The impact of object-oriented technology on software quality: Three case histories

by N. P. Capper
R. J. Colgate
J. C. Hunter
M. F. James

Techniques to obtain software quality are examined from the experiences of three very different object-oriented projects carried out by IBM Information Solutions Limited in 1991 and 1992. Object-oriented programming systems are sold on the promise of improved productivity from object reuse and a high level of code modularity. Yet it is precisely these aspects that also lead to their greatest benefit, namely improved software quality. In this paper, lessons learned from the three projects are described and compared, indicating approaches to consider in using object-oriented technology.

Three very different object-oriented projects were carried out by IBM Information Solutions Limited (ISL)[1] in 1991 and 1992. They are (1) a huge new business application that included the development of an object-oriented user interface within a "traditional" host-based development shop, (2) a commercial data processing development project to provide an advanced graphical user interface to an existing national marketing database, and (3) a technical systems software project to "port" an object-oriented programming system from a Multiple Virtual Storage Time Sharing Option (MVS TSO) platform to a Customer Information Control System/Enterprise Systems Architecture (CICS/ESA*) platform and to provide additional base business objects.

When development started in January 1991, plenty of theoretical advice was available on topics such as object-oriented design, graphical user interface design, iterative development patterns, end-to-end design, and so on. However, there was very little practical experience to draw on. This paper presents some of the quality lessons learned as theory met reality head-on.

Broad aspects of quality are contrasted. In addition to code quality, the direct and indirect effects of object orientation on conformance to user requirements, usability, maintainability, and performance are illustrated with the use of statistical and anecdotal evidence. The issue of code metrics in a reuse environment is briefly discussed, as are unexpected, general conclusions related to the effect of team experience levels on quality results.

Finally, conclusions are presented which show that our experiences in the three very different object-oriented projects confirm that this technology produces immediate benefits in many aspects of software quality and productivity. The paper outlines the ways in which ISL intends to maximize these benefits in future projects and gives some practical recommendations on planning and running an object-oriented development project.

Although object-oriented programming has been in existence for over 20 years, 1990 was the year in which it first really came into prominence in ISL's predominantly commercial data processing development arena. It occurred in two contexts.

The objective of object-oriented design is to mirror real-world objects.

First, an ISL task force investigating the relevance of the "new world"[2] of graphical user interfaces (GUIs) for business applications of IBM United Kingdom concluded that these GUIs would provide enormous benefits to end-user productivity but that they would come at a price—one large component being the increased development effort needed to produce these applications using current design techniques and procedural languages. The task force recommended that object-oriented technology be investigated to see if the productivity improvements claimed by the technical programming world would also be sufficient to give the increase in productivity necessary for commercial GUI applications.

The second context was in ISL's computer-integrated manufacturing (CIM) mission where direction was given to use an object-oriented programming system: the ProductManager*—Application Services Manager (ASM). Part of the mission was to port ASM run-time components from a TSO to a CICS/ESA platform.

Aspects of object-oriented technology

Object-oriented technology encompasses not only object-oriented programming systems (OOPS) but also other object-oriented aspects such as user interfaces (advanced workplace GUIs), analysis, design, and database management systems. Lastly, using OOPS facilitates an iterative style of development rather than the traditional "waterfall" approach. Our experience so far includes the object-oriented user interfaces, design, and programming systems and iterative development.

In this section, a brief description of the concepts of object-oriented technology is presented. Then each of these aspects is described, including a tabular overview of the object-oriented technology used by the three projects.

Conceptually, the object-oriented world view is that of a collection of interacting objects, each with a time-varying status expressed in terms of data attributes and each with behavior expressed as responses to interactions with other objects. Each object is an instance of a particular *class* (for example, bank account) whose behavior is expressed in terms of *methods* (that is, function), each triggered by a *message* (for example, debit account). Classes can *inherit* data attributes and methods from other, more general, classes (for example, savings account inherits from bank account). The data of an object cannot be normally accessed except via messaging, which is known as *data encapsulation*.

Object-oriented user interfaces are a form of graphical user interface in which icons represent real-world business objects. The user instigates system actions by direct manipulation of the icons. For example, a product item can be added to a customer order by selecting the icon representing the item and "dragging and dropping" it onto the icon representing the order. It is claimed by some people in the industry that such interfaces have a closer correspondence to the end user's mental model of the application than conventional GUIs (where icons represent application functions) and are hence easier to learn and are more efficient and accurate in use.

The object-oriented user interface used was IBM's Systems Application Architecture* (SAA*) Common User Access* Workplace Model, known as CUA'91.[3]

The objective of object-oriented design is to mirror real-world objects. Similar attributes or behavior are factored out into higher abstract classes of objects.

In the industry, there are many object-oriented analysis and design methods described in books and taught in seminars. The existing (prior to object-oriented technology) analysis/design method in ISL was the Business Systems Development Method.[4] It is a data-driven structured method featuring the decomposition of business processes and their mapping against a data model in a modified entity-relationship-attribute format. At the beginning of 1991, there were two debates: First, should the Business Systems Development Method (BSDM) be replaced by a "pure" object-oriented method that starts off by analyzing the business in terms of "objects" rather than "data entities"? Second, if it is assumed that BSDM is retained, how far would it be possible to algorithmically derive an OOPS design from a BSDM business model?

It was decided to test the second approach, and a draft method of linking BSDM to object-oriented design was devised. This approach was intended to be piloted by the relevant projects along with a "user centered design" method (starting with "user task analysis") for GUIs described in IBM's CUA* manual on user interface design.[5]

Object-oriented programming systems implement the object-oriented concepts described earlier by structuring an application into:

1. Objects that each encapsulate data with the methods that operate on it.
2. Messages sent from one object to another in order to trigger methods of the receiving object. The same message type can apply to more than one class and is known as *polymorphism*.
3. Object class hierarchies that specify the inheritance of methods and data definitions from classes higher up in the hierarchy.

There are three aspects to OOPS:

1. The object-oriented language itself. Features that differentiate languages include:
   - Dynamic versus static binding, that is, whether references to other objects are resolved at run time or earlier.
   - Single versus multiple inheritance, that is, whether a class can inherit from one, or from more than one, other class.
2. Common class libraries for the language. They consist of common classes for reuse by developers. In theory, there are two types: technical (for example, classes for implementing GUI components) and business (that is, classes representing business entities). However, at present, only technical class libraries are generally available.
3. Tools. They include:
   - Compilers—varying in type from those that are automatically invoked or instantaneous (incremental) to those that are explicitly invoked or batch
   - Class browsers—for displaying the structure of an application and for searching through class libraries (for example, for reuse)
   - Graphical user interface builders—for defining windows by direct manipulation using a palette of visual control objects
   - Debuggers—for error diagnosis and correction
   - Library managers—for integrating the work of teams of programmers

Table 1 lists these tools for the languages used by the three ISL projects. Note that C is not an object-oriented language; however, Operating System/2* (OS/2*) Presentation Manager* (OS/2 PM) embodies some object-oriented facilities (messaging and function inheritance between graphical windows). ENFIN is a commercially-available, Smalltalk-like object-oriented development environment. SEDL++ is an IBM internal language used for developing CIM business application packages.

The iterative development approach differs from the conventional "waterfall" development approach as follows. The latter splits development into a number of phases, for example:

1. Detailed business modeling—data and process
2. Requirements definition
3. External design, including identification of human tasks and the definition of "human-computer" flows such as screen layouts
4. Internal design
5. Build
6. System test
7. Pilot/user acceptance test

Table 1 Languages used by the three projects

	C and OS/2 Presentation Manager (PM) APIs	ENFIN	SEDL++
Development environment	No	ENFIN/2**	ProductManager—Application Services Manager
Features:			
• Binding	Inheritance†: Static Messaging†: Dynamic	Dynamic	Inheritance: Static Messaging: Dynamic
• Inheritance	Single†	Single	Multiple
Class library	Presentation Manager	Built-in	Built-in
Tools:			
• Compilation	IBM C/2: Explicit, batch	Built-in: Automatic, incremental	IBM C/370 (via generated C): Explicit
• Browser	No	Built-in	No
• GUI builder	No	Built-in	No
• Debugger	No	Built-in	No
• Library manager	Conventional—LAN	No	Conventional—mainframe

†Applies to Presentation Manager only

Each phase must be complete and quality-assured before the next phase is started since there is an underlying assumption that changes to the application become progressively more expensive in succeeding phases.

However, application enablers (such as OOPS) that facilitate the rapid building of prototypes negate this assumption. Their speed and flexibility enable users to experience the application (in particular, the user interface) early and allow changes to be made based on their feedback. OOPS, specifically, because of their high modularity (and hence, minimum coupling) offer the promise of maximum change with minimum side effects. These characteristics allow the iterative revisiting of earlier phases and also delivery of the application in small increments.

Overview of the development projects

The three ISL object-oriented projects examined in this paper are:

1. EOSE (Europe/Middle East/Africa Order and Supply Execution) is a centrally run hub application for scheduling and ensuring the fulfillment of IBM's European customer orders for the entire Europe/Middle East/Africa (EMEA) organization of IBM's World Trade subsidiary. It has been operational since October 1993. Although consisting mostly of automatic background transaction processing, a highly usable interface is required for handling conditions where user intervention is needed.
2. SRFE (Sales Representative Front End) is an IBM United Kingdom application aimed at enhancing sales representatives' effectiveness by providing an easy-to-use interface to a previously underutilized National Marketing Data Base and associated functions (for example, performing bulk mailing to customers).
3. CICS Base is a project that ported ProductManager—Application Services Manager from an MVS TSO to a CICS/ESA platform and provided additional base technical and business objects.

Table 2 shows the projects mapped against the various aspects of object-oriented technology that they used. The next three sections provide a detailed description of the business, technical, organizational, and development approach aspects of each project.

