Cover Feature

The Emergence of Distributed Component Platforms

Distributed component platforms isolate much of the conceptual and technical complexity involved in constructing component-based applications. The authors examine the concepts underlying DCPs, the two market leaders—Microsoft's DCOM and Sun's JavaBeans—and emerging Internet and OMG standards.

David Krieger

Richard M. Adler
Computer Sciences Corp.

Much has changed in the world of component software since we last surveyed emerging standards three years ago.[1] In that time, several vendors and consortia have independently developed standards that define the basic mechanics for building and interconnecting software components. Sun's JavaBeans has emerged as the leading rival to Microsoft's DCOM (Distributed Component Object Model), supplanting the OpenDoc standard from the now defunct Component Integration Laboratories. Component software is moving from its original focus on desktop-bound compound documents to enterprise applications that include distributed server components.

The backers of competing standards are racing to capture market leadership by delivering the tangible benefits of component standards via *distributed component platforms*—integrated development and runtime environments that isolate much of the conceptual and technical complexity involved in building component-based applications. With DCPs, businesses can assign their few highly skilled programmers to component construction and use less sophisticated developers to carry out the simpler assembly tasks. By making component standards available to the broadest possible spectrum of developers, DCPs essentially drive those standards to market.

In this article, we review the state of component software as embodied in DCPs. The two DCP market leaders are Microsoft's DCOM (or ActiveX/DCOM) and Sun's JavaBeans. However, Internet and Object Management Group (OMG) component standards are emerging that will likely impact both the content and status of these two DCPs. We also discuss component frameworks, which extend DCPs to provide more complete application development solutions.

DCP CONCEPTS

Software components are reusable building blocks for constructing software systems.[2] Components encapsulate semantically meaningful application or technical services, such as rating insurance applicants or authorizing client access to service resources. Components differ from other types of reusable software modules in that they can be modified at design time as binary executables; in contrast, libraries, subroutines, and so on must be modified as source code.

Component standards specify how to build and interconnect software components. They show how a component must present itself to the outside world, independent of its internal implementation. This single-minded emphasis on external interfaces and interaction protocols distinguishes component standards from other communication conventions. Well-thought-out component standards ensure that

- components with similar specifications are interchangeable and independently upgradable,
- developers can customize the appearance and behavior of components along predetermined dimensions, and
- components can be combined to form larger grained components as well as complete applications.

Thus, component standards play a critical role in ensuring that developers achieve the anticipated benefits from reusable components—enhanced productivity, uniformity, ease of use, and faster time to market.

Component interfaces

A component restricts access to its services and internal structures through one or more public *interfaces*. As Figure 1 shows, an interface defines a set of properties, methods, and events through which external entities can connect to, and communicate with, the

0018-9162/98/$10.00 © 1998 IEEE

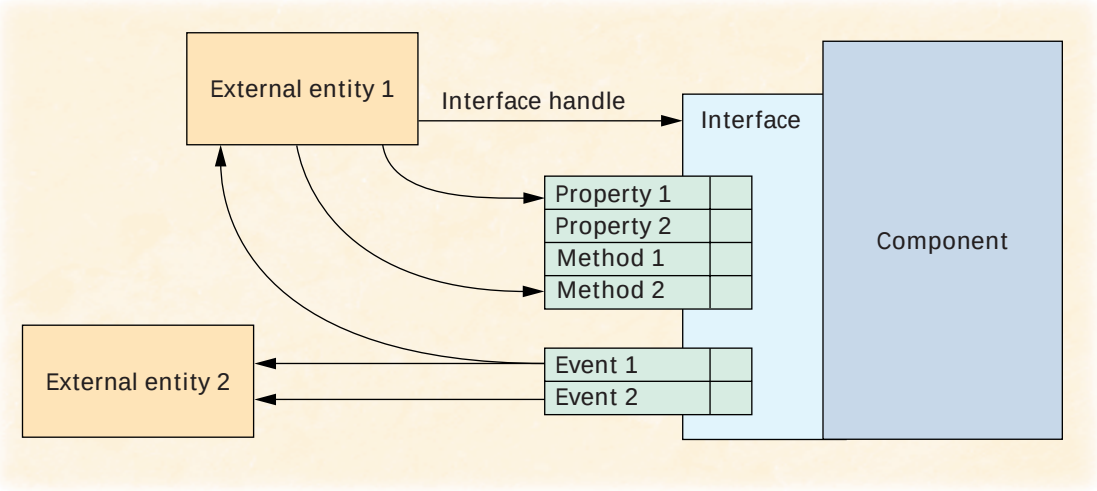

Figure 1. How a component Interface works. An interface defines a set of properties, methods, and events through which external entities can connect to, and communicate with, components.

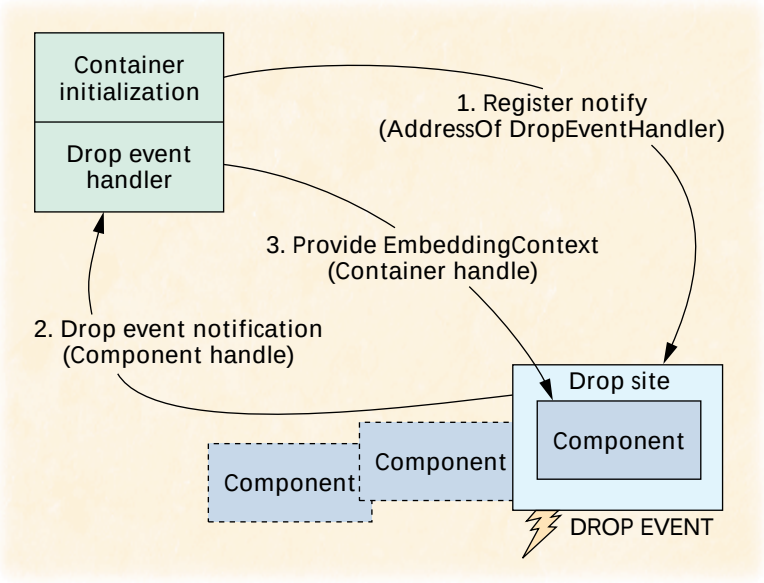

Figure 2. Event registration and notification. A drag-and-drop event is used to dynamically relate a newly embedded component to its container.

component. *Properties* and *methods* typically represent a component's callable API—the protocol used by external entities to access a component's services. Properties expose a component's public attribute data via accessor operations; methods inaugurate a component's behavior. *Events* specify a component's response to external stimuli or internal conditions, such as a property value changing. The component interface specifies the signature of the event it will raise when the condition occurs. It does not know or care how that event is consumed or who the consumer is. Consuming entities are responsible for registering interest in the event and providing a handler for its occurrence. This publish-and-subscribe model of interaction allows communication channels to be established dynamically.

