The Promise of Distributed Business Components

Dave K. Kythe

This paper discusses the advantages of replacing hand-crafted software with reusable components as a solution to the software crisis. *Object-oriented programming* provides insights on how to build components. Because components must have distributed implementations and well-defined interfaces, both the Microsoft Component Object Model (COM) and the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) models are described as distributed object architectures to support reusable components. A transaction processing monitor is also necessary for accessing business logic and the information contained in relational databases. Components are composed of object-oriented frameworks based on models of the problem domain or business.

Introduction

The problem with current software is that it is extremely labor intensive. It takes too long to build and costs too much; it is almost always late and over budget; it is inflexible and difficult to repair; and the final results do not deliver what the customer really needs. This paper describes a possible solution to the software crisis by replacing hand-crafted software with reusable components. Two of the major architectures for component development (the Microsoft Component Object Model [COM] and the Object Management Group [OMG] Common Object Request Broker Architecture [CORBA]) are discussed. A methodology for building components via object-oriented frameworks is also presented.

Component-Based Software Development

The following subsections discuss four key aspects of component-based software development: key principles, new industries, new tools, and object-oriented technology and components.

Key Principles. The term "component" has different meanings in different situations. Thus, for purposes of this discussion, the following definition is provided:

A component is a reusable software element that can be used by developers to assemble an application.

The component-based software development process is composed of components, methodologies, and tools for building software. For companies that can take advantage of this process, component-based software development is expected to cut development time in half. Customer needs, represented as business models, can be fed through tools to develop a design and generate code for the application. The application can be built by assembling components using high-level logic both to use component functionality and to manage the relationships between components.

Components allow the design of large systems by providing the following two important benefits[1]:

- Reduced development time (cost) by having to write less code when reusing components, and
- Higher quality by using well-tested components.

The fundamental assumption about components is that purchasing them off the shelf makes more economic sense than

building them from scratch.[2] Libraries of standard, reusable information systems components including architectures, designs, frameworks, code, test suites, and user interfaces will be available from a variety of the following sources:

- External commercial software component foundries or independent software vendors,
- Existing "legacy" systems wrapped into reusable components containing business rules and expert knowledge, and
- Internal component development groups for parts not available commercially or for parts required for a competitive advantage.

New Industries. The systematic reuse of components will create a new industry of software component developers producing reusable components for the market. Developers in the software industry will also separate into the following three segments:

- *Solutions builders,* producing applications for specific business needs;
- *Component builders,* producing components for reuse within a specific industry or vertical application domain—for example, components for banking, insurance, utility companies, and health care industries; and
- *Technology builders,* producing technology that spans multiple industries or horizontal application-independent domains—for example, operating systems, networking, graphical user interfaces, and databases.

New Tools. A new generation of *computer-aided software engineering (CASE)* tools is needed to define the application logic that governs the interaction of components. The strength of CASE tools is that visual programming techniques are used to assemble applications by manipulating graphical icons and not code. Currently, few CASE tools can manipulate components. This situation will change, however, as new tools for application developers are introduced.

Object-oriented CASE tools either translate object diagrams produced by object-oriented methodologies into code or reengineer existing code into object diagrams. Some CASE tools include a repository for storing and retrieving components. The development methodology or process for building components and applications is critical, especially object-oriented development methodologies[1,3].

Panel 1. Abbreviations, Acronyms, and Terms

CASE—computer-aided software engineering
COM—Component Object Model
CORBA—Common Object Request Broker Architecture
DLL—dynamic linked library
GUID—globally unique identifier
IDL—interface definition language
MFC—Microsoft Foundation Classes
ODL—object description language
OLE—Microsoft's Object Linking and Embedding
OMG—Object Management Group
ORB—object request broker
OWL—the Borland Object Windows Library
RPC—remote procedure call
TP—transaction processing

Object-Oriented Technology and Components. Components need not be object oriented. Object-oriented technology, however, has great advantages for building reusable components. It promotes component-based software development because it allows reusable designs in the form of classes and frameworks to be used instead of functions. Existing code that is made into a component for reuse need not be object oriented, but new components will most likely be developed using object-oriented methods.

