Object Oriented Programming

Bhanu Prasad Pokkunuri

ES Group
Central Electronics Engineering Research Institute
P.B. No. 1, Pilani, Rajasthan
INDIA - 333 031

ABSTRACT

Object Oriented Programming (OOP) is being used quite widely in the fields of both software engineering and artificial intelligence. The aim of this article is to give an idea of OOP and its features. It includes an indication of the available OOP environments and the Object-Oriented extensions made to the conventional languages such as C and Pascal. Some of the advantages in the use of OOP are also described briefly.

1. Introduction

Ever since the invention of 'Analytical Engine' by Charles Babbage in 19th century, there have been a variety of hallmarks in the information and computer technology. The modern digital computers have undergone four generations of technology in terms of the basic mechanisms of implementation, viz., the vacuum tubes, silicon transistors, Integrated Circuits, and Very Large Scale Integration (VLSI). Similarly there have been improvements in terms of the medium that is used for programming the computers, viz., machine code, assembly language, High Level Languages (HLL's), and Fourth Generation Languages (4GL's). Simultaneously, there has been the emergence of various approaches in the design of programs, viz., top_down, bottom_up, structured, modular, etc., and different programming paradigms, viz., functional/procedural programming, logic programming, object oriented programming, access-oriented programming, constraint based programming, etc.

Each approach and methodology has its advantages and disadvantages and hence can not be seen as universal. So there have been experiments with a fair degree of success to combine more than one paradigm in a single environment and reap the advantages provided by all such ingredient paradigms. The technique of using a combination of paradigms is getting wider acceptance both by the developers and the users of such programming environments.

1.1. Why Object-Oriented Programming?

The flexibility and the support provided by the conventional high level languages such as FORTRAN, ALGOL, Pascal, etc., is quite considerable when compared to the earlier generations of machine code and assembly language. But when larger programs are to be developed and maintained, these HLL's also succumb to the complexity and cannot support such applications successfully and easily.

To illustrate this let us consider a simple example:[1] i.e. programming an array of items, each potentially of a different type, and printing out the contents of the array. The solution in a conventional procedural language will have a procedure similar to the following one:

```
Procedure Print(item:polytype);
  Begin
  Case item_tag of
      inttag: Printint(item);
      chartag: Printint(item);
      stringtag: Printstring(item);
      otherwise: error('Type error: can't print this');
  end case
endproc;

for i:= 1 to MAX do Print(A[i]) ;
```

In this program 'polytype' is an union of all the possible types that may go into the array; and can be printed. If at a later date, another type element needs to be added into the array A, it is necessary to redefine the 'polytype', the 'Print' procedure and all other procedures that perform operations on items in the array A. In general, a simple addition of another type element can have complex ramifications all through the application and requires an extensive editing, re-compilation and may put back to the edit-compile-test cycle(s).

There will be a number of such occasions in most of the large applications. The activity of updating and maintaining such changes turns out to be a nightmare. Such applications have forced the different groups to look for alternatives. This search has one result in the form of object-oriented programming methodology, which is quite different from the functional or procedural approach followed by the conventional high level languages.

1.2. What is Object-Oriented Programming?

It is one of the programming paradigms or methodologies used in the programming languages. Each programming methodology places emphasis on some aspects or concepts in the programming effort. In the Object-Oriented Programming the attention gets focused on 'objects' - i.e. on their properties and the behaviour or interaction with other objects[2,3,4,5,6,7,8,9] .

In this approach 'object' is the primitive element. Each object combines attributes of procedures and data. It stores the data in variables and responds to messages by carrying out the procedures (methods). So each object can be viewed as an abstract data type. There is a flexibility to have a variety of data types and data structures. This contrasts with 'Data-structure-oriented programming' wherein a single powerful data structure; such as list (LISP), array (APL), set (SETL) or relational database (SQL); is utilised. Object Oriented Design/Development (OOD), is an approach to software design in which the modular decomposition of a system is based on the classes of objects the system manipulates, not on the function the system performs. Meyer[10] further refines OOD as "the construction of software systems as structured collections of abstract data-type implementations".

