JASEM ISSN 1119-8362

Full-text Available Online at www.bioline.org.br/ja

J. Appl. Sci. Environ. Manage. *December*, 2008
Vol. 12(4) 41 - 46

A Comparative Analysis of Structured and Object-Oriented Programming Methods

ASAGBA, PRINCE OGHENEKARO; OGHENEOVO, EDWARD E. CPN, MNCS.

Department of Computer Science, University of Port Harcourt, Port Harcourt, Nigeria.
pasagba@yahoo.com, edward_ogheneovo@yahoo.com. 08056023566

ABSTRACT: The concepts of structured and object-oriented programming methods are not relatively new but these approaches are still very much useful and relevant in today's programming paradigm. In this paper, we distinguish the features of structured programs from that of object oriented programs. Structured programming is a method of organizing and coding programs that can provide easy understanding and modification, whereas object-oriented programming (OOP) consists of a set of objects, which can vary dynamically, and which can execute by acting and reacting to each other, in much the same way that a real-world process proceeds (the interaction of real-world objects). An object-oriented approach makes programs more intuitive to design, faster to develop, more amenable to modifications, and easier to understand. With the traditional, procedural-oriented/structured programming, a program describes a series of steps to be performed (an algorithm). In the object-oriented view of programming, instead of programs consisting of sets of data loosely coupled to many different procedures, object-oriented programs consist of software modules called objects that encapsulate both data and processing while hiding their inner complexities from programmers and hence from other objects. @ JASEM

Structured programming can be viewed as the pulling together, or synthesization of such ideas as program modularity and top down design, and the concrete representation of them at the program-coding level. It is a manner of coding and organizing programs that makes them easier to understand, to test and to modify. Results have demonstrated that employed together with other improved programming technologies, can lead to spectacular increases in programmer productivity and correspondingly spectacular decreases in the error rate of resultant code (Champeaux, 1990), and (Istatkova, 2001). Structured programming methodology tries to resolve the issues associated with unconditional transfers to enable programmers follow the logic of programs.

Much of a program's complexity arises from the fact that the program contains many jumps to other parts of the programs - jumps both forward and backward in the code. Furthermore, as a program undergoes change during its development period, as it gets further debugged during its maintenance period, and as it gets modified in subsequent new projects, the complexity of the program grows alarmingly. New jumps are inserted, thus increasing the complexity. In some cases, new code is added because the programmer cannot find existing code that performs the desired function, or is not sure how the existing code works, or is afraid to disturb the existing code for fear of undoing another desirable function, and the result, after many modifications, is a program that is nearly unintelligible. This is the software equivalent of being shop-worn, the time when it is better to throw the whole thing out and start over (Louden, 1993), and (Owolabi, et al, 2005).

Indeed, structured programming concepts discourage the use of 'GO TO' statements and encourage program blocks, modularity, top-down design approach and reusability amongst others. Programs written with the structured approach are more readable and more reliable. Also, the cost and time of developing software is less when structured programming is adopted since smaller units of programs can be written independently (sometimes by different, individuals or groups) and then combined to achieve the desired end product. The art of programming is made flexible by structured programming especially in the area of program or software maintenance. Programs can easily be modified and updated to suit prevailing circumstances (Louden, 1993).

In structured programs, any function can be performed using one or a collection of three control structures: sequence, selection, and repetition as shown in Fig. 1.

