What is the "Object"

in Object-oriented Programming? [1]

O.M. Nierstrasz

Abstract

Object-oriented programming has become quite widespread in recent years, although there are few guidelines to help us distinguish when a system is "truly" object-oriented or not. In this paper we discuss what have emerged as the main concepts in the object-oriented approach, and we attempt to motivate these concepts in terms of how they aid in software development.

Résumé

Bien que la programmation-objet soit devenue assez courante ces dernières années, il n'existe que peu de règles qui nous permettent de savoir si un système est véritablement orienté objet ou pas. Le présent article expose les concepts principaux associés à l'approche orientée objet, qu'il tente de justifier en mettant en exergue leur rôle dans le développement d'applications logicielles.

1 Introduction

There is much confusion about the term "object-oriented", which it appears can be applied to anything from an operating system to an interface for a text editor. Although there are a number of things the term normally brings to mind, there is no clear way of telling whether system A is "really" object-oriented, and system B is not.

In this paper we will discuss a number of object-oriented "concepts", and put forward the hypothesis that object-orientation is an *approach* rather than a specific set of language constructs. In fact, one can probably use an arbitrary programming language and still write in an object-oriented style. However, object-oriented programming languages do exist with built-in constructs that support (or enforce) this style of programming.

The principle that is fundamental to the object-oriented approach is encapsulation, or more precisely, *data abstraction*: an "object" packages an entity (data) and the operations that apply to it. In its most basic form we may have a module of a program consisting of a number of static variables together with the set of procedures that are used to manipulate the variables. In general, those variables are "private": the object is defined by its behaviour, not its representation. So if I have an object called "Jaguar", I do not change its colour by modifying its "colour" variable, but by executing an operation such as "paint".

The notion of data abstraction is especially useful if one can apply it to multiple instances of an object "type". A language like Fortran provides a fixed set of data types, such as integers and real numbers, each of which supports a number of operations (addition, subtraction ...). Languages like C and Pascal support the construction of programmer-defined types, one cannot define new type-specific operations. Object-oriented languages enable the programmer to create his (or her) own object types. Each object is then an instance of some type, and supports the operations (called

[1] In *Proceedings of the CERN School of Computing*, CERN 87-04, pp. 43-53, Renesse, The Netherlands, Aug. 31-Sept. 13, 1986.

methods) that are defined for that type. (The word "class" is often used interchangeably with the word "type" when talking about objects, however, it is more precise to think of an object class as the *set* of instances of an object type.)

Object "independence" is the notion that each object is ultimately responsible for itself. Objects can only be manipulated via their methods.

Supporting this idea of object independence is a model for object communication which replaces the notion of operating *on* objects with that of passing messages to them. Every method invocation is seen as a message requesting the object to perform some action. The method is (conceptually) not performed by the caller, but by the object receiving the message. It follows, then, that the recipient of such a message is, in some sense, free to refuse the request, or choose from a variety of possible implementations of the method.

Classes of objects may not be disjoint. Object types may form a hierarchy in which a subtype "inherits" the methods of one or more parent types. The implementation of the subtype's methods may or may not be inherited from the parent. In fact, one may even have multiple implementations of a single type which only share their interface. By properly organizing such a type hierarchy, one may concentrate on the similarities between objects rather than on their differences.

Fully object-oriented systems have a strong degree of homogeneity. This means that *everything* is an object. One does not distinguish between, say, programs and processes and files and objects – one only has objects, and objects are both active (like processes) and persistent (like files). In particular, the object types (or, more precisely, the type specifications) are objects of the type called "type".

A natural step to take is to map objects to an "actor" model of computation. Simply put, an actor is a cheap, persistent, message-passing process. (The actor formalism as described in [11] is more rigorous, however this simple definition suffices for our purposes.) An actor must be inexpensive because we would like to be able to easily create and destroy them. If there is a one-to-one mapping between actors and objects, then it is clear that we may have many more actors than there are processes in a traditional multiprocessing environment, though at any one time only a few actors will actually be doing anything. The message-passing paradigm supports the notion of actors communicating with one another whether they are running on the same machine or not. If we consider objects to be active entities running "on top of" actors, then we see that the object-oriented approach also provides us with an easy way to understand concurrency, and applications running on a network of machines. At the same time, we note that there are several object-oriented languages and environments that do not support the notion of concurrently executing objects.

