Object technology in perspective

by G. Radin

Since its beginnings half a century ago, the technology applied to the development of software has continually evolved. Object technology is the result of a long progression of improvements, from the closed subroutine, through structured development techniques and data abstractions, to object-oriented languages, design patterns, and frameworks. In this essay, the author reflects on this evolution, specifically in the areas of development productivity, software maintainability, and paradigm consistency.

This issue of the *IBM Systems Journal* is devoted to papers that address various aspects of object technology. In my role as coordinator for this theme issue, I want to discuss why the technology deserves this level of focus. Much has been written about object technology, at levels varying from high-level overviews for business executives to detailed C++ code segments that illustrate a sample implementation of a design pattern.[1,2] This introductory essay is not yet another high-level tutorial about encapsulation, inheritance, and polymorphism. And it certainly does not include any C++ code. Instead it is an attempt to clarify the scope of this technology and to identify some of the challenges associated with its exploitation.

One reason that the term "object oriented," or "OO," is often confusing is that it is applied so widely. We hear about object-oriented user interfaces, object-oriented programming languages, object-oriented design methodologies, object-oriented databases, even object-oriented business modeling. A reasonable question might be: Is this term used because OO has become a synonym for "modern and good," or is there really some substantial common thread across all these object-oriented things?

I believe that there is such a common thread, and that it makes the object paradigm useful in all these diverse areas. Essentially it is a focus on the "thing" first and the action second. It has been described as a noun-verb way of looking at things, rather than verb-noun. At the user interface, first the object is selected, then the action to be performed on the object. At the programming language level, an object is asked to perform some action, rather than a procedure called to "do its thing" on a set of parameters. At the design level, the "things" in the application are defined, then the behavior (actions) of these things is described.

Because we see this common thread as a unifying concept, we have included in this issue papers that encompass many different aspects of object technology.

Object technology provides significant potential value in three areas, all closely related: productivity, maintainability, and paradigm consistency. I have deliberately said "potential" here, because, while there are already many projects that have benefited from its use, object technology is not as pervasive in the information technology world as, for example, personal computers or the Internet. The Internet is not yet as pervasive as third-generation languages or structured programming, but the whirlwind pace of its acceptance is impressive, and it allows us to think in new ways. Object technology will play an important role here in many ways, some of which

0018-8670/96/$5.00 © 1996 IBM

are not yet clear. OO programming languages, such as Java**, provide an example of the use of object technology on the Internet.[3]

Productivity

Many technologies have contributed to productivity improvement in application and system software development. High-level programming languages, such as COBOL and FORTRAN, led to a major breakthrough in productivity from the 1960s until today. In fact, it can be argued that most of the significant improvements in programmer productivity are associated with the evolution of programming languages. Symbolic assembly languages gave way to high-level programming languages, once it was demonstrated that compilers could produce acceptable code. High-level languages, such as Modula-2 and Ada, incorporated the notion of "data abstraction" that underlies the encapsulation aspect of object-oriented languages. Easier-to-use languages like REXX (Restructured Extended Executor) and BASIC allowed the rapid development of many applications. In fact, those of us who have worked in the programming language world often view object technology as just another step in the evolution of programming languages.

Productivity has also been dramatically improved by the evolution of design methodologies. When I began to write programs (in the mid-1950s), developing flowcharts before coding was a radical new idea. Gradually technologies such as data flow diagrams, process decomposition, and entity-relationship data models allowed programming teams to develop correct programs more effectively. Again we can argue that object-oriented design methodologies are evolutions of successful design techniques.

But at this point, further significant improvements in productivity cannot happen simply by making analysis, design, or coding more efficient. Given the exponential demand for new applications that exploit new hardware and networking, the need for businesses to compete by offering unique new value to their customers, and the radical increase in the performance of computers, there are just not enough programmers available to write all the code required. We must change application development from a people-intensive discipline to an asset-intensive discipline. That is, we must encourage and make feasible the widespread reuse of software components. It is exactly in this "reusable component" arena that object technology can contribute significantly.

