Composite Component Support for EJB

Steffen Goebel and Michael Nestler

Dresden University of Technology
Department of Computer Science
Institute for System Architecture
D-01062 Dresden, Germany

goebel@rn.inf.tu-dresden.de
mnestler@whoglue.com

Abstract: Component frameworks and component middleware like Enterprise JavaBeans (EJB) have been established successfully in the last few years. However, composite components are not yet a part of these platforms. They increase reuse of software and can be used to encapsulate run-time adaptation. We present a composite component framework developed as an extension of EJB 2.0. We describe general design objectives for composite components and required rules for the visibility of components. The necessary container support for composite EJBs is implemented based on the EJB application server JBoss. An example application demonstrates the use of the new EJB type and shows how to dynamically reconfigure graphs of composite EJB components after their deployment in order to achieve run-time adaptivity.

1 Introduction

The vision of simply assembling applications from small software building blocks – components – is the driving force behind component-oriented software development. Such components are implemented using standard technologies such as Enterprise JavaBeans (EJB) [1], the CORBA Component Model (CCM) [3], the Component Object Model (COM) [2], and others. These technologies do already allow for the creation and reuse of commercial and non-commercial software components, but they offer only limited capabilities for the assembly of bigger components from smaller ones (façade components, etc.). This implies that applications built using the above mentioned technologies usually contain a number of atomic (non-composite) components. Additionally, a lot of manually written “non-component” or “glue” code is required to bind the atomic components to the rest of the application. The resulting code is difficult to reuse in other applications.

Complex applications often consist of a big number of subsystems. With existing component frameworks such as EJB, these subsystems are composed out of a big number of individual components. From the perspective of the component assembler there is nothing that represents the subsystem itself. Reuse of subsystems is difficult to achieve. The larger the whole application, the more difficult it becomes to extract parts of it, since the application is fine-grained and composed of a big number of atomic component elements. A more coarse-grained structure would benefit reuse and maintenance of application subsystems. Composite components seem to be a promising solution to achieve this goal. They can also be used to encapsulate adaptive logic and hide run-time adaptation from other parts of the system.

In this paper we will investigate how the popular component architecture framework and middleware platform Enterprise JavaBeans can be extended in order to assemble larger, composite EJB components out of existent, less complex standard EJB components. This
work is part of the COMQUAD\(^1\) research project dealing with QoS support, security, and adaptivity in component-based systems.

The paper is structured as follows: Section 2 presents general design objectives for composite components and defines rules for scope and visibility of components. Based on this, the implementation and an example application for composite EJB components are introduced in section 3. The implementation is an extension of the standard EJB 2.0 component framework and platform. Upward compatibility for existing EJB 2.0 components will ensure that existing component assets are still valuable within the context of the new framework. A fully working prototype implementation of an application server that realizes and supports the composite component framework was developed based on the popular open-source EJB server JBoss [7]. We conclude with a brief discussion of related and future work.

2 Design objektives

A good composite component framework should have the following qualities:

1. Wherever in the framework an atomic component can be used, a composite component should be usable too. This makes it feasible to replace any atomic part of a system with a composite replacement, and vice versa.
2. There should be no limit to compositional size. Composite components might have one or a large number of constituent subcomponents. The inner size of a composite component should not concern clients.
3. The framework should have the quality of unlimited nesting. This means that it is theoretically capable of supporting an unlimited hierarchy of nested components.
4. Parts of a composite component should use contracts of some form to communicate with each other. They should not be concerned with how another part is working. The only concern of the composite component should be the contract, which makes it possible to replace a constituent subcomponent with another part that fulfills the same contract. Contracts can be implemented using interfaces, and if necessary using additional constructs.
5. Parts of a composite component should have access to meta information about the structure of the composite component they are working in. Constituents can alter the structure of the composite component if necessary. This can be triggered by certain conditions, depending on the business logic executed by the component. Therewith, the composite component can adapt to a changing environment or new outer requirements.
6. It should be possible to pre-configure the implementation structure of a composite component, and also to alter the structure at a later time. This means that when the composite component is created, the default structure is constructed internally. Nevertheless, nothing should prevent the rearrangement of the component's internal structure during its lifetime, if required.