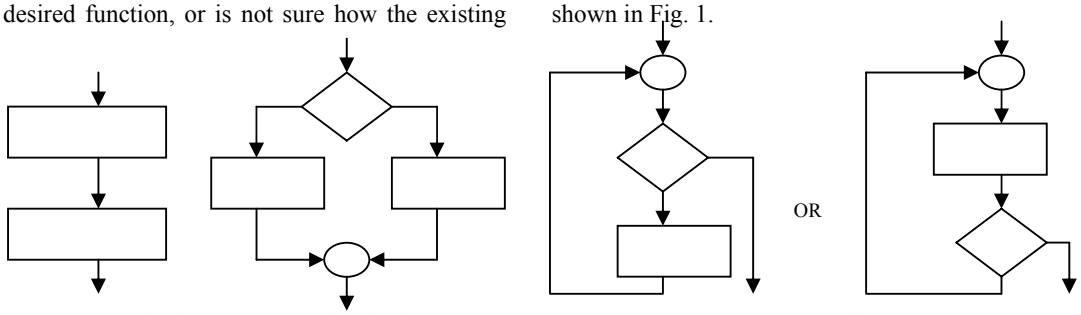

Fig 1. (a) Simple sequence (b) Selection (c) Repetition

* Corresponding Author: Asagba, Prince Oghenekaro

These control structures are quite adequate for any kind of processing, or any combination of decisions, or any type of logic manipulations without exhibiting back-tracking. *Pascal, PL/I, Ada,* and *ALGOL* are perhaps some of the better known structured programming languages.

Object-oriented programming (OOP) is a *programming paradigm* that uses "*objects*" and their interactions to design applications and computer programs. Programming techniques may include features such as *information hiding, data abstraction, encapsulation, modularity, polymorphism*, and *inheritance*. It was not commonly used in mainstream software application development until the early 1990s. Many modern *programming languages* now support OOP (Wikipedia, 2008). Some of the better known OOP languages are C++, Object Pascal, and, Java.

Modular Programming

Many programs can be decomposed into a series of identifiable subtasks. It is a good programming practise to implement each of these subtasks as a separate program module. The idea of modular programming is to sub-divide a program into smaller units that are independently testable and that can be integrated to accomplish the overall programming objective (Abott, 1993). The use of modular programming structure enhances the accuracy and clarity of a program, and it facilitates future program alterations (Louden, 1993).

One motivation for modularizing a program into methods is the divide-and-conquer approach, which makes program development more manageable by constructing programs from small, simple pieces. Another is software reusability - using existing methods as building blocks to create new programs. Often, you can create programs mostly from standardized methods rather than by building customized code. A third motivation is to avoid repeating code. Dividing a program into meaningful methods makes the program easier to debug and maintain (Champeaux, 1990), and (Deitel, et al, 2007).

Top-Down Approach

When developing a new program, the overall program strategy should be completely planned out before beginning any detailed programming. This allows you to concentrate on the general logic, without being concerned with the syntactic details of the individual instructions. Once the overall program strategy has been clearly established, the details associated with the individual program statements can be considered. This approach is generally referred to as "top-down" programming. With large programs, this entire process might be repeated several times with more programming details added at each stage (Louden, 1993).

In top-down design, the main program is first defined and then the remaining modules or units are specified. The central idea in top-down programming is that the design must progress from the general to the specific, each program unit being progressively refined. Usually, the main modules drives or co-ordinates the other modules specifying what each subprogram should do. It is also expected that the main module be the interface between the entire program and users. The hierarchical relationships existing among modules of a program are often displayed in a structure chart. This chart conveys the sense of orders and module or task is represented with a rectangle and modules are sub-divided at each level until they can no longer be sub-divided further. Fig. 2 shows a structure chart.

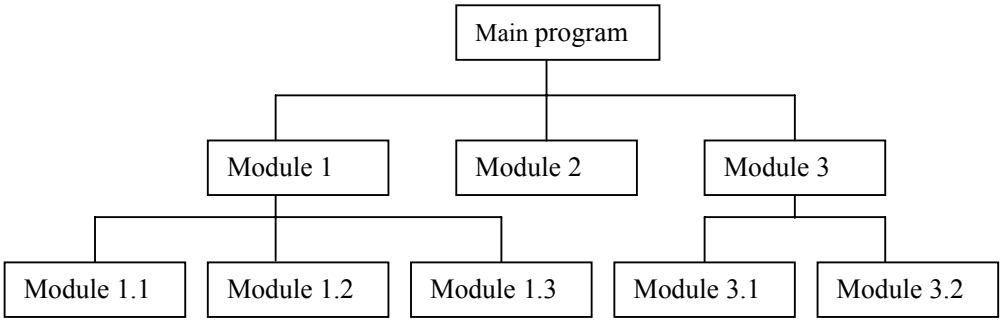

Fig. 2 Structure chart

As seen from the structured chart above, the main program is at level 0. This main program is divided into three modules as the application requirements grow. This is a major advantage of modular programming. The top-down design methodology often employs a process referred to as stepwise refinement or divide-and-conquer whereby the situation is progressively refined till the lowest level in the structure chart is reached. This process of step-wise refinement is also very often applied to the specification of the lower level procedures.

