Guidelines for Developing Adaptive Plug-and-Play Business Component Systems based on the J2EE[*]

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) is presented as a framework for the development of integrated, flexible and adaptive enterprise information systems. However, the J2EE is only a technological basis for the development of such systems. Its use is no guarantee to obtain a performant and extensible system based on integrated Business Components. In this paper we present guidelines for building enterprise information systems based on self-contained, plugable Business Components on top of J2EE. These guidelines allow also unexperienced developers to create J2EE-based applications that allow an easy integration of other Business Components and subsystems. Our experiences result form developing the J2EE-based information system CROFT.

Keywords: *J2EE, Business Components, Integration, Plug-and-Play, EJB, Enterprise Applications, Distributed Systems, Java*

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" [SUN99a]. 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 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. The functional goal of CROFT (Cooperative management of Resources and Organisational information For Teams) is to combine the creation and presentation of information about projects and teams for a third party (i.e. the website of our chair) with the management and cooperative use of these resources within the teams' every day business (i.e. sharing documents and addresses in our research projects). The non-functional requirements of CROFT are to use reusable, modular business components for each of the resource and organisational information entities (e.g. "person" or "document") and a pure HTML-based user interface. Furthermore, the system should be easily extensible by additional business components (for example "task") if required without redesigning the whole information system.

In the following section, we first present the requirements of the CROFT system in detail. The next section shows the architecture of CROFT and the J2EE reference model on which it's based. In

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

the main section we discuss the problems and solutions 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 paper closes with a conclusion and an outlook on our current and future work.

2 Requirements

The CROFT systems purpose is to support the chair's staff in managing and presenting their addresses, documents and projects. The CROFT system is a web-based project management tool. The business entities it manages are:

Persons - the chair's staff, these are the users of the system. One person maps exactly to one user account with a unique login and password in the CROFT application.
Projects - a project may have any number of persons as members and exactly one responsible person.
Documents - any kind of document a person or project may need. The document itself is a web resource that is referenced by its URL.The system saves only metadata about a document. If necessary any BibTex compliant information can be saved.
Addresses - these are addresses of the chair members as well as the addresses of other contacts such as industry partners, other academic sites etc.

Persons and projects can own or reference documents and addresses. Figure 1 shows the different entities and their ownership and member associations.

