OBJECT-ORIENTED PROGRAMMING TRANSITION

Many software-development organizations are currently going through a paradigm shift in the way they develop software. Twenty to thirty years ago, structured programming was heralded as the method for solving complex programming problems. Now object-oriented programming is emphasized and has been adopted by many organizations worldwide. The migration from a structured programming approach to an object-oriented approach is not without its risks. This article presents the risks and benefits of continuing with the traditional structured approach as well as the risks and benefits of migrating to the object-oriented approach. This article does not overglorify the technology but notes the challenges of adoption and presents serious factors that information technology (IT) managers planning to adopt object-oriented technology must consider.

Cost-effective construction of software has been extensively studied as a central issue in information system development, and a considerable number of principles, techniques, tools, and notations have been explored. Some have proven effective in practical projects.

During the 1970s, *structured programming* was emphasized as the solution for solving software-development problems. This approach introduced criteria for the development of modular programs through top-down decomposition techniques.

During the 1990s, the *object-oriented* approach was introduced and was increasingly emphasized as the most effective approach to software-development problems. In this approach, a component of the real world is represented as an object in the software solution. One main goal of the object-oriented approach is to maintain a direct correspondence between the real-world entities and their representation in system solution. The object-oriented approach provides advances toward software-engineering concepts such as abstraction, modularity, reusability, and so forth.

When choosing between the traditional structured programming and the new object-oriented approach, industrial software developers and information technology (IT) managers often rely more on what Fenton (1) calls "unsubstantiated advertising claims and biases of producers, both academic and industrial," than on actual benefits and/or risks of these two approaches. The developers and managers must, however, ensure that the adoption of a (new) technology contributes to, rather than detracts from, their established methods of development.

The main point is that object-oriented technology is not a panacea and thus bold claims that object-oriented techniques will solve software-development problems without considering the challenges of adoption could lead to unrealistic expectations and potential disasters.

According to Everett Rogers, there are five generic innovative attributes that affect the rate of adoption of a new technology (2):

1. Relative advantage
2. Compatibility
3. Availability
4. Complexity
5. Observability

The adoption of object-oriented technology has certainly been affected by the above attributes. The objective of this article is to explore the risks as well as benefits associated with the adoption of object-oriented technology. The organization of the article is as follows: First the benefits as well as the risks in continuing with the traditional structured techniques will be reviewed. The next section analyzes costs before migration to object-oriented technology. In particular, maintenance issues, training, perceptions, resources needed, complexity, and costs will be discussed. The following section explores the benefits of the object-oriented approach and is followed by discussion and conclusions, where primary and secondary considerations in the adoption of object-oriented technology are considered.

TRADITIONAL STRUCTURED TECHNIQUES

Benefits of Structured Techniques

Structured programming techniques are not likely to disappear even when object-oriented techniques make significant inroads into program design and implementation. Until the majority of business software development organizations commit totally to another methodology, academia will continue to teach structured techniques. The reason is that software-development organizations in business continue to see benefits in using structured techniques.

In many instances, a small stand-alone task fits better in a structured environment than in an object-oriented environment. For example, it makes sense for a computer operation organization to use structured techniques to design a program to clear a system buffer. Analysis, design, and implementation are straightforward for the structured method (i.e., a call to a system function). By comparison, the C++ code using a class to model the system buffer would be at least double that for a similarly structured design using the C language with a single system call. In other words, for small problems, structured techniques fit better as a development method than do object-oriented techniques. Structured programming techniques learned prior to conversion to object-orientation will not go to waste when a new methodology becomes accepted by software-development organizations in business. Methods written using object-oriented techniques require structured techniques. Any method must follow the basic properties found in good structured programs, like cohesion and coupling.

Another reason structured techniques will continue to survive is the simplicity in dividing systems into modules, using top-down decomposition and stepwise refinement. Structured techniques can have high binding strength and low coupling. Henry and Humphrey observe (3) that the result of structured programming is the independence of programming modules. Techniques employed in structured programming improve system maintainability and avoid development fiascoes (2). These benefits of structured programming techniques are not disputed.