The remainder of the paper describes how object-oriented technology and methodologies, along with distributed object standards and middleware, can be used to develop reusable components.

An Architecture for Building Components

This section describes the advantages of using *object-oriented programming* to build components. Distributed computing standards allow objects in different address spaces and on different computers to communicate with one another. Middleware, such as transaction processing monitors, are described as being the foundation for developing business components.

Object-Oriented Programming. An excellent tool for building reusable components is object-oriented programming, which is described as follows:

> Object-oriented programming is a method of implementation in which programs are organized as cooperative collec-

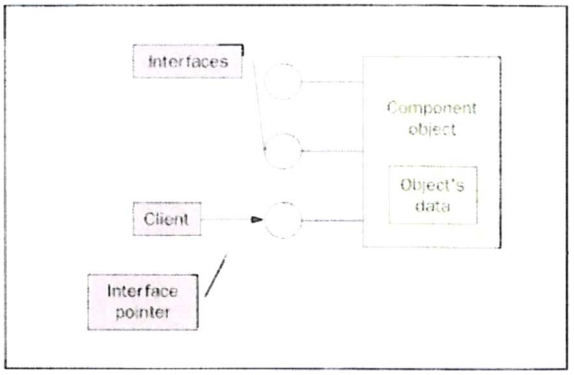

Figure 1. The Microsoft Component Object Model (COM) allows applications and systems to be built from components supplied by different software vendors. A COM *component object* or *component* is different from the traditional concept of objects in object-oriented programming in that a COM component object is a piece of compiled code providing some service. COM is the underlying architecture for higher-level software services, such as *object linking and embedding (OLE)*. This illustration shows a COM component object having three interfaces, one of which is being used by a client.

tions of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.[4]

A class is a template for an object and defines the object's methods (functions) and internal variables. An object encapsulates state and behavior. *Encapsulation* provides modularity and information hiding. Development methods other than object-oriented programming provide solely for encapsulation but only if very carefully structured programming techniques are used.

Polymorphism is the ability of objects of different classes that are related by a common super class to respond differently to the same message, reducing the number of interfaces for a class. Thus, any object of different classes that provides the same set of operations can be used in place of a specific class.

Inheritance is the ability to create new subclasses that are enhancements of existing classes. The subclass inherits all methods and internal variables of its parent class. Inheritance allows a class to be reused in a modified form by subclassing.

Although the three aforementioned properties of object-oriented programming can simplify the construction of reusable and easily maintained classes, the major reason that object-oriented systems have been successful at software reuse is the change in the way systems are designed. Design and not just code is reused. A mature design is one that can be reused and customized.[5]

Standards for Distributed Computing. The following four critical needs of businesses affect the architecture for components:

- Businesses want to cut costs by buying reusable components to build applications instead of hand-crafting them.
- Business components must have distributed implementations to satisfy the needs of global commerce.
- Business components must have well-defined interfaces preferably based on a language-neutral standard interface definition language for the purpose of interoperability.
- The functionality of components must be made available not only by linking them into programs but also by accessing the services of a component available on other computers in the enterprise.

The last need is perhaps the most important because it states that a component is more than just a reusable piece of code. A component is functionality that is available somewhere in the enterprise, and this functionality is available from remote locations exactly as if it existed locally. Two distributed computing models, the Microsoft COM and the OMG CORBA models, provide distributed object functionality. Both these architectural models can provide a basis for building components.

COM. A component software architecture, COM allows applications and systems to be built from components supplied by different software vendors. A COM *component object* or *component* (see Figure 1) is different from the traditional concept of objects in object-oriented programming in that a COM component object is a piece of compiled code providing some service. COM is the underlying architecture for higher-level software services, such as Microsoft's *Object Linking and*

Embedding (OLE). OLE is both a compound document framework and a visual component architecture at a higher level than COM. All OLE services use COM to allow binary software components to connect to and communicate with each other across processes and computers.[6] (Note that COM, the *Component* Object Model, is different from the Common Object Model architecture for enabling interoperation between Microsoft's OLE and Digital's ObjectBroker* technology.)

