Critique of Object-Orientation

Object-oriented Analysis

around computer representations of these oriented designers usually do not speak discussions of methods to find objects: if ity being modelled, the objects are just software objects will simply reflect these external objects.'

But perhaps Meyer has got a bit carried away here. It's certainly true that objects are INDIVIDUALS, and that the world is full of individuals. Sensors, devices, airplanes, employees, paychecks and tax returns are all individuals, because you can say things like 'I mean this paycheck, not that paycheck'. Recognizing the importance of individuals in our view of the world has been a crucially important contribution of object-oriented programming languages. Before Simula 67 and Smalltalk, most commonly used programming languages - notably Fortran and COBOL - had no useful notion of individual at all.

But the idea of an object is a programming idea, and doesn't fit most of the individuals in the world very well. Airplanes, paychecks, and tax returns are certainly individuals. But they are not really objects in the usual object-orientation sense. When did you last send a message to a paycheck? What reply would you get back if you sent a message to an airplane? What methods does a tax return perform in response to messages it receives? Steve Cook and John Daniels, in Designing Object Systems - one of the best books about object-oriented development - ask: When the sun rises, does it send a message to each bird to tell it to start singing? They conclude rightly that it's a nonsensical question.

Well, you may say, messages and methods are often used just as a mechanism for getting at the properties and attributes of individuals. Surely we can disregard the details of the mechanism? Perhaps. But there are other, more serious, problems. Here are three restrictions that can cause difficulty in the style of object orientation most commonly practised in analysis and specification:

- **Each object belongs to a fixed class**, determined when the object is created. But the world is not like that: pupils become teachers, bills become laws, partnerships become corporations, doctors become lawyers, cotton mills become offices or hotels, and caterpillars become butterflies.

- **Each object class inherits properties and behaviour from just one class at the next level up in the tree.** That's single inheritance. But the logistics manager wants to classify the company equipment as production plant, office equipment, and distribution vehicles. The finance director classifies it as owned, rented, and leased. The two classifications can't coexist in the same single-inheritance hierarchy.

- **Objects are reactive rather than active.** If you don't send a message to an object, it won't do anything. But the world is full of individuals - like
languages followed, including Eiffel and, of course, C++. In many application areas, especially those involving window interfaces, object-oriented programming has been spectacularly successful.

Inevitably, some people began to advocate an object-oriented approach to analysis and specification. What was succeeding so well in programming should surely succeed in the earlier stages of development. By adopting a uniform technology it should be possible to achieve a seamless progress in development, avoiding the dislocation that usually accompanied the transition to programming from analysis and specification.

Another reason offered was that objects would combine the virtues of data-oriented and process-oriented descriptions. A data-oriented point of view sees the world in terms of entities and relations and attributes; and a process-oriented point of view sees it in terms of behaviours built up from events and processes. Neither is enough on its own. Object orientation would combine the descriptive power of both, and throw in some extra benefits of its own – encapsulation and polymorphism and inheritance.

So object orientation in analysis and specification became cast in the role of a unifying phenomenology that accounts for everything we might see in the world. Everything is an object. The only phenomena are the phenomena of objects. Objects completely subsume entities: like entities they are individuals; they have instance variables, which are something like entity attributes; and like entities they are organized in classes and subclasses. Objects can play the part of relations, too. If you have Course objects and Lecturer objects, you can relate them by Taught-By objects, which have references to Courses and Lecturers as values of their taught and teacher instance variables. In some versions of data modelling a relation can have attributes; in the same way, a Taught-By object could have an instance variable popularity to represent an attribute of the relation, in addition to the variables that represent the related entities.

Objects also subsume events and processes. An object has a repertoire of operations, usually called 'methods', which it performs in response to 'messages' sent by other objects. Sending and receiving a message can be regarded as an event. Like a sequential process, an object can have a behaviour that restricts the events it is willing to participate in at any particular point in its history.

Serious advocates of object orientation don't see objects as a combination of entities and events and other phenomena, but as the most natural phenomena of all in their own right. And they see them everywhere. Bertrand Meyer says in *Object-oriented Software Construction*:

'When software design is understood as operational modelling, object-oriented design is a natural approach: the world being modelled is made of objects – sensors, devices, airplanes, employees, paychecks, tax returns – and it is appropriate to organize the model
people, and vessels in chemical plants and government departments - that do things spontaneously.

All of these problems have programming solutions. There are more powerful object-oriented programming languages offering multiple inheritance, delegation, dynamic object classification, concurrency, and objects that execute sequential processes throughout their lifetimes. Those languages may be very good for writing programs. We are free to build the machine to our own design. But the world outside the machine is simply too rich, too capricious and too recalcitrantly multifarious to be captured in this way.

The Package Router Problem

The package router problem appears in Swartout and Balzer's paper *On the Inevitable Intertwining of Specification and Implementation*. The problem and its description are full of interest. Here is the problem description as Swartout and Balzer give it, translated from the original German version of Hommel:

'The package router is a system for distributing packages into destination bins. The packages arrive at a source station, which is connected to the bins via a series of pipes. A single pipe leaves the source station. The pipes are linked together by two-position switches. A switch enables a package sliding down its input pipe to be directed to either of its two output pipes. There is a unique path from the source station to any particular bin.

'Packages arriving at the source station are scanned by a reading device which determines a destination bin for the package. The package is then allowed to slide down the pipe leaving the source station. The package router must set its switches ahead of each package sliding through the pipes so that each package is routed to the bin determined for it by the source station.

'After a package's destination has been determined, it is delayed for a fixed time before being released into the first pipe. This is done to prevent packages from following one another so closely that a switch cannot be reset between successive packages when necessary. However, if a package's destination is the same as that of the package which preceded it through the source station, it is not delayed, since there will be no need to reset switches between the two packages.

'There will generally be many packages sliding down the pipes at once. The packages slide at different and unpredictable speeds, so it is impossible to calculate when a given package will reach a particular switch. However, the switches contain sensors strategically placed at their entries and exits to detect the packages.