Proceedings of NordData-91, pages 107-121, Oslo, 16-19 June, 1991

Object-Orientation: Classification Considered Harmful

Michael Jackson
Software Development Consultant
101 Hamilton Terrace
London NW8 9QY
England
Tel: +44 71 286 1814
Fax: +44 71 266 2645

Abstract
The object-oriented approach to software has been promoted as a major advance, offering a more faithful medium for modelling the application domain. But we should be no less sceptical of these claims than we have learned to be of the claims of functional programming, structured programming, logic programming, and relational database. The object-oriented concepts of classification and inheritance, in particular, should be treated with some caution. The view that each object is inherently of a specific class has some serious disadvantages: often it will be inappropriate for modelling the application domain, and it can certainly make reuse more difficult.

1. Objects, Classes, and Inheritance

Strong claims are made for the virtues of an object-oriented approach to software development: after object-oriented programming we have been offered object-oriented analysis, object-oriented design, object-oriented methods, and object-oriented database systems, each promising large improvement in our ability to develop good software efficiently. Languages, environments, and methods that want to be taken seriously present themselves as candidates for the object-oriented seal of approval; enthusiasts discuss whether Ada is object-oriented and whether the latest version of Borland's Pascal environment merits the seal or not. The question naturally arises: what are the distinguishing characteristics of an object-oriented language or method?

Whatever may be the right answer to this question, I shall concentrate my attention on three characteristics the use of objects as abstractions in modelling and programming; the structuring of the set of all possible objects into classes, where each object is assigned uniquely to a class; and the arrangement of the classes in some kind of inheritance hierarchy in which each class inherits properties from classes above it in the hierarchy. other characteristics are often regarded as essential, especially some form of polymorphism and some form of communication by message-passing between objects. They are certainly important, but I shall not discuss them here.

My theme is centred on the use of objects, classes, and inheritance, because I think they are central to the claim that object-oriented methods make for more faithful models of application domains. Some twelve years ago William Kent wrote an excellent book Data and Reality [1], in which he explored the extent to which certain kinds of data model could adequately represent the reality about which a system was to compute. My theme is largely the same, but I am concerned with objects and classes rather than with pure data as Kent was.

In discussing these concepts and suggesting that they suffer from serious limitations, I am very conscious that the field of object-oriented programming in particular is very lively: for any limitation one might identify there is some relatively new language in which that limitation has been recognised and removed. But I am chiefly concerned with the core concepts as they seem to be most widely understood and advocated and as they are embodied in languages like Smalltalk [2J.

2. Objects

The essence of an object is that it is an identifiable individual: an object is identical to itself, and not to any other object. The individuality of the object persists over time, and the object can be referred to by a unique identifier. The identifier may be conveniently implemented by a pointer in a typical imperative programming environment: the object is then directly accessible wherever its identifier is known. (In the newer techniques of relational database modelling, identifiers are sometimes known as ‘surrogates’ — names that have no significance beyond their ability to identify objects uniquely.)

The concept of individual identity, however, is not dependent on any implementation of identifiers: we are accustomed in natural language to use quite complex expressions in place of identifiers, as when we ask ‘is the person you are talking about the person who came here yesterday?’.

The concept of individual identity is missing from many traditional languages such as COBOL and Fortran. In these languages we can ask whether two variables currently hold equal values: for example, we can ask whether two records are equal. But it is not meaningful to ask whether the records are identical - whether they are the same object or two distinct objects that happen to be currently indistinguishable.

As well as having identity, objects encapsulate data and operations on the data. The data encapsulated by an object - often called the object’s ’instance variables’ - may be thought of as recording its state at any moment. The set of possible operations on the data - often called the object’s ’methods’ - may be thought of as the repertoire of things that can happen to the object during its lifetime: these happenings may change the object’s state, and the object responds to each happening in a way that depends on its state at the time of the happening.

To say that an object encapsulates its instance variables and methods is to say that the instance variables are inaccessible except through the methods. From the outside the only visible properties of an object are its repertoire of methods, which are invoked by sending messages to the object, and the responses it returns to messages received. It is these properties — the ability to respond to certain messages and the responses made — that characterise each individual object.

It is worth noting that the state of an object, as recorded in its instance variables, need not completely determine its responses to messages. This is because the instance variables are themselves identifiers of other objects. For example, a SetOfPeople object has an instance variable value for each member of the set. If we send it a message to enquire whether object 1234 — perhaps an object representing the person John Smith — is a member, its response will be determined completely by these values. But if we send it a message to enquire whether it has any member who lives at No 1 King Street, then its response is determined by what has happened to its member objects: by the fact, perhaps, that John Smith has moved away from No 1 King Street since he became a member of the set. Nothing has changed in the state of the set, but it gives a different response to the message because something has changed in the state of one of its members.