Table 2 Aspects of object-oriented technology used

	EOSE	SRFE	CICS Base
Graphical user interface	CUA workplace (CUA'91)	CUA workplace (CUA'91)	CUA graphical model Text subset (CUA'89, not OO)
Business model	BSDM (not OO)	Informal (not OO)	Not applicable
Design method	Informal OOD	Informal OOD	Informal OOD
Prototyping language	ENFIN/2	ENFIN/2	Not applicable
Production language	C/2 and PM (semi-OO)	ENFIN/2	SEDL++ C/370
Development approach	Waterfall → iterative	Iterative	Waterfall for each object, plus some iterative

EOSE

Project description. EOSE is a component of the IBM Fulfillment Systems. Although small in comparison to some system software developments, it is one of the largest single *business* application developments ever undertaken by IBM—requiring some 2.5 *person-centuries* for the work on the first release alone. EOSE is mostly nonconversational transaction-driven processing. However, it does include a 22 person-year client/server object-oriented user interface component. This component enables the occasional manual adjustment to be made to some of the automated processes.

At first sight, describing EOSE as an object-oriented development may seem a little strange to the purist; it is written in non-object-oriented languages—C and PL/I, and it uses a non-object-oriented database management system (DBMS), DATABASE 2* (DB2). However, EOSE demonstrates a *practical* solution to what is becoming a more and more common problem: how to introduce and integrate object-oriented development into an existing "traditional" development shop, and how to build production-strength line-of-business applications with an object-oriented user interface while no proven object-oriented DBMS is yet available. (The choice of C, rather than, say, C++ or Smalltalk as the production language for EOSE was made simply because of availability and time. The code is structured according to object-oriented principles, using "home-grown" inheritance techniques. This object-oriented structuring was successfully tested by a small pilot development that recoded a part of EOSE in C++ to check the feasibility of a migration.) The host server components of EOSE, although having a strictly procedural structure, have a very strong degree of data encapsulation, so a future migration to an object-oriented DBMS is not precluded when it becomes practicable. This marriage of the old (procedural/relational) and the new (object oriented) is sometimes referred to as "object centered" rather than object oriented, and represents a practical way to make the transition.

Technical environment. The host (server) components of EOSE run under MVS/ESA, with IMS/ESA TM as the transaction manager and DB2 as the database manager. The code is written in PL/I. The programmable workstation (PWS) or "client" components of EOSE run under OS/2 Presentation Manager. (There is no client database.) ENFIN/2 was used in the early stages of development as a prototyping tool, but the code was subsequently recoded in C for the production version.

People and organization. The EOSE project consists of five development teams, four working on mainframe-only components and one on the PWS user interface. All these run under the same management hierarchy and comprise over 90 people in all. A separate organization of management and professionals is responsible for the independent testing, delivery, and implementation of the components (including EOSE) developed by IBM in the United Kingdom (UK) and for the installation and execution of all the Fulfillment Systems components that run at the UK central site. They also include an independent business support team that represents the users.

Development approach. Many lessons were learned from the development of EOSE. From these lessons a practical approach has evolved for planning and running an object-oriented development project within a "traditional" programming shop and for ensuring the quality of the delivered product. This approach is presented later in this paper.

The design phase for EOSE started with a series of modeling workshops to establish the database designs and the process boundaries. The methodology followed was the Business System Development Method (BSDM). Unfortunately, at that time the decision to develop an object-oriented user interface had not been made, so the resulting database design took no account of user interface requirements. It was six months before this decision was made, and EOSE then needed a design method that would bridge the gap between the now-established BSDM data model and an object-oriented graphical user interface design. Many people were willing to offer advice, but the particular step from BSDM to an object-oriented design was always dismissed as "intuitive," "obvious," or "trivial."

As part of the decision process for "going object-oriented," a short pilot project was run for EOSE to try to build a graphical user interface on top of an existing database. The design method that seemed to be the most promising was Clive Gee's Methodology for Object Oriented Design (MOOD).[6] An attempt was made to use this method to capture the requirements for the user interface objects, but there was soon an overwhelming mountain of paper with which to contend. Although it did help a little in deciding what the fundamental user interface objects might be, it proved almost impossible to relate all the MOOD design information to the database designs. As a result, when the design phase for the user interface started, no formal method had been identified that was felt to meet the needs of EOSE. Instead, the EOSE project set about producing its own.

By far the largest part of producing EOSE was a traditional mainframe development project. The user interface development team, therefore, had another requirement for the development method that was followed. It was recognized that a successful object-oriented development needs to be organized as an iterative process, but it also had to fit with the well-established waterfall approach to which the EOSE project management was committed. Many important lessons were learned during the evolution of the EOSE development method. The result has proved to be a successful marriage of the old and the new, which enables high-quality object-oriented code to be developed in an iterative manner, while interlocking with the waterfall approach required by the rest of the project.

The design phase for EOSE started with a series of modeling workshops.

The transition from the data and process business model to the object-oriented user interface design was made by analyzing the tasks the users wished to perform, in the context of sample business scenarios. From this analysis a prototype user interface was designed and iteratively refined. The design information (prototype and supporting documentation) was then formally agreed upon with the users, and funds were committed for developing the application code. In parallel with this activity a small subset of the development team started designing the technical infrastructure that would be needed to support the application. This infrastructure was to provide common services such as communication, security, user-profiling, messages, and error-handling, as well as to define the standards and protocols to be used.

Like the design phase, the build phase went through a number of iterations. First, the class hierarchies were developed to provide a consistent base for the rest of the development and to maximize opportunities for reuse of common code. Then the business objects themselves were constructed from these classes, with each object typically being the responsibility of a single programmer/analyst. To ensure high quality, the code was developed incrementally, with each addition being thoroughly tested. Quality checks and code inspections were held, but only to check for consistency across objects and for conformance to standards. The continual testing demonstrated that the code functioned correctly. Completed client business objects were subjected to a full end-to-end test with the corresponding servers.

Finally, the complete application (client and server code) was handed over to an independent testing organization for verification against the business requirements. Two levels of independent testing were carried out. The first was by information system staff only, using automated tools to verify conformance to the documented system requirements. The second was by representatives of the user community, in a simulated "real-world" environment. This last testing step served both to validate that the application was acceptable to the users and to educate them in using the new business processes it provided.

SRFE

Project description. SRFE is an ISL New World Project that started in January 1991, with the first release becoming available in September 1991 as a pilot release supplied to five IBM locations and some 16 users. Since then, SRFE has been distributed to 12 IBM locations and some 300 users, and a further three releases have been made available.

Two major objectives drove the SRFE project. There was a business need to supply an easy-to-use front end to a large internal marketing database, the National Marketing Database (NMDB). This database holds comprehensive information on IBM's current and prospective customer set in the United Kingdom and is perceived as supplying a competitive advantage to the IBM sales force, providing they can use the data available. There was also a desire to provide a pilot application with a technical solution that encompassed object-oriented technology, client/server (PWS to host), and GUI concepts.

Technical environment. The target environment for the application was a Personal System/2* (PS/2*) running OS/2. The application was to run either on a single workstation or as a public application from a single local area network (LAN) server in order to support a shared disk and PWS policy. The database resides on a remote Application System/400* (AS/400*) in an IBM location near London. Communications between the PWS and the AS/400 are LU 6.2 Advanced Program-to-Program Communications (APPC) via local LAN, backbone LAN, and the IBM network.

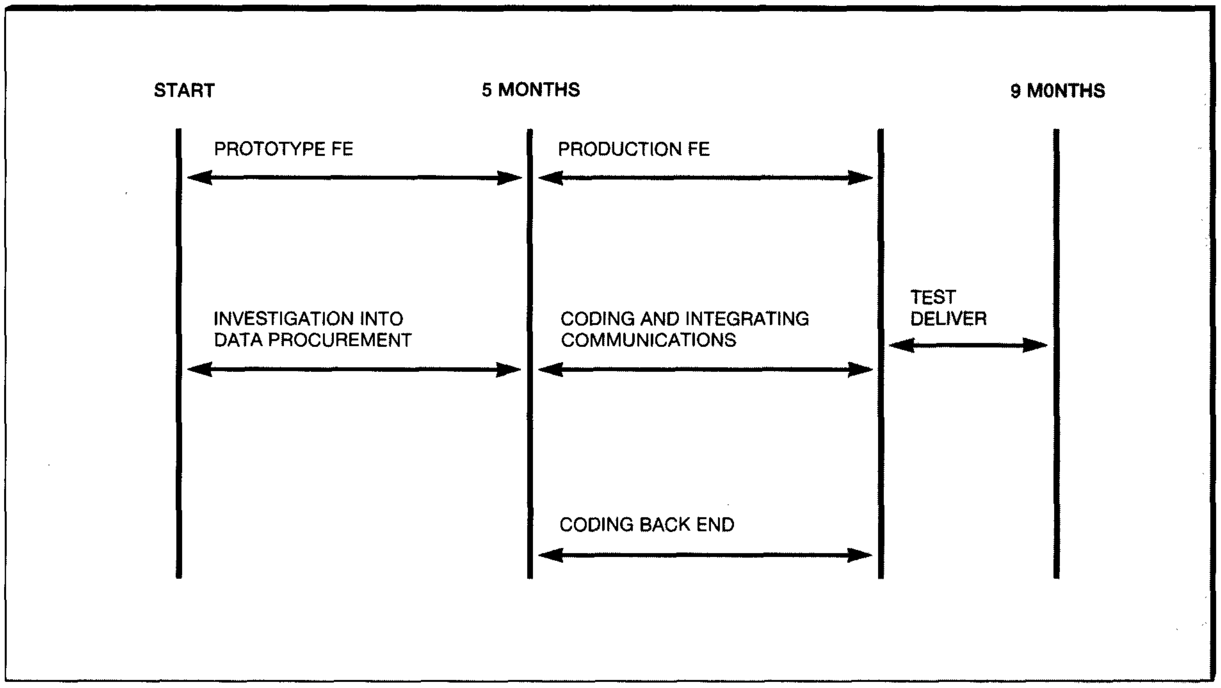
There were two development environments, AS/400 and PWS. The database access programs on the AS/400 were written in Report Program Generator (RPG), using the application development tool set (ADT). The PWS developers worked in a LAN environment, using PS/2s running OS/2.

