Communications of the Association for Information Systems

Volume 9 Article 12

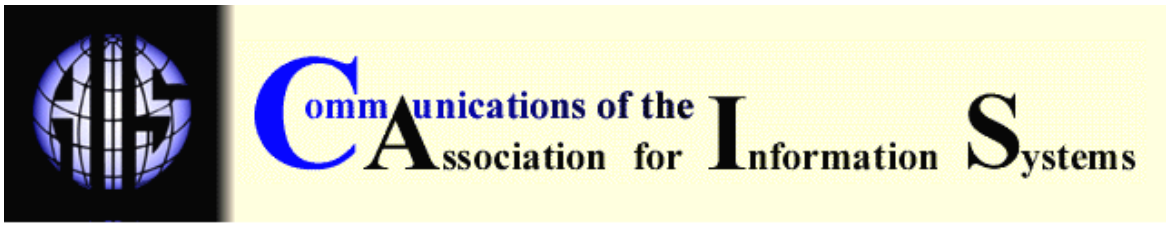
October 2002

Component-Based Development Using UML

Luyin Zhao
Philips Research USA, luyin.zhao@philips.com

Keng Siau
University of Nebraska-Lincoln, siauk@mst.edu

Recommended Citation
Zhao, Luyin and Siau, Keng (2002) "Component-Based Development Using UML," *Communications of the Association for Information Systems*: Vol. 9 , Article 12.
DOI: 10.17705/1CAIS.00912
Available at: https://aisel.aisnet.org/cais/vol9/iss1/12



COMPONENT-BASED DEVELOPMENT USING UML

Luyin Zhao
Healthcare Systems & IT
Philips Research USA

Keng Siau
Department of Management
University of Nebraska-Lincoln
ksiau@unl.edu

ABSTRACT

Component-based software development (CBD) is a potential breakthrough for software engineering. Unified Modeling Language (UML) can potentially facilitate CBD design and modeling. Although many research projects concentrate on the conceptual interrelation of UML and CBD, few incorporate actual component frameworks into the discussion, which is critical for real-world software system design and modeling. This paper reviews component-based development, including the use of UML for modeling CBD. The paper then discusses the means by which UML extension mechanisms can be used to better support the popular component framework -- CORBA. Two other important component frameworks, DCOM and Web Services, are also discussed.

KEYWORDS: Component-based development, UML, CORBA, DCOM, Web services

I. COMPONENT-BASED DEVELOPMENT

Software engineering faces new challenges with the increasing complexity and length of the software development cycle. In the search for alternative methods to develop software more efficiently and with higher quality, component and component-based development (CBD) for software reuse is a key approach [Ben-Shaul *et al.*, 1999; Brown and Wallnau, 1998; Norris *et al.*, 2000].

Component-based development is a software development approach in which all aspects and phases of the development lifecycle, including requirements analysis, design, construction, testing, deployment, and project management, are based on components [Herzum and Sims, 1999].

CBD evolved from the object-oriented methodology that encapsulates internal details of objects, and allows external applications to know and use the objects' interfaces. Over time, it proved difficult for objects developed with different languages, platforms, and running environments to work together (interoperability), and it was found that this difficulty impedes software reuse.

Therefore, distributed-object computing (DOC) was introduced, with CORBA, COM/DCOM, and JavaBeans as prominent examples [Hopkins, 2000]. Even though there are heated debates regarding whether CORBA or COM/DCOM are truly component-enabling frameworks, they are among the most popular component-enabling frameworks [Kozaczynski and Booch, 1998]. With the improvement of existing frameworks (e.g., the release of CORBA 3.0) and the emergence of Web Services and .NET architecture, we expect better component support in the near future.

WHY CBD?

Similar to plug-and-play computer hardware components that allow assembling a new computer in 10 minutes with little knowledge of the components, we would like to see such ease in the software industry even though software characteristics differ from hardware.

Component-based software development changes the way applications are developed. With components as building blocks, applications could be "assembled" with reusable pieces of software, thus reducing a large amount of work required for software design and implementation. The reuse of proven components also helps to achieve higher reliability and maintainability for the system.

For example, suppose your organization is using a large-scale business application that contains an accounting module. This application is likely to be expensive to replace. Further complicating matters, suppose that the software vendor informed your organization that the currently expensive accounting module will soon become outdated because of a new accounting standard proposed by Congress. CBD is a solution to this dilemma. If this application was developed using the CBD approach, the IT professionals in your organization would simply "remove" the current component, purchase anew accounting component module from an accounting software vendor, and "plug" the new component into the existing system. It could be as easy as replacing a computer monitor. Ideally, the organization could even buy the usage for this accounting component rather than purchasing the software component. The latter approach requires that organizations request the new accounting component through a network-based vendor, and receive only the desired functionality (Figure 1).

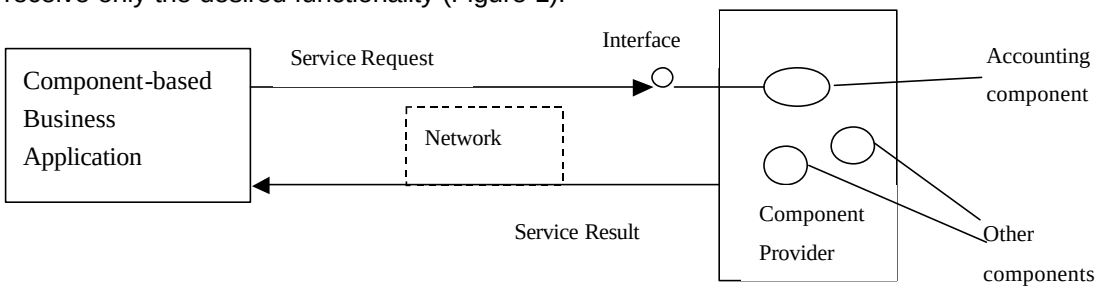


Figure 1. Component-Based Application

Although an appropriate business model for CBD is still under debate, and opinions about CBD differ, CBD does propose a facilitated approach to replacing old software components whenever new modules from the same vendor are released. This approach eliminates the need to reinstall the entire software system. CBD also renders legacy and other useful software components reusable in new systems.

