Reaping Benefits With Object-Oriented Technology

Yogeesh H. Kamath
Ruth E. Smilan
Jean G. Smith

Object-oriented technology (OOT) has been a valuable asset to our product development cycle. It resulted in significant reuse of code, design, and analysis; reduced development cycles; simplified system integration; and a more efficient method of generating and using requirements. By incorporating manageable pieces of OOT into our development process, a few techniques at a time, we have been able to learn about it, introduce it at a manageable rate, and still meet our customer commitments without asking for additional resources for OOT work. Our developers and system engineers continue to incorporate OOT techniques into their analyses, designs, implementations, and testing, and look for new areas to enhance our existing paradigm. This paper is for project managers, architects, system engineers, and developers who are using, or are considering using, OOT. Our methods can be used in other projects to shorten the development cycle and enhance product quality.

Introduction

For the last five years, the Call Attempt Data Collection System (CADCS) development group has been using object-oriented technology (OOT), a product development methodology that uses object principles rather than traditional data structures and procedures. The benefits of OOT have been reduced product development time, increased design and code reuse, analysis reuse, and improved quality. Three major areas addressed in this paper are:

- *A development approach using OOT*—How OOT was introduced and supported in our project development cycle.
- *Object-oriented requirements*—How CADCS requirements shifted from prose, functional requirements, to object-oriented requirements, and how the format and methodology were defined to produce them.
- *A new project startup based on CADCS*—How the project was extended into new application areas, and a new service management system was delivered in eight months.

CADCS Background

CADCS provides a set of capabilities for AT&T customers to monitor traffic in their 800-Service networks. It collects, processes, and delivers data associated with call attempts on 800 numbers. Real-time data is delivered through a computer-based graphical interface, and historical data is delivered through a computer-based ASCII interface, or by a printer. The service capabilities also are used by internal AT&T support organizations for service and system management.

CADCS consists of about 350,000 lines of code (C++ and other languages) that run on a central processor, and 100 remote systems distributed across the United States and interconnected via an X.25 packet network. CADCS was first released in October 1990, with subsequent releases delivered every six months. Twenty-five developers and four systems engineers are responsible for the planning, feature requirements, architecture, development, system test, and field support for CADCS. While the systems engineering and development groups are separated geographically and organizationally, we were able to refine our object-oriented techniques in a stable, friendly environment.

Development Approach Using OOT

The CADCS development paradigm is a mixture of functional and object-oriented approaches that has evolved over the past five

Panel 1. Acronyms and Terms Used in This Paper

Abstraction	A technique that talks about functions and data in higher-level terms in order to deal with complexity.
ASCII	American Standard Code for Information Exchange
C++	Programming language
CADCS	Call Attempts Data Collection System
CASE	Computer-aided software engineering, a generic term for software tools that support a specific way of documenting a process.
Class design	The software representation of an object.
Common class	A general class that is used by multiple applications.
Containment	The incorporation of one object within another.
Inheritance	The technique by which the key, or essential, elements of an implementation can be "inherited," or reused, by objects derived from an abstract object.
Invocation	The ability of one object or code segment to call on a behavior, or method, of an object.
MR	Modification request
NVT	Network verification test
OOA	Object-oriented analysis
OOD	Object-oriented design
OOP	Object-oriented programming
OOR	Object-oriented requirements
OOT	Object-oriented technology, a technology in which one works with objects, where data structures and behaviors or services are incorporated, as opposed to the traditional data structures and associated data manipulation routines.
PCS	Personal communications services
PSMS	PCS service management system
SMS	Service management system
SQL	Structured query language
X.25	CCITT packet protocol

years.[1] We generally have followed a bottom-up approach for OOT, replacing functional, traditional approaches with OOT at low-level design and programming stages, and then extending the use of OOT to higher-level designs and requirements definition. By carefully choosing the areas we would address with OOT, avoiding too much new technology (and risk), and monitoring the overall development cycle, we have successfully managed its introduction and growth while meeting our project commitments.

Transitioning from Functional to OOT Methods. The possible combinations of methods that can be used in different phases of the development cycle are shown in Table 1.

At project startup in 1988, we followed a traditional operations-systems development paradigm, with functional requirements, architecture, and high-level system design (Line 1 of Table 1). An object-oriented programming (OOP) approach was introduced into the detailed design and implementation phase of our initial software release (Line 3). Over the next six CADCS software releases, we gradually became C++/object-oriented design experts.

We then concentrated on the performance aspects and OOD reuse, as we enhanced our use of OOD and OOP in detailed designs and implementations, and used OOD and object-oriented analysis (OOA) in high level designs (Line 4). In the most recent release, a team of developers and system engineers wrote object-oriented requirements (OOR), thereby combining the system engineers' requirements phase with the developers' object-oriented analysis phase (Line 5).

