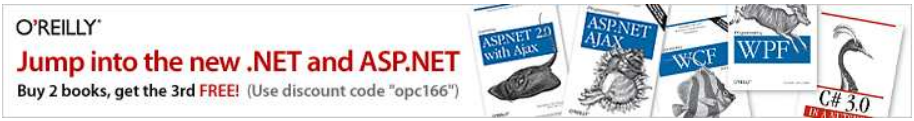

Dr. Dobb's Portal
The World of Software Development

The CORBA Component Model: Part 1, Evolving Towards Component Middleware

In the first installment of this new series, Douglas and Steve examine the CORBA Component Model, starting with the evolution of programming abstractions from subroutines to modules, objects, and components.

By Douglas C. Schmidt and Steve Vinoski, C/C++ Users Journal
febr. 01, 2004
URL:http://www.ddj.com/cpp/184403884

In the early days of computing, software was developed from scratch to achieve a particular goal on a specific hardware platform. Since computers were then much more expensive than the cost to program them, scant attention was paid to systematic software reuse and composition of applications from existing software artifacts. Over the past four decades, the following two trends have spurred the transition from hardware-centric to software-centric development paradigms:

- Economic factors. Due to advances in VLSI and the commoditization of hardware, most computers are now much less expensive than the cost to program them.
- Technological advances. With the advent of software development technologies, such as object-oriented programming languages and distributed object computing middleware, it has become easier to develop software with more capabilities and features.

A common theme underlying the evolution of software development paradigms is the desire for reuse; for instance, to compose and customize applications from preexisting software building blocks. Modern software development paradigms, such as object-oriented, component-based, and generative technologies, aim to achieve this common goal but differ in the type(s) and granularity of building blocks that form the core of each paradigm. The development and the evolution of middleware technologies also follow a similar goal of capturing and reusing design information learned in the past, within various layers of software.

This column is the first in a series that focuses on the CORBA Component Model (CCM). We first outline the evolution of programming abstractions from subroutines to modules, objects, and components. We then describe how this evolution has largely been mirrored in middleware, from message passing to remote procedure calls and from distributed objects to component middleware. We discuss the limitations of distributed object computing (DOC) middleware that motivate the need for component middleware in general and the CORBA Component Model (CCM).

The Evolution of Programming Abstractions

Composing software from reusable building blocks has been a goal of software researchers for

over three decades. For example, in 1968 Doug McIlroy motivated the need for software "integrated circuits" (ICs) and their mass-production and examined: (1) the types of variability needed in software ICs; and (2) the types of ICs that can be standardized usefully [1]. Since McIlroy's work predated the widespread adaptation of objects and components, he envisioned a software IC to be built from a standard catalog of subroutines, classified by precision, robustness, time-space performance, size limits, and binding time of parameters.

The earliest work in systems software, including operating systems and compilers for higher-level languages, also emphasized the use of prebuilt software. This software was often included in the OS or compiler runtime itself and was usually organized around functions that accomplished some common computing support capability ranging from online file systems to mathematical manipulation libraries. Application developers manually weaved these off-the-shelf functional capabilities into new applications.

Subsequent efforts to realize the vision of software ICs resulted in information hiding and data abstraction techniques that placed more emphasis on the organization of data than the design of procedures [2].Data abstraction resulted in formalizing the concept of modules as a set of related procedures and the data that they manipulate, resulting in partitioning of programs so that data is hidden inside the modules.

Information hiding and data abstraction techniques were embodied in programming languages, such as Clu, Modula 2, and Ada. These languages provided modules as a fundamental construct apart from explicit control of the scopes of names (import/export), a module initialization mechanism, and a set of generally known and accepted styles of usage of the features outlined above. Although these languages allowed programmers to create and apply user-defined types, it was hard to extend these types to new usage scenarios without modifying their interface definitions and implementations.

The next major advance in programming paradigms came from object-oriented design techniques, such as the OMT and Booch notations/methods (which eventually morphed into UML), and object-oriented programming languages, such as C++ and Java. Object-oriented techniques focus on decomposing software applications into classes and objects that have crisply defined interfaces and are related via derivation, aggregation, and composition. A key advantage of object-oriented techniques is their direct support for the distinction between a class's general properties and its specific properties. Expressing this distinction and taking advantage of it programmatically was simplified by object-oriented language support for inheritance, which allows the commonality in class behavior to be explicit, and polymorphism, which allows customization of base class behavior by allowing overriding of dynamically bound methods in subclasses.