Traditional software engineering methodologies execute analysis, design, implementation, and testing without capitalizing much on reusable components. CBD, with its focus on reuse, can have major impacts on software engineering. For example,

- CBD can reduce quality assurance work and lead to higher reliability and lower cost.
- CBD can reduce design work and allow developers to focus more on business problems.
- CBD can simplify implementation by using components with easy-to-use interfaces.

The design phase can benefit much from CBD. Design is the conceptual design of a system's behavior in terms of services, interfaces, and interactions. It is normally considered a critical phase in software engineering. Traditional design methodologies may not support CBD well. UML, on the other hand, with its comprehensive set of diagramming techniques and extension mechanisms, is helpful for CBD. Nevertheless, component-based systems depend heavily on the specific framework on which they are built. Therefore, in extending UML to support CBD, the developer needs to take into account the unique features of the component framework. This consideration also helps to simplify the transition from design to implementation (for example, source code generation).

II. COMMON COMPONENT FRAMEWORKS AND UML

The discussion of CBD modeling using UML in books and articles is usually high-level and generic [Kobryn, 2000; Barn, 1998]. Typically, no specific component framework is involved. Users who use components in their specific environment may encounter trouble implementing those ideas. In this paper, we discuss UML extensions that will support CBD design for three of the most popular frameworks: CORBA, DCOM, and Web services. Before discussing the means by which UML can be used to support CBD design under these frameworks, we first provide a brief review of CORBA and DCOM.

At first glance, CORBA and DCOM appear to be quite similar: both are component-based frameworks although developed by different organizations. CORBA is the standard proposed by the Object Management Group (OMG) whereas DCOM is the standard offered by Microsoft.

OMG COMMON OBJECT REQUEST BROKER ARCHITECTURE (CORBA)

OMG Common Object Request Broker Architecture (CORBA) is an open standard solution for distributed object computing. Basically, CORBA provides a platform for reusable components on heterogeneous environments to communicate and interoperate. An organization can use this middleware to build distributed component-based applications whose composite parts run on different machines [Mowbray and Ruh, 1997; OMG, 2000; Pope, 1997]. As shown in Figure 2, ORB (Object Request Broker) is a software bus connecting the client and the server. The Interface Repository stores all IDL (Interface Definition Language) interfaces provided by server objects. The Implementation Repository contains mapping information of server objects and executable files. The client makes service invocations to the server using either static Stub or Dynamic Invocation Interface (DII). Correspondingly, the server uses static Skeleton or Dynamic Skeleton Interface (DSI) to deliver invocations to object implementations. All invocations transmitted through ORB are in implementation-independent formats.

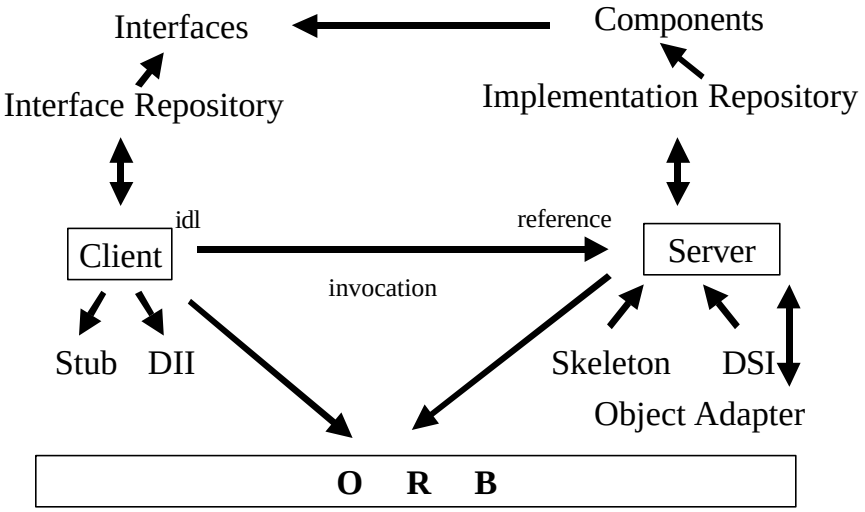


Figure 2. Common Object Request Broker Architecture

MICROSOFT DISTRIBUTED COM (DCOM)

Microsoft Distributed COM (DCOM), like CORBA, is a binary and network standard that allows any two components to communicate regardless of the machines they are running on (as long as the machines are connected), the operating systems (OS) the machines are running (as long as the OS supports COM), and the languages the components are written in. Figure 3 shows the architecture of DCOM. Similar to CORBA, DCOM uses a proxy object and stub that are counterparts of the CORBA stub and skeleton. MIDL (Microsoft IDL) is used to describe interfaces provided by the server. DCOM also uses a dynamic invocation method called COM Automation (not shown on the diagram).

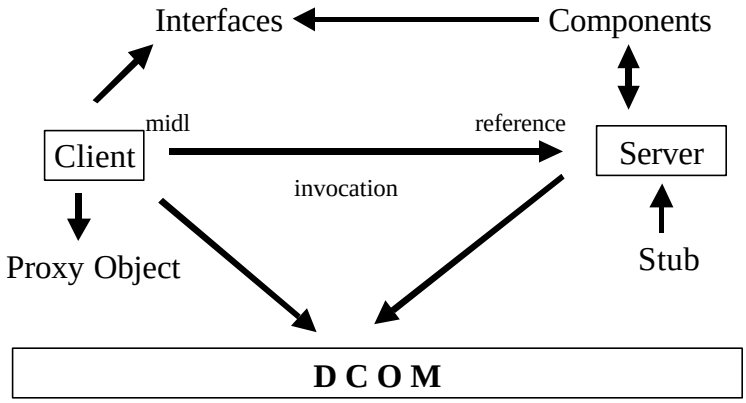


Figure 3. Distributed COM

The reason we discuss CORBA and DCOM here is that they are the two most popular component frameworks. Comparing these two frameworks is not the purpose of this paper. They both provide frameworks for components to "live" on. In the following sections, we use CORBA and DCOM to illustrate how UML can be extended to support component-based modeling.