Based on these qualities we have developed a new EJB type – the composite EJB – in addition to the existing stateful and stateless session beans, entity, and message-driven beans. The concept of composite EJBs is meant to extend the official EJB specification, version 2.0. Its addition does not affect in any way the semantics of the already specified Enterprise JavaBeans types. By simply extending the existing specification instead of completely overthrowing it, any application or EJB component compliant to EJB 2.0 becomes also compliant to the new extended version of the EJB specification that includes composite EJB. This eases the introduction of composite Enterprise JavaBeans components into already

\(^1\) COMponents with QUantitative properties and ADaptivity at Dresden University of Technology and Friedrich-Alexander University Erlangen-Nürnberg, Germany; supported by German Research Council; see also project website at http://www.comquad.org
existent EJB 2.0 compliant applications. Of course, a modified application server (cf. section 3.2) will be necessary to support the applications containing composite EJB components.

Composite EJBs may have remote and local interfaces as specified in the EJB 2.0 specification and they must have a remote or local home interface (or both).

A composite EJB component does not have an implementation class. Instead, the implementation is provided by the sum of all subcomponents of the EJB. At least one subcomponent is required, since every composite component interface method must have a method mapping to one of the composite components constituents. Each business method of a composite EJB is unambiguously mapped to exactly one of its subcomponents. The sum of all these mappings for one composite EJB component is called the component's Outer Interface Mapping. The composite EJB container is responsible for the run-time mapping of the composite EJB component's interfaces to its subcomponents.

2.1 EJB Component Visibility

An important issue regarding composite EJB subcomponents is the scope of the EJB name, which is used to get references to other components at run-time and therewith creating component nets. The EJB specification [1] defines the name of a component (ejb-name) as being unique and globally visible within the deployment unit. The component is visible server-wide (potentially cross-application) regarding the Java Naming and Directory Interface (JNDI) namespace.

This changes drastically with the introduction of composite components. The ability to have subcomponents with potentially the same names across different composite EJBs and a potentially unlimited level of nesting, does significantly complicate the definition of the component scope. Two completely different subcomponents in two different composite EJBs in the same deployment unit could have the same EJB name. It would be completely impractical to uphold the requirement for ejb-name uniqueness in the case of deployment units that may also contain composite components. The uniqueness requirement regarding the EJB name should only apply across peer subcomponents of composite EJBs and also on the top-level of the deployment unit for parentless components. Also, a composite EJB should only be visible and usable within the same scope. Therefore, scope rules have to be defined that increase order within a deployment unit and prevent name clashes and namespace pollution caused by conflicting components.

In order to declare rules for the visibility of a given component, all components deployed on an application server are arranged in a model of vertically stacked layers. The layer a component is placed in ultimately determines the component's visibility. Top level components are components that have no parent component. Their potential direct child components are placed in the layer below. And these child components' respective child components are placed in the next lower layer, and so on.

The scope of EJB components deployed on an application server that implements the composite EJB framework described in this paper is hereby defined as follows:

1. A subcomponent of a composite EJB must…
   • be visible to its parent (the composite EJB).
   • not be visible to other EJB components at the same or higher levels as the parent.
   • not be visible to clients that are not part of the deployment unit.
2. The parent component and other components on the same or higher level as the parent component must not be visible to subcomponents.
3. All direct subcomponent peers of a given parent EJB must be visible to each other.
4. Every component must be visible to itself.
5. A top-level component must be visible to every other entity (components and other clients), but not to any subcomponent of any deployed component.
3 Implementing Composite EJB Support

3.1 Deployment Descriptor

In order to support composite EJB, we had to make a few changes to the ejb-jar.xml deployment descriptor defined originally in the EJB 2.0 specification. The first change within the ejb-jar.xml DTD was the addition of the composite element as a child element of enterprise-beans. This new element contains the complete description of the composite EJB component and the declaration of all its subcomponents. Composite EJB components are declared at the same level with all other possible types of EJB components, such as session or entity beans:

```xml
<!ELEMENT enterprise-beans
    session | entity | message-driven | composite)+>
```

The definition of composite is based on the session element, but adds subcomponents and outer-interface-mapping child elements. The subcomponent element contains the declaration of all child components of the composite EJB named in ejb-name. The outer-interface-mapping element specifies how methods of the outer interface of the composite EJB component are mapped to methods of inner subcomponents, for example:

```xml
<outer-interface-mapping>
    <method-mapping>
        <outer-method>
            <outer-interface>example.SecureStore</outer-interface>
            <method-name>storeSecure</method-name>
        </outer-method>
        <inner-method>
            <subcomponent>Controller</subcomponent>
            <subcomponent-interface>IController</subcomponent-interface>
            <method-name>process</method-name>
            <parameter-mapping>1</parameter-mapping>
        </inner-method>
    </method-mapping>
</outer-interface-mapping>
```