Top-down design is often referred to by other names - structured design, composite design, programming by stepwise refinement, and so on. Though the names

* Corresponding Author: Asagba, Prince Oghenekaro

differ, a uniform approach is generally agreed upon: we identify first the major function to be accomplished, then we identify its sub-functions, their sub-functions, and so on, proceeding from the major function to any number of lesser functions until we are satisfied that we fully understand the nature of our solution algorithm. The top-down design process consists of a series of steps to define the functions required for the solution of a problem, in terms of the problem itself (Pressman, 2005).

Bottom-up Design

This method may be useful for programs that make use of independent program modules (that is, user-defined procedures and functions). The bottom-up approach involves the detailed development of these program modules early in the overall planning process. The overall program development is then based upon the known characteristics of these individual modules (Owolabi, et al, 2005). The bottom-up design is the opposite of top-down design. It involves writing a modular program from specific to general. That is, modules are built up from the least level upward until the general solution is obtained. This is not a very acceptable methodology in modular programming. However, it is a useful design method when the task at hand involves just the modification and updating of an already existing program to obtain the needed result.

Object Oriented Programming

In some time past, language design was often based on the size of the programs, which were generally small, however, when programs became very large, the focus changed. In small programs, the most common statement is generally the assignment statement. However, in large programs (over 10,000 lines), the most common statement is typically the procedure-call to a subprogram. Ensuring parameters are correctly passed to the correct subprogram becomes a major issue. The concept of object-oriented analysis (OOA) is to define all classes (and the relationships and behaviour associated with them) that are relevant to the problem to be solved (Biddle, et al, 1994), (Booch, 1986), (Pressman, 2005), and (Istatkova, 2001). A number of small programs can be handled using hierarchical structures. However, in large programs, the organization is more of network structures.

Although, structuring a program into a hierarchy might help to clarify some types of software, even for some special types of large programs, a small change, such as requesting a user-chosen ripple-effect with changing multiple subprograms to propagate the new data into the program's hierarchy. The object-oriented approach is allegedly more flexible, by separating a program into a network of subsystems, with each controlling their own data, algorithms, or devices across the entire program, but only accessible by first specifying named access to the subsystem object-class, not just by accidentally coding a similar global variable name. Rather than relying on a structured-programming hierarchy chart, object-oriented programming needs a call-reference index to trace which subsystems or classes are accessed from other locations (Hubbard, 2000).

The state of an object in an object-oriented language is primarily internal, or local to the object itself. That is, the state of an object is represented by local variables declared as part of the object and inaccessible to components outside the object. Secondly, each object includes a set of functions and procedures through which the local state can be accessed and changed. These are called methods, but they are similar to ordinary procedures and functions, except that they can automatically access the object's data (unlike the "outside world") and therefore can be viewed as containing an implicit parameter representing the object itself. Calling a method of an object is sometimes called sending the object message.

Objects can be declared by creating a pattern for the local state and methods. This pattern is called a class, and it is essentially just like a data type. Indeed, in many object-oriented languages, a class is a type and is incorporated into the type system of the language in more or less standard ways. Objects are then declared to be of a particular class exactly as variables are declared to be of a particular type in a language like C or Pascal. An object is said to be an instance of a class.

The central concept of object-oriented programming is the object, which is a kind of module containing data and subroutine. An object is a kind of self-sufficient entity that has an internal state (the data it contains) and that can respond to message (calls to its subroutines). A student-records object, for example, has a state consisting of the details of all registered students. If a message is sent to it telling it to add the details of a new student, it will respond by modifying its state to reflect the change. If a message is sent telling it to print itself, it will respond by printing out a list of details of all registered students.

The object-oriented programming approach to software engineering is to begin by identifying the objects involved in a problem and identifying the messages that those objects should respond to. The solution that results is a collection of objects, each with its own data and its own set of responsibilities. The objects interact by sending messages to each other ([Louden, 1993).