III. CBD DESIGN USING UML

Unified Modeling Language (UML) is a visual language for visualizing, specifying, constructing, and documenting the artifacts of a system-intensive system [Booch *et al.*, 1999, p. xv]. UML provides a number of diagrams for modeling a system from different points of view. UML is now the de facto standard for object-oriented software system design.

Unlike regular software development, CBD depends largely on the underlying supporting framework. Therefore CBD brings new issues to modeling. UML is suitable for CBD modeling because of its existing support for component and its extension mechanisms that provide the much-needed flexibility. In the following sections, we discuss the current UML support for CBD modeling, and the limitations of UML and possible extensions.

CBD DESIGN

Although CBD design contains many similarities to normal software design, its unique characteristics should be considered. That is, the general software design methodologies must be tailored to meet the special needs of CBD.

Interface

The Interface is one of the most critical concepts in CBD since it represents behaviors presented to the outside world by a component. Components are different from classes for two reasons. First, with interfaces, components represent the physical packaging of otherwise logical components. Second, components and classes are at different levels of abstraction [Booch *et al.*, 1999]. Therefore, interfaces constitute the boundary between the component framework and application layer.

For example, once a CORBA component is encapsulated and described by IDL, external applications that use CORBA components can only see the interfaces provided by them. For this reason, component-based systems design requires good support for interface modeling. In CBD modeling, the "bridges" that connect components are interfaces.

Component Framework

Component Framework is another important aspect of CBD. Components lose their "bed" without it. Currently CORBA and DCOM are among the most popular frameworks. Traditional design techniques are poorly suited to the requirements of component-based systems. They offer little in terms of techniques and guidance for defining and using interfaces as key design abstractions. Even though IDL provides a good way to describe component interfaces, it is basically intended for independent component description and is far from sufficient in a full-fledged component-based design process.

UML

Unified Modeling Language (UML), the standard language for modeling object-oriented systems, with its easy-to-understand graphic representations, is a good tool to model and build component-based systems. First, UML provides a component diagram and interface modeling mechanism that meet the most fundamental requirements of "interface-focused design." Second, UML extension mechanisms (stereotypes, tagged values, and constraints) provide the flexibility in extending UML semantics and tailoring UML for use in framework-specific component-based systems. More specifically,

- Stereotype allows creating new modeling elements (meta classes) for the modeling.
- Tagged values are key-value pairs associated with modeling elements. The main reason to "tag values" is that these values need to be recognized at the modeling stage.
- Constraints are rules that define the conditions that must be specified in the model.

In the following subsections, we address the following two questions:

- Which UML diagrams are related to CBD?
- How can we use UML extension mechanisms for CBD under CORBA and DCOM frameworks?

CBD RELATED UML DIAGRAMS

UML provides a single, broad view of a component as a physical and material element of a system that can reside on a node. A component is defined as a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Furthermore, with the help of the UML stereotyping mechanism, the logical description of the interface can be modeled as a special kind of a class [Booch *et al.*, 1999].

First, we examine the means by which some UML diagrams and their possible *variants* are related to CBD modeling:

UML Package

There are similarities between the component concept and the UML package concept. A package is a more generic and loosely coupled component. A natural extension of a package is to model the integration or composition among several components, which is derived from basic package semantics.

Class Diagram

After defining use case diagrams, a set of interfaces and interactions among components based on the framework they rely on should be defined to support the required behavior. This stage characterizes a component modeling approach.

Interfaces provided by components are widely defined by IDL. UML class diagrams, distinct from IDL, describe interfaces, classes, collaborations, and relationships in a more architectural, systematic, and easy-to-understand manner than text-based IDL. Moreover, using UML stereotypes, different component interfaces can be modeled as interface classes.

Component Diagram

UML component diagrams are used to highlight the organization and dependencies among a set of components. The most representative components are Microsoft DLL files or CORBA components.

Component diagrams are closely related to CBD modeling. However, current UML component diagrams are too simple to model complex component-based systems since only three generic modeling elements are defined: component, interface, and dependencies. UML component diagrams do not capture information related to component frameworks, which could be important for CBD modeling because design models that are too generic provide little help to CBD implementations that rely heavily on component frameworks. Another drawback is that the relationship between a component and the interfaces it provides are loosely coupled in component diagrams.

Deployment Diagram

UML deployment diagrams define the physical architecture of the system, that is, how physical components are deployed to execute on particulate nodes in the system.

UML diagrams presented in this section can play important roles in CBD modeling. Deviations and extensions are needed for modeling CBD more adequately, particularly if there is certainty regarding which component framework a specific system is going to use. Creating new elements and performing framework-specific extensions can greatly facilitate CBD modeling.

IV. UML EXTENSION FOR CBD MODELING

UML extensions can be used for customizing and extending UML [Alhir, 1999; Baumeister *et al.*, 1999; Siau and Cao, 2001; Siau and Halpin, 2001]. UML defines properties for each modeling element and a means for adding new types of model elements and for modifying the properties of existing model elements. Since each system has different properties that differentiate it, (i.e., whether it is a distributed application, a real-time application, or a business-oriented application) extensions can be tailored for each case. In other words, systems with different types/properties can use new domain-specific modeling elements created by extending UML. Extension is critical because the modeling tool must be flexible enough to capture sometimes subtle differences in systems; one size really does not fit all when it comes to design tools.

