MODELING COMPONENTS AND FRAMEWORKS WITH UML

For more than a decade vendors have hyped and under delivered the promises of object modeling and component technologies. As a result, object modeling tools are in disrepute in many development organizations, and the semantics of the term "component-based" have been diluted to an extent reminiscent of what occurred to the expression "object-oriented" in the previous software generation.

However, if we put aside our skepticism and evaluate recent advances in these technologies, we find encouraging signs that they are catching up with some of their hyperbole. Many object modeling tools can now generate production-quality skeleton code from structural models, and some of them can also derive useful methods from behavioral models. While they still fall far short of the ambitious goals of full "round-trip engineering," their improvements mark significant progress toward automating the software development process.

Likewise, the leading component standards have evolved from nascent definitions of simple architectures to mature specifications of complete runtime environments. For example, Sun Microsystem's JavaBeans definition has developed into the Enterprise JavaBeans (EJB) specification, which supports both transaction and persistence services for enterprise applications [4]. Similarly, Microsoft's COM definition has evolved into the DCOM and COM+ specifications, and supports the Microsoft Transaction Server (MTS) for enterprise applications [3].

While analyzing the evolution of these technologies it is important to note that strong synergies exist between them. The coarse granularity of components in relation to classes promotes the modularity and reuse goals of object modeling. Conversely, the rigorous specification discipline of object modeling supports the interface-based design and replaceability aims of components. Consequently, it is not surprising that component Integrated Development Environments (IDEs) are converging with visual object modeling tools, a trend that is likely to continue.

This article explores some of the synergies between object modeling and components by examining how the de facto object modeling language standard, the Unified Modeling Language (UML), supports the leading enterprise component archi-

As localized objects evolve into distributed components, developers are asking that UML provide better support for component-based development using EJB and COM+.

CRIS KOBRYN

tecture standards—EJB and COM+/MTS (or COM+ for short). It is presumed the reader is generally familiar with how UML is used to specify software systems; the central focus here is UML component-modeling capabilities.

Components in UML 1.3

The current UML specification, UML 1.3, defines a component as follows:

component: A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files [9].

It is important to note that this definition tends to be restrictive and emphasizes the use of components for implementation modeling (compare this with analysis and design modeling). This point will be addressed later in this article.

The UML notation for components is summarized in Figure 1. In this diagram a ShoppingCart component for an e-business application is shown as a rectangle with two smaller rectangles protruding from its side. At the top of the diagram the component is shown with attributes, operations, and the interfaces *Calculate* and *ChangeOrder*, which are drawn using longhand interface notation (rectangles that contain an operations compartment). At the bottom of the diagram the component is shown in a more compact representation where the attributes and operations are elided and the interfaces are drawn using shorthand ("lollipop") interface notation.

Although UML components may be shown in any structural modeling diagram, they are typically found in implementation model diagrams, such as component diagrams and deployment diagrams. A component diagram shows the organization and dependencies of components, and a deployment diagram shows how component and class instances are deployed on computational nodes. An example of a deployment diagram that uses components is shown in Figure 2.

The example shows a simple failover scenario for

Figure 1. Component and interface notation.

