Object-Oriented Development at Brooklyn Union Gas

JOHN DAVIS, *Andersen Consulting*
TOM MORGAN, *Brooklyn Union Gas*

◆ *Using dynamic object-oriented features, a mainframe implementation of a Smalltalk-like execution environment supports a critical application and can accommodate change.*

Object-oriented principles can resolve long-standing problems of software construction and evolution. The Brooklyn Union Gas Company found this to be true when it replaced its outdated customer information system with the new Customer-Related Information System developed using object-oriented techniques.

Brooklyn Union's decision to use object orientation in CRIS-II simplified development, improved communication, and motivated personnel.

Object-oriented concepts focused on the model, or simulation, aspects of system building and let us base the system architecture on the essential aspects of the underlying problem domain. System developers and users could speak the same language because the new system modeled the company's business environment. We could design technical and organizational artifacts as replaceable components.

CRIS-II's implementation in a Smalltalk-like execution environment with object-oriented features lessened coupling between components, compared with other object systems. Though it is fully object-oriented, the system accommodates many traditional, non-object-oriented components such as a relational database manager, an on-line transaction manager, a batch report writer, and a user-interface dialogue manager. Also, CRIS-II interfaces to more than 15 other applications. Two years of maintenance and enhancements since its start-up in January 1990 have demonstrated that we can naturally adapt its object model to change and to incorporate more applications.

APPLICATION ARCHITECTURE

System architectures are often described in terms of today's artifacts for re-

0740-7459/93/0100/0067/$03.00 © IEEE

alizing the architecture. In contrast, the CRIS-II architecture is structured in three layers that are derived from essential aspects of the underlying problem domain. The interface, process, and business-object layers address, respectively, the "when," "what," and "how" in the gas utility environment.

The architecture model uses layers to package the system. The layers of the architecture isolate functional concerns from the user interface and from physical representations of data. Separation of interface concerns from other layers in the architecture allows business functions to be used from multiple user interfaces.

The model avoids brittleness — it can adapt to evolving company policies and practices over time and still retain its conceptual structure.

The use of object-oriented techniques in all layers of the architecture provides a convincing and sustainable model for representing the system. The use of objects within the framework of the architecture allows a very direct representation of the problem domain in the implementation.

OUR SYSTEM HAS AN OBJECT PERSPECTIVE BUT USES NON-OBJECT-ORIENTED SERVICES.

Interface layer. The interface layer connects the system with its users. It contains on-line dialogues, batch processes, interfaces to other applications, and internal, system-triggered events. Interface components are not always objects, but they have access to objects and message-passing facilities.

The interface layer triggers business functions in the process layer by message sending. Because each business function is implemented as a first-class object, any interface component has potential access to every business function in the system, permitting the widespread dynamic sharing of large units of application function.

Process layer. The process layer is built out of objects called function managers. The principal behavior of a function manager is to carry out the response to events received by the interface layer. This behavior is described by a script-like control structure, which expresses the shallow logic used to describe the system's response to events.

The script is implemented by message sends to objects in the business-object layer. The function-manager script deals with "what" to do, delegating the "hows" of the event response to objects in the business-object layer. The function manager coordinates actions among the many business objects that are involved in the event response. It also provides additional behaviors to provide accountability and control and to deal with any side effects of the event response.

During the design of CRIS-II, some developers argued that the process layer was unnecessary because business objects could implement its behaviors. However, we included the process layer in the architecture because it narrows the interface from the objects at the interface layer to the objects in the business-object layer. The function manager objects hide inessential detail and supply large units of reusable business function to the interface layer.

Business-object layer. The business-object layer consists of objects and methods that model a specific business. The object behaviors simulate the enterprise with their knowledge of "how" to implement function. Both the interface and process levels access object behaviors through messages.

