Developing Component Systems based on the J2EE: Problems and Solutions*

Michael Fahrmair, Frank Marschall, Sascha Molterer, Maurice Schoenmakers
Technische Universität München
Institut für Informatik
D-80290 München, Germany
{*fahrmair|marschal|molterer|schoenma*}*@in.tum.de*

Abstract

The Java 2 Enterprise Edition (J2EE) seems to be a promising base for the development of distributed information systems. However while developing the information system CROFT based on this technology we experienced, that the J2EE builds a good and convenient platform but its use is not yet a guarantee to obtain a performant and extensible system. Rather a lot of knownledge about the applied technologies and its capabilities and shortcomings is necessary.
We show how to build an extensible system by avoiding fixed associations between the business components. Further we demonstrate how to keep this degree of modularity at the GUI and what is necessary to design a performant solution with Enterprise Java Beans (EJBs). Finally we show how to satisfiy the security needs that are out of scope of the J2EE specification.
These guidelines allow also unexperienced developers to create J2EE-based applications that allow an easy integration of new components and subsystems and are still very performant.

1 Introduction

A white paper by Sun about the Java 2 Platform Enterprise Edition (J2EE) states the following proposition: "The J2EE application model partitions the work needed to implement a multi-tier service into two parts: the business and presentation logic to be implemented by the developer, and the standard system services provided by the J2EE platform" [SUN99]. After developing an information system based on business components on top of the J2EE we argue that this proposition is misleading. So far for developing self-contained, plugable business components [HS99, Sut98, AF98] you do not simply implement the business logic, but to build modular, scaleable and extensible systems you also need to have a profound knowledge of the technical possibilities as well as the technical shortcomings of the underlying J2EE technology.
In this position paper, we introduce some of the missing guidelines and solutions to allow also unexperienced developers to design and implement their components without jeopardizing modularity, scalability and performance. Our experiences result form developing the J2EE-based information system CROFT.
In the following section, we first present the architecture of CROFT and the J2EE reference model on which it's based. In the main section we discuss the problems like extensibility, performance and security followed by the resulting design model of CROFT. This model shows a reasonable mapping of the domain specific CROFT business components to the J2EE architecture. The position paper closes with a conclusion and an outlook on our current and future work.

2 Architecture and the J2EE

CROFT uses a technical standard architecture that can be found in many business information systems. It distinguishes between the following logical layers and is described in depth in [Hir96], [NMMZ00].

- The *presentation layer* contains the components to provide an user interface to the system. In our case we used a web-based HTML interface.
- The *control layer* contains the components to control the process of the user interaction via the GUI with the system. For example the order of the dialogs. Changes made by the user are propagated to the correct business component in the next layer. In CROFT this layer is implemented by server side web components like classes (web beans), Java Servlets and Java Server Pages (JSPs), which dynamically generate HTML pages. The web beans contain reusable client logic that

*This paper originates from the project MARLIN (Models and ARchitectures for Large Information system Networks) and was supported by BMW, Munich

may be used by other kinds of presentation technology like wml files for mobiles phones.

- *The business logic layer* mainly includes the business components and their associations. These components encapsule the business data and main functions to alter this data. A main task of this layer is to maintain the overall system and keep the business components and their associations always in a consistent state. Invalid changes to a business component are prevented and reported as exceptions to the control layer. The business logic was implemented with Enterprise Java Beans (EJBs) as described below.

- *The persistence layer* is the place where all enterprise data is stored and retrieved by the business logic layer. This layer keeps all information in a safe and recoverable place. In our case we used the relational Oracle database for this purpose.

