Theme Feature

Object Technology and Reuse: Lessons from Early Adopters

Four longitudinal case studies illustrate the costs and risks of early OO adoption, including the difficulty of achieving systematic reuse in practice. The authors recommend general strategies and specific tactics for overcoming adoption barriers.

Robert G. Fichman
Boston College

Chris F. Kemerer
University of Pittsburgh

The recent commercialization of object-oriented software process technologies has been driven by pragmatic desires to increase productivity, shorten cycle times, enhance maintainability and extensibility, and more fully satisfy user requirements. Although an organization's transition to OO typically centers on the use of an OO language such as C++ or Smalltalk, other changes are also necessary if expected benefits are to be realized. These changes include new approaches to analysis and design, greater use of iterative/incremental development, and—perhaps most crucial—an increased emphasis on component reuse.

The image of developers assembling applications from existing components—rather than building them from scratch—has been a central part of the OO vision since it emerged as a commercially viable technology in the late 1980s.[1] Besides potentially increasing productivity and reducing cycle time, the reuse of proven components (especially black-box reuse) should reduce the maintenance burden and lead to more reliable and efficient systems.[2]

Although adopting a new software process technology is rarely easy, the early adoption of complex, frame-breaking technologies such as OO is likely to be particularly difficult and risky. In a previous article, we used innovation diffusion and economic theories to develop a two-dimensional framework that identified adoption barriers at the organization- and community-wide levels.[3] The sidebar "What Drives Innovation Adoption?" describes some of our conclusions.

In 1992, we initiated four case studies of early OO adopters, which we continued until 1996. These longitudinal case studies gave us a richer description of the actual challenges early adopters faced and helped us develop recommendations about how organizations can succeed with adoption in spite of potential barriers. Our results are particularly significant because longitudinal case studies avoid the inevitable bias of most case studies, in which results are analyzed retrospectively.

The cases illustrate lessons that should be of direct interest to organizations attempting to achieve greater reuse through OO and those having to confront issues about software process technology adoption. All four case sites encountered learning barriers and barriers from immature technology; more important, all sites found it difficult to achieve systematic reuse.

SITE OVERVIEWS

Table 1 summarizes the characteristics of the four case-study sites. Each site met four criteria: They were implementing a fully OO language; the project was staffed primarily by permanent employees, the adoption process was new enough for the ultimate outcome to be unknown, and the organization seemed well positioned for adoption success as measured by funding, staffing, training, top management support, and so on. Figure 1 shows the timelines of OO adoption activities for all four case-study sites. To gain this information, we conducted a series of interviews with staff at various levels, including developers, project managers, and other executives. Interviews typically lasted an hour, and we conducted on average seven sessions for each site. Between April 1992 and June 1993, we conducted interviews in person at each site. In August 1995, we conducted telephone interviews. In October 1996, we made telephone contact to get a final update on the adoption status.

EnergyCo

In May 1992, we visited EnergyCo, our first site. The company had begun implementing a Smalltalk development environment in fall 1989 to support the construction of two very large, complex systems—a Land Management System to manage land leases and related rent and royalty payments and a Gas Management System to manage gas accounting and so on. (All company, project, and system names are fictitious.) As Table 2 on page 49 shows, the primary trigger for this adoption was a division-wide reengineering effort, which stimulated EnergyCo's desire to use state-of-the-art tools to develop the systems that would support the new processes. As one manager put it, "We wanted to select the right platform for five to 10 years from now; we don't want to catch up, but to leapfrog

0018-9162/97/$10.00 © 1997 IEEE

What Drives Innovation Adoption?

In developing our framework on OO adoption barriers, we first looked to research on innovation diffusion—how innovations spread through a population of potential adopters over time. This research was a natural starting point if you view OO as a software process engineering innovation. However, most research in this area fails to consider communitywide forces, which can determine whether technologies such as OO will be sustained over a broad community of adopters.

For that reason, we decided our framework should have two dimensions: organization and community. For each, we considered the characteristics that facilitate or hinder adoption.

On the organization dimension, we identified five characteristics that innovation researchers have shown lead to adoption barriers.

- *Low relative advantage*. The technology offers few advantages (in cost, functionality, and so on) over existing technologies.
- *High complexity*. The technology is relatively difficult to understand and use.
- *Low compatibility*. The technology is incompatible with existing methods. Either existing staff must be reskilled or new staff with the right skills must be hired.
- *Low trialability.* The technology is difficult to experiment with and implement incrementally and still get a positive net benefit.
- *Low observability*. Results and benefits are hard to observe and communicate.

We performed a conceptual analysis of OO on these characteristics and concluded that the last four strongly applied. As a result, we expected that early adopters would experience difficulties, especially related to learning, in their initial attempts to implement object technologies.

Communitywide, we identified four characteristics that economists have shown lead to adoption barriers for new technologies.

- *Prior technology drag.* The technology must displace a well-entrenched prior technology.
- *Irreversible investments.* The technology requires large irreversible investments in products, training, and accumulated project experience.
- *Lack of sponsorship*. The technology has no strong entity to define the technology, set standards, and subsidize early adopters.
- *Low expectations.* The technology faces negative expectations about whether it will be pervasively adopted in the future.

We concluded that the first two characteristics strongly applied to OO, indicating that a large community of committed OO adopters would be slow to emerge. As a result, we expected that early adopters would bear substantial additional costs arising from an extended period of technological immaturity.

Table 1. Characteristics of sites in the four case studies.

	EnergyCo	BankCo	FinCo	BrokerCo
Parent company	US-based international energy company	UK-based retail banking firm	US-based international financial services company	US-based international financial services company (subsidiary of FinCo)
Business unit	IT organization for Exploration and Production division	Corporate IT department	IT organization for Retail Marketing Services division	IT organization for UK-based brokerage division
IT development budget in 1993	$15 million (division)	£ 400 million (company)	Not available	£ 3.2 million (division)
IT development staff in 1993	80 (division)	1,000+ (company)	75 (division)	50 (division)

current technology." Respondents generally viewed OO as a powerful technology that enabled a "building-block approach" to software development. Prior to adopting OO, EnergyCo had a fairly typical mainframe environment staffed with 80 IT professionals.