Although the object-oriented paradigm enhanced previous programming paradigms, it also still had deficiencies. For example, object-oriented languages generally assume that different entities in a software system have interfaces that are: (1) amenable to inheritance and aggregation; and (2) written in the same programming language. Many applications today must run in multilingual and even multiparadigm environments, which necessitates a higher level of abstraction than can be provided by a single programming language or design paradigm.

The use of object-oriented techniques also indirectly led to many small relatively homogeneous parts that needed to be connected together by developers using detailed (and thus often error-prone and tedious) coding conventions and styles. This approach therefore required significant work beyond the existence of the primitive objects and even some supporting services, when developing large systems. What was needed was a way to package together larger units that were comprised of collections of the (sometimes heterogeneous) off-the-shelf

parts needed to provide higher-level capabilities, including their support attributes. For example, a typical capability might have the need for all of a variety of types of object invocations, events that trigger certain responses, transactions that manage the ordering and acceptance of state changes, and fault tolerance behavior that responds to failures. While each of these may individually be off-the-shelf, they needed to be separately and carefully knitted together in each instance to form a complete module handling some particular part of an overall application.

The conditions outlined above motivated the need for component-based software development techniques [3]. A component is an encapsulated part of a software system that implements a specific service or set of services. A component model defines: (1) the properties of components, such as the version of each component, dependencies between components, access modes, component identifiers, and operational policies; (2) the set of interfaces that components export; and (3) the infrastructure needed to support the packing, assembly, deployment and runtime management of component instances. Although components share many properties with objects (such as the separation of interface from implementation), they differ from objects in the following ways:

1. Multiple views per component and transparent navigation. An object typically implements a single class interface, which may be related to other classes by inheritance. In contrast, a component can implement many interfaces, which need not be related by inheritance. Moreover, components provide transparent "navigation" operations; for instance, moving between the different functional views of a component's supported interfaces. A single component can therefore appear to provide varying levels of functionality to its clients. Conversely, navigation in objects is limited to moving up or down an inheritance tree of objects via downcasting or "narrowing" operations. It is also not possible to provide different views of the same object since all clients are granted the same level of access to the object's interfaces and state.
2. Extensibility. Since components are described at a higher level of abstraction than objects, inheritance is less crucial as a means to achieve polymorphism. Objects are units of instantiation, and encapsulate types, interfaces and behavior that model the (possibly physical) entities of the problem domain in which they are used. They are typically implemented in a particular language and have some requirements on the layout that each inter-operating object must satisfy. In contrast, a component need not be represented as a class, be implemented in a particular language, nor share binary compatibility with other components (though it may do so in practice). Components can therefore be viewed as providers of functionality that can be replaced with equivalent components written in another language. This extensibility is facilitated via the Extension Interface design pattern, which defines a standard protocol for creating, composing, and evolving groups of interacting components.
3. Higher-level execution environment. Component models define a run-time execution environment—a "container"—that operates at a higher level of abstraction than access via ordinary objects. The container execution environment provides additional levels of control for defining and enforcing policies on components at runtime. In contrast, the direct dependency on the underlying infrastructure when using objects can constrain the evolution of component functionality separately from the evolution of the underlying runtime infrastructure.

The Evolution of Middleware

The emergence and rapid growth of the Internet, beginning in the 1970s, brought forth the need for distributed applications. For years, however, these applications were hard to develop due to a paucity of methods, tools, and platforms. Various technologies have emerged over the past

20+ years to alleviate complexities associated with developing software for distributed applications and to provide an advanced software infrastructure to support it.

Early milestones included the advent of the Internet protocols, message-passing architectures based on interprocess communication (IPC) mechanisms, micro-kernel architectures, and Sun's Remote Procedure Call (RPC) model. The next generation of advances included OSF's Distributed Computing Environment (DCE), IBM's MQ Series, CORBA, and DCOM. More recently, middleware technologies have evolved to support distributed real-time and embedded (DRE) applications (for instance, Real-time CORBA), as well as to provide higher-level abstractions, such as component models (for example, CCM and J2EE), web-services (such as SOAP), and model driven middleware (Cadena and CoSMIC, for instance).