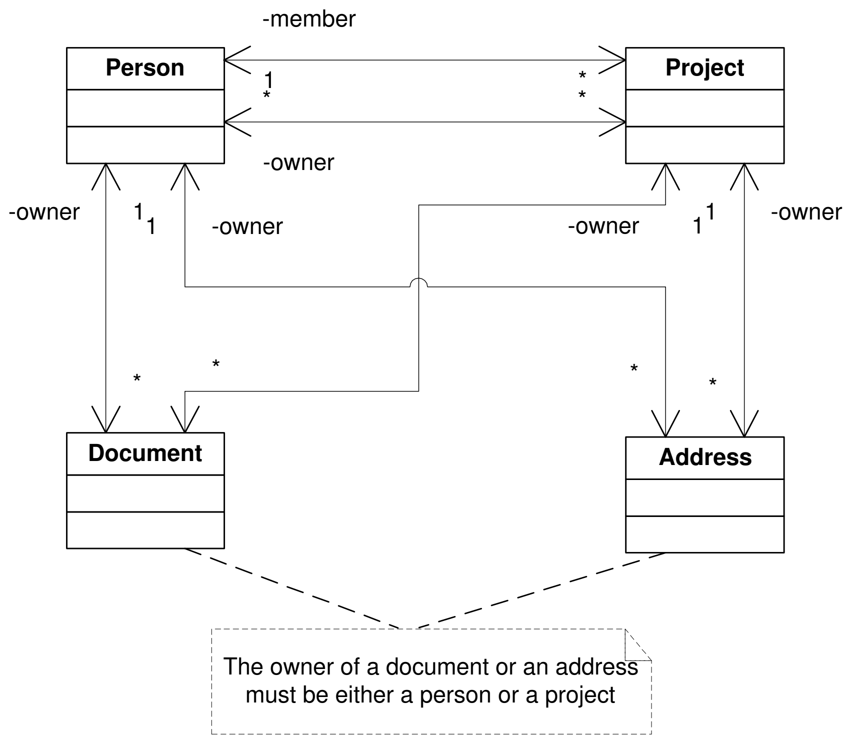
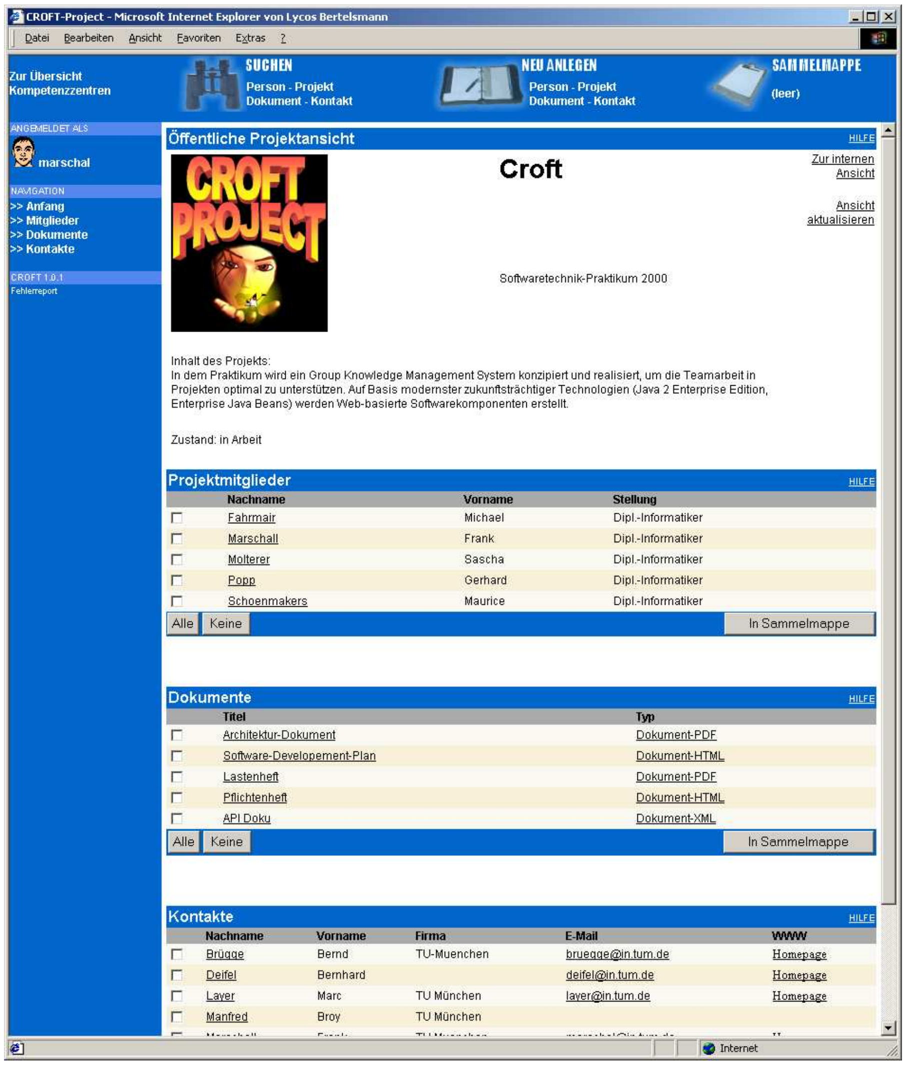


Fig. 1. The basic entities and their most important relations



Öffentliche Projektansicht

Croft

Softwaretechnik-Praktikum 2000

Inhalt des Projekts:
In dem Praktikum wird ein Group Knowledge Management System konzipiert und realisiert, um die Teamarbeit in Projekten optimal zu unterstützen. Auf Basis modernster zukunftsträchtiger Technologien (Java 2 Enterprise Edition, Enterprise Java Beans) werden Web-basierte Softwarekomponenten erstellt.

