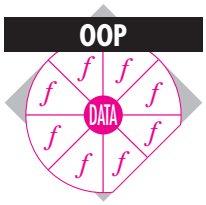


OO Programming Utilized by Rochester Lab

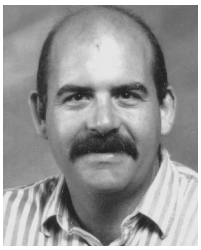
By Bill Berg, Chris Jones, and Mike Tomashek

Object-Oriented (OO) programming and C++ were used extensively in the effort to produce a level of System Licensed Internal Code (SLIC) to run on the RISC-based AS/400®. This article[1] discusses the justification, experience, and early results of that project.

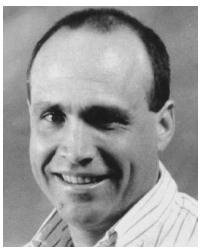
The Need to Shift SLIC to OO

As the IBM Rochester Lab approached moving the AS/400 operating system from the IMPI processor architecture used since its introduction to a new PowerPC-based RISC architecture, many changes were required in the lower layers. A comprehensive assessment of our strategy yielded these observations:

- A major design impact was evident for low-level software components that are hardware-dependent by nature.
- Storage Management, the Bus Manager, Machine Indices, and common functions like exception management, tasking, and queuing were impacted to some degree.
- Traditional processor microcode (Horizontal Microcode, or HMC on the AS/400) would not exist on a RISC system, so this had to be implemented as part of the new operating system.
- The existing Licensed Internal Code (LIC) was de-evolving due to frequent change and upgrade. Programmer productivity and quality measurement numbers would be lower without some technology change.



Bill Berg



Mike Tomashek

Rewrite or Port

There were two ways to attack this problem:

- Redesign/rewrite the low-level components from scratch, expecting a more extensible design and implementation. (Rewriting in a new language while keeping the old structure was not considered acceptable.)
- Attempt to port these components, with minimal changes to accommodate the new hardware, to bring up the system quickly. This, therefore, meant retaining existing data structures and control flow.

In either case, components not affected by the processor change would be migrated with as little modification as possible; for example, database, communications, and so on.

There were questions as to the feasibility of any redesign at all. The AS/400 was strongly based on design elements brought forward from the S/38, and essentially had 15 major functional releases that had not only added functions, but also had been performance-tuned. To many, it was inconceivable that major portions of this software could be rewritten in just two years.

Training: 'In-Demand' Skills Enhancement

There are ample warnings in print about the fallacies of trying to do your first OO project on a short schedule with inexperienced developers.

[1]Reprinted with permission from the Spring 1994 issue of *ASsociation/400*, the newsletter for the Association of AS/400 Business Partners.

We hoped our case would be different for the following reasons:

- The technology had matured since the initial horror stories. There was a better set of information available. Grady Booch, Bjarne Stroustrup, James O. Coplien, Scott Meyers, and Stanley B. Lippman all had excellent books out, some in second editions.
- The language was "ready." Templates and exceptions were approved and would be available in our Toronto-sourced compiler in time to meet our needs.
- It was possible to hire quality consulting skills. And, we did just that! We hired a solid consultant who put our people through six weeks of intensive training. When not teaching, he sat through many whiteboard talks while those of us with some self-taught OO/C++ skills bounced fledgling designs off him. These sessions yielded diagrams and class descriptions for our recent class graduates. The synergy of domain experts working with an OO design and a development expert is a common experience of the SLIC project. We started with a consultant, but now the most sought after internal skill is the ability to leverage domain expertise to OO design through the "knowledge map" of a more experienced OO person. Building these OO design skills is an important output of the SLIC project.

The half-life of new OO/C++ knowledge appears to be about three weeks. New converts must immediately code and design. The switch to "thinking objects" is hard. We still don't have a magic bullet for getting there. We observed that if ten people hammered out a project, two or three would become OO designers—maybe!

This education also contributed to morale. Our 150 people benefited from cutting-edge design/coding knowledge—highly marketable skills. More importantly, these skills are critical to our continued development efforts and our support of independent developers in their OO efforts.