Multiple users can access the CROFT system through the presentation layer simultaneously and may view and change shared business components. The CROFT architecture corresponds with the one defined in the Java 2 Platform Enterprise Edition (J2EE) by Sun [SUN99]. The J2EE is a standard architecture for developing multi-tier, web-enabled business information systems. It includes the JSP and EJB technologies and many others (JNDI, JMS, JDBC, RMI-IIOP, JTA, JavaMail).
The technology relevant for the guidelines presented in this paper is EJB. EJB is an open standard for a server-side component model. Unlike the classic Java Beans, which are often used in the form of user interface elements at the presentation layer, EJB components run within a EJB container at the business logic layer of a distributed system. This container is part of a J2EE platform and offers a number of services, needed to enable distributed, persistent and transactional objects in common business applications.
One proposed benefit using an EJB container is that the developer of EJBs should be released from recurring technical tasks during the implementation. The container implementation for coordinating transactions, multithreading, database connection pooling, security checks, and lifecycle handling can be reused. This should permit the encapsulation of pure business logic within the EJB components and so decisively increases maintainability and modifiability of the new system.
However, our experiences with the development of CROFT show, that the availability of these container services and the use of J2EE concepts does definitly not lead necessarily to a scalable and modular system. The question is *how* to use the J2EE concepts. To ensure modularity and scalibility the business logic must be distributed and mapped to J2EE concepts very carefully. Also the interface design of the EJB components must still take technical issues in account. For example the influence on the performance if remote calls are used. These issues require experienced developers in this field. Developers can not concentrate on the business logic alone.
The next chapter describes the guidelines we have developed to overcome these deficiencies. We also noticed that the J2EE is missing some security concepts, especially concerning object based security checks. In this area we propose some requirements for an extension of the J2EE in the next chapter.

3 Problems & Solutions

3.1 Modularity

One of the major non-functional requirements was the request for an easily extensible system, especially that it should be possible to extend the system by adding new entities in the form of business components and relations without changing and reviewing the code for already existing relations, components or other functionality. Therefore it was necessary to design a modular architecture in which the following rules are applied:
No hard-coded associations: Associations between business components (at the moment there are four entities mapped to business components in CROFT: person, project, contact and document) can be connected by different associations, e.g. "persons are members in projects" or "persons own documents". These associations could be integrated directly into the business components implementing the entity. However, this solution has some disadvantages in case additional business componentes are going to be integrated in the system or in case an existing business component should be reused in another system. Already existing business components need to be changed, if new associations are added to them. This problem can be solved by modeling separate associations.
All associations are handled by a special component called *association manager*. The association manager can be queried for a list of all associations (collection of association EJBs) connected to a given business component (e.g. a document).
This way one is able to build composeable subsystems and furthermore achieves easy extensibility. To use this flexibility also the user interfaces needs to be composeable in a similar way (see paragraph below) and moreover a communication controlling component is needed to connect all these loosely coupled parts (see paragraph about the "folder" component).
Flexible User Interface: To achieve a composable and easy extensible user interface, it needs to be designed modular, in the way it is possible to add displays

for new associations and business components without changing already existing views.
Therefore *Java Server Pages* and *web beans* are used to generate the necessary HTML code to display the corresponding GUIs. For example there is a pair of one web bean and one JSP to display the project homepage. This bean uses another web bean that requests all current associations for one business component (in our example the project-component) instance from the association manager EJB and returns a corresponding JSP with specialized web bean for each association found. This way all associations (like the persons belonging to that project) are resolved in the web bean before they are viewed. These specialized association web beans gather all associated business components. A JSP generates the necessary HTML code to display these components in a list. This list can be included in the project's view JSP. For example, the bean for the the association "person–project" returns an HTML coded list of all persons belonging to this project. Moreover this association web beans contain interfaces for adding and removing entries to the relation.
Using this mechanism it is possible to extend the system later on by an additional subsystem, for example to manage tasks. There might be an additional association added to the project business component containing tasks within a given project. The project view JSP and web bean have not be changed, because the additional relation and its corresponding list (all tasks belonging to a project) is resolved at runtime by the association manager. Its representation as a list is generated by a separate JSP/web bean and automatically added as an additional list in the project's view.
The Folder: For the combination of the flexible user interface elements described above, a general control communication component is needed. The folder component allows the transfer of business components between different masks of the GUI and to collect references to components for further printing, etc. For example it is possible to open a person's public homepage, mark interesting documents and addresses and copy them to the folder. Later on the user may visit one of his or her projects' workspaces and add the interesting documents in the list of documents associated with this project. Afterwards the user has the possibility to visit his or her own workspace and add the gathered addresses.
Result: Due to loosely indirect coupling of business components with separate association EJBs and a central component to resolve the associations at runtime it is possible to design an easily extensible system where new business components can be integrated without changing already existent business components. This extensibility can also be achieved for the GUI by realizing a flexible modular user interface with a general control component, that connects the single user-interface parts and business-components together.