The COM defines a binary standard for invoking methods on an object. Any language that can call functions through double-pointer indirection (in C, C++, Smalltalk, for example) can be used to write COM-based component objects that can interoperate. Well-defined collections of functions (or *methods*) that a component provides are called *interfaces*. Component objects always access other component objects through interface pointers. A component object can never access another component object's data. Only interfaces are exposed to other component objects. This encapsulation of data and processing is a fundamental requirement of component software.

A component object can have one or more interfaces. An interface carries no implementation and cannot be instantiated by itself. A component object must implement the interface and be instantiated for the interface to be accessible. The interface is simply a related group of functions providing functionality for the component. A developer uses the *object description language (ODL)* to create a description of the interface's methods. The ODL compiler generates program header files and code for proxy and stub objects to use the interface. Other components only interact with pointers to interfaces, not with pointers to component objects. A component object can implement multiple interfaces representing different services provided. Every interface has a unique identifier (called a *globally unique identifier* or *GUID*) to eliminate naming conflicts. Interfaces never change. A new version of an interface is an entirely new one and is assigned a new identifier.

Component objects are programming-language independent because COM represents a binary-object standard. Each component object must implement a special interface called *IUnknown* that allows clients to discover at runtime whether an interface is supported by the component object. If the component object is not in process (as a dynamic linked library [DLL]) and exists as a separate process executing either on the same machine or another one, a *remote procedure call (RPC)* is made to enable local/remote transparency (cross-machine RPC has not yet been released by Microsoft).[6]

OLE is a set of system services built on COM for constructing compound documents and reusable components. OLE structured storage, drag and drop, embedding and linking, and in-place activation of objects are services to support compound documents composed of objects from different applications, available in a document-centric (as opposed to an application-centric) model. OLE automation and OLE controls, however, provide for component-based software. OLE automation allows a component object to expose all its methods and properties through the *IDispatch* interface, allowing late binding of method calls. OLE controls use most other OLE and COM technologies, including OLE automation and dispatch interfaces in addition to providing a user interface to receive input. OLE controls are Microsoft's definition of reusable components at the highest level.

CORBA. The OMG is a consortium of more than 500 hardware, software, and end-user companies founded in 1989 by a group of 11 firms including NCR Corp., Digital Equipment Corp., Hewlett-Packard Inc., Hyperdesk Inc., and SunSoft Corp. These companies, along with Object Design, authored the 1991 CORBA 1.0 specification. The goal of OMG is twofold: to establish agreement among system and application software vendors on a universal object-oriented set of standards for describing interfaces to distributed services and to use such interfaces for distributed services.

An *object request broker (ORB)* manages the interaction between client and server objects, including locating objects and marshaling (translating and transferring) object parameters and results between computers. OMG defines CORBA (see Figure 2) as describing interfaces for distributed services and how ORBs of different vendors would interoperate. The CORBA specification defines the architecture of an ORB, which enables and regulates interoperability between objects and applications. The recently approved CORBA 2.0 specification describes an interoperability protocol for accessing objects in ORBs provided by different vendors. In addition, the well-defined CORBAservices* are common

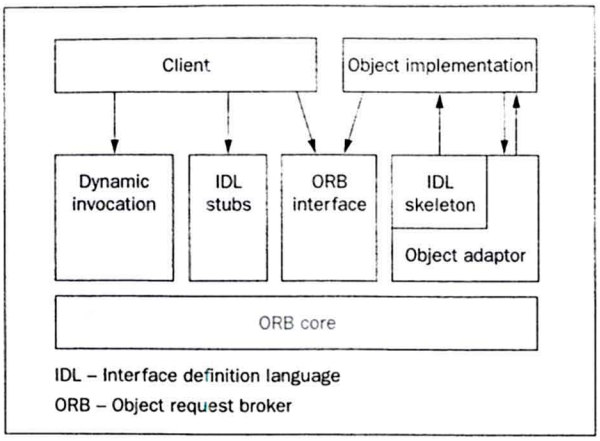

Figure 2. An *object request broker (ORB)* manages the interaction between client and server objects, including locating objects and marshaling object parameters and results between computers. OMG defines CORBA as describing interfaces for distributed services and how ORBs of different vendors would interoperate. The CORBA specification defines the architecture of an ORB, which enables and regulates interoperability between objects and applications. The illustration shows a simplified architecture of a CORBA ORB.