Zustand: in Arbeit

Projektmitglieder

	Nachname	Vorname	Stellung
☐	Fahrmair	Michael	Dipl.-Informatiker
☐	Marschall	Frank	Dipl.-Informatiker
☐	Molterer	Sascha	Dipl.-Informatiker
☐	Popp	Gerhard	Dipl.-Informatiker
☐	Schoenmakers	Maurice	Dipl.-Informatiker

Dokumente

	Titel	Typ
☐	Architektur-Dokument	Dokument-PDF
☐	Software-Developement-Plan	Dokument-HTML
☐	Lastenheft	Dokument-PDF
☐	Pflichtenheft	Dokument-HTML
☐	API Doku	Dokument-XML

Kontakte

	Nachname	Vorname	Firma	E-Mail	WWW
☐	Brügge	Bernd	TU-Muenchen	bruegge@in.tum.de	Homepage
☐	Deifel	Bernhard		deifel@in.tum.de	Homepage
☐	Layer	Marc	TU München	layer@in.tum.de	Homepage
☐	Manfred	Broy	TU München		

Fig. 2. A projects' homepage within the CROFT system

We describe these associations in more detail in section 4.3. The system allows its users to create, modify, view and delete these entities as well as to search for certain entries using a HTML interface. Therefore the GUI contains permanently present links for each kind of entity to open a search dialog or a dialog for creation of an entity. Wherever an entity is presented in a list, such as a search result list, a link leads to its view page. Only if a user has the necessary rights (s)he may reach another dialog to change this information from there.

Users and projects have public homepages where all information of the project or person, its ownership and references to other entities of interest are presented. Beyond this CROFT provides similar private web pages, so called *workspaces*, for every user and project. Figure 2 shows an example of a project homepage.

A main requirement to the CROFT system is its easy extensibility. It must be possible to integrate new business components and subsystems into the existing system with very little effort. This includes not only the realisation of new business components but also the integration of existing foreign ones. These new components must fit into the system's infrastructure and security mechanisms. Ideally it should be possible to integrate a foreign bussiness component without changing the CROFT infrastructure core or the new component. This allows to assemble existing business components or subsystems to applications for different purposes, based on the core CROFT system. For example someone might use the system to manage customers and bills, using the CROFT core infrastructure that includes object-based security, user profiles, etc., adding the necessary components "customer" and "bill".

Besides this a change or extension of any existing business component must only cause a minimum of necessary modifications at very few locations within the code. It also has to be taken into account that the GUI may be changed or eventually completely replaced, e.g. by a WAP interface or a Java application.

To allow an efficient use of the application certain performance demands must be fulfilled. Even a complicated or unusual request has to be completed within a few seconds. However, a user should get some kind of feedback that his request is being processed immediately. This condition should be attainable with 50 parallel sessions active, producing an average of two requests per second.

To ensure a maximum of platform independence the implementation of the CROFT system is based on the J2EE. Remaining platform dependencies like database product specific code are encapsuled in a small set of classes.

Within the following section we give a brief overview of the architecture we have chosen to realize our system requirements.

3 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 may be used for 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, 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.

Figure 3 shows the relations between the different layers.

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 [SUN99a]. 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

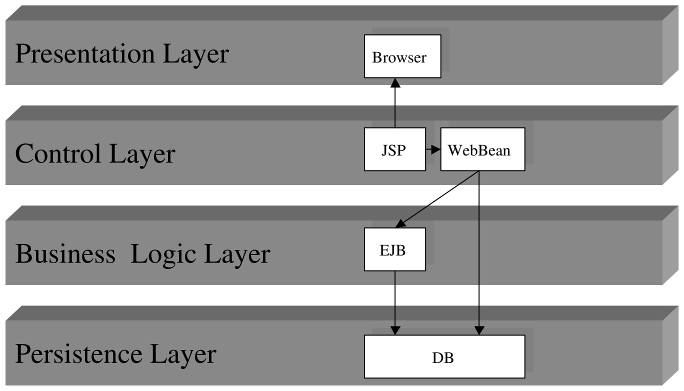


Fig. 3. The basic layers and component types with their most important relations

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.