The UML concepts can be used for CORBA modeling. However, providing framework-specific stereotypes for some common situations provides a common terminology for this domain. More importantly, new modeling elements created by stereotypes are more definitive and reusable in the system domain than other temporary notations. OMG created a CORBA Profile of UML [OMG, 2001]. CORBA Profile mainly focuses on CORBA type definition and modeling using the UML extension mechanisms. Since CORBA is a text-based standard, architectural or distributed-related characteristics are not represented by graphs. Simply extending UML for modeling CORBA types cannot make full use of the powerful graphical features of UML to model some architectural aspects. Therefore we try to take a tentative step to come up with some more intuitive architectural modeling extensions to create a more readable UML representation and enable easier system design transition to the implementation phase. For example, one-way invocation is represented by a class-based stereotype in the CORBA profile. But we represent it using a UML dependency; by putting the method name on the relation we can show the one-way invocation relationship more clearly.

This section describes UML extension mechanisms that can be used to tailor the use of UML for framework-specific CBD modeling. We select CORBA as our main example because CORBA is

quite mature and it represents the component standards that are implemented by a large number of ORB (Object Request Broker) products. Different ORBs may use different implementation approaches and even additional APIs, but they must conform to the same specification. This restriction is much like "software design", and thus, CBD modeling based on CORBA largely depends on the standard. In addition, we try to incorporate both "objects" and "components" in the same diagram because of the unique characteristics of CBD that are different from traditional object-oriented modeling.

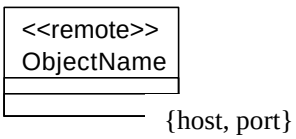
This paper is not meant to be a complete definition of CORBA modeling concepts. We selected a list of commonly used CORBA concepts as new building elements. When more CORBA elements are needed, users can create their own extensions through the methodology introduced here.

A UML EXTENSION FOR CORBA

CORBA is basically a standard and specification for component-based application domains. CORBA can be viewed as a software middleware. Components (including IDL objects, high-level services, and facilities) distributed in networks are able to interoperate with one another. Again, the UML extension mechanisms enable us to create new modeling elements for domain-specific systems. Client-side modeling requires much more design and modeling than the server (or component provider) side. The main reason for this asymmetry is that client side application building needs more complex business logic and object interactions than the server side, which is basically a repository storing sets of consistently encapsulated components. Therefore, we selected the following CORBA concepts that are closely related to client-side design as our new modeling elements because they represent commonly used elements in most CORBA-based systems. They are also listed as highly important concepts in OMG's CORBA specification – under the section "CORBA Overview" [OMG, 2000]. In addition, in choosing specific extension elements, we follow the definition of extension mechanisms as described in the previous section. For example, the reason to define {host, port} as a tagged value is because we believe the locations of different components need to be realized during the design phase of a distributed CBD application to oversee the system architecture and to choose appropriate dynamic/static invocation approaches.

Local and Remote Objects

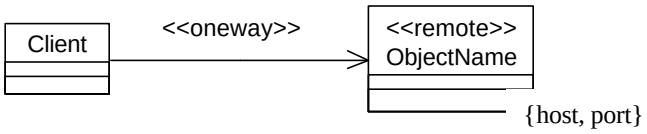
Extension used: Stereotype, TaggedValue
Metamodel class: Class
Notation: <<remote>>, {host, port}
Description: Local objects are regular objects that are located in the user (client) application domain. In contrast, remote objects have an IDL description and are reusable. They can be located on any machine inside the network. Those on the client-side can find them through object references. That is, the client side must always keep track of the object by a reference containing host and port. The reason for using a tagged value is to keep track of where the remote object is at any time in order for the distributed application to work correctly. Although more or less related with implementation, we prefer to place this information at the design phase because it reflects the deployment of a distributed system and will possibly affect implementation activities.



One-Way Invocation

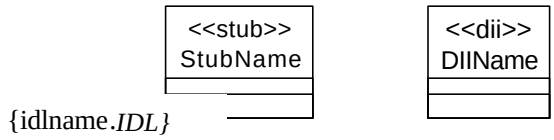
Extension used: Stereotype, TaggedValue
Metamodel class: Dependency
Notation: <<oneway>>, {host, port}

Description: When the one-way invocation is used, the operation gives the client an immediate return to the thread of control. Otherwise, the client thread will be blocked until the request is processed and the result is returned.



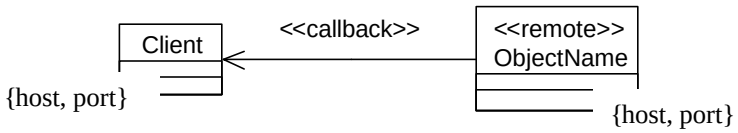
Static and Dynamic Invocation

Extension used: Stereotype, Constraints
Metamodel class: Class
Notation: <<stub>>, <<dii>>, {idlname.idl}
Description: Static invocation means that the IDL description is available at compile time. Therefore, the client-side has a static stub for calling a server-side component. Dynamic invocation allows dynamic construction of an object invocation; that is, rather than calling a stub routine that is specific to a particular operation on a particular object, a client may specify the object to be invoked, the operation to be performed, and the set of parameters for the operation through a sequence of calls. The reason for using a constraint is that each stub must be compiled from an IDL file containing component interfaces.



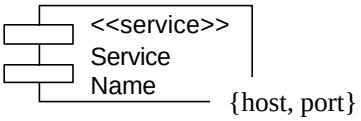
Callback Invocation

Extension used: Stereotype, TaggedValue
Metamodel class: Dependency
Notation: <<callback>>, {host, port}
Description: Callback discards a pure "client/server" pattern. After the client sends a request to the server, it is possible that the server needs to "callback" the client to obtain service from client objects. This process requires the client-side, in a similar fashion to the server-side, to provide the IDL component with references. Thus, the client component must also have a TaggedValue recording the object reference.