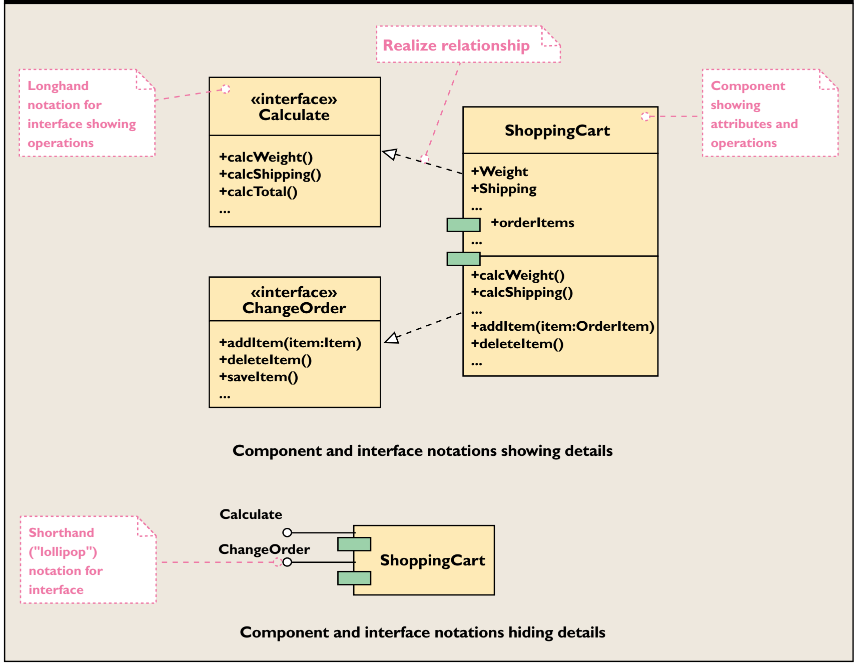


Figure 2. Deployment diagram example.

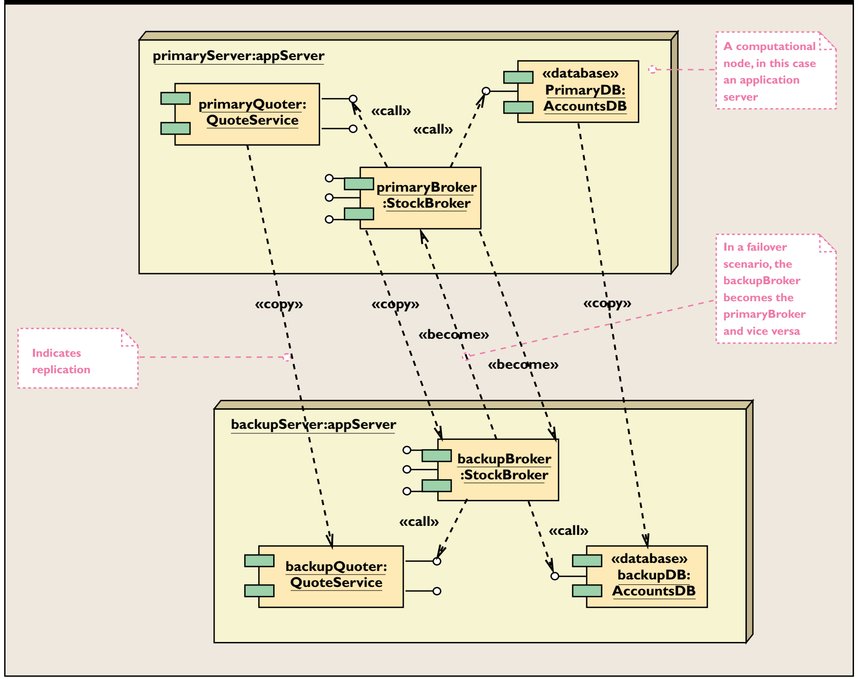

primaryServer:appServer
primaryQuoter: QuoteService
«call»
«call»
«database» PrimaryDB: AccountsDB
A computational node, in this case an application server
primaryBroker :StockBroker
In a failover scenario, the backupBroker becomes the primaryBroker and vice versa
«copy»
«copy»
«copy»
«become»
«become»
Indicates replication
backupServer:appServer
backupBroker :StockBroker
«call»
«call»
backupQuoter: QuoteService
«database» backupDB: AccountsDB

Table 1. Semantic comparison of components, subsystem, and classes.