Risks in Continuing with Structured Techniques

The benefits of structured programming techniques do not erase the shortcomings inherent in the methodology given the software problems prevalent now and what is expected in the future. Software problems in all parts of the business community are more complex than they were when the structured methodology became accepted by businesses 20 years ago. These shortcomings become risks for a business organization when it comes to software development.

All business organizations have to deal with risk, regardless of the programming techniques they are currently using or the types of software problems they must solve. There is risk in remaining with a programming methodology, as well as with migrating to a new programming methodology. The overall solution is to minimize the risk, regardless of the solution, now and for the future. Therein lies the dilemma: the direction a business is taking today may be divergent from where it must be in the future. According to Crouch (4), the perception of the magnitude of risk from some event depends on some form of product of how often the event will occur and how serious it is considered to be in its effects. Or, more simply put, *probability* (how much or how often), cross-multiplied by the *severity* (some risk per unit of action or per event). When translated to the investment in structured techniques for a software-development organization, the use of the structured methodology increases the business risk when solving new and more difficult problems. When the benefits of changing or migrating to a new methodology outweigh the costs of the new methodology, that is the time the software development organization should plan their migration strategy.

According to Fichman and Kemerer (5) the risk of remaining with structured techniques or migrating to object technique is typically unknown. The reason is that no two software organizations are the same with respect to the software problems that they must solve and the techniques they use to accomplish their task. However, the characteristics of structured techniques are clear, based on experience with the techniques and models used since the 1970s. The waterfall model requires that the previous phase in the model be completed before the next starts. There is no direct mapping between analysis and design (6). The model's phase approach leaves a software project open to the risk of incomplete analysis (7). Additionally, process decomposition can lead to unstable designs. Structured methods suggest developing software from scratch, as opposed to reuse of existing code, thereby saving time. Saving time is critical to a business in a project's time-to-market (7). Part of the reason code is difficult to reuse in the structured environment is the difficulty of reusing a generic application. A structured methodology application developed from scratch typically did not have reuse in mind when it was designed.

Applications considered to be second and third generation (i.e., client/server applications) will require a new solution to develop software efficiently (8). The third-generation software problems are more complex than the previous generation of software, which lent itself to the top-down centralized computer environment. Structured programming does not effectively address information hiding and encapsulation required by the third-generation applications (6). Thus the ripple effect in software maintenance for structured techniques is much more prevalent. Software-development organizations have experienced these problems consistently in the structured environment. Even though software-development organizations can minimize the ripple effect with quality control measures (e.g., code reviews), they are aware that increased complexity in an application increases the chance for the ripple effect to occur. These organizations are beginning to realize that a different strategy is needed to manage the risk of future software development for their organization and the new set of programming problems.

ANALYZING COSTS BEFORE MIGRATION

Fichman and Kemerer (5) suggest formulating a transition strategy when a software-development organization is at the stage where migration to object-oriented software development is necessary to solve the new set of problems. Coleman and Hayes (9), in their experience, mention that disappointing results are attributed to the underestimation by management of the introduction of new technology. Management should have a list of their current assets or resources as a starting point. These include people and the tools that use the structured methodology. Fichman and Kemerer (5) indicate that object-orientation is a refinement of the best of software-engineering ideas from the past. To assume that an organization can just merely adopt these good ideas only and move on to developing software using object-oriented techniques is a myth. The migration requires a major investment in time and other resources for the software-development organization. Planning should not be taken lightly. Once migration toward object-orientation has taken place, there is living with the decision, because falling back to the old structured methodology will not be an option.

The primary risk when considering adoption of a new technology like the object-oriented methodology is the length of time taken to achieve the migration completely. The time frame is a major concern for the software-development industry. This risk is realized in the slow acceptance by a software development organization whose management has allowed adoption to take place without setting specific goals during the adoption process. Lack of leadership could lead to implementation failure.

Maintenance

The costliest part of a system in its software life cycle is maintenance. Maintenance costs will not change when using object-oriented techniques (10). It will continue to be the most costly part of the object-oriented methodology.