Properties of OOP

The following properties are exhibited by OOP: Data abstraction, encapsulation, inheritance, and polymorphism.

* Corresponding Author: Asagba, Prince Oghenekaro

Data Abstraction

Data abstraction is a methodology that enables us to isolate how a compound data object is used from the details of how it is constructed from more primitive data objects (Ryan, 2000). Data abstraction is the enforcement of a clear separation between the abstract properties of a *data type* and the concrete details of its implementation. Data Abstraction is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem. Abstraction is also achieved through *Composition*. For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to *interface* with them, that is, send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other (Wikipedia, 2008).

Encapsulation

Encapsulation is the ability to package codes and data together in a place and hide (or prevent) that data from external contact thereby forcing anyone who wants to access it to pass through the associated code. Structured programming encourages code everywhere to deal directly with data structures.

Inheritance

This is the ability of an existing class to create new classes. Thus existing class is referred to as a base class and the newly created classes are called derived class. The derived class inherits all the features inherent in the base class. Inheritance is perhaps one of the most powerful features of object-oriented programming paradigm. Inheritance can support program (or software) reuse, reliability, and modification of the base class (Asagba, 2002). Inheritance is a powerful programming tool and it supports reusable component. Inheritance establishes a parent-child dependency relationship between objects in a class. The inheritance graph is a tree. A single inheritance is a case when each derived class can inherit from only one base class, whereas a multiple inheritance is a case in which a class may inherit from two or more base classes. Newer object-oriented languages such as Java and C++ provide multiple inheritances. In a language with multiple inheritances, its graphs can be acyclic instead of a tree. Multiple inheritances can be useful but its approach can be complex. One issue is that methods may be inherited in more than one way. For instance, a method from class A is inherited by class D in two separate ways. Fig. 3 shows multiple inheritances graph.

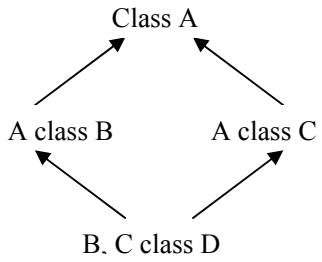

Fig. 3 Multiple inheritances graph

Polymorphism

Polymorphism is a mechanism that allows objects of different types to respond differently to the same function call. Overloading and template can be considered primitive polymorphisms because the decision of invoking a particular function is made at compile time rather than at run time. At compile time, the exact nature of some objects cannot be determined. Such objects had to be delayed until run time where decisions on which function to invoke (or call) will be available. This is a technique that brings out true polymorphism (Asagba, 2002). Polymorphism is the ability to identify certain aspects that several data types have in common, and write code that works equally well with all of them by ignoring the differences in situations where they do not matter (Biddle, el al, 1994).

Other Differences of Structured Programming and Object-Oriented Programming

Structured programming is task-centric while object-oriented programming is data-centric, that is, structured programming is based around data structures and subroutines.

Object-oriented programming, on the other hand shifts your primary attention to the data itself. Instead of asking "what do I want to do and what will I need to know to do it", you ask "what kind of things do I want to have and what can those things do for me". Instead of designing your functions first and then coming up with data structures to support them, you design types first and then come up with the operations needed to work them (Booch, 1986), and (Liang, 2001).

* Corresponding Author: Asagba, Prince Oghenekaro

Again, object-oriented programming is a superset of structured programming. A pseudo code of a structured programming is as follows:

```
... Program start
var
var
var
function {...}
function {...}
function {...}
main {...}
... Program End
```

Here, you have units of code which operate on variables and are called in references to those variables, to follow a structure, acting on those variables.

A pseudo code of an object oriented programming is as follows:

```
... Program start
Object {
Var
Var
function {...}
function {...}
function {...}
}
var
var
function {...}
main [...}
... Program End
```

Variables can be objects, which have their own data and functions. Thus, instead of referencing a function (a block of code) and telling it to operate on a variable q, you reference an object and tell it to perform an operation, most often on itself, specific to itself, using its own data. Instead of creating units on them, you create objects and have them perform operations (on themselves) (Booch, 1986).

Similarities between Structured Programming and Object oriented Programming

Both structured programming and OOP require rudimentary understanding of programming concepts and basic control flow. Loops, conditional statements, and variables are concepts that are important whether you are using a procedural language or an OOL.

