Object Oriented Programming

Tim Rentsch
Computer Science Department
University of Southern California
Los Angeles, California 90089-0782

Recently I had the opportunity to read the Intel manual introducing the iAPX 432 architecture. The manual is an amalgam of an informal overview of the hardware, a quick tutorial review of computer architectures in general, an introduction to the advanced concepts of the 432 in particular, and marketing hype proclaiming the advantages of the 432 over its competition. The 432's architecture has many interesting, unusual characteristics, and Intel is wise to introduce them in a form which is quickly assimilable. What most caught my interest is the claim that the 432 supports the "object oriented design methodology."

What is object oriented programming? My guess is that object oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.

Surely there's a better way. I hope this paper will clarify object oriented programming by characterizing its more important aspects. These characterizations, while not a definition *per se*, should at least provide a better understanding, and ideally a commonly held one. Thus I hope we can avoid the deplorable and confusing situation just described.

History

The immediate ancestor of object oriented programming is the programming language Simula. The Smalltalk programming system carried the object oriented paradigm to a smoother model. Although other systems have definitely shown some object oriented tendencies, the explicit awareness of the idea – including the term "object oriented" – came from the Smalltalk effort. Furthermore, Smalltalk is still the strongest representative of object oriented programming in the sense of being the most unified under a single (object oriented) paradigm. Comments on some related efforts will be given in a later section. For now it is enough to note that object oriented programming arose when Smalltalk was developed and therefore the history of Smalltalk serves as the history of object oriented programming.

Smalltalk is the software half of an ambitious project known as the Dynabook. The Dynabook is a sort of computer holy grail, with the goal of being a truly personal computer. The Dynabook ultimately is expected to handle with equal facility any and all kinds of information management, and to be all (computer type) things to all people. Accordingly Smalltalk has to carry quite a burden of expressiveness and convenience.

Alan Kay is the man chiefly responsible for the vision of the Dynabook. In the late 1960's he did work on a preliminary version, known in that incarnation as the Flex machine. Then in the early 1970's he went to the Xerox Palo Alto Research Center and there formed the Learning Research Group. Alan's goal was still a truly useful personal computer, with the Xerox Alto being the interim hardware for the Dynabook, and with LRG doing Smalltalk as the software.

Smalltalk drew heavily from Flex, which in turn was an "Eulerized" version of Simula. While a LISP influence is clearly evidenced in the deeper structure of Smalltalk, the *class* notion from Simula dominated the design. The language became completely based on the notion of a class as the sole structural unit, with instances of classes, or *objects*, being the concrete units which inhabit the world of the Smalltalk system. Smalltalk did not completely give up its LISP heritage; rather that heritage is felt more as a flavor of the system than as specific ideas of the programming language.

Relationship of Smalltalk to Object Oriented Programming

More than a programming language, Smalltalk is a complete programming environment, all of which reflects the object oriented philosophy. Object oriented programming is so much a part of Smalltalk that it is difficult to tell where one leaves off and the other begins. While the entire Smalltalk system is worthy of investigation in its own right, we are now concerned only with object oriented *programming,* which is just a part of the Smalltalk system. A brief discussion will make clear the distinction and remove confusion about which term means what.

Note: this paper is not a tutorial on the Smalltalk language. I will discuss the internals and specifics of Smalltalk only insofar as they relate to the topic at hand. Readers unfamiliar with Smalltalk may wish to read the references first, although I have tried to make this unnecessary. I suggest to the confused reader that he finish reading this paper, then consult the references to clear up any confusion, then (optionally) read this paper again.

Smalltalk may be thought of as comprised of four pieces, *viz.*, a programming language kernel, a programming paradigm, a programming system, and a user interface model. These pieces are fuzzily defined and not explicit within the actual Smalltalk system. They are basically hierarchical, though there is overlap and some convolution. Thus the user interface is built on the programming system, which is built following the programming paradigm and using the programming language kernel.

The *programming language kernel* is the syntax and semantics as determined by the Smalltalk compiler. The *programming paradigm* is the style of use of the kernel, a sort of world view or "meaning" attached to the entities in the kernel. The *programming system* is the set of system objects and classes that provides the framework for exercising the programming paradigm and language kernel, the things necessary to make programming possible and convenient. The *user interface model* is the use and usage of the systems building materials in order to present the system to the user – in other words, the given user interface plus the user interface "flavor." The combination of these four pieces is the Smalltalk system.

