From Structured to Object-Oriented Programming

ir. J.J. van Amstel

"Dramatic changes are on the horizon which have the potential to fundamentally reshape the systems profession. These changes are likely to significantly affect the jobs of nearly all programmers and analysts. Object Oriented Programming (OOP) is forecast to produce a 10-25 times improvement in the productivity of systems and software developers, and some industry observers claim that it holds the promise of substantially solving the quality, cost, and on-time delivery problems for which the systems profession has been so severely criticized". [1]

Buzzwords

The world of software construction bristles with buzzwords and promising ideas. The quotation above gives a nice example. Maybe the use of buzzwords is characteristic for each new field that has not yet reached its maturity. In his article Mr. Howard continues: "It is easy to be sceptical - there is a long litany of productivity techniques that have not fulfilled their promise. Structured analysis once was touted as *the* solution to the software development problem, but structured techniques have yielded only about a 10% - 15% productivity improvement. Fourth generation languages, once expected to transfer much of the responsibility for application development to users, have loaded mainframes unmercifully, but have not significantly reduced the application development backlog. Artificial intelligence has produced rapid prototyping tools, but has made no inroads into the process of designing and developing large business transaction systems. Given the disappointing history of these and other 'solutions' to the systems development problem, it is easy to be sceptical about Object Oriented Programming. It is tempting to reject the claims as hype."

Methods

A method is an explicit (often non-deterministic) prescription for an activity, or set of activities, as required by a selected approach for software development. We must bear in mind that a method is not a recipe that can be followed blindly. A method, when applied to the right problem and used by the right people, will, with a high degree of probability and with a predictable amount of resources, lead to a solution of the problem. A technique is the deterministic part of a method. The notion of method is of a hierarchical nature. This implies that any method will rely on other methods (and techniques) for the sub-problems identified by the method. If we take the top level view of methods of software development, we arrive at what is sometimes called a methodology (particularly in the American literature). The latter can be viewed as an approach to successfully run a software development project, from the conception to the end. Methodologies generally include a host of methods and techniques that are brought together in a more or less coherent framework. Various aspects of a method can be identified.

- *The way of thinking peculiar to the method; the philosophy of the method.*
  The way of thinking must explain the ideas and theoretical principles on which the practical techniques are based. It must give practical, usable definitions of the terms used in the method, and it must explain how to recognize the abstract phenomena it describes.

- *The way of working of the method.*
  The way of working describes the way the philosophy is translated into a set of practical techniques which assist in the analysis, design, and implementation of software systems. The working procedure should say which techniques are relevant to which activities of the software development process (the pragmatics).

- *The way of representing.*
  The way of representation describes the notations of the documentation to be produced, and at which stage of the development process each document should be produced. It should also say how the documentation is to be organized. Each document must be fully

described. The formal definition of any languages it uses should be given.

Structured Programming

In the seventies the word 'structured' was much in vogue. It started with structured programming, then came structured analysis and structured design. The word 'structured' became a synonym for 'good'; it was a kind of hallmark. Often used descriptions of structured programming were on the level of representation.

- do not use `gotos`, or
- use those programming constructs that have a single entry and a single exit.

These descriptions are more statements about the resulting program than statements about the way of programming. (There were even people who wrote 'structured programs' by first writing the programs with `gotos` and then removing these `gotos` afterwards.) Over the years we have learned the essential aspects of structured programming, of which the mentioned properties of programs (no `gotos`) are a result.

- *Way of thinking.*
  A program is a formal text defining transformations from states into states. These states can be described by predicates. Thus a programming construct can thus be seen as a predicate transformer.
- *Way of working.*
  We use the predicates and their transformations to find the (correct) program. Program construction and correctness considerations go hand in hand.
- *Way of representation.*
  To write (correct) programs, we use those programming constructs, which realize the transformations that have been recognized in the *way of working*.

An important concept in the context of structured programming is *functional* (or *procedural*) *abstraction*. In the development of a program 'high-level' programming constructs (procedure calls) are used to realize certain effects, laid down in the pre- and post-conditions of the constructs. The pre- and post-conditions are then used as the specification of a separate module: the function or procedure declaration, the implementation of the 'high-level' construct. In this way, we work on different levels of abstraction. Information hiding and the use of parameters have to guarantee that the function (procedure) can be constructed independently of the way it is used in a particular environment and that its integration into a larger program unit may be accomplished without knowledge of its inner mechanisms (laid down in the body of the procedure). The module should communicate with the outside world only through a well-defined interface. Of course, we had already before the rise of structured programming the subroutine construct. But the subroutine is just an encapsulation: a unit that contains one or more programming items, without the explicit intention of information hiding. The purpose of the use of subroutines was not abstraction, but code sharing.

