in which the rate of practice of paradigm 1 continues to decline while the rate of practice of paradigm 2 tend to increase until paradigm 2 will hold sway. This trend is generally referred to as paradigm shift. Paradigm shift continues until paradigm 3 and so on. The challenge is on what happens if developer community find out that for certain projects paradigm 1 works better than paradigm 3.

In the developer community, there will then be the challenge of which paradigm to use in developing projects and what project will match a particular paradigm. This will then call for a need to critically examine the paradigms vis-à-vis various project groups. Worst still, the rapid changes in computing technology and software development needs by users have always kept developers on their toes in an effort to discover better philosophies and models that can be better adapted to solving numerous problems facing software development and software developers. This increased effort result in the development of newer paradigms within relatively shorter periods. The rapid nature of the changes have equally kept young developers at bay as to which paradigm to subscribe to and which one to jettison. Developers get entangled with the question...
of whether to change to a new paradigm or to continue using the seemingly classical one which they are very familiar with even when the developer community have accepted the new ones and expected them to make paradigm shift.

One important aspect of Kulau's paradigms is that the paradigms are incommensurable. meaning two paradigms cannot be reconciled with each other because they cannot be subjected to the same common standard of comparison. That is, no meaningful comparison between them is possible without fundamental modification of the concepts that are an intrinsic part of the paradigms being compared. This way of looking at the concept of "paradigm" creates a paradox of sorts, since competing paradigms are in fact constantly being measured against each other before it will be wise to recommend a shift to new paradigm.

Simple common onalome: A simplified analogy for software engineering paradigm is a habit of reasoning by developers, or the box" in the commonly used phrase "thinking outside the box". Thinking inside the box is analogous with paradigm. "Thinking outside the box" is usually unsuccessful, and very rarely leads to new paradigms. However, when they are successful they lead to large scale changes in the software engineering worldview. When these large scale shifts in the software engineering view are implemented and accepted by the majority, it will then become "the box" and software engineering as an engineering science will progress within it (Jacobs, 2006).

Nonetheless, competing paradigms are not fully intelligible solely within the context of their own conceptual frameworks. So this may bring shot! the practice of many paradigm together within a shift period as illustrated in figure 2. In contemporary time, it seem that all things change Over time, due to the practice of many paradigm at the same time for different projects or even for the same project. For this reason, paradigm as Concept in software engineering might more meaningfully be defined as a self-tenant explanatory model or conceptual framework. This definition makes it clear that the real baffler to comparison is not necessarily the absence of common units of measurement, but an absence of mutually compatible or mutually intelligible concepts. Using this masonry, a new software engineering paradigm which replaces an old paradigm is not necessarily better, because the criteria of judgment are controlled by the paradigm itself and by the conceptual framework which defines the paradigm and gives it explanatory value. But because the reason of use in a given software project tills towards the conceptual framework defined by the contending paradigm.

PROCEDURAL AND OBJECT-ORIENTED PARADIGM DESCRIPTION

There are many software engineering paradigm in use today in the field of computer science. however in this project we will consider the comparison of Object-oriented software development paradigm and the classical Procedural (Structured) paradigm. There is inevitably some overlap in these non mutually exclusive paradigms but the main features or identifiable differences are analyzed below:
COMPARATIVE ANALYSIS OF OBJECT-ORIENTED SOFTWARE DEVELOPMENT AND CLASSICAL PARADIGM

PROCEDURAL PARADIGM
Procedural / structured Paradigm specify the steps that must be followed to reach the desired state. Procedural paradigm is derived from structured patterns which is based upon the concept of modular system development or the procedure call. Its main characteristics is that it relies much on local variables, sequence, selectional and iterative way of developing application. It is equally modular. It is strongly related to the attribute of being computational in terms of statement that directly change a program state or its datafields.