Although I have represented the pieces as separate and independent, they are not, really. In fact they are inseparable and very interdependent. Not only could each piece itself not exist in a vacuum, the design for each piece influenced the design for all the other pieces, i.e., each design could not exist in a vacuum. A more faithful representation would be as interrelated aspects of the Smalltalk system. Following the note, however, I shall continue to consider them as "pieces" rather than "aspects."

Using this view of the Smalltalk world, imagine a line drawn within the programming system piece such that the objects and classes relating to the user interface model are on one side and the objects and classes relating to the programming paradigm and language kernel are on the other. We now find Smalltalk divided naturally into two parts: a user interface part, and another part. This other part is the object oriented programming aspect of Smalltalk. With the understanding that by "Smalltalk" I mean only as much of Smalltalk as is below the line of the user interface part, I shall henceforth use the terms "Smalltalk" and "object oriented programming" interchangeably.

Characterizing Object Oriented Systems

Object oriented programming is *not* programming using a Simula-like class concept, just as structured programming is not GOTO-less programming. Most of the definitions of structured programming fail to impart any real understanding of the term. We can do better by *characterizing* object oriented programming rather than giving a formal definition.

No explanation of object oriented programming could get off the ground without a discussion of objects. In characterizing such systems we are interested not in what an object is but in how an object appears. In what follows an object is always viewed from outside (the characterization of objects), not from inside (what objects "are").

It is no accident that in explaining object oriented programming objects are viewed from outside. The shift of viewpoint from inside to outside is itself an essential part of object oriented programming. In my experience this shift occurs as a quantum leap, the "aha!" that accompanies a flash of insight. In spite of this, I am convinced that the view from outside is the natural one – only my years of training and experience as a programmer conditioned me to "normally" view objects from inside. Probably due to the influence of formal mathematics, programmatic behavior was originally thought of as extrinsic. For understanding complex systems, however, intrinsic behavior provides a better metaphor, because people think that way (for example, children learn Smalltalk very quickly.) As Dijkstra cautions, we must be careful not to think something is convenient just because it is conventional. The first principle of object oriented programming might be called *intelligence encapsulation*: view objects from outside to provide a natural metaphor of intrinsic behavior.

Objects

The Smalltalk world is populated by items seen uniformly to be "objects." These "objects" are the sole inhabitants of an otherwise empty universe. This is not just a trick of nomenclature; the items are uniform in a number of ways, that uniformity producing the items' "objectivity." In what ways are objects uniform?

Objects are uniform in that all items are objects. An item found floating in the void of Smalltalk's universe is certain to be an amorphous blob. The blob has the properties of *objectness*: inherent processing ability, message communication, and uniformity of appearance, status, and reference. The item being amorphous means that no other properties are evident. The object may possess properties outside objectness, but these are made available by the whim of the object - any such properties are not visible to an outside observer.

Objects are uniform in that all objects are equally objectlike. What this means is that all objects communicate using the same metaphor, namely message passing. Objects send messages to communicate with objects. (Message sending is discussed later in the paper.)

Objects are uniform in that no object is given any particular status. Thus, "primitive" objects, such as integers, are objects just like any other. Also, "system" objects, such as class *Class*, are objects just like any other. Finally, user defined objects are objects just like any other. There are no "second class citizens."

Not only are objects themselves uniform, the means of referring to objects is uniform. An object is always dealt with as a whole, by using its (system internal) name; a given name may name any object, since all objects have the same kind of names. It follows that there is no way of opening up an object and looking at its insides, or updating ("smashing") its state. What is more important is that the concept of opening up an object does not exist in the language. (This is like trying to imagine that something is true that can't be thought of in the first place.)

Of course it is possible for the object itself to act as if it could be opened up. With suitable methods an object can choose to provide behavior that duplicates, say, a Pascal record. Pascal records can certainly have their insides looked at, or their state updated (or smashed). The distinction is that the object itself has chosen to provide this behavior - it is not part of the language. Furthermore, most objects do not normally provide such behavior. In a true object oriented system, this is as true of the system philosophy as of the actual mechanisms.