Software written properly using an object-oriented programming language can be difficult to modify, the reason being class hierarchies that are multilevels deep. They can make understanding an application's code more difficult. The newly maintained object-oriented software could also incur more errors as a result of this misunderstanding. The reason is that maintainers spend much of their time reading code in an attempt to thoroughly understand it before modifying it safely (10). As time goes by, maintainers will become better at reading code written using an object-oriented language. Initially there will be more time needed to understand the code at the beginning of the migration period and more errors attributed to maintenance during that same period. An experience by Coleman and Hayes (9) showed that debug times rose from 15% using a Pascal-structured programming environment to 45% using an object-oriented environment. However, their test was not accomplished using a true object-oriented language. Therefore, their percentages may be consistent for the typical migration process.

Training

The largest investment in preparing for object-oriented technology is training—both time and cost. Training can take months. One-week-long courses are effective for introducing managers and technical directors to object-oriented techniques but not the trained computer professional who develops code as his or her primary responsibility (11). If C++ is chosen as the object-oriented language for a software-development organization, it is highly recommended that the staff be trained in C first, according to Nebesh and Rabi (11). The reason is that C++ is a superset of C.

Some leaders in the field of object orientation do not recommend the use of a hybrid object-oriented language like C++ for learning object-oriented techniques (12). The reason is that it will be too easy for the programmer who is only trained in structured techniques to fall back and use structured techniques and not the object-oriented features present in the language. Bamigboye (13) indicates that the initial training should be language independent. Nebesh and Rabi (11) somewhat reflect the belief that a hybrid language is harder to teach and apply object-oriented techniques immediately on a project. In their experience, the first project used by the trained professionals used C++ as a better C and not an object-oriented language. However, in the next project, use of object-oriented inheritance was better. By the third project, dynamic binding was being used. For teaching C++ in a classroom setting, basic understanding and some effectiveness can be shown after 10 weeks of instruction according. The technical professional should master the basic concepts of object-orientation first (8). However, for a C language programmer to become proficient in C++ requires about a year, according to Nebash and Rabi (11). Object-oriented productivity will likely begin to climb starting at three to eight months. After 18 months, the real productivity curve begins. It is unlikely, however, that a software organization's real productivity will be achieved in less than three or four years (14). In line with training will be the need for specialists in the various areas of object-oriented programming and design (i.e., application programmers, class designers, class programmers, etc.). Gone will be the "Jack-of-all-trades," as seen in many large software organizations currently using structured techniques (7,15). The object environment simply requires the need for more specialists in a particular area of object technology for development to progress optimally. If a software organization chooses object-oriented training using on-the-job training methods, the pace must be tolerable (16). The software organization must realize that the first project may be more expensive than if the older technology methods were used. Primarily, the software organization must realize that object-oriented methods, and the time taken to learn them, are an investment in the future. Those software organizations going through the migration process are likely to experience an initial productivity decline as a result of extra initial effort to design modules for reuse (2). A library of reusable modules is difficult to achieve without proper training and tools for such a purpose. The first-time code-reuse designer will have a difficult task learning the new skill.

Another way of approaching training in the area of on-the-job training is the adoption of an apprenticeship or mentoring program for teaching the object-oriented methodology (7,8,12). The mentor is someone who can coach the student and immediately give feedback on the decisions made with respect to object-oriented analysis, design, and implementation. Learning the technology is aided if the migration effort has a champion of the object-oriented approach (9). The champion ensures that there is an appropriate budget and handles any problems that may develop with respect to the diffusion of the adoption process. The main problem with the apprenticeship/mentoring approach is finding masters or journeymen with experience in object-oriented ways. There is a shortage of trained object-oriented professionals in the business community. Training and consulting companies can supply some of the experience but are not the same as having a technology master on-site going over work on a real-life project during the learning process.

For formal classroom training, training companies recommend using lecture and lab facilities for teaching not only the language but also the object-oriented concepts (11). Weekly training sessions should be spread over a longer period of time. Nebesh and Rabi (11) have had success with a 10-week class meeting three times per week for a total of 100 h. There should be appropriate breaks during the training process for the student to implement the ideas presented in the classroom. The cost of the learning curve must be taken into consideration in all schedules by management (9). Management should realize that the time taken away from productive, project-related work is required for a greater payback on future projects using the new methodology.