Containers

Components exist and operate within *containers*, which provide a shared context for interaction with other components. Containers also offer common access to system-level services for a component's embedded components (such as process threads and memory resources). Containers are typically implemented as components, which can be nested in other containers. An example is embedding widget field arrays into panels within GUI windows. Event-based protocols are commonly used to establish the relationship between a component and its container.

Figure 2 illustrates dynamic event protocols for dragging and dropping a component onto a container. Upon initialization, the container registers its interest in drag-and-drop events with a drop site. (Drop sites are usually implemented as interfaces on the container itself.) Later, when a drop event occurs, the drop site notifies the container by calling the container's previously registered event handler, passing a handle to the dropped component. The event handler might change the appearance of its container's icon to, say, signal the user that the drag-and-drop operation has successfully completed. Typically, it will pass the dropped component a handle to the container, which lets the component access the container's environmental services.

Pervasive metadata

Component standards specify the self-descriptive information that a component must publish to flexibly communicate with other components. This *metadata* lets containers, scripts, development tools and other components discover a component's capabilities—either statically at design time, or dynamically at runtime.

Figure 3 illustrates the basic categories of metadata used in DCPs. DCPs generally implement metadata as a type of component whose interfaces are termed introspection or reflection. The italicized terms correspond to the categories in the figure.

- *Component info* describes the component's general compile-time and runtime properties, including where to find it and how to activate it (for example, a path name and its process or its caller process).
- *External references* point to metadata that describes other components.
- *Type descriptor* and *Type* form the fundamental metadata elements.

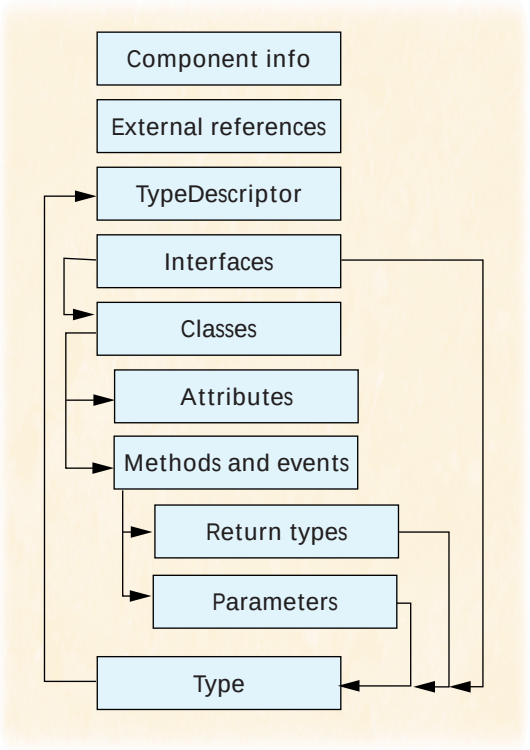

Figure 3. Generic model of metadata categories. Metadata lets containers, scripts, development tools, and other components discover a component's capabilities at either design time or runtime.

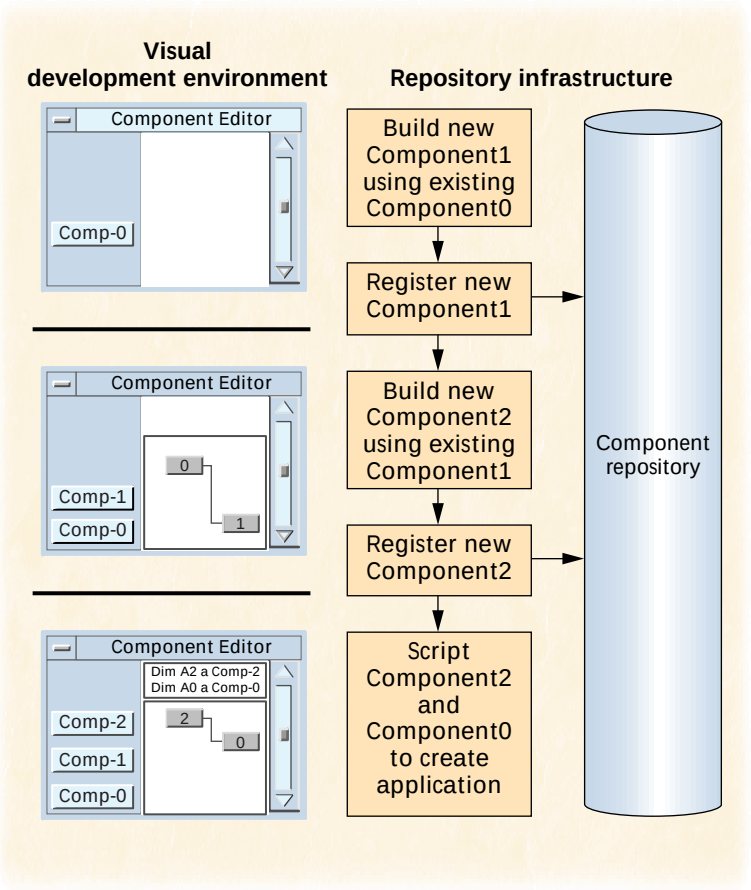

Figure 4. Using an integrated development environment (IDE) to build an application with existing and new components. The developer scripts component interactions.

