to degrade exogenous p53, although this effect was not seen in cells lacking Mdm2. Treatment of human cells expressing endogenous wild type p53 and Mdm2 with this proteasome inhibitor resulted in enhanced stability of the endogenous p53. An inhibition of Mdm2-targetted p53 degradation could also be seen following cotransfection of these cells with *mdm2* and *p53*. These results strongly indicate that Mdm2 targets p53 for degradation through the proteasome, and suggests that Mdm2 is involved in the normal regulation of p53 stability.

1. Michael, H. G., Kubutat *et al.*, *Nature*, 1997, **387**, 299–303.
2. Donehower *et al.*, *Nature*, 1992, **356**, 215–222.
3. Oren, M., *Behring Inst. Mitt.*, 1996, **97**, 32–59.
4. Oren, M. *et al.*, *Mol. Cell Biol.*, 1981, **1**, 101–110.
5. Fritsche, M. *et al.*, *Oncogene*, 1993, **8**, 307–318.
6. Vogelstein, B. *et al.*, *Cell*, 1992, **70**, 523–526.
7. Cahilly-synder, L. *et al.*, *Soma Cell Mol. Gen.*, 1987, **13**, 225–234.
8. Oliner, J. D. *et al.*, *Nature*, 1992, **358**, 80–83.
9. Wux *et al.*, *Genes Dev.*, 1993, **7**, 1126–1132.
10. Momand, J. *et al.*, *Cell*, 1992, **69**, 1237–1242.
11. Olsen, D. C. *et al.*, *Oncogene*, 1993, **8**, 2333–2360.
12. Montes de Oca Luna, R., *Nature*, 1995, **378**, 203–206.
13. Ygal Haupt *et al.*, *Nature*, 1997, **387**, 286–299.

V. K. Nampoothiri is in the P.G. Department of Zoology and Research Centre, Sanatana Dharma College, Alappuzha 688 003, India.

OPINION

Why does object technology hasten slowly?*

Rajendra K. Bera

The entire evolution in programming languages and programming strategies seems to point to one major effort – how to cordon the scope of data and functions so that they can be accessed, changed and used without causing unintentional side-effects. In short, software engineering is progressively trying to enforce, among software developers, the discipline that mathematicians bring naturally to their tasks while solving problems. Object technology (OT) is the currently fashionable effort in this direction. At its heart is the requirement that the software developer be able to see (and invent) concepts and patterns and translate them into user-defined data types, better known in the literature as abstract data types (ADTs), in terms of which one can effectively describe a problem and its solution.

All programming languages form a branch of mathematics. All requirements analysis is essentially an exercise in mathematical modelling. All software design is a further detailed elaboration of that analysis with a view to developing an implementation strategy; and coding is the step which spells out exactly how the design will be executed. Intellectual control at every step is the key to software development. Analysis, design and coding, although occurring at different levels of abstraction, must ideally be equivalent in a mathematical sense.

*The views expressed are those of the author.

Mathematics has developed over many centuries, has a rich tradition, and at any given time it is served by a good number of extraordinarily talented people devoted to furthering the subject. Mathematicians work without text-editors, debuggers, case-tools and so on. They do not generally commit errors of syntax, type mismatch or make a religion out of reusability even though a very high percentage of their work is in reusable form (theorems, lemmas, methods, data types, etc.). They conceptualize, create abstract data types, encapsulate, inherit and reuse. They go about their job quietly, methodically, carefully, rigorously, painstakingly, and above all, knowledgeably. After completing a piece of work, they take the time to sit back, reflect and make it elegant. They provide a polished document at the end of their endeavours stating what they have done so that others may review, criticize and finally use. Mathematicians generally avoid self-inflicted complexity in their work by being very systematic. They usually focus on resolving domain knowledge complexity. Indeed they already practise what practitioners of OT hope to achieve in the future. *So why are software developers so different from mathematicians?*

The reasons

There are perhaps several reasons. First is the multidisciplinary nature of the field – hardware, software and applications. They involve practitioners from very different academic fields with different traditions, training, prejudices and goals. Indeed, many managers in the software industry have none or very little formal training in software engineering. Consequently, there are problems of depth, focus, traditions, customs, role definitions, etc., all of which remain rather hazy today. What is really needed is an interdisciplinary approach. That is, software practitioners must acquire a broad range of skills that at least span the core areas of hardware, software and applications so as to develop a wider perspective. In particular, professional grade software developers must be trained in operating system design, complier design, programming language design, functional analysis, logic and theoretical physics. Such a training will provide them with the competence to understand and deal with both domain knowledge as well as software development complexities at appropriate levels of abstraction.

