Adoption of Software Engineering Process Innovations: The Case of Object Orientation

Robert G. Fichman • Chris F. Kemerer

SOFTWARE DEVELOPMENT, SO CRITICAL TO THE EFFECTIVE USE OF INFORMATION TECHNOLOGY, IS POORLY UNDERSTOOD AND MANAGED. NUMEROUS SOFTWARE engineering process innovations have been proposed to improve software development, the latest of which is object orientation. How can information systems managers decide whether to invest in such technologies? This paper proposes a general two-dimensional framework based on theories about organizational and communitywide technology adoption. The authors test the framework by applying it to three previous innovations, and it accurately describes their adoption trajectories. Then they apply it to object orientation and take the controversial position that this new technology is not likely to be quickly adopted by large in-house business information systems groups.

Robert G. Fichman is a doctoral candidate and Chris F. Kemerer is the Douglas Drane Career Development Associate Professor of Information Technology and Management, both at the MIT Sloan School of Management.

Organizations increasingly rely on information technology (IT) both to perform their day-to-day operations and as a source of new products and services. The popular focus has been on hardware components, and that story has been overwhelmingly positive: the new technologies that come to market are cheaper, more reliable, and more portable than previous ones.

However, much less has been written about the software side of IT, and what has been written has not been positive. We commonly read that software is late, over budget, and of poor quality. Specific examples of software in the news are almost always negative — microcomputer software vendors that deliver releases late, federal government systems projects that never reach the implementation stage, and the recent telephone system outage that was blamed on a Texas supplier who "changed only three or four lines of the program."[1]

This imbalance has reached such proportions that it has been termed the software crisis. Software production represents the single biggest obstacle to the successful use of IT in organizations; all precepts such as "using IT for strategic advantage," "reengineering the business," and "informating the workplace" become mere slogans if the necessary software is not properly delivered on time.

Into this void have rushed a multitude of vendors and consultants offering solutions to the crisis. A quick review of trade press magazines will reveal articles on such topics as computer-aided software engineering (CASE), rapid applications development, cleanroom software engineering, software factories, and object-oriented (OO) approaches. All of these represent *software engineering process technologies*, that is, means by which an application software product is created (hereafter "software process technologies").[2] The idea that process technologies in general are critical sources of economic success is gaining acceptance.[3] However, the problem with this rich set of software process technologies is that they may be technically incompatible, or, if not, then at least sufficiently expensive and requiring significantly large changes in an organization's operation that they are unlikely to be successfully adopted at the same time.

How, then, can the chief information officer (CIO) of a large business application development organization choose which, if any, of these new software process technologies to adopt? Unfortunately, the choice of a new approach is rendered very difficult by the lack of unbiased information sources. All of the technologies are shrouded in new vocabularies and explainable only by a self-selected cadre of experts.

Moreover, many previous technologies have been so oversold that it is now sometimes difficult to tell legitimate enthusiasm from marketing hype. Fred Brooks has even coined a term for such oversold software process innovations — silver bullets.[4] One of Brooks's silver bullets is artificial intelligence (AI) expert systems. These were supposed to solve many problems, not just software development, and they merited at least two cover stories in *Business Week* over the past decade, perhaps the only software process technology to merit such general press attention. However, according to Brooks and others, the gains delivered by this technology have been much less than promised.[5] In contrast to the initial waves of enthusiasm, it now appears that relatively few new systems in business organizations are being developed using this technology and that their use remains plateaued at a small niche of applications, most of them limited to diagnostic tasks.

Will OO become the dominant software process technology of the future for in-house business application development?

This is not to say that AI expert systems are a poor technology, just that they did not meet the inflated expectations. The critical point is that AI expert systems have not become a dominant software process technology. An intuitive test of technology dominance is whether the majority of organizations developing applications software would choose this technology in developing or replacing their core applications. For example, third-generation software languages such as COBOL, FORTRAN, and C are dominant over their predecessor second-generation assembly languages in that second-generation languages are now relegated to highly specialized portions of applications and are not chosen as the primary software process technology for programming if any other alternative is available.

The reason that a CIO should care about sorting out which new software process technologies are likely to be dominant is that the costs of choosing a nondominant technology can be substantial. In particular, even technologies that offer some technical advantages over their predecessors may not end up dominating because they fail to achieve a critical mass. And once an organization commits to a technology that then fails to become dominant, it faces a number of additional problems, such as:
• Difficulties in hiring experienced staff;
• Limited enhancements to core technology available on the market;
• Few complementary products available;
• Third-party training opportunities less available;
• Lack of accumulated wisdom in the industry as to how best to improve the technology's performance; and
• Possible loss of vendor support.

Given the significant learning and adoption startup costs, and the increased risk of failure, these additional burdens may make the adoption of a nondominant technology extremely expensive in the long run.

The question remains: How can you choose tomorrow's dominant software process technology from today's list of candidates? Arguably, the leading candidate at the moment is the object-oriented approach. Although some of the key ideas of OO date back to late 1960s computer science research, it is in the past six years that OO has become a popular research topic and that commercial products have become widely available for its use. Recently OO was featured on the cover of *Business Week* as the key to "making software simple."[6] The central ideas of OO fit well with previous advances in software engineering, and proponents argue that the approach lends itself to the assembly of previously developed, well-tested components into software systems, thereby avoiding the labor-intensive and less reliable approach of building everything from scratch.

Will OO become the dominant software process technology of the future for in-house business application development? Our thesis is that for any software process technology to become dominant, it must first overcome a series of obstacles to adoptability. In particular, we evaluate adoption from two perspectives: diffusion of innovations and economics of technology standards. There is a rich academic and practical history of studying the diffusion of innovations, that is, the adoption of technology by individuals and their organizations, and of sorting out the characteristics of those innovations that have been successfully adopted. The economics of technology standards perspective examines technologies that have significant increasing returns to adoption. That is, the benefits of adoption largely depend on the size (past, present, and future) of the community of other adopters. Technologies have a greater likelihood of success to the degree that barriers to adoption are lowered.

In the sections that follow, we describe these two points of view and apply them to three past software process technologies (structured analysis and design methodologies, production fourth-generation languages, and relational database management systems). We construct a framework that evaluates the likelihood of dominance based on the technology's adoptability for both individual organizations and for whole communities or industries. And, finally, we introduce OO and analyze it in terms of the framework. We find that object-oriented technology faces considerable hurdles to both individual and industry adoption.

Perspectives on Technology Adoption

Why are some innovations adopted more rapidly than others? This simple question has been the subject of intense study by innovation diffusion researchers. Object orientation qualifies as an innovation — some say a radical one — in software engineering process technology; as a result, the vast body of literature on the diffusion of innovations (DOI) is a natural starting point in the search for clues about the technology's ultimate disposition. DOI researchers have typically studied the technology adoption decisions of individuals or organizations without taking into account community issues that strongly affect the innovation's inherent economic value.

Yet, community effects are likely to be crucial for software engineering process innovations because the benefits of adoption usually depend on the size of the current and future network of other adopters. (For example, widespread adoption of a software engineering process innovation increases the likelihood of the availability of complementary software tools.) Fortunately, a second line of research, in the area of the economics of technology standards, is focused on the role of community effects on technology adoption. Hence, for the purposes of the current analysis, we will use these two complementary perspectives.