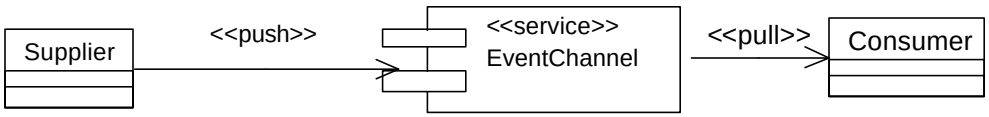
Service Components

Extension used: Stereotype, TaggedValue
Metamodel class: Component
Notation: <<service>>, {host, port}
Description: CORBA services are extensions of the CORBA core. They are called Object Services officially. They are a set of IDL components that can be used in any application. Common service components include: naming services, trading services, transactions, event services, and security services. Compared to normal components, service components are more independent and encapsulated. Therefore we use the component as a metamodel for this type of object. Service objects are also remote objects, and their references therefore need to be traced by a TaggedValue.



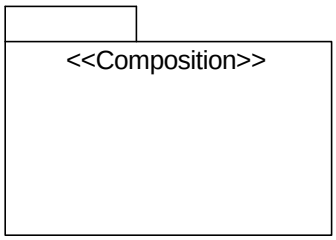
Event Service

Extension used: Stereotype
Metamodel class: Component, Class
Notation: <<service>>, <<push>>, <<pull>>
Description: A CORBA event is a communication between two or more entities regarding the occurrence of a state transition. Although it belongs to the CORBA service category, a CORBA event should be modeled separately because the components that use it behave differently than those components that use other CORBA services. In the push mode, the supplier pushes the message into the event channel with which consumers are already subscribed for messages, regardless of the consumers' status. In the pull model, a consumer pulls the event message from the event channel, thus pulling from the supplier.



Composition

Extension used: Stereotype
Metamodel class: Package
Notation: <<Composition>>
Description: Component composition means using reusable components that are distributed on the network together with local objects to build a complete application. To represent this concept, we use a package as a metamodel.



Example

To illustrate our extension mechanisms, the following example (illustrated in Figure 4) shows the design of a simple word processor created with several components, either local or remote. Though simplified, it provides a general idea of how to model a component-based CORBA application using UML and the suggested extension mechanisms.

- The word processor is composed of both local components (such as a WordCounter, an Editor, and an Email function) and remote components (such as a SpellChecker, a Printer, and a WebConnector). The latter could reside on remote network hosts.
- The <<composition>> stereotype encloses what are needed to form a system from the point of view of a "client".

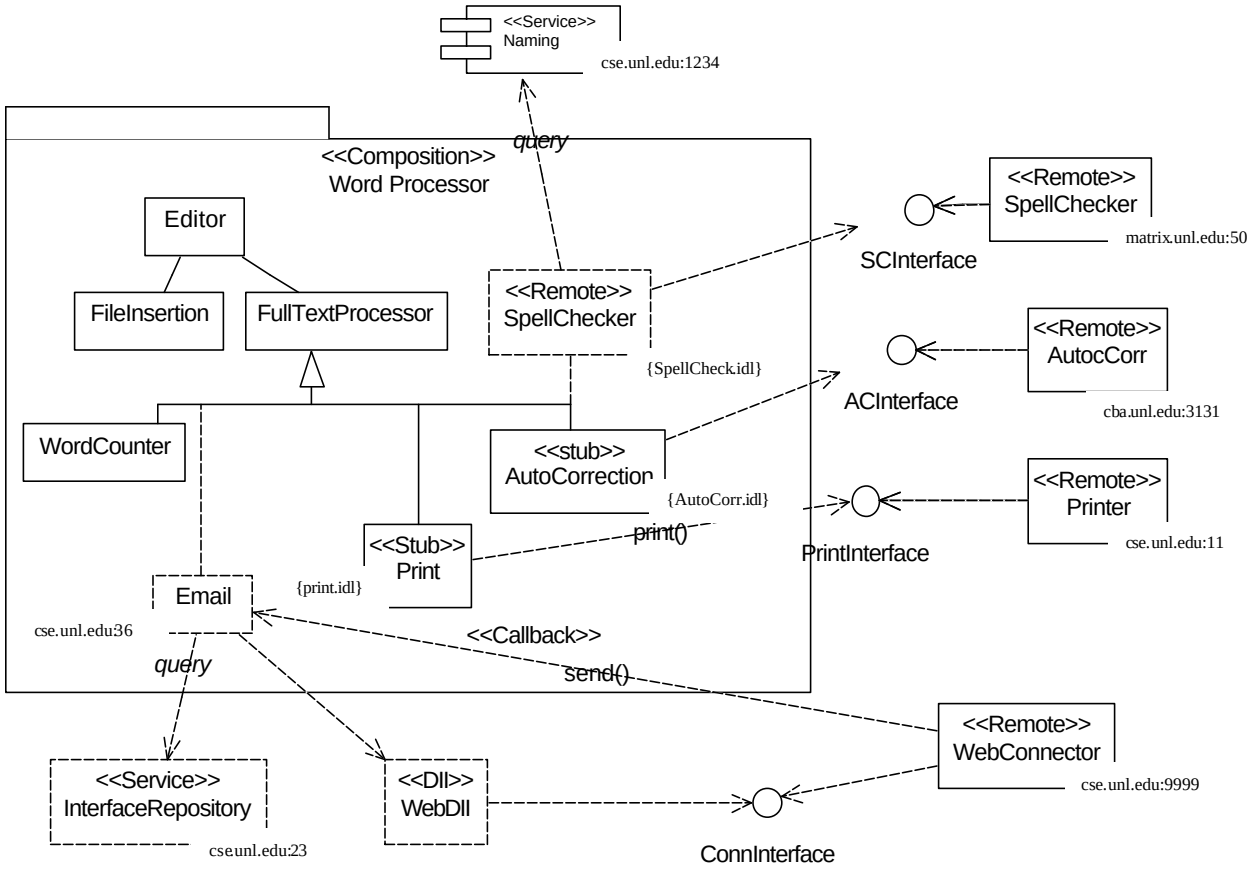


Figure 4. Simple Word Processor Modeling Using UML Extensions

- For a remote component, a <<stub>> is generated by a corresponding IDL file on the client side, which interacts with the <<interface>> provided by the remote component with a reference (such as {cse.unl.edu:4651}).
- For instance, when the word processor needs to do automatic correction, it invokes a remote component “AutoCorrector” by means of the “AutomaticCorrection” stub.
- <<Service>> components need to indicate a reference too, even though they do not have an interface object indicated because that does not change.
- For instance, when we are unsure where the component “SpellChecker” is, we can simply call the service component Naming and retrieve the object reference.
- If the text being edited is to be sent by email, the “Email” component may compose a dynamic call to invoke the “WebConnector” component by means of a DII (Dynamic Invocation Interface), because we may not know the specific interface for a specific internet connection component at compile time. Since the connection establishment may take a long time, an asynchronous mode allows the “WebConnector” to call back when the connection is no longer needed. Then, the Email object can do whatever it needs to do.