One of the most distinctive features of the company's approach was its willingness to substantially invest in the training and infrastructure to support OO development. As Figure 1 shows, in 1990 the company had put a team of 15 developers—all existing staff—

Table 2. EnergyCo's OO adoption background.

Characteristic	Description
Prior environment	Mainframes running IBM operating systems and databases (MVS, IMS, DB2), PL/1. Standard development methodology based largely on Yourdon
Primary adoption triggers	Desire to use state-of-the-art technology to develop systems to support redesigned business processes; strong recommendation of consultants
Prior OO activities	None
Approach to staffing and training	Existing staff (15 developers), extensive training, mentors
Languages and tools	Initially: ParcPlace Smalltalk, internally developed tools (modeler, GUI builder, query tool, RDB interface), DB2, Unix, Windows, Banyan Vines Eventually:ParcPlace Smalltalk and VisualWorks, Envy, Versant ODB, Unix, Windows, Banyan Vines.
Style of OO development	Iterative; heavy use of prototyping; proprietary methodology initially, but later a combination of CRC cards, Wirfs-Brock, and use cases
Primary stated barriers/concerns	Learning curve, absence/immaturity of tools, complexity of problem domain, managing iterative development, finding and retaining OO staff, explaining OO benefits to senior management
Primary mechanisms to promote reuse	Putting developers in same room; e-mail broadcasts
OO expenditures through 1995	Over $10 million, almost a third of which went for infrastructure tools
Status of OO adoption as of each date of contact	May 1992: Piloting/enhancement of LMS planned for next 12 months, rollout to follow; OO considered technology of choice for new systems June 1993: Implementation of about a third of LMS imminent; GMS under development; OO considered technology of choice for new systems August 1995: Selected portions of LMS implemented in phases over preceding two years; Packages now the technology of choice; OO intended only for especially complex problems October 1996: LMS terminated due to cost of loading leases; portion of GMS implemented and considered a success

through six weeks of OO indoctrination. Several team members had then begun developing a series of infrastructure tools, including a CASE-like design tool, a graphical user interface development tool, a relational database schema generator, a tool to manage a dynamic interface between Smalltalk and DB2, and an ad hoc query tool. Only when these tools were complete at the end of 1991 would they be ready to develop the LMS in earnest, which would become the focal project at this site. The GMS was initiated a few months later.

Over the next four years, the six-member LMS team encountered several setbacks. First they discovered the initial design architecture inherited from the original reengineering effort was inadequate. Then a pilot system, which they developed after reanalyzing requirements, also had a fundamentally flawed architecture. At the same time, most of the infrastructure tools were retired, although some were replaced with commercially available alternatives. The team then launched a second reanalysis, this time with use cases. Finally, after replacing key portions of their technical infrastructure, the team implemented small portions of the original system in phases. By August 1995, ambitions for the system had been scaled back considerably; it was unclear which remaining parts of the system (as originally envisioned) EnergyCo would fund. When we last contacted them in October 1996, EnergyCo had terminated the LMS project after determining that the cost to actually load up 20 million leases was prohibitive relative to benefits over the current land management system. However, a portion of the GMS to support gas trading was fully operational, and EnergyCo managers viewed it as a particularly innovative and worthwhile application. The GMS may have succeeded because it did not have the unanticipated and prohibitive data entry cost involved with leases.

BankCo

At our first visit in January 1993, BankCo had just finished developing the Process Modeler/Generator, a small single-user system to design and generate applications that would support the rollout of new business processes in the branch offices. The company viewed PM/G specifically as a project to evaluate object technology.

As Table 3 shows, the primary trigger for OO adoption was the company's goal to replace all branch-based systems with modern workstations in a client-server architecture within five years. BankCo's IT function was large (2,000+ IT professionals) and fairly traditional with a centralized mainframe-based architecture.

As Figure 1 shows, the PM/G development project began in March 1992 and continued for eight months. The external mentor mentioned in Table 4 helped establish management procedures and made it easier for the team to grasp OO concepts and techniques. Although the original PM/G was never implemented as intended because of technical difficulties unrelated to OO, a pilot project with a slightly revised PM/G was implemented within the IT function to track mortgage applications.

In 1993, the pilot project culminated in a detailed report describing the advantages of the OO approach and recommending object technology as the strategic future direction. Partly because of this report, BankCo started three initiatives: It created an internal consulting group of the pilot team members, tasked with