Figure 1 shows this OOT evolution—how we learned about an OOT area; how we incorporated it into our methodology; how we became proficient at it; and, finally, where we tackled a new area with OOT.

Small Teams. The use of a small-team approach on CADCS has enhanced our use of OOT techniques by facilitating information sharing, and providing a forum for team ownership of issues and decisions.

Within the development group, teams of two to five people were set up to decide about system design, class design, reuse, methodology, common classes, etc. The teams cover common-class development, application and feature development, database work, and data communications. They have clear goals and deliverables, and document their results via manual pages and methodology guidelines.

Some teams cross organizational boundaries to include system engineers, developers, and testers for system release planning, architecture definition, system integration, and system verification.

Project Management. We planned OOT work as part of the overall system design and development program. An incremental development strategy was created, including well-defined schedules and demonstrations that were tracked with other project deliverables. Initially, OOT was considered a risk item, and managed so that it did not jeopardize project commitments. Along with OOT, we strongly feel other quality practices, such

Table I. Alternatives for requirements and development methods

Line	Requirements	Analysis	Design	Implementation
1	Functional	Functional	Functional	Functional
2	Functional	Functional	Functional	OOP
3	Functional	Functional	OOD	OOP
4	Functional	OOA	OOD	OOP
5	OOR/OOA		OOD	OOP

as best current practices (BCPs), must continue to be used. Booch^2 cautions project managers, saying "The use of Object-Oriented Design doesn't give one the license to abandon the established practices of quality assurance."

By checkpointing our development progress monthly, we reestimated OOT work, along with other project work, to keep us on an achievable schedule. Sometimes we lowered the priority of OOT work when other higher-priority items were in conflict. Our incremental approach and frequent demonstrations provided ways of seeing early results of the working system—as well as potential problems. In every CADCS release, time and quality commitments have been met.

Methods. When we began looking at OOT for CADCS, our process and methods were defined at a high level. The teams knew, generally, what was going on and understood their roles. As new people joined the group, the methodology document was one of the first things they read.

Currently, development methodology (see Figure 2) includes:

- Working with system engineers to produce OOR,
- Transitioning from OOR to produce development objects, and centralizing the list of objects and their resulting classes,
- Establishing conventions for style, design, and class documentation, such as manual pages, and
- Forming object teams composed of designers, developers, and application users to define the class functionalities needed in the various objects.

The object teams then prepare manual pages for the classes, so that application developers can proceed while the classes are coded and tested. After a class is tested and entered in a library, it can be used in the application. At the same time, non-C++/non-object-oriented development also proceeds.

Guidelines

General guidelines that were followed for using OOT to develop CADCS are discussed.

Common Class Development. Initially a few domain-class developers took on the task of learning about OOD, and they supported the team's application developers as the latter transitioned to OOD techniques. We focused on a core set of code, central to all parts of the system, because code reuse was one of the main benefits to be achieved with OOT. By defining this team at the beginning of the development process, along with the responsibility of defining common classes, class reuse was enforced. Now, common-class development is re-examined as the features are analyzed by developers at the beginning of each release. The domain-class developers focus efforts on:

- Performing object-domain analysis of the system to identify common classes, then building common/abstract classes that can be used by multiple application developers. These are general-purpose objects within our problem domain. Examples of problem-domain classes are dialed numbers, service type, and area codes—common objects at the heart of the system.
- Providing good documentation by maintaining manual pages and design documents, soliciting feedback from the application developers, and establishing small teams of users that review and revise the documents.
- Implementing key parts of the domain/common classes early in the development cycle. This permits application developers to use working code. These key parts are identified jointly by the common-class development team members and the application developers, so deliverables are understood by both groups.

Application Design and Development. The application developers focus their efforts on:

- Deciding how much object-oriented design they want to handle, and when they should use functional-design