OBJECT-ORIENTED PARADIGM
Object-oriented paradigm (OOP) is a development method that uses objects (data structures consisting of datafields and methods together with their interactions) to develop applications and computer programs. It uses objects, methods, message passing, information hiding, data abstraction, encapsulation, polymorphism, inheritance and serialization in its development process (Shally, 2008). None of the main programming paradigms have a precise, globally unanimous definition, let alone an official international standard. There is equally no any agreement on which paradigm constitutes the best approach to developing software. The reason for the disagreement is the empirical nature of software method measurement and the mix in usage in real life. The subroutines that actually implement methods might be ultimately coded in an imperative, functional or procedural style that might, or might not, directly alter state on behalf of the invoking program. Non-standardized implementations of each paradigm in numerous programming languages further complicate the overall picture, especially those languages that support multiple paradigm like C++ or even Java. However we can still analyze the variations in an academic winner based on certain known major considerations as presented below.

CONCLUSION ANALYSIS
Purely in terms of haat instruction path length, a program coded in an imperative style, without using any subroutines at all, would have the lowest count. However, the binary size of such a program might be larger than the sante program coded using subroutines (as in functional and procedural programming) and would reference more "non-local" physical instructions that may increase cache misses and increase instruction fetch overhead in modern processors. The paradigms that use subroutines extensively (including functional, procedural and object oriented) and do not also use significant miming (via compiler optimization) will, consequently, use a greater percentage of total resources on the subroutine linkages themselves (Harrison, 1996). Object oriented programs that do not deliberately alter program state directly, instead using mutate! methods (or "setters") to encapsulate these stale changes, will, as a direct consequence, have a greater overhead. This is due to the fact that message passing is essentially a subroutine call, but with three more additional overheads: dynamic memory allocation, parameter copying and dynamic dispatch. Obtaining memory from the heap and copying parameters for message passing may involve significant resources that far exceed those required for the state change itself. Accessors (or "getters") that merely return the values of private member variables also depend upon similar message passing subroutines, instead of using a more direct assignment (or comparison), adding to total path length.

PSEUDOCODE COMPARISON
A pseudoxlle comparison of procedural, and object oriented approaches used to calculate the area of a trianglea assuming no subroutine inlining no macro preprocessors. register arithmetic and weighting each instruction Step’ as just I instruction - as a crude measure of instruction path length - is shown in table I. Note that the actual arithmetic operations used to compute the area of the triangle are the same in die two paradigms. The opem ions were "raped in a subroutine call that makes the computation general and reusable. ihe saute effect could be achieved in a purely imperative program using a macro preprocessor at just the cost of increased program size (only at each macro invocation site) without a corresponding pro rata runtime cost (proportional to n invocations - that may be situated within an inner loop for instance). Conversely, subroutine inlining by a compiler could reduce procedural programs to something similar in size to the purely imperative code. However, for object-oriented programs, even with inlining, messages still have to be built (from copies of the arguments) for processing by the object oriented methods. The overhead of calls, virtual or otherwise, is not dominated by the control flow alteration itself - but by the surrounding call convention costs, like prologue and epilogue code, stack setup and argument passing.
Table 1: Pseudocode comparison of procedural and object-oriented paradigm

<table>
<thead>
<tr>
<th>Procedural</th>
<th>Object-oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{triangle.areaT method(a,b)}:)</td>
<td>(\text{triangle.areaT method(a,b)}:)</td>
</tr>
<tr>
<td>Push stack (9)</td>
<td>Push stack (9)</td>
</tr>
<tr>
<td>Load (a); (10)</td>
<td>Load (a); (10)</td>
</tr>
<tr>
<td>Load (b); (11)</td>
<td>Load (b); (11)</td>
</tr>
<tr>
<td>(R=a=b); (12)</td>
<td>(R=a=b); (12)</td>
</tr>
<tr>
<td>(Ms=rs^2*0.5); (13)</td>
<td>(Ms=rs^2*0.5); (13)</td>
</tr>
<tr>
<td>Pop stack (14)</td>
<td>Pop stack (14)</td>
</tr>
<tr>
<td>Return; (15)</td>
<td>Return; (15)</td>
</tr>
</tbody>
</table>

Main proc:
\[ \begin{align*}
\text{Load } x & \quad 1 \\
\text{Load } y & \quad 2 \\
\text{Result} = \text{triangle.areaT}(x,y) & \\
\text{allocate heap storage} & \\
\text{copy } x \text{ to message} & \\
\text{copy } y \text{ to message} & \\
\text{load } p = \text{address of parameter list} & \quad 3 \\
\text{load } v = \text{address of subroutine} & \quad 4 \\
\text{I goto } v \text{ with return} & \quad 5 \\
\end{align*} \]

The advantages of procedural abstraction and object-oriented-style polymorphism are not well illustrated by a small example like the one above. This example is designed principally to illustrate some intrinsic performance differences, not abstraction or code re-use.