	COMPONENT	SUBSYSTEM	CLASS
Is a classifier?	+	+	+
Can have operations?	+	+	+
Can have methods?	–	–	+
Can have attributes?	+	–	+
Can have interfaces?	+	+	+
Can be associated?	+	+	+
Can have thread of control?	–	–	+
Can be nested?	+	+	+
Is a grouping construct?	–	+	+
Can import/access?	–	+	–
Represents a unit in a physical system?	+	+	–
Contains implementation of model elements?	+	–	–
Can create instances?	+	+ (optional)	+
Instances typically reside on nodes?	+	–	–

an online stockbroker system, where several components deployed on an instance of the primary application server node (*primaryServer*) are replicated on the backup application server node (*backupServer*). On the *primaryServer* the *primaryBroker*, an instance of the component *StockBroker*, calls the interfaces of the *primaryQuoter* (an instance of the *QuoteService* component) and the *primaryDB* (an instance of the *AccountsDB* component). Component replication and migration are shown by «copy» and «become» dependency flows, which are part of standard UML.

Semantic Overlap: Components, Classes, and Subsystems

The fact that UML is a general-purpose modeling language is both a strength and a weakness. Although software developers find they can use UML to model most software problems in a variety of ways, they are frequently overwhelmed by their options. The problem is exacerbated by the dearth of pragmatic examples and the lack of tools that support advanced constructs. Many of the examples in UML modeling books tend to be trivial or academic, and frequently do not address the practical problems faced by developers. As for modeling tools, the author knows of none that fully implements the UML 1.1 semantics and notation (adopted three years ago), let alone one that completely or correctly implements the current UML 1.3 specification (which was adopted a year ago).

Consequently, many modelers find the modeling of components, which is sometimes considered an advanced UML modeling topic, problematic. One of the most common problems modelers encounter is the semantic overlap between components and related classifiers, such as classes and subsystems, which are defined as follows:

class: A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment [9].
subsystem: A grouping of model elements that represents a behavioral unit in a physical system. A subsystem offers interfaces and has operations. In addition, the model elements of a subsystem can be partitioned into specification and realization elements [9].

The semantics of components, subsystems, and classes are compared in Table 1. The table shows that all three classifiers can have operations and interfaces, may be associated with other classifiers, can be nested, and may create instances. Components are similar to subsystems and differ from classes in that they cannot have threads of control and they represent units in physical systems. Components differ from subsystems and classes because they are not a grouping construct; they alone can contain the implementation of model elements, and their instances typically reside on computational nodes. Only subsystems can import or access other model elements.

It is important to note that, while all three classifiers may conform to a set of interfaces, interfaces are usually most closely associated with components. Although it is relatively common to see a class without an interface during analysis, and subsystems may use model elements other than interfaces for specification (for example, use cases and statecharts), a component without an interface may be technically well-formed but suspect. This reflects the longstanding and intimate relationship between component-based development and interface-based design. For example, both COM+ and the CORBA Component Model (CCM) are closely associated with Interface Definition Languages (IDLs). In addition, both the Java and EJB specifications emphasize interface-based design.

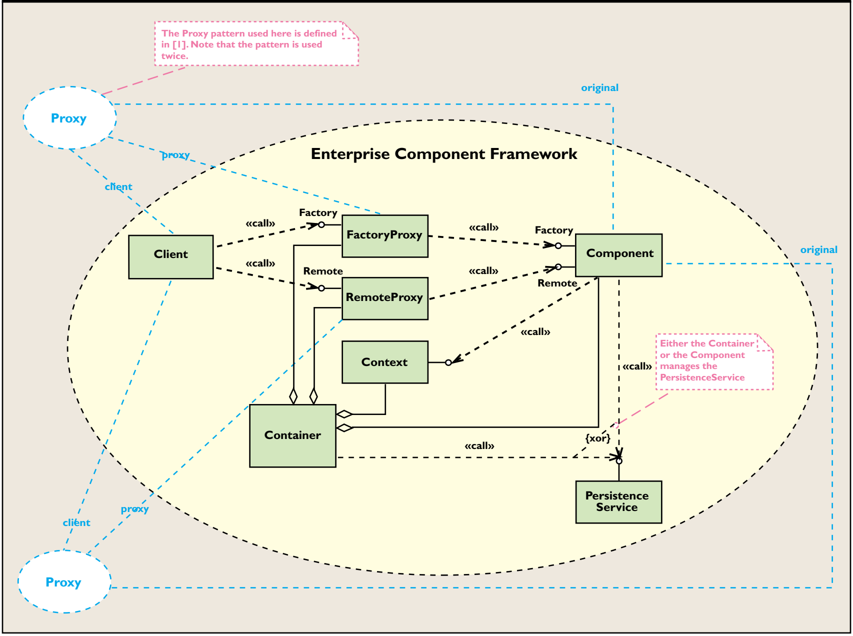
When choosing among these related classifiers modelers may also find it useful to consider some usage heuristics. Although the UML is a modeling language and not a software method, the specification does provide some usage notes regarding how some constructs can be used during various phases in the software life cycle. Other heuristics may be gleaned from component methods, specifications, and applications. With the caveat that these are basic guidelines and not rules, Table 2 summarizes some heuristics for applying the classifiers in question.