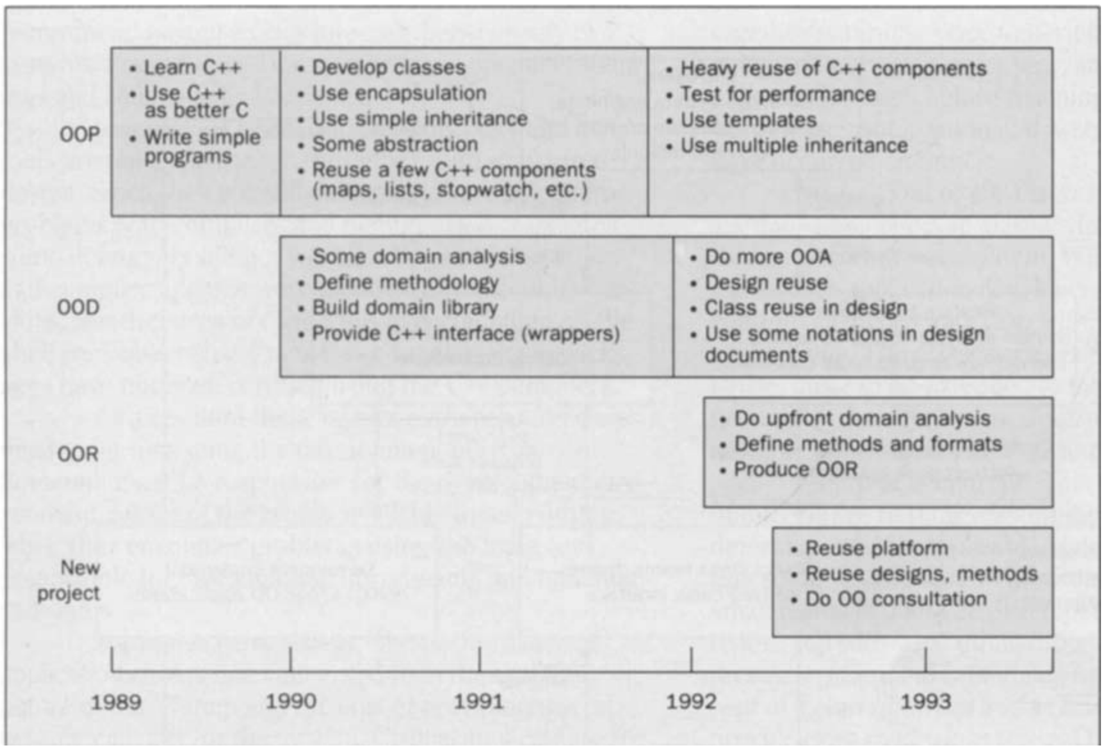


Figure 1. Object-oriented technology (OOT) was introduced over a four-year period by the developers, starting with high-level tutorials in 1989, and progressing in an incremental but systematic way until the project was completed in 1993. The developers garnered enough expertise in this time that they were able to reuse much of their work on a new project starting in 1992.

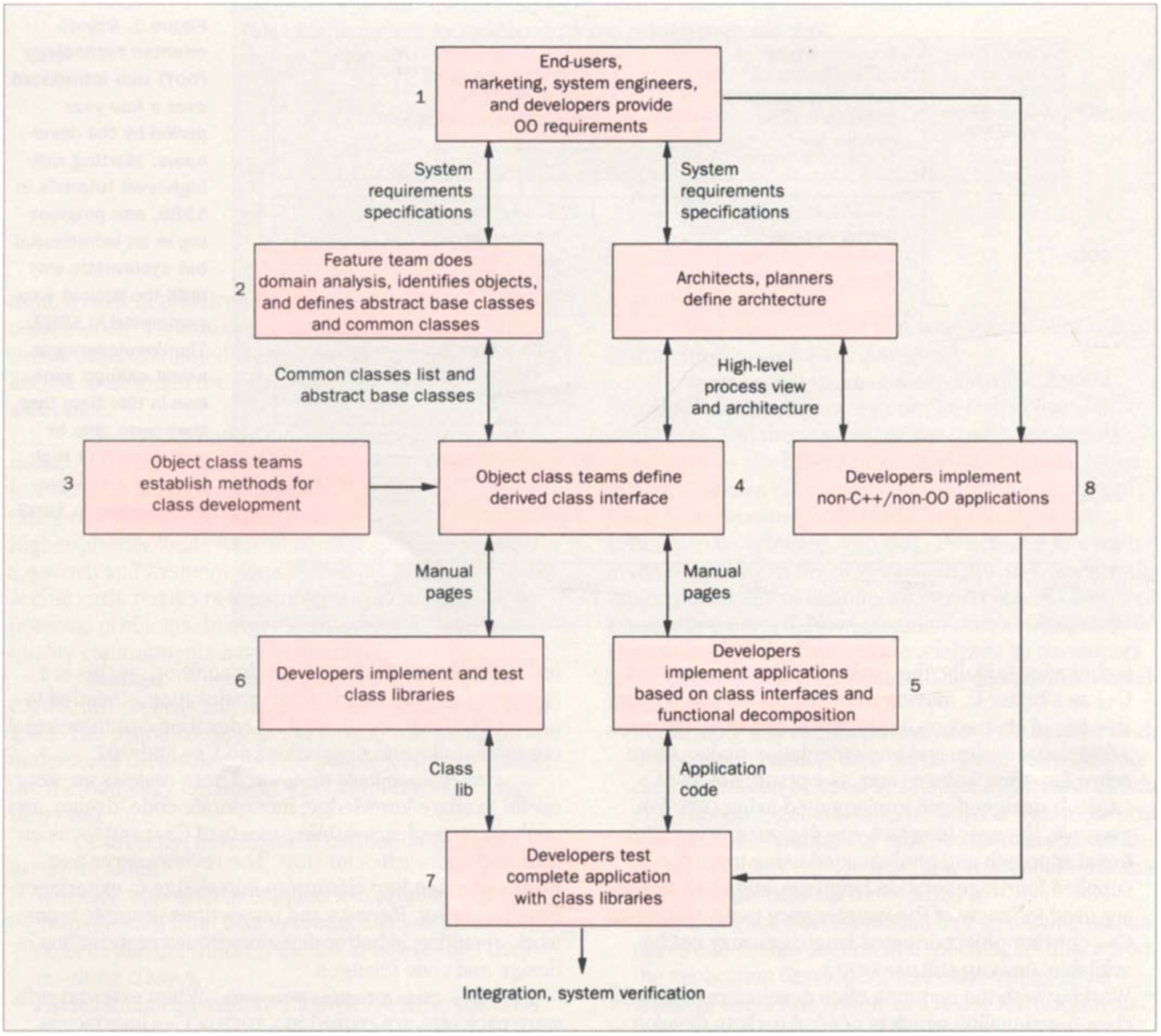
techniques. Initially, the application developers used C++ as a better C, moving over time to the more extensive use of abstraction, inheritance, and OOD. At times, a functional design and implementation makes more sense for some features and, as a result, not all of CADCS is designed and implemented using OOT. For example, the user interface was designed using a functional approach and implemented using a vendor-supplied fourth-generation language, and shell scripts are used for many of the maintenance tools. (Although C++ or other object-oriented languages may not be available, one can still use OOD.)