Background

Both the AS/400 and the new RISC machine divide the operating system in two pieces: OS/400® and SLIC. (See Figure 1.) SLIC supports an architected interface for OS/400 called the Machine Interface (MI), which is effectively a high-level instruction set. This MI instruction set is targeted by the compilers rather than the actual hardware instructions. (SLIC includes a translator which converts MI programs into a combination of hardware instructions and SLIC calls.)

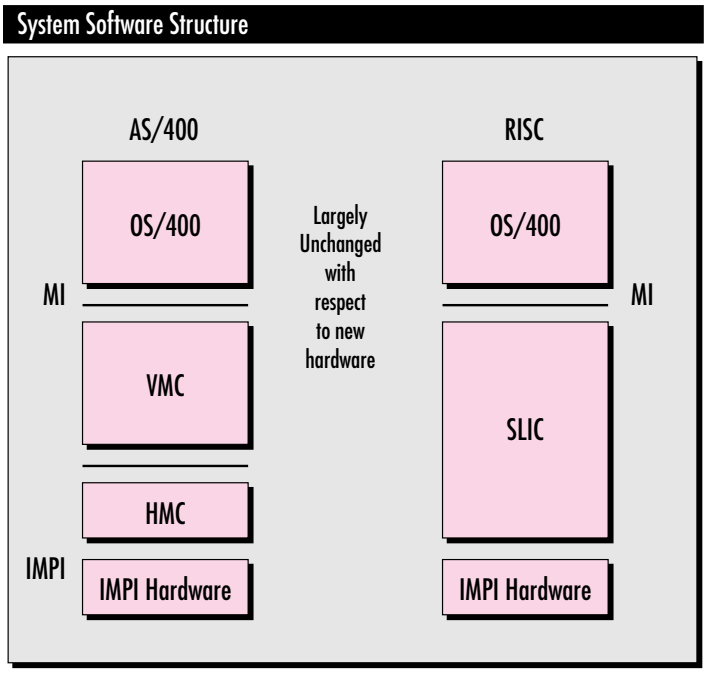


Figure 1. System software structure

The AS/400 is 'object-based': its objects are Abstract Data Types (ADTs) from an MI perspective. The MI provides Application Programming Interfaces (APIs) to represent a wide range of facilities and functions. For example, queues, programs, user profiles, and all I/O facilities are represented by these ADTs in the existing AS/400 MI. To the MI, these are real ADTs consisting of data and code. But, below the MI, they are implemented as global data structures that SLIC components are free to read and modify.

The experience of dealing with this encapsulation of APIs at the MI and maintaining compatibility for code at the MI was useful. Existing MI objects provided no experience with inheritance or polymorphism, however, so this still needed to be learned. Since we added the C++ code into SLIC mostly from the bottom up, we gained experience in interfacing existing functional decomposition-style code with new OO/C++ code. In general, this technique was a positive experience. Typically, the old code is improved since a macro or call is introduced to interface to an abstract C++ facility, and often a large chunk

of existing code that exhibited inappropriate coupling with another component is removed. We have less experience with wrapping existing function with objects, but this was done in some places successfully. The existence of a firm base and firm target definitely assisted this project. At the simplest, the mission could be described as "run the MI on the RISC processor." Much more was actually involved, but that core requirement provided a clear anchor against "creeping elegance." There was always existing reality (existing code interfaces/MI) against which we could test new abstractions.

The Process

First, we benefited from the following:

The ability to iterate in design is a cornerstone of OO.

- Teams naturally formed that paired a domain (AS/400) knowledgeable veteran with a UNIX- and C-literate new hire.
- Domain experts with OO and C++ knowledge were given fewer assignments to free them to consult with those less skilled in these categories.
- Everyone participated in coding. A representative from each major component formed a "petal" team that met weekly to make broad policy and direction decisions. These meetings allowed the team to raise their heads above water to understand what was really happening at a project level.

The petal name comes from a chart that showed the key domain experts and OO skills assigned to various major componentry of SLIC (database, communications, storage management, I/O, and so on). The chart looked like a flower with the chief designer being in the center and the other component experts around the outside as "petals."