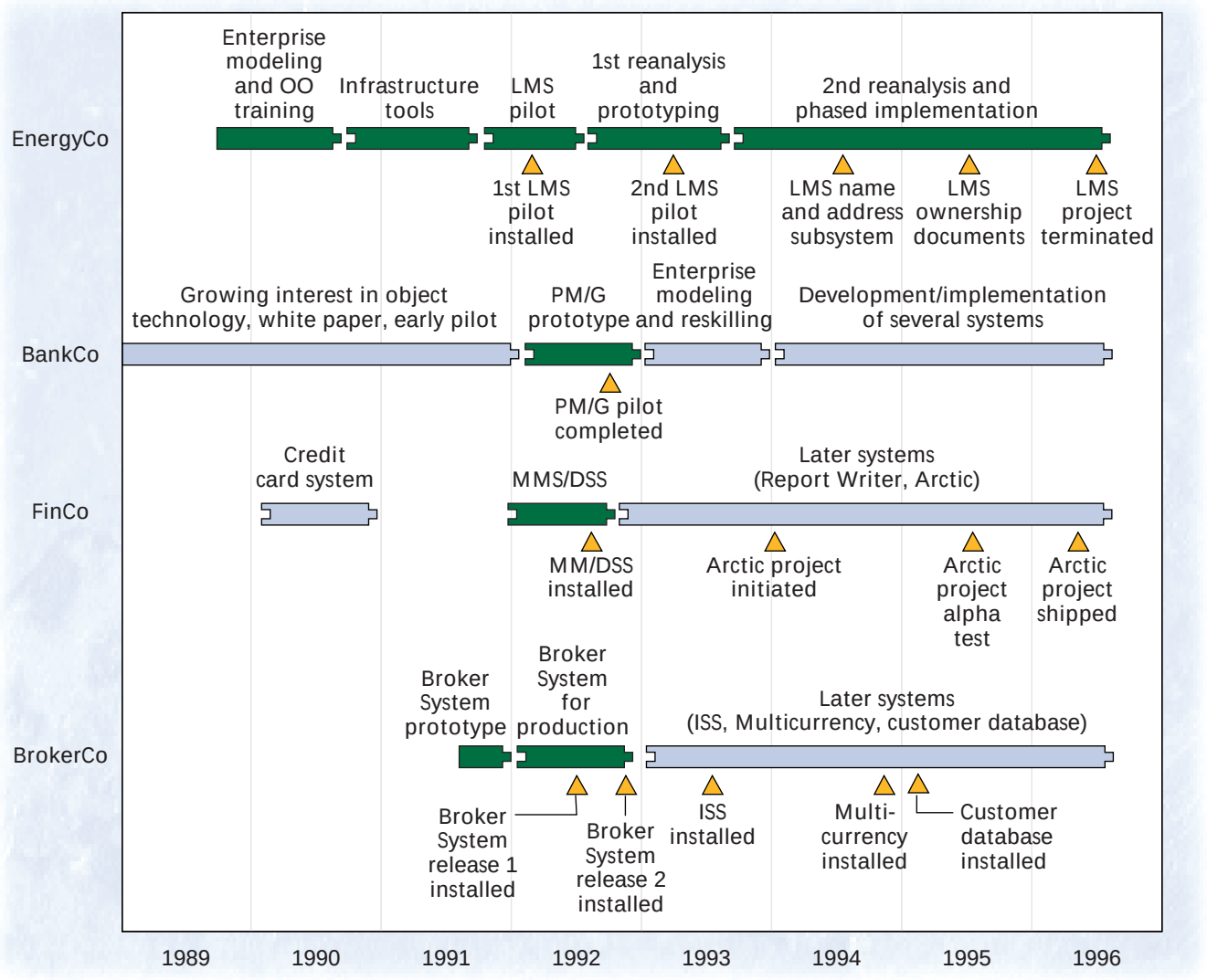

Figure 1. Timelines for OO activities in the four case studies, including activities related to focal projects in the adoption of OO (green lines) and activities related to OO but not to focal projects (blue lines). The triangles identify project milestones.

investigating tools and "spreading the gospel of OO." It began a far-reaching program to retrain developers in the "brave new world of PCs, client-server, Windows, and object technology." It launched a large enterprise modeling effort, using OO tools, to design systems that would support the "bank of the future."

In the ensuing three years, BankCo assembled a C++ development organization of about 100 developers—half internal and half from outside—and set it to work delivering client-server applications using C++, Windows NT, and Sequel Server. Although the company made steady progress delivering applications (completing about six significant projects), concerns lingered about the extent to which OO principles (especially subclassing) were actually being followed and the extent to which reuse was being achieved. One manager observed,

> I would have taken the original team and used it to concentrate expertise and build things up more slowly. What we have ended up with is a lot of people who know a bit about OO and actually apply some of it, and some who just know the words and don't bother.

BankCo also became interested in Visual Basic as an alternative to OO and formed two factions—one advocating a "build for the long term" philosophy using C++ and OO, and another advocating a "build it fast" philosophy using Visual Basic.

FinCo

At our first visit in June 1993, FinCo had completed a mass- mail system to support marketing efforts, and two other OO projects were underway. As Figure 1 shows, the company's first foray into object technology had occurred in 1990, with the development of a system to support the processing of correspondence from credit card customers. An executive in the Retail Marketing Services division had hired a systems developer/consultant with extensive OO-related academic

Table 3. BankCo's OO adoption background.

Characteristic	Description
Prior environment	Mainframes, IBM operating systems, Cobol, PL/1, Assembler, IMS, DB2
Primary adoption triggers	Interest in using latest technology to support widespread implementation of client-server in branch offices
Prior OO activities	Membership in Object Interest Group; white paper on object technology; help desk application built by external contractor using rule-based system
Approach to staffing and training	Pilot project: Four existing staff members and one external mentor; some training provided Later projects: Combination of retrained internal staff and external hires
Languages and tools	Pilot project: Intellicorp Kappa PC, DOS/Windows Later projects: C++, Sequel Server, Windows NT
Style of OO development	Pilot project: Iterative, prototyping, little formal analysis Later projects: Iterative/evolutionary, prototyping, custom methodology based on OMT with elements of Martin & Odell, Wirfs-Brock, and use cases
Primary mechanisms to promote reuse	Pilot project: Training; very small team; informal communication Later projects: Unknown
Primary stated barriers/concerns	Pilot project: Learning curve Later projects: Learning curve; spreading OO expertise too thinly; possible reversion to non-OO principles; controversy over OO versus Visual Basic
OO expenditures	For PM/G pilot: £ 400,000
Status of OO adoption as of each date of contact	January 1993: Initial pilot/evaluation completed; future use of OO uncertain; further evaluation projects planned and ongoing August 1995: Several systems completed; OO the strategic direction for new distributed systems supporting retail services October 1996: OO is technology of choice for distributed systems; research under way on how to bring some of the benefits of OO to legacy systems

Table 4. FinCo's OO adoption background.

Characteristic	Description
Prior environment	Mainframe applications in Cobol; client-server applications based on C and Windows
Primary adoption triggers	Belief that object technology provides a more robust platform for development of client-server applications
Prior OO activities	One large system developed in 1990-91 but not implemented because of political infighting
Approach to staffing and training	All hired externally, most with extensive OO experience, all with at least some experience in C or client-server
Languages and tools	Initially: C++, internally developed tools (foundation classes, GUI builder), SQL database, Windows Eventually: Objective C, internally developed tools (foundation classes, GUI builder), SQL database, NextStep
Style of OO development	Iterative; heavy use of prototyping; combination of Booch, OMT, and use cases
Primary stated barriers/concerns	Understanding of OO by top management; acceptance of OO by tool vendors and other third parties
Primary mechanisms to promote reuse	Weekly design reviews; new classes centrally reviewed
Total OO effort through 1995	25+ person years [our estimate]
Status of OO adoption as of each date of contact	January 1993: MMS/DSS complete; technology of choice within RMS; use by other divisions uncertain August 1995: Multiple systems completed in RMS; Arctic under way, OO the strategic direction companywide October 1996: Arctic shipped

training and industry experience. The developer assembled a team consisting of an OO architect and seven other developers—most of whom had OO and/or client-server experience. The RMS executive viewed the resulting credit card system as a success from a technical standpoint. However, because the system had been developed outside the main IT division, political battles over its control had led, in the executive's opinion, to the system's ultimate failure.