Second is the pretence that software engineering can largely manage without a great deal of training in mathematics. It simply cannot in the future.

Third is the poor repertoire of symbols in software engineering. Much of human civilization's efforts in building knowledge has gone into devising expressive symbols to convey concepts and relations among them so that knowledge generation and its transmission can be handled with

ease. The mathematicians, in particular, have evolved it to a fine art. To understand its implication one has to go no further than imagine doing arithmetic in Roman numerals rather than in Arabic. Yet software engineering is still struggling, even at the programming language level, to evolve a rich, concise and an expressive array of symbols, even though many of the symbols it needs can be readily borrowed from mathematics. A major constraint in this evolution has been our anachronistic attachment to the old 'qwerty' keyboard. While any modern DTP software implements the full range of mathematical symbols and many others, programming languages do not! We need a new keyboard and a whole bunch of new symbols if program verification is to become a reality.

Fourth is the deliberate desire to keep programming languages small in the belief that it will encourage the masses to learn programming. However, the same masses learn their mother tongue by memorizing a few thousand words of vocabulary and pick up the nuances of its semantics without any fuss. Basically, people learn when they have a need to learn, rarely because it is easy. Today software engineering is the backbone of the knowledge-based industrial sector. It is no longer a profession meant to be practised by the masses but by the specialists and the highly trained. To cater to their needs programming languages have to be extensively extended and supported through function and class libraries.

Fifth is the dubious practice of teaching people programming languages in a crash course lasting a few days or weeks and then putting them on the job. In contrast, to communicate effectively in a new natural language we go through years of training! Programming languages need to be taught with at least the same care and thoroughness as is done in natural languages, and preferably, as in mathematics. Students need to be exposed to and study programs written by master programmers.

Sixth, and the most serious, is the lack of intellectual giants in the field. There are no Newtons, Einsteins, Poincarés and so on. The field is too young – roughly 45 years old and the majority of its practitioners even younger – which may be a contributing factor. (Incidentally, quantum mechanics made remarkable progress in its first 45 years of existence to become the crown jewel of theoretical physics because of people like Planck, Bohr and Heisenberg!) But the fact that top-flight mathematicians do not feel attracted to contribute to the development of programming languages and programming techniques is a matter of grave concern. They are needed to bring true professionalism to software engineering and to provide schools of thought for the younger generations. Their absence has led to a disturbing trend where software engineers continue to create new, and sometimes, personalized, jargon which is non-standard, confusing and occasionally meaningless. And often, techniques proven and routine in other branches of knowledge are presented as radical innovations in software engineering[1].

OT is a major example of this trend. To quote Meyer[2] (the designer of the object-oriented language *Eiffel*),

> '... if ObjectSpeak confuses you, do not despair: you are not alone. People who have been practising object technology for years feel just as dizzy, and in fact some of those who *invented* the concepts do not necessarily fare much better'.

Another distinguished author, Page-Jones[3] begins his book by observing that *the term object oriented is intrinsically meaningless*. The dictionary meaning of object is: 'A thing presented to or capable of being presented to the senses'. The word oriented means 'directed toward'. Object-oriented usually appears as an adjective. Thus he concludes,

> 'Object-oriented: Directed toward just about anything you can think of.'

And,

> 'No wonder the software industry has failed to alight upon an agreeable definition of "object-oriented". And no wonder that this lack of clarity has allowed any peddler of softwares to claim that his shrink-wrapped miracles are "object-oriented".'

Such confusion and skepticism is unusual for a field that claims to provide a better paradigm for software development than in the past, emphasizes the supreme role of concepts and abstractions and when for its terminology, notations and symbols, it can readily draw upon the storehouse of mathematics. Surely, the central theme in OT is

> *The practice of mathematical modelling; if necessary, by inventing new data types along the way.*

This is a more demanding task than practising classical mathematical modelling where one generally assumes that one will work with available and well-understood data types. Inventing new data types is a demanding intellectual activity. Not many of us would have invented complex numbers, vectors, matrices, etc. Data types deal with operands and operators. The so-called class in OT is a data type, an object is a variable of a given data type. In OT, the data type *integer* and the integer variable *k* would be known, respectively, as class *integer*, and object *k*. And to sound exotic, one would also say that *k* is an instantiation of class *integer*! One also calls a function a *method* (and what we usually understand by a method is called a *methodology*) or a *service*, and a function call is a *message*. In essence, a class is nothing but a list of operands and a list of operators meaningful to those operands, syntactically expressed as a unit called a data type.