- *Interfaces* describe public attributes, events, and methods.
- *Classes* describe the implementation of one or more interfaces. The metarelationships between classes, types, and interfaces differ across DCPs.
- *Attributes*, *methods*, and *events* characterize the classes and interfaces. All have associated metadata.
- *Return types* and *parameters* specify method inputs and outputs.

Recent advances in DCPs have been fueled primarily by enriching component metadata and exploiting it aggressively. For example, metadata drives component composition and dynamic collaboration, enabling components that discover and manipulate each other's interfaces at runtime. Similarly, tools that extend development environments, such as object browsers, debuggers, and smart code editors rely on component metadata to populate themselves. A DCP's longevity will depend strongly on the expressiveness and extensibility of its metadata model.

Integrated development environments

The notion of a component is inseparable from the notion of a component development environment or builder. The value of a DCP depends heavily on the efficacy and usability of its *integrated development environment*. IDEs are rapidly moving from text-based programming toward the direct manipulation of visually rendered components. Commercial IDEs include Microsoft's Visual Studio, IBM's VisualAge for Java, and Symantec's VisualCafe, supplemented with scripting languages such as VBScript and JavaScript.

IDEs typically include or are evolving to include

- one or more palettes for displaying available components (rendered as icons);
- a "canvas" container onto which components are placed and interconnected, typically through drag-and-drop operations and pop-up menus;
- property and script editors that let users customize components within their containers;
- editors, browsers, interpreters, compilers, and source-level debuggers for developing new components and testing and refining component applications;
- a component repository and associated design-time browser services to locate components by matching user search criteria and using inspectors to view component metadata; and
- configuration management tools that structure and coordinate team-based development and release processes—tools that are essential for large software projects.

Figure 4 depicts a scenario in which a developer uses an IDE to construct an application by composing new components visually and scripting interactions between existing and new components.

This scenario illustrates that an IDE not only must

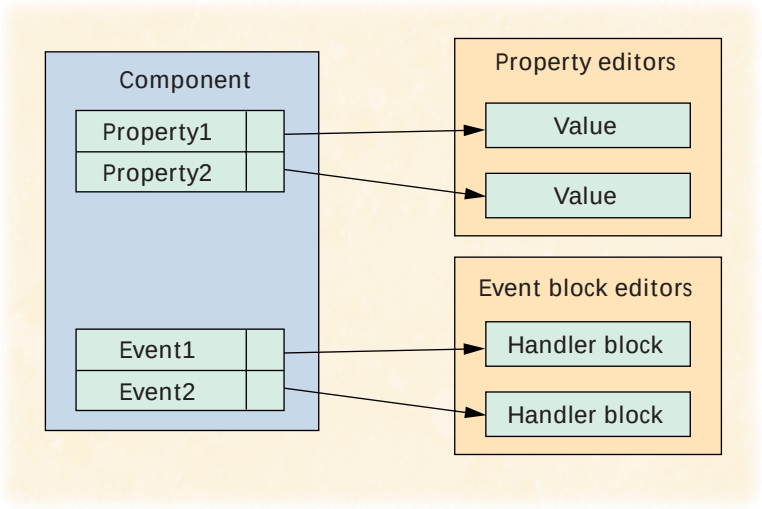

Figure 5. Customizing components in the IDE.

support the creation and composition of reusable components, it must also act as a framework for seamlessly integrating new components into itself. The IDE relies on the DCP's component standard to ensure agreement among the components, containers, and IDE on the patterns for hierarchical composition and metadata exchange.

Developers can also customize existing components by setting property values and providing new handlers for their events, as shown in Figure 5. Developers modify components either explicitly, via IDE menus and editors, or implicitly, as side effects of visual manipulations such as dropping component icons onto containers. Either way, the IDE must know how to access a component's metadata to display and populate editors or handle drag-and-drop events.

Components that can be customized must know if they are executing in a runtime or design-time context. Components expose different interfaces and display different behaviors at these times. During design, the component collaborates with the IDE through its metadata to expose its property and event editors. At runtime, however, the component manifests the behaviors specified during design. For example, clicking on a button component during design generally opens an event editor; at runtime, the same mouse click signals the button's window container to perform some action, such as closing itself.

Distributed server components

Enterprise computing relies on a robust set of services for accessing and managing shared services, information, and computing resources. To move beyond the desktop, components require five distributed services:

- *Remote communication protocols*, which enable interactions at the application layer among components distributed over a network. Protocols can be synchronous (for example, remote procedure calls) or asynchronous (for example, messaging services that enable an efficient, nonblocking store-and-forward model).
- *Directory services*, which provide a coherent global scheme for naming, organizing, and accessing shared services and resources.
- *Security services*, which protect shared resources by ensuring that communicating parties are properly authenticated and have suitable authorization, and that third parties have not intercepted or tampered with their messages.
- *Transaction services*, which coordinate concurrent updates to enterprise-critical data, and ensure that all updates leave data in correct and consistent states.
- *System management services*, which provide a unified set of facilities to monitor, manage, and administer services and resources across the enterprise.

A large installed base of mature products that support distributed-service APIs already exists. However, because these product sets deliver overlapping services through different APIs, it is difficult to select products and integrate their services into applications. This makes it much more challenging to design enterprise server components that can effectively incorporate distributed services.

In fact, many distributed facilities are not services in the traditional client-server sense. Rather, they constitute part of a component's runtime environment. For example, a server component may obtain transactional persistence from its runtime context, rather than implementing it directly. The relationship between transaction context and server components is roughly analogous to visual containment of desktop components.

DCP providers are attempting to generalize containment models to encompass the relationships between server components and their runtime hosts; like visual containers, transaction contexts provide runtime access to basic life-support and custodial services for their components. However, server components generally provide concurrent services to many users, whereas desktop components support single users. Also, server components are often multithreaded, replicated, and pooled, to achieve scalability and reliability. Consequently server components can't readily be organized into static containment hierarchies. In short, the complexities of distributed computing threaten the viability of container-based models for designing and using server components.