In 1992, the executive reassembled several of the original members of the credit card system staff, plus three new hires experienced in developing workstation applications, OO programming, or both, and initiated an OO project to develop a mass-mail system and a marketing decision support system (MMS/DSS). Both were to be supported by information in a large customer database. The original technical architecture for the system was MS Windows and C++ on the workstation side, and a relational database on the server side. Later, after the system was nearly complete, FinCo decided to port the systems to more powerful Next workstations and Objective C.

The initial version of MMS/DSS took eight months. The team developed the initial system in six months, and could reuse up to 85 percent of the code when FinCo decided to port the system to the Next platform. As of June 1993, the system had been in production use for several months, some enhancements had been completed, and an additional series of enhancements were planned or under way. As Table 4 shows, the

Table 5. BrokerCo's OO adoption background.

Characteristic	Description
Prior environment	AS/400
Primary adoption triggers	Arrival of new IS director after failure of several prior directors; visit by two primary players in FinCo's OO efforts
Prior OO activities	None at BrokerCo (but considerable at FinCo)
Approach to staffing and training	Most hired externally; temporarily "borrowed" key developers from FinCo
Languages and tools	C++, internally developed tools (foundation classes, GUI builder), Sun servers running Sybase, AS/400 database as repository
Style of OO development	Iterative; heavy use of prototyping; no formal methodology
Primary stated barriers/concerns	Not doing a formal analysis phase; not defining objects generically enough; not putting in procedures to ensure reuse
Primary mechanisms to promote reuse	None on initial projects; subsequently staffed an infrastructure group to institute a reuse program
Total OO effort through 1995	50+ staff years (authors' estimate)
Status of OO adoption as of each date of contact	January 1993: Broker System installed; Investor Support System and Multicurrency projects under way; OO used for all new development; long-term transition plans uncertain
	August 1995/October 1996: several systems installed; OO used for all new development; AS/400 still heavily used

RMS executive was committed to using object technology for all major new projects in his area, although he was unsure of the long-term role for object technology within other divisions. By August 1995, however, FinCo had established a fairly mature organization of developers delivering genuinely OO applications in C++ and Objective C, and had initiated Arctic, a very large project to develop consumer software that would support online trading.

Arctic encountered setbacks in 1995 and 1996, including missed internal shipping dates. Reasons cited include feature creep, difficulty getting required integration between workstations and back-end mainframes, and a culture clash between the OO expert team and other divisions. Despite these setbacks, Arctic shipped in summer 1996.

BrokerCo

In January 1993, when we visited BrokerCo, the IT group had completed its first OO application, Broker System, an information system to support the brokerage staff. The group was developing enhancements to that system and to some related systems. BrokerCo traditionally relied heavily on purchased packages and used traditional practices for developing any custom systems.

As Table 5 shows, two events in summer 1991 triggered BrokerCo's adoption of object technology. The first was the hire of a new systems development director, who was given a mandate to create a more effective and responsive development organization. The second was a visit from two main players in FinCo's OO activities, the RMS executive described earlier and the expert who had assembled the original OO team. Together, they convinced the director that, given his desire to move to a client-server approach, he should strongly consider using object technologies.

Training was minimal; the initial Broker System team consisted of six externally hired professionals who either had experience developing OO applications using C++ or had some other skill specific to the technical architecture (such as Sybase experience). The elapsed development time was about one year; about six staff-years were required for the initial version. One developer was devoted to maintaining and enhancing a tool kit inherited from FinCo, which contained classes to support basic system functions like data structures and GUI objects.

The director viewed the Broker System as successful. In 1993, five brokers were using the system daily, conducting some 500 transactions between them. Some schedule slippage had occurred, in part because the director had set unrealistic deadlines as a motivational device. BrokerCo had also decided to skip a formal analysis phase and instead to rely entirely on rapid prototyping for requirements analysis. Consequently, the team did not uncover gaps in system functionality until user testing.

In January 1993, the director was still uncertain about the long-term role for object technologies. By August 1995, however, BrokerCo had delivered multiple systems based on object technology (including an Investor Support System and support for multiple currencies) and OO was the technology of choice for all new development. At our last contact in October 1996, more systems had been implemented, and OO remained the technology of choice for new systems.

ADOPTION CHALLENGES

All four case sites encountered considerable challenges in making the transition to OO development. Of these, three were pervasive.

Organizational learning

Respondents at all four sites noted the difficulty of grasping OO concepts, and the steep learning curve in general:

> Everyone would be working with Smalltalk for a couple of months and then have a religious experience,

> . . . you have this tremendous awakening—like a curtain parting.—EnergyCo manager, 1992

> A good C programmer does not turn into a good C++ programmer. A structured analyst is not necessarily going to make a good OO analyst. And an [expert] Cobol programmer is likely to become unbelievably demotivated when he is a lousy C++ programmer for the first year.—FinCo executive, 1993

> Is it better to hire fresh people, untainted by 18 years of whatever, or can you really retrain people? We can teach everyone here how to program in C, but it's the way of thinking [that's hard].—BrokerCo director, 1993

To ensure success with OO, adopters must learn fundamental OO principles, how to use commercial implementations of OO languages and supporting technologies, and how to reconfigure team roles, structures, procedures, and incentives in light of the radical differences between OO and conventional development.[4] In highly complex technologies such as OO, the difficulty of organizational learning can be the primary barrier to successful adoption.[5,6]

Moreover, to truly grasp OO principles, adopters need an extended period of learning by doing. As one FinCo developer put it, knowing what the words are and knowing what they mean are two different things. Respondents typically noted that, although classroom training and book learning was important, they did not truly understand OO principles until they had had considerable hands-on experience—typically months of working closely with an experienced developer or mentor.