Each object is an autonomous entity with its private data and methods. Its behaviour is characterised by the actions that it suffers and that it requires of other objects. Data being private to the object, the important responsibility of selecting the compatible operator is thrust upon the object - the supplier of the service. This contrasts the style of conventional programming wherein the consumer of the service has to select compatible operator required for the data on hand.

In the conventional programming, if a data-structure is to be printed, the print function or subroutine compatible to the data-structure is to be selected by the consumer. Hence the consumer has to keep track of the various functions, subroutines and the minor variations among them, so as to select the proper one. In OOP this is done by sending a simple message to the corresponding object; probably owning the data-structure; to provide the service of printing the data-structure. The supplier, the object, selects the compatible method and provides the service. This is the primary difference of OOP from the conventional programming, which brings a variety of advantages.

Object oriented programming is undergoing an evolution. The various features of OOP cannot be put in a simple and formalised definition. Hence the basic and important features of OOP are described in the following section.

2. Features of Object-Oriented Programming

2.1. Information hiding

Each object is an integral and autonomous entity. It has got the required resources to manifest its state and behaviour. It can be viewed as a shielded and encapsulated entity. Its data is private to itself and is hidden from others. Its interaction with other objects and the outside world is through its response to the messages sent to it. Thus each object has limited view of and by other objects., which is defined in the object itself. This restricts the unwanted effects due to changes in specifications, design, etc.

2.2. Data abstraction

Data abstraction is the principle that states that programs should not make assumptions about implementations and internal representations. In OOP any object can be requested for any required information or service by sending the suitable messages. The requester is not bothered about how the object provides the information or service. It gets **what** it asked for. The **how** part, i.e. the implementation or information generation is internal to the object, the supplier. Thus the emphasis is on **what** rather than **how** when the interaction among objects is concerned. This provides the programmer an opportunity to think in terms of problem specification rather than be taken away by the implementation details.

2.3. Dynamic binding

In the conventional programming languages; the operators, functions, etc. are bound to their respective operations at the time of compilation. This is called static binding. In such cases each operator has got, in general, an unique name and an unique operation. The language 'Ada' is an exception to this due to its 'operator overloading' feature, which enables a single operator to have more than one operation. But in the case of 'Ada' also, the address corresponding to the operation/procedure invoked is fixed at compile time itself.

In object oriented programming, the binding of operator to a particular operation takes place at run time. The operator/message 'print' to an object invokes the operation/method specific to that object, as decided by the object at run time. This is called dynamic binding. Naturally the same message 'print' sent to different objects, activates different methods. Thus the single message 'print' has got different responses, which is termed **polymorphism**[11] .

2.4. Inheritance

In conventional languages, the addition of new types, requires the common routines such as 'Print' to be rewritten, as described in section 1.1. This is overcome in OOP by means of inheritance. The objects are organised into classes and the common operations/methods are associated with the suitable classes. New objects are created as instances of the classes which inherit all the properties of the class. Specialisations of classes are made as subclasses which also inherit the properties. The subclasses or child classes can redefine the methods inherited from the super-class/parent-class. They can add new methods to distinguish themselves and have special behaviour. Inheritance need not get limited to one parent class. The child class can inherit from different parent classes. This is called **multiple inheritance**.

Thus the openness of classes is a fundamental advantage of object oriented languages over the closed modules of languages such as Ada and Modula-2. Thus inheritance lends itself for **reusability** of code., as the common operations defined and implemented in a class can be accessed by all the classes and instances down the whole inheritance network. It also helps in effecting the required changes and the maintenance during and after the evolution of an application.

By making a change in a super class, in effect all its sub_classes are changed at once, without the tedium of identifying and implementing the same changes at a number of places. If changes are made in one sub-class to get distinctions, the code in the super class and the other sub-classes remains safe[12] .

3. Differences in implementations of OOP

The object oriented programming approach has been implemented at different places in different programming languages and environments. Simula is widely noted to be the forerunner in defining and implementing some of the similar and related concepts as that of OOP. *Smalltalk* is perhaps the purest form of OOP as it uses no other formalism other than that of OOP[2] . The OOP approach has been incorporated in the conventional languages as extensions such as object-Pascal[13] for Pascal, C++[14,15] and Objective-C[8,16] for C. These extensions have enhanced the power of the conventional procedural languages, in that users have the flexibility to use the object-oriented extension or the usual procedural facilities provided in the language or a judicious combination of both. These extensions provide a handle to the programmers, who have considerable experience with conventional procedural

languages, to explore the different concepts and their uses provided by the object-oriented approach without leaving their home base.