Perception

Introductory software engineering is taught using hierarchical nesting of procedures and control-flow-based computing paradigms. They make new languages and new methodologies difficult for all types of developers. The reason is that unlearning a paradigm is difficult (8). Object-oriented technology is viewed as a radical paradigm shift (5,7,8,16). The reason the perception exists is due to the understanding and use of the concepts of object orientation such as objects, classes, instances, messages, methods, encapsulation, abstraction, polymorphism, inheritance, persistence, binding, and typing. Once developers understand these terms and concepts, they can apply them to design and development. Structured developers may also feel anxiety associated with learning these concepts. Managers can either help the adoption of new technology or hinder it based on their level of anxiety.

Kozaczynski and Kuntzmann-Combelles (17) feel that object-oriented technology is experimental and will continue to be for a long time to come. Coleman and Hayes (9) and Bordoloi and Hwa-Lee (6) agree that the object-oriented technology is immature but evolving. The reason for nonacceptance on a widescale by businesses is the lack of business-like tools (e.g., class libraries). However, Jacobson (16) disagrees and feels that the technology is mature today because there are at least 5000 programmers worldwide developing systems using object-oriented technology. Pei and Cutone (7) are not sure that the benefits of object-oriented development are clear today. Considering the views by many of the industry experts, there is suspicion by management viewing object-oriented technology from the inside-out that there will be a new paradigm to follow object orientation. According to Jacobson (16), there will probably not be a new paradigm in the next few years. However, no one can say for sure. If another paradigm should appear, there is a danger that the object paradigm will be abandoned just when production increases are possible for a software-development organization going through the migration process (14). In any case, object orientation suffers from low observability (2).

Resources

The resources needed to implement the object-oriented technology include not only training and a computer language but also the selection of a development method, automated tools to do design and analysis, a database manager to handle persistent objects, an operating system capable of supporting all aspects of object orientation, and other repositories that help support the development process (16).

For businesses, there are very few existing component libraries. The component library is key to fast development using the object-oriented methodology. Even with component libraries, it is uncertain that there is little more than a basic building-block approach to be attained for base structures in business applications (12). A resource, in another sense of the word, can also be considered a developer or a group of developers. For a manager to remove a person from accomplishing productive work to learn object orientation, there becomes one less person accomplishing that productive work. For many businesses today, additional personnel for the purpose of learning new techniques and methods are planned. Rarely does new methodology training occur as a whim in a business climate that tends to reduce headcount when it sees a need to do so.

Loss of Knowledge

Once migration to the object-oriented methodology takes place, use of structured ways (tools and techniques) do not mix easily with objects (2,5,17). Developers are not going to give the structured ways up to the object-oriented methodology without considering the costs and benefits to them personally. The new techniques must be of real value for a developer to want to adopt them. As an example, real personal value may mean job security, better pay, or even satisfaction in learning a new technology. As time goes by, knowledge of structured or conventional methodologies may get lost. The reason is that new personnel are beginning to be hired with knowledge of object technology only. Old software using structured techniques and languages will either have to be rewritten using object-oriented techniques or key personnel with structured knowledge will need to be retained. If rewriting from a structured methodology to object-oriented methodology is chosen, costs should be carefully considered. The expense could be large. Structured code projects needing to be rewritten using object-oriented technology have been shown to reuse as much as 60% of the original code (8). There is a significant cost to the business for the maintenance of older structured code. Should a decision be made to reengineer the structured code to object technology, the amount may look expensive at first glance. However, managers should realize that costs for an organization to maintain two development methods are higher than if only one method is used.