object services that include life cycle, naming, events, persistence, and transactions, all built atop a CORBA-compliant ORB. CORBAfacilities* are built over the ORB and CORBAservices* and provide application-level functionality, such as mail, database queries, and compound documents.[7]

The CORBA object model is a client/server model in which clients send messages to servers having zero or more parameters and receive back a return value or an exception if a failure occurs. The interface is strictly separated from the implementation. In the OMG model, objects are identified not by memory addresses as in C++ but rather by *object references*. Object references identify a specific instance of an object at a specific location. An instance is the state of the object married to its functionality (much like a process is the instance of a running program, differing from other running copies of the program by its state).

CORBA objects are defined in interfaces expressed in the *interface definition language (IDL)*. The term *interface* describes the methods that can be called on an object and the object's accessible attributes (public state variables), which describe how the object appears to the ORB and to clients. Interfaces do not address any implementation details. Clients use CORBA object services by either the IDL stub interface or the *dynamic invocation interface*, a mechanism for specifying requests at runtime. IDL is strongly typed. Basic types are fundamental data types: integers, floating-point numbers, chars, Booleans, enums, strings, octets (8-bit data types), and a nonspecific type called *any*. Constructed types are more complex types, such as structs, unions, sequences, arrays, and the interface type that specifies which sets of operations an instance of that type must support.[8]

The IDL describing an interface is mapped into different programming languages, such as C or C++. A CORBA IDL compiler takes IDL as input and outputs the code for the server and/or client parts of a distributed object. These are stubs for the methods on the client side and skeletons for the method implementations on the server side. The code for marshaling parameters and results, for invoking objects, and for making the actual network calls are typically generated by the IDL compiler. The only thing that is not output is the actual implementation of the methods of the distributed class. These methods must be implemented by a developer.

Components in COM and CORBA. Both COM and CORBA interfaces can be used to define high-level functionality of components. These interfaces describe object services that can call other services as needed to implement the functionality. By wrapping business functionality with COM or CORBA to build reusable components, high-level components can provide functionality at a level that makes sense to the application developer. Components implemented as COM or CORBA objects can be used by other components and applications, and they can be accessible from ORBs of different vendors. The OMG is currently evaluating proposals for the COM and CORBA models to interoperate with each another. In the future, an enterprise will have both COM and CORBA envi-

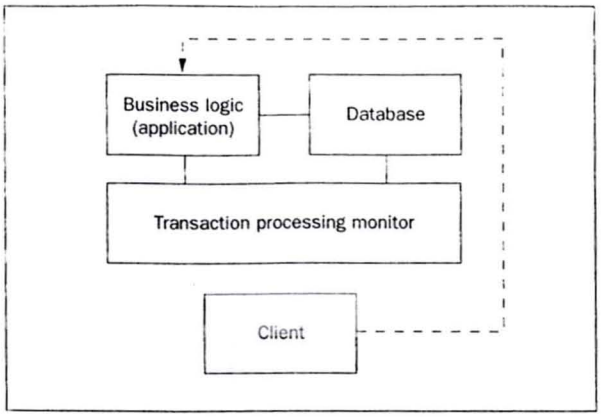

Figure 3. Both distributed data and distributed functionality require a transaction processing (TP) monitor. Business logic can be executed on multiple server computers and the data may also reside on multiple servers. The inexpensive processing power of PCs has forced the presentation layer to the workstation. The business logic, however, must be maintained across some number of servers for centralized administration. A TP monitor, shown linking clients with applications and data, manages the business logic on the servers.

ronments, and resources in one environment must be accessible by the other.