The business-object layer has sublayers. A simple example is its response to the arrival of a new meter reading passed from the interface layer:

```
What do you do with a meter read?
  If the read is suitable for billing, then
    render a bill.
    How do you know a read is
      suitable for billing?
      How do you render a bill?
```

Successive answers to "How do you...?" questions produce layers of functions that produce additional layers of functions within the business-object layer, beginning with business policy and practice and ending with the technical details of the current implementation. Fundamental to the design is procedural abstraction or separation of concerns. We adopted object-oriented techniques to realize the system architecture directly.

TECHNICAL DESCRIPTION

CRIS-II operates on an IBM mainframe. It uses the MVS/ESA operating system, the DB2 relational database manager, the CICS (Customer Inquiry and Control System) on-line transaction manager, and PL/I with object-oriented extensions as its principal language. CRIS-II is a small to medium-size commercial system in this environment.

According to Peter Wegner's definition,[1] CRIS-II is object-oriented: It supports object functionality, object management by classes, and class management by inheritance. Its customized technical model borrows much terminology and many architectural concepts from Smalltalk-80[2] and Brad Cox's work.[3] But CRIS-II is really a hybrid: It retains an object perspective but uses the non-object-oriented services of MVS/ESA, DB2, and CICS.

General characteristics. CRIS-II provides for untyped variables and dynamic messaging, uses encapsulated object descriptions to lessen the effect of change, and carries class information into the execution environment. The execution environment most nearly resembles Smalltalk:

- An object is an instance of a class. It inherits instance variables and behaviors from a superclass. All objects are descendants of class Object.
- An object has both class and instance methods.
- Objects' behaviors are implemented as methods written in PL/I. A method is a separately compiled, executable unit. The necessary bindings for a method are completed at execution time.
- A method is known to the environment by its "selector" name.
- Classes and their methods are repre-

sented in an entity-relation-attribute dictionary, extended to include design items for an object-oriented system. The dictionaries are loaded into memory on demand. In memory, the dictionaries are first-class objects.

- The instance variables of a class may be changed without recompiling the class's or descendant classes' methods.
- Objects communicate through message passing. To support polymorphic behaviors, message passing is dynamic. Second-order messaging, the equivalent of the Smalltalk "perform," is supported (and used frequently). An object sends itself a message using a "self" token and addresses its immediate superclass using a "super" token.
- Atomic items, such as strings, numbers, and characters, are represented in memory as PL/I data types rather than as objects.

CRIS-II consists of 8,600 methods and 650 classes. The methods contain 70,000 message-sending locations.

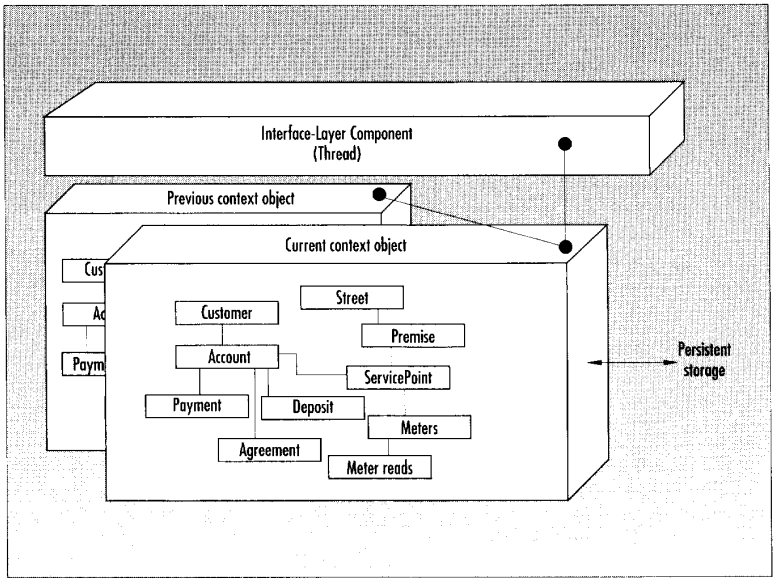


Figure 1. The object memory model for a thread. A thread sees objects through Context, a container object for application objects that it manipulates and accesses. Alternate objects can exist within a thread.

