Mapping Component Specifications to Enterprise JavaBeans Implementations

Yi Liu
Computer & Information Science
201 Weir Hall
University of Mississippi
University, MS 38677 USA
liuyi@cs.olemiss.edu

H. Conrad Cunningham
Computer & Information Science
201 Weir Hall
University of Mississippi
University, MS 38677 USA
cunningham@cs.olemiss.edu

ABSTRACT
Component-based software development has become an important approach to building complex software systems. Much research focuses on component specification to achieve the advantages of the component-based approach in theory. Most of this research pays little attention to the mappings from component specifications to component implementations. However, the mappings are important because they determine whether the implementations perform satisfactorily to meet the specifications. After presenting a general approach to component specification and the technology of Enterprise JavaBeans (EJB) component model, this paper presents three approaches to mapping from component specification to EJB implementation. This paper uses a course registration system as an example to demonstrate the ideas. The approaches presented will be helpful to those who are working on the realizations of component systems.

Categories and Subject Descriptors
D.2.12 [**Software Engineering**]: Interoperability – *distributed objects.*

General Terms
Design.

Keywords
Component software, Enterprise JavaBeans, design mapping

1. INTRODUCTION
Component-based software development has become an important approach to building complex software systems. A potential advantage it delivers is reuse. A software application can be built quickly and reliably by assembling preexisting frameworks and components with a few new components. Another advantage of the component approach is management of change [2]. A component can be easily replaced by a new component with minimal impact on the clients of that component. In such a way, a software application can be updated easily by replacing the existing components with new ones.

Much research focuses on component specification to achieve the advantages of the component-based approach in theory. Most of this research pays little attention to component implementation. Implementation is the realization of the specification, so, in theory, an implementation just needs to provide the functionality given in the specification to construct a reliable component-based system. However, in the real world, the component implementation is not that easy to achieve. A component is not a programming language object. Although component objects have most of the characteristics of objects in Java or C++ programs, they exist, and can only exist, in the context of a component standard [2]. The implementation varies among different programming languages and different component models. We call the approaches applied to realize an implementation from a component specification a *mapping*. Although some research focuses on component models, such as Enterprise JavaBeans (EJB), most of the work is on the technologies of the component model while little work has been done on the connections between the component specification and the component implementation.

This paper presents approaches to mapping from component specifications to component implementations. The component specification method used in this paper is that of Cheesman and Daniels [2]. The component model used to implement the component specification is Enterprise JavaBeans (EJB). EJB is a software component model for developing and deploying enterprise-level, server-side computing applications that are scalable, transactional, and multi-user secure [1]. It is a hot technology in component-based system implementation.

The paper is organized as follows. Section 2 introduces the system architecture and component specification approach of Cheesman and Daniels. The paper uses the Unified Modeling Language (UML) [3] to express the structural relationships. Section 3 briefly describes the EJB technology. Section 4 presents three approaches to mapping from a component specification to an EJB implementation. An example *course registration system* is used to demonstrate the ideas. A short discussion concludes the paper in Section 5.

ACMSE '04, April 2–3, 2004, Huntsville, Alabama, USA.

2. COMPONENT SPECIFICATION

Cheesman and Daniels [2] propose an approach to building enterprise-scale component systems. The work described here adopts that approach.

2.1 Concept of Component

There is no universally accepted definition of software component. There are many definitions, each of which focuses on different aspects of the component concept.

First, we need to define what a *component* is for the purposes here. A component system is composed of independent, large-grained units of development, deployment, and execution. The internal design and implementation are strongly encapsulated and hidden from the outside world. Each component has one or more *interfaces* that are specified with signatures and design contracts (pre/post-conditions, invariants) [5,6]. Components communicate with each other exclusively through their interfaces. In this paper, we assume that components are organized in a flat structure. That is, no component contains other components in the specification model.