The concepts behind encapsulation, inheritance, reusability, etc. are neither new to mathematicians nor to procedural programmers, but they seem to have been recently rediscovered by the OT community. These concepts are often understood intuitively and certainly well practised by the mathematicians (since at least the past two thousand years; recall that Euclid's book on geometry, written two thousand years ago is still used practically in its original form) and professional programmers and in just about any established profession or branch of knowledge – because these are the pillars around which all knowledge is built. The subroutine, invented in the 1940s, introduced encapsulation of code into procedural modules; inner blocks in C codes inherit the variables of the outer blocks; and everybody reuses function libraries.

Till it is firmly accepted that serious OT requires high level mathematics and that today's average programmer does not have the requisite mathematical training to practise OT, we will continue to wonder and debate why OT is not taking off[4]. Vlissides[5], I think, neatly sums up the present situation,

> 'Yes, object technology is still emerging. It's not clear how many people who think they're doing object-oriented

programming actually are doing object-oriented programming. I think it's a small percentage, so the signal-to-noise ratio is fairly low'.

And,

'It just hasn't become a productive medium for the majority of programmers and designers out there. People have a lot of mechanism thrown at them, but they can't put that mechanism to use in systems that make good on the promise of object technology: that is, to gain reusability, flexibility, extensibility and elegance in their software.'

All this will change if software developers are trained in mathematical modelling.

Seamless software development

Traditionally, software engineers have looked upon software development projects as consisting of the following steps:

- eliciting user requirements;
- analysing those requirements;
- designing the software architecture;
- coding the software for implementation.

Essentially analysis, design and implementation deal with three different perspectives of a given problem viewed from an hierarchy of conceptual levels. Hierarchies localize decisions and correctness demonstrations. That is how we conquer and control complexity and maintain intellectual manageability. Well-conceived hierarchies allow the trained mind to jump from one level of hierarchy to another with ease. However, what complicates the software developer's task enormously is the fact that different terminologies, notations, symbols, etc., are used in the different steps, which, in large projects, force developers to compartmentalize their thought processes, severely constrains their ability to traverse across hierarchies and virtually mandates that the development team deal with the steps one at a time. The result is an enormous load of documentation created at each step and the frequent lack of compatibility among these documents. Moreover, post-fixing an error in a step is a major and often nerve-racking exercise. Consequently, band-aid fixes are common, and the software and its related documentation slowly and surely degenerate to unmanageable levels from both intellectual and developmental points of view. Essentially software construction is being handled in the same way as one would handle the construction of a building!

On the other hand, when applied mathematicians deal with a user problem they make a mathematical model to capture the essence of the problem, decide upon the methods to solve the problem, and solve the problem. They do all this seamlessly by the simple expedient of using the same terminologies, notations and symbols throughout. Therefore they are able to produce a single document from problem definition to problem solution and go back-and-forth in their steps with far greater ease and intellectual control than is currently possible by the software developers. Mathematicians long ago realized and exploited the fact that their products are not physical products but abstract constructs of the mind. Software developers have yet to realize this in any real sense.

I believe that the problem of symbology is the biggest hurdle in the path to producing quality software products. If this problem is addressed with vigour, software development and verification would eventually become much more simple. Indeed one would then be practising mathematics. And, indeed, one would no longer need ISO certification to assert the quality of one's software product – it would be verifiable to mathematical standards.

The OT myth

A general myth in software engineering is that the unit of programming in procedural programming is the function and in object-oriented programming it is the class; that C programmers concentrate on writing functions and C++ programmers concentrate on creating their own user-defined types, called classes; that procedural programmers concentrate on the verbs and the object-oriented programmers concentrate on the nouns. This is really perplexing. *OT is not about objects but about abstraction.* When Newton discovered the law of gravitation, he was not concerned about the apple but the falling of any object in a force field. When engineers design a control system they do not concentrate on the physical objects constituting the system but on their abstract representation, the Laplace transform of their dynamics, and the manipulation of polynomials. Naively concentrating on the real-world nouns would not have got either Newton or control system designers very far.

In *all* programming, the macro unit of data is the structure (or record) and the macro unit of action is the function or an operation on data. Together they form a class of operators and operands – mostly implicitly in procedural programming and mostly explicitly in objected-oriented programming. Unchanging data by itself is not interesting (unless they are universal constants such as the speed of light, the value of π, etc.) but transformation of data is. Niklaus Wirth's aphorism 'Algorithms + Data Structure = Programs' essentially describes *all* software. Data is transformed by action, that is, functions. Functions are where all the interesting action is! Classes are merely a good synthetic means of clubbing data structures and related actions together, specially if they can be used in a well-defined context. Data structures do not inspire action, interesting actions (specially coordinated actions) inspire data structures. A class is interesting only if it contains functions that do interesting things, not because it contains data. The more versatile or generic the actions, the more the likelihood of its supporting a variety of data structures. That is why templates in C++ are a more powerful syntactical unit than functions and classes[6]. In templates, the unquestionable focus is on generic action. It is action which holds data together and gives meaning to it.