Object memory model. To fully support a multiple-user, transaction-oriented application, the object system assigns each user a processing thread for the duration of the user's session with the system. As Figure 1 shows, each thread has an associated instance of a Context object. Context is a container object that houses the objects the thread is currently manipulating and accessing. The Context object implements behaviors for managing storage. It also provides the basic memory-management services required for object management, allocation, and reclamation of storage from discarded objects.

The Context service can find all objects by class. This behavior is critical to supporting object identity as objects move to and from persistent storage. However, Context does not provide garbage collection. Object storage is allocated in fixed-size units, so unlike objects can reuse object space for fast object allocation. The object system is largely implemented in itself; however, we reimplemented the lowest levels of object storage in PL/I to improve performance.

As an application executes, it creates new objects and puts them in a Context object; Context grows and shrinks in response to object creation and destruction. Context relocates objects in memory to consolidate the memory it manages.

Transaction management. In addition to basic object-management services, Context enables transaction management by implementing basic versioning behaviors. As the application responds to the events it processes, it creates new objects and positions them in a Context object. Generally, new objects are initialized with data physically stored in a relational database manager. Each database row has an identifier guaranteed to be unique across the system.

Before introducing a new object, the behaviors that implement object persistence search the Context object to ensure that the object is not already present within it. This technique lets applications address objects by their unique identifier with complete assurance that the most recent version of the object is returned.

When it places a new object in Context, an abstract behavior saves the current values of the object's instance variables. Thus, an object can tell if it has been modified by comparing its current values with its saved values. This technique optimizes processing when objects are returned to the database manager. Also, the system can create detailed audit trails in a uniform way. System controls often dictate that "before" and "after" values of instance variables be retained in special Activity objects.

Many application exceptions and side effects are triggered when certain variables change. For example, a service-order status change may need to be communicated to other systems. Generalizing the identification of changes lets the system abstract side-effect and exception handling into a superclass for all persistent objects. This helps application maintainers, who may need to change side-effect definitions.

Multiple instances of Context may exist. By reinstating varying Contexts, the system can provide very powerful operations, including a full logical unit Undo to reset the state of all objects to the last checkpoint. A single processing thread can also pursue alternate transaction

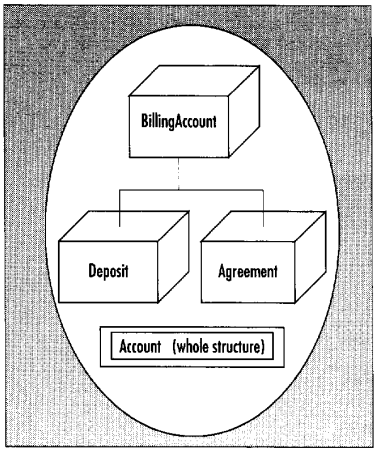


Figure 2. A partial view of the whole-part structure of an Account. In the BillingAccount object, the Deposit and Agreement objects are represented as instance variables that are ordered collections.

paths.

The external interface layer controls the logical unit of work. When a business event completes, the interface component informs its Context object. The Context object iterates over its components, telling each one to save itself. Objects that have experienced real changes write themselves to the database. Each object understands its denormalization requirements and foreign key relations, and optimizes its write operations appropriately. Overall, the Context object ensures that database-manager operations occur in a consistent sequence to improve database-manager response and to avoid deadlocks, a nagging problem in maintaining complex shared structures in relational systems. Following these actions, the Context object begins processing side effects for the current event and, as a last step, makes the saved and current values of each object the same.

At the end of a logical unit of work (when a company representative completes a telephone call with a customer, for example, or when a billing function creates a bill), the system no longer needs many of the objects in Context: In these situations, the Context object is sent a "clear" message to erase most of its objects and reclaims their storage space. Some objects, such as those that identify the current thread and its user, survive a "clear" message and respond only to the more forceful "purge" message.