AS/400 and PWS were the two development environments for SRFE.

The application was developed using ENFIN/2, a 4th-generation language (4GL) object-oriented development environment. ENFIN/2 is based on a Smalltalk kernel, but comes with its own class library, and in the version used in the SRFE development, its own language, which is like Smalltalk with some syntactical differences. The latest versions of ENFIN produce Smalltalk code and use the Smalltalk base classes with extra classes for graphical elements, communications protocols (APPC, high-level language application programming interface, or HLLAPI) and other functionality. ENFIN provides a complete development environment, including a GUI builder, APPC support, database support, class browser, debugger, class definition facilities, and much more.

The communications were LU 6.2 protocol-implemented with native APPC. The link with the ENFIN environment was via C dynamic link libraries (DLLs), and with the AS/400 programs via the AS/400 Intersystem Communications Facility (ICF) file. Networking was via a LAN with the development AS/400 attached directly for development purposes, or as in the target environment for test purposes.

Figure 1 SRFE project integration—pilot release



People and organization. The SRFE development team consisted of eight people, although not all of them were on the project for the full duration. There were a number of clearly defined roles in the team, generally shared between two or more team members. In certain roles the team had considerable previous experience, in particular with the AS/400 environment, including RPG coding, the NMDB database, and the existing application that provided services for this database. The business was also well understood, and there was considerable project management experience in terms of traditional project disciplines among team members.

However, none of the developers had previous object-oriented development experience, and a major inhibiting factor for the team was the lack of this knowledge and the difficulty of obtaining the right education at the right time. Also, none had experience with developing GUIs, with the IBM CUA standards, or with developing a cooperative processing solution using LU 6.2 and APPC. These considerations turned out to be less important when compared to the lack of previous object-oriented knowledge and the nonavailability of a skilled mentor.

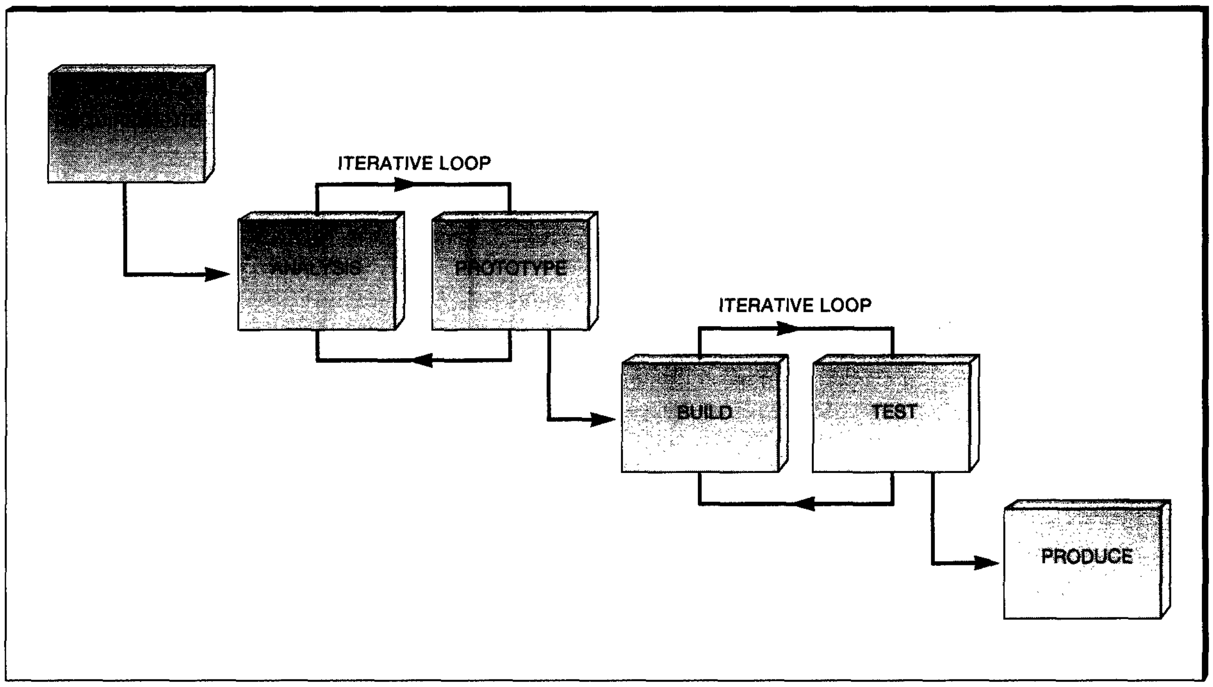
The project was divided into three discrete pieces of work. The object-oriented application itself on the PWS (the "front end"), the code to support it on the host (the "back end"), and the communications piece that linked the two. For the pilot release of SRFE these three pieces ran essentially in parallel, as shown in Figure 1.

Coordination between these subprojects relied on good interpersonal communication between team members. A small dedicated team working in close proximity was one of the (not specifically object-oriented) keys to success with SRFE.

Development approach. The host and communications pieces of this project were run along traditional development lines, and so will not be discussed in further detail.

For the pilot release of SRFE a phased iterative technique was used. The release followed a requirements gathering stage, an iterative analysis

Figure 2 Phased iterative development method for SRFE pilot release



and prototype phase, then an iterative design, code, and test phase, succeeded by the final produce phase. This technique is illustrated in Figure 2.

Requirements were gathered by the business analyst and project manager and given to the team as a set of functions for prototyping. A CUA'89 graphical interface was developed for the prototype, with all of the function contained in the view controller classes. (These classes control the operation of the windows.) This prototype then went through several iterations, with heavy end-user involvement, thus improving its utility and acceptability to the sales representative. A finalized prototype, running from OS/2 Database Manager* (DBM) tables, was delivered within five months. This prototype was given to a number of selected users for a trial period, and it also underwent a CUA review. After the review it was decided that a CUA'91 Workplace implementation was desirable for the production code, being potentially more intuitive and productive for the user.

An object-oriented analysis and design process initiated the design and code phase of the project one month later. No standard method was used for the analysis and design, but rather a synthesis of several existing methods.

From the prototype and with existing business knowledge, a full appreciation of the process and entities necessary for the business model was available. This fulfilled the part of the analysis usually taken up by data and process modeling or task analysis, or both. The business object model was derived from this basis using a CRC card method after Wirfs-Brock et al.[7] CRC cards (class, responsibility, collaboration) allow the designer to define the objects of their responsibilities in terms of state and behavior and the kinds of objects that will be collaborators. This process has several passes, the inheritance hierarchy starting flat and gradually deepening as "kind of" relationships are identified. With this method the onus is on defining the behavior of an object rather than in placing the data.

After the business object model was defined, a similar process was followed to identify the kinds of objects that would be needed to support the necessary business processes in addition to the business object model. For example, two basic business objects identified were contacts and establishments, but an important process was handling lists of these objects. Thus, additional list objects with appropriate behavior had to be defined.

Finally a high-level design was done, where implementation and language considerations were taken into account.

Along with the CRC card method, sender-receiver matrices were defined to provide more information on how objects were to interact. The user interface design was an extension of the prototype work already done.

Future experience confirmed this stage as critical for the application in terms of maintainability, both perfective and adaptive.

For coding purposes each business object or group of business objects (module) was assigned to a developer, who was then also responsible for developing the appropriate infrastructure and view classes.

As the function of the object groups was built up, it was tested by the developer—a significant feature of the incremental compiler found with Smalltalk-type languages. Consequently much of a particular object group had been "in use" for at least two weeks and tested several times before being handed over for the integration test. During the integration testing the business analyst was responsible for ensuring that each object group performed its responsibilities accurately.

All of this testing had a cumulative effect on the final user acceptance testing because by this point many of the base modules of the application had been "in use" for some months.

CICS Base

Project description. The CICS Base and Extensions project was really two projects in one. The first part was to "port" a large TSO-based object-oriented system (which was due to be released as a program product, i.e., the ProductManager Application Services Manager) onto a CICS/ESA platform. The stated objective was to change as few lines of code as possible, but also to take advantage of CICS facilities and provide some additional function specific to the CICS platform. The overall objective was to enable applications written using the "base" in TSO to be run on CICS/ESA with as little change to the applications as possible. The second part was to develop new extensions to the existing base that would also function on the new CICS base. The "base" or Application Services Manager (ASM) is an object-oriented system that provides applications with common functions such as screen-handling, security, file-handling, data-manipulation, etc., as well as an object-oriented framework in which to execute the application. Most of the ASM is written in an IBM internal object-oriented language known as SEDL++. The existing base was being developed in Atlanta, Georgia, along with all of the tools to support SEDL++. The project(s) described here were based in Portsmouth in the United Kingdom.

Technical environment. The target platform for the new system was CICS/ESA using DB2 and VSAM (virtual storage access method) files for data storage. (VSAM was used in the CICS version to implement a file system similar to MVS partitioned data sets and sequential data sets, which were used in the TSO version.) CICS/ESA facilities were used to implement asynchronous communications, and SAA Common Programming Interface for Communications (CPI-C) was utilized to implement functions to enable applications to communicate synchronously over LU 6.2 links.

The library management system (an IBM internal-use-only system designed for multisite development) ran on the virtual machine (VM) operating system as did all of the development tools. Each developer had a VM Conversational Monitor System (CMS) for editing code, accessing the library management system, and running the tools to generate C code. The C code was compiled on VM, and the object deck was stored in the library management system and "shadowed" onto the MVS/ESA system. The link-edits were then initiated on the VM system but executed on the MVS system. The resulting load modules remained on MVS only. As a result of this complex development environment, the time from discovering a program error to being able to test the corrected program (in unit test) was quite long (1 to 4 hours).