Thus, unless experienced OO developers are hired from outside, learning will necessarily be time-consuming and unpredictable. EnergyCo and BankCo managers felt that the on-site assistance of one or two mentors was enough for a team of novice OO programmers to learn by doing. Managers at FinCo and BrokerCo disagreed; they felt they needed OO developers with years of experience to build an adequate team. With experienced personnel carrying the primary burden of development, novice OO programmers might begin with minor roles, growing into proficient developers over many months. When asked in 1993 to comment on having a team of novice OO programmers carry the primary burden of development, a FinCo manager replied,

> If managers were committed to it and were going to make people do it, they'd have some chance. But that's a recipe for frustration. You'd better be committed [enough] to have your first projects fail ... or show no net gain... and to stick with it for two or three years. Back in my consulting days, we came in on the second wave of lots of projects like that. But at least management was committed enough to know it was the process [at fault].

For some developers, the burden of learning by doing was compounded by a lack of experience with other technologies typically used on OO development projects. As one EnergyCo project leader explained in 1993,

> It was a series of hurdles. Some people had never done anything on workstations, never used a mouse, never done anything in a graphical environment. I went to college late enough to be on somewhat familiar ground. Other people seemed to have barriers.

FinCo and BrokerCo avoided this additional burden by almost exclusively employing people on OO projects who were at least proficient in surrounding technologies, such as C and GUI- and workstation-based development. BankCo approached the problem another way for their pilot project. Initially they used KappaPC, a high-level OO development language that insulated developers from many of the technological details of the underlying platform.

Technological immaturity

At all four sites, the costs of adoption, including learning-related costs, were magnified considerably by the absence or immaturity of tools to support OO development:

> We had thought Smalltalk and OO were more ready for prime time. It was not so much the language itself, as the lack of a GUI, a code-management tool, a report writer, and so on. —EnergyCo manager, 1995

> [Internal critics at FinCo] go out and read articles and say show us the 4-to-1 or 10-to-1 development improvement. But a couple of pieces were missing. There were no good class libraries—we were writing our own—substantially for what we were doing. There are no real benefits in productivity to doing OO development unless you have some class libraries. It's fun and all but it isn't real productive until you have a code base that you can work on top of. And it hurt.—FinCo executive, 1993

Early adopters of network technologies like OO incur considerable costs simply because they have joined a small and immature technological network.[3,7] With our longitudinal approach, we can clearly document the nature of some of these costs. While OO is, of course, more mature now than it was in the early 1990s, we believe our results can help organizations contemplating the early adoption of future technologies.

Most early adopters are aware of immaturity as a general problem, but they fail to anticipate the com-

The costs of adoption were magnified considerably by the absence or immaturity of tools to support OO development.

Managers also complained about the extra costs of software development from the lack of supporting tools.

plete array of necessary supporting tools and to establish the status of such tools in advance. This can be difficult given the tendency of vendors and other parties to exaggerate the variety and quality of tools and the degree to which they actually work well together. EnergyCo in particular was unpleasantly surprised by OO's immaturity:

> OO is advertised as a fast development tool, and as being more mature than it really is. Initially the learning curve was very steep.—EnergyCo manager, 1993

> ...we got the impression that lots of reusable components were available... that part of it turned out to be false.—EnergyCo developer, 1993

Managers at both FinCo and BrokerCo also complained about the extra costs of software development from the lack of supporting tools:

> I'm a senior VP of financial services, and I have 70 or 80 people in this group. We're in another Windows class library evaluation—why? How is that possibly in [FinCo's] best interest? And NT compatibility. I don't want to think about this stuff. It's a waste of time.—FinCo executive, 1993

> Probably 200 vendors could do a better job of maintaining and enhancing the toolkit...So we are really well out in front of the industry; [this] puts us in the position of doing things that don't add value. At the end of the day, our clients don't care about our toolkit.—BrokerCo manager, 1993

Three case sites—FinCo, BrokerCo, and EnergyCo—devoted considerable resources to developing complementary tools, which they expected would one day be available commercially. FinCo and BrokerCo developed tools to support the creation of GUIs and to support integration with other technologies, such as relational database management systems. EnergyCo made a far greater investment in a portfolio of five infrastructure tools. It developed three of these tools—the schema generator, the dynamic interface manager, and the query tool—because of the perceived absence of an industrial-quality OO database management system. (EnergyCo had planned to use a particular OO database, but discovered that it did not integrate well with Smalltalk.) In the end, the company retired four of the five infrastructure tools. In some cases, changes in the technical approach made the tool unnecessary; in others, a commercial product became available that could do the same functions. As one manager put it,

> We're in the business of building applications, not tools. Management doesn't want to pay to maintain those tools.—EnergyCo manager, 1993

This was a common sentiment at FinCo and BrokerCo as well.

Elusive reuse

None of the four case sites reported reuse consistent with the OO vision, in which developers deliver systems primarily by assembling existing components. When reuse did occur, it was at the most basic levels (string classes, containers, time and date) or in the least leverageable forms (porting code, salvaging code during a system rewrite). Although these forms can be quite valuable, an organization can achieve them without adopting OO. Leading non-OO development environments—centered on RDBMS technology, for example—have for many years supplied built-in language functions and/or code generation options that cover much of the functionality in base classes (string manipulation and GUI screen painting, for example) and supporting tools like modeling and querying. Some of the best ways to port code or salvage code during a subsequent rewrite are to use nonproprietary tools and good architectural design concepts like layering.

Using an OO language on top of these approaches can certainly help—FinCo achieved its 85 percent code salvage this way—but the benefit is likely to be only incremental. OO has the most potential for increased reuse as a technological enabler for black-box component reuse and for reuse through specialization.[2]

Interestingly, each site—and sometimes projects within a site—had different reasons for the failure of the OO reuse vision. BankCo suspected that developers were using C++ "as a better C." EnergyCo had different barriers:

> We didn't have a librarian or someone who knew what all these methods are and what they do. I could be creating a method that does exactly the same thing somebody else's does...even though we have access to each other's code. We might call them different names and we might have a bit different way of doing it, but we're still doing the same thing.—EnergyCo developer, 1993

And BrokerCo had still different ideas:

> Because of time pressures there was no up-front analysis and so no reuse; we had a prototype and were told to code it....Reuse is good but comes at a cost. People who have sold OO talk about benefits but not the cost. Now we see the costs of getting to a true OO environment: doing a global business model, including future business plans.—BrokerCo developer, 1993

BrokerCo eventually made a strong commitment to

invest in reuse three years after initial OO adoption and created a new distributed systems development group with this in mind. Nevertheless, hopes for actually achieving significant reuse were viewed as "a lofty goal."