There is another line of development towards object oriented programming that stems from the idea of frames as proposed in[17] . This has been the basis for a variety of knowledge representation languages and environments such as *KEE*, *Loops*, *KRL*, *MUSE* etc. The basic concept is that knowledge about a part of real world can be held in a frame in terms of different slots attached to it. The slots contain the default values to represent prototypical behaviour, and/or might also point to other frames.

The objects in most object languages are divided into two categories: classes and instances. A class is a description of one or more similar objects. The class is similar to 'type' in the conventional procedural languages. For example if 'integer' is a type (class), then '5' is an instance of type (class) 'integer'. Similarly 'Leopard-1' and 'Leopard-2' can be two instances of a class 'Leopard'. Classes and instances have a defined declarative structure in terms of object variables for storing state, and methods for responding to messages. There can be two categories of object variables: class variables and instance variables. Class variables are variables stored in the class whose values are shared by all instances of the class. Instance variables (sometimes called slots) are variables for which local storage is available in the instances. Since all the instances of a class share the same methods, any difference in response by two instances is determined by a difference in the values of their instance variables.

There are variations in implementing the different concepts of OOP. The basic concepts such as information hiding, data abstraction, dynamic (late) binding, inheritance, multiple-inheritance, operator overloading, automatic garbage collection, etc., are all supported in the *Smalltalk 80*. C++ does not support multiple inheritance, operator overloading and automatic garbage collection; whereas Objective-C does not support operator overloading[8] .

There are few more characteristics of variation in the implementations of different object oriented languages and environments. *Smalltalk* and *Loops* have indexed instance variables, thus allowing some instances to behave like dynamically allocable arrays. Further in *Smalltalk*, *Loops* and also in *KEE*, the class variables and instance variables are distinct. *KEE* provides this distinction by means of 'own' and 'member' declarations for slots[18] . *Flavors* does not have class variables.

Another difference concerns the access of instance variables. *Smalltalk* follows strict encapsulation and does not allow access to instance variables from other than a method of the object. *Loops* allows direct access to object variables to support a knowledge representation style of programming[19] . This is useful in writing programs that compare two objects. Similarly property annotations for variables are not provided in all object languages. *Smalltalk*, *Flavors* and *Object-Lisp* do not provide this feature. In these languages the variables have just values. *Loops* and *KEE* have this feature and hence any auxiliary information such as dependency records, documentation, history of past values, constraints, etc., can be attached as annotations on variables. This feature is quite useful in development, testing and maintenance.

4. Advantages and disadvantages of OOP

4.1. Modularity

Since the objects are autonomous entities and share their responsibilities only by executing methods relevant to the received messages, each object lends itself to greater modularity in the whole system. The behaviour of each object is defined and implemented in the object itself and is independent of the implementation of other objects. Co-operation among different objects, to achieve the system operation, is done through exchange of messages[20] . This independence of each object eases the development and maintenance. Further most of the data being available locally in its variables/slots, the usual burden of passing a number of parameters in procedural languages gets greatly reduced.

4.2. Deferred Commitment

The object oriented concept is the ultimate in the deferred commitment. The internal working of object can be redefined even when rest of the system is actually running, and without changes to

other parts of the system. This heavily contrasts to the highly typed languages such as Pascal wherein everything is committed at compile time itself.

4.3. Defaults

The concept of default state of an object has come from Minsky's work on frames. The object's variables/slots have default values in the absence of better information. That is even when an object is not currently active or has not participated yet in the system effort, it has got a defined and meaningful state to start with.

4.4. Conceptual pleasantness

Object oriented design involves the identification and implementation of different classes of objects and their behaviours. The objects of the system closely correspond and relate in a one-to-one way to that of the real world. Thus it is easier to design and implement the system consisting of objects, as is observed and understood by the brain. The effort required in the conventional procedural languages, for the translation or transformation of the facts perceived by the brain into some other form suitable to the system, gets greatly reduced with OOP.