Two different types of EJBs essentially exist: Session Beans and Entity Beans.

Session Beans are components whose state is not persistent. They are usually used for the control of processes (e.g. a dialog process between client and server). Session Beans can have a conversational state for each session (e.g. member-variables of a class), though. In this case they are called *Stateful Session Beans*. This condition is, however, irrecoverable at a system crash or restart. Session Beans without any state are called *Stateless Session Beans* and the same bean can shared amoung multiple users.

Entity Beans are persistent components, i.e. their state is persistently saved, for example in a database. In the case of a system crash, the state an entity bean had after the last completed transaction is available again. An entity bean often represents a row in a database table. If the persistence guaranteed by the container (*container managed persistence*) the deployment descriptor of the bean indicates, which fields of a database table correspond to which attributes of the Bean. The container then takes care about storing this data. With *bean managed persistence* every enterprise bean is itself responsible for the storage of its state. Therefore usually the Java Database Connectivity Interface (JDBC) is used.

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.

4 Problems & Solutions

4.1 Modularity

One of the major non-functional requirements was the request for an easily extensible system (see section 2), 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 components 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 modelling separate associations:

In the business (EJB)-tier each kind of association is realized by a separate EJB that manages the associations between two kinds of business components. An association EJB needs to implement the following functionality:

– Create new associations between business components and check consistency conditions.
– Delete associations between business components and check the consistency.
– Return a list of all business components instances having associations to a given business component instance.
– Return the type of associations (reference, business association, or ownership association).

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).

Using this method it is possible to extend the system by new associations, e.g. an association between the existing business component "person" and a newly introduced component "task" without changing the person component. This is possible because the association manager can be used to detect the new association between with the person and the task components. A separate association-EJB "person–task" introduces all necessary code to create and delete this association.

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 instance (in our example the project-component) from the association manager EJB. For each association found a corresponding JSP with a specialized web bean is returned. This way all associations (like the persons belonging to that project) are resolved in the web bean before they are viewed. Each specialized association specific web bean gathers all associated business components using the association EJB. The JSP generates then the necessary HTML code to display these component instances in a list. This list can be included in the enclosing project's view JSP. For example, the JSP and its web bean for the association "person–project" returns an HTML coded list of all persons belonging to this project. Moreover this association specific web bean and HTML list contains interfaces for adding and removing entries to the relation. Thus links between the business component instances can be created using the association EJB via this HTML fragment and the attached association specific web bean.

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 its web bean do not have to 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.

The folder component acts on a very abstract level since it doesn't have to know about the type of the components it contains. Thus it needs not be changed if further business components or associations are added. The folder component acts as an indirect connection between the different business components' presentations instead of adding hard coded connections between them (e.g. a button "add marked contacts to my private addresses") that needs to be updated or changed with every new business component or association introduced.

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.

4.2 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.

System Usage Analysis: A main lesson is that performance issues must be considered from the beginning before system design starts. They basically influence the design and our solutions show also that other goals like modularity do not have to be sacrificed for a high performance. The main question is: will the system be accepted by its end users or not? During system usage analysis another important question has to be answered: How up to date does the displayed information have to be? For a web based information system the most *frequent* thing a user does is to *view* information. *Infrequently* users will select dedicated business components and *change* them. Users will also seldom *create* or *remove* business components and associations.

The business components are shown and edited in a web form. If the user wants to store the changes then the final request for changing is sent to the business logic layer and the change is performed within a transaction. However while a user views or edits an object within a web form somebody else may have changed this object. For most systems this case rarely occurs and mostly it is sufficient to inform the user that somebody else changed the object.

In most applications it is not necessary to have a permanent up-to-date presentation of the data on every screen on every object like promoted by the observer concept [GHJV95]. Especially a web user does not expect this behaviour. A simple refresh or reload will suffice in most cases.