Data Abstraction

In structured programming the emphasis is on the abstraction of *actions*. The next step is the abstraction with respect to data. What we need is not only the possibility to abstract from the simple actions, but we must also be able to abstract from the simple data in our programs. This can be done by bringing together in one module data and the operations on the data: a type. The interface is given by the name of the type and the operations. The operations are specified by pre- and postconditions (see above). The specifications are the description of an *abstract data type*. In the program the values of the type can only be used via the defined operations. The internal representation of the values and the implementations of the operations are encapsulated in an implementation module for which information hiding is applied.

Take for example a program in which sets of integers should play a role. In the program we use these sets and the operations on them. The operations are defined by their pre- and postconditions in terms of sets: (the specification of) the abstract data type. In the implementation we use a known data type, for example a boolean array, and the operations are realized in terms of this data type.

The next step in the data abstraction idea is the parameterization of the specification. In this way an abstract data structure is created. We can use this data structure just as we can use an array in a language like Pascal: it is a generator for different types by parameterizing it with existing types. So, instead of the definition of the type 'set of in-

tegers', now the structure `set` is defined, which can be used in a type definition to define the type `set(integer)` or `set(character)`, etc.

Object-oriented programming

Now we have object oriented programming and proponents suggest that it is *the* ultimate way of analyzing, designing, implementing, and maintaining complex systems. OOP is said to support modularity and the reuse of software, and systems can easily be extended by using inheritance. But what is OOP? The simple definition, one sometimes sees, is: the use of objects, classes, and the inheritance construction in the development of software. But a class is like an abstract data type, and a variable of an abstract data type is like an object. So, what is new? New is the inheritance construction. But are there no other aspects of OOP that make OOP an attractive way of developing software systems? To answer this question we must find the *way of thinking* and the *way of working* for OOP.

The central theme in the way of thinking is that (not necessarily concrete) entities in the 'real' world are modeled as objects [2]. Abstraction is used to find the objects. Each object has a real entity as its counterpart in the 'real' world. The application domain plays an important role in this modeling process. For a PC the important properties in the world of electronic engineering are quite different from the important properties in the context of text processing. The entity in the real world is the same, but the perception is different for the two application domains.

This *way of thinking* has consequences that are not always taken into account in the various object-oriented approaches in the literature. A striking example is the inheritance construction. One defines, for example, an object like 'vehicle' and uses this object - via inheritance - for the definition of objects like 'motorcycle' and 'automobile', 'vehicle' being the more general object. But in the real world there is no entity that corresponds to the defined object 'vehicle'. So, at the level of analysis, finding the relevant objects, inheritance is of no use. Inheritance can be used at the representation level for sharing code. But in many object-oriented approaches inheritance is introduced as the feature of object-oriented software development. And this construction is then promoted to the level of the *way of thinking* in statements like: "Object-oriented software development is the use of inheritance in the ..." But when we define OOP as the use of inheritance, we make the same mistake as when we define structured programming by forbidding the `goto`.

In the *way of working* we can make use of the way of working of structured programming (notions like encapsulation, information hiding, and data abstraction play an important role [3]), complemented with some new ingredients (like process abstraction).

At the start of this section, the question was asked: What is new in OOP? Now we know the answer: New is the *way of thinking*. If you forget this new way, you will walk on a way that is already known for a long time, or you will get lost.

References

[1] G.S. Howard,
"Object Oriented Programming Explained",
Journal of Systems Management, July 1988

[2] INTOOM Lecture Notes, Method Overview,
Philips Research Laboratories, Information and Software Technology, RWB-519-RE-92024

[3] J.J. van Amstel,
"Van gestructureerd naar object-georiënteerd programmeren",
Academic Service, Schoonhoven, 1990

□

Ir. J.J. van Amstel works at the department Information and Software Technology of Philips Research Laboratories in Eindhoven. He is member of the External Advisory Board of OOTI.