2.2 System Architecture

The system architecture is the overall structure of the final system. It identifies the components and defines their responsibilities and interconnections. Cheesman and Daniels [2] define four system architecture layers, which are assumed to be built using a client/server paradigm. The layers are identified as follows.

The *Business services* layer typically resides on a server. This is the bottom layer of the architecture and is the repository of core business information that is shared by all clients. These services usually have associated databases. Components correspond to stable business types. Operations can be combined with others in a transaction. Components are decoupled from each other and thus can be shared among several systems and users.

The *System services* layer also typically resides on a server. This layer is above the business services layer. It is the external representation of the system, providing the clients access to the services of the system. Components correspond to whole business systems and operations are new transactions. This layer holds no dialog or client-related state.

The *user dialog* layer is above the system services layer. It includes the software that manages the interactions of a client with the system. Logically, this layer is in the client part of the client-server system.

The *user interface* layer is above the user dialog layer. It includes the software that creates what a client (user) actually sees and is, hence, logically in the client part of the client-server system.

The bottom two layers form the system that is user interface independent. When a user interface is connected to the system, an application is built.

2.3 Component Specification Design

Building component-based software begins with the analysis of the system requirements. In this *requirements definition* phase, two models are generated. One is a *use case model*, which captures the user requirements for the system to be developed by detailing user interactions. Another is a *domain model*, which represents classes for real-world entities and related concepts [7].

The *component specification* phase follows the requirements definition phase. Its purposes are to identify the components and then to define the interfaces and functionalities of the component objects. There are three stages in the specification phase—component identification, component interaction, and component specification.

The *component identification* stage takes the domain model and the use case model as inputs. For each use case, we define an initial system interface. The operations on the interface are the major steps in the use case. We next transform the conceptual domain model into a business type model that gives design information. A key step in this process is the identification of the *core business types*; these are the concepts that emerge as being independent business-related entities as we analyze the type model. For each core type, we define a business interface to manage it and its related subordinate concepts. The whole system can be divided into several components based on the core types. Usually, we can set a core type as a component and add the types which are "managed" by the core type into that component. For example, as shown in Figure 1, we can divide a *course registration system* into three components based around the *Person, Term* and *Course* core types. (We will discuss this example in section 2.4). Each interface of a component is a manager for a core type in the business services layer. For example, interface *IPersonMgt* is the interface for the core type *Person* or component *PersonMgr*. The initial component architecture thus consists of the system interfaces and business interfaces and their relationships.

The second stage of the component identification phase is *component interaction*, whose purpose is to determine how the components work together to deliver the required functionality [2]. The approach is to decide how to implement the operations on the system interfaces by sequences of interactions with the various business interfaces. This helps the designer discover what operations are needed on the business interfaces defined in the business type model.

The final stage of the component identification phase is *component specification*. In this stage, the detailed specification of the operations and constraints takes place [2]. It specifies the interactions between the component object performing an operation and other component objects that are required to complete the operation. It is also necessary to specify the constraints that need to apply to the operations. An interface information model is introduced to enable the definition of these interactions and constraints. The general approach taken is design by contract [2,5,6]. In this approach, preconditions and postconditions are defined to give the meaning of an interface's operations in terms of its information model. Invariants can also be defined to set the constraints on the integrity of the interface information model. The component specification stage completes the specification of the system.

2.4 Course Registration System

The example used here is a course registration system for a college. With this system, a student may register for classes. Once given access, the student may select a term and then build a class schedule from among the classes offered. A student may add and delete classes from the schedule. The system passes the information about the student's schedule to the tuition billing system. An instructor may use the registration system to print a

listing of the students in his or her class. The administrator may maintain student and instructor lists and course information.