Processing, Communication, and Message Sending

For a processing system to be a useful one, processing activity must take place. In Smalltalk, the processing activity takes place inside objects. An object, far from being inert matter, is an active, alive, intelligent entity, and is responsible for providing its own computational behavior. Thus processing capability is not only inside the object, it is everpresent within and inseparable from the object.

The other property essential to a programming system is communication. An object in process may at times be entirely self sufficient, but when it is not it must have some way of interacting with objects outside of itself. Also, the user wants processing done to bring about his wishes. Both of these are needs of communication, and both are served by the mechanism of message passing.

Objects process and send messages to effect the user's desires as well as their own. A user asks an object to carry out some processing activity by sending to the object a message. The object may in turn ask other objects for information, or for some computational work to be done, by sending them messages, and so on.

Message sending serves as the uniform metaphor for communication in the same way that objects serve as the uniform metaphor for processing capability and synthesis. This uniformity is an important part of Smalltalk. In what ways is message sending uniform?

Message sending is uniform in that all processing is accomplished by message sending. The same mechanism serves to do addition, compute arctanh, request the most complicated file service operation, or provide whatever behavior is available from a user defined object. No other mechanism - such as "operating on data" - is available.

Message sending is uniform in that one message is just like another. By this I do not mean that all messages are identical, or that all messages have the same format, but that messages are sent the same way irrespective of the recipient. An example should clarify this: the message "+" to an integer, denoting addition, is sent the same way as the message "+" to a dictionary, (possibly) denoting adding an entry.

The distinction between communicating and accomplishing processing is a fine one in Smalltalk. Objects react to messages sent to accomplish processing by sending messages to accomplish processing. The buck has to stop somewhere (and indeed it does), but it seems as if it could be passed on indefinitely. A corollary of messages all being sent the same way is that each object potentially can respond to any message by sending other messages in turn. The principle as it applies to object oriented programming is: any object can accomplish processing requested by any message by directing message flow to other objects.

Messages

As the sending of a message is the only way of communicating, the message itself must be the information to be communicated. Conceptually a message is the text of the message-request. Additionally a message may be parameterized by sending along with the text one or more object names. The object name parameters are part of the language of discourse and are different from the text in that the text is constant whereas the parameters may vary. An object responds to a message with a *reply*, which is an object name.

A message serves to initiate processing and request information. The text of the message informs the object what is requested. The parameters supply any additional necessary information or computational ability. The reply confirms activity completed and returns the information requested.

There is a subtle but important distinction between a conventional procedure call, which denotes an action, and sending a message, which makes a request. In a typical procedural programming language it is hard to give up the notion that the caller of a procedure is somehow "in control." The caller and callee share a language, a set of procedural interfaces, by which the caller directs the callee to perform actions. In Smalltalk, on the other hand, a message is a request of what the sender wants with no hint or concern as to what the receiver should do to accomodate the sender's wishes. The sender, presuming all objects to be quite intelligent, trusts the receiver to do "the right thing" without worrying about exactly what the right thing is. Thus assured, the sender relinquishes control philosophically as well as actually, so that the interpretation of the message is left entirely up to its recipient. This notion, a sort of *call by desire*, is central to the object oriented philosophy.

Sharing

In order to meet the goals of the Dynabook project, Smalltalk must do more than provide computational activity *in situ*. A successful personal computer system will be understandable, usable, modifiable, and adaptable. All of these requirements can be met by a facility for *sharing*.

Sharing allows understanding because it is a good match to the way people think. For example, the earth and a basketball are different, yet each has the property *round*. There isn't one "round" that belongs to the earth and another that belongs to the basketball; the common attribute round is shared. People model the universe by collecting together archetypal attributes and allowing them to be shared by the things to which they are common.

Sharing makes for a usable system by facilitating *factoring*, the property of one thing being in only one place. Successful factoring produces brevity, clarity, modularity, concinnity, and synchronicity, which in turn provide manageability in complex systems. In a project as ambitious as the Dynabook, manageability is essential to usability.

Sharing increases the ease of making modifications. This may sound paradoxical, since changing shared parts increases the chance of producing unforeseen consequences. The paradox disappears on realizing that sharing is not a binary decision but a spectrum from totally shared to totally individual. One can choose an appropriate place for a modification by moving around in the sharing spectrum, whereupon consequently the level of detail is neither too great nor too trivial. To put this another way, you can concentrate on what must be changed, not what must be left alone. This is made possible by an explicit framework providing a spectrum of sharing.