In general, components and subsystems tend to be more coarse-grained than classes. Indeed, it is common for a component to implement multiple design classes. Similarly, it is typical for a subsystem to model the specification and realization of a set of model elements, which may include both specification types and implementation classes. Components are further distinguished from both subsystems and classes in that they are typically modeled during the implementation phase (compare analysis and design phases). It is important to note, however, that the classes and/or subsystems specifying components and frameworks may be modeled during an earlier life-cycle phase, such as analysis or design.

Table 2. Usage heuristics for components, subsystem, and classes.

	COMPONENT	SUBSYSTEM	CLASS
Tend to be coarse-grained?	+	+	–
Typically modeled during analysis?	–	+	+
Typically modeled during design?	–	+	+
Typical modeled during implementation?	+	–	+

Figure 3. Enterprise component framework pattern.



Modeling Component Frameworks

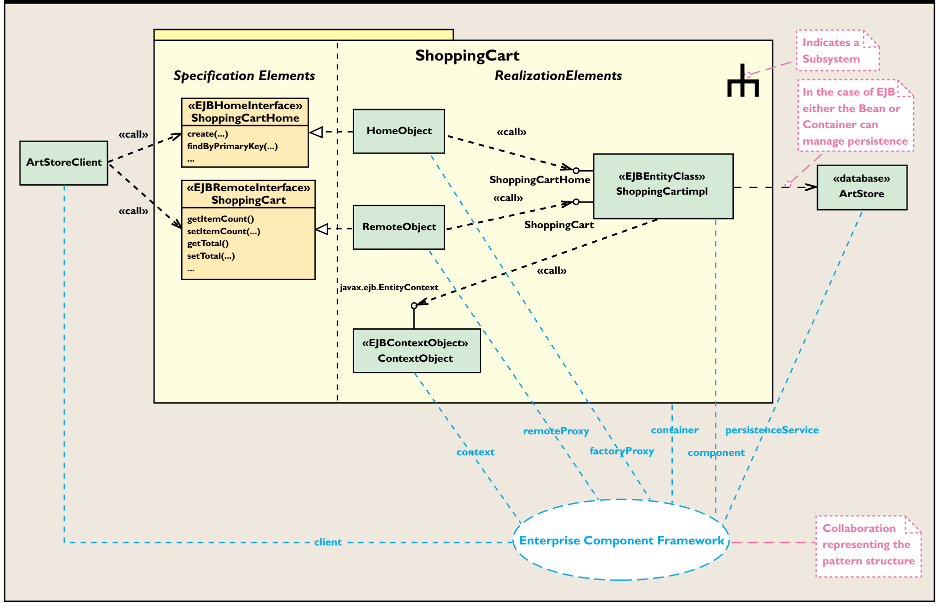
Components are quintessential software building blocks that can be recursively composed to build systems of increasing size and complexity. Just as the hardware industry has used integrated circuits to recursively build larger and more complex hardware components, the software industry now endeavors to do something similar with software components. Component frameworks are important mechanisms being used to accomplish this.