DCOM

Microsoft's DCP is based on DCOM, which consists of the Component Object Model (COM) binary standard, augmented with a runtime infrastructure to

support component communication across distributed address spaces.[3-4] (Earlier COM-based incarnations were named VBX and OCX, for VisualBasic and OLE controls, respectively.) DCOM specifies the type and the structure of the interfaces components must implement. All DCOM components must implement at least one interface, `IUnknown`, which supports basic mechanisms for casting interface references and reference counting.[5-7]

DCOM itself is a relatively simple component standard; its utility resides more in specialized interfaces, such as compound documents, drag-and-drop, and persistent streaming and storage. DCOM's event notification mechanism is implemented through `IConnectionPoint` and several related interfaces.

DCOM component interfaces are specified using an Interface Definition Language (IDL) derived from the Open Software Foundation (OSF). DCOM is independent of language and implementation. Until recently, DCOM was restricted primarily to the Windows platform. Maturing ports to Unix and MacOS are reducing DCOM's platform dependency.

Development environment

Microsoft's Visual Basic 3.0 provided a rudimentary but productive environment for assembling VBX components, which have since been replaced with components based on the DCOM standard. In addition, Microsoft has infused progressively more component-oriented programming capabilities into its IDEs. For example, VB 5.0 lets developers create new components as well as use existing ones. In addition, the VB IDE now creates much of the boilerplate code components require at runtime. For example, it automatically generates and registers component interfaces and metadata.

The Microsoft IDE currently supports component development in three languages—Visual Basic, Visual C++, and J++ (Microsoft's Java). The IDE is itself an application built using the native DCOM model, making it extensible and customizable through standard DCOM mechanisms. It also provides mechanisms (add-ins) that help developers customize it to, say, have a different look and feel or enforce desired development patterns.

Metadata

The basic COM model supports a rudimentary form of component self-description through `IUnknown`. In addition, a set of metadata components, *Type Libraries*, express in machine-readable form what IDL expresses in human-readable form.

A Type Library collects metadata for its associated component and provides `ITypeLib` and `ITypeInfo` interfaces to access and navigate the metadata. A Type Library contains five general kinds of information:

- *CoClass* is a metadescriptor for a COM object, describing all incoming and outgoing interfaces for that COM class, including public properties and methods.
- *Interface* provides memory layout and descriptive information for public operations such as names, return-type, and optional dispatch identifier.
- *Module* describes the dynamic linking library (DLL) module, including path name global variables and exported functions.
- *Typedef* is the metadescriptor for user-defined data structures.
- *Importlib* provides a way to get the metadescriptor for a referenced Type Library.

DCP providers are attempting to generalize containment models to encompass the relationships between server components and their runtime hosts.

Each metadescriptor specifies a common set of properties, including a human-readable name, machine-readable globally unique identifier (GUID), version, documentation string, help file name, and flags (for example, hidden or restricted).

Given a class identifier (CLSID), a client can get a handle to a component's Type Library and query its metadata without an instance of the component. Alternatively, if a running component instance is available, a client can obtain the Type Library through the instance's `IProvideClassInfo` interface. The first mode typifies a design scenario. The second mode aids in dynamically discovering interfaces and in component collaboration at runtime.

Server platform

Microsoft is strongly committed to delivering a credible server component platform by successively enhancing the NT operating system. NT's current and beta-level distributed enterprise services include

- *Remote communication protocol.* DCOM uses remote procedure calls to communicate across the network. Microsoft's RPC, derived from the OSF DCE RPC, allows communication among distributed components through a standard proxy/stub mechanism. Microsoft also plans to support an asynchronous protocol directly in DCOM, most likely through tighter integration of their messaging product, MS Message Queue.
- *Directory services.* Microsoft Active Directory combines Domain Naming System, the dominant Internet name resolution scheme, with the Lightweight Directory Access Protocol (LDAP), which is ISO X.500 compliant. Microsoft plans to integrate these services into the next release of Windows NT.
- *Security services.* Microsoft NT provides Secure Sockets Layer public-key-based security and a proprietary security protocol that is based on NT

DCOM is language-neutral but platform-dependent; JavaBeans is platform-neutral but language-dependent.

LAN Manager (NTLM). However, the next release of NT will include a security service compliant with Kerberos 5.0 and tightly integrated with Microsoft's Active Directory.

- *System management services*. NT 4.0 includes a Management Console (MMC) that comprises a UI container for third-party systems management components. NT 5.0 is slated to provide a true management information bus that will support mechanisms to monitor and manage the resources and services on a distributed NT installation. The bus will also publish a set of APIs for integrating existing system management products.
- *Transaction services*. The Microsoft Transaction Server integrates transaction services into the component development model and provides a transactional runtime environment for server components. MTS defines `ObjectContext`, an analog to containers for server components. Server components access their operational context through this component's `IObjectContext` interface. Components created within, or added to, an `ObjectContext` participate transparently in that context's transactions and share that context's associated security.

JAVABEANS

JavaBeans has quickly gained market attention, emerging as the dominant competitor to DCOM. Whereas DCOM is language-neutral but platform-dependent, JavaBeans is platform-neutral but language-dependent. DCOM capabilities are built up by composing more elementary binary components. For example, the `IPersist` interface is composed from `IStorage` and `IStream`. In contrast, JavaBeans component capabilities are implemented as a set of language extensions to the standard Java class library. Thus, JavaBeans is a set of specialized Java programming language interfaces. It achieves platform portability through the Java Virtual Machine.

Like DCOM, JavaBeans interfaces expose properties, methods, and events. The JavaBeans property model is richer than DCOM's: In addition to single- and multivalue properties, JavaBeans defines bound and constrained property types. *Bound* properties use Java events to notify other components of a property value change; *constrained* properties let those components veto a change. Constrained properties provide a uniform language-based approach to basic validation business rules. Both bound and constrained properties are missing from most object-based systems. They let developers factor application logic in a modular and maintainable way that makes it easy to preserve the consistency of business data.