Sharing provides for adaptation by being variable along another dimension, the dimension of individuality. What this means is that attributes can be shared by a group while allowing for individuals within the group to reinterpret some "shared" behavior *as it applies to the individuals themselves*. The previous sharing spectrum varies as to which objects share which collections of attributes; the dimension of individuality determines to what extent individuals within a given collection actually share the "common" attributes. The result of allowing individual variability is that, given something close to what you want, it is easy to produce exactly what you want by overriding shared behavior with individual behavior – to *adapt*. This situation of "I want something just like that, except ..." invariably arises in real use. Adaptation is supported by sharing in this form and is referred to as *differential programming*.

Smalltalk meets the goals of sharing by providing a framework of classing, subclassing, and superclassing, collectively referred to as *inheritance*. In relation to object oriented programming what is important is not the mechanism of classing and sub-superclassing but the provision for the merits of sharing. Inheritance is also an excellent paradigm – probably any particular framework would be – for *elucidation*, providing as it does a sort of road map for the system universe. The conclusion: object oriented systems provide an explicit framework for sharing so as to accrue the attendant advantages.

Notes on Related Systems

Smalltalk remains the model object oriented system. But no programming language is an island, and a mention of other systems is clearly called for. Following is a capsule summary of relevant systems. The point here is to view the work as it relates to object oriented programming, not to appraise the merits of the various systems.

A group of installations with Burroughs B220's used a clever scheme to insure portability of files from one installation to another, in order to get around the difficulty of not having operating systems. A file was arranged as a transfer vector with relative pointers, followed by actual B220 code, followed by data. Although the code and representation of data varied from installation to installation, the desired effect could be obtained by reading the file into a standard memory location and branching indirectly through the appropriate location in the transfer vector. This idea was later carried over into the operating system of the B5000. Indeed, the entire architectural concept of the B5000 pointed in this direction, offering the first hardware implementation of what are now known as capabilities. Unfortunately, the B5000 is almost legendary for being ahead of its time, unappreciated, and misunderstood, and so the impact of the earliest object oriented ideas was hardly felt.

The Sketchpad system of the early 1960's is an object oriented system that apparently was programmed using an object oriented style. But Sketchpad is a graphical interaction system, not a programming system. While Sketchpad was one of the earliest object oriented systems, its application to programming systems was not widely appreciated, perhaps because the object oriented philosophy was not explicit enough.

The programming language Simula is an ALGOL based simulation language which first introduced the class concept. Simula certainly can be used with a style which is highly object oriented. But old ideas die hard, and Simula's extensions over ALGOL were used in support of simulation features with traditional ALGOL style programming taking up the slack. In practice, Simula falls short of realizing object oriented programming for several reasons. One, primitive data types, system data types, and user data types do not all have equal status. Two, the extensions to ALGOL, while including the class concept, also included many "features" such as INSPECT and IN, which are contrary to the object oriented philosophy. Three, the object oriented metaphor has not really caught on in the Simula community, due to the roots of the language in ALGOL. Four, the typing mechanism in Simula often makes it difficult to realize the free-spiritedness of object oriented programming. If Sketchpad is an object oriented system without the language, Simula is an object oriented language which is rarely used in an object oriented fashion.

The LISP-AI notion of frames captures well the idea that behavior rides along with the thing whose behavior is being described. Also, the resemblance of the common/default mechanism of frames to the inheritance mechanism of Smalltalk is striking. I think of frames as an object oriented extension to LISP. Similar to Simula, however, the problem with frames is that the LISP notion very strongly remains, so that the inclination towards the object oriented philosophy is not nearly so strong, nor as clearly defined, as in a completely object oriented system.

The programming language Alphard is related but still in a rather uncertain state. After more use the character of Alphard will be clearer, but generally speaking Alphard centers on abstraction, which is roughly evenly divided between data encapsulation and abstract data types. If anything Alphard leans to the abstract data types side, though not so far as, for example, algebraic axiomatic specification. One clear oversight is that Alphard does not use inheritance in any strong fashion.