People and organization. The developers were split into small teams working on almost self-contained parts of the overall project. One team was given the tasks of converting the existing TSO base to operate on CICS/ESA and adding some new function specific to CICS/ESA. The other teams were given one extension each: namely calendar, currency conversion, work queue, or organizational unit. The teams varied between one and seven people plus a team leader or project leader, or both.

The teams consisted of a typical mixture of experienced and junior professionals, with the team or project leaders usually being the most experienced within a team. However, none of the team members or leaders (or managers) had experience with object-oriented design or programming. Also, the language and the tools to support the language were all new to everyone in the project. Because the original developers were based in Atlanta, it proved difficult to obtain the required levels of education. Most of the education was gained through videos, teleconferences, reading manuals, and just "having a go at it." Personnel from Atlanta attended most of the inspections and thus were able to give direction.

Development approach. The development teams were directed to use a waterfall approach to development, with inspections after each stage. The stages were product functional specification (PFS)—a definition of requirements, high-level design (HLD)—a detailed external design, and low-level design (LLD)—really the coding stage in the language used. With the (multisite) inspections taking place at each stage it discouraged iterative development (which is the way in which nearly everyone is told to approach object-oriented development). In fact, in some cases the team "broke the rules" and did some iterative development before producing an HLD or LLD document for inspection. It was risky because a lot of rework may have been required, but it also meant that the developer was confident that the solution worked and could answer any doubts in the minds of the inspectors.

The users (who in this case were application developers) were usually involved in the PFS inspections as this was the point at which their requirements were clarified and spelled out in detail. However, this approach did not apply to the conversion of the base to CICS/ESA. Here the requirement statement was to "make it work on CICS."

Often a complete system rebuild would be required for changes to the object database in the CICS Base project.

Consequently the real detail only came out during HLD, at which point the users were not interested as most of the detail did not affect their application code. The main focus of the inspections was therefore on implementation details.

Because the development tools were not stable at the time this product was being developed, the developers found the whole development process complex and frustrating. The process of debugging code was complicated because the debugging tools used were products that had been designed for debugging traditional applications. For example, it was often difficult to locate the offending object instance, let alone the line of code that was causing the problem. Debugging this system under CICS/ESA was challenging since only standard CICS-supplied tools were being used, which again were not designed with object orientation in mind.

Often a complete system rebuild would be required for changes to the object database, for example. Rebuilding was done less often over time, but true incremental compilation would have proved much faster and reduced development time a great deal. In some ways the need for frequent complete system rebuilds helped to improve quality by highlighting potential cross-system errors and conflicts early.

Non-object-oriented subsystem approach. One part of the CICS work, to build a file system to simulate the MVS file system, was written in C and not an object-oriented language. As this did not fit neatly into the object-oriented stages, a technical design instead of an HLD or LLD was produced. The technical design was a combination of HLD and LLD that went down to the detail of pseudocode. This technical design was inspected, but there were no formal inspections of the C code. At least two team members were involved in each C program in order to encourage readability and to spot performance problems, and so on. However, in hindsight, this code should have been subject to formal inspection, as some of it was badly commented. Nonetheless, the C code produced was the highest quality in terms of errors per KCSI (thousand changed source instructions) of all the CICS base code. It is believed that this quality was due to the skill and experience of the programmers involved rather than being technology-related, although the C developers did have the advantage of a stable language and compiler. It may also be a reflection on the number of source instructions that had to be written in C to produce the required function. So even though there were less errors per KCSI, the errors per function point, for example, may have been higher (function points were not calculated for this project).

Quality results comparison

Application quality is a multidimensional metric; its individual components are defined in the following subsection. Next comes the overall results matrix showing the effect of each object-oriented technology on each of these components. The overall results are then substantiated by the results for each of the three projects and a consolidation of the quality lessons learned. Finally, a practical development approach is presented, which evolved independently for SRFE and EOSE, based on their experiences. This approach combines traditional business modeling with object-oriented design and programming and is aimed at maximizing the quality benefits for future projects.

Quality metric definitions. The component metrics to be discussed are:

1. Code quality
2. Correctness
3. Usability
4. Adaptive maintainability
5. Perfective maintainability
6. Performance

Code quality. Code quality is the density of functional defects found in the application code. A defect is defined as nonconformance with the outline design agreed on with the application owner.

The units used in this paper are defects per thousand changed source instructions (KCSI), which is the total number of new and modified source instructions. Only defects that prove to be valid are counted. (Defect density can also be measured in terms of function points, an implementation-independent measure of the amount of application function.[8]) For a discussion of code metrics in a reuse environment and the rationale for the choice of units in this paper, please see the Appendix.

Code quality is measured at three points in development as well as after delivery. Because of differences in terminology in the industry, these points will be defined as follows within this paper:

1. Development test—Functional testing carried out by the developers themselves. It may consist of two phases known as "unit test" and "integration test"; however, the latter may be done in the next test stage.
2. Independent (IS) test—Rigorous, third-party testing by information services personnel. It may consist of two phases known as "integration test" and "system test."
3. User test—Testing carried out by representative potential users of the application. It is sometimes known as "user acceptance test."
4. Production—Post-delivery, real-life application use.

Correctness. Correctness is the degree of fit with the business objective of the application. It is measured by the number of project change requests (during development) and improvement requests (after delivery) that are not the result of unforeseeable business changes.

Usability. Usability is that aspect of application design that enables a user to understand and use an application easily. It is in itself a multidimensional metric with components of self-sufficiency, ease of learning, ease of use, consistency, user accuracy, and user attitude.

Adaptive maintainability. This metric is also known as flexibility or enhanceability. It is the relative effort needed to extend the functionality

Table 3 Impact of object-oriented technology on software quality

	Code Quality	Correctness	Usability	Adaptive Maintainability	Perfective Maintainability	Performance
OO user interface			++/−			−
OO design	++			++	++	−
OO programming system	++			++	++	−
Iterative development	++	++	++			+
Net benefit	++	++	++/−	++	++	−/+
Quality metric units	Defect density	Project change/ improvement requests	Self-sufficiency, ease of learning, ease of use, consistency, user accuracy, user attitude	Relative enhancement effort	Relative fix effort	Conformance to target

Key: +	Positive impact	Compared with:
++	Major positive impact	· Action-oriented user interface
−	Negative impact	· Structured design
− −	Major negative impact	· Procedural programming
		· Waterfall development

Table 4 EOSE quality metrics

Component	Thousand Changed Source Instructions	Cost (gross person-months)	Defects in Developer Test	Defects in Independent Test	Defects in User Test	Defects in Production
PWS development	109	236	129	122	31	0

of the application. This metric is only discussed qualitatively in this paper.

Perfective maintainability. This metric is the relative effort needed to diagnose and fix defects in the application. It is only discussed qualitatively in this paper.

Performance. Performance is conformance to the agreed-on application performance targets (for example, response time to the end user).

Cost of quality. In order to establish that the quality results were not achieved at the expense of productivity, this latter metric is also presented in the following sections. The unit of effort used is "person month" and consists of the direct gross developer effort, excluding project leadership and management costs.

Overall results. Table 3 summarizes the overall results regarding the impact of object-oriented technology on software quality as an impact matrix. The ratings were determined by consensus in a group discussion with a representative from each of the three projects.

Project results. The following subsections present in detail the experiences of the individual projects and the quality metrics they obtained.

EOSE project. Table 4 shows the sizes and costs of the object-oriented development in EOSE and the total defects recorded in the different testing phases. Table 5 shows the productivity (source instructions per month) and defect densities (defects per thousand changed source instructions) derived from these data. It also shows the equivalent data for the mainframe component of EOSE, which was a traditional non-object-oriented development. These data allow some interesting comparisons to be made. The data illustrate one of the major benefits of object-oriented development and of an iterative development process—a

Table 5 EOSE comparative quality results

Component	Productivity (source instructions per month)	Defects per KCSI in Developer Test	Defects per KCSI in Independent Test	Defects per KCSI in User Test	Defects per KCSI in Production
PWS development (OO)	462	1.2	1.1	0.3	0
The figures below can be compared with the equivalent data for the mainframe component of EOSE, which is a traditional non-OO development:					
Rest of EOSE (non-OO)	95	Data not available	3.7	1.8	0.03

dramatic difference in errors detected at all stages of testing. Even though the PWS development team faced new technology, new hardware, new software, new programming languages, and a new design and development approach, the defect rates measured in independent testing were only one-third of those encountered in the mainframe (non-object-oriented) components of EOSE, built by skilled teams using familiar technology. The defect rates measured in user test were only one-sixth of those in the non-object-oriented components, and there have been no defects reported at all in the first two months of production running of the object-oriented code. Although the development of the class hierarchy and the technical infrastructure took considerably longer than originally planned, once they were complete and tested, new functions could be added quickly and easily and to very high standards of quality, through the exploitation of inheritance.

The data also illustrate the greater overall productivity of the object-oriented development. Despite the apparently slow progress in the early stages of the project (several of the early tasks took up to five times the planned effort because of inexperience with object-oriented methods), overall the project was completed within 10 percent of the original estimate.

As summarized in Table 3, there are also tangible benefits for correctness and usability from following an iterative development approach. The use of early user interface prototyping, along with the close and frequent involvement of the users from the very earliest design stages, has resulted in a very low level of late design changes. The project team had a high level of confidence that the code being developed would meet the user requirements for functionality and usability. Just seven (minor) requests for design changes to the user interface were recorded during independent testing.

The benefits of increased productivity through the use of inheritance are becoming even more apparent in the follow-on project to EOSE Release 1.0. It is now possible to use C as the prototyping language. New EOSE objects and methods are quickly modeled using largely inherited function from the EOSE class library; only a few lines of new code are needed to provide, for example, a different field edit rule or a new server request for data.