- Working with the common-class developers to identify class functionality, which is needed early to develop working prototypes.
- Designing processes, as well as classes. There are two main areas of design: class design and process design, or system architecture. As OOT was applied to class design, process design using common and application-specific classes was done in parallel, using the traditional, structured design paradigm.

Education. We recognized early on that we needed a set of required courses and time allocated for team members to attend them. In addition, as this is a complex paradigm, and ongoing education is needed to use it effectively, we created an education committee that organizes talks and discussions on C++ and OOT.

Design and Code Reviews. These reviews are very useful to share knowledge; incorporate code, design, and analysis reuse; learn subtle aspects of C++; and focus on standard approaches for OOD. The reviews serve as a means of extending classroom knowledge to experience-based learning. Reviews and inspections promote teamwork, resulting in better designs with fewer iterations in design and code changes.

C++ Class Interface Wrappers. When external software packages are reused in CADCS, a C++ interface is added, whenever possible, to take advantage of C++ and OOT benefits. For example, a thin layer of C++ class libraries was added on top of the vendor-provided X.25 library package. The interface provided a convenient way of hiding low-level details of the vendor package from the object-oriented applications used for data collection and distribution. Another example is the SQL++ class, designed by another project, which provides a generic interface to a relational database.



1
End-users, marketing, system engineers, and developers provide OO requirements
System requirements specifications
System requirements specifications
2
Feature team does domain analysis, identifies objects, and defines abstract base classes and common classes
Architects, planners define archtecture
Common classes list and abstract base classes
High-level process view and architecture
3
Object class teams establish methods for class development
Object class teams define derived class interface
4
Developers implement non-C++/non-OO applications
8
Manual pages
Manual pages
6
Developers implement and test class libraries
Developers implement applications based on class interfaces and functional decomposition
5
Class lib
Application code
7
Developers test complete application with class libraries
Integration, system verification

Figure 2. Currently, the development methodology used includes: Working with system engineers to produce object-oriented requirements (OOR); transitioning from OOR to produce development objects, and centralizing the list of objects and their resulting classes; establishing conventions for style, design, and class documentation, such as manual pages; and forming object teams composed of designers, developers, and application users to define the class functionalities needed in the various objects.

Class Interface for Interprocess Communication. One of the implicit rules that has worked well is process-to-process interfaces via a class—any data sharing between processes must have a class wrapper around it. These wrappers, by being defined as classes, help ensure a clean, well-defined interface, with the functionality reused on both sides of the IPC, as well as in the test tools. This hides the detailed information of the class implementation from users. Where possible, we recommend having a class interface between any two systems sharing data. Doing so reduces misunderstandings and facilitates maintenance.

Development Environment Support. C++ and OOD tools are not as mature as those for C and structured design. Since they are still emerging, there were some problems with compilers and debuggers. For example, some debuggers did not work for large process sizes, and compiler updates were inconsistent with older versions. Another area of concern is in recompiling off-the-shelf packages using a new C++ compiler. Some packages have not been certified using the C++ compilers.

As a result of these issues, extra resources are needed for managing the development environment. Someone must be responsible for the development environment, aware of the issues, available to help others when they encounter problems using the tools, and responsible for investigating, introducing, and updating OOD tools.

Application Performance. The performance of applications is a prime concern, due to the real-time nature of our system and the cost of providing the processing capacity for the system. Critical applications are prototyped, and performance measured, to ensure that they stay within acceptable limits.