The main object-oriented concepts, then, are:

1. Data abstraction.
2. Independence.
3. Message-passing paradigm.
4. Inheritance.
5. Homogeneity.
6. Concurrency (some systems).

We shall elaborate on each of these concepts in turn, and attempt to explain their importance.

2 Data abstraction.

By far the most important concept in the object-oriented approach is data abstraction. By this we mean that we are interested in the behaviour of an object rather than in its representation. Every object has a clearly defined interface which is independent of the object's internal representation. The interface is a collection of operations, or "methods", which may be invoked by another object.

Furthermore, one may have many instances of an object type. The programmer is not restricted to a fixed set of types, but is free to add new types as required. This capability first appeared in the programming language "Simula" [3], a simulation language in which the programmer was able to define his own "classes" of objects (i.e., corresponding to the real objects one wished to model).

A type definition looks very much like a module: there are a collection of permanent variables that encode the "state" of the object, and a set of methods that use and change that state. A module, however, only defines a single instance, whereas one may have many instances of a new type. In order to make use of an instance of that type, one only needs to know what the interface is, namely, the names of the methods, and the types of the input and output parameters. (Naturally some specification of the semantics of the methods is also required.)

The most immediate benefit of this approach is that the programmer is free to use higher levels of abstraction as appropriate. (I.e., at each level of abstraction one concentrates on that level's functionality, while hiding the lower-level details of implementation.) One is encouraged to decompose a programming problem into a collection of cooperating objects of varying levels of complexity.

The separation of interface and implementation of a new type means that types are, to some extent, representation-independent. This makes it easier to experiment with different implementations of an object type, and increases the portability of software. It is, however, crucial to design an interface very carefully, since that is the part of an object type that remains stable across different implementations.

A language that allows programmers to define their own types is a natural candidate for strong typing. Type-checking is done as soon as possible, usually at compile-time. (Sometimes one does not know at compile-time what kinds of objects will be manipulated, so one must do explicit type-checking at run-time; for example, a mailbox object may hold various types of mail objects.)

In an object-oriented language, one is typically allowed to use the same method names for a variety of different object types. This is called "overloading", or *polymorphism*. For example, one may wish to define addition for complex numbers, and use the symbol "+" as the method name, even though it is already defined for integers. The type of the object to which such a method applies is, in a sense, an implicit argument to the method. Without overloading, one would be forced to define the methods "complex_add", "integer_add", and so on. Overloading allows for more readable code. If used in an undisciplined fashion, however, overloading can have quite the opposite effect. (As a grotesque example, consider the use of the symbol "+" as the method name for complex subtraction.)

Well-designed objects are typically applicable in a number of situations. One can therefore expect in an object-oriented environment that most of the object types one defines can be re-used,

or, conversely, that when putting together an application, one can find many of the parts one needs already available as pre-defined object types.

3 Independence.

There are several notions of independence displayed by objects. The most fundamental is that objects have control over their own state. An object's methods are the *only* interface to its state. Since this is not always guaranteed in languages that enable programmers to add new data types, it is worth emphasizing. Clearly, if one has access to the internal representation of an object, then one loses the property that the implementation is independent of the interface.

Objects also have control over their own existence. Once created, an object will continue to exist (or *persist*) even if its creator dies. Persistent objects eliminate the need for "files". One need not be concerned as a programmer with the location of an object in main memory or secondary memory. Ideally, dormant objects automatically "migrate" to passive memory. Garbage collection, however, is compatible with this notion, since an object that is neither active nor accessible is effectively dead already.

Another form of independence is the ability to add new object types at run-time. This is crucial if the object-oriented environment is also the development environment. Otherwise the addition of new types must take place "outside". If new types are to be added dynamically, then old objects must be capable to talking to instances of the newly-created types. (Compile-time type-checking is clearly not possible for those objects that will talk to objects of types that do not exist yet.)