One major difference between COM and CORBA is in the process by which both these architectures have been created. OMG has used an open process with respect to CORBA in which different vendors submit proposals in response to a request for proposals. Microsoft has taken a closed, proprietary approach in developing COM. It is anticipated that Microsoft's COM will dominate the PC and workgroup environments, and the OMG's CORBA will dominate midrange and enterprise server environments.

Middleware for Transaction Processing. Why examine transaction processing systems? Today, most business logic resides in applications that run on database systems. This business logic is an excellent candidate for being wrapped in components and made reusable.

TP monitors. The majority of database applications on which businesses run have the following three major components:

- *Presentation,* or interaction with the end-user (for example, a Windows*-based front end);
- *Business logic,* or rules of the business (for example, server-based applications); and
- *Data,* or a persistent copy of the business state (for example, one or more database systems).

In non-distributed systems, all three of these components reside in one computer. A typical distributed data architecture has the data residing on a server, as well as multiple clients containing the presentation and business logic accessing it. These types of architectures frequently use Microsoft's LAN Manager* or Novell's NetWare* for remote file access, or Gupta, ORACLE, or Sybase products for remote database access. The business logic, however, resides on the client and the data resides on the server. To move more business logic to the server, stored procedures and applications offload more processing to the server on which the database is located.[9]

Both distributed data and distributed functionality require a transaction processing (TP) monitor (see Figure 3). Business logic can be executed on multiple server computers and the data may also reside on multiple servers. The inexpensive processing power of PCs has forced the presentation layer to the workstation. The business logic, however, must be maintained across some number of servers for centralized control and administration. A TP monitor manages the business logic on the servers by providing concurrency, transactions, and security—each a feature of the business-critical mainframe environment that no longer exists in the distributed client/server model.

A TP monitor is essentially a scheduler optimized for short-running applications. TP monitors allow many clients to access efficiently a much smaller number of application servers and yet still allow the configuration to grow by adding extra servers to handle increasing load. The TP monitor often supports multiple users by scheduling multiple instances of an application. This situation is ideal for languages like COBOL that do not support multithreading. Examples of leading TP monitors are the NCR

TOP END™ product, the Novell Tuxedo* monitor, and the IBM CICS* and Encina* products.

TP monitors allow global transactions over databases from different vendors. The X/OPEN* XA standard guarantees two-phase "commits" of transactions across multiple databases. Along with a transaction manager and systems management facility, a TP monitor also includes an application programming interface (API) to allow applications to use the asynchronous messaging and remote data update facilities. The NCR TOP END TP monitor, for example, includes location independence, load balancing, and automatic application-restart facilities, along with configuration and security management.

Integrating TP monitors and ORBs. Many advantages are realized by integrating a TP monitor and an ORB. TP monitors manage the data and the business logic of today's businesses. ORBs offer architectures for building the next generation of reusable distributed components. By combining the two, components can access existing business logic (in the form of database applications managed by the TP monitor) and the existing data in relational databases.

Object invocation is very similar to the message passing architecture of some TP monitors. Operations are the only way to access data. However, object activation (starting object instances not currently running) is a feature of ORBs while process scheduling across computers is not.[9] There is value in making the scheduling function of TP monitors available to ORB environments as well. Integrating a TP monitor with an ORB allows COM or CORBA components to be wrappers of existing business functionality stored as, for example, NCR TOP END applications. In the case of CORBA, IDL interfaces can be used to access TP monitor applications. In the case of OLE/COM, OLE controls provide a way to access the interface of a TP monitor application remotely. Integrating a TP monitor with an ORB also provides X/OPEN XA-compliant standards as a way to implement object-oriented transactions. A TP monitor can be used as the middleware supporting reusable business components.

Methods for Designing Components

After covering the architecture for components and the required technology, the discussion now shifts to the techniques by which components are actually developed. Objects help in developing components in that a component can be considered a high-level object, *but objects are not enough*. Frameworks are needed to manage complexity.