A FinCo manager and architect we interviewed in 1993 was the most philosophical about the difficulty of achieving reuse:

> It's hard to say what reuse really means. The objects that get reused tend to be generic, not domain specific—date, time, string classes. For domain specific, most objects get reused for major system revisions. Not many objects are common across systems, when they are, say, a customer object, they have different behaviors, for example, inquiries versus updates. At the architectural level there is pretty consistent reuse —containers, strings, date, and time. *Why not bundle all behaviors into a single common object?* By the time you define all the behaviors the business will have changed. If you have to describe a car in excruciating detail—the engine, the ashtray, the trunk —you'll never get finished. Developers are not interested in doing this all-encompassing design up front. *Why not build a base class when you first encounter an object, and add new behaviors to it?* Going back to the car analogy, if they did the engine but you need the trunk, why bother? At some point you could have a separate group to pull components together and build a car. Until then, people have real deliverables to be done tomorrow. Engineers are pragmatic. If it does a fair amount of the work for them they will use it. But if it has to be maintained and "revved" it's not worth it. It sounds like [reuse] is the right way to do it, but it does not happen in practice.

One of the most telling observations is the tendency for many respondents to believe, at least starting out, that high levels of reuse come more or less automatically—that achieving systematic reuse is primarily a technical problem that OO solves. This may partially explain the limited attention given to specific mechanisms for encouraging and enforcing reuse. Even after years of experience with OO development, developers commonly had narrowly focused explanations for why less reuse was occurring than expected.

These results appear typical of organizations that first become interested in reuse as a result of OO adoption, rather than seeing a reuse-based development process as a substantial innovation in its own right.[8] Although reuse scholars have argued for years that achieving systematic reuse requires a host of organizational and process changes that go beyond technology,[9] it's not realistic to expect early adopters to know everything that might pertain to a technology they are adopting. Part of the burden of being an early adopter is the lack of understanding about the nature of the innovation being adopted, and the managerial practices required to achieve the desired results.

LESSONS LEARNED

We synthesized the experiences of the case sites into four key lessons, incorporating the wisdom of others in the OO and reuse fields where it was consistent with our case-study observations. The lessons are specific enough to guide organizations considering immediate OO adoption, but we also believe most can be extended to apply to other software process innovations (new languages, database technologies, methodologies, and so on) that make similarly radical demands on would-be adopters.

We synthesized the experiences of the case sites into four key lessons, incorporating other wisdom in the OO and reuse fields where it was consistent with our case-study observations.

Invest in organizational learning

Sooner or later, most adopting organizations will have to educate new hires or retrain existing staff in the ways of OO development. Some estimates say about a year is required for an experienced procedural programmer to make the transition to OO; becoming an object designer or architect takes considerably longer.[10]

EnergyCo's approach to reskilling existing staff was exemplary. Team members, all with fairly standard mainframe skills starting out, were given six full weeks of training up front. OO novices first worked on the development of internal tools, which served as an extended training ground. The company then hired mentors to guide and review the team's efforts, thus providing an ongoing mechanism to facilitate training and experiential learning.

Not all companies have the calendar time or resources to bring existing staff up the OO learning curve. BankCo chose to get around this initially by adopting a simpler technology variant. FinCo hired all OO expertise externally. Each strategy has its own pitfalls, however. Adopting a simpler technology variant is cheaper and less risky, but it is less flexible in the kinds of systems capabilities you can develop and is therefore harder to scale up to the larger systems that support core business processes. BankCo, even after proclaiming the virtues of KappaPC on their pilot project, felt compelled to choose the far more complex (and low level) C++ for their OO production development. Even so, adopting a variant can be a good strategy in some circumstances. Organizations that feel they can develop needed systems adequately with existing technologies, but are concerned about keeping current with new technologies, might consider using simpler variants as a niche tool until the technology matures. Ultimately, if the technology does succeed and become dominant, even full-function technology variants will have become much simpler to use.

The second strategy—relying on externally hired personnel—may well be a necessary condition for the technical success of initial projects with reasonable size and

Commercial technologies like OO languages are incomplete and must be combined with other products, services, and expertise to be useful.

complexity. However, because qualified staff are not always available or easy to identify when they are, not every organization can use this strategy. There is a greater risk of invoking widespread skepticism and resentment among existing staff, and hence the transition may be more challenging in the long run: As a BrokerCo project manager put it, "You need an insider to spread the gospel; we're very much a group of outsiders."

Develop a complete architecture

By "architecture" we include not only tangible products (languages, databases, middleware, CASE tools, class libraries, library managers) but also the organizational processes, structures, methodologies, and guidelines needed to effectively use those products in significant projects. An OO language is of very little value to the typical IT organization unless it is bundled with developers who know how to use the language, a compatible OO methodology, a workable approach to database access, and so on. As a BrokerCo project manager put it in a 1993 interview, "We had to rely on a mishmash of tools; we had an OO analysis model. How do you take that and get a database design?"

Thus, at least initially, commercial technologies like OO languages are incomplete[11] and must be combined with other products, services, and expertise to be useful. Eventually the market will begin to coalesce, key products will mature and be enhanced, and nearly complete and robust products and services will become available. But we're not there yet; no current off-the-shelf product provides a complete solution to a variety of software development problems. Organizations can acquire most of the individual pieces of a complete OO architecture off the shelf, although they must then integrate them into a coherent whole. But for the case-study adopters, many important components (automated object-modeling tools, production-quality class libraries for GUI development) simply were not available. This is part of the large cost adopters incur when they leave a large, mature network (like a third-generation environment), and join a small, immature one (like OO). Ongoing problems at EnergyCo with the link between Smalltalk and DB2 are a good illustration.

> Some major architectural pieces were missing. Our database connection was inadequate. We were using an open gateway with dynamic SQL to map objects onto relational structures...It is really difficult to do any functional kind of prototype when it takes eight minutes to retrieve something for users.—EnergyCo developer, 1993

BrokerCo also suffered costly (though not project-threatening) gaps. Most notably, they realized too late that they needed a formal OO analysis phase to develop systems generic enough to support significant levels of subsequent reuse.