4 Message-passing paradigm.

Independence of objects is supported conceptually by using message-passing as a model for object communication. In terms of the model it is clear that an object cannot "operate on" another object. The only way an object can interact with the outside world is by sending messages. Consequently, object A "invokes" a method of object B by sending B the message, "please execute this method".

Assuming all goes well and the message can actually be delivered, B is free to interpret the message as it pleases. It may choose amongst several ways of handling the request, it may delay responding, or it may decide that it does not wish to handle the request at all, and return an exception report. "Returning" is also accomplished by message-passing.

It is important to realize that message-passing is a model for object communication rather than an implementation requirement. In fact, more than one object-oriented language translates message-passing into straightforward procedure calls. This is possible when flow-of-control is single-thread, and no more than one object is "executing" at any time. However, in a concurrent environment where objects are truly active, "real" message-passing is a natural way in which to implement communication between objects.

Message-passing may be synchronous or asynchronous. Basically, the difference is that with asynchronous message-passing the message is put on a queue, and the sender is immediately free to concentrate on another task (though it usually just waits for a response). With synchronous message-passing, the sender blocks until the message can be delivered, sometimes even until the

recipient prepares a response. The problem with asynchronous message-passing is that it is (strictly speaking) impossible to implement, since there is no bound on the size of the message queues. The problem with synchronous message-passing is that it says nothing about the order of arrival of messages when multiple senders compete for a single recipient, and it limits the amount of concurrency by blocking the sender.

Aynchronous message-passing is perhaps more consistent with the idea of object independence, but it is not a pre-requisite for an object-oriented system.

5 Inheritance.

From the point of view of organizing software and reusability of code, the most important mechanism in an object-oriented language is the ability to *specialize* object types. A specialized type (a *subtype*) inherits the properties of its parent type, then possibly adds more properties.

A program source, for example, may be seen as a specialization, (or *refinement*) of a text object. An instance of a subtype responds to all the messages that the parent type understands. The reverse is not true. A subtype inherits the method interfaces of the parent type, but may add new methods that the parent cannot respond to. In addition, a subtype usually inherits the implementation of the parent by default, that is the set of instance variables that encode the state of a parent instance, and the implementations of all of the methods. The subtype could add new instance variables and methods.

A subtype may, however, have a different set of instance variables, and quite different implementations of the inherited methods. For example, real numbers are a refinement of integers, but they are represented and implemented quite differently in a computer.

The set of all object types forms a specialization hierarchy with the type *object* at the root. With simple inheritance, this hierarchy is a tree.

If *multiple inheritance* is supported, the hierarchy is, in general, a directed acyclic graph. This means that a subtype may have more than one parent type (but no type can be its own ancestor!). The parent types in this case are called “traits” [7] or “flavours” or “mixins”.

The difference between a trait and a type (if any) is that a trait specifies an interface, but it need not define an implementation. One reason for this is that a trait may not be of interest as a type on its own. Consider for example the trait “ownership”. Such a trait might specify methods for setting the owner of the object, changing ownership, and determining access rights. Ownership is only of interest in conjunction with other properties. An “owned object” would be of little use on its own.

Multiple inheritance increases the reusability of software. Simple inheritance is inadequate for describing many of the useful relationships between object types. Suppose that we would like to have both owned and unowned pieces of text. Similarly we might have owned and unowned graphical objects. (An unowned object might be part of a larger owned object.) The type *owned_text* is a refinement of *text*, and *owned_graphic* is a refinement of *graphic*, but with simple inheritance there is no way to say that the methods pertaining to ownership are to be shared. Multiple inheritance encourages the programmer to define traits that can be combined much more flexibly.

The added flexibility is not without its cost – there are no easy solutions if one wishes to combine

two traits that use the same method name to mean different things. Provided there are no naming conflicts, multiple inheritance is an elegant mechanism for aiding in the design of new object types.