The diagrams shown here are interface responsibility diagram of the business type model (Figure 1) produced from component identification stage and the final component architecture diagram (Figure 2). Figure 1 identifies *Person, Course* and T*erm* as core types. Based on the three core types, we divide the system into three components and each component has an interface. We define the boundary between two components and assign the remaining types to the component. Thus, the *PersonMgr* component includes the *Person, Student, Instructor, StudentSchedule, InstructorSchedule*, and *Administrator* types. The *CourseMgr* component includes types *Course* and *Section*. The *TermMgr* component only includes the type *Term*. Three interfaces, *IPersonMgt, ICourseMgt* and *ITermMgt*, are assigned to *Person*, *Course* and *Term*, respectively. Figure 2 shows the main use cases and components required for the system. Interface *IBilling* is out of the system bounds, so we assume it can be supplied by a separate Billing system. These two diagrams are the best representatives of the system to provide an overall view of the system and help to demonstrate the mapping of the component specification to its implementation.

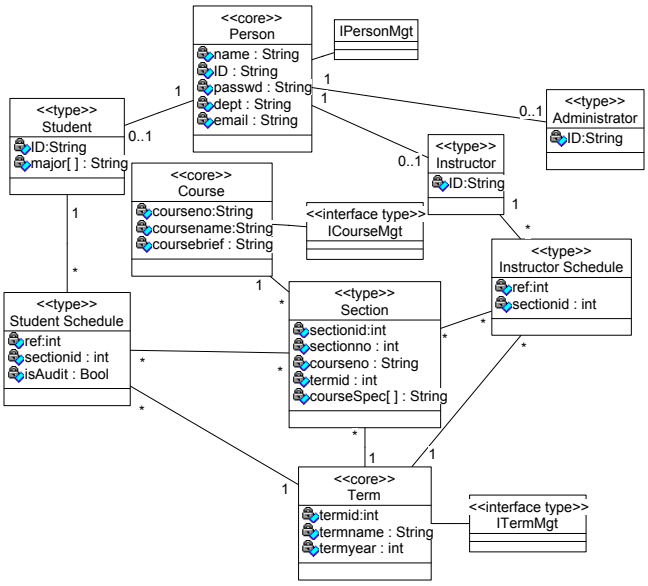

Figure 1. Interface Responsibility Diagram for Course Registration System

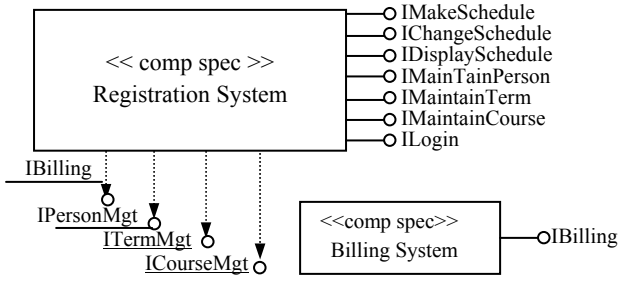

Figure 2. Component Architecture for Course Registration System

3. EJB TECHNOLOGY

Enterprise JavaBeans (EJB) from Sun Microsystems is a component model for building server-side, enterprise-class applications [8]. Figure 3 shows the EJB model and the relationships among the parts within that model.

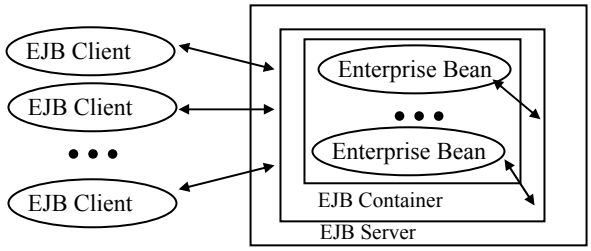

Figure 3. **EJB Model**

The two most important parts in the EJB component model are *enterprise beans* and the *EJB container*. The EJB container exists on an EJB server and provides enterprise beans a runtime environment including remote access to the bean, security, persistence, transactions, concurrency, and access to and pooling of resources. Enterprise beans are server-side components, which encapsulate the business logic of an application and are deployed and execute in an EJB container. The beans are reusable and shareable components on a server that can be remotely accessed by a client program. EJB is thus suitable for building distributed, reusable systems [1].