JavaBeans' event notification mechanism involves three related Java-level class interfaces—`Event`, `EventSource`, and `EventListener`. `EventSource` notifies all registered `EventListeners`, passing each an `Event` object when the event of interest occurs. The event model supports two other features that enhance ease of use. `EventSources` default to multicast but can be set to unicast—allowing at most one `EventListener`. `EventAdapters` can be added to relieve developers from having to write handlers for all the events defined in a listener's interface.

In addition to component interface constructs, JavaBeans incorporates mechanisms for component containment and pervasive metadata analogous to DCOM. However, DCOM components implement persistent identity through GUIDs. JavaBeans use string names, which may not be globally unique.

Development environment

The JavaBeans API explicitly supports visual development of Bean components using *property sheets*—built-in property editors and editor aggregations. This IDE or application builder support is analogous to DCOM's `IPropertyPage` interface.

A growing number of IDEs (including Symantec's VisualCafe, IBM's Visual Age for Java, Borland's JBuilder, and Sun's Java Workshop) support visual development with property sheets, palettes, and design-time drag-and-drop behaviors. These tools offer productivity features comparable to the Microsoft IDE. In addition, their visual model for interconnecting components is significantly more expressive and intuitive than the current versions of the Microsoft IDE.

Metadata

JavaBeans inherits Java's Core Reflection API, which provides a specialized set of classes for metadata. Each of Java's methods, fields, constructors, interfaces, and classes has a corresponding metadata class that supports dynamic interrogation, instantiation, and invocation. Unlike the Java language, ANSI C++ and Visual Basic don't have built-in reflection support. Consequently all metadata on DCOM (except for J++) is in the component model, not the language.

In addition to Java's Core Reflection API, JavaBeans provides an `Introspection` interface that returns a different set of metadata classes tailored to support component-based development. For example, the visual icon that a Bean displays on an IDE's palette is specified by the `BeanInfo` metadata class. The other JavaBeans metadata classes are derived from a common base class, `FeatureDescriptor`, and roughly correspond to DCOM metadata,[8] as summarized in Table 1. To foster the systematic use of JavaBeans metadata, the DCP provides a utility class,

Table 1. Comparison of DCOM and JavaBeans metadata.

Generic metadata	DCOM	Java Core Reflection	JavaBean, BeanInfo
Class	CoClass, ITypeInfo	Class	BeanDescriptor.GetBeanClass
Type	TypeDef	Class	
To Get from running object	AnObject.IprovideClassInfo()	anObject.GetClass()	
To dynamically instantiate a class	ITypeInfo.CreateInstance()	aClass.newInstance()	
To get reflected member information	GetFuncDesc(),GetTypeAttr(), GetVarDesc(),GetNames()....	GetClasses(),GetFields(), getMethods(), getConstructor()	GetEventDescriptors, GetMethodDescriptors, GetPropertyDescriptors
To navigate up	ITypeInfo.GetContainingTypeLib()	GetDeclaringClass()	
To dynamically invoke a method	Invoke()	Invoke()	
ComponentInfo	ITypeLib.GetLibAttr() ITypeInfo,GetDLLEntry()		BeanDescriptor, BeanInfo
External References	ImportLib	Package import conventions	
Standard metadata conventions	Element Attributes		FeatureDescriptor

`Inspector`, which navigates the Introspection and Core Reflection APIs.

Despite their many useful features, the JavaBeans metadata facilities rely heavily on error-prone naming conventions called *design patterns*. For example, the accessor and mutator methods for a property *X* must be named `GetX` and `SetX` in order for the `Introspection` interfaces to work properly. `EventListeners` and `BeanInfo` names are similarly constrained by string-oriented templates.

Distributed server components

Sun has recently released a preliminary specification for Enterprise JavaBeans (EJB). The goal is to provide the same kind of scalable, enterprise-capable server environment for JavaBeans that Microsoft is delivering and hardening for Windows NT servers.[9] The standard explicitly specifies that all Enterprise JavaBeans will run inside a container that isolates the JavaBean from the server execution environment. The container automatically allocates process threads to the component and manages concurrency, security, persistence, and transactional activities on behalf of the component. EJB's server environment includes

- *Remote communication protocol.* JavaBeans has full access to Java's native remote method invocation (RMI), which runs directly on top of TCP/IP. However, enabling RMI for a class requires making explicit changes to an existing Java class. Also, instances of remote classes cannot be passed by value.
- *Directory services.* Sun has released a beta version of the Java Naming & Directory Interface (JNDI), which provides an implementation-independent API that allows applications written in Java to leverage existing directory services such as LDAP and Domain Name System. EJB containers must be locatable through the JNDI.
- *Security services.* EJB can use all security services specified by the standard `java.security` package. This includes public- and private-key authentication, encryption, digital key management, and access control lists.
- *Systems management services.* At the time of this writing, Sun has not met its scheduled public released date of fourth quarter 1997 for the 1.0 specification for its Java Management API (JMAPI). However, available documentation indicates that JMAPI will specify a comprehensive set of monitoring, management, and administrative services, including a UI style guide for an administrator's console (http://java.sun.com/products/JavaManagement/document.html).
- *Transaction services.* The EJB specifies a flat transaction model based on the OMG's Object Transaction Service, explicitly discouraging use of the existing Java Transaction Service (JTS) API. Instead it delegates transaction management to the Bean's container.

JavaBeans components can be packaged for embedding in containers that support Microsoft's DCOM component model, including Visual Basic, Internet Explorer, Office, and Lotus Notes. This form of interoperability is driven by a communication bridge. The core technology behind the bridge is a packaging utility that generates an OLE Type Library and Win32 registry information for selected JavaBeans. The resulting data lets DCOM containers properly analyze, present, and manipulate Beans (for example, catching Bean events, invoking Bean service methods, and creating property sheets to customize Beans).