The success of the middleware technologies outlined above has added the middleware paradigm to the familiar operating system, programming language, networking, and database offerings used by previous generations of software developers. By decoupling application-specific functionality and logic from the accidental complexities inherent in a distributed infrastructure, middleware enables application developers to concentrate on programming application-specific functionality, rather than wrestling repeatedly with lower-level infrastructure challenges.

One of the watershed events during the past 20 years was the emergence of distributed object computing (DOC) middleware in the late 1980s/early 1990s [4]. DOC middleware represented the confluence of two major areas of information technology: distributed computing systems and object-oriented design and programming. Techniques for developing distributed systems focus on integrating multiple computers to act as a unified scalable computational resource. Likewise, techniques for developing object-oriented systems focus on reducing complexity by creating reusable frameworks and components that reify successful patterns and software architectures. DOC middleware therefore uses object-oriented techniques to distribute reusable services and applications efficiently, flexibly, and robustly over multiple, often heterogeneous, computing and networking elements.

The Object Management Architecture (OMA) in the CORBA 2.x specification [5] defines an advanced DOC middleware standard for building portable distributed applications. The CORBA 2.x specification focuses on interfaces, which are essentially contracts between clients and servers that define how clients view and access object services provided by a server. Despite its advanced capabilities, however, the CORBA 2.x standard has the following limitations:

1. Lack of functional boundaries. The CORBA 2.x object model treats all interfaces as client/server contracts. This object model does not, however, provide standard assembly mechanisms to decouple dependencies among collaborating object implementations. For example, objects whose implementations depend on other objects need to discover and connect to those objects explicitly. To build complex distributed applications, therefore, application developers must explicitly program the connections among interdependent services and object interfaces, which is extra work that can yield brittle and non-reusable implementations.
2. Lack of generic application server standards. CORBA 2.x does not specify a generic application server framework to perform common server configuration work, including initializing a server and its QoS policies, providing common services (such as notification or naming services), and managing the runtime environment of each component. Although CORBA 2.x standardized the interactions between object implementations and object request brokers (ORBs), server developers must still determine how: (1) object implementations are installed in an ORB; and (2) the ORB and object implementations interact. The lack of a generic component server standard yields tightly coupled, ad-hoc

server implementations, which increase the complexity of software upgrades and reduce the reusability and flexibility of CORBA-based applications.

3. Lack of software configuration and deployment standards. There is no standard way to distribute and start up object implementations remotely in CORBA 2.x specifications. Application administrators must therefore resort to in-house scripts and procedures to deliver software implementations to target machines, configure the target machine and software implementations for execution, and then instantiate software implementations to make them ready for clients. Moreover, software implementations are often modified to accommodate such ad hoc deployment mechanisms. The need of most reusable software implementations to interact with other software implementations and services further aggravates the problem. The lack of higher-level software management standards results in systems that are harder to maintain and software component implementations that are much harder to reuse.

An Overview of Component Middleware and the CORBA Component Model

Component middleware is a class of middleware that enables reusable services to be composed, configured, and installed to create applications rapidly and robustly [6]. During the past several years component middleware has evolved to address the limitations of DOC middleware described above by

- Creating a virtual boundary around larger application component implementations that interact with each other only through well-defined interfaces.
- Defining standard container mechanisms needed to execute components in generic component servers.
- Specifying the infrastructure to assemble, package, and deploy components throughout a distributed environment.

The CORBA Component Model (CCM) [7] is a current example of component middleware that addresses limitations with earlier generations of DOC middleware. The CCM specification extends the CORBA object model to support the concept of components and establishes standards for implementing, packaging, assembling, and deploying component implementations. From a client perspective, a CCM component is an extended CORBA object that encapsulates various interaction models via different interfaces and connection operations. From a server perspective, components are units of implementation that can be installed and instantiated independently in standard application server runtime environments stipulated by the CCM specification. Components are larger building blocks than objects, with more of their interactions managed to simplify and automate key aspects of construction, composition, and configuration into applications.