Frameworks. Design reuse is facilitated by the use of object-oriented frameworks. In a 1991 paper, frameworks are described as follows:

> A framework is the design of a *set* of objects that collaborate to carry out a set of responsibilities. Thus, frameworks are larger scale designs than abstract classes.[5]

A class describes the behavior of a single object. A framework describes the behavior of a set of classes that work together. Frameworks are abstractions of a set of interrelated classes. Individual classes may certainly be reusable, but the most value is achieved by using frameworks because the design is then reused, not just the code. A major difference between a framework and an arbitrary collection of classes, however closely related those classes might be functionally, is that a framework describes not only the objects but also their interactions with one another. Creating frameworks is a technique for organizing classes for reuse by maintaining their relationships.[10]

A framework is an example of a *white-box component* that must be modified to be used in different applications. Furthermore, you can "look inside" the component to reuse it. A *black-box component*, on the other hand, never changes and is used as is. With inheritance, the internals of parent classes are often visible to subclasses in a framework. Frameworks are one of the most useful forms of components. Frameworks, as white-box components, often include reuse of much larger parts, and they increase productivity significantly more than reuse of black-box components.[1]

Two types of frameworks currently exist: base and business frameworks. Base frameworks are the enabling technology that encapsulates the software infrastructure or platform, which can include the operating system, graphical user interface, distributed object request broker, TP monitor, and other middleware. Examples of base frameworks include the Microsoft Foundation Classes* (MFC*), the Borland Object Windows Library* (OWL*), and the Taligent CommonPoint* frameworks.

Business frameworks contain the knowledge of the objects in a business model and the relationships between the objects. Business frameworks can be used to

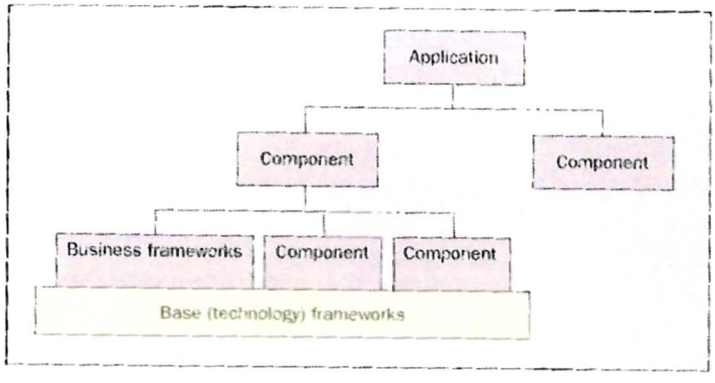

Figure 4. In progressing from models to components, a framework provides for successive refinement and composition of simpler components. The highest-level component is usually composed of many other components, either as a framework or via aggregation of other components. This is how complexity of components is managed. The illustration shows how applications are developed using components composed of business frameworks and other existing components atop base frameworks.

build many different components for a single industry. Economically, it makes more sense to have a set of frameworks specializing in certain targeted industries while having a set of base frameworks that are used by all industries. These industry-specific frameworks would be built atop base frameworks that encapsulate the underlying technology. OMG is now developing standardized architectures for the business frameworks of various industries.

From Models to Components. As components grow to encompass greater functionality, the complexity of the component will increase significantly. Object technology uses abstractions in an attempt to solve the problem of how to build complex systems. Frameworks deliver functionality that resides at a higher level than classes. Components and frameworks become more complex as the application domain increases. But how does one decide what makes up a framework?

Good knowledge of the application domain is critical to building white-box components, such as frameworks. A domain expert must work with a component designer to use a modeling methodology and extract the valuable design patterns that occur in that domain. A *design pattern* is the core of a solution to a problem that occurs over and over.[11] Various methodologies, such as OMT[3] or Objectory[1], allow an expert to capture all the functional requirements of a system.

Inheritance frameworks may not always be acceptable. For example, unlike CORBA, COM does not allow implementation inheritance. In this case, COM containment and aggregation techniques (instead of inheritance) must be used to build new components composed of other components. Nevertheless, a framework provides for successive refinement and composition of simpler components (see Figure 4). The highest-level component is usually composed of many other components, either as a framework or via aggregation of other components. This is how complexity of components is managed. Thus, frameworks are not only examples of white-box components, but they also provide a means by which simple components can be used to develop more complex components.