Diffusion of Innovations

DOI research has been broadly defined as the study of how innovations spread through a population of potential adopters over time.[7] From the DOI perspective, diffusion is predominantly a process of communication; when and how a potential adopter learns about an innovation are important determinants of whether that individual will adopt. Other factors include the characteristics of the adopters and the means employed by vendors and change agents to persuade them to adopt.

To address the broader question of the likely rate of adoption of a specific innovation across an entire population, one must look to attributes of the innovation itself. Everett Rogers reviewed hundreds of diffusion studies and identified five generic innovation attributes that influence rates of adoption: (1) relative advantage, (2) compatibility, (3) complexity, (4) trialability, and (5) observability.[8] Although Rogers's synthesis is based mostly on studies of adoptions by individuals (e.g., of consumer goods), Van de Ven and others have argued that innovation attributes also play an important role in adoptions by organizations.[9] In fact, these researchers maintain that innovation attributes take on a broader role in the context of organizational adoption of complex technologies, affecting not only the initial decision to adopt, but also the ease of traversing later stages of adoption such as implementation, adaptation, and routinization. Hence, the analysis of innovation attributes provides a basis for assessing not only the likely rate of adoption across a population, but also the technology's comparative "adoptability" within individual firms (i.e., the overall ease of reaching a state of routinized use).

Table 1 briefly defines Rogers's five innovation attributes, tailored somewhat to better fit the context of organizational adoption of complex technologies.[10] With the exception of complexity, "high" values of the attribute are favorable to easier implementation within a given organization.

The explanations for the relative advantage, compatibility, and complexity attributes are straightforward enough: organizations are more likely to be willing and able to adopt innovations that offer clear advantages, that do not drastically interfere with existing practices, and that are easier to understand. Trialability and observability are both related to risk. Adopters look unfavorably on innovations that are difficult to put through a trial period or whose benefits are difficult to see or describe. These characteristics increase the uncertainty about the innovation's true value.

Economics of Technology Standards

The traditional DOI approach is a valuable but incomplete view of the dynamics of technologies, which, like software process technologies, are subject to the phe-

Table 1 Attributes of Innovations

Relative Advantage	The innovation is technically superior (in terms of cost, functionality, "image," etc.) than the technology it supersedes.
Compatibility	The innovation is compatible with existing values, skills, and work practices of potential adopters.
Complexity	The innovation is relatively difficult to understand and use.
Trialability	The innovation can be experimented with on a trial basis without undue effort and expense; it can be implemented incrementally and still provide a net positive benefit.
Observability	The results and benefits of the innovation's use can be easily observed and communicated to others.

nomenon of *increasing returns to adoption.*[11] Increasing returns to adoption means that the benefits of adopting an innovation largely depend on the size (past, present, and future) of the community of other adopters. Even though the DOI perspective recognizes the effects of community adoption on how potential adopters *perceive* a technology (e.g., by increasing social pressure to adopt), community adoption levels also affect the *inherent* value of any class of technology that has large increasing returns to adoption.

Economists have identified several sources of increasing returns to adoption, but the three most applicable to software process technologies are *learning by using, positive network externalities,* and *technological interrelatedness.* Learning by using means that a technology's price-performance ratio improves rapidly as a community of adopters (vendors and users) accumulates experience in developing and applying the technology. Positive network externalities (sometimes called network benefits) means that the immediate benefits of use are a direct function of the number of current adopters. (The classic example here is the telephone network, where the number of people available to call depends on the number of previous subscribers.) Technological interrelatedness means that a large base of compatible products — and hence a large base of likely adopters — is needed to make the technology worthwhile as a whole.

Several economists have developed analytical models to predict the circumstances under which technologies subject to increasing returns to adoption are likely to achieve critical mass and be widely adopted as a standard.[12] Among them, Farrell and Saloner have noted that a group of adopters, facing a decision to adopt a new and technically superior standard, may still fail to adopt in numbers necessary to achieve critical mass because of each adopter's reluctance to be the first to pay either of two early adoption penalties: *transient incompatibility costs* (because of delays in achieving a satisfactory network for the new technology) and *risk of stranding* (because of failure to ever achieve a critical mass of adoption).[13]

These two penalties can cause "excess inertia" to develop around an existing technology and prevent the adoption of an otherwise superior new technology.[14] A well-known example of this situation is the persistence of the inefficient QWERTY layout for typewriter keyboards (designed in the late 1800s to slow typists down sufficiently to avoid key jamming) despite the later invention of many superior keyboard layouts.

What determines whether a technology will achieve critical mass? Several factors have been identified by economists: (1) prior technology "drag," (2) investment irreversibility, (3) sponsorship, and (4) expectations (see Table 2).

The reasoning behind these factors is as follows. When a prior technology exists that has already developed a mature adoption network (i.e., it has a large installed base), the disparity in short-term benefits between the old and new technologies is likely to be large, even though the new technology may hold more promise in the long term. Hence the mere existence of a prior technology's installed base represents a "drag" on

Table 2 Economic Factors Affecting Technology Adoption

Prior Technology Drag	A prior technology provides significant network benefits because of a large and mature installed base.
Irreversibility of Investments	Adoption of the technology requires irreversible investments in areas such as products, training, and accumulated project experience.
Sponsorship	A single entity (person, organization, consortium) exists to define the technology, set standards, subsidize early adopters, and otherwise promote adoption of the new technology.
Expectations	The technology benefits from an extended period of widespread expectations that it will be pervasively adopted in the future.

the community's progress toward switching to the new technology because few adopters are willing to absorb the transition costs associated with joining a small, immature network. When adoption requires large, irreversible investments, this reluctance grows even stronger because a substantial risk premium must be added to adoption costs to take into account the chance of being stranded should a satisfactory network never develop for the new technology.

If prior technology drag and irreversible investments were the only relevant factors, dominant technologies might never be overturned. There are, however, two common ways that a new technology can overcome the head start of a prior technology: strong sponsorship and positive expectations. Sponsors can tip the cost-benefit equation in favor of the new technology by actively subsidizing early adopters, by making credible commitments to develop the technology regardless of the initial adoption rate, and by setting standards that ensure that a single network will emerge around the new technology instead of a pastiche of smaller, potentially incompatible networks (with correspondingly diffused network benefits).

Expectations about a technology's chances for dominance are also a crucial dimension in the technology standards view, for expectations largely drive critical mass dynamics. Early in the development cycle, promising new technologies typically enjoy a kind of "honeymoon" period during which some firms join an immature network assuming that widespread adoption will occur later. If not enough firms hold these positive expectations to begin with, or if the honeymoon period is cut short unexpectedly, then critical mass is unlikely to be achieved.

What determines the length and robustness of a new technology's honeymoon period? Positive characteristics include a strong scientific base and a clear match between the technology's unique strengths and apparent industry trends. For example, in the software industry, data-oriented design methodologies emerged at the same time many organizations were beginning to view data as a shared corporation resource. Such characteristics can lend an air of inevitability to a technology. On the negative side, widely publicized adoption "horror stories," substantial improvements to the existing dominant technology, or the rise of a new, even more promising substitute technology can cut the honeymoon period short.[15] Also, the expectation that the technology itself is soon to be significantly improved can induce inertia in potential adopters.