There are three types of enterprise beans: session beans, entity beans, and message-driven beans.

Session beans are in-memory objects that are non-persistent. They are designed to perform the processes of a business. A session bean typically executes on behalf of a single client and cannot be accessed by other clients. Session beans can be either stateful or stateless. A *stateful* session bean holds conversational state on behalf of its client and stores information for a relatively short amount of time. A *stateless* session bean does not maintain conversational state and it immediately processes the information received from the client. Though session beans are not persistent, they can update data in a database and participate in transactions [1, 8]. A session bean consists of a remote interface, a home interface, and a bean implementation class. The remote interface declares publicly available methods of the session bean. The home interface declares the create methods for creating new EJB instances. The bean implementation class implements the methods declared in the remote interface and home interfaces.

Entity beans are persistent. Each entity bean allows shared access from multiple EJB clients. Their states can be persisted and stored across multiple invocations. An entity bean can be used to represent data stored in a database. Persistence in entity beans has two types, container-managed and bean-managed. If the container handles the synchronization of the bean's data with the external stores, this is called *container-managed persistence*. If the entity bean itself is responsible for maintaining its own persistence, this is called *bean-managed persistence*. An entity bean also consists of a remote interface, a home interface, and a bean implementation class. The remote interface declares publicly methods for the EJB and the home interface declares methods for creating new instances and locating instances. The bean

implementation class implements the methods declared in the remote and home interfaces.

Message-driven beans are stateless, server-side, transaction-aware beans that are driven by a Java message. A message-driven bean is invoked by the EJB container when a message is received (asynchronously) from a Java Message System (JMS) Queue or Topic. The bean acts as a simple message listener.

The course registration system example is implemented with Sun's Java 2 Enterprise Edition (J2EE) platform. J2EE is designed to provide a multilayer distributed application model [4]. The architecture of J2EE is shown in Figure 4. The course registration system uses HTML in a browser, JavaServer Pages (JSP) and JavaBeans in a Web container, and Enterprise JavaBeans in an EJB container. The database used is Cloudscape and the database connector is the Java DataBase Connectivity (JDBC) library.

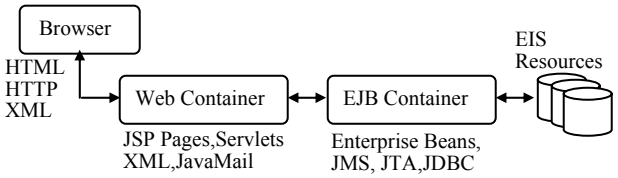

Figure 4. J2EE Architecture

4. MAPPING APPROACHES

This paper focuses on the EJB container components. It is not concerned with the browser and the Web container software. For convenience, we call the application in the user interface and dialog layers of the system architecture *dialog software*, call the components in the system services layer *system components*, and call the components in the business services layer *business components* [2]. As addressed in the section 3, the example implementation used HTML and JSP as tools for the dialog software.

This section presents how the business components and the system components are implemented according to the component specification and the system architecture. The following shows three mapping approaches: the manager bean, hierarchical, and singleton EJB approaches. It uses the course registration system to illustrate the mapping approaches.

4.1 Manager Bean Approach

In the manager bean approach [2], all components are session beans. Figure 5 uses the *PersonMgr* component to show the mapping from system architecture to EJB.

In the business services layer, each business component is implemented as a session bean. To make the session bean easy to implement, we usually use helper classes (regular Java classes) to implement business types and let each component manage a set of instances of the types.

In the *PersonMgr* component, we have the following six business types: *Person, Student, Instructor, Administrator, StudentSchedule,* and *InstructorSchedule.* We use three helper classes to implement them: one for the *Person* type (functions for Student, Instructor, and Administrator are included), one for *StudentSchedule*, and one for *InstructorSchedule*. These three Java classes provide methods for dealing with the access to the database.