The objects' ability to understand clear and purge messages permits specialization of behavior as objects are finalized. Thus, objects can be synchronized with resources outside the object environment's control (the transactions and data resource managers).

Reclaiming storage on the basis of a logical unit of work supplies a rudimentary garbage-collection mechanism for short-lived transactions but still requires the explicit destruction of objects. This was the largest single source of technical program errors. A real garbage collector would have been invaluable.

Database-manager interface. CRIS-II's persistent data storage consists of approximately 150 relational tables managed by DB2. The tables occupy approximately 100 Gbytes of disk space. The largest table contains approximately 75 million rows of information.

Each business-object class has, in most cases, a one-to-one correspondence with a relational table, in which rows in the table correspond to instances of the object. But an instance of an active business object has a more complex structure than its corresponding database row. As Figure 2 shows, it contains additional instance variables that represent the object's whole-part structure. Extensions to the accessor protocols also let clients of objects retrieve the values of instance variables as the values were at the start of logical application units of work.

Persistent objects, those business objects that have corresponding relational tables, are descendants of DB2Object, an abstract class that implements generalized data access behaviors using Structured Query Language, a relational data-manipulation language.

Because more than 100 SQL operations are issued each second in the on-line system, each concrete class of DB2Object has methods for performing these same operations with static SQL. In most cases, a repository provides the code for these methods, but we sometimes handcrafted methods to improve access efficiencies. During the initial development, we used default behaviors because

- they generated dynamic SQL statements, which provided flexibility and lessened the need for changes in the database schema, and
- they let each developer have his own relational tables to reduce the coordination required among developers when testing.

The implementations in the concrete classes are a concession to performance that would be unnecessary if the data manager were more sophisticated.

The database-manager interface is a sublayer of the business-object layer and separates application concerns from the technical details of the current implementation. It is a low-level abstraction in the architecture, and most business behavior is insulated from it. We can easily substitute alternate technical implementations. One operating mode substitutes basic file operations for the data manager.

THE EXTERNAL INTERFACE LAYER CONTROLS THE LOGICAL UNIT OF WORK.

Object navigation. Navigational messages are used to traverse the whole-part structure of an object. Each object understands its component parts and implements a selector for returning each part. For example, in Figure 2 the BillingAccount object's whole-part structure includes the Deposit and Agreement objects. To the BillingAccount object, these two parts are instance variables that are OrderedCollections associated with it. BillingAccount's methods return to the environment its Agreements (in response

to the selector "agreements") and its Deposits (in response to the selector "deposits") as OrderedCollections.

In relational tables, foreign keys point to the parent in a whole-part relation. In this example, Deposit is part of BillingAccount, so storing the BillingAccount key in the Deposit row's foreign key column makes that Deposit row part of a given BillingAccount. These rows are accessed with straightforward SQL statements.

CRIS-II uses a "lazy initialization" technique: When an object is first initialized, the instance variables that correspond to its parts are assigned null values. The first time a message is sent for its contents, that part is initialized.

Objects also include a behavior to navigate to the parent or "whole" of the whole-part relationship. For example, a Deposit object could access its parent BillingAccount object.

In this way, CRIS-II's navigation behavior assembles, from related encapsulated objects, the rich data states an application function requires. Other commercial applications based directly on relational databases contain bulky code to gather this much data from the relational tables. CRIS-II's approach reduces coupling and eliminates most of the data-access code.

Typing. CRIS-II permits, but does not require, object type specification for a method's variables. Declaring an object's type is a promise that the variable is at least of the declared type in the inheritance hierarchy. Type declarations improve performance for accessors (the behaviors that allow clients to set and return the values of instance variables). A review of our record of program errors shows that eliminating mandatory typing did not cause a significant number of runtime errors.