Variation of types is another form of inheritance. A new type may be defined as being equivalent to an existing type, adding no new methods. The type definition serves mainly to ensure correct use of objects of that type – type clashes can be detected by the compiler, or at run-time. One would not wish to confuse names of people with names of programs, even though they would both be implemented as strings.

Furthermore, equivalent types may have implementations tailored to a particular environment or usage. Window objects may behave the same on a variety of graphic devices, yet will be implemented in different ways. Similarly one might choose from a variety of implementations of lookup tables with different overhead and performance characteristics, though all are functionally equivalent.

A weak form of "inheritance" is parameterization. A "container" object is an object that manages objects of a given type (or set of types), yet is not concerned with what that type is. When an instance of a container is defined (or created), the type of the contained object should be known. In the definition of the container type, the type of the contained objects is specified as a parameter. For example, one would define an array as an array *of* a contained type. When declaring an array, one would specify an array of integers, or an array of mail messages. (The array object does not care what it manages, but at some point the compiler needs to know what it will be.)

6 Homogeneity.

In a "fully" object-oriented environment, "everything" is an object. One does not distinguish between programs and objects – objects *are* the active entities. Object types (or rather the specifications and implementations of object types) are objects too (of type *type*). This is necessary in a complete environment, since a programmer will need to instantiate new types from *within* the environment.

To say that "everything" is an object is formally attractive, but one must take care. For example, are messages objects? If so, in order to manipulate a message, is it necessary to send it a message? This sort of circularity is normally broken at some level. It is fundamental to the object-oriented model that messages can be sent between objects, without having to define "how" this is done.

More important is the ability to think of instance variables as objects. This enables a programmer to construct complex objects whose parts are also objects. Again, the circularity most be broken at some point. Is it really useful to talk about the "parts" of an integer as objects? What are the instance variables of a single bit? In any case, it is the *behaviour* of objects that is crucial, not their representation. We should, therefore, not feel uncomfortable with the idea that certain basic object types are given, and all other types are built up from them. There is no need for us to know how the basic types are implemented.

The principle of homogeneity can be carried even to the level of code. In Smalltalk, control structures are implemented using objects. A *block expression* is an object that can be executed by sending it the message *value*. Conditional execution is accomplished by sending an object of type

boolean two block expressions, one to be evaluated if the value of the boolean is *true*, the other if it is *false*. Similarly, iteration is accomplished by sending an integer the message *timesRepeat:*, with a block expression to be executed as an argument.

Carrying the notion of homogeneity to this degree certainly makes for a very consistent view of the environment, and it means that interpreted (rather than compiled) code can be understood in terms of communicating objects, but it also means that programmers may have to change the way they think about problem-solving at even very low levels. Does it help a programmer to think of the expression:

$$1 + 2$$

as "send the message '+ 2' to the object 1"? Probably not. At the same time it is reassuring that the object-oriented approach is sufficiently powerful to capture computation at whatever level of abstraction.

7 Concurrency.

The message-passing paradigm of object communication lends itself well to an environment in which every object is an active entity, i.e., a process. Since there are large numbers of objects, one must be able to have large numbers of processes (most of which are normally "asleep"). Furthermore, since objects are persistent, it is desirable that the processes that implement them be persistent too.

We call such processes *actors*, after the computational model invented by Hewitt [11]. The actor formalism can be useful for understanding object interactions. From a purely practical point of view it is attractive to think of objects being implemented "on top of" actors. Actors, then, are inexpensive, persistent, message-passing processes. Objects add structure to the state of an actor, and provide structure to the message-passing that may take place. (For examples of actors as they appear in a programming language or an operating system, see [4] or [15].)

Message-passing can be defined in several ways. We have already discussed synchronous and asynchronous message-passing. In addition, an actor may be able to receive messages at a number of different *ports*. Ports typically have different priorities, so that messages sent to a port of high priority will be received before those waiting at another port. By analogy, people can receive messages by letter or by telephone. Telephone messages have higher priority than letters, though a second telephone call will not interrupt a call in progress.

Multiple ports are not strictly required, though they do provide us with a mechanism for "interrupting" an actor while it is executing.