Retraining of structured programmers can be expensive for an organization. The mentality of retraining and ignoring what was learned in structured programming can be difficult to overcome (17). An additional problem to consider with retraining concerns traditional personnel who are allowed to use structured languages using a subset of an object-oriented language; they will revert to the old functional style (14). The retraining can be the way of integrating object orientation into the current development environment. The difficulty is in the effort to integrate slowly into object-oriented technology. The slow integration appears to be a disadvantage with respect to the tools not being appropriate for development during the early stages of a project (13). The object-oriented methodology then becomes an all-or-nothing proposition. The methodology must become the natural way to doing software development, much like many software organizations today consider structured software development.

Complexity

The object-oriented decomposition process merely helps control the inherent complexity of a software problem; it does not reduce or eliminate complexity. Small programs presented in an object-oriented fashion can potentially have many complex relationships. Of the three essential principles of object orientation: (a) encapsulation, (b) class specification of objects, and (c) inheritance; inheritance can make the trace of program dependencies even harder to find (10). Even a simple architecture can lead to complex run-time structures (9). Considering the complexity of these structures at run-time, flow of control can be almost impossible to discern when attempting to troubleshoot a program problem manually. Considering the situation described previously, object-oriented code can be difficult to navigate when simulating program execution and subsequently complicate program maintenance. Another useful feature of object-oriented programming is the use of dynamic binding. Dynamic binding is a powerful feature but creates an element of the unknown not present in the structured methodology. Debugging tools are essential to reduce complexity in the object-oriented methodology. As a process technology, object orientation rates unfavorably in the area of complexity (2). Unfortunately, part of the hype associated with object-oriented methodology when presented to managers thinking of introducing the methodology, describe the simplicity of the object-oriented methodology and not what could complicate its use when adopted.

Other Costs

When considering migration from one technology to another, there is value in examining the process used at a similar organization. However, simply using a migration method employed at another company will not work when considering migration in a different company. Every business environment is different, as are the problems that they must solve. Object technology must be offered as an industrial process that can be tailored to various types of development organizations (16). Regardless of the nonsimilarities between groups when adopting a new technology, it is helpful to have personnel available who have been through a similar conversion process. Software engineers have traditionally been biased toward writing new code. There will be a need to remove the development mindset of “not invented here” when it comes to reuse or the need for new functionality. The traditional mindset must change to adopt a new culture and values, according to Fichman and Kemerer (2) and Bamigboye (13). The reason the traditional mindset does not work for the object-oriented methodology is that the focus of development is shifted toward analysis and away from implementation (16).

Costs can also rise due to underestimation by management of the problems associated with the introduction of new technology (9). Project managers need to act prudently. The focus away from the main track of object technology and quality production can be diverted by concentrating on errors instead of the process (2). Business and human practices must be reengineered along with software (12).

BENEFITS OF OBJECT-ORIENTED TECHNIQUES

The reason object orientation has a promising future is the type of technology that is being deployed in the industry and the kind of software being written by software-development organizations for that technology. Client–server processing and parallel processing have high visibility in supporting today’s business processes. If a business is not employing one of these new technologies on a current project, it will be in the future. Inherent in the object-oriented architecture is the ability to identify software in separate and distinct sender (client) and receiver (server) roles. These roles yield units of low coupling, stronger cohesion, and higher modularity, compared with similar structured modules.

The characteristic of inheritance in object orientation for the client–server model yields the most capability for producing new software in a development organization. Inheritance takes reusability to a higher level than in the structured methodology sense. Reusability, as a result of inheritance, reduces risk. Reusability is accomplished with evolution from smaller, proven systems (7,8). Reusability leads to increased understandability, simplicity, and is closer to human cognition (7). Reusability has the biggest payoff for businesses. Object orientation produces more maintainable code than does procedure-oriented code. The main reason program code is more maintainable is because the code is localized in the design of a class. The localization makes the module (or class) more resilient to change (7), reducing the ripple effect. With maintenance being the most costly part of a software’s life cycle, the resilience affects the financial situation of a software-development company or organization. When the design of a system is completed correctly, there will be fewer modules, fewer sections, and fewer lines of code to consider during maintenance (6,7). Less volume to maintain translates to less time understanding the code in the maintenance phase and less time for initial development. Fewer errors translate to more time for a development organization to do other development and maintenance. The bottom line of a business is affected by its costs. The claim to reduce costs forces a business to look seriously at object technology.