4.5. Re-usability

Inheritance has got a great role in object oriented programming. As the classes inherit from superclasses, the code developed in super classes gets reused. There is a concept of 'mixins', which are classes containing special code. These 'mixins' are not instantiated, but they are only inherited by other classes. *Smalltalk*, *Loops*, etc., provide a lot of class libraries and browsers to browse through the libraries. The user/developer need only to inherit from relevant classes and generate specialisations suiting the application without bothering about the implementation details of the basic operations. This puts him/her at an higher abstraction and advantage.

4.6. Different approach

The object oriented design is a completely different approach to problem analysis and decomposition to that of the functional or procedural approach. It uses the most important data structures as the basis for modularisation and attaches each function routine to the data structure to which it applies closely. But in the procedural approach the modularisation is on the basis of actions and operations as implemented in procedures or functions. This shift in the outlook poses a problem to those many programmers and system analysts that have considerable experience in conventional programming. As noted in earlier section, the conventional languages having object oriented extensions would be helpful in getting to grips with the new approach also.

OOP resembles the real world more closely than the conventional procedural programming. OOP has been recognised and sought after more vigorously in the eighties, as structured design was sought in seventies. The combination and coexistence of the OOP and knowledge engineering paradigms such as rules and frames brings each other greater efficiency and credibility. This combination will be extensively exploited in a growing number of applications.

References

1. Cook, S, "Languages and Object-Oriented Programming," *Software Engineering Journal*, pp. 73-80, March, 1986.
2. Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
3. Ramamoorthy, C.V. and Sheu, P.C., "Object-Oriented Systems," *IEEE Expert*, pp. 9-15, Fall 1988.
4. Micallef, J., "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages," *JOOP*, vol. 1, No. 1, pp. 12-35, Apr/May 1988.

5. Booch, G, "Object-Oriented Development," *IEEE Trans. on Software Engineering*, vol. SE-12, pp. 211-221., Feb, 1986.

6. Wegner, P., "Classification in Object-Oriented Systems," *SIGPLAN Notices*, vol. 21, No. 10, pp. 173-182, Oct 1986.

7. Cox, B.J., "Message/Object Programming: An Evolutionary Change in Programming Technology," *IEEE Software*, vol. 1, No. 1, pp. 50-61, Jan 1984.

8. Cox, B.J., *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wessley, 1986.

9. Rentsch, T., "Object-Oriented Programming," *SIGPLAN Notices*, vol. 17, No. 9, pp. 51-57, 1982.

10. Meyer, B, "Reusability: The Case for Object-Oriented Design," *IEEE Software*, pp. 50-64, March, 1987.

11. Pascoe, G.A., "Elements of Object-Oriented Programming," *Byte*, vol. 11, no. 8, pp. 139-144, 1986.

12. Tesler, "Programming Experiences," *Byte*, vol. 11, no. 8, pp. 195-210, 1986.

13. Tesler, "Object Pascal Report," *Structured Language World*, 1985.

14. Stroustrup, B, *The C++ Programming Language*, McGraw-Hill, 1986.

15. Weiner, R.S., "Object-Oriented Programming in C++ - A Case Study," *SIGPLAN Notices*, vol. 22, No. 6, pp. 59-68, June 1987.

16. Cox, B.J., "Object-Oriented Programming in C," *UNIX Review*, pp. 67-71 & 76, Oct/Nov 1983.

17. Minsky, M., "A Framework for Representing Knowledge," in *Winston, P.H. (Ed.) 'The Psychology of Computer Vision'*, pp. 211-277, McGraw-Hill, New York, 1975.

18. Fikes and Kehler, "The Role of Frame-Based Representation in Reasoning," *Comm. of ACM*, vol. 28, no. 9, pp. 904-920, 1985.

19. Stefik, M. and Bobrow, D., "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, vol. 6, No. 4, pp. 40-62, 1986.

20. Littins, M, "The Role of Object-Oriented Programming in Knowledge Engineering," *Proc. Conf. on Knowledge Based Systems*, pp. 249-260, On-line publ., Pinner, U.K., 1986.