V. DISCUSSION

We used CORBA as an example to illustrate the UML modeling extension mechanisms. CORBA represents a common specification for all implementations (Object Request Broker) that conform to it. Apparently that is not enough for real-world component-based systems, especially for modeling a large-scale system. Even though we described some diagrams that are useful for modeling component-based systems, other UML diagrams (such as collaboration diagrams, deployment diagrams) can also be customized to improve CBD modeling. Those who use CORBA or other popular component frameworks may adapt the extension methodology to create more modeling elements for their specific use.

Microsoft DCOM is a major component framework other than CORBA. DCOM is both a specification and an implementation. Although this model differs from CORBA, especially in implementation, the design method used for CBD in DCOM can largely be the same as that used in CORBA.

We show an example to provide a simple comparison: Suppose our client likes to use the interface GetTimeElaps(integer t2, integer t1) exposed by another timing component named *Timer* without knowing the parameter values at compile time. In CORBA, using the Dynamic Interface Invocation (DII) could solve this problem, but in DCOM, it is a typical Automation solution. Using UML, we have similar class diagrams (Figures 5 and 6).

Figures 5 and 6 show the same design ideas despite the architectural differences. The component Timer uses an interface named TimerInterface with a GetTimeElaps() operation. The ClientObject needs the Dynamic Invocation Interface to compose the invocation dynamically. Therefore, it is quite straightforward to apply our extension approach to the DCOM environment.

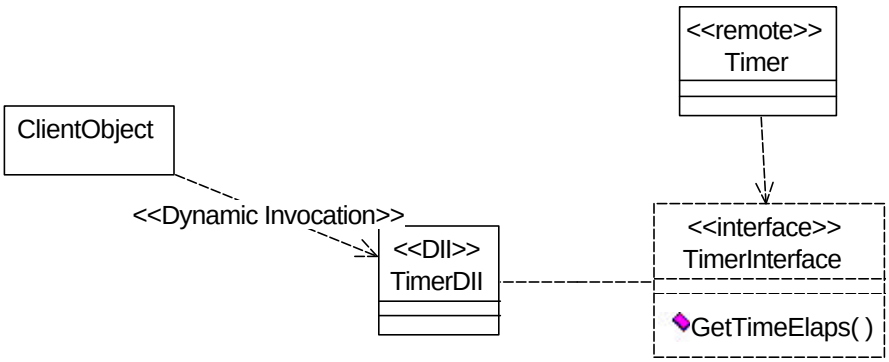


Figure 5. CORBA Version of Example

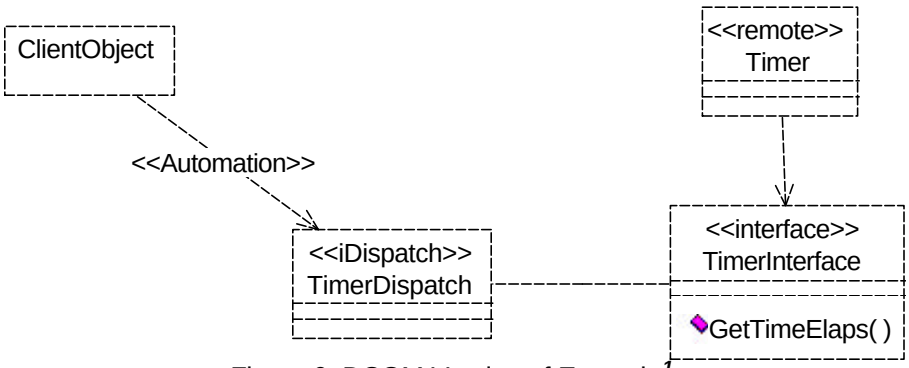


Figure 6. DCOM Version of Example[1]:

[1] Extension Used: Stereotype; Metamodel class: Class; Notation: <<remote>>, <<interface>>, <<iDispatch>>, <<Automation>>

Of course, DCOM is not the same in every aspect as CORBA. Figure 7 is another example of how to model DCOM Containment using UML[2].

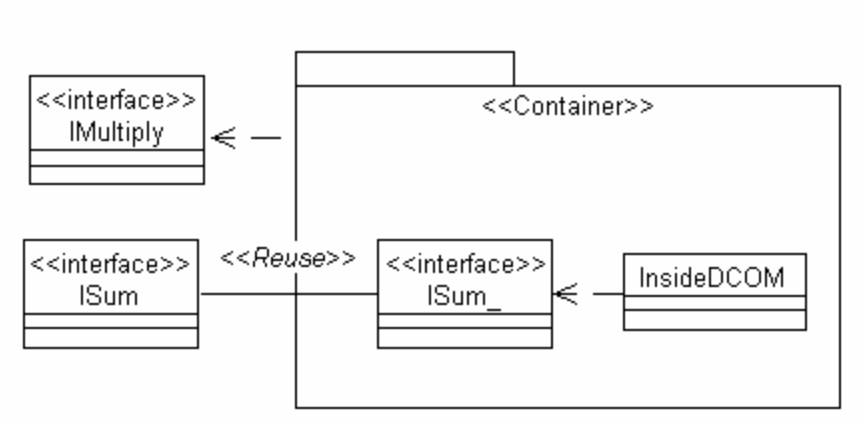


Figure 7. Modeling Containment in DCOM

Figure 7 shows that the *container* has two interfaces: IMultiply and Isum [Eddon and Eddon, 1998]. Instead of providing implementation for the ISum interface itself, the container delegates the invocation of ISum to the ISum interface of another component, named InsideDCOM.