3.2 Performance

Performance is a critical issue in distributed systems. A remote call costs a multiple of time and resources compared to a simple local call. The main advice is to examine use cases of the system and to extract the most common types of requests to optimize their implementation by reducing the number of remote calls. For requests at the persistence layer the number of tables to join should be reduced for frequent calls. Based on these general rules we extracted the following general useful guidelines.
Transfer business components contents per value: The presentation layer requires the data contents of the business components but there is no need to preserve identity by means of a single EJBs instance, where each client application has an own remote reference on it. Instead it is sufficient to get the business component's data and id *per value as a copy* encapsuled in an object. A similar solution is introduced in [Bro99].
This reduces the number of remote calls dramatically as the control layer can perform local calls on these so called *model objects*. Thus a `PersonEJB` does *not* need to have remote `getName()`, `setName()`, `getBirthday()` and `setBirthday()` business methods. Instead the control layer needs to invoke only a single `getModel()` call that returns a `PersonModel` per value, which contains all the data. This model object in turn provides access to it's internal data by the methods `getName()`, `getBirthday()` etc. If a business component has to be changed then the control layer calls the `setModel()` on the EJB. The EJB will in turn store the change in the database.
When to use Entity and Sessionbeans: As mentioned above in most cases a set of business components is just *shown* within lists. Stateless session beans are ideal candidates for providing these sets of business components' data or associations. Examples for such sets are the result set when searching persons or the set of all documents belonging to a person. One single stateless session bean can be shared between thousands of users and they all receive their own copies of the current version of the business components' data from the database. This way the required container resources are minimized. (Note: stateless means not that the bean can not access a common state on the database). The result sets delivered by the session beans should be just sets of model objects and *not* entity bean references.
We used session beans that deliver sets of complete model objects for all web forms that only present lists of entities. Only in those pages where a business component is created, edited or removed, the according entity bean is contacted when the user requests

the change. This is performed by extracting the primary key from the model object and using the standard `findByPrimaryKey()` method from the the entity bean's home interface.
As a general standard we required for each enterprise component a model object, an entity bean to incorporate the business logic to create, change and remove an entity instance and at least one stateless session bean to perform the queries to search for instances in various ways.
Optimize for the most common use during design Another point is to improve the performance to optimize the database requests that will be very often performed. To find these requests a good starting point is to consider the things the system does mostly. In our case this is displaying lists. This includes that attribute values are displayed and security checks are performed. Thus two good candidates for this type of optimization are the retrieving of attribute domain values and the object based authorization checks, which have to be performed for each entry in a list.
Results Due to the careful consideration of performance related issues already during the design use cases we were able to define design guidelines such, that a scalable fast application could be implemented. We believe that these guidelines are useful in general.

3.3 Security

To use the CROFT system in practice an effective security mechanism to protect private or confidential data from unauthorized access is vital. Therefore the J2EE standard offers several facilities. However we had to implement an additional security manager EJB to realize our security requirements.
The security concept of CROFT is based on the notion of ownership. Naturally the owner of a business component always has full access to it and is able to restrict the access for other users and the public. For every business component the following rights can be set:

- right to read for external users
- right to read for users who have an account
- right to write for users who have an account

Capabilities of the J2EE: For the realization of the security concept described above we could use some of the features the J2EE standard offers. The J2EE application server provides access control and user management at the control layer and at the business logic layer. The control layer security is similar to the Apache webserver's capabilities. It can prohibit GET and POST statements to Java Server Pages. Besides it offers basic, form based and certificate based authentication. Within the EJB container every user is mapped to a role. In the deployment descriptor someone may allow or deny the use of certain business methods to all users in a certain role, which is called declarative authorization. Within the business code itself the methods `getCallerIdentity()` and `isCallerInRole()` can be used to realize programmatic authorization.
Object-based authorization: Though the application server allows us to restrict the use of business methods to certain users (roles), this possibility is not enough to implement our security needs. For example it's not a good idea to prohibit access to a document to all users in the *croftuser* role as declarative authorization would allow us to do. In fact someone must be able to modify those documents that are owned by him, by one of his projects or that are marked as writeable for all users. To prove these instance-dependent (not class-dependent) rights we implemented a method `getPermission()` in the `SecurityManager` session bean.
The security managers' methods are used by all business components and by the association EJBs. Latter need to prove that they only return objects the actual user is allowed to see. Further they have to check if the user has the permission to write to a business component, if (s)he wants to add or remove association instances to other business components.
Result We experienced that the container's facilities were not sufficient to fulfil our needs. Hence the implementation of some kind of additional security infrastructure was neccessary. Therefore it would be desireable that the container provides either an object-based security mechanism or a hook where the developer has the possiblity to execute code whenever a components business method is called. If foreign subsystems or components are going to be integrated in the system their business interface must be wrapped to enforce our security policies.