Processes can grow very large when we include large C++ libraries, which can result in performance issues due to process swapping. As a result, we needed to fine tune some processes. Unlike C functions, in C++ you cannot choose subsets of the class/object libraries that are of interest. Instead, you have to include the whole class hierarchy defined in the library.

Performance implications of an object must be understood before reusing the object for purposes other than its original intent. Sometimes, it is desirable to build a new object with less performance overhead, rather than reusing an existing object with unused functionality and performance overhead.

Benefits of OOT

By using OOT, CADCS has realized the benefits discussed below:

Reduced Development Cycle. Development estimates have been reduced by one to two weeks per feature, in many cases, where average feature developments run in the three-to-six month range. Much of the savings comes from reuse of existing designs and classes. Integration testing shows significant savings—features fall together naturally, since underlying classes are shared between interfacing processes, and classes have been extensively tested before reaching integration. In all of our releases, integration and system-verification handoffs have occurred on time.

Reuse. Out of 492 classes developed in CADCS, 190 (38%) are common classes that are reused in more than one software subsystem. With a rich set of classes available to application developers, the application teams continually incorporate OOD within all CADCS mainline applications. The classes are generally straight-forward to use, and can be extended to meet new variations needed by the developers. Considerable reuse also is possible for designs and requirements, along with code.

A caveat is in order. There is a cost involved in finding things to reuse. Searching libraries of classes to determine what is applicable and reusable can sometimes take longer than writing a new class. Even within small teams of about 20 developers, it is sometimes hard to keep up with what other subteams are doing to see if there are additional candidates for class reuse. The concept of a *class librarian* and a *class browser* might help provide even more class reuse. One person could serve as the contact point as separate teams identify the need for new classes. The class librarian could determine if others also have identified this need, thereby making it a new common class. Or, if no one else had yet identified the need, the librarian could use expert judgment to determine if it should be a common class and, if so, perhaps identify additional member functions to make it more general purpose.

Resilience to Change. When good class definitions are created and developed, the system seems to grow gracefully—new features tend to follow the same system flows, and are able to reuse existing designs and classes.

For example, as the CADCS service grew, new service types were added to our customer records, and the system requirements for area codes were changed. Because these changes were localized to objects, we could accommodate the modifications in a straightforward manner, without causing ripple effects.

Fewer Modification Requests in OOT Areas. Since working code is used, rather than newly developed code, many bugs can be eliminated through code and design reuse. Commonality in design approaches also leads to more standard interfaces and anticipated behaviors, resulting in fewer coding problems.

In our most recent release, the number of bugs

Table II. File types and files modified

File Type	Total KLOC	Total # of Files	% of Total Files	# of Modified Files	% of Modified Files
c/c++ mainline	131	1125	36	24	8
c/c++ ui	24	65	2	36	11
shell	60	466	15	43	14
forms/ui	44	768	25	183	58
other	118	690	22	28	9
total	377	3114	100	314	100

found in our network verification test (NVT) phase (field testing that occurs after system testing is completed) was analyzed. Testers created 78 modification requests (MRs) in two months of NVT testing. As a result of these MRs, developers modified 314 files. Only 8% of these modified files were C/C++ files in the mainline processing. This file-type breakdown is shown in Table 2.

Most (81%) of the files modified during this testing phase were in shell scripts, user-interface forms, and user-interface supporting code. OOT techniques were not applied either in designing and coding shell scripts, or in the code implemented in user-interface languages. The user interface is a complex area, and many changes are required late in the development cycle, as user needs become better known. Certainly not all modifications are attributable to the structured/functional design paradigm, but it is reasonable to assume that this section of code would be more bug-free, and easier to maintain, if it had been designed and implemented using OOT.

Object-Oriented Requirements

The most recent OOT change on this project was the incorporation of OOT into the requirements phase. Our goal was to facilitate the process of producing requirements that are easy for systems engineers to write and developers to understand. The requirements must be complete, precise, and facilitate the transition from requirements to design.

The term "object-oriented requirements" can have several interpretations. What many people mean, when referring to object-oriented requirements, is the notation needed to transition from functional requirements to object-oriented design. For example, Bailin[3] describes "a method of analyzing requirements for object-oriented software." That is not what we mean. We mean writing the requirements specification in the form of objects, so that no transition is needed from text requirements to object-oriented design.

The main trigger in moving us toward object-oriented requirements was a new set of reporting features. From a marketing and systems engineering point of view, the features were independent features, separate from each other, since they were used for different types of customers. But from a development point of view, there was considerable software commonality among the reporting features. It was difficult to find this commonality in the functional requirements, and it took a major effort to shift from requirements to design. It was this difficulty that led to the idea of using objects for requirements. It became clear that by using objects, synergies among features could be discovered early and used to advantage.