Web Services

Before closing our discussion, we would like to mention the newly emerged technology called Web services. Web services embody a new distributed computing model enabled by UDDI [UDDI Community 2001], SOAP (Simple Object Access Protocol), and WSDL (Web Services Definition Language). Although similar to other existing distributed computing models such as CORBA [Vinoski, 2002] and DCOM, Web services model incorporates state-of-the-art XML technology, and is light-weight and more powerful because of its interoperability and firewall-friendliness.

Moreover, the Web services model brings the simplicity of modeling because of its simple architecture. The UML class diagram shown in Figure 8 could become the template for most of the Web-services based applications.

Many tools, such as Rational Rose, Visio, and Visual UML, support UML. But few of them provide adequate support for UML extensions ranging from element icon generation, diagram customization to diagram validation. Simply supplying modeling elements for users to choose among cannot meet the requirements of different users. This diversity is likely an opportunity for UML tool producers.

VI. CONCLUSIONS

This paper discusses the relationship between UML and component-based design from the following perspectives:

- Component-based development with its supporting framework offers great potential to begin a revolution in software engineering.
- Though similar to traditional design, CBD design needs some changes and flexibility because of the interface-centric and framework-based characteristics.

[2] Extension Used: Stereotype; Metamodel class: Class; Notation: <<Container>>, <<interface>>, <<Reuse>>,

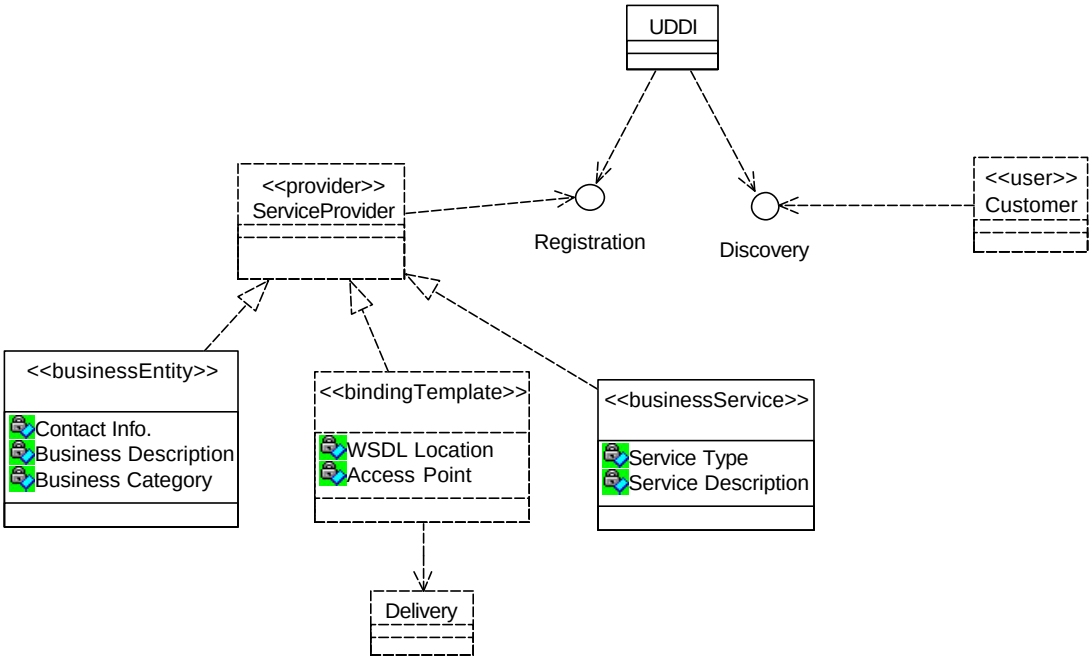


Figure 8. UML Template for Web-Services Based Applications

- For component-based systems design and modeling, diagrams and extension mechanisms provided by UML are among the best choices because UML enables better interfaces modeling and can be enhanced by means of extensions.
- For better modeling on the framework side of CBD, UML extension mechanisms should be adopted to include framework-specific new elements. The reason for using extension mechanisms is not to increase complexity, but to model the system more precisely.
- For a specific framework, what should be introduced into extension are those common elements that might be used in all implementations.

This paper discusses an approach that will enable CBD designers to find new solutions when they are faced with certain component frameworks and are unsure about how aspects related to it should be modeled, and how to build models that are closer to implementation. Although CORBA and DCOM are used as examples for illustrating the UML extension mechanisms, other component-based development approaches and elements can be modeled via this mechanism.

Editor's Note: This article was received on January 2, 2002. It was with the authors for approximately two and a half months for two revisions. It was published on October 11, 2002.

REFERENCES

EDITOR'S NOTE: The following reference list contains the address of World Wide Web pages. Readers who have the ability to access the Web directly from their computer or are reading the paper on the Web, can gain direct access to these references. Readers are warned, however, that

1. these links existed as of the date of publication but are not guaranteed to be working thereafter.
2. the contents of Web pages may change over time. Where version information is provided in the References, different versions may not contain the information or the conclusions referenced.
3. the authors of the Web pages, not CAIS, are responsible for the accuracy of their content.
4. the authors of this article, not CAIS, is responsible for the accuracy of the URL and version information.

Alhir, S.S. (1999) "Extending the Unified Modeling Language (UML)", http://home.earthlink.net/~salhir/extendingtheuml.html (current Sep. 25, 2002).