A framework is a generic term for a powerful object-oriented reuse technique that typically emphasizes the reuse of design patterns and architectures. One common definition is that "a framework is a reusable design of all or part of a system represented by a set of abstract classes and the way their instances interact." Another frequently used definition is that "a framework is the skeleton of an application that can be customized by an application developer" [5]. These definitions are complementary, not conflicting, since the former describes a framework from a design perspective, whereas the latter describes it from a functional viewpoint. However, the differences between the two definitions point out the variety of ways in which the term is used.

Although the UML definition of a framework is more restrictive than the preceding definitions, it is compatible with them:

framework: 1) A stereotyped package consisting mostly of patterns. 2) An architectural pattern that

Figure 4. Application of pattern to EJB.



provides an extensible template for applications within a specific domain.

The first definition is less than illuminating, since it explains the concept in terms of the UML construct (the «framework» stereotype of Package) with sparse additional semantics. The second definition is more informative and reinforces the associations between frameworks, patterns, and architectures.

As one might intuit, component frameworks are frameworks typically designed, constructed, and extended using components. Since a comprehensive discussion of component frameworks is beyond the scope of this article, readers are referred to [5] for more information about them. The remainder of this section focuses on two component frameworks that are industry standards for building enterprise applications: Enterprise JavaBeans and COM+.

Defining a common pattern. The architectures of Enterprise JavaBeans and COM+ are based on a common architectural pattern that we call the *Enterprise Component Framework*. The purpose of this pattern is to describe the basic mechanisms for a component runtime environment that supports distributed services for interprocess communication, transactions, and persistence.

For the purposes of this article a pattern is defined as a common solution to a recurring problem in a particular context. UML 1.3 prescribes that the structure of patterns should be modeled using parameterized collaborations, but does not prescribe the modeling of non-structural aspects, such as behavior and heuristics for applying patterns. As was the case for the framework concept, there are diverse opinions regarding the definition and use of patterns. In consideration of this and the scope of this article, we will restrict our modeling of patterns to UML parameterized collaborations.

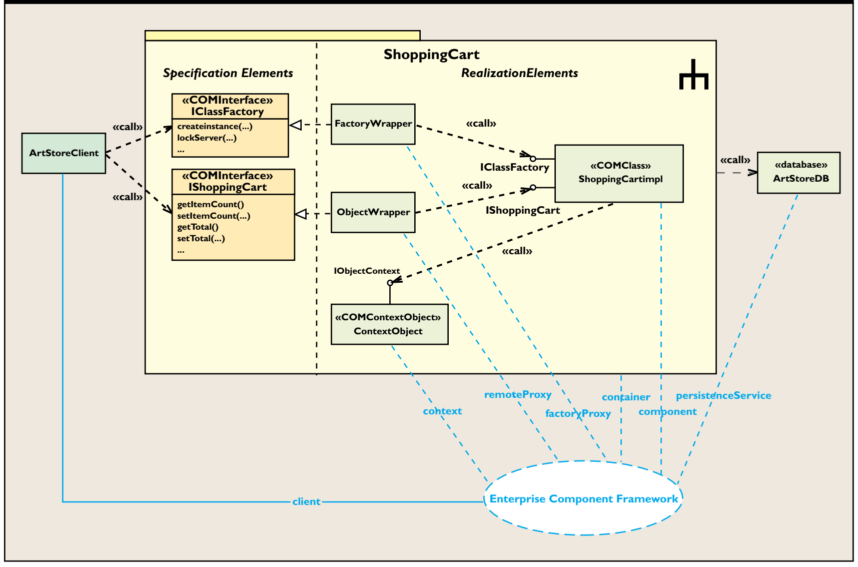
The structure of the Enterprise Component Framework pattern is modeled as a parameterized collaboration in Figure 3, where the collaboration is shown as a dashed ellipse. This pattern contains seven classifier roles that participate in the collaboration, which are depicted as rectangles: Client, FactoryProxy, RemoteProxy, Context, Component, Container, and PersistenceService. The Client represents an entity that requests a service from a Component in the framework.