Typing of variables is only one example of more general schemes for compile-time assertions. The hope is that the compiler can prove (or disprove) the assertion for efficient, early detection of error. With typing of variables, a program *may* be correct, if and (unfortunately) only if the values present in the variable are of the specified type. The "only if" clause puts costly limitations on the program's flexibility. Nothing in our experience with CRIS-II shows that object typing makes error-detection efficient. On the contrary, the inflexibility introduced by mandatory typing of variables is harmful. A better approach is to regard typing of variables as one among a variety of assertions you can make at some point in the development cycle, and one of a variety of techniques to ensure efficient error prevention and elimination.

DEVELOPMENT ENVIRONMENT

The fine granularity of object-oriented applications and our system's size made extensive development tools essential. You cannot do object-oriented development on this scale with just a compiler and an editor. When we constructed the tools early in the project, we used many traditional mainframe components, including IBM's ISPF, DB2, and PL/I preprocessor. Other development software includes a screen painter, a code generator, and a report writer.

At the center of these tools is an extensible entity-relation-attribute dictionary with information about the applications that comprise CRIS-II and its environment. The central dictionary holds and manages all system components.

For the object environment, it provides facilities for browsing object descriptions and methods. It provides cross-referencing of messages, so a developer can locate all methods that are senders of a particular message or all methods that implement a particular message.

Frequently, developers locate behav-

WHY BROOKLYN UNION NEEDED AN OBJECT-ORIENTED SYSTEM

Brooklyn Union Gas is a utility company that distributes natural gas to commercial and residential customers in the New York City boroughs of Brooklyn, Staten Island, and Queens. Its customer information system manages the major revenue cycle, which includes field-service orders, cash processing, credit and collections, meter reading, billing, and general accounting. Each year, the system processes a one-billion-dollar annual revenue stream.

Like many utilities, Brooklyn Union first automated customer information management in the 1960s and then reimplemented the system during the early 1970s. In 1985 and 1986, the company planned to replace its early 1970s vintage customer information system, CRIS-I, using state-of-the-practice techniques (structured and data-driven design techniques and integrated CASE tools). As this effort went forward, the project seemed in danger of duplicating the old system's monolithic characteristics. Structured analysis showed that the system's complexity was much larger than anticipated. Something was wrong; it was time to rethink the problem. We reflected on the nature of business systems themselves and concluded that they are best understood as simulations of the enterprise and its environment.

We decided that the state-of-the-practice techniques in 1987 were not adequate to address this understanding. An object-oriented design and implementation that directly represented the problem domain in the implementation would best serve the company's needs for its primary information system.

Development and implementation of the new application, CRIS-II, took place between 1987 and 1989, at its peak involving 180 Brooklyn Union Gas and Andersen Consulting employees. The system took 365 work years to develop, plus an additional 24 for user training.

REFERENCE

1. E. Andersen and B. Konsynski, "Brooklyn Union Gas: OOPS on Big Iron," Harvard Business School case study N9-192-144, Boston, 1991.

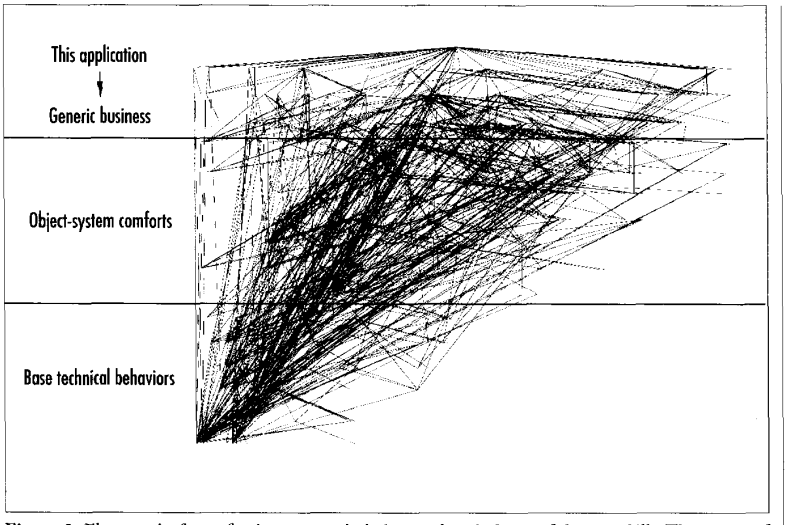


Figure 3. *Flow graph of transfers between methods during the calculation of three gas bills. The pattern of reused behaviors repeats at all four abstraction levels.*