These assumptions result in design rules described now in more detail.

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.

Creating entity beans and handling their associations is very resource consuming. While container implementations *may* contain code to improve the performance by caching, it still does consume enormous resources when a container holds references to thousands of objects that are only displayed. For each displayed object the container must perform a lot of computation, just for the unlikely event that a user might change it. However creating and referencing entity beans is only necessary while *creating*, *removing* and *changing* a business component. As a consequence we used session beans that deliver sets of complete model objects for all web forms that present lists of entities. Only in those pages where a business component is created, edited or removed, the corresponding 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 entity bean's home interface. We thus use a similar use of session beans as proposed by the wrapped entity bean solution described in [Val99]. However our solution defers the required entity bean instantiation until its really needed.

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. For each association a separate stateless session bean is required to deliver the set of persons belonging to a single document and vice versa and incorporate the business logic for adding or removing single association links. For example for the entity person there are a `PersonModel`, a `PersonEntityBean` and a `PersonSearchBean`. For the association between documents and persons there is a `PersonToDocumentBean` that uses `PersonModel` and `DocumentModel` objects.

Optimise for the most common use during design: Another point is to improve the performance to optimise the database requests that will be performed very often. To find these requests a good starting point is to consider the things the system does mostly. In our case it is displaying lists. This includes that attribute values are displayed and security checks are performed. Thus two good candidates for this type of optimisation are the retrieving of attribute domain values and the object based authorization checks, which have to be performed for each entry in a list. We now look at these two cases in more detail.

Often only certain values for attributes are allowed. For example for the attribute `Person.title` only a single value from a dedicated set of values such as “Mr.”, “Miss.” and “Prof.” can be selected. These values are usually stored in separate database tables and a foreign key is stored in the title attribute of a persons table entry (this is done because the value may vary according to the language used in the user interface). Instead of performing joins or even single requests to retrieve the values for a certain key we used stateless session beans to cache these values. This improved the overall performance dramatically. The use case that somebody changes these values is very unlikely, thus the cost of the caching, sporadic invalidating and reloading from the database is low compared to the costs of ongoing requests.

When performing object based security checks the system often has to navigate through the model to check if a person has a certain right. For example a person is logged in and wants to see a project with a list of documents. The system has to check for each document whether the document belongs to a project, whether the person that is logged in is a member of that project and if the document is marked to be displayed for project members. These frequent requests can easily be optimised by caching the projects where a person is a member in a statefull session bean inside the business logic layer. If the `DocumentModel` is also extended by the foreign key of its owner, no more database accesses are required. Again the use case that a person becomes a member of a project, without knowing it himself, is (or at least should be) very unlikely. Thus the restriction that a new project member who is logged in must logout and login again to see the complete accessible list is acceptable considering the performance gain.

In general one should look at the most common use cases and identify their required database requests, especially those with many joins or those that are often called. Then consider roughly the probability that a result set differs from the previous one by looking at the use cases that would cause a different result set. If these use cases are very unlikely, look at a simple way to reduce the number of joins and requests. Either by flattening tables or caching results in statefull or stateless sessions beans.

We noticed that using this approach based on use cases helps to identify performance bottle necks faster than profiling. But even more important is that potential bottle necks are already found during the design phase to prevent them from creeping in during development. This draws the developers’ attention onto the important performance issues right from the beginning. Profiling however helps to identify ”unexpected” existing performance killers later on.

Results: Due to the careful consideration of performance related issues already during the design use cases we were able to define simple design guidelines such, that a scalable fast application could be implemented even by inexperienced developers. We believe that these guidelines are useful in general.

4.3 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.

Ownership concept and security: To organize the different entries every business entity except persons must have exactly one owner as already shown in picture 1 in section 2. This may be a person or a project, only projects are always owned by a person. An owner may hand over the ownership of an entity to any other user or project in the system.

The security concept of CROFT is based on this 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.