Certainly, the advantages of reusable components are not just now being recognized. Alan Turing[4] used (invented) subroutines in the 1940s, and one could argue that everything else has been just a set of incremental improvements. There have been many "application customizer" products and projects in the past 30 years. And the "package" concept of Ada was explicitly invented to allow reuse. In fact, I believe that reuse has been extraordinarily successful. When senior citizens like myself first began programming, we were presented with a bare computer. We had to write our own input/output packages, our own loaders, etc. Operating systems are great examples of real reuse. Subroutine libraries, such as mathematics libraries, are other examples. But object technology gives us the potential to extend the scope of reuse beyond systems, subsystems, and low-level subroutine libraries to components that constitute elements in the application itself. The recent growing interest in "business objects"[5] is a good example of this.

The aspects of object technology that help in reuse are encapsulation and inheritance.

The aspects of object technology that help in reuse are encapsulation (which allows the developer to see a component as a "black box" with specified behavior) and inheritance (which encourages the reuse of code to implement identical behavior among different kinds of objects). Polymorphism[6] allows the application itself to be reused when bound to different kinds of objects that support identical behavior. For instance, an application that provides messaging capability can be reused in environments supporting different network protocols, if those protocols are accessed as objects. (I guess I broke my promise not to discuss these terms.)

Reuse can occur at many stages in the development process. Business objects can be reused in design, and later in source or even binary forms. But there are many inhibitors to reuse, only some of which are technical.

We need intelligent search engines so that we can find the objects that are potential candidates for reuse. We need natural ways to describe the behavior of objects, and the descriptions must be precise enough that the resulting application does exactly the right thing. The converse of this may be even more effective, namely to follow the lead of manufacturing industries and standardize components, so that reuse can happen with little or no customization. The insurance and banking industries are moving in this direction by defining standard business objects.

We often will need to customize an object that is not quite what is needed. It is very important to understand that objects will not generally be reusable unless they have been written with reusability in mind, often at the expense of performance, size, and complexity. Customization can be made possible in many different ways. The reusable object itself can be created so that many expected variations are available within the object. Object-oriented "frameworks" allow customization by subclassing and overriding default methods. And even black box components can be customized by wrapping "scripts"[7] around them.

We need a run-time facility to allow late binding of reusable components to applications, so that several applications can share code. Dynamic link libraries (DLLs), for example, provide this capability.

We need a common run-time infrastructure so that diverse components can work together coherently (e.g., a common transaction facility).

But as important as these technical requirements are, the application development organization must make it advantageous for component developers to make the extra effort required to produce reusable components, and for application developers to take the effort to find and use them. This requires a renumeration approach with scope beyond that of a particular programmer doing a particular task. Unfortunately, object technology has no good solution for this.

Maintainability

Generally, it is not possible to distinguish maintenance from development in a way that satisfies everyone. Is the modification of a tax algorithm maintenance or development? Is a new function implemented against an existing database maintenance or development? Many of our customers, using their own definitions, assert that they spend as much as 80 to 90 percent of their information technology dollars on what they call "maintenance." So it is clearly a very important aspect of computing.

No matter how it is defined, the essential aspect of maintenance is that some code must be changed (or added, or deleted) while other related code is unchanged and must continue to run correctly.

All of the essential aspects of object technology (encapsulation, inheritance, polymorphism, late binding) can contribute significantly in making maintenance more efficient and reliable. In fact, it can be argued that all of the aspects of object technology that support reuse also support maintainability.

Paradigm consistency

Because object technology is so pervasive across the life cycle of application development, it can be remarkably effective in allowing the structure of the application to be consistent throughout its development and maintenance phases. One of the great weaknesses of the "Information Engineering"[8] analysis and design methodology was the need to dramatically change the structure and components of an application when going from analysis to design to code. This had two disadvantages:

- The change in paradigm made it very difficult to go from phase to phase. And it made it difficult to ensure that the design really reflected the analysis, and that the code really reflected the design.
- Even when the team of analysts, designers, and coders was successful at each phase, the relationship between the phases did not last very long. When new requirements emerged they were often implemented directly, by changing the code. As a result the design no longer really represented the application.

Some CASE (computer-assisted software engineering) tool vendors attempted to address this problem by automatically generating code from design. Others required developers to first change analysis and design work products as needed, then proceed once more through the entire life cycle, disallowing changes attempted in any other way.