DISTRIBUTED COMPONENTS ON THE WEB

The Internet and private intranets are increasingly

W3C efforts align with DCPs in assuming that pervasive metadata enables semantically rich interoperability, whether between components or Web documents.

perceived as critical to enterprise computing architectures. Unfortunately, attempts to use the Web as a platform for distributing component applications have been impeded by the limited expressiveness of the HTML document standard.

As a short-term response, the World Wide Web Consortium (W3C) has incorporated an `<object>` tag into HTML 4.0. W3C is also sponsoring more broadly based architectural solutions. These emerging standards could potentially lead to a convergence between component and Web document architectures.

Extensible Markup Language, or XML (http://www.w3.org/TR/REC-html40/), is a metamodel for structured document exchange based on the Standard General Markup Language (SGML, an ISO standard). HTML restricts Web documents to a predefined set of tags for specifying content and format. In contrast, XML supports the definition of customized markup languages. For example, XML tags could be defined for classifying component applets according to company- or industry-specific classifications.

The Resource Description Framework, or RDF (http://www.w3.org/TR/WD-rdf-syntax) is a metamodel, to be expressed using XML, for capturing metadata about Web resources. Such metadata could be used by search engines, automated agents, and other Web client and server applications for component searching, rating, access control, licensing, and management.

The Document Object Model, or DOM (http://www.w3.org/TR/WD-DOM), specifies automation interfaces through which scripts or applications can access and manipulate Web documents. Given suitable XML tags, DOM would allow Web documents to be manipulated as components or containers.

These W3C specifications represent a coordinated attempt to define object semantics for lightweight networks of distributed documents. They appear to accommodate component-level semantics as well. W3C efforts align with DCPs in assuming that pervasive metadata enables semantically rich interoperability, whether between components or Web documents.

OMG COMPONENT STANDARDS

The Object Management Group has played a leading role in establishing open system specifications for distributed object computing.[10] Until recently, OMG focused on object-level standards. Responding to ease-of-use concerns from members aligned with the JavaBeans standard, the OMG issued a request for proposal for component-level specifications last year (http://www.omg.org). The RFP identified four core categories of requirements for standards:

- a *component model* that defines a component type system; interfaces for exposing and managing properties; mechanisms for raising and handling events; life cycle structure; and serialization;
- a *component description facility* that consists of a reflective information model supported by existing or new CORBA-related repositories;
- a *programming model* that maps component descriptions to languages that support CORBA IDL mappings, and that lets components be passed as value parameters in CORBA requests; and
- a *mapping to the JavaBeans component model* that supports both design and runtime interoperability needs, including component inspection and the automated generation of software to integrate CORBA components into Java-based tools.

The OMG has several other RFPs in process for standards relating to the component model category. One RFP addresses the problems of composing objects using multiple IDL interfaces (for disjoint services) and of resolving conflicts among multiple interfaces on a given object. A second RFP solicits proposals for interfaces to pass CORBA objects by value parameters in CORBA object operations. Currently, CORBA supports the passing of object parameters only by reference, which impedes the port of RMI from a Java-based protocol to IIOP. A third RFP focuses on scripting languages to support automation for CORBA components.

OMG generates specifications rather than software products or DCPs. Some DCPs, such as JavaBeans, are virtually certain to comply with these standards, while others such as DCOM (ActiveX) probably will not. However, the OMG developed dedicated specifications to ensure interoperability of CORBA and COM objects to reflect Microsoft's dominance of desktop computing. Similar market pressures will likely lead to analogous OMG specifications for the interoperability of DCOM and CORBA components.

COMPONENT FRAMEWORKS

DCPs do not of themselves guarantee complete and satisfactory software applications. They help users construct, reuse, and connect components, but they supply no guidance for application-level semantics or structure—how to design and arrange specific components to solve specific business problems. They also don't guarantee robust, scalable, and agile systems—areas that continue to require considerable engineering experience and discipline.

Component frameworks are an increasingly popular strategy for augmenting DCPs to fill these gaps. A *component framework* is a concrete implementation of one or more design patterns tailored to a particular business or technical domain, such as help desks, health care, or user interface construction.[11] A design pattern