The technology standards approach to adoption of new technologies complements the DOI approach in three key ways. First, it defines a special class of innovations, those subject to increasing returns to adoption, to which software process technologies clearly belong. Second, it identifies several communitywide factors (e.g., prior technology drag) that are not included in the DOI view and that have an impact on the adoption of such technologies. Third, it predicts different patterns of adoption for technologies subject to increasing returns to adoption. According to the DOI perspective, innovation attributes may distinguish whether the cumulative adoption of an innovation is a steep or gently rolling S-curve, but in any case adoption should still follow an S-curve. The technology standards perspective, by contrast, sees adoption much more dichotomously: if a technology achieves critical mass within some reasonable period of time (during its honeymoon period), it will become dominant. Otherwise, the tide of expectations about the technology will turn among "swing voters," those who will consider adoption only if they expect the technology to dominate, and adoption will abruptly plateau or even turn negative as adopters discontinue use. Therefore, both perspectives add value to our discussion.

Examples of Software Engineering Innovations

The two perspectives described above have been useful in explaining adoption in many industries, but how accurately do they describe the speed and pattern of adoption for historical innovations in software engineering?

To answer this question, we examined three recent candidates for dominance in the realm of software engineering: (1) structured analysis and design methodologies (hereafter "structured methodologies"), (2) production fourth-generation languages (4GLs), and (3) relational database management systems (RDBs). Each of these innovations was intended to revolutionize a different segment of the software engineering discipline — analysis and design, coding and testing, and data management, respectively. And, like object orientation, each was preceded by tremendous publicity. Table 3 provides an overview of these technologies.

• **Structured Methodologies.** In the mid-1970s, structured methodologies (such as DeMarco structured analysis and Yourdon/Constantine structure design) emerged to replace the informal analysis and design practices of the day with an engineering style dedicated to rigorous procedures, formal diagrams, and measures of design quality. These methodologies went beyond the

Table 3 Software Engineering Innovations

	Structured Methodologies	Production 4GLs	RDBs
General Goals	Bring engineering-style rigor to the process of systems analysis and design.	Substantially reduce the quantity of program code in typical business applications and eliminate the applications backlog.	Make databases more easily accessible and more flexible to meet changing requirements.
Prior Technology	Informal, "homegrown" analysis and design methodologies.	3GLs, especially COBOL.	Hierarchical and network database models.
Key Characteristics	Decomposition of systems development into a sequence of well-defined, mandatory activities.	Nonprocedural constructs for screen management, menus, reports, and graphics generation.	Data represented as self-contained tables of values.
	Formal, standard techniques and diagrams for eliciting user requirements and representing candidate designs.	Many built-in language functions (mathematical and statistical calculations, searching for character strings, etc.).	Relationships between tables created by shared values instead of programmer-defined links.
	Metrics for judging system design quality.	High-level control clauses to facilitate structured programming (e.g., "select case").	Data manipulated via powerful set-level operators (select, join, project, etc.).
Notable Example	DeMarco Structured Analysis	Natural	Oracle
Early Predictions	A revolution in software engineering.	The long-awaited death knell for COBOL.	Domination of database market.
	Significant reduction in systems maintenance costs.	Ten-to-one improvements in programmer productivity.	Improved database design stability and flexibility due to increased data independence.
	Elimination of large-scale development fiascoes.	Applications development "without programmers."	Nontechnical users develop their own queries.
Adoption History	Slow acceptance as the sanctioned approach to systems development in most large companies.	3GLs still dominant for business applications.	Rapidly adopted as dominant technology for new development after some early "growing pains."
	Significant resistance to use in practice.	4GLs in danger of becoming "stranded" in face of newer technologies (CASE, code generators, object orientation).	Gradual conversion of existing installed base of applications.

mere adoption of a systems development life cycle and prescribed very specific schedules of tasks and activities.

• **Production 4GLs.** Although a wide variety of special-purpose 4GLs had previously existed, it was not until the early 1980s that a new breed of production 4GLs became commercially available. Production 4GLs, unlike end-user query-oriented 4GLs like FOCUS and RAMIS II, are those languages such as Natural, ADS/Online, and IDEAL that were designed to support development of large-scale business systems, a domain that was then dominated by COBOL.

• **RDBs.** Commercial RDBs emerged in the early 1980s as an alternative to the hierarchical and network model databases such as IMS and IDMS. Although RDBs and production 4GLs emerged concurrently, and some vendors bundled the two technologies together, most vendors took a mix-and-match approach wherein a production 4GL could be used with nonrelational databases, and a relational database could be accessed via 3GLs, especially COBOL.

Because the adoption context is important, the analysis below assumes a large, technologically capable information systems department of a business organization during the early adoption window for the respective innovations — a typical early adopter. For structured methodologies, the early adoption window occurred in the mid-1970s when they were proposed as a replacement for informal (or no) processes. For production 4GLs and RDBs, this window was the early 1980s, when they were slated to replace 3GLs and earlier database models, respectively.

The point of studying prior technological innovations is that their adoption trajectories can be used to test dominance theories. Of the three technologies, only RDBs have proven to be dominant in the sense that they replaced the prior technology for new development activities. It is unlikely that an organization would routinely proceed with developing a new application using an older generation (nonrelational) database model, whereas organizations continually choose the older, third-generation languages for new development. Similarly, structured methodologies have not reached the stage of widespread routinized use. Many organizations still cling to ad hoc methods within the life cycle approach or have abandoned the life cycle altogether for prototyping approaches that make no use of structured methods. The lack of adoption of structured methods has even been cited as a major barrier to the adoption of integrated CASE tools.

Analysis from DOI Perspective

According to the DOI view, an innovation's adoption rate and ease of implementation are largely determined by its degree of relative advantage, compatibility, complexity, trialability, and observability. Of the three software process innovations, structured methodologies stand out as having faced particularly difficult challenges on these terms. Although structured methodologies (like any legitimate candidate to dominance) offered relative advantages over informal methodologies, relatively high complexity and relatively low compatibility, trialability, and observability raised substantial barriers to rapid widespread adoption.

Structured methodologies were largely incompatible with the practices of most development groups because they overturned the "craft-oriented" values of experienced project managers of the 1970s, who had individually acquired an idiosyncratic array of guidelines during their apprenticeships. Structured methodologies had other compatibility problems as well. They required adopters to learn an extensive new skill set, and they changed many standard work practices associated with project organization and developer-client interactions (such as requiring that greater resources be devoted to a project's analysis and design phases). In addition, structured methodologies were far more complicated than the ad hoc methods they replaced. Entire books were devoted to presenting the details, including procedural guidance, diagram notations, and diagram construction rules.

Structured methodologies were also difficult to put through a trial period because of significant upfront training costs. But as the benefits of structured methodologies were mainly improved system maintainability and avoidance of development fiascoes, an extended trial period was needed to allow time for these benefits to unfold. Finally, structured methodologies suffered from low observability because it was difficult to describe concretely how benefits such as higher maintainability were to be achieved.

In summary, the DOI view would have suggested a slow and problematic adoption of structured methodologies, owing to relatively low compatibility, high complexity, low trialability, and low observability.