But where development is object oriented through all phases, it is much easier to do rapid prototyping, to maintain consistency across the life cycle, and even to reuse components. If a "customer" object, for in-

stance, is represented consistently at each phase, it can be reused. But if its run-time implementation is spread across the application's running code, it is very difficult to reuse it in another application.

Requirements for exploiting object technology

Exploiting these reuse, maintainability, and pervasive paradigm potentials requires:

- Tools that allow the creation, discovery, and customization of object-oriented components at all levels of abstraction (e.g., executable code, source code, design patterns, frameworks, and analysis and design constructs)
- Tools that support the composition of the components into applications, again at all levels of abstraction
- A run-time infrastructure of system, middleware, and class libraries that ensures that these components can, when combined, run seamlessly in a common environment

It is, therefore, toward the development and enhancement of such tools and infrastructure that object technology is primarily directed. This technology has moved well beyond its somewhat narrow beginnings. It reaches into systems areas, such as persistence, transactions, security, distributed directories, etc. It reaches into business analysis and modeling as well as programming.

It must be noted, in conclusion, that many risks and potential problems arise in the move to object technology. There are not enough experienced OO developers to meet the demand. And inexperienced developers often write OO programs that have performance problems and are difficult to test and debug. So the transition to object technology may well take longer than we would hope. But as we begin to have, through education, mentoring, and experience, enough OO developers to meet the demand for programs that execute efficiently and are easy to reuse and maintain, the wait will have been worth it.

Contents of this issue

This issue of the *IBM Systems Journal* includes papers that span the wide scope of object technology. Their authors are very much aware of the difficulties, as well as the potential advantages, that object technology may bring. They will likely be leaders in realizing this potential.

**Trademark or registered trademark of Sun Microsystems, Inc.

Cited references and notes

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley Publishing Co., Reading, MA (1995).
2. F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu, "Automatic Code Generation from Design Patterns," *IBM Systems Journal* **35**, No. 2, 151–171 (1996, this issue).
3. J. Gosling and H. McGilton, *The Java Language Environment: A White Paper;* view on the WWW via URL http://java.sun.com/whitePaper/java-whitepaper-1.html.
4. B. E. Carpenter, R. W. Doran, M. Woodger, and A. M. Turing, *A. M. Turing's Ace Report of 1946 and Other Papers,* The MIT Press, Cambridge, MA (1986).
5. O. Sims, *Business Objects: Delivering Cooperative Objects for Client/Server,* McGraw-Hill, Inc., New York (1994).
6. Quoting from David Taylor, "Hiding alternative procedures behind a common interface is called polymorphism." See Reference 9, below.
7. In this context, "scripts" are (generally) small programs, usually written in an interpreted language such as BASIC or REXX, used to customize the behavior of the black box component wrapped inside.
8. J. Martin and C. McClure, *Structured Techniques: The Basis for CASE,* Prentice-Hall, Inc., Englewood Cliffs, NJ (1988).
9. D. A. Taylor, *Object-Oriented Technology: A Manager's Guide,* Addison-Wesley Publishing Co., Reading, PA (1981).

General references

G. Booch, *Object-Oriented Analysis and Design with Applications, Second Edition,* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA (1994).

A. Goldberg and D. Robson, *Smalltalk-80: The Language,* Addison-Wesley Publishing Co., Reading, MA (1989).

B. Stroustrup, *The C++ Programming Language,* Addison-Wesley Publishing Co., Reading, MA (1986).

Accepted for publication February 5, 1996.

George Radin *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: radin@watson.ibm.com).*

Mr. Radin, an IBM Fellow, began programming in 1955. Among many accomplishments in his long career, he led the team that produced the PL/I programming language and the research team that produced the first reduced instruction set computer. He was manager of architecture for the IBM Future Systems project in the mid-1970s, Director of Architecture for the Systems Products Division in the early 1980s, Chief Architect for AD/Cycle® in the early 1990s, and is currently on the technical staff in Software Group. Mr. Radin was awarded the B.A. degree from Brooklyn College in 1951, and the M.A. degree in English literature in 1952 and the M.S. degree in mathematics in 1961, both from Columbia University.

Reprint Order No. G321-5597.