Once the problem domain has been modeled, the model can be specified in CORBA IDL or Microsoft ODL. Various products available today will convert different object methodologies into IDL, allowing an expert to define the model entirely using a preferred methodology. Once the interfaces of the components have been designed, a developer would only have to code the actual implementation of the methods. This has the advantage of separating the coding from the design.

Component development is divided into two groups of experts: component designers and implementors. Similarly, application development is also divided into two groups of experts: component and application developers. The application itself is built by "knitting" components together using a development tool, such as Cadre Technology's ObjectTeam Application Factory*. The application includes both business strategy and processes (work flows) for implementing components into useful applications.

The component has a high-level interface, and the implementation of its interface methods may call

other components or interact with other frameworks. The component's interface is very similar to that of a COM or CORBA object except that a component provides functionality at a much larger granularity. For example, a customer account component may be built using an accounting framework atop other base frameworks. The component's client-based graphical front end may be running on Windows* workstations as an OLE control and require MFC as a base framework. The component's server-based part, however, may be running on a UNIX* workstation with an ORB to manage object activation and access. The ORB itself may be working in conjunction with a TP monitor to schedule processing and to provide transaction support to access the customer data in a relational data-base. Other components and frameworks may provide support to the server part of the distributed component.

Conclusion

The central problem that components address is complexity. To reduce it, efficient mechanisms for constructing components are implemented, and hand-crafted software is replaced with reusable components. Both the COM and CORBA architectures support reusable components. For accessing existing business logic, which frequently depends on the concept of transactions and access data residing in relational databases, a TP monitor must also be included in the architecture. Object-oriented technology and methodologies, along with distributed object standards and technology, provide techniques for building components as object-oriented frameworks.

*Trademarks

CICS and Encina are trademarks of International Business Machines Corp.

CommonPoint is a trademark of Taligent Corp.

CORBAfacilities and CORBAservices are trademarks of Object Management Group.

LAN Manager, MFC, Microsoft Foundation Classes, and Windows are trademarks of Microsoft Corp.

NetWare is a registered trademark and Tuxedo is a trademark of Novell, Inc.

ObjectBroker is a trademark of Digital Equipment Corp.

ObjectTeam Application Factory is a trademark of Cadre Technology.

Object Windows Library and OWL are trademarks of Borland Corp.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/OPEN Company Ltd.

X/OPEN is a trademark of X/OPEN Company Ltd.

References

1. I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use-Case Driven Approach*, ACM Press, Wokingham, UK, 1992.
2. *Streamlining Software Development*, SRI International, Business Intelligence Program, Report No. 833.
3. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
4. G. Booch, *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.
5. R. E. Johnson and V. F. Russo, "Reusing Object-Oriented Designs," University of Illinois Technical Report UIUCDCS 91-1696, May 13, 1991.
6. K. Brockschmidt, *Inside OLE*, Second Edition, Microsoft Press, Redmond, Washington, 1995.
7. M. Betz, OMG's CORBA, *Dr. Dobb's Special Report*, Winter, 1994/95.
8. *The Common Object Request Broker: Architecture and Specification*, The Object Management Group, Revision 1.2, December 29, 1993.
9. E. E. Cobb, "TP Monitors and ORBs: A Superior Client/Server Alternative," *Object Magazine*, February 1995, pp. 57-61.
10. D. K. Kythe, "Strategies and Tools for Building Frameworks of Reusable Objects," *Proceedings of the AT&T Symposium on Software Reuse*, Holmdel, New Jersey, May 17, 1995.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1995.

(Manuscript approved March 1996)

Dave K. Kythe *was a senior principal software engineer at AT&T Global Information Solutions (now NCR) in West Columbia, South Carolina. He was responsible for the AT&T COOPERATIVE FRAMEWORKS, a set of C++ class libraries for CORBA-based distributed object computing. He has a B.S. degree in computer science and mathematics from Tulane University in New Orleans, Louisiana, and an M.S. in computer science from Indiana University in Bloomington. Mr. Kythe, who joined AT&T in 1989, recently left the company to accept another position.*