3.4 Design Model

Comprising the guidelines introduced in the last sections, figure 1 shows the final design model of CROFT. It shows a person and a document business component which are associated with the means of a person-to-document component. As you can see there are no direct dependencies betwee the person and the document business components. Thus all other existing business components follow this pattern as well as those that have to be integrated in future.

4 Conclusion & Future Work

From a technical perspective, a standard component model like the EJB component model provides a suitable infrastructure, a programming model to some extent and a (technical) standard architecture like for example the J2EE architecture.

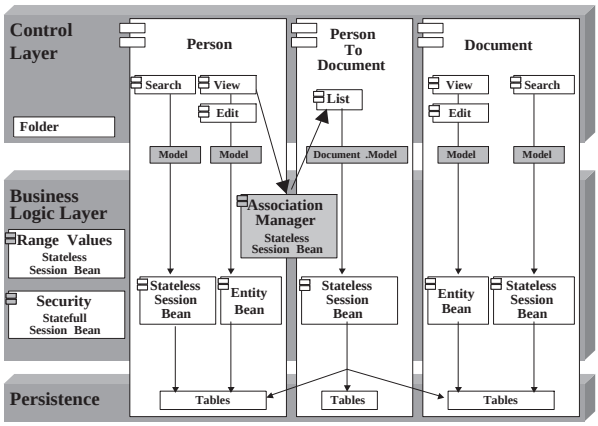

Figure 1. The component structure for persons, documents and an association with their depedencies.

We presented solutions and guidelines for a subset of problems which occurred during the development of the CROFT information system based on the J2EE. The main problems solved included:

- Avoiding unnecessary relations between business components to obtain plug-and-play modules for different CROFT-based systems.
- Avoiding performance problems due to unnecessary remote calls to EJBs and textbook-oriented database calls.
- Introducing a feasible object-based security concept without compromising the modularity of the business components.

With the new EJB 2.0 standard [SUN00], some problems like undermining the modularity of business components through direct, programatic relations will probably be solved. EJB 2.0 introduces container managed relations (CMR). With CMR it should be possible to declaratively add relations between entity beans during deploy time without changing the beans. Although the standard was released in April 2000, there's still no J2EE platform available which implements CMR to test this new possibility. Currently, we extend the

CROFT system by business components which rely on workflow and calendar related functionality. Still using J2EE as the target platform, it's foreseeable that we will have to solve further problems since it is not yet clear, how to integrate a workflow service into the J2EE and how to implement "active" business components, i.e extend J2EE by a service which concurrently invokes business components if certain time constraints are fulfilled.

References

[AF98] Paul Allen and Stuart Frost. *Component-Based Development for Enterprise Systems.* Cambridge University Press, New York, 1998.

[Bro99] Kyle Brown. A small pattern language for Distributed Component Design. http://members.aol.com/kgb1001001/-Articles/PLoP99/brownfinal.pdf, 1999. Presented at the EuroPLoP'99.

[Hir96] R. Hirschfeld. Three-tier distribution architecture, 1996.

[HS99] Peter Herzum and Oliver Sims. *Business Component Factory : A Comprehensive Overview of Component-Based Development for the Enterprise.* John Wiley & Sons, Inc., New York, 1999.

[NMMZ00] J. Noack, H. Mehmaneche, H. Mehmaneche, and A. Zendler. Architectural patterns for web applications, 2000.

[SUN99] Java 2 Platform Enterprise Edition Specification, Version 1.2. Sun Microsystems (http://java.sun.com), December 1999.

[SUN00] Enterprise JavaBeans Specification, Version 2.0. Sun Microsystems (http://java.sun.com), May 2000.

[Sut98] Jeff Sutherland. Business Object Component Architectures: A Target Application Area for Complex Adaptive Systems Research. In Delip Patel, Jeff Sutherland, and Joaquin Miller, editors, *OOPSLA Workshop Proceedings on Business Object Design and Implementation II.* Springer, London, 1998.