An important aspect of this pattern is that the Client does not call the Component directly. Rather it communicates indirectly via a pair of proxy roles (FactoryProxy and RemoteProxy) that delegate (relay) calls from the Client to the Component. This level of indirection supports two important functions:

- **Location transparency:** At a lower level of abstraction the proxies may be realized by stub-skeleton pairs, such as those used by CORBA, Java RMI, and COM+.
- **Message interception:** The use of proxies allows the component's runtime environment (the Container) to intercept method calls and insert services based on a set of attributes defined at deployment time.

The roles of FactoryProxy and the RemoteProxy as surrogates are made explicit by the use of two Proxy sub-patterns in the main pattern. These sub-patterns are shown as dashed ellipses, but their classifier roles are elided to reduce distracting detail. Readers interested in details about the structure and behavior of the Proxy pattern are referred to [1] and [6]. The bindings of the roles of the Proxy pattern described in [1] to their counterparts in the main

Figure 5. Application of pattern to COM+.



pattern are shown by dashed lines that are labeled with role names. For example, the roles *client*, *proxy*, and *original* from the Proxy pattern shown in the top left of the diagram are bound to the classifier roles Client, FactoryProxy, and Component in the main pattern. Likewise, the roles *client*, *proxy*, and *original* from the Proxy, pattern shown in the lower left of the diagram are bound to the classifier roles Client, RemoteProxy, and Component in the main pattern.

The FactoryProxy and RemoteProxy surrogates perform distinctive roles in the pattern. The former proxy handles object factory operations such as *create* and *find*, while the latter proxy handles business operations specific to the Component (for example, getItemCount, setTotal). The separation of concerns between the proxies is consistent with UML's classifier-instance dichotomy: the FactoryProxy facilitates class methods, while the RemoteProxy facilitates instance methods.

The Component and both proxy roles are held in a component Container, where the containment is shown by aggregation relationships (lines with unfilled diamonds at the aggregation end) between the Container and its contents. The Container represents the framework's runtime environment, which supports various distributed processing services, such as interprocess communication, security, transactions, and persistence. The Container also includes a Context entity for each Component that maintains specific context information, such as the states of transactions, persistence, security, and deployment.

Persistence services for the Component (for example, database storage and retrieval) can be coordinated directly by the Component or can be delegated to the Container. This choice is shown by the {xor} constraint on the two calls to the PersistenceService in the diagram. We will return to this point in the next section, where we will use it to illustrate an important difference between the EJB and COM+ architectures.

Applying the pattern. The common solution the Enterprise Component Framework pattern provides for distributed component architectures can be demonstrated by applying it to model EJB and COM+ examples. We first apply it to an EJB example in Figure 4, which shows a ShoppingCart component for an e-business application modeled as a UML subsystem. The pattern being applied is shown as a dashed ellipse in the lower right of the diagram, and the classifiers to which the pattern roles are being applied are shown above the pattern.

It is noteworthy that the names of several of the classifiers in the collaboration are preceded by UML keywords, which are marked by guillemet delimiters (for example, «EJBEntityClass»). The keywords here indicate that these classifiers are UML stereotypes (that is, user-customized extensions to the language) defined external to the model and not part of standard UML. These stereotypes highlight areas where UML needs to be customized to meet the specific needs of particular component architectures, such as EJB or COM+. For example, Java and EJB interfaces are defined as stereotypes of the UML metaclass Class, since Java and EJB interfaces may declare constants, whereas UML standard interfaces cannot.

The dashed lines between the pattern and the classifiers are labeled with the roles that the classifiers play in the collaboration: client, context, remoteProxy, factoryProxy, container, component, and persistenceService. These roles are respectively bound to the following EJB classifiers: ArtStoreClient, «EJBContextObject» ContextObject, RemoteObject, HomeObject, ShoppingCart, «EJBEntityClass» ShoppingCartImpl, and «database» ArtStoreDB.