Abstract data types make implementation of distributed processing practical (14). In order to model the real-world processes that a business must deal with every day, new structures to represent real-world entities need to be created while current models need to evolve completely. Abstract data types are the key to defining classes when the implementation phase is presented.

Ultimately, object orientation development will shorten the development time and reduce a software product’s time-to-market (7). To many software companies, time-to-market is key to acquiring the customer at the most opportune time. Fichman and Kemerer (2) agree, by mentioning that object-oriented qualities support the reduced time-to-market principle by reducing development, reducing maintenance cost, improving flexibility, and improving overall software quality.

DISCUSSION AND CONCLUSIONS

The complexity of migrating to the object-oriented methodology shows that research by business management is needed before the technology can be adopted by any software-devel-

Table 1. Primary and Secondary Consideration in Adopting Object-Oriented Technology

Item	Primary Considerations	Secondary Considerations
Training	• Classroom • On-the-job • Language-independent	• Apprenticeship • Journeyman • Conferences and seminars • Books, technical articles
Cost	• Conversion money • Document the process	• Small team for first project • Start with a low-risk project • Pilot projects; prototypes • First project normally unsuccessful
Tools	• Language • Libraries • Database • Operating system • Browser, debugger, repositories	• Should be complementary • Interface easily • Pure vesus hybrid language • Support multi-projects simultaneously • Adaptable to changing environment
Current investment	• Integration (minimize disruption) • Migration (strategic change)	• Incremental
Role definition	• Shift toward analysis • Emphasis on application • Requirements analyst • Application design, programmer • Class designer, programmer • Database designer • GUI designer, programmer • Library designer, strategist	• Reengineer the business
Time	• Increased learning time	• Time to gain proficiency
Evaluation	• Realistic conversion strategy • Conversing original investment • Integrating into the current system	• Modify the reward structure • Overcome resistance to change

opment organization. The individuality that each software-development organization possesses suggests that a formal process be used. A current assessment of the organization needs to take place with respect to a series of technology issues. The process needs to be tailored to the organization based on the assessment. As of yet, there is no known formal process put in place for assessment.

Due to the lack of a process, the software-development organization considering adoption of object-oriented techniques should put together a list of activities or issues that need to be resolved. As a first step, the organization can act on the list.

Table 1 highlights the issues discussed. It is more of a template that will require fine-tuning and additional data by the migrating software-development organization. The fine-tuning would include additional details about the business and the types of issues the business is faced with as it migrates to the object methodology. In addition to the list of items (column one), the table includes lists of major events (primary consideration) that will be needed to accomplish each item. The events are stated in broad terms and can be accomplished in various ways. Because of the number of ways to accomplish the events, items may be difficult to complete. Therefore, column three (secondary consideration) suggests additional sources or modifiers to the events in column two. The introduction of object-oriented technology should be accomplished using incremental steps and can be aided by a champion of the technology (9), but it is not required. Education is the only item on the list that can be decided and accomplished by an individual in a software-development organization without management's approval to start the migration process. Training for a new method also requires buy-in by the individual when management announces intentions to migrate to a new methodology. It happens to be the first item on the list. Not every developer will progress through the training at the same pace. For some, the training will be difficult. Minimum time and knowledge levels should be required for the development staff to gain proficiency.

The first project to seriously use object-oriented technology and methodology should involve a small team (9) and a low-risk project (7). The principal roles and responsibilities, and how they interact, need to be defined beforehand (12). Documentation by way of an activity log should be used to indicate problems or methods in the processes that were tried, failed, and remedied for future reference. A wait-and-see approach on a pilot project is the wiser choice before investing more heavily in the object-oriented technology. The trial project should be developed through a series of prototypes by emphasizing performance over functionality. The prototype approach can be risky, however. A prototype product should not be delivered (9). Integration of the final product should be accomplished transitionally (8). The development group should establish standards to control when and how dynamic binding and polymorphism are used. Documentation is key for future development (10), using the key features of the object methodology.