In general, changes to the existing code, both enhancements and fixes, are also much easier to apply. The encapsulation of data and function allows the scope of the changes to be identified quickly, and there is thus a very low risk of accidentally introducing errors in other parts of the application. (Just occasionally the object-oriented structure can be a hindrance to analyzing a problem, rather than a benefit. If the problem is caused by incorrect messages passing between objects, and there are many objects involved in the processing of an event, it can be quite difficult identifying which particular object is the real culprit. Overall, however, an object-oriented structure is seen as a major benefit to maintainability, as reflected in Table 3.)

No problems have been reported against the performance of the EOSE user interface (though, it should be said, there were no particularly critical performance requirements since it is only intended for use infrequently).

SRFE project. The metrics in Table 6 and Table 7 are for two releases of SRFE, the pilot release and the last release that was known as the corporate release. The difference in the data indi-

Table 6 SRFE quality metrics

Component	Thousand Changed Source Instructions	Cost (gross person-months)	Defects in Independent Test	Change Requests in Independent Test	Defects in User Test	Change Requests in User Test
SRFE pilot release	19.4	72	41	6	9	4
SRFE corporate release	4.5	6	7	0	5	4

Table 7 SRFE comparative quality results

Component	Productivity (lines of code per month)	Defects per KCSI in Developer Test	Defects per KCSI in Independent Test	Defects per KCSI in User Test	Defects per KCSI in Production
SRFE pilot release	269	Not available	2.1	0.46	0
SRFE corporate release	367	Not available	1.6	1.1	0

cates the improved competence of the SRFE team in object-oriented development.

Data given are for the object-oriented front end only, and do not include the data for source instructions and errors occurring in the host or communications part of the code. The errors counted are those that were coding errors or errors in the business process. Cosmetic errors, such as having names on "buttons" that the user did not like, are excluded.

Code quality was enhanced by two aspects of the object-oriented development approach. The first was the use of inheritance: often new function was developed using existing classes, thus inheriting the quality of these super classes. Second, the iterative nature of the development had a significant effect on the quality of the code. A substantial part of a particular object group had been "in use" for at least two weeks and tested several times before being handed over for integration test. Hence, very few code errors were found in the integration test. These benefits are reflected in Table 3.

Also, before going to the users for the user acceptance test many of the base modules of the application had been "in use" for some months, thus giving high-quality code to the users for acceptance testing. It will be noted that defects found in the user test increased between the pilot release and the corporate release. For the corporate release there was a policy decision to reduce the amount of independent testing to a nominal level to reduce the expense of the project. This decision was made in the light of previous code quality results. The result of this decision manifested in the user test and was a salutary lesson that early independent testing is still necessary for object-oriented projects.

To go further, no errors relating to the base code have been raised on SRFE in production.

The iterative nature of the development also contributed significantly to the fact that SRFE was fit for its purpose when delivered. Although not specifically related to object orientation, facts such as heavy user involvement at the prototype stage and an ongoing user involvement through the code and test stage meant that at the user acceptance test few problems arose with the process being followed.

The analysis-prototyping iterative loop supplied several benefits. It ensured that the application adequately modeled the business and also that the process flow was both correct and fitted the user's intuitive mental model. The result was an application that was not only useful but also well accepted by the sales personnel at which it was aimed. The benefits obtained from following an iterative development life cycle are summarized in Table 3.

Table 8 CICS Base quality metrics

Component	Thousand Changed Source Instructions	Cost (gross person-months)	Defects in Developer Test	Defects in Independent Test	Defects in User Test	Defects in Production
Calendar	3.3	41	128	20	N/A	0
Currency	1.4	9	50	3	N/A	0
Work queue	6.8	70	141	12	N/A	0
Organizational unit	5.0	60	66	17	N/A	0
CICS port	17.0	96	357	13	N/A	1

Table 9 CICS Base comparative quality results

Component	Productivity (lines of code per month)	Defects per KCSI in Developer Test	Defects per KCSI in Independent Test	Defects per KCSI in User Test	Defects per KCSI in Production
Calendar	79	39.3	6.1	N/A	0
Currency	150	37.0	2.4	N/A	0
Work queue	97	20.7	1.8	N/A	0
Organizational unit	83	13.2	3.4	N/A	0
CICS port	178	21.0	0.8	N/A	0.1

The later releases of SRFE confirmed that the analysis and design phases are critical stages for the application in terms of maintainability, both perfective and adaptive. A well-modularized, stable class structure means that any errors that do occur in the code are localized and easily traceable and that extending the base code is facilitated because potential impacts on existing code are easy to identify. These improvements are identified in Table 3.

CICS Base project. Throughout the writing of this paper, and indeed throughout the latter stages of the project, attempts have been made to pinpoint why such good quality results (compared with earlier projects of this nature) were achieved. The truth is that, in reality, just one factor cannot be identified. It would be ideal to point out one thing that was done and tell the world that this is how to achieve good quality, but instead it appears to be a combination of factors that at times conflict. It will become apparent that, unlike some theories, practical experiences rarely fit into neat little labeled packages that behave consistently.

Table 8 and Table 9 show the results for this project. Calendar, currency, work queue, and organizational unit are new reusable business classes that were developed. CICS port is the porting of ProductManager Application Services Manager from TSO to CICS/ESA.

Note that the column labeled Defects in Developer Test in Table 8 shows all defects recorded by the developer testing against a test plan. The Defects in Independent Test column shows all programming defects found by an independent tester. The Defects in Production column shows defects found to date by the first CIM application developers using the classes and ported code in their development. The applications they have produced have yet to go into production with business end users.

Because this development was a new type for the organization, there are no data for comparison. However, the perception is that the "independent test" data represent an order of magnitude improvement versus large business systems produced in the past in the same group. The "production" data are from the products used by application developers over a three-to-six-month period. Other large systems produced by the same group in the past have had hundreds of (minor) errors raised during the first three-to-six months.

The only other problems raised have come from the realization that the users would have liked to have specified additional requirements, which have only come about from their use of the system. A prototype may have highlighted some of these requirements, but they effectively had a fully working system on a different platform, and the new requirements relate to the new platform. Producing a sensible prototype on the new platform would result in almost the same project as producing the final system, so it could not be justified.

As shown in Table 3, the usability of the user interface varies between "major positive impact" and "negative impact." Because the user interface of the CICS Base was presented via host-based "dumb" terminals, the "object-action" interactions became tiresome at times. Later, some improvements were made to the user interface to make it more usable on dumb terminals; however, it was still slower to use than a traditional menu-based system. When the product is ported to workstation platforms, the benefits of the user interface design will become apparent.

Object orientation proved to be a major positive impact on adaptive maintainability as shown in Table 3. All of the extensions (calendar, currency conversion, organizational unit, and work queue) were written to be platform-independent by using the facilities of the base. Being independent meant that they were written and tested once, with just regression testing for other platforms. Adapting the behavior of the entire system is possible by changing a single object, and similarly, objects may be changed for different platforms to ensure that they behave consistently for their users.

Lessons learned

Many lessons were learned during the development of the three projects. The lessons described in the following subsections are probably the most important ones and include those that were most instrumental in the evolution of the development approach to which EOSE and SRFE, quite independently, converged (described in the next section). (Although not a part of this paper, international dual-site development was a big influence on the quality produced, both positive and negative, in the CICS Base project. It also had a negative impact on productivity.)

Prototyping. Prototype the design with real end users, or their empowered representatives, as early as possible in the development cycle to ensure that full requirements are gathered and are understood by both parties. Ideally the prototyping should be done during the modeling phase. Prototyping does not just mean building a fully functional user interface. In the very early stages, a prototype design may just be on pieces of paper to test out the results of a task analysis workshop. Later prototypes may evolve through static window designs, made with a screen painter or a graphical design tool, through working window flows written in a prototyping language, to complete business objects.

Object orientation proved to be a major positive impact on adaptive maintainability.

An iterative development life cycle can contribute significantly to increased code quality, code correctness and application usability and, in SRFE, to performance problems being spotted and fixed earlier. (There can be negative aspects to an iterative development life cycle: it can be difficult to manage; keeping track of progress is not easy; and regular status meetings and checkpoints are necessary.)

Business models and object-oriented design. Do not assume that there is a one-to-one correspondence between data entities identified during business data modeling and the objects that will be presented on the user interface. Although there will be a relationship between the two, there may well be a different level of granularity. For example, if most of the function being modeled is to be run as background processing (as is the case for EOSE), the resulting primary data entities needed to satisfy the business requirements may be finer-grained than the view objects presented to the on-line users. In contrast, if the application is mainly a real-time user interface, such as a game or a process simulation, much more detail may be presented to the on-line user than is needed for permanent storage in a database.

Infrastructure and tools. Identify all technical infrastructure requirements and finalize their design before designing the business functions. Include services such as client/server communications, error-handling, message-handling, security functions, and directory functions. Similarly, design all cross-object and common processing before starting individual object design. If a change is identified after business object development has started, the rework costs can be very high.

If at all possible, use a mature tool set for development.

The SRFE project used an integrated object-oriented development environment with built-in debugging tools. By contrast, the tools (and language) used in the CICS Base project were being developed at the same time. This tool development caused a lot of wasted developer time and was the cause of much frustration. The key lesson here is to either use tried and trusted tools or spend a lot of time making sure that the tools are of excellent quality and as intelligent as possible, and that they perform well. Every week spent on ensuring good tools will save months of developer time later.

Feasibility verification. If at all possible, use a mature tool set for the development, and have peer support (either inside or external to the team) for the analysis and design methods and the language that is being used. This type of support and tools gives confidence in the structure of the application being developed and in the environment in which it is being developed. Developing your own tools should be considered only as a last resort.