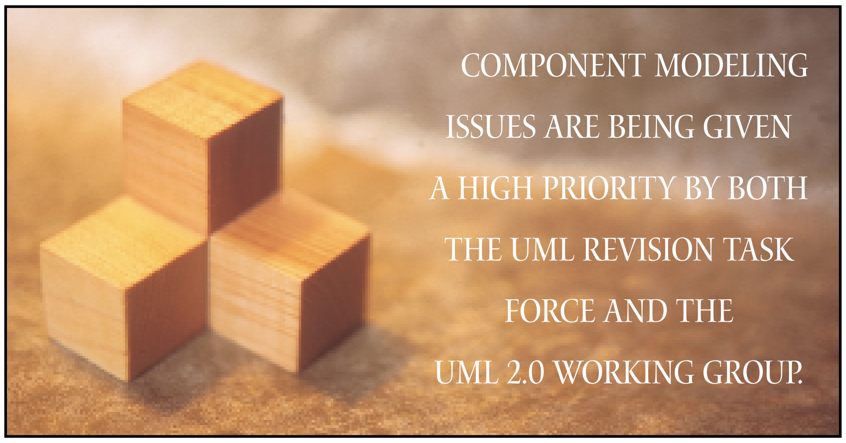
In the EJB example, the subsystem calls the ArtStoreDB directly for persistence services (for example, database stores and retrievals), indicating that the container will manage this at runtime.

In order to facilitate comparison between the two component architectures, the Enterprise Component Framework pattern is also applied to a COM+ example in Figure 5. As was the case when applying the pattern to the EJB example, the pattern roles map to classifiers in a straightforward manner. Although this confirms that there are strong structural similarities between the two component architectures, we note the following differences:

- **Stereotypes usage.** The different stereotypes used in the two examples indicate areas where the two architectures are likely to vary. Although an analysis of these detailed differences is beyond the scope of this article, readers are encouraged to perform this comparison.
- **Persistence services.** Whereas in the case of EJB, either the component or the container may coordinate persistence services, in the case of COM+ only the component can coordinate persistence services. This is a significant architectural difference between the two frameworks.

Alert readers will notice the UML component construct was not used to specify either application of the framework pattern. This points out the rather limited semantics and restricted use of the current component construct, which is mostly used in implementation model diagrams. Since the pattern application examples are design models and not implementation models, they do not show the use of the UML component construct.

However, in both cases it is relatively straightforward to generate implementation models that use the UML component construct. For example, the «EJBEntityClass» ShoppingCartImpl in Figure 4 might be implemented on a «EJBEntity» component that executes in an «EJBContainer» component running on an application server node in a deployment diagram. Similarly, the «COMClass» ShoppingCartImpl in Figure 5 might be implemented on a «COMObject» component that executes in an «MTS Executive» component running on an application server node in a deployment diagram.


COMPONENT MODELING ISSUES ARE BEING GIVEN A HIGH PRIORITY BY BOTH THE UML REVISION TASK FORCE AND THE UML 2.0 WORKING GROUP.

Issues and Recommendations

Although the preceding examples show that the UML 1.3 specification can effectively model many aspects of components and frameworks, they have also identified some significant issues. In addition, users and vendors have identified many other problems as they apply standard UML and custom profiles to specify large and complex component applications. Some of these issues, along with recommendations to resolve them, are discussed here.

Increase clarity and reduce overlap. The current semantics for the component construct are vague and, as pointed out previously, overlap the semantics of related classifiers, such as class and subsystem. In order to fully support component-based development, the semantics of components should be refined and the overlap with related constructs should be reduced.

Support components at an earlier phase of the software life cycle. The semantics for component are dated and emphasize implementation diagrams, which typically occur at the tail end of the modeling process. These semantics should be updated so that modelers can specify components earlier in the software life cycle (for example, during design).