iors using a cross-class browser, which locates behaviors by matching portions of class and behavior names. This is more useful in practice than searching the fixed class hierarchy.

During the development of CRIS-II, all project team members had access to the dictionary and used it to communicate design information and changes. After installation, the dictionary became the central facility for the maintenance and extension of CRIS-II.

In addition to performing basic object-oriented development functions, the dictionary maintains links between the interface components and the events they process, as well as the objects responsible for implementing the response to the events. The dictionary tracks system test scenarios and stores scripts to execute them. It supports code generation, testing tools at the programmer and integration level, and configuration management for object behaviors.

Our work on CRIS-II shows that extensive training in object techniques is required only for a small number of highly skilled application developers. Most application developers should work in highly structured frameworks in which they do not have to understand the techniques completely.

PERFORMANCE

CRIS-II operates from a central computer and supports more than 400 concurrent users. Every evening it processes more than 100,000 business transactions. Overall, it uses as much or less processor capacity and more memory than traditional implementations in this environment:

- The combined path length through the transaction and database manager is about 1.1 million instructions for the average interactive exchange. The system delivers subsecond response in such exchanges.
- The system's interactive component processes 10 to 12 CICS transactions per second and uses approximately 12 MIPS on an IBM 3090J processor.
- Three million SQL statements are executed per day.
- The working set size for the object system and the transaction and database managers is about 120 Mbytes.

When busy, the system completes 4,000 message sends per second. Message passing is dynamic, but we optimized it to recognize that object types and selectors have a tendency to remain constant at a given calling point. With this optimization, less than two percent of processor time is spent in messaging activities, yet the system retains its flexibility.

Obviously, message sending takes longer in a procedural language than does subroutine calling, but its value far outweighs its cost. A too-narrow focus on optimizing performance at very low levels can actually lessen the system's overall performance. For example, dynamic messaging lets our object system more effectively control the application's use of the database manager, which in turn provides overall performance gain.

We believe performance assessments should emphasize the life-cycle difficulties of maintaining tightly coupled systems over the technical problems of optimizing individual subroutine invocation.

REUSE EXPERIENCE

Our experience with CRIS-II for its more than two years of operation provides partial answers to questions about reuse in object systems and the differences between object-oriented systems and systems constructed using other techniques. Here we report measurements of code reuse and system behavior.

New dialogue example. A new dialogue added to CRIS-II provides four displays so that users can correct mismatches between field conditions and database contents. A new information-systems employee at the gas company — one not involved in CRIS-II's development — implemented the dialogue. When she started the assignment, she was unfamiliar with the available classes and behaviors. In addition, she had to create a new class of persistent object to support the dialogue, so the work was more complex than most maintenance and enhancement tasks.

After she completed the work, we counted the raw lines added to the system. This analysis showed that 2,000 lines had been added, but a total of 40,000 lines of business-object behaviors had been used to support the dialogue. In other words, the ratio of added code to reused code was

1:20. The 40,000-line count includes only behaviors that implement business function — we excluded technical behaviors for dialogue management and so forth.

Even this amount of reuse wouldn't be a win if had been very hard to find the behaviors to reuse. This wasn't the case. When she saw the total line counts, the programmer was surprised at the large number. Her impression of the job's size came from the lines she added, not the total lines.

This and other examples show the need for a new role in the development environment: software assembler. The software assembler would create new interface components and some shallow logic, and occasionally add an application behavior. Most of the necessary behaviors would be in inventory.