The programming language ADA, though still in the implementation stages, is clearly centered and focused on data encapsulation. The sense of operating on data is very strong in ADA, from syntax to operator overloading to separation of types from modules. Generic procedures capture some, but not nearly all, of the polymorphism inherent in object oriented programming, and ADA's facility for inheritance is weak at best. ADA has followed traditional (i.e., ALGOL style) language design principles so that the issues of implementation and usage would be clear at the outset. ADA is one of the few languages that was designed with extensive, specific requirements definitely given *a priori*; none of these requirements mentions object oriented programming.

The programming language CLU makes a serious effort in the object oriented direction. The sense of unification of reference mode is very strong in CLU, narrowing the distinction between different object categories. CLU falls a little short on the syntax side, which is still a conventional functional (i.e., data oriented) format. Also, the typing mechanism of CLU doesn't give a sense of freedom to the "receiver" of a message, since the particular response mechanism is specified by the type, which is known and explicit in the source at the call site. This restriction shifts the sense of control from the receiver (object oriented) to the sender (data oriented). Furthermore, CLU resembles Alphard and ADA in lacking a good inheritance mechanism. As with Alphard, CLU tries to balance abstract data types with data encapsulation by centering on abstraction, but CLU leans to the data encapsulation side more than Alphard does. Even so, CLU remains as the strongest "traditional" contender for supporting object oriented programming.

Conclusions

It is usual to understand new ideas in terms of familiar ones. Object oriented programming might be (and probably already has been) likened to abstract data types, data encapsulation, information hiding, and modularization. This may be more palatable to some audiences, but it ducks the issue. These comparisons neither do justice to, nor capture the essence of, the object oriented methodology. To quote Alan Kay:

> "Though Smalltalk's structure allows the techniques now known as data abstraction to be easily (and more generally) employed, the entire thrust of its design has been to supercede the concept of data and procedures entirely; to replace these with the more generally useful notions of activity, communication, and inheritance."

Object oriented programming holds great promise, but is not yet widely understood. In spite of object oriented programming just now becoming popular, the term of its use is already in danger of being overworked and misunderstood. My goal has been to argue for an understanding of object oriented programming, not just a dogmatic definition. Let us hope that we have learned our lesson from structured programming and find out what the term means *before* we start using it.

Acknowledgements

My thanks to all the people who have cheerfully let me express their ideas as though they were my own.

References

Dynabook and Related

Kay, A., "The Reactive Engine," Ph.D. Thesis, University of Utah, September, 1969

Kay, A., "A Personal Computer for Children of All Ages," ACM Nat'l Conf., Boston, Aug., 1972

Kay, A., and A. Goldberg, "Personal Dynamic Media," Computer, March, 1977

Kay, A., "Microelectronics and the Personal Computer," Scientific American, September, 1977

Other systems mentioned

Birtwhistle, *et al.*, *Simula Begin*, Petrocelli/Charter, 1975

Ichbiah, J.D., *et al.*, "Preliminary ADA Reference Manual," ACM SIGPLAN Notices 14, 6A, June, 1979

Intel Corporation, *Introduction to the iAPX 432 Architecture*, Manual Number 171821-001

Intel Corporation, *iAPX 432 Architecture Reference Manual*, Manual Number 171860-001

Liskov, B., *et al.*, *CLU Reference Manual*, MIT-TR 225, October, 1979

McKeag, R.M., "Burroughs B5500 Master Control Program," Queens University of Belfast, Northern Ireland, 1971

Roberts, B., and I. Goldstein, "The FRL Manual," MIT-TR 409, September, 1979

Sutherland, I., "Sketchpad," Ph.D. Thesis, MIT, 1963

Wulf, London, and Shaw, "An Introduction to the Construction and Verification of ALPHARD Programs," IEEE Transactions on Software Engineering SE-2, 4, 1976

Smalltalk

Byte special issue on Smalltalk, BYTE, August 1981 (primarily the following articles)

Xerox Learning Research Group, "The Smalltalk-80 System"

Robson, D., "Object-Oriented Software Systems"

Tesler, L., "The Smalltalk Environment"

Ingalls, D., "Design Principles Behind Smalltalk"

Goldberg, A., and J. Ross, "Is the Smalltalk-80 System for Children?"

Ingalls, D., "The Smalltalk-76 Programming System: Design and Implementation," 5th Annual ACM Symposium on Principles of Programming Languages, January, 1978