3. Classes

Objects are classified according to their properties. Each individual object belongs to one and only one class, and the class definition specifies the properties that each object of the class shall have.

So fundamental is classification that objects are created by ’instantiating’ their classes. To make a new rectangle object we send a ‘new’ message to the Rectangle class, and it responds by returning to us a shiny new, rectangle. There is no possibility of creating a new object that does not yet belong to any class, that is simply an individual that has not yet lived long enough to acquire any properties.

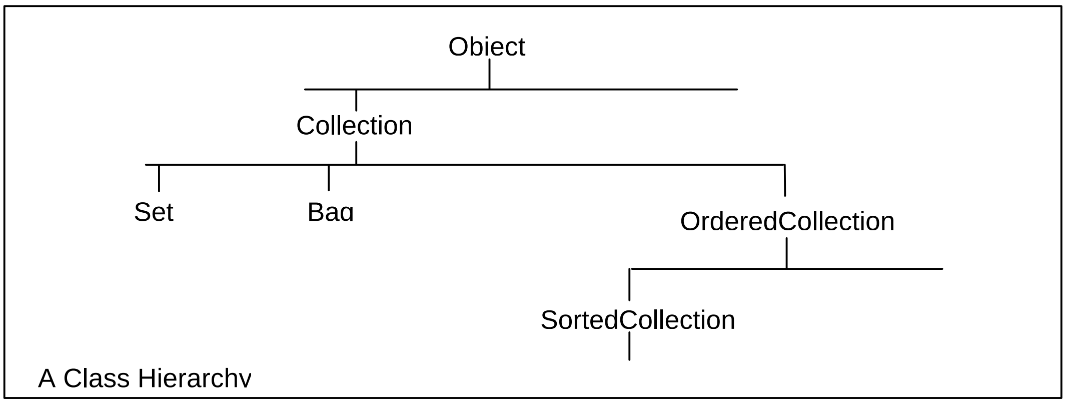
A class definition specifies the properties of an instance of the class by providing the declarations of its instance variables - sometimes just their names, but sometimes also their types - and the method texts for responding to messages. The method texts are essentially implementation-oriented, being executable programs, but in some languages, such as Eiffel [3], it is possible to give pre- and post-conditions for methods.

Classes are themselves objects, and the instance variable and method declarations they provide for their instances are part of their own state. So although the class of an object may be fixed for the object’s lifetime, the properties specified by the class may change.

4. Inheritance

Object classes may be arranged in a hierarchy in which the properties specified by each class are partly 'inherited' from the class or classes above it in the hierarchy. In single inheritance, the hierarchy is a tree. Each class has exactly one 'superclass', except the class object which is at the root of the tree and has no superclass.

In the class hierarchy the most general class — the Object class — is at the highest level, and the lower levels are populated by successively more specialised classes. One subclass of Object may be Collection; subclasses of Collection may be OrderedCollection, Bag, and Set; one subclass of OrderedCollection may be SortedCollection; and so on.



Ideally, each class inherits all the properties of its superclass unchanged, and adds some new more specialised properties of its own: Set has all the properties of Collection, and in addition has the more specialised property that all its members are distinct. This is called strict inheritance. It offers the benefit that wherever an instance of the superclass will do, an instance of its inheriting subclass will do just as well: a Set can serve perfectly well as a collection; a Rectangle can serve perfectly well as a Polygon.

In Smalltalk, inheritance is non-strict: the inheriting class is free to redefine the methods of its superclass. There is no guarantee that an instance of the inheriting subclass will serve when an instance of the superclass is needed. In practice, however, it is usual to preserve some consistency, redefining superclass methods only for reasons of efficiency and leaving their externally visible effects unchanged. The pre- and post-condition specifications of Eiffel provide considerable support for this approach.

Inheritance may also be regarded as a mechanism for saving repetition in development. In defining the Set class we do not need to repeat our definitions of the methods for the Collection and Object classes: they are automatically imported because Set is a subclass of Collection and Collection is a subclass of Object. Multiple inheritance carries this saving further.