User concept and authorization: Within the system two roles are defined: *croftadmin* and *croftuser*. While croftadmin has all rights, croftuser is the role of a user with an account in the system, which corresponds to exactly one person business component in the system. There is only one user admin, who has the role croftadmin and croftuser.

Since the J2EE reference implementation does not support the creation of a new user account at the runtime we had to write an own SecurityAdapter. This adapter allows us to add and delete users while the system is running. It uses some container specific code and is similar to the solution chosen in the Sun's pet store example [SUN99b]. The methods to create and delete a user are located in a session bean "`SecurityManager`" which encapsules all security-specific parts of the application.

Declarative authorization: At the control layer two security constraints are defined in the deployment descriptor. The first one makes a set of Java ServerPages accessible for croftusers only. These are all pages where business components are manipulated, created or removed. As a result a guest user will be asked for her/his login and password whenever she or he tries to access such a page. The second security constraint allows only users in the role croftadmin to view some pages like "Create Person", etc.

Within the EJB tier declarative security is used to prevent guest logins from accessing any business method with writing access to the system. This seems redundant, but in fact the purpose of the web tier security settings is only to prevent users from seeing masks they can't really use. At the EJB-tier the security constraints are finally enforced.

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 manager's 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 necessary. Therefore it would be desirable that the container provides either an object-based security mechanism or a hook where the developer has the possibility 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 completely wrapped to enforce our object based security policies.

4.4 Design Model

Comprising the guidelines introduced in the last sections, figure 4 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 between 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.

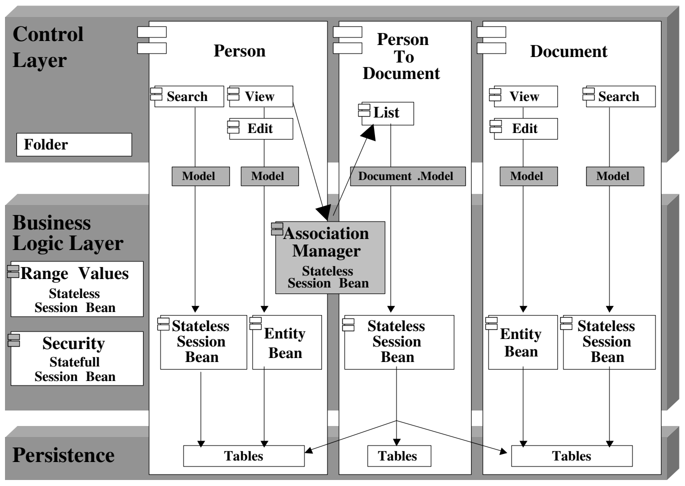


Fig. 4. The component structure for persons, documents and an association with their dependencies.

5 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.

During the development of CROFT, we made the experience that the main problem is that the J2EE does not state much about the structure, granularity, and interconnection of components implemented on its top. There is still a gap between specification and implementation models, between a multiplicity of technically possible options and a small set of reasonable solutions. There is also a lack of guidelines how to develop modular, reusable business components based on the J2EE to build productive, flexible, efficient and scalable information systems.

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.

Nevertheless, platforms like the J2EE, providing a standard component model, an object infrastructure (persistence, life-cycle, transaction control and security) together with an appropriate programming model for these components will help to establish a standard way of building scalable and flexible information systems with reusable, plug-compatible business components. However, future work will be necessary to enhance the J2EE to avoid too much coupling between business and technical logic as well as introducing more patterns and example-architectures for developers in such a way that the proposition quoted in the introduction will come true.

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.

Acknowledgments We would like to thank the whole CROFT team for a lot of overtime work.

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 PLoP'99.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable objectoriented software, 1995.

[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.

[SUN99a] Java 2 Platform Enterprise Edition Specification, Version 1.2. Sun Microsystems (http://java.sun.com), December 1999.

[SUN99b] Developing Enterprise Applications with the Java 2 Platform, Enterprise Edion, Version 1.0. 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.

[Val99] Thomas Valesky. *Enterprise Javabeans: Developing Component-Based Distributed Applications.* Addison Wesley, New York, 1999.