Even with single-port actors we can model many kinds of object interactions. Object A invoking method M of object B can be easily understood as follows: Object A sends a CALL message to object B. When B receives the message, it prepares a response and sends a RETURN message to A. If there is a problem with A's call, B sends back an EXCEPTION message instead. Furthermore, if A itself receives other CALL messages before B's response, those messages are delayed until B returns.

B's response could also be directed to a separate port of A, thus avoiding the need to delay

messages.

B might be acting as an *agent* for another object C. In that case, B would *forward* A's message to C (including the information C needs to send the response). C could then send its response directly to A, rather than through B.

These protocols may be modified to allow, for example, recursion. Suppose that B wants to call some method of A before sending its response. If A delays all calls because it is expecting a response from B, then B's call will never return. Recursion could be permitted by having A accept calls that it recognizes as originating from its own call to B. A "token" is thus passed around, which A will recognize when it receives a CALL message from B, or another intermediary.

We can model *transactions* with message-passing as well. A transaction is a sequence of operations on a collection of objects which is performed atomically, that is, either all of the operations are performed, or none of them (in case the sequence is aborted), and the intermediate states are not visible to other objects. Object A would, for example, ask object B to ignore calls from other objects while the transaction is taking place. A would then be guaranteed that B is not seen in an inconsistent state. By analogy, one can imagine the objects involved in a transaction as having a meeting behind closed doors. Anyone who is requested to join the meeting may enter the room, but no one may leave the room until the meeting is adjourned.

Typically one may also "back out" of a transaction – if the meeting is aborted, then the participants return to their state before the meeting started. The CALL/RETURN message-passing protocol is followed as before, except that after returning, each object enters a "ready" state in which it is prepared to either commit or abort. At any time an object taking part in the transaction may issue an ABORT, which causes all participants to back up to a checkpointed state. (This message is broadcast along the already-established CALL/RETURN paths.) If the transaction completes without an abort being issued, the initiator of the transaction commits it by sending a COMMIT message to the participants (again, along the paths that the CALL/RETURN messages took).

Nested transactions are a useful concept (meetings within meetings). Each object would then keep a stack of checkpointed states, one for each nesting level. (Note that it is not necessary to completely save each state, but only those parts that may be modified in the transaction.) Argus [12] is an example of a programming language that supports both data abstraction and nested transactions.

Deadlock is still a real possibility. Consider two independent meetings, each of which suddenly requires the participation of someone in the other meeting.

Also note that the success of the two-phase commit protocol outlined above assumes reliable message-passing and checkpointing. A real implementation would have to address these issues in an object-oriented environment, as it would in any other setting.

As a final example, we can model triggering of activities by message-passing. Object A may send a SET trigger message to B, that says, "let me know when x happens". Later, when x happens as a result of yet another object making a request of B, B may send a NOTIFY message to A. When A receives the notification it initiates a new activity. A may also UNSET the trigger, telling B that it is no longer interested in x. A general triggering mechanism like this enables applications to know about interesting events without having to poll for them.

The examples given in this section were not intended so much to show what kinds of mechanisms

are required for a concurrent object-oriented language or environment, but rather to indicate what the possibilities are, and how easily independent, message-passing objects can be mapped to an actor world. Since message-passing is the only medium for communication, it is also a natural step to map actors, and therefore objects, onto a heterogeneous environment of different processors connected through a network. See [8] for a discussion of similar issues in an environment that supports synchronous message-passing.

8 Concluding remarks.

The fundamental concepts in an object-oriented environment are data abstraction and inheritance. A programmer specifies an interface to a new object type as a collection of methods which can be invoked. Separately, one defines the representation of the object's state, and the implementation of the methods. The methods should be the *only* acceptable interface to an object.

One must be able to create multiple instances of an object type.

A subtype inherits the methods of its parent type, but not necessarily the implementation of the methods. If multiple inheritance is supported, a subtype may inherit methods from several parent types.

The inheritance mechanism "overloads" methods, since methods apply to objects of different (though usually related) types.