If it is necessary to use new languages or tools, it is essential to try them on a small development first, before tackling a major development project. For example, select one of the business objects of the product and develop it completely as a "fast path" project.

Learning curve. Allow adequate time for training and the "learning curve" when working in new or unfamiliar environments. With SRFE it was found that language considerations were not a problem—the real challenge was the change in paradigm from a procedural approach to an object-oriented approach. A mentor experienced in object orientation would have been invaluable in pointing the way forward at certain stages of the project. The improved quality figures between the pilot release and the corporate release of SRFE are partly attributable to the familiarity of the development team with both the development tool, ENFIN, and the object-oriented paradigm.

Also allow for a high percentage of project time being absorbed by "technical overheads," in addition to the normal project overheads of project control, reporting, and meetings. This time includes activities such as general technical support and mentoring, investigations into new techniques and technologies, problem and change request investigation, maintaining the technical environment, and developing common infrastructure code. Experience in the EOSE project indicates the total overheads can be up to 25 percent of the project cost.

Reuse. Code reuse results in excellent benefits in terms of productivity and code quality. However, developing the initial objects with reuse in mind takes time and effort and has to be scheduled for where it is appropriate. Later releases of SRFE benefited from reusing code developed earlier, with an attendant increase in both quality and productivity. In terms of productivity, with the pilot release the team produced 269 source instructions per person-month. With the corporate release this production had risen to 367 source instructions per person-month, a 36 percent increase. Although this increase was largely due to the absence of a learning curve, the reuse of objects from the first release was also a significant contributing factor. The CICS Base project reused approximately 85 percent of the original code.

Team size and work allocation. Object-oriented developments usually mean small teams, or at least encourage (and facilitate) small teams. Even so, it has been found that a small team can feel and behave just like a big team. Ensure that the team is able to make decisions for itself and that communications are adequate between team members. (Just because they all sit in the same area does not mean that they talk to each other!) However, these small teams must not work in isolation. The application design must be "end to end." It must contain full details of the graphical user interface, the client (PWS) to server (mainframe) interface, and the mainframe servers. Do not try to develop the client and server parts in isolation and bring them together for the first time at system test.

For coding purposes each business object or group of business objects was assigned to a developer.

Initially the EOSE PWS development team was established as a completely independent project. However, an early lesson learned was that there had to be close and continuous contact between this team and the four mainframe projects to which it provided services. As a result, the analyst/designers were made responsible for the end-to-end design of the user interface objects (that is, both client and server part). They, and the programmers of the mainframe servers, remain within the relevant mainframe development teams, but are responsible to the PWS development project leader for planning, task assignments, and status reporting. The composition of the EOSE PWS development team, as it finally evolved, is:

- One project manager
- One full-time project leader
- One chief programmer
- One technical analyst (host and test environments)
- Two technical infrastructure designers/developers
- One overall team leader, design coordinator, and test coordinator
- Six PWS analyst/programmers

Working closely with this team are:

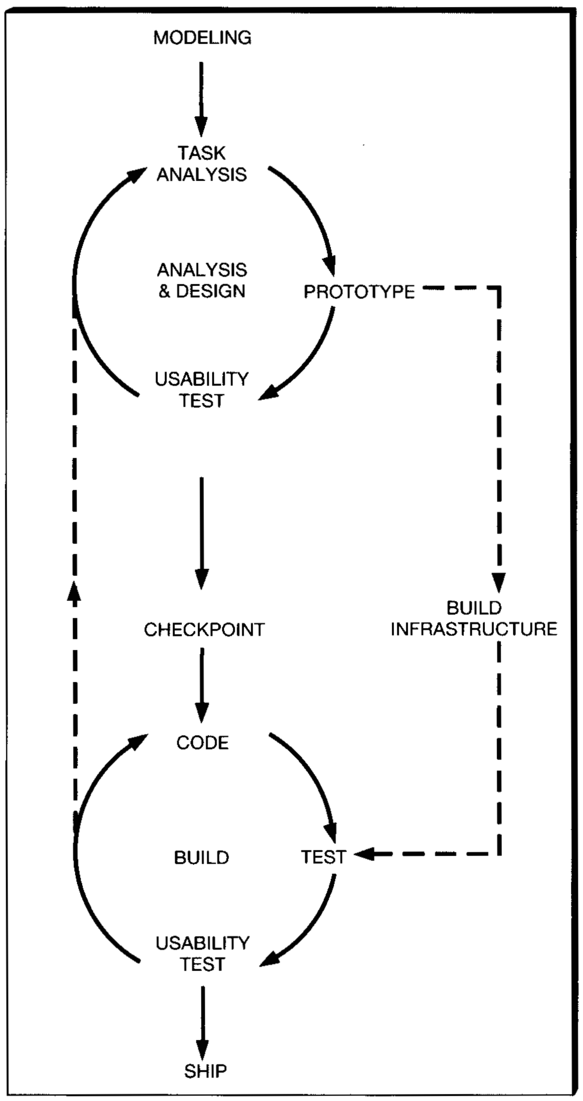
- Four end-to-end analyst/designers (client and server designs)
- Sixteen mainframe server analyst/programmers (four for each host component)
- One technical architect
- Business/user analysts from the business support group as required

In all three projects, for coding purposes each business object or group of business objects (module) was assigned to a developer, who was then also responsible for developing the appropriate infrastructure and view classes. This vertical integration (rather than horizontal integration in which different teams develop the interfaces and application objects) worked very well.

Object-oriented development certainly benefited the CICS Base project. It was possible to identify objects and assign teams (or individuals) to objects, which clearly gave them the boundaries of their task, and helped by labeling "meaningful" sections of the system, rather than just a set of programs. The fact that for the conversion to CICS only 10–15 percent of the original code was changed is a tribute to the object-oriented design. The team was able to port over 150 thousand source instructions (KSI) with a team of six in well under two years, including adding new function and accounting for the learning curve involved. Quick completion of this work, despite the immature development tools, is due partly to the fact that objects could be easily identified that would not even require desk-checking.

Metrics and status reporting. Decide at the outset what measurements are desired at the end of the project and ensure that everyone knows and that the measurements are collected regularly. One of the problems with small teams working on individual objects is that it is easy to lose control of the metrics. Status reporting is another area that must be watched as the teams tend to see status reporting as dead time and would rather continue with their work. Such reporting has been tried in some areas, and major problems are apt to be encountered because teams tend to only disclose

Figure 3 A practical development approach



the information that they are going to fail to meet a target when they have actually failed! A regular status report is a must; do not abandon all past project experiences just because object-oriented techniques are now being used.

Development approaches

Iterative, waterfall, or both? Having settled on some adaptation of standard tools and language, consider the development approach that best suits the language and your requirements. "Object-oriented development" is normally assumed to be synonymous with "iterative development" (or CABTAB—Code A Bit, Test A Bit). However, this need not be so. In the case of CICS Base, for example, the language and development environment did not encourage iterative development because of the length of time it took to compile and link code. So do not just assume that because it is object-oriented you must have iterative development.

An almost identical development approach evolved quite independently for both the SRFE and EOSE project teams, based on their experiences. It is similar to the approach discussed by Booch[9] in that it embodies distinct steps (like the waterfall approach) but takes advantage of the iterative nature of object-oriented development, allowing all the previous steps to be revisited if necessary. The required application objects are extracted from the requirements in the initial analysis phase, and the prototype uses these objects as a base for its function, thus making the transition to the design and coding phases significantly easier.

The result is an 11-step development process for building object-oriented business applications, contained within two distinct phases. This process includes the prototyping and usability testing steps needed to ensure the quality of the user interface. Although it is essentially an iterative process, it still includes a formal checkpoint, or continuation point, to allow the project costs, estimates, and schedules to be monitored. This checkpoint also allows the iterative development pattern, which is so well-suited to an object-oriented development, to be integrated with more traditional waterfall work patterns and managed with the same set of project controls.

A practical development approach. Figure 3 shows an overview of the phases of the development pattern and how it combines iterative development with formal checkpoints. This development pattern can be applied to a single object, an infrastructure component, a group of related objects or even a complete development project. It can be used for a formally planned development project or for a fast path feasibility study.

The first phase, analysis and design, starts with formal data and process modeling in order to understand the business process for which a solution is to be provided. EOSE, for example, has used BSDM as the modeling method, but any formal method which produces equivalent information is just as good. (Some other formal methods the reader may have come across include IDEF—Integrated Computer-Aided Manufacturing Definition Method—and SSADM—Structured Systems and Design Method.) The information gathered during this phase is used to create the database designs, agree on the process boundaries, and agree on the areas of the problem domain for which a computer solution is to be provided. This phase is followed by iterative analysis and design steps to collect all the user requirements, design the user interface, and identify any common infrastructure code that will be needed. For a feasibility study, or the first object(s) of a new project, four or more iterations may be needed to arrive at a design that meets the user requirements. For subsequent objects in the same project two iterations should be sufficient.

Following this phase is a formal checkpoint, where the design material (documents, prototypes, data models, and so on) are reviewed by all interested parties and the appropriate commitment is secured to continue the project, or possibly change the scope. Given this commitment, the development team can then build the complete product with the knowledge that all of the requirements are understood both by themselves and by the users.

The second phase, the build phase, is again an iterative one of at least two cycles, building first the class hierarchy needed and then the complete business objects.

Even though this work pattern is iterative, normal quality assurance processes still apply. All documents and code are subject to a planned quality check (review or inspection, as appropriate). All defects are recorded for feedback into the quality improvement process. One difference from a waterfall process is in the scope of code inspections. The code inspections are now primarily to check for completeness in relation to the design, for maintainability, and for adherence to standards. The frequent and incremental testing is the principal means of ensuring that the code functions correctly.