The ability to iterate in design is a cornerstone of OO. As we looked at the work, it was also clear that some clear deliverables needed to be created that allowed us to honestly gauge progress. The solution was "Bring Up Binds" (BUBs) which gave rise to the classic phrase, "This BUB's for you." Each BUB had a clear set of functions that let us iterate at the interface level by putting together a large set of objects and verifying/learning at early stages. The process included the following.

- **Early design**: This involved a short text design description, Booch diagrams, basic header files, some main classes, and key methods.
- **Interface availability**: Completed header files provided the complete interface required by a set of clients.
- **Function availability**: The developer unit tested the classes provided, and the client could reasonably expect the function would operate.

We started in June 1992, iterated within components through the Fall, and pulled together the first BUB in early December. It took nearly two full months to get this bind to the point that a screen was displayed by the infant operating system. Much time was taken scaffolding and creating functions we discovered we needed, not due to bugs. The type checking provided by C++ is a major benefit in getting initial versions of code running. C++ does not intrinsically assist with issues of concurrency, assumptions about side effects made in old code, or in failing to set a pointer correctly. The code must be structured so that type checking is utilized as much as possible, and to leverage reviews and testing, to ensure other problems are caught.

The following points are important to the construction as well as the future of OO systems:

- Functionality must be staged through the delivery of objects to at least minimally meet the contract provided by their interface with a simple implementation.
- At the same time, the implementation of these newly delivered objects must be able to be upgraded at a later date without breaking clients.

The Results

The SLIC project has over 750,000 lines of code and 6,000 classes in C++. Most of this code is very low level and performance sensitive. The following key elements were converted to C++:

- **Storage Management**: This code provides the abstraction of Single Level Storage (a unique feature of the AS/400 architecture).
- **I/O Support**: The base support to communicate the buses and manage hardware resources was converted to C++. The SLIC support for DASD, tape, and optical devices were also converted.

- **HMC Replacement**: With the advent of RISC processors, many higher level operations provided by the HMC (queuing, tasking, hold record support) needed to be coded in SLIC.
- **Service Code**: Support for screens and tools, provided with SLIC to service the machine.

Execution Experience

Not surprisingly, there were performance challenges. (This is common and predictable.) In general, the C++ code has been very "tune-able." The ability to change implementations without changing the interface is extremely powerful. C++ provides solid mechanisms like in-lining methods and object containment that are good for performance tuning, and the ability to override new and optimize storage allocation is always a good tuning knob. It is far easier to tune a well-understood design.

OO—A "Do as you learn" Technology

OO is one of the first effective ways developers have had to create and communicate abstractions. As the project progressed, we had to stop thinking about how we did our job and start thinking about what we did. It was common to spend much time trying to "OO-ize" a data structure that had been in the solution space for years, only to discover that the data structure may be unimportant, or even inappropriate.

Through the project, the scope of tools has increased. First, we only had a few re-usable classes of the stack, string, list, or map ilk. Then, we acquired idioms like when-done which allows a possibly asynchronous server to call back to the desired method after a completed operation. This idiom was extensively used. Others include:

- **Constructor branch table**: Allows the construction of a subclass of a known base without changing to existing code. Support for the new subclass just needs linking into the system.
- **Proxy objects** (also ribbon cables): Allows an object to stand in for another used in lieu of smart pointers when the real object may be destroyed. It can be aware of its proxies and inform them. Clients deal only with proxies. As the project progressed, we heard of frameworks and began to see them in the design. There are two views of a framework.
  - **The client view:** Interfaces the client sees. Normally very stable abstract base classes.
  - **The framework extender view:** When extending the framework, developers will subclass from the appropriate classes to create a new function. This is where the "don't call us, we'll call you" framework designation comes from. As a new subclass is produced, the code will simply run at the right time when the framework is used.