Production 4GLs and RDBs, on the other hand, would have rated much more favorably, suggesting comparatively rapid and smooth diffusion. By the 1980s, programming languages had evolved through three previous technology "generations," and it was reasonable to assume that these new languages, optimistically called the fourth generation, would enjoy similar success. Vendors and the business press were reporting productivity gains of ten-to-one; indeed, one industry guru *defined* 4GLs in this way. RDBs enjoyed the advantage of a respected theoretical foundation and near universal acclaim in the academic community as a simpler, more elegant data model that was also more robust to change. Therefore, the relative advantages of production 4GLs and RDBs were perceived as being particularly high.

Production 4GLs and RDBs did not have the compatibility problems that structured methodologies had. Adoption of production 4GLs and RDBs required a more straightforward substitution of new skills for old, with no major changes in work organization or reductions in autonomy for system developers. Although some idiosyncratic 3GL skills were lost with production 4GLs, the effect was less severe than with structured methodologies. In addition, inherently high complexity was not a problem. Production 4GLs were intended to simplify programming by letting the machine do more of the work, which is not to say that production 4GLs were simple to use in some absolute sense, but rather that they were simpler to use than alternative languages like COBOL. Likewise, RDBs were much simpler to use than first-generation database systems, with relatively intuitive select and join commands replacing arcane pointer references.

Although production 4GLs and RDBs required upfront investments in software and training comparable to structured methodologies, it was possible to demonstrate value on a single pilot project. As a result, the costs of experimenting with either technology could be kept at reasonable levels. Finally, production 4GLs were highly observable because adopters could draw concrete comparisons between applications written in a production 4GL and a third-generation language (e.g., number of work hours). RDBs, by contrast, suffered from lower observability because it was difficult to describe in concrete terms how they would achieve greater flexibility and data independence.

In summary, 4GLs and RDBs faced many fewer obstacles to adoption than did structured methodologies, and therefore the DOI view would have suggested more rapid and smooth adoption of these two technologies.

However, only RDBs have become dominant. Although the DOI view accounts for the fate of structured methodologies, it does not effectively discriminate between 4GLs and RDBs and, therefore, would have been an inadequate single explanation of dominance.

Analysis from Economics of Technology Standards Perspective

The three technologies under discussion are all subject to increasing returns to adoption; they become much more valuable to individual adopters to the extent that others adopt. Widespread adoption leads to faster maturation (learning by using), wider availability of qualified personnel (positive network externalities), and a larger array of complementary products and services (technological interrelatedness). Hence, the technology standards view is appropriate for analyzing the communitywide adoptability of these three technologies. As described earlier, four factors largely determine whether a technology will achieve critical mass and become a dominant technology: prior technology drag, irreversibility of investments, sponsorship, and expectations.

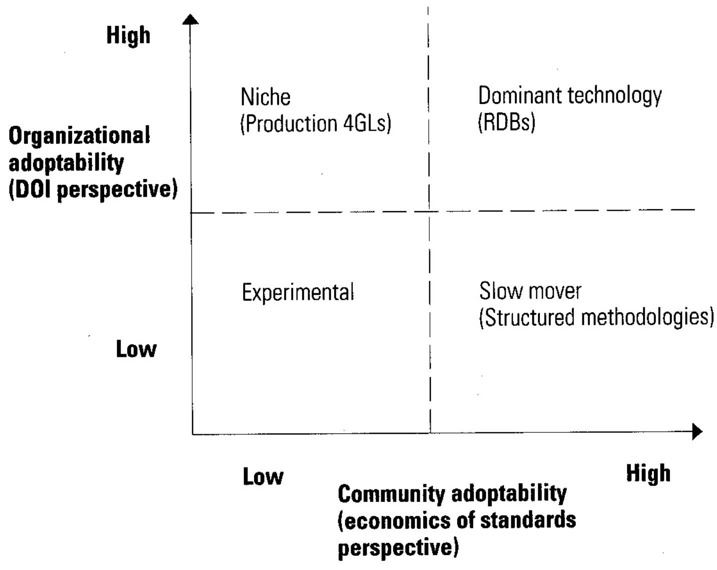
Production 4GLs stand out among the three technologies as having faced especially severe obstacles from the economics of technology standards view. Production 4GLs faced extensive prior technology drag, required large irreversible investments, were poorly sponsored, and had diminished expectations (due to publicity surrounding adoption fiascoes and the lack of a strong scientific base).

When introduced in the early 1980s, production 4GLs were a textbook case of an innovation facing a monolithic installed base, namely, the COBOL programming language. Forgoing the COBOL standard meant missing out on a large network of experienced personnel and compatible tools (e.g., database systems). Hence, prior technology drag was especially high.

Adoption of production 4GLs required largely irreversible investments in staff training and software — specialized assets that lose most or all of their value should the investment project be abandoned later. Naturally, organizations are reluctant to incur the risk associated with such investments, especially when the investment pertains to technologies that, like most software engineering innovations, require a critical mass of other adopters to achieve full benefits over the long term.

Lack of sponsorship also created a hurdle for 4GLs.

Figure 1 Software Engineering Process Technologies Adoption Grid


High
Organizational adoptability (DOI perspective)
Low
Niche (Production 4GLs)
Dominant technology (RDBs)
Experimental
Slow mover (Structured methodologies)
Low
Community adoptability (economics of standards perspective)
High

Niche	Adoption will start out fast among adopters who are relatively insensitive to standards issues or who have optimistic expectations about future levels of adoption. But adoption will plateau at a position short of dominance because of a failure to achieve critical mass.
Dominant Technology	The technology will be rapidly adopted as a dominant process technology. It will face relatively low barriers to individual or community adoption.
Slow Mover	The technology will diffuse steadily but slowly because of the difficulty of individual organization adoption.
Experimental	The technology will need to evolve before it is widely adopted by mainstream organizations as a dominant technology.

A dominant sponsor can advance the adoption of a technology by promoting a unified standard, by subsidizing early adopters, or by making a credible commitment to develop the technology even in the face of expected delays in widespread adoption. In the case of production 4GLs, many languages emerged — none supported by a dominant sponsor. Production 4GLs even lacked a single authority figure to define exactly what capabilities a production 4GL should provide.

Finally, expectations worked against community adoption. Production 4GLs were developed primarily by innovators in the commercial sector and suffered from a lack of scientific support or more objective boosterism from the academic community. Perhaps more damaging, 4GLs experienced some widely publicized adoption fiascoes, such as the infamous New Jersey Department of Motor Vehicles case.[16]

In summary, the economics of technology standards view offers an explanation for the failure of 4GLs to achieve critical mass because of high prior technology drag, large irreversible investments, relatively poor sponsorship, and rapidly diminished expectations.

Structured methodologies and RDBs, on the other hand, would have rated much more favorably from the economics of technology standards view. On the first and most important dimension — prior technology drag — structured methodologies were an attempt to impose order on chaos and thus faced essentially no installed base. RDBs faced an installed base of first-generation database systems, although not one so mature and ubiquitous as COBOL. In addition, the first-generation database market was fragmented, meaning that the benefits associated with joining the network of any given database product (e.g., IMS or IDMS) were correspondingly reduced.