On the basis of our observations, we have identified four techniques that can lower the risk of an architectural collapse and make the development of a complete OO architecture more manageable.

Hire an OO architect. A qualified architect possesses knowledge and skill in OO design principles, methodologies, and project management practices; alternative application architectures; the specifics of the technology set the organization intends to adopt; and the distinct elements of the problem domains in which systems are to be developed.

Architects also serve as the top layer in the three-level structure commonly recommended for OO development teams (architect/class designer/OO programmer), so they should also have leadership and communication skills. The case-study sites that had the smoothest transition either had an experienced architect on staff (FinCo), or borrowed one for an extended period (BrokerCo).

An architect ensures that the blueprint on which all or subsequent efforts will depend is sound. At EnergyCo, the inability to recognize architectural inadequacies twice led to retrenchments. As an EnergyCo developer put it in a 1993 interview, "We had a brand-new technology, a brand-new modeling technique, and no single point of understanding for the whole thing."

Many roles other than architect can be developed internally, but an architect must be hired from outside. It is not feasible for existing staff to acquire the required knowledge to be an effective architect without years of hands-on experience in a variety of settings.[10]

Simplify architectural demands. Organizations should defer the development of large, complex, or mission-critical applications, if possible, and keep to well-understood application domains that play into OO's strengths. We realize this advice can be difficult to follow; some of the most common reasons for early adoption are the desire to avoid building large new systems with yesterday's technology (EnergyCo) and the desire to develop new kinds of systems with complexities thought to exceed the capabilities of existing tools (FinCo and BrokerCo). Nevertheless, we believe organizations must balance these immediate motivations with more fundamental long-term objectives, such as improving the odds of successfully assimilating a technology that could benefit many future projects.

Limit initial development. Organizations should confine initial developments to niche areas defined narrowly enough to make assembling an adequate architecture off the shelf a more feasible undertaking. They can then move from niche to niche, letting the competencies and assets acquired in initial niches contribute to success in subsequent ones. Object technology could be used for only early portions of the life cycle, selected application components (like the GUI), selected application types, or selected organi-

zational purposes (like information retrieval). Over time, the organization can merge islands of competence and tools into a more generalized architecture. Of course, any of these niche approaches initially curtails the benefits of OO, but not as much as overly ambitious undertakings that ultimately fail.

Find proven architectural examples. Organizations can aggressively network with other companies to find existing examples of complete architectures that have demonstrated success for problems of the same type and scale as the intended application environment. An extreme example is to attempt to locate and acquire a complete system similar to the intended application—source code and all—as a working reference model and design template.[12] Granted, the probability of locating such a system may be low, but if successful, the payoff can be substantial.

Develop missing components. These components would be developed with the understanding that, sooner or later, they are likely to be replaced with commercial solutions. This approach is not quite as onerous as it may first appear, because these kinds of investments often have side benefits. The development of an internal process and tools is a comparatively safe way for novices to learn about object technology. Also, custom tools can be tailored to the needs of the local environment. At a time when an organization is already undergoing considerable pains to adapt to generic commercial tools, it can be advantageous to have some areas in which the burden of adaptation is placed on the tool.

View reuse as separate

Better support for a building-block approach to software development has been promoted as perhaps the single most important benefit of object technology.[10] And for good reason: systematic reuse of this sort, unlike most advances in software development practice, may well be able to deliver the long-sought-after 10x increase in programmer productivity—at least under the right circumstances. But although OO provides some key technological enablers for systematic reuse, reuse consistent with this vision is rarely achieved:[9]

> I have not experienced a great amount of reuse, and I don't know anyone who has. It's still too early to tell how effective reuse can be...It requires a discipline and a support infrastructure. Putting all that in place requires a significant investment and a culture change. That takes a long time.—BrokerCo director, 1995

> OO is a great tool. But bringing it into a business is hard. It's very hard to get reuse in practice—especially with an extremely large group. We have a fairly small team, and it's hard enough to get reuse and get everyone on the same page.—EnergyCo developer, 1993

> I think it's probably still too early to answer the reuse question for business objects. Is there a customer object at [FinCo] that we have reused under a number of applications or even an account object? Nope, not really. Not nearly at that level.—FinCo executive, 1993

The addition of technological enablers such as OO does nothing to remove other reuse barriers. In fact, an emerging consensus in the reuse community is that the primary reuse barriers are nontechnical:[8,13,14] a lack of incentive to build for reuse; no mature processes, methods, or tools for managing a large-scale reuse environment; and disagreement about what kinds of reuse are most promising (large- or fine-grained?) or about the best technical architectures for achieving reuse (pervasive or minimal use of inheritance?).

Our case studies illustrate a number of common pitfalls to avoid when trying to establish systematic reuse: allowing project expediency to take priority over building for reuse; skipping OO analysis and modeling; operating without a standardized reuse process; and failing to provide some kind of library function. In addition, a growing body of literature contains considerable positive wisdom on how to establish systematic reuse, much of it based on the experiences of leading companies with vigorous reuse programs.[8,13] Although major investments are required to follow the advice in these sources, they need not be made all at once. Bertrand Meyer has argued, for example, that organizations should concentrate on becoming good reuse consumers before they try to produce reusable components.[2] Martin Griss and Marty Wosser describe an incremental approach to establishing reuse that has worked well at Hewlett-Packard.[15]

Organizations adopting OO because they believe it is an easy path to systematic reuse must ask themselves: Can our organization accommodate two major process innovations at the same time? If we had to choose between OO and reuse—at least for the next several years—which would we pursue more vigorously?

An organization may decide to pursue systematic reuse as a primary innovation with OO as a potential enabler, or it may choose to adopt OO and view reuse as a "nice to have but not essential" element. Many of OO's benefits—a more graceful paradigm for building GUIs, better support for multimedia and other applications that require complex data types, and better support for system interoperability—are not predicated on systematic reuse. For some organizations, these kinds of more incremental benefits may be sufficient to warrant OO adoption.

Sell adoption as process technology R&D

Our case studies show that adopting a new technology like OO is rarely predictable and rarely has a clear and easily achieved result.

Our case studies show that adopting a new technology like OO is rarely predictable and rarely has a clear and easily achieved result.