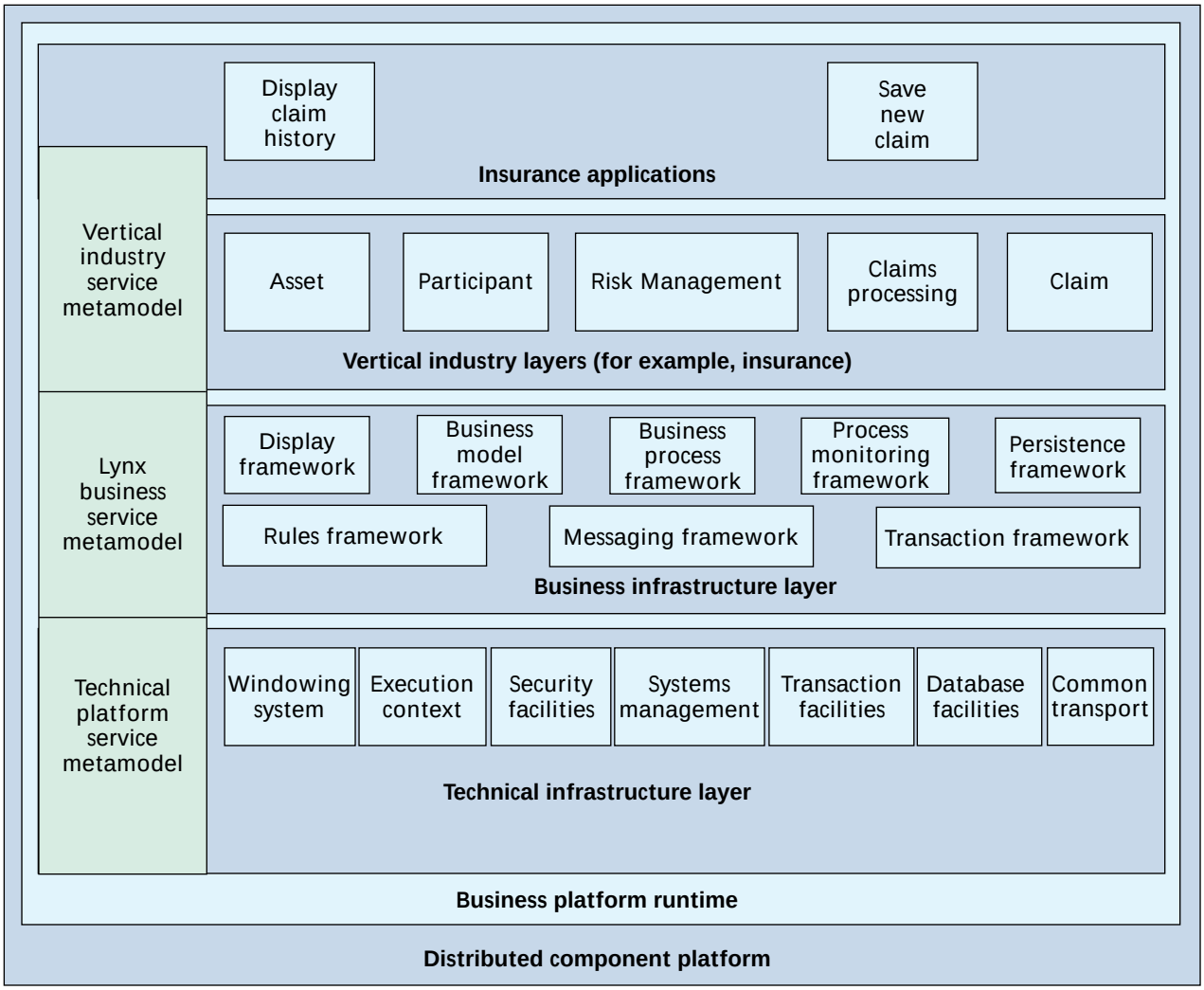

Figure 6. The Lynx framework extends directly out of the DCP substrate. Lynx's prefabricated components are organized into layers that expose a succession of technical and business services. Each layer is segmented into functionally specialized frameworks. At the lowest layer, Lynx simply packaged technical service APIs to achieve high performance and scalability. Each higher layer infuses a more specialized business perspective that hides the complexities of the lower layers. All layers conform to the DCP's uniform programming model.

expresses an abstract solution to a recurring design problem, as a set of components and component interactions.[12] Examples include the Smalltalk Model-View Controller and market auction patterns, which help synchronize graphic displays and allocate scarce resources, respectively. A *framework methodology* guides application developers in determining exactly what to build and how to build it using that framework.

Computer Sciences Corp.'s Lynx framework illustrates how a component framework can extend a DCP into a complete development solution. Lynx provides a prebuilt application skeleton of component templates tied together by collaborative patterns. Developers extend the templates with business-specific components to construct custom applications such as insurance underwriting, toll collection, process control for plastics manufacture, and customer service.

As do all component frameworks, Lynx defines and factors functionality into components and patterns on the basis of specific technical and business design goals. CSC's focus is on large-scale, mission-critical business systems, so Lynx's overarching goal is to support high-volume, online transaction-processing applications. Typical Lynx applications support thousands of users, performing 20 to 30 complex *business work units* per second with subsecond response times. (We use "business work unit" instead of "transaction" to avoid confusion with TPC database transaction metrics. Business work units span multiple tiers, and may encompass a few or dozens of TPC transactions, depending on the application.) Lynx aims to balance performance and scalability against secondary goals of developer productivity and application agility; Lynx lets developers adjust this mix systematically by customizing or extending framework components and patterns to reflect application-specific trade-offs.

As Figure 6 shows, Lynx is actually a framework of frameworks, which collaborate to achieve a set of goals. Lynx attacks its *performance* goal by minimizing the number of process boundary traversals

The use of DCPs for large projects will hinge on the availability of robust component-oriented methodologies.

required to complete business work units: The transaction and message-routing frameworks collaborate to package an entire work unit into a single message. Lynx addresses *scalability* by minimizing the scope and duration of locks against shared services. For example, the messaging framework uses remote service proxies with connectionless service references rather than RPC-style proxy stubs; overall concurrency is maximized by preventing any one client from monopolizing access to shared middle-tier services. Message routing and system monitoring also cooperate for scalability, which allows dynamic replication and distributed load balancing of middle- and back-end services to accommodate increased demand.

Inappropriate coupling between components makes applications less flexible. Lynx minimizes design-time dependencies between components that capture the core business model and those that support presentation, workflow, and database persistence. Isolating components by functional roles means that part of an application can change with little or no effect on the other parts, promoting *agility*. For example, persistence implementations can be tuned or switched without affecting business or presentation components. Isolation also fosters *productivity* because developers can train quickly and specialize on manageable pieces of the framework.

Lynx's layering scheme isolates application-level development from the complexities of the technical infrastructure, which also contributes to agility. To date, we have implemented Lynx on two distinct platforms with minimal architectural and design changes: Forte Software's distributed object development environment and Microsoft's DCOM. Lynx's higher layers preserve and expose both patterns and metadata from lower layers, allowing selective adaptation and tuning. Finally, layering promotes extensibility, both vertically and horizontally. For example, we are incorporating frameworks for finance, process control, legacy integration, business rules, and electronic commerce. In contrast to JavaBeans' constrained properties, the rules framework supports rules that connect multiple attributes across business entities (for example, to constrain the salaries of managers and their employees).