Barn, B. and A.W. Brown (1998). "Technology of Object-Oriented Languages", *Proceedings of 26th Technology of Object-Oriented Languages and Systems*, pp. 385-395

Baumeister, H., N. Koch, and L. Mandel (1999). "Towards a UML Extension for Hypermedia Design", http://www.fast.de/Projekte/forsoft/uml99/ (current Sep. 25, 2002).

Ben-Shaul, I., J.W. Gish, and M. Robinson (1999). "An Integrated Network Component Architecture", *IEEE Software*, (15)5, pp. 79-87.

Booch, G., J. Rumbaugh, and I. Jacobson (1999). *The Unified Modeling Language User Guide*, Upper Saddle River, NJ: Addison-Wesley

Brown, A.W. (2000) *Large-Scale Component-Based Development*, Upper Saddle River, NJ: Prentice Hall

Brown, A.W. and K.C. Wallnau (1998) "The Current State Of CBASE", *IEEE Software*, 15(5), pp. 37-46

Eddon, G. and H. Eddon (1998) *Inside Distributed COM*, Washington, Microsoft Press
Herzum, P. and O. Sims (1999) *Business Component Factory*, New York, John Wiley & Sons, Inc.

Hopkins, J. (2000) "Component Primer", *Communications Of The ACM*, (43)10, pp.27-30
Kobryn, C. (2000) "Modeling Components And Frameworks With UML", *Communications Of The ACM*, 43(10), pp. 31-38

Kozaczynski, W. and G. Booch (1998) "Component-Based Software Engineering", *IEEE Software*, 15(5), pp. 34-36

Mowbray, T.J. and W.A. Ruh (1997) *Inside CORBA*, Upper Saddle River, NJ: Addison-Wesley

Norris, M., R. Davis, and A. Pengelly, (2000) *Component-Based Network System Engineering*, Norwood, MA: Artech House Publishers

OMG (2000) "Common Object Request Broker Architecture (CORBA) v3.0", http://www.omg.org/technology/documents/formal/corba_iiop.htm, (current Sep. 25, 2002).

OMG (2001) "UML Profile for CORBA Specification", http://www.omg.org/technology/documents/formal/profile_corba.htm, (current Sep. 25, 2002).

Pope, A. (1997) *The CORBA Reference Guide*, Upper Saddle River, NJ: Addison-Wesley

Siau, K., and Q. Cao (2001) "Unified Modeling Language – A Complexity Analysis," *Journal of Database Management*, 12(1), pp. 26-34

Siau, K., and T. Halpin (2001) *Unified Modeling Language: Systems Analysis, Design, and Development Issues*, Hershey, PA, Idea Group Publishing

UDDI Community (2001), http://www.uddi.org (current Sep. 25, 2002).

Vinoski, S. (2002) "Where is middleware", *IEEE Internet Computing*, 6(2), pp. 83-85

LIST OF ABBREVIATIONS

CBD – Component Based Development

CBD is a software development approach in which software are developed by reusing interoperable software components. CBD enables the development of software systems with higher efficiency, higher quality, lower cost, and shorter development time. Usually CBD is supported by component-enabling frameworks such as CORBA and DCOM.

CORBA – Common Object Request Broker Architecture

CORBA is a distributed object computing infrastructure standard managed by OMG (Object Management Group). In general, by using CORBA based ORB (Object Request Broker), application developers are able to produce distributed, interoperable components for reuse by other applications.

DCOM – Distributed Component Object Model

DCOM is both a standard and a software implementation framework from Microsoft. Similar to CORBA based ORBs, DCOM is a binary and network framework that allows any components to communicate regardless of the locations, operating systems, and languages the components are written in.

DII – Dynamic Invocation Interface

DII enables CORBA clients to invoke server objects dynamically. In contrast to static invocation, which requires obtaining IDL description and generating Stub during client-side application development. DII composes invocation requests at runtime and is more flexible at communicating with dynamic objects.

DOC – Distributed Object Computing

DOC integrates objects distributed on the network and build software application based on these objects. CORBA, DCOM, and Web Services are all infrastructures to support DOC.

DSI – Dynamic Skeleton Interface

DSI is the server-side counterpart of DII. It provides a way for CORBA servers to receive invocations dynamically without building static Skeleton during development time.

IDL - Interface Definition Language

IDL is an implementation independent interface description language for component producers to publish component behaviors or functions through standard and universally acceptable way. This enables components users to invoke component functions without knowing the implementation specific information.

UML - Unified Modeling Language

UML is the standard modeling language for object-oriented software development. It is used for specifying, visualizing, constructing, and documenting the artifacts of software systems.

XML - eXtensible Markup Language

XML is a text based universal format for exchanging self-descriptive information among different parties over the Internet. It is a standard managed by W3C (World Wide Web Consortium). Many other standards, including Web Services, are based on XML.

ABOUT THE AUTHORS

Luyin Zhao received a Master of Software Engineering and Business Management from the J.D. Edwards Honors Program at the University of Nebraska-Lincoln in 2001, a Master of Computer Science from Beijing University, and a Bachelor of Computer Science from Beijing University of Aeronautics and Astronautics. He is currently a member of the research staff at the Healthcare Systems and IT department of Philips Research USA and a part-time Ph.D. student at the State University of New Jersey–Rutgers. His research interests include software engineering, component and object-oriented technology, workflow systems in medical IT, interoperability, and application of latest web technologies in the business domain.

Keng Siau is Associate Professor of Management Information Systems at the University of Nebraska-Lincoln. He is Editor-in-Chief of the *Journal of Database Management.* He received his Ph.D. degree from the University of British Columbia where he majored in Management Information Systems and minored in Cognitive Psychology. His master and bachelor degrees are in Computer and Information Sciences from the National University of Singapore. He is the author of over 40 refereed journal articles which appear in such journals as *Management Information Systems Quarterly, CACM, IEEE Computer, Information Systems, Data Base, IEEE Transactions on Biomedical Engineering, Journal of Database Management, Journal of Information Technology, International Journal of Human-Computer Studies*.

Communications of the Association for Information Systems

ISSN: 1529-3181