Systemwide example. To analyze behavior reuse from a different perspective, we summarized all additions made during a year. In calendar year 1991, CRIS-II was extended to deal with work-queue management, automatic collection-call handling, and automatic bill payment. Forty classes and 908 message-sending components were added. There are 3,344 message-sending locations within these components. Because the CRIS-II object system supports dynamic messaging, it is impossible to find all implementations that support the message-sending locations. However, by analyzing the source code we could trace 669 distinct methods. Of these methods, 446 were created before 1991. That is, of 669 behaviors required to support the work done in 1991, 446 (67 percent) were already in the behavior inventory.

TYPICALLY, BUSINESS SYSTEMS HAVE REUSED VERY FEW APPLICATION COMPONENTS.

The number of reused application components in business systems is typically very small. When reuse occurs, it is of minor or technical components (date-handling routines, common I/O routines). Our findings are significant because of the number of reused *application* components.

Billing example. Figure 3 shows a flow graph of all transfers among methods during the calculation of three gas bills. This is a dynamic view of behavior reuse in the running application during its response to the arrival of a meter reading. To produce the figure, a graph-layout algorithm processed the distinct pairs of method-to-method calls during the creation of three gas bills. Wherever a line starts or ends, there is a node representing a method invocation. The layout rules are as follows:

- Center a parent (a sending method) above all its children.
- Position a child (an implementing method) below all its parents.
- Working from left to right, never reposition a node. If a node has already been placed and is used again, just draw another line to it.

As a graphic, the figure is badly balanced: There is nothing in the lower right corner. This imbalance demonstrates the extent of component reuse. As the layout algorithm proceeds from left to right, reused behaviors result in a line that runs down and to the left. Methods in the upper right found fewer and fewer behaviors not already in the layout.

The graph shows very fine-grained reuse — the components being reused have an average size of 30 lines.

Because of the rule that places a child below *all* its parents, low-level or basic behaviors drift toward the bottom of the chart and high-level or application behaviors drift to the top. This corresponds to the system's internal layering of the "hows," progressing from how the business is to be done, down to how to do business with the current technical artifacts.

The node at the lower left corner is basicNew, the inherited behavior that handles the raw mechanics of object creation. This behavior is highly reused simply because the technicalities of memory management can be reused. Much more interesting are the left-leaning lines at every level in the diagram, indicating reuse even of high-level application behaviors. Furthermore, while there are fewer left-leaning lines at the top of the diagram, each reused component at the upper levels of the diagram represents a very large unit of reuse.

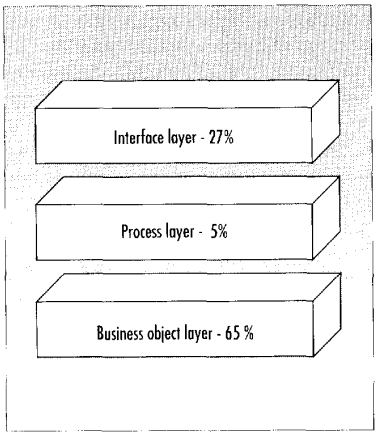


Figure 4. *Lines of code by layers in the application architecture model. In the business-object layer, the largest object contains six percent of the code. Technical objects represent less than 10 percent of the code.*

The extent of reuse across the system is even greater than this figure indicates, because it shows only one of many roots in the application. A search from an accounting behavior that appears in the upper third of this diagram showed that the behavior can be reached from 164 roots.

This pattern of fan into and fan out from highly reused behaviors repeats at all abstraction levels. Labels in the figure show this by indicating four general categories of behavior:

- specific business application (create a gas bill),
- generic business behaviors (apply a ledger entry),
- useful high-level technical behaviors (behaviors of collection classes and so on), and
- basic technical behaviors (like basicNew and other memory-management objects).

In a system with properly modeled ge-

neric business functions and robust support for object-oriented mechanics, you can develop and add specific applications rapidly.