There is a manager bean *PersonMgr* that coordinates operations on these types. The possible processes a client can do include accessing his student account and making or changing his schedule. All data shared among clients are stored in the database and no conversational session state is needed. So, it is reasonable to make the *PersonMgr* a stateless session bean in which each client has a session bean object. Thus, the *PersonMgr* component is built upon a session bean *PersonMgr*, several helper classes, and an interface.

In the *CourseMgr* component, we use the same approach. First, we build two helper classes, *Course* and *Section*, which provide the operations on the database. Then, we use a stateless session bean *CourseMgr* to wrap the two classes and build the course component. And, an interface is provided for use by the system component.

Similarly, the *TermMgr* component consists of a stateless session bean *TermMgr*, a helper class for the *Term* type (which operates on database), and an interface provided for use by the system component.

Each business component provides an interface to the upper level. For example, the *PersonMgr* component provides the interface *IPersonMgt*, which is implemented by *PersonMgr*; the *TermMgr* component provides the interface *ITermMgt*, which is implemented by *TermMgr*; and the *CourseMgr* component provides the interface *ICourseMgt*, which is implemented by *CourseMgr*. The upper level can access the business component objects through these interfaces.

In the system service layer, a system component wraps the business components and provides an interface for the dialog software to access. For accessing the system, it is better for the client to have its own instance, which means that one instance for one client. So, we choose a session bean to implement the system component. A session bean manages the collective of the three business components − *PersonMgr, CourseMgr, and TermMgr* − and carries out business workflows by calling the three business components.

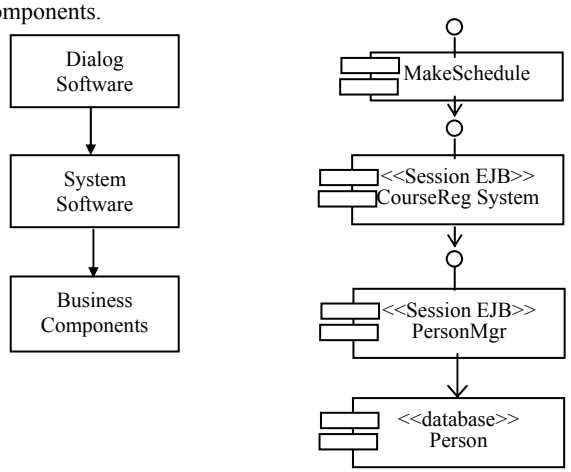

Figure 5. Manager Bean Approach

Stateful session beans, which involve many interactions with clients, can be used to implement the user dialog software level, in which conversational state would be stored. However, the example course registration system implemented the dialog software in the Web container using JSP.

The advantage of this approach is its simplicity as the session bean is the simplest among the three types of enterprise beans. One disadvantage of this approach is that session beans do not deal with persistence, so we have to add persistence code for each manager. Another disadvantage is it does not have in-memory data sharing, so performance may be affected.

4.2 Hierarchical Approach

The hierarchical approach is more complicated than the manager bean approach. The idea is to decompose a business component into a manager and several subcomponents. Figure 6 shows this mapping approach with the example *PersonMgr* component.

In the business services layer, to avoid the performance penalty and the persistence problem arising from using session beans to implement the lowest level types, this approach uses entity beans to implement those types that access the database directly. Since entity beans are persistent and can be shared among multiple clients, it is ideal to use entity beans to encapsulate database tables [2]. Take the *PersonMgr* component as an example. There are six database tables built corresponding to these six business types. Since the types are heavily dependent on the database, the course registration system uses an entity bean to implement each type of component.

The *PersonMgr* bean manages these types and coordinates operations on the *PersonMgr* component by accessing the instances of the types that are implemented as entity beans. The hierarchical mapping approach uses the *PersonMgr* bean since all sharing of state is done in the entity beans.

Similar to the manager bean approach, this approach also uses a stateless session bean to implement the system component.