As with production 4GLs, RDBs and structured methodologies required largely irreversible investments in staff training. Additionally, adoption of RDBs required the purchase of expensive software. This suggests that the irreversible investments dimension is not a significant discriminator among any of the three technologies. On the sponsorship dimension, RDBs had a founding father in E.F. Codd who clearly established the criteria for database management systems (DBMS) to qualify as fully relational. Structured methodologies also had easily identifiable founding fathers (e.g., Ed Yourdon and Tom DeMarco, among others). Although none of the three technologies was strongly sponsored in the traditional sense of a single organization encouraging adoption, RDBs and structured methodologies had widely recognized leaders defining the technology and proselytizing for widespread adoption, and they therefore rank relatively higher than production 4GLs on this dimension.

With an unassailable scientific base and near universal support in the academic community, RDBs benefited from high expectations. In addition, the strengths of RDBs — data independence and greatly simplified information retrieval — complemented industry trends toward more data-intensive applications. Structured methodologies were certainly in line with the trend in the 1970s toward large-scale development projects, and they also escaped widely publicized disasters. Therefore, production 4GLs rated the poorest on this dimension.

To summarize, compared with production 4GLs, RDBs faced a much less well-established installed base and had the advantage of positive expectations and a strong sponsor to push forward a cohesive definition of the technology. Structured methodologies had effectively no installed base to overcome, required less in the way of irreversible investments, and were somewhat better sponsored. Hence, other things being equal, the economics of technology standards view would have implied a relatively easy community adoption of structured methodologies and RDBs.

Again, however, only RDBs have actually become dominant. Therefore, the economics of technology standards view, although accurately reflecting the fate of production 4GLs, does not effectively discriminate between structured methodologies and RDBs and therefore would have been an inadequate single explanation of the ultimate dispositions of these technologies.

Framework for Assessing Software Engineering Process Technologies

To summarize, the DOI perspective rates production 4GLs highly but fails to take into account the effects of increasing returns to adoption present in software process technologies. The economics of technology standards perspective rates structured methodologies highly but fails to consider the delays caused by low organizational adoptability. Of the three, only RDBs have become a dominant technology, and only RDBs rate highly on both criteria. This suggests that it may be necessary to consider both perspectives in order to accurately forecast the likelihood of the innovation dominance.

Figure 1 illustrates a unified framework for assessing the likelihood of software engineering technologies becoming dominant. The vertical axis reflects the DOI view of organizational adoptability. The horizontal axis reflects the economics of technology standards perspective of community adoptability. The space defined by the two continuums can be simplified by dividing each in half and creating four quadrants, each of which implies a distinctive adoption trajectory.

This framework effectively explains the adoption patterns of these three past software engineering process innovations. Production 4GLs diffused rapidly to a number of organizations but never displaced 3GLs as the dominant choice for new development, except possibly in some niche applications. 3GLs are still the language of choice for code generators, which may imply that a plateau has been reached for production 4GL adoption. Structured methodologies, almost twenty years after their introduction, are still being adopted very slowly, as evidenced by the difficulty in introducing CASE tools that are tied to these methodologies. RDBs became a dominant technology during the 1980s and now represent the vast majority of new implementations of large-scale business systems.

What Is Object Orientation?

Object-oriented technologies are being touted as tomorrow's software process technology. In this section, we give a very brief introduction to OO as it is portrayed by its advocates. The section is designed as a primer to these ideas for the CIO or equivalent.

A daunting lexicon has developed around object-oriented technology, partly because the object-oriented approach touches all aspects of software engineering and partly because many of the concepts associated with OO have no direct analogs in the world of conventional systems development (see Table 4 for a list of key terms). Yet the essence of the object-oriented approach can be captured by two principles for structuring systems: storing data and related operations together within objects (encapsulation) and sharing commonalities between classes of objects (inheritance). Advocates of OO claim that adherence to these ideas advances such long-standing software engineering goals as abstraction, modularity, and reuse.

Designers achieve *encapsulation* by placing a set of data and all the valid operations on that data together inside a metaphorical "capsule" called an object. The logic behind encapsulation is simple: as there is usually a limited number of sensible operations for any given data structure, why not restrict processing to only these operations and put them together with the data in one place? Although this seems reasonable, it is not how traditional systems are organized. In a traditional system, the various operations associated with any given set of data (edits, calculations, transformations, update logic, and so forth) are typically repeated, more or less consistently, in many otherwise independent application programs. To take a simple example, variants on an "update customer balance" operation might appear in a billing program, a cash processing program, and a credit pro-

Table 4 Object-Oriented Terminology

Class	An abstract definition or template for a collection of objects that share identical structure and behavior.
Encapsulation	Enclosing a data structure and all operations that access the structure inside a "capsule" or object, which prevents direct access to the data by means other than those operations.
Information Hiding	A design principle that states that the internal structure of a given module should be a black box that stays hidden from other modules. Information hiding insulates other modules from changes that affect a given module's internal representation but not its external interface.
Inheritance	A mechanism that allows objects from different but related classes to share common characteristics (variable and method definitions) by placing those common characteristics in higher-level classes and creating links to those classes.
Message Passing	The practice of using explicit messages sent from one object to another as the only form of object communication. A message consists of an object identifier, a method name, and a set of arguments.
Method	A small program associated with an object that performs an atomic and cohesive operation, usually on that object's data.
Object	An abstraction of a real-world object that encapsulates a set of variables and methods corresponding to the real-world object's attributes and behaviors.
Polymorphism	A programming mechanism that allows the same message protocol to be sent to objects from distinct yet similar classes to invoke a common action. For example, polymorphism allows the same PRINT message protocol to be sent to a document or spreadsheet rather than having separate message protocols for each.
Variable	An item of data associated with an object.

Table 5 Software Engineering Goals

Abstraction	Abstraction is a "simplified description or specification of a system that emphasizes some of the systems details or properties while suppressing others."* The evolution of software engineering can be seen as a stream of improved software abstraction mechanisms, beginning in the 1950s with the invention of symbolic assemblers and continuing over the next three decades with the invention of callable procedures, stepwise refinement, structured programming, and, especially, successive generations of higher-level programming languages.
Modularity	Modularity is the principle of decomposing program logic into a collection of well-defined, self-contained units (modules) rather than creating a single, monolithic program. Ideally, modules should be loosely coupled and highly factored. Loosely coupled modules have a minimum of interdependencies, which means that an individual module can be modified or extended without causing a cascade of related changes. Factoring, the practice of putting one thing in one place, seeks to eliminate duplicate code across similar but not quite identical modules. By eliminating redundant coding efforts, factoring reduces the size of development and maintenance projects and avoids the creeping inconsistencies that may appear over time in modules that share redundant code.
Reuse	Software reuse means applying a piece of program code for some purpose beyond what it was originally intended for. Subroutine libraries, program skeletons, and cutting-and-pasting code from one program to another are all forms of reuse. The reuse of software eliminates the need to redundantly develop software, thereby improving productivity, and allows the use of tested and proven components, thereby improving quality.

* M. Shaw, "Abstraction Techniques in Modern Programming Languages," *IEEE Software,* October 1984, pp. 10-26.

cessing program. Traditional systems are designed this way because they were developed prior to the widespread advocacy of encapsulation and because conventional languages make encapsulation, as a primary structuring principle, awkward to achieve.