Lines of code by layer. Figure 4 shows an analysis of lines of code by architecture layer. Business objects, which are implemented in a purely object-oriented fashion, represent more than half of the total. In the business-object layer, the largest object contains no more than six percent of the total code, showing that the application functionality is well distributed across the object set.

DEVELOPERS SHOULDN'T TRY TO SPECIFY THE SYSTEM MORE THAN A FEW STEPS INTO ITS LIFE CYCLE.

The process layer comprises five percent of the total, which shows that the architecture lets the application economically state "what" is to be done, based on the inventory of behavior in the business-object layer.

The hybrid interface layer is bulky, yet the average size of a component in this layer is still only 100 lines. This layer has more than 1,000 components. The interface layer was built using integrated CASE tools, which tend to generate duplicate code in each component. The layer would be much smaller if these components were objects that could reuse code through inheritance.

CRIS-II is an example of a commercial data-processing system that is critical to the economic viability of the enterprise it supports.

Systems like CRIS-II are better thought of as small economies, rather than large computer programs. As they are used and evolve, their trajectory through a design space is the product of their own history and incidents in their environment. Such systems are so large they actually alter their environment.

System developers must recognize that specifying more than a few steps ahead in this trajectory is simply impossible. The essential requirement for business systems is orderly evolution over time, with minimal constraints on the design trajectory.

If the system models the real environment, and the technical artifacts it needs are well insulated from each other, changes to the system are likely to exhibit "proportionate effort" characteristics: Things easy to ask for will be easy to do.

For CRIS-II, we extended an entity-relation-attribute dictionary to include design items for an object-oriented system and then developed tools that operated on the data. We think a better technique is to use the object system itself to contain its own development, design, and analysis description. Our experience shows enormous benefit from having fully bootstrapped environments. In CRIS-II, it would have been better simply to extend the object to fully maintain its own metadescription.

In CRIS-II, the class description and inheritance and messaging mechanisms are constructed within the object system they describe. However, the mechanisms are outside the extended PL/I language definition that allows methods to be coded and messages to be sent.

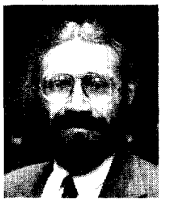
This separation seems very generally applicable. If object-oriented extensions are desirable in existing languages, developers should consider constructing them using a language-neutral base for describing the class structure and messaging mechanism. They should limit language extensions to bindings to the class descriptions, method implementation, and behavior invocation. The work on the CLOS (Common Lisp Object System) metaobject protocol[4] shows how this might be done, as does the System Object Model in IBM's OS/2 Version 2.0.[5] ♦

REFERENCES

1. P. Wegner, "Dimensions of Object-Based Language Design," *Sigplan Notices*, Nov. 1986, pp. 168-169.
2. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Mass., 1983.
3. B. Cox, *Object-Oriented Programming — An Evolutionary Approach*, Addison-Wesley, Reading, Mass., 1986.
4. G. Kiczales, J. des Rivières, and D. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, Cambridge, Mass., 1991.
5. *OS/2 Technical Library System Object Model Guide and Reference*, Document S10G6309, IBM Corp., Armonk, N.Y., 1991.

John Davis is a partner in the Advanced Systems Group of the New York Office of Andersen Consulting. He is interested in the development of tools and architectures for modern business systems.

Davis received an MBA from the University of Michigan in quantitative methods and statistics.



Tom Morgan is manager of information technology development at the Brooklyn Union Gas Company.

Morgan received a BS in mathematics from the University of Washington. He is a member of the IEEE and ACM.

Address questions about this article to John Davis, Andersen Consulting, 14th Floor, 1345 Ave. of the Americas, New York, NY 10105; or to Tom Morgan, Brooklyn Union Gas Co., 13th Floor, 1 MetroTech Center, Brooklyn, NY 11201; Internet 74155.416@compuserve.com.