Tour de force tells all about objects

Object-Oriented Software Construction by Bertrand Meyer (Prentice-Hall International, London, 1988, $37, 534 pp.)

Ronald D. Levine, Dorian Research

What is object oriented programming, and why should I be interested in learning how to do it? Even though numerous articles and a few books treating these questions have been published during the past several years, it has not been easy for the uninitiated to arrive at a clear and complete picture of the essence of object based software design, which is a complex set of concepts.

Now that object-oriented design is attaining buzzword status, there is a bandwagon effect, with proponents of several distinct software subcultures each presenting their own system as the object oriented ideal.

With the enormous influence of the Smalltalk language, it is perhaps not surprising that to more than a few computer professionals, "object-oriented" involves, in an essential way, the use of multiple windows and icon-based user interfaces. To others, the essence lies deeper in Smalltalk, particularly in its completely dynamic binding and related messaging syntax for procedure calls: All things are objects, even classes, even numbers. To perform a simple arithmetic operation on a number. you send it a message.

However, there are many other programming systems that may rightfully claim to be object-oriented but that do not use icons, do distinguish between classes and objects, and do insist on static typing, and allow early binding. Other languages, particularly Ada and Modula2, are frequently characterized as object-oriented because of their modularity: They support encapsulation and information hiding. But many people would exclude these languages from the object oriented fold because they lack other essential ingredients, most notably inheritance.

Bertrand Meyer's purpose in *Object-Oriented Software Construction is* to show that, behind the buzzword, objec-oriented design has real substance in terms of the goals, principles, and practical problems of software engineering. Although it is different from the methodology most people use today, it is exciting because it has the potential to significantly improve software quality.

Meyer succeeds admirably in leading the patient reader to these conclusions through a presentation of the fundamental software development issues that is independent of any programming system, language, or application area.

To be sure, the book introduces the Eiffel language that Meyer designed, and it describes the positive features of the software-development system that his company sells. However, you can well regard Eiffel, which is a small language, simply as a convenient notation for codifying the results of Meyer's language-independent analysis.

It is natural to compare this book with one of its predecessors of the same genre, Brad Cox's *Object-Oriented Programming: An Evolutionary Approach* (AddisonWesley, 1986). One of the most striking differences is Meyer's careful, analytical, and logical style of developing his subject - this contrasts sharply with Cox's tendency to rely on very informal definitions through examples and on justification through anthropomorphic analogy. As a mathematician, I especially appreciate the precision of Meyer's definitions and the clarity of his reasoning about the relationships among the central concepts.

Almost twice as long (534 pages) as Cox's, Meyer's book explores many more facets of the object oriented paradigm and exposes its nuances more deeply. It is also more suitable for use as a course textbook because of its short set of substantive exercises at the end of each chapter.

Finally, Meyer's book is, in a sense, much more explicit about the evolutionary approach to object-oriented programming: Each chapter contains a section of bibliographical notes tracing the history of the development of the chapter's theses.

Meyer adopts, as first principles, the desirability of certain external factors of software quality that no one would dispute: correctness, robustness, extensibility reusability, and compatibility In subsequent presentation, Meyer measures the relative importance of various internal features of software construction such as modularity, structure, and visibility control against these external qualities.

He analyzes modularity, which is a desideratum in virtually every formal and informal description of every software engineering methodology, in detail according to five criteria: decomposability, composability, understandability, continuity, and protection. These criteria, he argues, then lead to five principles of modularity: linguistic modular units, few interfaces, small interfaces, explicit interfaces, and information hiding.

From a presentation on the scope of the meaning of "reusability," Meyer derives further requirements on modular components, such as the ability to accommodate variations in type, which leads eventually to the object-oriented notion of genericity.

Meyer helps the reader take the big step toward object orientation when he provides several arguments that all these criteria and requirements on modularity are better served by organizing the construction of software around the data structures than around the functions that operate on them. His eventual technical definition of object-oriented design is "the construction of software svstems as structured collections of abstract-datatype implementations."

With this definition in mind, you may naturally have several questions.

What is abstract about these data types? I find Meyer's treatment of the term "data abstraction" the most lucid I have read. To most authors, an abstract data type is simply a user-defined type as opposed to a simple type built in to the language; it is a data structure encapsulated in a module with the procedures or functions that operate on it. Meyer makes it clear that abstraction is required by the principle of information hiding-the meaning of a type is distinct from and logically precedes any implementation of it.

Where, then, *is* this meaning specified? Surely not in the member function prototype declarations of C++ (because these have no semantic content), nor in the definitions of those functions (because the definition is implementation-dependent and must remain hidden). The answer lies in the notion of a formal type specification, an abstraction of the essential semantic properties of the type, which may be expressed by adding some axioms to the function prototype declarations.

In the Eiffel language, the axioms may be made manifest in Boolean expressions called assertions that are used as preconditions, postconditions, and class invariants. These assertions are one of Eiffel's unique features and help to ensure correctness, the most important of he external factors of software quality. Meyer uses the word "structured" in his definition to refer to two primary structuring relationships: the relationship between the provider of code and his client, and the all-important inheritance relaionships that provide the means to specialize and extend types. Meyer refines his definition by imposing further requirements on these structuring reladonships. For example, types should be identified with modules, called classes, and inheritance should be multiple and repeated The notion of repeated inheritance which also seems to be unique to Eiffel, deals with the problem of inheritng features from one ancestor by more han one path.

This is a good place to find a relatively complete and balanced answer to the question of what object-oriented design is.

The first part of the book is devoted to hese general issues, ending with the definition of object-oriented design. The second part helps you learn the practice of object-oriented design with case studies, gives further advice on techniques, and introduces the Eiffel language and development At the same time, the second part continues the theoretical development by refining some of the notions introduced in the first part and by introducing new issues such as exception handling and memory management.

The third part of the book treats, very briefly, the application of object-oriented techniques in other languages: first, classical languages (Pascal, Fortran, and C), then some newer languages that are object-oriented at least to some degree (Ada, Modula-2, Simula, Smalltalk, C++, and Objective C) , and finally just the briefest mention of the Lisp extensions (Loops and Flavors).

The only blatant technical error I've noticed in the book occurs in the section on C++, where it is stated incorrectly that attributes (data members) in C++ classes may not be exported (be made public).

Meyer stresses an important point that some other writers have missed: Object-oriented design has a decidedly bottom-up character that is antithetical to the well-known principles of top-down functional decomposition, the buzzword of a earlier era. He gives convincing arguments that such a bottom-up approach is the only way to realize reusability and the other software-quality factors, and goes so far as to discount the validity of the topdown approach for any realistically complex system. "Real systems have no top," he says.

This book is intended for experienced software practitioners, both professionals and advanced students. It is well organized and highly readable. Meyer's high standards for precision of expression do not interfere with a literate style or prelude the occasional injection of humor. recommend it as a good place to find a relatively complete and balanced answer o the question of what object-oriented design is.