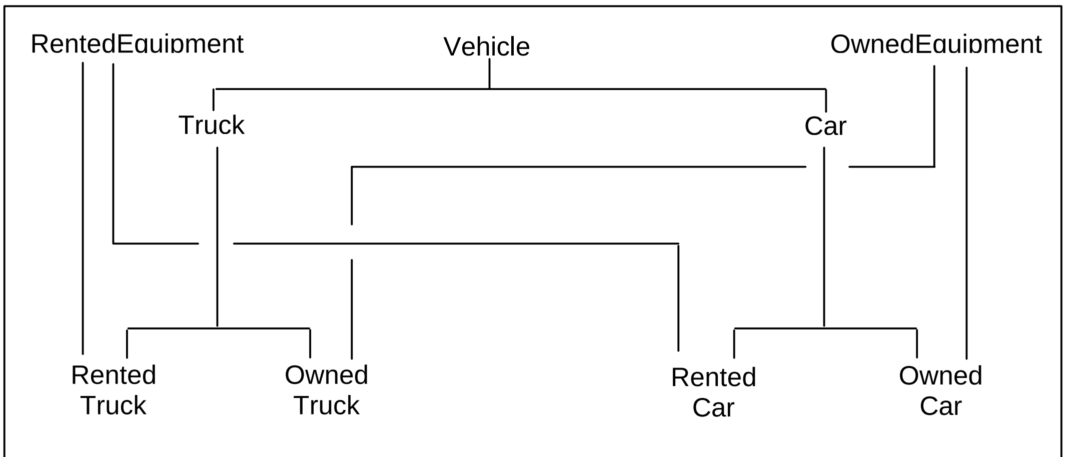
5. Multiple Inheritance

In multiple inheritance, which is provided in Flavors [4] and Eiffel, the hierarchy is not a tree: each class may have several superclasses from which it inherits some of the properties that it specifies for its instances. This can help to carry the saving of repetition much further.

Suppose, for example, that we are concerned with such things as trucks and cars, and also with the fact that the organisation owns some of its equipment and borrows other equipment from rental companies. We might identify such classes as vehicle, Truck, Car, RentedEquipment, and OwnedEquipment. Clearly Truck and Car are subclasses of vehicle, but how are they related to the other classes? Some vehicles are rented and some are owned: so at the lowest level we need classes such as OwnedTruck, RentedTruck, Ownedcar, and RentedCar.

With single inheritance we now have a choice. Either we abandon the classes OwnedEquipment and RentedEquipment and repeat their properties in OwnedTruck and OwnedCar and RentedTruck and Rented-Car respectively, or else we abandon the classes Vehicle, and Truck, and repeat their properties similarly in the lowest-level classes. Neither is attractive.

With multiple inheritance we can make OwnedTruck a subclass of OwnedEquipment and also a subclass of Truck; and we can make RentedCar a subclass of RentedEquipment and also a subclass of Car; and similarly for the other two. In this way we retain the classes we need, and no property need be defined in more than one place in the class hierarchy.



A Class Hierarchy with Multiple Inheritance

Using multiple inheritance in this way reflects an important truth about the application domain. An individual vehicle plays more than one role in the scheme of things: it is not only a particular kind of vehicle, of interest to the fleet operations manager, but it is also an economic good with a particular status, of interest to the finance director. With single inheritance we are compelled to subordinate one of these other; with multiple inheritance we can treat them evenly, and so model the application domain more faithfully.

McAllester and Zabih's concept of Boolean classes [5] takes this idea further still. Their insight is that a class really only defines a cluster of properties, and an individual object can partake freely in any set of such clusters. This is an important conceptual shift, because in essence it demotes classes to a very minor place: an object is still of one class, but that class is a Boolean combination of clusters of properties, and those clusters are more fundamental. Even so, it is possible to regard Boolean classes as no more than a shorthand for ordinary multiple inheritance: if there are N clusters of properties or 'Basic classes' then there are 2^N distinct possible Boolean classes.

6. Reuse and Related Considerations

We may regard the development of multiple inheritance and of Boolean classes as efforts to achieve more effective reuse of software.

Reuse of software is, arguably, the area in which we have been least successful. Certainly, some examples of success can be cited: mathematical subroutine libraries have been widely used since the subroutine was invented in 1949; Smalltalk classes for simple abstract data types (such as Collections) are universally used in the Smalltalk environment, and so are classes for more complex objects (such as windows and menus) that are standardised in the environment. But there is a serious difficulty of a general nature that remains unsolved.

This difficulty concerns the complexity of the units that are to be reused. If a unit is very simple, then it is more likely to be reusable because it is more likely to fit a requirement closely; but its reuse will not be very profitable, and so is not especially attractive. If the unit is very complex, then its reuse would be profitable, but it is much less likely to fit the requirement closely, or even well enough to justify the reuse.

Multiple inheritance, and, more importantly, Boolean classes, can be seen as a way of encouraging the use of simpler units in the definition of object classes, and of making their reuse easier.

7. A Somewhat Different View

A fundamental assumption of the object-oriented disciplines I have mentioned so far has been the permanence of classification. An object is born into a class — indeed, born from it — and remains a member of that class and of no other until it dies. The properties of an object, as evidenced by. its ability to respond to messages, are inherent in it as an individual.