Within these two phases (plus the checkpoint) are the following 11 development steps:

Phase 1: Analysis and design

- Step 1: *Model the data*—The modeling is done in formal data modeling workshops, using a method such as BSDM. The result of this activity is a complete list of entity and attribute definitions for the business process, from which the table or database definitions can be created.

- Step 2: *Model the processes*—This modeling is done in formal process modeling workshops, again using a method such as BSDM, and with the same workshop participants as attended the data modeling sessions. The result of this activity is a set of scope definitions within which the rest of these development steps can take place.

- Step 3: *Analyze user tasks within each process*—This analysis may either be in (small) workshops or in individual interviews with users. By talking through the tasks a user needs to follow to achieve the business process, this step identifies candidates for the objects and methods in the user's world. The result is a "paper prototype" of the solution, which can be used to verify that the business processes can be achieved.

- Step 4: *Create sample business scenarios*—The project design team now takes the results of the modeling and task analysis and creates a set of sample scripts, or scenarios, that describe typical business processes. There should be a sufficient number of scenarios to cover the scope of the project. These scripts describe, step by step, the tasks a user has to perform to achieve the process. When complete, the design team should verify the scripts with the users. From these scripts a preliminary class hierarchy can be established and any infrastructure objects can be identified.

- Step 5: *Combine/group objects for commonality across process boundaries*—The project design team rationalizes the lists of candidate objects named in Step 3 to identify common business objects and objects that will be presented in the user interface. Again, the results must be verified with the users. As an example, the task analysis may have identified three objects in the user's world: (1) a "delivery instruction" within the business process "deliver product," (2) a "scrap instruction" within the business process "scrap product," and (3) a "transfer instruction" within the business process "transfer product."

However, by looking at the tasks that involve these three objects, we can see that they are very similar. All three objects represent a document that contains the instructions to move a piece of equipment from one place to another; the first is from the warehouse to the customer, the second is from the customer back to the warehouse, and the third is from one customer to another. Rather than present all three objects on the user interface, it may be much simpler to present a single object "movement instruction," for which much of the processing is common to all three types of documents. Only when the document details are complete does the user have to say what type of movement instruction is required.

- Step 6: *Prototype the user interface*—Prototyping is done by the development team in working sessions with the users. The result is a set of draft window designs. Depending on the prototyping tool used, it may also be possible to include prototype window flows as well as the static designs. Another result of this activity will be changes and refinements to the business processes and scenarios as the users start to see the solution "come alive." As this happens, the development process will cycle back to Steps 3, 4, or 5 and will iterate until the users are happy with the designs.

- Step 7: *Test prototype for usability*—By this step the prototype should be stable enough and robust enough that the users can subject it to a formal test of usability in a controlled environment (possibly even using a purposely-built usability laboratory). As a result of the testing, further changes may well be requested, and the development process will cycle back to Steps 3, 4, 5, or 6. The result of this step must be the final window designs, and the analysis and design phase iterates until this result is achieved.

Checkpoint

- Step 8: *Publish the design documentation*—The design documentation can take many forms. It can include hard-copy documents, "hypertext" (on-line) documents, working prototypes, and anything else that makes the proposed design understandable. The users are asked to formally commit to the designs before the next phase, the build phase, can continue.

Phase 2: Build

- Step 9: *Build common function and infrastructure code*—The development team designs, writes, and tests the common code. As much existing code should be reused as possible to reduce the chance of introducing coding errors. The common code will produce the base class hierarchy on which the business objects can be built. (Note that some infrastructure development can be started after Step 4.)

- Step 10: *Build the complete business objects*—The development team designs, produces, and tests the complete business objects, again reusing as much existing code as possible.

- Step 11: *Test the product against usability requirements*—The users subject the completed product to a formal test of usability by means of a usability laboratory or equivalent controlled environment. If this is the first cycle within the build phase, this step will almost certainly result in some requests for design changes, and the project plans must allow sufficient time for at least one more iteration after the first usability test. If the changes are relatively minor, the build phase cycles back to Step 10. For major changes it may be necessary to go back as far as Step 6 and rework the original prototype designs. Such design changes should be controlled in the same way as the original project and should be subject to a similar checkpoint and commitment to proceed.

The final cycle through the build phase should end with a usability test verifying that the product meets all requirements and is ready for release.

Conclusions

General conclusions. This section consolidates the individual project results described earlier and attempts to draw general conclusions on how they were obtained.

Code quality. Table 10 summarizes the code quality improvement factors where a measured base for comparison exists. In absolute terms also,

Table 10 Code quality improvement factors compared with non-object-oriented development

Component	Independent Test	User Test	Production
EOSE	3x	6x	100x estimated
CICS Base	10x estimated	No comparison base numbers	100x estimated

production code quality was excellent, for example, the zero defects of SRFE in over 12 months of production.

The quality was felt to be primarily due to reuse from inheritance and from data encapsulation; it applied to both object-oriented design and to OOPS. However, using an OOPS (particularly with automatic, instantaneous, incremental compile) led to further improvements from enforcing object-oriented design and enabling a CABTAB approach.

Further benefits were felt to be due to the increased code exercise from earlier in the development as a result of using an iterative approach. However, this approach did not obviate the need for traditional quality assurance activities (inspections, for example).

It should be noted that people's skill and experience will have a greater influence on code quality than any technology (including an object-oriented one). This observation was highlighted by the non-object-oriented subsystem in CICS Base.

(It is perhaps interesting to note that a completely independent and parallel study into work patterns for developing object-oriented business solutions has arrived at the same conclusions as those from the EOSE and SRFE projects. Working from the theoretical considerations, Ghica van Emde Boas has developed a method called "Framework for Object-Oriented Development"[10] which, like EOSE, has a development process consisting of 11 steps, encompassing essentially the same tasks.)

Correctness. There was a perceived overall reduction in both project change requests at independent test time (for example, just seven minor user interface changes for EOSE), acceptance test time, and post-delivery improvement requests, thus indicating an improvement in correctness (fitness for purpose).

This reduction was felt to be an indirect benefit of object-oriented technology, namely the increased user involvement resulting from prototyping as part of iterative development (and, in the technical software development arena, from "exploratory programming" aimed at understanding an unfamiliar platform).

Usability. Although formal usability laboratory testing was not carried out, feedback from users suggested that most aspects of usability were improved: users were more self-sufficient and applications were, overall, easier to use and were more consistent. All this improvement led to a more positive user attitude. With regard to negative aspects, some users did experience difficulty in learning, and there was some "ease of use" frustration where highly repetitive, fixed sequences of actions were involved. No impact was noticed with respect to user accuracy.

The main influencing factor was thought to be the object-oriented user interface. The claim that this interface more accurately reflects the user's mental model of the application was felt to be true. Ease of learning difficulties were believed to be caused by the "culture shock" of this different user interface style for long-time users of action-oriented interfaces. Consequently, this factor will disappear with subsequent object-oriented user interface applications and can thus be viewed as a justified migration cost. Highly repetitive, fixed sequences of actions were a relatively small proportion of the two relevant applications in this paper but could suggest that, where such sequences form the bulk of the application, an object-oriented user interface may not be appropriate.

The ability of OOPS to enforce standard window behavior via inheritance and, in an iterative development context, to facilitate GUI prototyping was also felt to have improved many aspects of usability.

Adaptive maintainability (flexibility, enhanceability). The relative cost of making functional changes to the two business applications, EOSE and SRFE, is lower than for traditional, procedural applications. Even more dramatic has been the low cost of porting the already object-oriented code of ASM from one environment (TSO) to another (CICS).

Object-oriented design led to higher modularity.

Object-oriented user interfaces were felt to encourage a looser coupling between objects implementing various screen representations ("views") and those holding the data of the represented business object. Consequently, new alternate views of the same business object could be developed quickly, as could changes to existing views.

Object-oriented design led to higher modularity, that is, objects having well-defined boundaries. In turn, this led to easier impact analysis of proposed functional changes and tended to isolate changes to a minimal number of object classes (for example, just adding a new method to a class). In particular, the ability of the EOSE developers to do prototyping using a non-object-oriented language "class library" is felt to be proof of the power of object-oriented design to produce flexibility.

The inheritance capability of OOPS enabled some global changes (for example, to the behavior of a specific type of view) to be made by just modifying a single class high in the inheritance hierarchy. The class browser helped SRFE developers to understand objects in terms of their associated messages and resultant processing without having to look at the code. Again, it helped reduce adaptive maintenance costs.

In CICS Base, higher modularity enabled isolation of platform-dependent code in specific classes and methods, thus leading to the low cost of adapting ASM to a new platform.

Perfective maintainability. Overall the relative effort to analyze defects and fix them was improved, though for EOSE and CICS Base a degradation was seen in analyzing defects where interactions between many objects were involved (caused, for example, by rogue messages).

The higher modularity engendered by object-oriented design and OOPS was again felt to be the main reason for the improvement. With regard to the exception above, analyzing multiobject error situations was no problem for SRFE using the built-in source level debugger of ENFIN/2 with its messaging history analysis facility. No such debugger was available for the software used by the other two projects.

Performance. Although all performance targets were met by the projects, there was a perception on EOSE and CICS Base that accomplishing the target was somewhat more difficult than with a conventional approach (the experience of SRFE on this was neutral). On the positive side, potential performance problems tended to be identified earlier in the development cycle.

The performance overhead was strongly suspected by the EOSE and CICS Base projects to be caused by the higher modularity (and hence increased messaging traffic) from object-oriented design and OOPS; for example, the performance of ASM dramatically improved when the object granularity was coarsened. However, the overhead was small relative to the overall time for GUI window painting itself. Performance problems tended to be identified and fixed earlier because of the early prototyping inherent in the iterative development approach.

Cross-project comparison. Very different projects were deliberately chosen for discussion in this paper. It was done in order to illustrate the scope of applicability of object-oriented benefits. However, it does make detailed numerical quality comparisons between the projects difficult. In particular, the coding rates differed significantly: EOSE was 462 source instructions per person-month, SRFE was 269 and 367 for the two releases, and CICS Base averaged 121.