Inheritance in OO is something like inheritance in the real world. To take a particularly literal example, a sporting dog inherits attributes (physical characteristics) and behaviors (pointing and retrieving) from its parents. Likewise, within an object-oriented system, classes of software objects inherit common variables and methods from ancestor classes. The net result of inheritance is that developers can avoid coding redundancies by placing each operation at the appropriate level in a hierarchy of system modules (object classes). For example, suppose that employees are classified as hourly or salaried. With inheritance, a designer places common data and operations in a parent "employee" class and unique characteristics (e.g., regarding compensation) in two subclasses, "hourly employee" and "salaried employee." This avoids the need to either (a) create a single set of data and operations with extra logic to handle differences between kinds of employees or (b) create two separate and largely redundant sets of data and operations (one for each employee type). As with encapsulation, inheritance is rarely used in traditional systems development because conventional programming languages make it difficult to be implemented as a primary structuring principle.

Advocates of OO have argued that encapsulation and inheritance can advance the software engineering goals of abstraction, modularity, and reuse (see Table 5). These qualities in turn can drastically reduce system development and maintenance cost while improving system flexibility and quality. Not surprisingly, computer scientists have been seeking better ways to promote these three qualities for decades.

Abstraction is a tool for managing complexity. Designers promote abstraction of computer systems (or any other technical artifact) by emphasizing details that are important to the user while suppressing details that are unnecessary for the task at hand. For example, consider the different level of detail required to explain a combustion engine to a driver, mechanic, or physicist. Encapsulation promotes abstraction by making it easier to define objects in terms of *what they do* instead of *how they do it.* OO advocates believe inheritance is potentially even more powerful for promoting abstraction. One of the best ways to create useful abstractions is to describe some idea first in its most general form, followed by progressively specific versions (e.g., mammal, canine, sporting dog, retriever). This is precisely what an inheritance hierarchy does — it defines a hierarchical taxonomy of object classes from most general (at the top of the hierarchy) to most specific (at the bottom).

Modularity is a powerful tool for managing large problem domains. In fact, the structured programming movement in the late 1960s and early 1970s was largely an attempt to develop methods to improve the extent and quality of modularity in delivered systems. The net result of high-quality modularity is to localize system functionality so that changes can be made to one part of a system without a cascade of related changes. This cascade of changes (the so-called ripple effect) can be devastating; in many systems, hundreds or thousands of programs may be interdependent because of program code redundancies or, more subtly, the sharing of common data. (A seemingly minor change to the meaning or structure of data can ripple across all programs that share the data.) Encapsulation promotes a looser coupling of modules by disallowing sharing of data and limiting the range of module interdependencies to their external interfaces. Inheritance, when properly used, provides an elegant tool for extracting logic common to a group of classes at one level and placing it in a single, higher-level "superclass."

Reuse is the well-known principle of avoiding extra work by not reinventing the wheel. Although software reuse is perhaps the most persistently claimed advantage of object orientation, it has existed in one form or another since the early days of programming. But although conventional approaches allow for reuse of code, object orientation provides mechanisms that facilitate and enforce reuse. Encapsulation promotes information hiding, which reduces the burden of describing and finding reusable components — a significant obstacle to reuse in a large-scale environment. Inheritance plays an even more important role. Each time a new class is defined in an inheritance structure, this implicitly enforces reuse of operations from the parents of the new class. Finally, encapsulation and inheritance can both support reuse indirectly by improving system modularity. When system components are highly modular, they are much easier to reassemble in new combinations to support a new context.

In summary, object orientation is a software engineering process innovation whose essential principles — encapsulation and inheritance — have been claimed to lead to systems that are more abstract, more modular, and more reusable.

Object Orientation: Where Does It Fit?

Where does object orientation fit on the new framework? Is it more likely to become a dominant technology or to end up in the slow mover, niche, or experimental categories? To answer this question, we must analyze object orientation along the dimensions of organizational adoptability and community adoptability.

Organizational Adoptability

Like production 4GLs and RDBs, object orientation rates highly on perceived relative advantage. The practitioner literature has taken an almost uniformly positive tone toward OO, and a strong body of theoretical evidence supports the approach. Proponents have argued that object orientation promotes abstraction, modularity, and reuse, all of which are long-standing objectives of the software engineering field. It is fair to assume, then, that many potential adopters will develop favorable opinions of the advantages of object orientation based on positive reviews in the literature and their own assessments of the technology's potential merits.

As in the case of structured methodologies, however, compatibility appears to be a significant pitfall. Object orientation is a new development model and requires new skills in analysis, design, and programming that replace, rather than build on, those associated with conventional development.[17] In addition, several authors have noted that successful reuse requires marked changes in culture and values.[18] It has been observed that to maximize reuse, developers must learn to assemble applications using objects developed by others. This means that developers must learn to trust classes they did not develop and to overcome the "not invented here" bias. Furthermore, OO adoption as an organization's standard approach will likely require a restructuring of development teams. Some proponents have suggested the need to institutionalize reuse through creation of a new function similar to data administration that administers and controls the common repository of reusable components.[19] Others have gone so far as to suggest that object orientation requires the creation of new categories of developers — class "producers" who develop the foundation classes in the repository and class "consumers" who assemble existing components into new applications.[20]

Object orientation, as a process technology, rates unfavorably on the complexity dimension.

Object orientation, as a process technology, also rates unfavorably on the complexity dimension, even relative to other complex software engineering technologies like structured methodologies. To be successful with OO as a software engineering process technology, an adopter must absorb a new lexicon, new development methodologies, and new development tools. The need to master such concepts as encapsulation, inheritance, information hiding, polymorphism, and many others beyond those presented in Table 4 represents a significant barrier to achieving an overall understanding of object orientation and its proper application.

Object orientation, as a process technology, is also difficult to put through a trial period. As with the structured methodologies, production RDBs, and 4GLs, OO will require substantial upfront expenditures on software and training in order to conduct meaningful pilot projects. Many of the ultimate benefits of object orientation come from the repository of reusable components, which may take many years to create.[21] In the meantime, adopters are likely to experience an initial productivity decline because of the extra initial effort to design modules for reuse.

As a software engineering process innovation, object orientation is largely a "black box" technology — unless one mistakenly equates object orientation, as many do, with iconic user interfaces. The benefits of object orientation resist direct observation, and few organizations collect and track the software metrics required to demonstrate increased reuse, improved maintainability, or incremental improvements in productivity. As a result, object orientation suffers from low observability.

In summary, object orientation rates comparably with structured methodologies, the least favorable of the three previously reviewed technologies, on ease of adoption, with high relative advantage, but equal or lower ratings on compatibility, complexity, trialability, and observability.

Community Adoptability

For object orientation, the rival entrenched technology is not just a language generation or a database model, but the entire procedural paradigm for software development. It is therefore difficult to imagine a more compelling instance of prior technology drag in software engineering. New approaches have been proposed in every segment of software engineering: analysis (OOA), design (OOD), programming (OOP), and databases (OODBMS). Full object orientation also requires more extensive irreversible investments than production 4GLs or RDBs because of a substantially larger training burden and a wider array of potential software purchases (i.e., CASE tools to support analysis and design, new programming languages, and a new DBMS).