Once the programmer has selected and created his/her data types, he/she reverts to procedural programming because that is how all mathematics is done. The vast majority of object-oriented programming *is* functional decomposition *and* procedural programming. How else are its methods programmed? And why will classes be important but for the methods they contain?

If the science and art of programming is to progress, the standard data types available in a programming language should be expanded greatly to include at least complex, vector, matrix, set, string, list, etc. Their implementation should really be the task of compiler writers and not application programmers. I believe that software engineering will come of age when a programmer will not have to struggle to implement what a mathematician takes for granted when solving

a problem. When all the data types that a programmer needs become available in class libraries, then the distinction between a procedural program and an object-oriented one will vanish.

The OT hype

Though OT has been much talked and written about since the 1980s, the software community remains polarized into two camps: the cynical, sniggering object-oriented *reactionaries* claiming that[7] 'Nothing significant in software has happened since the sixties!' and 'All this object-oriented malarkey is just the same old stuff that we've always done, but with a few fancy name changes'; and the die-hard, hot blooded object-oriented *revolutionaries* claiming that 'Anything known before 1980 isn't worth knowing!'

Most of the claimed success stories in OT are anecdotal without any clear evidence that alternative programming styles (for example, structured programming) adopted by competent software developers would not have done an equally good or an even better job. Furthermore, the OT literature does create the impression that procedural programming is practised by the naive if not the moron and doing object-oriented programming somehow makes you clever. An example[8] from one of the *gurus* of OT,

> 'Don't be quick to assume that your system falls in a category where OOA [object-oriented analysis] is not helpful. For example, consider an aircraft simulation system. A traditional approach would be to build a single, gigantic event-driven simulation (a big algorithm, not well-partitioned). Yet an object-oriented approach partitions the system into parallel subsystems (electrical, mechanical, hydraulic and the like); each subsystem follows a similar pattern; and each subsystem has many Class-&-Objects.'

Interestingly, flight software developers, without using OT, have always worked at the cutting edge of software technology (real-time, fault-tolerant, distributed, mission critical), contributed enormously to its growth and development, and have helped put a man on the moon! That such people cannot even do a decent job of partitioning a software which an object-oriented *approach* (not a domain expert!) will routinely do is truly incredible.

We know that doing mathematics does not make people clever, clever people do mathematics and they try to be objective (and not object-oriented!) in the way they attack a problem. Similarly, only clever people do serious software development. Object technology would be far more comprehensible and better served if the jargon and the hype were removed and the equivalences between an object-oriented approach and a procedural approach were clearly brought out.

It is not object technology which is the great breakthrough in software engineering as some claim, but the invention of programming languages which allow *professional programmers* to define their own data types safely and conveniently and which lets them select the style of programming suited to the problem at hand. For professional programmers the biggest event in recent times, in my view, is the creation of C++ by Bjarne Stroustrup. In some future version of C++, I am sure, many are waiting to see it extended to allow user-defined operator symbols and user-specified associativity and precedence rules for operators. It will then allow programmers to deal directly with mathematical models much more effectively than is possible now.

1. Although F. P. Brooks, Jr., in his book *The Mythical Man-month* (Addison-Wesley Pub. Co., Mass., 1982, p. 14) makes a similar comment in the context of why software projects go awry, the observation is generally true for all of software engineering.
2. Meyer, B., *Object Success*, Prentice Hall, London, 1995, p. 1.
3. Page-Jones, M., *What Every Programmer Should Know About Object-Oriented Design*, Dorset House Publishing, New York, 1995, pp. 1–2.
4. An ACM convened Industry Advisory Board sponsored by IBM's Object Strategy and Implementation Group met in 1995 to discuss future applications of object technology in industrial settings. In particular, they addressed the question, 'Is object technology still emerging, and why is it taking so long to gain acceptance?' The discussions were reported in *Communications of the ACM*, October, 1995.
5. *Communications of the ACM*, October, 1995, p. 39.
6. We are essentially seeking to increase the scope of polymorphism of a method so that a method is meaningful over as large a set of classes as possible.
7. The quotes are from Page-Jones, M., *What Every Programmer Should Know About Object-Oriented Design*, Dorset House Publishing, New York, 1995, pp. 55–56.
8. Coad, P. and Yourdon, E., *Object-Oriented Analysis*, Yourdon Press, New Jersey, 1991, p. 32.

Rajendra K. Bera is a Consulting Software Specialist in IBM Global Services India Pvt Ltd, Golden Enclave, Airport Road, Bangalore 560 017, India.