Tightly coupled to the Lynx framework is a methodology that defines the processes, techniques, work products, and management model for using Lynx productively. The methodology adapts and unifies a set of standard analysis methods for static and dynamic object modeling. These methods are synthesized with techniques for modeling and reengineering business processes. The methodology then specifies how to map the resulting business object and process models onto the Lynx framework in terms of components, business services, application tasks, and GUI storyboard layouts. For example, business entities, such as `Policy` or `Claim` are translated into sets of collaborating display, business, and persistence components. Because the default components and collaboration patterns are uniform across business components, Lynx can exploit automated code-generation techniques at the framework level to maximize productivity. Project management processes are driven by a road map that defines development activities, their iteration and sequencing, dataflows, work roles, and team structures.

Frameworks such as Lynx embody architectural blueprints for building component-based applications. Frameworks and their supporting methodologies augment DCPs to minimize risk and help ensure design uniformity and semantic interoperability across applications.

Component software standards continue to evolve, along a fault line formed by rival technology sets from Microsoft and Sun. The distributed component platforms that extend and package these standards are maturing rapidly into commercial products accessible to a broad spectrum of developers.

Interest in component software has grown and intensified in recent years. Business interest has been driven by competitive pressures to deliver agile applications more rapidly and economically. Developer interest derives from the ease of use of the latest IDEs, which approaches the satisfying tactile experience obtained from assembling physical parts. Finally, technical interest has been driven by three converging trends for reuse. The patterns movement fosters reuse at the level of software designs. Component frameworks promote reuse of design patterns and code. Component IDEs enable the construction and deployment of new frameworks, driving reuse of entire application skeletons.

Important challenges remain for component software in enterprise settings. DCPs continue to expose technical complexity to users: Server components and supporting IDE extensions for developing, using, and managing them are in their infancy. Standards for deploying components across intranets to Web clients are similarly immature. In addition, current DCPs support only basic one-to-one interactions between remote components. Such substrates provide meager support for developing the complex coordination patterns required for collaborative workgroup applications, such as voting, negotiation, or sharing.

The use of DCPs for large projects will hinge on the availability of robust component-oriented methodologies.

The value of DCPs to end users depends directly on a critical mass of ready-to-use components. This presupposes a healthy market for third-party components, complete with distribution channels, sustainable pricing models, and standards for packaging and certification. Market growth also depends on some form of stabilization or resolution to the conflict between competing DCPs, most probably through utilities such as the JavaBeans/DCOM bridge. Absent interoperability solutions and standards equilibrium, it will be difficult for component markets to expand.

To date, the majority of off-the-shelf components consist of business-neutral GUI controls, such as spreadsheets and graph or report generators. Components are needed that represent business-level entities and processes. Software vendors and industry consortia such as IBM, Microsoft, and the OMG are actively pursuing generic business components targeted for various vertical markets. Prompt convergence on standards is critical to preventing the proliferation of custom components with incompatible business semantics. Such a trend would obstruct interoperability, reusability, and the growth of component markets.

We believe the ultimate value of software components lies in frameworks that infuse progressively more targeted business-level perspectives into DCPs. The goal is to replace IDE palettes of text fields, data grids, push-buttons, and similar GUI controls with palettes that contain business objects, services, and functional views. Developers would then select, customize, and assemble these items into specialized components such as insurance coverage, and script complex business processes such as order fulfillment. Thus, the real challenge of component software is to let developers build business systems on business component platforms, rather than software systems on software component platforms. ❖

References

1. R. Adler, "Emerging Standards for Component Software," *Computer*, Vol. 28, No. 3, Mar. 1995, pp. 68-77.
2. A. Thomas, "A Comparison of Component Models," *Distributed Object Computing*, July 1997, pp. 55-57.
3. K. Brockschmidt, *Inside OLE*, 2nd ed., Microsoft Press, Redmond, Wash., 1995.
4. N. Brown and C. Kindel, *Distributed Component Object Model Protocol—DCOM/1.0*, Microsoft Corp.; Redmond, Wash., 1996, http:/www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt.
5. *DCOM Technical Overview*, Microsoft Corp., Redmond, Wash., 1996.
6. D. Chappell, *ActiveX and OLE*, Microsoft Press, Redmond, Wash., 1996.
7. "DCOM Architecture," white paper, Microsoft Corp., Redmond, Wash., 1997.
8. *JavaBeans API Specification, Version 1.01*, Sun Microsystems, Mountain View, Calif., 1997.
9. A. Thomas, "Enterprise JavaBeans Server Component Model for Java," prepared for Sun Microsystems by Patricia Seybold Group, 1997; http://java.sun.com/products/ejb/white_paper.html.
10. J. Siegel, *CORBA Fundamentals and Programming*, John Wiley & Sons, New York, 1996. Current OMG specifications can be found on http://www.omg.org.
11. R. Johnson, "Frameworks = (Components + Patterns)," *Comm. ACM*, Oct. 1997, pp. 39-42.
12. E. Gamma et al., *Design Patterns: Elements of Reusable Software*, Addison-Wesley, Reading, Mass., 1996.

David Krieger is the head of development and chief architect for the Lynx framework at CSC Consulting in Waltham, Massachusetts. Previously he worked in product development as a consulting engineer at Lotus Development Corp. and as a manager and technical architect at Bachman Information Systems. His computing interests include large-system architectural patterns, distributed-object frameworks, and high-volume transaction processing. Krieger holds six patents in software modeling and visual development environments. He received a BS in computer engineering from Clemson University and is a member of the IEEE Computer Society.

Richard M. Adler is an architect for the Lynx framework at CSC Consulting in Waltham, Massachusetts. Previously, he directed development of distributed-computing tools at a middleware start-up company and consulted with NASA on intelligent systems for launch support operations. His computing interests include distributed and intelligent systems, software agents, and knowledge management. Adler received a BS and an MS in physics from the University of Michigan and the University of Illinois at Urbana, respectively, and a PhD in the philosophy of physics from the University of Minnesota. He is a member of the IEEE Computer Society and the ACM.

Contact the authors at CSC Consulting, 266 Second Ave., Waltham, MA 02154; {david_krieger, richard_adler}@csc.com.