A component is an implementation entity that exposes a set of ports, which are named interfaces and connection points that components use to collaborate with each other. Ports include the following interfaces and connection points:

- Facets, which define a named interface that services method invocations from other components synchronously.
- Receptacles, which provide named connection points to synchronous facets provided by other components.
- Event sources/sinks, which indicate a willingness to exchange event messages with other components asynchronously.

Components can also have attributes that specify named parameters that can be configured later via metadata specified in component property files.

A container provides the server runtime environment for component implementations called executors. It contains various pre-defined hooks and operations that give components access to strategies and services, such as persistence, event notification, transaction, replication, load balancing, and security. Each container defines a collection of runtime strategies and policies, such as an event delivery strategy and component usage categories, and is responsible for initializing and providing runtime contexts for the managed components. Component implementations have associated metadata written in XML that specify the required container strategies and policies.

In addition to the building blocks outlined above, the CCM specification also standardizes various aspects of stages in the application development lifecycle, notably component implementation, packaging, assembly, and deployment, where each stage of the lifecycle adds information pertaining to these aspects.

The CCM Component Implementation Framework (CIF) automatically generates component implementation skeletons and persistent state management mechanisms using the Component Implementation Definition Language (CIDL). CCM packaging tools bundle implementations of a component with related XML-based component metadata. CCM assembly tools use XML-based metadata to describe component compositions, including component locations and interconnections among components, needed to form an assembled application. Finally, CCM deployment tools use the component assemblies and composition metadata to deploy and initialize applications.

The tools and mechanisms defined by CCM collaborate to address the limitations with DOC middleware described earlier. The CCM programming paradigm separates the concerns of composing and provisioning reusable software components into the following development roles within the application lifecycle:

- Component designers, who define the component features by specifying what each component does and how components collaborate with each other and with their clients. Component designers determine the various types of ports that components offer and/or require.
- Component implementers, who develop component implementations and specify the runtime support a component requires via metadata called component descriptors.
- Component packagers, who bundle component implementations with metadata giving their default properties and their component descriptors into component packages.
- Component assemblers, who configure applications by selecting component implementations, specifying component instantiation constraints, and connecting ports of component instances via metadata called assembly descriptors.
- System deployers, who analyze the runtime resource requirements of assembly descriptors and prepare and deploy required resources where component assemblies can be realized.

The CCM specification has recently been finalized by the OMG and is in the process of being incorporated into the core CORBA specification. The CORBA 3.0 specification released by the OMG only includes changes in IDL definition and Interface Repository changes from the Component specification [7]. CCM implementations are now available based on the recently adopted specification, including OpenCCM by the Universite des Sciences et Technologies de Lille, France, K2 Containers by iCMG, MicoCCM by FPX, Qedo by Fokus, and CIAO by the DOC groups at Washington University in St. Louis and the Institute for Software Integrated Systems (ISIS) at Vanderbilt University. The architectural patterns used in CCM are also used in other popular component middleware technologies, such as J2EE and .NET.

Concluding Remarks

In this column, we've provided a brief history of middleware and explained how it's evolved towards components. Component middleware represents a new generation of middleware that overcomes many of its predecessors' shortcomings. For example, by providing models that capture frequently used middleware patterns, such as interface navigation and event handling, and also by encapsulating common execution functions within generic component server containers, component middleware promises to ease the burden of developing distributed applications. The fact that component middleware also addresses assembly, packaging, and deployment concerns means that it enhances and simplifies the traditionally hard process of deploying distributed applications into production environments. Finally, we introduced the CORBA Component Model (CCM), which builds on the well established and widely deployed CORBA base to provide a practical component model for real world distributed applications.

References

[1] Doug McIlroy, "Mass Produced Software Components," *Proceedings of the NATO Software Engineering Conference*, October 1968.

[2] David L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol 15, no 12, December 1972.

[3] Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[4] Richard E. Schantz and Douglas C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," *Encyclopedia of Software Engineering*, Wiley, 2002.

[5] "The Common Object Request Broker: Architecture and Specification Revision 2.6, OMG Technical Document formal/05-09-02", May 2002.

[6] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan, "An Overview of the CORBA Component Model," Component-Based Software Engineering, Addison-Wesley, 2000.

[7] "CORBA Components," OMG Document formal/2002-06-65, Jun 2002.

Copyright © 2006 CMP Media LLC