Define model management constructs that support large component systems and frameworks. As component applications grow and proliferate, so will the need for abstractions to manage their size and complexity. Consequently, UML model management constructs should be refined, extended, or augmented to support large component systems and frameworks. These constructs include, but are not limited to, containers, frameworks, and subsystems.

Clarify how components and interfaces are "wired." When combining components with various entities that realize or implement interfaces (classes, implementation classes, and subsystems) it is not always clear how to connect them correctly, especially when they are heterogeneously nested. The specification should include better guidelines and examples for mixing and matching these constructs.

Provide standard UML profiles for specific component technologies. Standard UML profiles should be defined for specific component technologies, such as EJB, COM+, and CCM.

Component modeling issues are being given a high priority by both the UML Revision Task Force (UML RTF) and the UML 2.0 Working Group. The UML RTF will make clarifications and corrections within the scope of a minor revision in their recommendations for UML 1.4, which they expect to be finalized and adopted later this year. Larger issues that are outside the scope of a minor revision will be deferred to the UML 2.0 Request for Proposals, which is scheduled to be issued later this year. Improvements to support the modeling of EJB components and frameworks may be further facilitated by the informal liaison between the OMG UML RTF and the Java Community Processes expert group for the UML/EJB Mapping Specification [8].

Conclusion and Future

The current UML 1.3 specification provides basic support for modeling components and component frameworks. Users can specify components in various ways, including those outlined by software methods that support component-based development [2, 7]. In addition, tool vendors can customize UML profiles that support the round-trip engineering of EJB and COM+ components.

However, there are also substantive issues related to modeling components with the current UML 1.3 specification. These range from restrictive yet overlapping semantics to the lack of robust model management constructs and component technology profiles. We hope the OMG UML Revision Task Force and the UML 2.0 Working Group will address these issues in an effective and timely manner.

As we await these improvements we should keep in mind that component technology is still at the beginning of its adoption curve. As it enters more into mainstream business computing we can expect it to have a dramatic impact on how we design, construct, and deploy software systems. We have every reason to believe that UML will evolve along with components to meet their special needs, and will not be surprised if future modelers think of UML as a component-based, rather than an object-oriented, modeling language. C

References

1. Buschmann, F., et al. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, NY, 1996.
2. D'Souza, D. and Wills, A.C. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, Reading, MA, 1999.
3. Eddon, G. and Eddon, H. *Inside COM+ Base Services.* Microsoft Press, 1999.
4. Enterprise Java Beans Specification, v. 2.0. Public draft, Sun Microsystems, May 2000.
5. Fayad, M., et al. *Building Application Frameworks.* Wiley, NY, 1999.
6. Gamma, E., et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.
7. Jacobson, I., et al. *The Unified Software Development Process.* Addison-Wesley, Reading, MA, 1999.
8. UML/EJB Mapping Specification, Java Specification Request ID# 000026, Java Community Process, Sun Microsystems, July 1999.
9. UML Revision Task Force, OMG Unified Modeling Language Specification, v. 1.3, document ad/99-06-08. Object Management Group, June 1999.

Cris Kobryn (ckobryn@acm.org) is the chief scientist and a senior director at InLine Software (www.inline-software.com). He is the co-chair of the UML Revision Task Force and the co-chair of the Analysis and Design Platform Task Force at the OMG.

WEB REFERENCES

www.celigent.com/omg/umlrtf is the OMG UML Revision Task Force home page. Contains UML specification artifacts, including the latest UML specification and drafts of works-in-progress. Also includes a link to the UML 2.0 Working Group page.

java.sun.com/products/ejb/ is the Enterprise JavaBeans home page. Contains links to diverse resources related to EJB, including specifications, white papers, and tutorials.

java.sun.com/aboutJava/communityprocess/ is the Java Community Process home page. Contains information about the JCP process, specification proposals, calls for experts, and specification drafts.

www.microsoft.com/com/tech/complus.asp is the COM+ home page. Contains links to a wide range of resources related to COM+, including white papers, presentations, and books.