In the I/O part of the project, a framework evolved to add new I/O hardware to the system. Clients work with a set of abstract base classes, and when a new piece of I/O gear is added, we

"The pundits have said you must sneak up on OO, but our experience says the opposite. We had tried 'stick your toe in' approaches before, with non-conclusive results. If the project succeeded, the naysayers claimed, 'It was an isolated, non-real-world set of code.' But, if there were problems, the OO champions claimed, 'There were too many interactions with old code.' This project is large, performance-sensitive, and interacts heavily with existing code. We've come far enough to comfortably say this project will be a big score for OO."

—Mike Tomashek

"A good bridge designer respects the properties of his materials and uses them to enhance the design. Similarly, a good software designer builds on the strengths of her implementation language and—as far as possible—avoids using it in ways that cause problems for implementers."

—Bjarne Stroustrup, designer of C++ and author of *The C++ Programming Language*

"In our experience, the design of classes and objects is an incremental, iterative process. Except for the most trivial abstractions, we've never defined a class exactly right the first time."

—Grady Booch, author of *Object-Oriented Analysis & Design with Applications*

"The knowledge acquired on this project is invaluable; there is no substitute for putting something into production that enhances our AS/400 OO development offering. This also puts us in a better position for joint projects with outside organizations using RISC and C++."

—Bill Berg

Useful Vocabulary

Abstraction—Process of making complex ideas and structures more understandable by removing the detail and generalizing their behavior. An abstraction is, in many ways, the antithesis of hard-coding.

Class—Templates for defining kinds of objects; usually defined as special cases of each other, organizing information about objects in a natural, intuitive manner.

Framework—A set of classes that provide a set of services for a particular domain.

Idiom—A standard way to solve a class of problems in a given programming language.

Inheritance—When classes are defined in a hierarchy, special cases share all the characteristics of their more general cases.

Iterate—The ability to successively refine a design by improvements.

Polymorphism—Allows methods to be written that generically tell other objects to do something without requiring the sending object to have any knowledge about the way the receiving object will understand the message.

provide a set of subclasses to allow the new gear to operate. Once added, these classes are used automatically by the I/O framework when an operation is requested at the new device.

OO is moving forward as higher level concepts are built on top of the paradigm and languages. Idioms form the first step. An evolving second step is of design patterns. The current top of the abstraction heap is the framework, offering exciting promises.

Conclusion

We are very pleased with our first large scale experience with OO and C++. After some initial performance and object code size scares, SLIC is stabilizing. We are working to create more code commonality. Here are some observations:

- An OO project can be started from scratch assuming solid technology transfer. OO technology is mature enough to use consultants and printed information to bootstrap a project.
- Strong education pays royally, but the new skill must be used right away, and training must be ongoing.
- Start coding quickly—just do it! And iterate, iterate, iterate!
- Explicit, frequent checkpoints showing success/failure are important.
- C++ performs in low-level OS situations.
- Idioms are powerful as both design and coding constructs.

Bill Berg, *IBM Corporation, Rochester, Minnesota. Mr. Berg is a programmer in the advanced technologies area of the Rochester lab, exploring the usage of Taligent® and microkernel technologies. He helped design the OO framework used to attach I/O gear to the AS/400 RISC machine. His interests include C++ and the application of OO technology to operating systems.*

Chris Jones, *IBM Corporation, Rochester, Minnesota. Mr. Jones works as a development manager in the AS/400 Division.*

Mike Tomashek, *IBM Corporation, Rochester, Minnesota. Mr. Tomashek is a product manager in the AS/400 Division. He has managed SLIC from its inception. For the past ten years he has designed and managed embedded controllers and operating systems.*

Upcoming Object-Oriented Programming Classes

Date	Course	Code	Location
12/05-12/09	Object-Oriented Programming and Design with C++	(Q1073)	Research Triangle Park, NC
12/12-12/16	Object-Oriented Programming and Design with C++	(Q1073)	Charlotte, NC
12/12-12/16	Object-Oriented Software Engineering	(N1498)	Austin, TX
12/13-12/15	SOMobjects Toolkit Developer's Workshop	(N1602)	San Jose, CA
01/16-01/20	Object-Oriented Software Engineering	(N1498)	San Jose, CA

For additional information call, 1-800-IBM-TEACh