The experience motivating us to do OOR was a powerful demonstration of Berard's observation that, "A functional decomposition 'front-end' to an object-oriented process, in effect, breaks up objects and scatters their parts. Later, these parts must be retrieved and relocalized around objects."[4] Also, by using objects in requirements, reuse could apply when writing requirements, both within a release across features and from one release to the next. Refer to Jordan[5] for more details about the format, methods and benefits of object-oriented requirements.

OOR Guidelines

This section describes some of the guidelines used to produce our first OOR.

System Engineers—Developers Team. The shift to the new requirements format followed a number of meetings between the systems engineers and developers. Using the project's existing base of object-oriented techniques, methods were defined. A hierarchical list of requirements objects in the existing system was defined, so we could all understand what was meant by requirements objects.

Requirements objects are basically real-world objects, not design or programming objects. It is important to note that design is not specified in the requirements, and that requirements objects do not always map directly to programming objects. Requirements objects, such as a telephone switch or a report, are in the problem domain. Programming objects, such as a list or message queue, are in the solution domain. Thus, requirements writers do not have to become object-oriented programmers, but do have to learn how to describe the real world in terms of objects.

OOR Methods. Methods and formats were created, rather than relying on a CASE tool. This was done for three reasons. First, it meant both very low overhead and a small learning curve. Second, we wanted to take advantage of the OOD work that already existed in the development community, and to extend this to the requirements phase. Third, it saved time and, based on previous studies, object-oriented CASE support is still somewhat new and limited.

Requirements Document Format. Our format is based on the class responsibility collaborator cards (CRC),[6] with extensions defined for inheritance, containment and invocation. Two templates were defined for requirement objects, one for an object definition and one for each object responsibility. An object has one definition, and usually more than one responsibility. Each responsibility defines a specific requirement. The data and related functions are tied together by the object name field of the two templates.

An object definition consists of the following data:

Object:	Name of object
Definition:	A one-line description
Status:	New or existing
Child Of:	Any parent (inheritance)
Contains:	List of objects within this one

The "Status" field indicates whether this is a new object in this release, or an object modified from a previous release, to aid in reusing objects across releases. A "Parent Of" field was initially defined to indicate the inheritance relationship, but we felt it would take too much effort to keep this up to date. It also is redundant with the "Child Of" field.

An object responsibility consists of the following data:

Object:	Name of object
Abstract:	One-line abstract of this responsibility
Responsibility:	The detailed requirement
Collaborator:	Other objects invoked, if any
Invoked By:	Object(s) calling this responsibility

By writing requirements within the context of object templates, a precise language is imposed upon the requirements. Bailin[3] suggests using a *requirements database* as a starting point of identifying problem-domain objects, rather than starting with text requirements. He defines a requirements database as "distilling the original textual statements into a set of traceable requirements." Our OOR serve as both a requirements database and the complete set of traceable requirements.

Benefits of OOR

After the first cycle with OOR, we identified several benefits. Further use of OOR and metrics are needed to quantify these results, particularly in the areas of reuse and resilience to change. The benefits are:

Completeness. The requirements were more complete, since questions were asked as requirements were written. For example: What object invokes this requirements object? What object does it collaborate with to get its job done?

Communications. OOR improves communications between the systems engineers and the developers. This leads to a better understanding of the requirements, as they are being written, and provides developers with a head start on the design/development phase.

Reduced Cycle Time. OOR resulted in eliminating the paradigm shift between the requirements phase and the development phase. Berard[4] notes that, "There have been quite a number of attempts to reconcile the output of a non-object-oriented process with the input requirements of an OOD process... none of these scenarios are as clean and easy as using an object-oriented approach from the very beginning of the software life-cycle."

In previous releases, the end of the requirements phase and the beginning of the development