Object orientation does have a sponsor, the Object Management Group (OMG). OMG, a consortium of over two hundred technology vendors and users, performs all of the roles of a traditional technology sponsor: coordination of standards, subsidization of early adopters (e.g., through technology-sharing agreements), and general promotion of the technology (e.g., through seminars and technology fairs). However, an industry consortium is a relatively weak type of sponsor as it is subject to defections and even disbandment if the vendor participants decide it is in their interest to do so. In addition, standards generated by committee generally take longer to form than de facto or other unilateral standards and may therefore be less successful. For both these reasons, a consortium can be a weaker sponsor than a single vendor or other source.

Object orientation possesses many of the characteristics that lead to an extended period of favorable expectations. Object technology has a strong scientific base and enjoys widespread support in the academic community

Although expectations for OO are currently positive, they were similarly positive for 4GLs when first introduced.

as a "better way" to develop software. It provides powerful mechanisms for software modularity and abstraction; these can be seen as central to most software engineering advances to date. In addition, object technology is aligned with many current technology trends. It appears well suited to event-driven graphical user interfaces, multimedia systems (voice, imaging, animation), and highly parallel processing (e.g., for full text searching and retrieval). These kinds of applications tend to require complex multilevel data structures and data encapsulation, both of which are difficult to implement within the procedural paradigm, but they play to the particular strengths of object orientation. However, production 4GLs also had high positive expectations, reflected in their very name, which implied an inevitability to their replacing 3GLs. In addition, OO faces a more sophisticated and skeptical adoption community as the passing years have provided a larger array of proposed software process technologies that failed to live up to early expectations.

In summary, object orientation rates about the same as production 4GLs, the lowest of the previously reviewed software engineering innovations, from the economics of technology standards view. Object orientation faces an even more thoroughly entrenched standard (the procedural paradigm) and requires a more extensive investment in irreversible assets than production 4GLs.

Although expectations for OO are currently positive, they were similarly positive for 4GLs when first introduced. While object orientation does appear to be better sponsored than production 4GLs, this sponsorship is in the form of a potentially fragile consortium.

The combined ratings on ease of both individual and community adoption place object orientation in the experimental quadrant of the framework. This suggests that further development of the technology will be needed before widespread adoption will occur.

Conclusions

The poor state of software development practice has engendered a large number of proposed process technology improvements. Managers seeking advice on new software process technologies should learn from the lessons of technological change in other domains while recognizing the relatively unique features of software process technologies. The two-dimensional framework combines the work on diffusion of innovations and economics of technology standards to create a method for evaluating the potential of technology dominance. As we have shown, the framework offers an explanation for the adoption trajectory of previous technologies in analysis and design, coding and testing, and data management. Object orientation, the latest widely touted software process technology, rates unfavorably on both scales relative to these prior technologies and thus is unlikely to become the dominant software process technology for large in-house business application developers without significant changes.

What are the risks for the CIO of ignoring the framework when evaluating software process technologies? The primary risk associated with early adoption of niche technologies is being stranded on a technological "spur" away from the main track of technology development; the primary risk associated with early adoption of slow movers is implementation failure or the need to weather an extended period of transient incompatibility costs while waiting for a robust network to emerge. For some organizations, under some circumstances, these risks may be acceptable. Companies facing a wave of crucial new development projects that cannot feasibly be done with current technologies might consider a particularly well-suited niche technology — with the understanding that an expensive redevelopment or conversion may become necessary sooner (e.g., within five to ten years) rather than later (e.g., within ten to twenty years). Companies that are facing a major system replacement decision, that have a successful track record in adopting "bleeding edge" technologies, *and* that have the resources to support a robust internal training program might consider adopting a slow mover.

The risks of full-scale organizational adoption of current OO technology include both of the above, given its predicted trajectory into the experimental cell of the matrix. It will prove difficult to make the technology a routine part of software development. And, even if the organization successfully adopts the technology, the crucial benefits that accrue from a large network of users may never develop, given the barriers to industry adoption. The organization may end up locked into a stranded technology, finding it difficult to hire experienced staff, to purchase complementary tools, and in general to achieve the benefits that adopters of dominant process technologies enjoy.

Having said this, in the immediate future, there will no doubt be a series of OO success stories — some genuine and others overstated for dramatic effect, as vendors and early adopters try to encourage a bandwagon. In assessing these stories, however, managers must consider several key questions. First, to what degree is the system truly "object oriented"? As with expert systems, a new technology often accrues a certain status so that vendors quickly give the new label to all their technologies within reach, whether or not they actually deserve it. Second, who did the development? Results achieved by handpicked internal stars or a team of industry consultants cannot be generalized to an entire IS development organization. Third, what kind of system was developed? Here again, a stand-alone application in a specialized domain might not be representative of the broader range of large business-oriented systems that comprise the core applications of most IS organizations. And finally, has the adopting company truly made the transition to routinized use, where OO has become the default technology for new applications? All too many software innovations end up as "shelfware" after promising pilot projects, because of the difficulty in replicating the success of one team across the entire organization.

But isn't it desirable to be different from your com-

First movers will be in the ironic position of having to hope they are quickly followed, so that critical mass will be reached and the technology will become dominant.

petitors? Wouldn't being a first mover in adopting OO provide a competitive advantage as many proponents suggest? This is unlikely. Publicly available technologies rarely provide a *sustainable* competitive advantage in and of themselves: they require mating with some other relatively unique organizational competence, otherwise all competitors would also adopt the technology and quickly close the gap. In fact, in the case of OO, first movers will be in the ironic position of having to hope they are quickly followed, so that critical mass will be reached and the technology will become dominant. Meanwhile — even assuming dominance eventually is achieved — the benefits of being a first mover (e.g., riding the internal learning curve sooner, building general innovative capabilities, and attracting leading-edge personnel) could easily be outweighed by disadvantages (e.g., joining an immature network with high transient incompatibility costs, adopting an early and less favorable technology vintage, and experiencing a loss of trained staff to other companies). As a result, instances of first-mover advantages for corporate IS departments are likely to be rare. (For software vendors, however, an alternative scenario exists where heavy initial investment in OO could trigger a virtuous cycle of increased market share and resulting increased economics of scale.)

The framework we have described outlines the likely diffusion pattern at the time the ratings are performed. The most appropriate time to perform the assessment is during the technology's first widespread commercial availability. It is possible that, over a period of years, ratings on individual dimensions could evolve enough to move a technology's trajectory to a different cell. Having said that, such a move is probably unusual given the rapid arrival of new technologies and the inevitable loss of uncritical media attention typically bestowed on an emerging technology. Technologies can be new and exciting only once; premature deployment and excessive claims about benefits can be very damaging.[22] However, some speculation on the possibility of OO's trajectory changing is clearly appropriate, both for proponents who wish to increase the adoption rate of their technology and for potential adopters who wish to track the progress of this technology for signs of increasing likelihood of its widescale adoption.