Compared to the first approach, an advantage of the hierarchical approach is that it increases the reliability of the database access and data sharing. The disadvantage is that the hierarchical layer may bring inefficient and complex implementation.

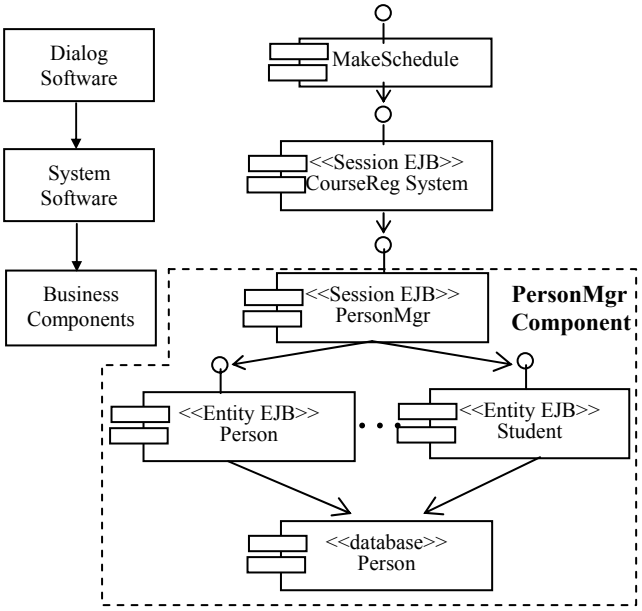

Figure 6. Hierarchical Approach

4.3 Singleton EJB Approach

The previous two approaches use pure Enterprise JavaBeans to implement the components. In the singleton EJB, we use both EJB and regular Java classes to perform the implementation (as shown in Figure 7).

Compared to regular Java program development, EJB development is more complicated. EJBs are very useful to meet distributed system requirements. For some large systems, the business components might be developed by different groups and distributed to different servers. The situation is similar for the system components. The clients are distributed and will use the system from different areas. So, the previous two approaches are suitable for these situations. Every approach has its trade-offs. Compared to regular Java programs, enterprise beans are more complicated to develop and less efficient to execute because of the communication overhead.

Consider relatively small systems, such as a course registration system that will only be used within a single college. The business components are typically stored on the same server. The database may also be on that server. Because a high level of distribution is not needed, we can use regular Java classes for some components.

We designed the singleton EJB approach for developing a relatively small system in which business components are not distributed all over the world. This approach can be described as using regular Java packages (or classes) to implement business components and using an enterprise bean to wrap these business components at the system services layer. Figure 7 shows the approach with the example course registration system.

The singleton EJB approach looks similar to the manager bean approach at the first glance. But, actually, the two approaches are different. We use regular Java packages to implement business components in this approach instead of the session beans in the manager bean approach.

Each business component is a regular Java package. An interface for each business component is provided to the system component. Inside each Java package, we can design hierarchical structures if necessary. For example, *PersonMgr* is a complicated business component. It contains six business types: *Person, Student, Instructor, Administrator, StudentSchedule*, and *InstructorSchedule*. We define a class for each business type. *PersonMgr* wraps these classes and implements the methods required in the interface *IPersonMgt*. For a simple component, such as *Term*, hierarchical structures might be unnecessary. These business components access the database and build on the database system capabilities to implement any needed transactions and concurrency control features.

In the system service layer, we use a single enterprise bean to implement the system component *Course Registration System*. Since the clients are distributed, the use of the enterprise bean satisfies the requirements. Similar to the previous two approaches, we use a stateless session bean to implement the system component. The system component gathers those business components together and provides an interface for the dialog software.

The singleton EJB approach is the simplest approach among the three approaches presented. As noted earlier, writing regular Java classes is much easier than writing enterprise beans. Programmers must write three Java classes for a session bean or an entity bean and must deploy each bean. When using regular Java classes, programmers only need to write a main class for each type and compile it. This approach simplifies the development process and makes it more accessible to novice programmers.