But this assumption clearly runs counter to what we see in many — perhaps most — application domains. The individual who today is a company employee was a job applicant last year, and in some years' time will be a pensioner of the company. The identity of the individual persists from birth to death, but the individual's properties as job applicant, employee, and pensioner are more temporary.

It comes to this. The idea of classification is to do with the properties or attributes of an individual that are constant over some time period. But there is no good reason to assume that this period must always be the whole lifetime of the individual.

Hendler's concept of enhancement [6] goes some way to exploit this insight. Enhancements are classes in the sense that they are clusters of properties, but they are different from classes in two important ways. First, they are not defined in the class hierarchy. second, they are added dynamically to individual objects during their lifetimes. For example, if we have an enhancement that defines the properties and behaviour of someone who is qualified as a PhD, then we can add it to an existing Person object by an operation such as 'Mary enhanceWith: PhD'; from that point on Mary will have the properties and behaviour of a PhD.

More generally, we may take a view of objects that differs radically from the conventional object-oriented view. Only an object's identity is necessarily inherent in it and persistent over the whole of its lifetime. Properties are not inherent in objects, but are attributed to them for various purposes at various times. To use another terminology, we apply different descriptions to one object just as we may apply different grammars to one string.

8. Transient Properties

Applying a description temporarily to an object to serve a temporary purpose is a standard technique for solving problems. It is also a standard technique for reuse: we reuse old descriptions, and the inferences that may be drawn from them, by applying them *ad hoc* to new objects. Unfortunately, this technique is not well supported by most software development environments, and our practice of it is fraught with unnecessary obstacles.

Consider, for example, depth-first traversal of a graph, where the graph nodes and arcs are already in existence. Consulting any standard text on graphs, we find an algorithm for depth-first traversal; the algorithm contains phrases like 'mark the node as visited' and 'if the node is not marked as visited'. Clearly, during the traversal we want the nodes to have the property of being markable, but once the traversal is complete, we may have no further need of this property. Essentially, we would like to be able to add the 'MarkableObject' properties to each node before the traversal, and to remove them afterwards.

In this simple case it is easy to see how the difficulty can be overcome. Instead of marking the nodes themselves, we create a new object to serve as the set of marked nodes: the instance variables of this object will be pointers to the currently marked nodes. But this kind of expedient is unsatisfactory in the general case. We need to be able to handle the problem more directly.

We might wonder whether such use of transient properties can be accommodated in a classification hierarchy. Boolean classes can be regarded as a shorthand for multiple inheritance: can transiently applied properties be similarly regarded?

Emphatically, the answer must be No. It would be necessary for the class hierarchy to anticipate every transient use of every property, and the only way to do this would be to ensure that almost every class had almost every property. Quite apart from the dense obscurity this scheme would bring to the inheritance hierarchy, it would make the treatment of individual objects much more cumbersome. Each object would be weighed down with a mass of properties that at any particular time would mostly be dormant: each soldier would carry not only a field-marshal's baton in his knapsack, but also the rest of the field-marshal's heavy and expensive equipment. Worse, objects would often be burdened with properties that were simply not true: a graph object would have the properties of a tree even when it was not in fact a tree.

9. A Summary

It is fruitful to regard software development as the business of making descriptions and applying them to objects: these activities lie at the heart of much problem-solving.

The concept of an object as a uniquely identified individual with a lifetime that has a beginning and an end is appropriate to modelling parts of many application domains. During an object's lifetime we may need to apply different descriptions to the object: partly because the object itself changes; partly because its relationships to other objects change; and partly because our needs and purposes change.

It is therefore impossible to bind ourselves to apply a fixed set of descriptions to an object throughout its lifetime, and this is what we do when we treat it as being of one fixed class from birth to death. Any discipline, such as the current standard object-oriented discipline, that restricts us in this way is causing us harm and preventing proper modelling of our application domains. It is in this sense and for this reason that we may consider classification to be harmful.

References

[1] William Kent; *Data and Reality*; North-Holland, 1978.

[2] Adele Goldberg and David Robson; *Smalltalk-80: The Language and its Implementation*; Addison-Wesley 1983.

[3] B Meyer; *Object-Oriented Software Construction*; PrenticeHall 1988.

[4] David A Moon; *Object-Oriented Programming with Flavors*; Proc OOPSLA'86; ACM Sigplan Notices 21(11) November 1986, ppl-8.

[5] David McAllester and Ramin Zabih; *Boolean Classes*; Proc OOPSLA'86; ACM Sigplan Notices 21(11) November 1986, pp417423.

[6] J Hendler; *Enhancement for Object-Oriented Programming*; ACM Sigplan Notices 21(10) October 1986, pp98-106.