SubroutineiMethod call overhead

The presence of a (called) subroutine in a program contributes nothing extra to the functionality of the program regardless of paradigm, but may contribute greatly to the structuring and generality of the program. making it much easier to write, modify, and extend (Robens,2008). The extent to which different paradigms utilize subroutines (and their consequent memory requirements) influences the overall performance of the complete algorithm. Guy Steele (1977) pointed out that, a well-designed programming language implementation can have very low overheads for procedural abstraction (but laments, in most implementations, that they seldom achieve fins in practice - being "rather thoughtless or careless in this regard"). But Guy Steele also makes a considered case for mating procedure calls with tail recursion and concludes that "we should have a healthy respect for procedure calls" (because they are powerful). Roberts however, suggested we "use them sparingly" (Robens,2008).

In terms of the frequency of subroutine calls:

- for procedural programming, the granularity of the code is largely determined by the number of discrete procedures or modules.
- for object oriented programming, the number of method calls invoked is also partly determined by the granularity of the data structures and may therefore include many "read-only" accesses to low level objects that are encapsulated (and therefore accessible in no other, more direct, way). Since increased granularity is a prerequisite for greater code reuse, the tendency is towards fine-grained data structures,
Comparative Analysis of Object-Oriented Software Development and Classical Paradigm

and a corresponding increase in the number of discrete objects (and their methods) and, consequently, subroutine calls. The creation of "god objects" is actively discouraged. Constructors also add to the count as they are also subroutine calls (unless they are inlined). Performance problems caused by excessive granularity may not become apparent until salability becomes an issue.

ALLOCATION OF DYNAMIC MEMORY FOR MESSAGE STORAGE AND OBJECT STORAGE

Uniquely, the object oriented paradigm involves dynamic allocation of memory from heap storage for both object creation and message passing. A 1994 benchmark - "Memory Allocation Costs in Large C and C++ Programs" conducted by Digital Equipment Corporation on a variety of software, using an instruction-level profiling tool, measured how many instructions were required per dynamic storage allocation. The results showed that the lowest absolute number of instructions executed averaged around 50 but others reached as high as 61 (David, 1994). The above pseudocode example does not include a realistic estimate of this memory allocation pathlength or the memory prefix overheads involved and the subsequent associated garbage collection overheads. In the Java code ooadd below objAtld uses dynamic allocation from heap to be created and for the Add message passed to objAdd.Addl(). This increased the memory allocation cost of ooadd package when compared to the proadd.

DYNAMICALLY niisrxrcilEn MESSAGE CALLS V. DIRECT PROCEDURE CALL

OVER II EA DS

In their Abstract "Optimization of CMed-Orieided Prognims Using Sadie Class Hierareln. Analysis" Jeffrey 1)ean, David Grove, and Craig Chambers (2005) of the Department of Computer Science and Engineering at the University of Washington, claim that "Heavy use of inheritance and dynamically-bound messages is likely to make code more extensible and reusable, but it also imposes a significant performance overhead. compared loan equivalent but non-extensible program written in a non.object-oriented manner.

```java
package ooadd;
import java.util.Scanner;
Class AtIdTwo

private double A;
private double B;
public void setA(double x)

A = x;

public void &013(double y.)

B = y;

public double gctA0

return A;

public double getlatO

return B;
```

13$
public double Add()
{
    return A+131;
}

public class Main
{
    public static void main(String[] args)
    {
        double m, n;
        Scanner input = new Scanner(System.in);
        AddTwo objAdd = new AddTwo();
        System.out.println("Enter a value");
        n = input.nextDouble();
        System.out.println("Enter a value");
        m = input.nextDouble();
        objAdd.setA(n);
        objAdd.setB(m);
        System.out.println("Added Number = " + objAdd.Add());
    }
}

package proadd;
import java.util.Scanner;
public class Main
{
    public static void main(String[] args)
    {
        double m, n, s;
        Scanner input = new Scanner(System.in);
        System.out.println("Enter a value");
        n = input.nextDouble();
        System.out.println("Enter a value");
        m = input.nextDouble();
        System.out.println("Enter a value");
        s = input.nextDouble();
        System.out.println("Added Number = " + s);
    }
}