The difference in coding rates was due to factors such as language power (source instructions per function point, for example) and development tools. ENFIN was the most powerful language, followed by SEDL++, then C. The most comprehensive and stable development tools were provided with ENFIN, followed by C (and PM), and finally SEDL++ in 1991 and 1992.

However, the latter factor does seem to correlate with "defect propensity" (the total number of defects introduced into the project for each person-month effort) given in Table 11.

Thus, it can be concluded that, in quality terms, the choice of an object-oriented language should be driven by how comprehensive and stable the associated development environment is, rather than by the power of the language.

Follow-on directions. As a result of our experiences in these and other projects, ISL now has a policy of using object-oriented technology to develop graphical user interfaces, specifically conforming to the CUA Workplace Model (CUA'91). The following actions should lead to even higher quality results from this technology:

1. Establishment of a preferred OOPS and associated, centrally administered business and technical class library, thus maximizing reuse of quality-assured code.
2. Establishment of an object-oriented mentoring group. It will help ensure good object-oriented quality practices on new projects (for example, in object-oriented performance management). Although the inexperience in the projects was felt to lead to a cautious and communicative approach to development and hence higher quality, it is not a sensible long-term strategy.
3. Object-oriented quality metrics improvements, in particular, in user-perceived quality concentrating on "new plus reused function points" as a base for calculation.
4. Introduction of a specialized library manager. A specialized library manager is necessary for version control in large team development using an OOPS with an incremental compile technique.

Summary

The results from three projects of differing types, sizes, languages, and platforms have a remarkably close correspondence when it comes to analyzing the quality impacts of their common factor, namely object-oriented technology. A number of important lessons were learned. From these evolved a practical approach to developing object-oriented software, which combines the benefits of an iterative working pattern with the quality checkpoints of a waterfall approach.

Table 11 Defect propensity

Projects	Defect Propensity
EOSE	1.2
SRFE	0.8
CICS Base	2.9

There is a common consensus that object orientation leads to all-around quality benefits. The importance of using an object-oriented language with a browser and a debugger was also demonstrated. Although a performance overhead was suspected by two of the projects, targets were met in all cases. One of the key benefits demonstrated by the projects was that defect rates measured in independent testing were significantly lower than those in the equivalent non-object-oriented components. Also, early user interface prototyping, and the close and frequent involvement of the users, resulted in a very low level of late design changes.

Other key benefits included the fact that new functions could be added quickly and easily, and to very high standards of quality, through the exploitation of inheritance, and that object-oriented development resulted in a measurably greater overall development productivity.

Encapsulation techniques meant that any errors that did occur in the code were localized and easily traceable and potential performance problems could be identified early in the development cycle.

Overall, object-oriented technology was perceived and demonstrated as a major "tool" to assist in improving software quality.

Appendix: Code metrics and reuse

A general metrics issue is the comparison of quality between two radically different technologies. For object orientation specifically, a current issue is whether the amount of reused code should be included when calculating developer productivity or, as relevant here, code quality.

Here are some examples that illustrate this point:

- If an existing program is 10 KSI in size and 1 KSI is changed to make it work on a different platform in one month, is the developer productivity:

  a. 1 KSI per month? (new code written)
  b. 10 KSI per month? (resulting program)
  c. 11 KSI per month? (old program size plus new code written)

- The issue gets more complicated when defect densities are considered; for example, if two defects are reported against the new program, is the defect density:

  a. 2 defects per KSI? (divided by new code)
  b. 2/10 defect per KSI? (divided by resulting program size)
  c. 2/11 defect per KSI? (divided by old program size plus new code written)
  d. 0 defects per KSI? (if none of the errors were in the new code!)

- What about the situation where 10 source instructions are changed in a complex suite of programs of unknown size? Should the size of the suite be counted just to determine what the productivity or defect rate is? Should the code in just the methods invoked or, alternatively, in the complete classes inherited from, or, invoked via messaging, be counted? Does it matter how many source instructions there were? (For example, in the base code that was ported to CICS, it is believed there were 150 KSI. But this is not certain, and it was not considered a good use of resource to count them, particularly since they were split across hundreds of modules in different languages.) This consideration is aside from the general issue of the different productivity rates of programming languages.

So, should KCSI be used as a measurement at all? Maybe function points should be used, but, for example, how many function points do you count when changing one source instruction (or even no instructions) to make something work on another platform?

Another question is, how many errors should be counted if a user raises one error when at fix time it transpires that there were in fact two errors? And what about errors in logic that would never materialize in the end user's function? (The CICS Base project had such a case where an internal function would have crashed if it was passed a zero value. The function was supposed to handle a zero parameter. However, none of the calls made to it would ever have passed zero because of other factors. Was this a defect? It was meant to handle zero because it was intended to duplicate the same function as on another system.)

It is thus evident that both approaches are valid, but for different purposes: metrics based on "new plus reused code or function point" reflecting an external, customer-perceived view of application "size" and metrics based on "new code only or KCSI" being most suited for internal, comparative analysis of the development tools and techniques used. Hence, this paper has used the latter base for metric calculation, admittedly at the cost of a tendency to underestimate the value of object orientation.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of ENFIN Software Corporation.

Cited references and note

1. IBM Information Solutions Ltd. is a wholly owned subsidiary of IBM United Kingdom Holdings Ltd. It was set up in April 1992 to provide outsourcing capability to customers as well as to IBM itself. The Software House is the part of ISL that develops applications for both external customers and IBM using advanced tools and techniques such as those that are object oriented.
2. O. Sims, *The New World*, IBM UK System Design, Basingstoke, United Kingdom (1989).
3. R. E. Berry and C. J. Reeves, "The Evolution of the Common User Access Workplace Model," *IBM Systems Journal* **31**, No. 3, 414–428 (1992).
4. *Business Systems Development Method* (5 volumes), IBM AD Solution Centre, IBM UK Ltd., Chiswick, London (1993).
5. *Systems Application Architecture Common User Access Guide to User Interface Design*, SC34-4289, IBM Corporation (October 1991); available through IBM branch offices.
6. C. S. Gee, *A Methodology for Object Oriented Design*, IBM International Technical Support Center, Roanoke, TX (1989).
7. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software,* Prentice-Hall, Inc., Englewood Cliffs, NJ (1990).
8. *Function Points Counting Practices Manual,* The International Function Points Group, IFPUG, Blendenview Office Park, Westerville, OH (1990).
9. G. Booch, *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Reading, MA (1990).

10. G. van Emde Boas, *Framework for Object-Oriented Development,* IBM European Systems Architecture and Technologies, Uithoorn, The Netherlands (1992).

Accepted for publication September 13, 1993.

Nigel P. Capper *IBM Information Solutions Limited, P.O. Box 41, North Harbour, Portsmouth, Hampshire P06 3AU, United Kingdom (electronic mail: IBM Mail Exchange: GBIBMWTZ at IBMMAIL; Internet: GBIBMWTZ@IBMMAIL.COM).* Mr. Capper is a Senior AD Specialist in the ISL Software House. After completing full-time education he entered the IT business in 1980 as a trainee programmer with local government. After five years of programming and analysis of application systems he joined a software house as a senior programmer/analyst. Here he progressed through project leader, project manager, and deputy business manager in the IBM marketplace. He joined IBM in 1990 and led "CICS Base," a major international object-oriented software development project. His current focus areas are transaction processing, particularly CICS, client/server applications, and reuse.

Roger J. Colgate *IBM Information Solutions Limited, P.O. Box 41, North Harbour, Portsmouth, Hampshire P06 3AU, United Kingdom (electronic mail: IBM Mail Exchange: GBIBM4MX at IBMMAIL; Internet: GBIBM4MX@IBMMAIL.COM).* Mr. Colgate is an advisory analyst in the ISL Software House Technology Group where he is responsible for application development technical strategy. He studied at Cambridge University, gaining a degree in mathematics in 1971 followed by a postgraduate diploma in computer science. He joined IBM internal information systems in 1973 as an application programmer and, following work on IMS on-line applications, became a database design specialist at a European level on an assignment from 1981–84. Since then he has driven the introduction of new technology and techniques into ISL, specifically: technical assurance reviews, application performance management, DB2, and CSP. His current focus areas are AD/Cycle and object-oriented technology.

Jim C. Hunter *IBM Information Solutions Limited, P.O. Box 41, North Harbour, Portsmouth, Hampshire PO6 3AU, United Kingdom (electronic mail: IBM Mail Exchange: GBIBM5ZX at IBMMAIL; Internet: GBIBM5ZX@IBMMAIL.COM).* Dr. Hunter is a programmer/analyst in the ISL Software House Technology Group currently working on the SRFE project. He studied at City University, London, gaining a degree in mechanical engineering in 1984, and a Ph.D. in 1987. He joined IBM International Information Systems in 1990, and almost immediately moved into the object-oriented arena with the SRFE project, which he has been working on ever since.

Martin F. James *IBM Information Solutions Limited, P.O. Box 41, North Harbour, Portsmouth, Hampshire PO6 3AU, United Kingdom (electronic mail: IBM Mail Exchange: GBIBM9XX at IBMMAIL; Internet: MJAMES@VNET.IBM.COM).* Dr. James is a technical architect for the EOSE development projects in the ISL Software House, with particular responsibility for PWS development. He studied at the University of Birmingham, gaining a degree in physics followed by a doctorate in high energy physics for research done at Birmingham and at the *Centre Européenne pour la Recherche Nucléaire* (CERN) in Geneva. He joined IBM in 1976 as an application programmer on IMS business applications. After 11 years designing and developing MVS technical infrastructure software in the United Kingdom and the Netherlands, in 1989 he joined the EOSE development group as one of a small team of technical architects to advise on the use of new tools, techniques, and technologies.

Reprint Order No. G321-5536.