In this example, there is only one enterprise bean – the *Course Registration System* bean at the system services layer. However, this enterprise bean gives the system the characteristics of EJBs. It can successfully deal with the distributed clients.

The singleton EJB is a good approach for those who are familiar with Java but less familiar with EJBs. However, this approach has disadvantages. Since the approach does not use enterprise beans in the business components in which EJB container can take care of the transactions and concurrency, programmers have to deal with the transactions and concurrency themselves in the Java classes implementing the business types.

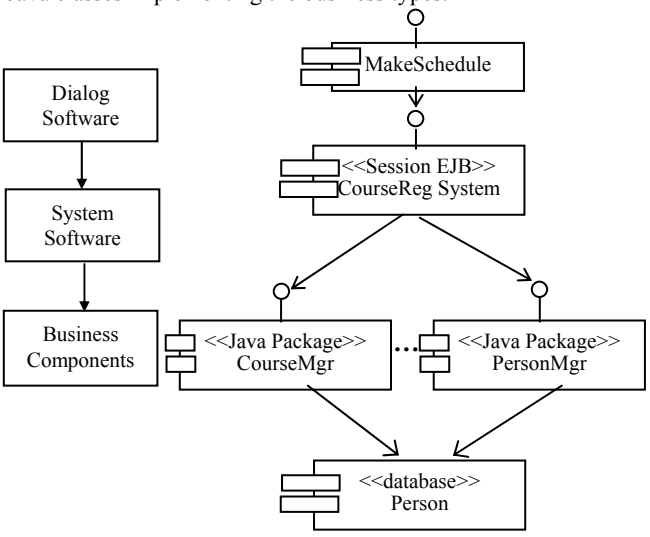

Figure 7. Singleton EJB Approach

5. CONCLUSION

The realization of the mapping from component specification to implementation is an important issue in component software.

This paper described three approaches, which are shown to be feasible in practice. Comparing these three approaches, the hierarchical approach provides the most reliable persistence maintenance. However, it is the most complicated and has the longest development cycle. The manager bean approach is simpler than the hierarchical approach, but it may get some performance penalty because the session beans do not have in-memory sharing. The last approach, the singleton EJB, is the simplest in realization; the development time is often less than of the previous two approaches. But its disadvantage is that developers must deal with transactions or concurrency problems in the code.

The mapping approaches presented in this paper should be helpful to the developers of component-based systems.

6. ACKNOWLEDGMENTS

This work was supported, in part, by a grant from Acxiom Corporation titled "The Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE)." Mingxian Fu and Pallavi Tadepalli assisted with the design of the course registration system. Fu was instrumental in the EJB implementation. Cuihua Zhang and Tadepalli each proofread the paper and suggested several improvements.

7. REFERENCES

[1] D. Blevins. "Overview of the Enterprise JavaBeans Component Model", In G.T. Heineman and W. T. Councill (editors), *Component-based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.

[2] J. Cheesman and J. Daniels. UML *Components: A Simple Process for Specifying Component-Based Software*, Addison Wesley, 2001.

[3] M. Fowler and K. Scott. *UML Distilled*, Second Edition, Addison Wesley, 1999.

[4] N. Kassem and the Enterprise Team. *Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition*, Addison Wesley, 2001.

[5] Y. Liu and H. C. Cunningham. "Software Component Specification using Design by Contract," *Proceedings of the SouthEast Software Engineering Conference*, Tennessee Valley Chapter, National Defense Industry Association, Huntsville, Alabama, April 2002.

[6] B. Meyer. *Object-Oriented Software Construction*, Prentice Hall PTR, 1997.

[7] D. Rosenberg and K. Scott. *Use Case Driven Object Modeling with UML: A Practical Approach*, Addison Wesley, 1999.

[8] I. Singh, B. Stearns, M. Johnson, and the Enterprise Team. *Designing Enterprise Applications with the J2EE™ Platform*, Second Edition. Addison Wesley, 2002.