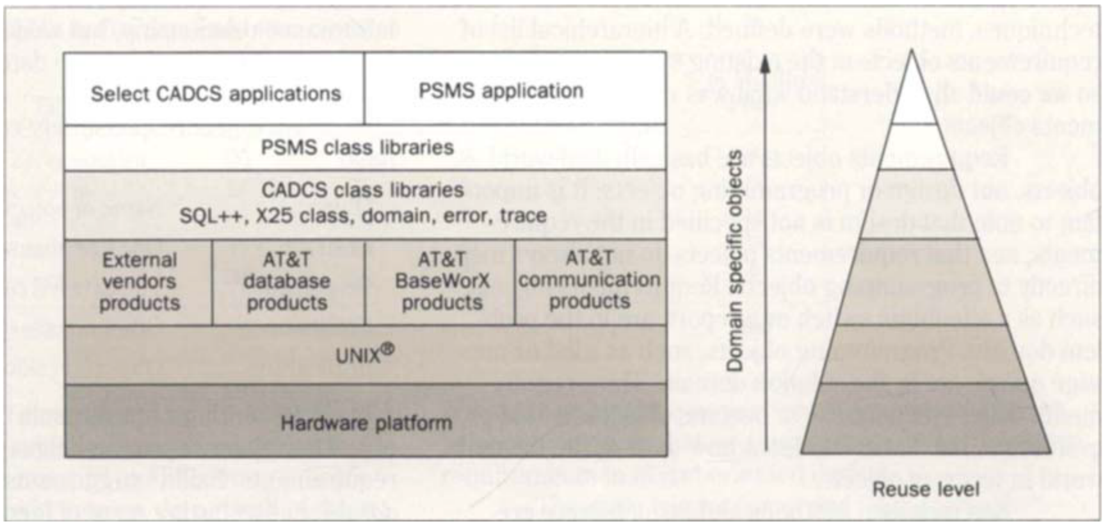


Figure 3. This illustration shows the architecture layers in the PCS service management system (PSMS) product, built using the CADCS core functionality. By using the CADCS class libraries, PSMS was able to reuse the same lower layers of the CADCS system, such as external vendor products, BaseWorX, etc. New layers were added, of course, in the domain-specific PSMS application layer and class-library layer.

phase were marked by what was called a "conceptual design review." When the requirements were complete, the developers reviewed them and produced a high-level design of the system, incorporating the new features into the existing design. The developers and systems engineers then met to review this design.

By using object-oriented requirements, the conceptual design review step could be eliminated. Developers were more involved with the requirements as they were being written. Both groups agreed on what the requirements objects were before the requirements were written, so there were few surprises in the final specification.

In comparing the development cycle times for the last two releases, and normalizing the data to take the release sizes into account, we found an:
- 8% reduction in requirements time,
- 30% reduction in requirements staff effort,
- 30% reduction in development time, and
- 20% reduction in development staff effort.

Reuse and Resilience to Change. One of the primary obstacles in writing requirements is the instability within the customer base and feature-set base. A lot of information about customers and features is not available at the start of a release cycle. This causes a tremendous amount of churn while writing requirements, reviewing requirements, and starting development. The reuse of objects should help as feature sets shift during a release.

General requirements objects should be able to be reused from one release to another. There are base objects that pertain to the system in general that will span releases. Also, per-feature requirements objects should be reusable as features are enhanced in subsequent releases.

New Project Startup

After applying OOT to the product development cycle through several releases of CADCS, we had an opportunity to apply this experience to building a new system. It is interesting to see how much of this experience, as well as the assets, such as source code, tools, processes, and documents, could be leveraged in this effort.

The result is a PCS Service Management System (PSMS) for the Personal Communication Services (PCS) Business Unit. We had eight months to define requirements, design, develop, and verify the system, and deliver it to first-office application testing. The first release of the PSMS system consisted of about 160,000 lines of application code (C++ and others) that ran on a central processor. In addition to this delivered application code, we integrated into the system packages of third-party vendors. A team of 12 developers was

Table III. PSMS class reuse and new development

Description	CADCS classes	Reused as is	Modified reuse	Developed new
Domain specific classes	302	4	7	47
Common classes providing functionality such as networking, error handling, and DB access	190	49	5	13
Total classes	492	53	12	60
C++ NCSL (out of 160K total code)		25K	5K	27K

responsible for the architecture, design, development, and system verification of the PSMS system. We met our goal of delivering a high-quality system on time, while spending less than the allocated budget. To summarize the key ingredients of our success in one sentence, we reused the assets from CADCS by strategically including reuse in the PSMS development plan.

Startup Guidelines

Some of the startup steps that we took for the new project included:

Planned Reuse. McClure points out that one of the reasons for limited success in software reuse across projects is a lack of up-front planning for reusability.[7] From the conception of the PSMS system, reuse was considered strategically required to meet the cost and schedule demands of the customer. Because the PSMS development was started with a few seed developers from the CADCS project, several forms of reuse occurred. This reuse includes CADCS hardware and base software platform, the CADCS user interface package and general approach, the development environment and methods, application source code (C++ and other languages), and design ideas. Also, we reused the test tools and test environment, system verification approach and tools, internal documentation formats and templates, external user documents and procedures, project management tools and techniques. In addition, we were able to take advantage of staff and first-line manager continuity.

All of these areas of reuse contributed to producing a high-quality system on a tight schedule.

Software Reusability. Figure 3 shows the architecture layers in the PSMS product, built using the CADCS core functionality. New layers were added in the application layer and class-library layer. By using the CADCS class libraries, PSMS was able to reuse the same lower layers of the CADCS system, such as external vendor products, BaseWorX, etc.

The CADCS and PSMS applications have some similarities but, from a functional viewpoint, they are very different. By focusing on underlying similarities that would apply for most operations systems, such as system administration, data communications, event reporting, and report generation, much of CADCS software was reused. Differences were localized to the application and class-library layers. Here, PSMS software supports provisioning, billing-data processing, and service-data analysis features, in lieu of CADCS features, such as real-time graphical status displays and historical reports.