In a fully object-oriented system, types are also objects, and one can instantiate new types at run-time. In a homogeneous design, expressions in the programming language are also objects, as are instance variables of an object's permanent state, and even the messages objects use to communicate with.

If concurrency and distribution are at issue, each object may be an active entity that sits "on top of" a message-passing process, called an "actor".

Examples of fully-integrated languages are Smalltalk [1, 9, 10] and Loops [13]. Smalltalk does not support concurrency, distribution or multiple inheritance, but it does provide a strongly homogeneous view of its language, in which even control structures are objects.

Languages like Ada [2] support data abstraction, but no multiple instantiation or inheritance. It is impossible to add new types at run-time.

Some languages like Lisp and C have been enhanced to give them some object-oriented capabilities, without taking away anything that was in the language before. In the case of Lisp [14], "flavours", or traits have been added, so that one has both data abstraction and multiple inheritance. With Objective-C, (or its predecessor, OOPC), [5, 6] a pre-processor was written that translates Objective-C object classes into C code, that can then be compiled by a regular C compiler.

These two examples suggest that it is possible to program in an object-oriented "style" without actually using an object-oriented language. In fact, this can be done, though one would then not be able to make full use of inheritance or overloading of operations. Nor would one be able to add types at run-time unless an interpretable language (like Lisp) were used.

As with any other programming style (procedural, applicative, logic-oriented ...), one requires

a programming discipline, or methodology. With object-oriented programming, perhaps the most difficult task is deciding how to naturally decompose a problem into objects. How does one design object types (i.e., interfaces) for maximum reusability and applicability?

In summary, the issues addressed by the object-oriented approach are:

1. Maintainability, through the separation of interface and implementation.
2. Reusability, as a function of well-designed general-purpose object types, and through the mechanism of inheritance.
3. Reliability, through strong type-checking
4. Portability, again, through the separation of interface and implementation.
5. Concurrency and distribution, through object independence, and the message-passing communication model.

References

[1] "Special issue on Smalltalk", Byte, vol. 6, no. 8, Aug 1981.

[2] J.G.P. Barnes, "An Overview of Ada", Software – Practice and Experience, vol. 10, pp. 851-887, 1980.

[3] G. Birtwistle, O. Dahl, B. Myhrtag and K. Nygaard, *Simula Begin*, Auerbach Press, Philadelphia, 1973.

[4] R.J. Byrd, S.E. Smith and P. de Jong, "An Actor-Based Programming System", Proceedings ACM SIGOA, SIGOA Newsletter, vol. 3, no. 12, pp. 67-78, Philadelphia, June 1982.

[5] B.J. Cox, "The Object Oriented Pre-Compiler", SIGPLAN Notices, vol. 18, no. 1, pp. 15-22, Jan 1983.

[6] B.J. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology", IEEE Software, vol. 1, no. 1, Jan 1984.

[7] G. Curry, L. Baer, D. Lipkie and B. Lee,, "TRAITS : an Approach for Multiple Inheritance Subclassing", Proceedings ACM SIGOA, SIGOA Newsletter, vol. 3, no. 12, Philadelphia, June 1982.

[8] W.M. Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", Software – Practice and Experience, vol. 11, pp. 435-466, 1981.

[9] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation*, Addison-Wesley, May 1983.

[10] A. Goldberg, *Smalltalk 80: the Interactive Programming Environment*, Addison-Wesley, 1984.

[11] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages", Artificial Intelligence, vol. 8, no. 3, pp. 323-364, June 1977.

[12] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", ACM TOPLAS, vol. 5, no. 3, pp. 381-404, July 1983.

[13] M. Stefik, D.G. Bobrow, S. Mittal and L. Conway, "Knowledge Programming in LOOPS: Report on an Experimental Course", The AI Magazine, pp. 3-13, Fall 1983.

[14] D. Weinreb and D. Moon, *The Lisp Machine Manual*, Symbolics Inc., 1981.

[15] H. Zimmermann, M. Guillemont, G. Morisset and J. Banino, "Chorus: A Communication and Processing Architecture for Distributed Systems", Research report no. 328, INRIA, Rocquencourt, Sept 1984.