Conclusion: In this paper, we have discussed the concepts of structured programming and object-oriented programming and pointed out the similarities and differences between them. We have pointed out that object-oriented programming is an approach to software design that facilitates rapid development of complex applications and software reuse. Object-oriented language is developed from the necessity to organize the programming process into a language. We also pointed out that object-oriented programming is a technique of writing programs using objects. Object-oriented programming languages provide general mechanisms for building software modules whose behaviour can be customized or specialized.

Traditionally, programmers would write programs that were called structured programs. The program would be designed to solve one big problem, but the programmers would break the problem down into smaller, more manageable problems and write small sections of code to solve each one. Object-oriented programming is the natural successor to this traditional way of programming. Instead of simply breaking the problem down into smaller problems, object-oriented programmers break the problem down into objects, each with a life of its own. The programmer then has to figure out what properties an object needs to function, and the methods necessary to bring it to life. Like most interesting new developments, object-oriented programming builds on some old ideas, extends them, and puts them together in novel ways. The result is many faceted and a clearer step forward for the art of programming.

With the traditional, procedural-oriented/structured programming, a program describes a series of steps to be performed (an algorithm). In the object-oriented programming, instead of programs consisting of sets of data loosely coupled to many different procedures, object-oriented programs consist of software modules called objects that encapsulate both data and processing while hiding their inner complexities from programmers and hence from other objects. This can make object-oriented programs more flexible and easier to maintain.

Finally, in terms of similarities, both require rudimentary understanding of programming concepts and basic control flow. Loops, conditional statements, and variables are concepts that are important whether you are using a procedural language or an OOL.

REFERENCES

Abott, R. (1993), Program Design by Informal English Descriptions, CACM, Vol. 26 No. 11, pp. 892 - 894.

Asagba, P. O. (2002), Understanding C++ Programming, Port Harcourt, Gitelle Press (Nig.) Ltd., pp. 157 - 175.

Biddle, R. L. and Tempero, E. D. (1994), Teaching C++ Experience at Victoria University of Wellington. In Proceedings of Software Education Conference, Dunedin, New Zealand, pp. 274 - 281.

* Corresponding Author: Asagba, Prince Oghenekaro

Booch, G. (1986), Object-Oriented Development, IEEE Trans. Software Engineering, Vol. SE -12, No. 2, pp. 211.

Cashman, M., (1989), Object-Oriented Domain Analysis, ACM Software Engineering Notes, Vol. 14, No. 6, pp. 67.

Champeaux, D. D. (1990), Panel: Structured Analysis and Object Oriented Analysis, ECOOP/ OOPSLA '90 Proceedings, pp. 135 - 139.

Deitel, P. J. and Deitel, H. M. (2007), Java How To Program, USA, Pearson Inc., 7th Ed., pp. 421 - 423.

Hubbard, J. R. (2000), Programming with C++ Schaum's Outlines, New York, McGraw-Hill Companies, Inc., pp. 273 – 299.

Istatkova, G. (2001), Algebra of Algorithms in Procedural and Object-Oriented, structured Programming, Automatica & Informatics, Vol. 3, No. 4, pp. 56 - 62.

Liang, Y. D. (2001), Introduction to Java Programming, New Jersey, Prentice-Hall, Inc, pp.

Louden, K. C. (1993), Programming Languages: Principles and Practice, Boston, PWS Publishing Company, pp. 300 - 345.

Owolabi, O, and Ndeekor, C. B. (2005), Structured Programming with Pascal, Aba, Granite Ventures Nig. Ltd, pp 19 - 21.

Pressman, R. S. (2005), Software Engineering: A Practitioner's Approach, New York, McGraw-Hill International 6th Ed., pp. 217 - 218.

Ryan, B (2000), Introduction to Data Abstraction, http://mitpress.mit.edu/sicp/full-text/sicp/book/node27.html, last accessed in Oct., 2008.

Wikipedia (2008), the Free Encyclopedia, http://en.wikipedia.org/wiki/Object-oriented_programming, last accessed in Oct., 2008.

* Corresponding Author: Asagba, Prince Oghenekaro