Once the first product using the object-oriented methodology is delivered, key questions need to be asked about the current and proposed future software-development environment (8):

- Is the methodology replacement strategy realistic?
- Are there ways to conserve the current investment?
- What are the ways object systems can be integrated into the current system?

• What are the needed skills and tool sets required for the new environment?

If a software-development organization cannot easily answer the above questions, their implementation strategy is not complete. The time is right for a manager to intervene and take control of the migration process. Hastily completing a project using object technology and thereby jumping on the object-oriented bandwagon is not the best strategy for success (9). Once the object-oriented methodology has become a natural part of an organization's environment, the object-oriented approach could provide the basis for developing a completely automated approach to system analysis, design, and implementation (18).

Trends in computing are leaning toward more complex data types and more complex systems. These seem to favor the object technology approach. At face value, there appears to be more risks than benefits. However, there can be ways of managing the migration to object technology without a major shock when it comes to allocating time and resources. Planning remains the key.

BIBLIOGRAPHY

1. N. Fenton, How effective are software engineering methods, *J. Syst. Softw.,* **22**: 141–146, 1993.
2. R. G. Fichman and C. F. Kemerer, Adoption of software engineering process innovations: The case of object orientation, *Sloan Manage. Rev.,* **34** (2): 7–22, 1993.
3. S. Henry and M. Humphrey, Object-oriented vs. procedural programming languages: Effectiveness in program maintenance, *J. Object-Orient. Programm.,* **6** (3): 41–49, 1993.
4. E. A. C. Crouch, *Risk/Benefit Analysis,* Cambridge, MA: Ballinger, 1982, Chapters 2, 3, 4, and 5.
5. R. G. Fichman and C. F. Kemerer, Object-oriented and conventional analysis and design methodologies, *IEEE Comput.,* **25** (10): 22–39, 1992.
6. B. Bordoloi and M. Hwa-Lee, An object-oriented view, Productivity comparison with structured development, *Inf. Syst. Manuf.,* **11** (1): 22–30, 1994.
7. D. Pei and C. Cutone, Object-oriented analysis and design: Realism or impressionism? *Inf. Syst. Manage.,* **12** (1): 54–60, 1995.
8. S. Rabin, Host developers to object technicians: Transition strategies for OO development, *Inf. Syst. Manage.,* Summer: 30–39, 1995.
9. D. Coleman and F. Hayes, Lessons from Hewlett-Packard's experience of using object-oriented technology, in *International Conference on Technology of Object-Oriented Languages and Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1991, pp. 327–333.
10. N. Wilde, P. Mathews, and R. Huitt, Maintaining object-oriented software, *IEEE Softw.,* **10** (1): 75–80, 1993.
11. B. Nebesh and M. Rabi, Teaching object-oriented technology through C++ to professional programmers, *11th TOOLS Conf.,* 1993, pp. 627–636.
12. C. M. Pancake, The promise and the cost of object technology: A five year forecast, *Commun. ACM,* **38** (10): 33–49, 1995.
13. A. Bamigboye, Object technology: To migrate or to integrate . . . that is the question, *Object Mag.,* **5** (6): 41–44, 1995.
14. R. T. Due, Object-oriented technology: The economics of a new paradigm, *Inf. Syst. Manage.,* **10** (3): 69–73, 1993.
15. M. Page-Jones, Education and training for real object-oriented shops, *J. Object-Orient. Programm.,* **10** (1): 51–53, 1994.
16. I. Jacobson, Is object technology software's industrial platform? *IEEE Softw.,* **10** (1): 24–30, 1993.
17. W. Kozaczynski and A. Kuntzmann-Combelles, What it takes to make OO work, *IEEE Softw.,* January: 21–23, 1993.
18. T. J. Heinz, An object-oriented approach to planning and managing software development projects, *Inf. Manage.,* **20**: 281–293, 1991.

HOSSEIN SAIEDIAN
University of Nebraska at Omaha
JACK URBAN
U.S. West Telecommunications

OBSERVABILITY. See CONTROLLABILITY AND OBSERVABILITY.