An interface mapping is a 1:1 mapping of all methods of an outer composite component EJB interface to the exact same interface on exactly one subcomponent. Using an interface mapping is a convenient shortcut around the verbose details of a method mapping, whereby each method of an outer-interface is mapped to one method of an inner-interface. Each interface mapping can be replaced with an equivalent method mapping. The interface mapping element is declared in the new ejb-jar.xml DTD as follows:

```xml
<!ELEMENT interface-mapping
    (outer-interface, subcomponent, subcomponent-interface)>
```

3.2 Integration into JBoss

We chose JBoss [7] as base for the implementation because of its modular microkernel based architecture making it easy to extend or modify functionality. The microkernel implements the Java Management Extension (JMX) specification [6] and ensures that the minimal functional core of JBoss is independent from parts that implement the latest EJB specification, or other customized parts. This way the core server is flexible enough to allow the incorporation of new requirements, such as the support of composite components.

Each of the standard EJB component types – stateless and stateful session beans, entity and message-driven beans – are supported by a different container implementation within JBoss. The implemented system described in this section adds another, fifth container to this list, in order to support composite components. Currently, this new container only supports session
beans and composite components as subcomponents but support for entity beans as constituents is under development.

An important task of the container is the interface mapping. At deploy-time the outer interface mapping of a composite EJB component is parsed and transformed into a collection of MethodMapping objects. At run-time they are used by CompositeEJB-Container to map an incoming invocation to one of its subcomponent's methods by means of Java Reflection API.

In order to provide support for the EJB scope implementation introduced in section 2.1, all JBoss EJB containers need to support parent-child relationship and have an associated ClassLoader. This makes it possible to associate multiple EJB environments with the constant JNDI namespace java:comp/env. Within the composite EJB environment, the actual component that is associated with an ejb-name or JNDI can therewith be determined depending on the viewpoint of the caller.

3.3 Evaluation

To test our implementation, we have developed an application using the new composite EJB components (cf. fig. 1). The SecureStore component can be used to securely store arbitrary Java objects sent to its store() method. The given object is serialized (converted into a byte array) by the Serializer subcomponent, encrypted by the Encryptor subcomponent, and finally made persistent by the Storage component.

The Encryptor composite component contains a Compressor subcomponent and three different crypto subcomponents each implementing a different encryption algorithm with different security level (low, medium, and high). A high security level also means low encryption speed and vice versa. Data compression is applied before encryption to reduce the entropy of the serialized object representation.

To demonstrate run-time adaptation, the Encryptor component selects a crypto subcomponent based on the current system load. If the system is executing on a high CPU load, then the fast encryption algorithm with low security level is selected. The adaptation is implemented using a custom interceptor for the container of the Encryptor component. The interceptor is activated when the encrypt() method is called and forwards the request to one of the crypto components. To date, this interceptor has to be developed by the component developer but we plan to integrate a declarative description of the adaptational behavior.

4 Related Work

A commonly seen approach to compose EJB components is called the Session Façade pattern [4]. It is a special incarnation of the much more general Façade pattern, which is described in
detail in [5] and it is probably used in almost every complex application based on the EJB architecture.

The hidden subsystem in the case of the Session Façade pattern consists of entity beans, and often also additional session or message-driven beans. The façade itself is a normal session bean that forwards method calls to subsystem parts. The motivation for hiding entity beans behind a session façade is often to reduce expensive fine-grained remote calls to entity beans. The downside of the session façade pattern is that it does not allow the component builder to take relatively simple atomic or primitive components and encapsulate them as a unit into a new, higher-level composite component that shows no difference in terms of composability compared to the original atomic components. There is no clear definition of the subsystem, which is hidden behind a specific session façade – making it very hard to reuse such a subsystem.

[8] describes a system for using the C2 architectural style [9] in combination with EJB to achieve composite components. The implementation consists of a graphical EJB component assembly tool called COBALT, which allows for composition of composite EJB components from individual EJB components. In contrary to our approach, the support of composite components is not handled at run-time but at deployment time by means of generating wrapper code for each used EJB. This wrapper code handles C2 message flow and translates incoming C2 messages into EJB invocations.

5 Conclusions and Future Work

In this paper we described a framework for composite components based on the Enterprise JavaBeans specification. The open-source EJB application server JBoss was enhanced in order to provide container support for composite EJB components. An example application demonstrated the feasibility of this concept.

Composite component are not only an effective means to achieve reuse of software but can also be used to encapsulate run-time adaptation. Our next goal is the development of a declarative description language for the structural adaptation (insertion, replacement, removal) of composite components. We also want to investigate how parameters of a composite component can be mapped to different internal configurations of the subcomponents.

References