Some domains, such as structured graphics packages, the performance cost of the extra flexibility provided by using a heavily object-oriented style is acceptable. However, in other domains, such as inscription libraries, numerical computing packages, rendering libraries, and trace-driven simulation frameworks, the cost of message passing can be too great, forcing the programmer to avoid object-oriented programming in the "hot spots" of their application.

SERIALIZATION OF OBJECTS
Serialization imposes quite considerable overheads when passing objects from one system to another, especially when the transfer is in human-readable formats such as XML and JSON. This contrasts with compact binary formats for non-object-oriented data. Both encoding and decoding of the objects data value

136
Comparative Analysis of Object-Oriented Software Development and Classical Paradigm

and its attributes are involved in the serialization process (that also includes awareness of complex issues such as inheritance, encapsulation and data hiding).

RESULT AND DISCUSSION OF FINDINGS
In the test our units where an index used to represent a unit of measure of the actual value. For instance 200byte of memory is one unit while one instruction length is one unit. Reusability index where also used. The values from the test performed using the two simple programs for computing the area of a triangle and the adding of two values in variable a and b is shown in table 2.

<table>
<thead>
<tr>
<th>Measure aremeter</th>
<th>Procedural</th>
<th>Object-oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Path Let h</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>Memory Resources</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td>instruction Fetch</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Reusability</td>
<td>14</td>
<td>59</td>
</tr>
<tr>
<td>Binary Site</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From the plot of the rasa is presented in figure 3 it is clear that instruction path length or object-oriented paradigm is longer making procedural paradigm better in instruction path length. In the memory a:sources the procedural paradigm uses less memory resources than the object-oriented paradigm placing the procedural paradigm in a better place in themes of memory resources used in the execution of application instruction. The variation may even be worst for a much large application other than the one used in the experiment in this paper.

In instruction fetch it is also clear that the procedural paradigm is better than the object-oriented paradigm since the object-oriented paradigm requires more instruction fetch operations to complete a task as illustrated in the figure 3. In figure 4 the point shows procedural paradigm clearly below the object-oriented paradigm in the instruction fetch.

REUSABILITY AND BINARY SIZE
In code reusability is very clear that the reusability index of object-oriented paradigm is clearly higher than the procedural paradigm. This implies that there will be high effort geared towards rewriting code segment which cannot be reusable in new applications in procedural paradigm than in object-oriented paradigm. This has an over head effect of increasing development cost in procedure system development in most software engineering tasks than the object-oriented paradigm. The binary size of procedural paradigm is higher than the object-oriented paradigm this can be easily explained due to the referencing of more physical instructions that may increase cache 110SSCS in the system.
RECOMMENDATION

It is clear that from the results presented object-oriented paradigm is best of large projects that involve large coding effort and repeated instruction flows. It is also beneficial for tools and code types that are repeatedly used in subsequent applications such as menus, tool bars, windows actions and other sections of applications that are often reoccurring in many applications. However when the application is involving high computational base systems the procedural paradigm will be better deployed due to its shorter instruction path length and instruction fifths. But what happens if it is a software engineering application involving large system and a lot of computational algorithms? We recommend that an object-oriented paradigm be deployed with procedural techniques used in all computational methods of the system. We also advise that the main computational variables should not be made part of the class fields (except they involve security) rather they should be localized within the method where they are being used.

CONCLUSION

In conclusion we have comparatively analyzed the object-oriented software engineering paradigm and what classical paradigms such as the procedural paradigm. We have also identified its strength and weaknesses based on certain parameters considered in our study. Our study may not be exclusive but it highlighted certain points that make or may mar the paradigms from a simplified angle using two highly simplified code. The finding lead us to make powerful software engineering recommendation based on facts deduced from our result. Our limitation however, is that we did not test this facts with very large applications to check the multiplier effect if it is actually larger than we have expected in this research. However our work serves a guide towards such finding.

REFERENCES