Table 6. Advantages and disadvantages of early OO adoption.

Advantages	Disadvantages
Enables new kinds of systems that exceed the capabilities of existing technologies	Higher risk of adopting a technology that never develops a large following (stranding)
Avoids continuing investment in new systems built around possibly soon-to-be obsolete technologies (tomorrow's legacy systems)	Higher risk of having the approach congeal around a patched-together technical and process architecture
Moves general benefits (such as development productivity) of using new technology forward in time	Much higher adoption and adjustment costs (technology evaluation, training, conversions, learning curve, and so on)
Grappling with the complexities of early vintages leads to a deeper understanding of the technology	Can't benefit from accumulated wisdom about the technology, so risk of initial adoption failure and technology stigmatization is higher
Use of the latest technologies makes it easier to attract and retain talented staff, contributes to innovative capabilities	Net loss of knowledge (trained personnel) across company borders
Recognized lead users receive positive press, and can impact direction of technology as it develops	May be badly positioned to adopt a new and better class of technology about to emerge

clear and easily achieved result. Many pitfalls await the early adopter. As Table 6 shows, the advantages of early adoption can easily be offset by the disadvantages. Yet when introduced, innovations like OO are often promoted as something anyone can and should adopt to solve all problems in all situations. This sets management up with unrealistic expectations. What is actually adequate or even excellent progress in adoption and assimilation is often discounted or even discarded.

To help avoid setting up unrealistic expectations, adopters must counter the view among many senior executives that software process technologies can be inserted into an organization in a turnkey fashion. One way is to educate executives on the parallels between adopting new process technologies and other learning-intensive activities with uncertain payoffs, such as new product development or R&D. In this view, the early adoption of OO is a multiyear R&D effort culminating in a staff, process, and technology infrastructure that can produce industrial-strength applications.

Of course, production applications must be developed along the way; they are essential to the learning process and to sustaining interest and funding. Nevertheless, we believe that these projects should be viewed not just as an end in themselves, but also as the means by which effective processes and an infrastructure are instituted, and they must be judged accordingly.

If an organization does not have the time, money, or expertise for R&D in process technology, it should delay OO adoption until the technology has matured and knowledge barriers have been lowered—both natural products of technology evolution.[6]

Object orientation and systematic reuse are prominent examples of attempts to improve the practice of software development through radical process innovation. Organizations contemplating adoption must balance the benefits of early adoption against the costs and risks we describe, and then develop an appropriate innovation strategy. The tactics we recommend to promote success as an early adopter require considerable investments, and we realize that not all organizations have the resources or inclination to make such investments. For these, our advice is simple: Wait. Over time, technologies destined for broad acceptance inevitably get simpler and easier to use; standards coalesce; complementary tools emerge that make it easier to assemble a complete architecture off the shelf; training gets better, cheaper, and more readily available; the supply of professionals already proficient in the technology increases; and wisdom accumulates in the industry at large about how and where to best apply the technology. Even among organizations that can afford substantial investments in process innovation, the benefits of waiting can often be compelling. ❖

Acknowledgments

We thank the managers and developers at the case sites who gave their time so generously to support this research. We also thank the MIT Center for Information Systems Research for their financial support. We especially thank Jack Rockart for his encouragement and support of this work.

References

1. B. Cox, "Planning the Software Industrial Revolution," *IEEE Software*, Nov. 1990, pp. 25-33.
2. B. Meyer, *Object Success*, Prentice Hall, Englewood Cliffs, N.J., 1995.
3. R. Fichman and C. Kemerer, "Adoption of Software Engineering Process Innovations: The Case of Object Orientation," *Sloan Management Review*, No. 2, 1993, pp. 7-22.
4. R. Fichman and C. Kemerer, "Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique," *Computer*, Oct. 1992, pp. 22-39.
5. P. Attewell, "Technology Diffusion and Organizational Learning: The Case of Business Computing," *Organization Science*, No. 1, 1992, pp. 1-19.

6. R. Fichman and C. Kemerer, "The Assimilation of Software Process Innovations: An Organizational Learning Perspective," MIT Center for Information Systems Research WP 281, *Management Science*, to appear.
7. W. Arthur, "Competing Technologies: An Overview," in *Technical Change and Economic Theory*, G. Dosi, ed., Pinter Pub., London, 1988.
8. M. Pittman, "Lessons Learned in Managing Object-Oriented Development," *IEEE Software*, Jan. 1993, pp. 43-53.
9. M. Griss, "Software Reuse: From Library to Factory," *IBM Systems J.*, No. 4, 1993, pp. 548-566.
10. T. Love, *Object Lessons*, SIGS Books, New York, 1992.
11. G. Moore, *Crossing the Chasm*, HarperBusiness, New York, 1992.
12. J. Hofman and J. Rockart, "Application Templates: Faster, Better, and Cheaper Systems," *Sloan Management Review*, No. 1, 1994, pp. 49-60.
13. W. Frakes and S. Isoda, "Success Factors of Systematic Reuse," *IEEE Software*, Sept. 1994, pp. 14-19.
14. H. Mili et al., "Reusing Software: Issues and Research Directions," *IEEE Trans. Software Eng.*, June 1995, pp. 528-562.
15. M. Griss and M. Wosser, "Making Reuse Work at Hewlett-Packard," *IEEE Software*, Jan. 1995, pp. 105-107.

Robert G. Fichman is an assistant professor at the Wallace E. Carroll School of Management at Boston College. His research interests include the management of technical innovation and software engineering management. He received a BSE and an MSE from the University of Michigan, and a PhD in information technologies from the Massachusetts Institute of Technology's Sloan School of Management.

Chris F. Kemerer is the David M. Roderick chair in information systems at the University of Pittsburgh. His research interests include management and measurement issues in information systems and software engineering. He received a BS in economics and decision sciences from the Wharton School at the University of Pennsylvania and a PhD in systems sciences from Carnegie Mellon University.

Contact Fichman at Boston College, 354D Fulton Hall, Chestnut Hill, MA 02167-3808; fichman@bc.edu. or Kemerer at the University of Pittsburgh, 278a Mervis Hall, Pittsburgh, PA 15260; ckemerer@katz.business.pitt.edu.