On the positive side, better than expected growth in the demand for applications that fit well with the OO approach is likely to increase OO's relative advantage. Thus a growth in multimedia applications and the demand for applications running in client-server environments would clearly help. New tools that reduce the technology's complexity or make OO more compatible with existing techniques would also be positive steps. Adoption of OO techniques in university programs would enable the next generation of software engineers to avoid the compatibility trap. Favorable trade press coverage about OO success stories and the absence of widely publicized OO disasters would further improve the technology's observability and would improve the general level of expectations. The ability of the Object Management Group to maintain the coalition and generate some initial successes on standards would solidify its position as external sponsor of the technology.

On the negative side, slow progress on the above-mentioned fronts will be warning signs that OO's potential for dominance is limited. Additionally, widescale adoption of OO is threatened by growth from competing technologies, such as CASE (specifically CASE templates), which offers many of the same advantages of productivity, quality, and reuse. Rapid progress in this or other areas may sharply reduce OO's honeymoon period. Finally, the development of a newer, yet-to-be-announced technology could supplant the current interest in OO, riding the same wave of enthusiasm that currently benefits OO.

Our conclusion that object orientation has low prospects for becoming dominant in large in-house IS organizations is in sharp contrast to the enthusiastic touting we hear in the trade press.[23] We see many obstacles in object orientation's way. Some of these can be worked around with an appropriate adoption strategy, while others depend on communitywide actions beyond any individual adopting organization's control. For vendors and other OO proponents, this framework offers a clear agenda for the future by outlining several avenues where an aggressive, coordinated effort is needed to lower obstacles to individual and communitywide adoptability. ◆

References

We received helpful comments on earlier versions of this paper from S. Brobst, E. Brynjolfsson, J. Quillard, W. Orlikowski, J. Rockart, W. Stevens, L. Votta, and participants at the UCLA Information Systems Colloquium. We gratefully acknowledge the funding provided for this work by Credit Suisse and the MIT Center for Information Systems Research.

1. M.L. Carnevale, "DSC Says Software Change Led to Phone Outages," *Wall Street Journal*, 10 July 1991, p. 5.

2. The innovation literature distinguishes *process* technology innovations from *product* innovations. A process innovation is one that changes the production process, that is, the way a product is produced. Most industrial tools, when first introduced, are simultaneously process and product innovations — a product innovation for the tool producer and a process innovation for the tool consumer. For example, for a DBMS vendor, a new RDB offering is a product innova-

tion; for a systems developer, RDB technology is a process innovation that, among other things, involves the purchase of a product. As this article is targeted at consumers rather than producers of software technologies, we define them as process innovations. For further reading on the topic of process and product innovations, see:
W.J. Abernathy and J.M. Utterback, "Patterns of Industrial Innovation," *Technology Review*, June-July 1978, pp. 40-47.
3. L. Thurow, "Who Owns the Twenty-First Century?" *Sloan Management Review*, Spring 1992, pp. 5-17.
4. F.P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer* 20 (1987): 10-19.
5. W.M. Bulkeley, "Bright Outlook for Artificial Intelligence Yields to Slow Growth and Big Cutbacks," *Wall Street Journal*, 5 July 1990, pp. B1, B3.
6. "Software Made Simple," *Business Week*, 30 September 1991, pp. 92-100.
7. E.M. Rogers, *Diffusion of Innovations* (New York: Free Press, 1983).
8. Ibid., ch. 6.
9. See, for example:
J.E. Eveland and L.G. Tornatzky, "The Deployment of Technology," in *The Processes of Technological Innovation*, eds. L.G. Tornatzky and M. Fleischer (Lexington, Massachusetts: Lexington Books, 1990), pp. 117-148;
T.H. Kwon and R.W. Zmud, "Unifying the Fragmented Models of Information Systems Implementation" in *Critical Issues in Information Systems Research*, eds. J.R. Boland and R. Hirshheim (New York: John Wiley & Sons, 1987);
D. Leonard-Barton, "Implementation Characteristics of Organizational Innovations," *Communication Research* 15 (1988): 603-631;
G.C. Moore, "End-User Computing and Office Automation: A Diffusion of Innovations Perspective," *Infor* 25 (1987): 214-235;
J.M. Pennings, "Technological Innovations in Manufacturing," in *New Technology as Organizational Change*, eds. J.M. Pennings and A. Buitendam (Cambridge, Massachusetts: Ballinger, 1987), pp. 197-216; and
A.H. Van de Ven, "Managing the Process of Organizational Innovation" in *Changing and Redesigning Organizations*, ed. G.P. Huber (New York: Oxford University Press, 1991).
10. We use Rogers's five innovation attributes mainly because they are familiar in the DOI field. Van de Ven, Moore, and Kwon and Zmud also use Rogers's definitions, although others have provided alternative taxonomies of the salient attributes of complex organizational technologies. Leonard-Barton identifies transferability, organizational complexity, and divisibility. Pennings identifies concreteness, divisibility, and cost. Eveland and Tornatzky identify trialability, lumpiness, adaptability, degree of packaging, and the "hardness" of the underlying science. In most cases, these attributes can be mapped to one or more of Rogers's original five attributes, at least as they are used here.
11. W.B. Arthur, "Competing Technologies: An Overview," in *Technical Change and Economic Theory*, ed. G. Dosi (New York: Columbia University Press, 1987).
12. See:
Arthur (1987);
J. Farrell and G. Saloner, "Competition, Compatibility, and Standards: The Economics of Horses, Penguins, and Lemmings," in *Product Standardization and Competitive Strategy*, ed. H.L. Gabel (Amsterdam: North-Holland, Elsevier Science, 1987);
M.L. Katz and C. Shapiro, "Technology Adoption in the Presence of Network Externalities," *Journal of Political Economy* 94 (1986): 822-841.
13. Farrell and Saloner (1987).
14. Ibid.
15. N. Rosenberg, "On Technological Expectations," in *Inside the Black Box: Technology and Economics* (New York: Cambridge University Press, 1982).
16. G. Rifkin and M. Betts, "Strategic Systems Plans Gone Awry," *Computerworld*, 14 March 1988, pp. 1, 104-105.
17. R. Fichman and C. Kemerer, "Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique," *IEEE Computer* 25 (1992): 20-39.
18. See S. Atre, "The Scoop on OOPS," *Computerworld*, 17 September 1990, pp. 1115-1116;
J. Moad, "Cultural Barriers Slow Reusability," *Datamation*, November 1989, pp. 87-92; and
M. Stewart, "Object Projects: What Can Go Wrong," *Hotline on Object-Oriented Technology* 2 (1991): 15-17.
19. Moad (1989).
20. Stewart (1991).
21. Atre (1990).
22. Rogers (1983) notes that unfulfilled expectations about an innovation's benefits are a primary cause of subsequent discontinuance. Leonard-Barton has argued that discontinuers can become influential "negative" opinion leaders, and that entrenched opinions about an early technology generation are hard to overturn, even when later and more viable technology generations become available. See:
D. Leonard-Barton, "Experts as Negative Opinion Leaders in the Diffusion of a Technological Innovation," *Journal of Consumer Research*, 11 March 1985, pp. 914-926.
23. As mentioned previously, adoption context is important to the ratings. In some segments, such as CAD/CAM, CASE, operating systems, and simulation-oriented applications more obviously suited to OO's strengths, a different classification may result.

Reprint 3421