Instead of creating one large library, CADCS developers grouped objects into specific libraries, depending upon the nature of the work the objects performed, such as an X.25-class library for general-purpose communications, a SQL++ class library for vendor database access, and a CADCS domain class library for the application-specific objects. As a result, it was easy for the PSMS project to pick up what was needed. Reuse can be maximized if objects are more general, and are packaged so that other projects have the flexibility to pick and choose. Reuse should be based upon a choice of libraries and components, as opposed to an all-or-nothing basis.

As a measure of class reuse, Table 3 shows the number of classes that were reused with and without modification from CADCS, and the number of new classes developed in PSMS.

Notice that as the objects become more domain-specific, less reuse occurs, as is shown in the pyramid diagram. Efforts should be made to push the class libraries down in the hierarchy to maximize reuse. We also can see that 52% of the PSMS classes came from

CADCS, many from the lower levels of the hierarchy, such as error handling, database access, etc.

Quality, Cost, and Schedules. Software and methodology reuse has resulted in a quality product delivered in a short time frame. The modification request density, that is, the number of MRs per kilo-lines of code, was calculated to be less than 0.1%. This rate is much lower than we projected, based on the CADCS project. While the entire reduction cannot be attributed to OOT, it seems to have played an important role.

The PSMS development team estimated that had the PSMS been developed from the ground up, and not based on CADCS, it would have cost 1.8-2.2 more in budget and 1.5-2.0 more in development time. The general consensus was that planned reuse resulted not only in savings in development effort, but also in integration and testing efficiencies, leading to an overall cycle reduction. OOT eased the modification and reuse of existing CADCS code for PSMS, allowing us to leverage the technology investment of CADCS in building PSMS.

Conclusions

Many projects ponder the question of whether or not to get into OOT. The experience described in this paper may help answer some of the questions. OOT does not solve all of the problems of software development and project management. For example, people issues and quality assurance remain outside the scope of OOT. But by introducing a few techniques at a time, and treating them initially as risk items that need to be managed, we have been able to learn about OOT, implement it at a rate the team was comfortable with, and meet our commitments without asking for additional resources.

OOT can be practiced at a low cost without using commercially available CASE tools. A small, empowered project team defined a methodology that the project could handle, and then implemented it. Our OOR method is simple and straight-forward to use with UNIX tools.

OOT has been a valuable asset to our product development cycle, resulting in significant reuse. However, reuse has to be a project norm—planned from the beginning—and reuse can apply to much more than source code.

We are glad that we took the risk of introducing OOT in our product development cycle. Our developers and systems engineers continue to use it, and we believe these methods can be used in other projects to improve the development cycle and enhance product quality.

References

1. Kamath, Y.H. and Smith, J.G., "Experiences in C++ and Object-Oriented Design," *Journal of Object-Oriented Programming*, Nov.–Dec. 1992.
2. Booch, G. *Object-Oriented Design with Applications*, Benjamin/Cummings Publishing Company, Redwood City, California, 1990.
3. Bailin, S.C., "Object-Oriented Requirements Specification Method," Communications of the ACM, May 1989, pp. 608-623.
4. Berard, E.V., *Essays on Object-Oriented Software Engineering Volume 1*, Prentice-Hall, Inc. 1993.
5. Jordan, R.D., Smilan, R.E., Wilkinson, A.C., "Accelerated Project Cycle Using Object-Oriented Requirements," AT&T Software Symposium, Oct. 1993.
6. Beck, K. and Cunningham, W., "A Laboratory for Teaching Object-oriented Thinking," *OOPSLA-89 Proceedings*.
7. McClure, C. *The Three Rs of Software Automation, Re-Engineering, Repository, Reusability*, Prentice Hall, 1992.

(Manuscript approved November 1993)

__Yogeesh H. Kamath__ is a distinguished member of technical staff in the Network Management System Development Division at AT&T Bell Laboratories in Columbus, Ohio. He is responsible for the planning and development of Service Management System (SMS). He has a B.E. degree in electronics and communications from the University of Mysore in Mysore, India, and an M.S. degree in computer science from the University of South Carolina in Columbia. He joined the company in 1986.

__Ruth E. Smilan__ is a member of technical staff at AT&T Bell Laboratories in Columbus, Ohio, and is a software developer on the Call Attempts Data Collection System (CADCS) Team. She joined the company in 1983. She has a B.S. degree and an M.S. degree in computer science from Ohio State University in Columbus.

__Jean G. Smith__ is a technical manager in the Network Management Systems Development Division at AT&T Bell Laboratories in Columbus, Ohio. She manages the Service Management System (SMS) project. She has an M.S. degree in computer science from Rutgers University, New Brunswick, New Jersey, and a B.S. degree in mathematics from Ohio University in Athens. She joined the company in 1972.