Generating a Pattern-Based Application Development Environment for Enterprise JavaBeans

Imed Hammouda and Kai Koskimies
Institute of Software Systems, Tampere University of Technology
P.O. Box 553, FIN- 33101 Tampere, Finland
{imed, kk}@cs.tut.fi

Abstract

Enterprise JavaBeans (EJB) is Java's component architecture for server-side distributed enterprise applications. The architecture of EJB applications is based on well-established solutions common to most distributed business systems. To utilize the architecture in an optimal way, proven EJB specific design solutions have been identified and collected as a set of design patterns. The use of these EJB design patterns as a tightly interconnected pattern system can significantly ease the development process of EJB based applications and improve the quality of the produced software. We will study in this paper how a general architectural tool (Fred) can be used to generate an EJB programming environment, when given the specifications of the EJB design patterns as input. This environment can be viewed as an architecture-centric wizard that guides the user through the development of the application, following the EJB design patterns.

1. Introduction

Enterprise JavaBeans (EJB), part of the Java 2 Enterprise Edition Platform, is an architecture for setting up program components that run in the server parts of a client-server system. This architecture allows developers to quickly create and maintain scalable enterprise applications at lower cost and higher usability. Developers need to concentrate on writing the business logic rather than dealing with general distributed systems problems such as persistency, transaction management, security etc. EJB architectural standard is currently supported by many products from different vendors, implementing a software platform for distributed business applications. An important advantage of EJB is that applications become independent of the environment and can be easily transported from one environment to another, assuming that both support EJB.

Enterprise JavaBeans development is still an evolving art. During the last few years, EJB has gained a lot of popularity in industry, but there is still relatively little experience in writing high-quality EJB applications among the majority of EJB developers. Hence it is extremely important that proven, good design solutions are systematically collected and made available to EJB developers. These solutions are known as *EJB design patterns*. Recently, a collection of EJB patterns has been published in a text book [2] and widely distributed in the industry, following the traditions of [1].

While EJB patterns are being recognized and published, several commercial and non-commercial software products with EJB tool support have been released and adopted by the industry. These systems offer a development environment for EJB applications, providing assistance in generating code, testing the applications and deploying them within the environment. Some environments even allow the user to select certain EJB patterns and generate code according to those patterns [7]. The idea of having programming environments dedicated to a certain category of applications is of course not limited to EJB. Such environments are called Application Development Environments (ADE).

However, a general problem with the conventional ADEs is that the tools hide the architectural aspects and the design decisions of the automatically produced code. Hence, the programmer has little hope to be able to understand the generated source code, and to modify it when necessary. Maintenance becomes problematic if the generation steps cannot be repeated, for instance after adding hand-written code. This is a common problem of tools that generate large portions of source code without user intervention.

Another significant limitation of the ADEs is that the tools assume a particular architecture and code generation patterns. The user can affect the produced code by changing some parameters, but the overall structure of the code is predetermined. This implies that it is hard to exploit the tools for systems that do not comply with the architectural assumptions of the ADE. It may also be hard to integrate the generated code with legacy code. In the case of an EJB-based ADE, the environment should understand and support EJB patterns, but the problem is that the set of EJB patterns evolves and grows all the time, and there is often need for domain-specific or even company-specific patterns. Hence no predefined set of patterns can be sufficient for all purposes and for a long time.

These observations suggest that an EJB ADE should adapt itself to arbitrary architectural or design patterns, given by the user of the ADE. The pattern specifications should serve as an engine that runs a generic ADE, guiding the programmer through the architecture and making sure that the given patterns are followed. We also propose that an EJB ADE should be open in the sense that it does not perform mysterious generation actions behind the scenes, but rather offers an intelligent architecture-sensitive editor. If the ADE generates code, it should do it only in a limited context where the programmer can easily understand the meaning of the code. For example, the code generated for an individual attribute or method of an existing class can be usually understood by the programmer fairly easily, whereas generating a whole set of classes leaves the programmer hopelessly without track of the process.

There have been several attempts to formalize the concept of a (design) pattern and to develop tool support based on such formalizations (for a recent work and summary of existing approaches, see e.g. [6], [4]). These approaches are often motivated by the need to define the extension interface of application frameworks, so that the application-specific code can be given according to the requirements of the framework. Such an interface specification can be defined as a set of patterns. In this context, a pattern is viewed as a set of *roles* and *constraints*. Each role can be bound to a program element (say, class), and the constraints define requirements that must be satisfied by the elements bound to certain roles. A framework extension point is defined by a pattern in which some of the roles are bound to program elements in the framework and some roles are left unbound. The latter roles will be bound to application-specific program elements, following the constraints.

In this paper we demonstrate that it is possible to solve the above mentioned problems of conventional ADEs for EJB by applying a general pattern-based approach for architectural modeling and framework tool support. We will utilize a prototype tool called Fred ("Framework Editor", [3], [4]) providing pattern-based assistance for framework specialization. We show that the specialization pattern concept of Fred fits the EJB patterns very well. A covering set of EJB patterns has been specified as Fred patterns, and an ADE for EJB has been produced by Fred based on those patterns. The resulting ADE has been used to develop a small example EJB application to demonstrate the capabilities of the pattern-based environment.

We argue that the Fred-based EJB environment in particular solves the basic problems with ADEs discussed above: hidden code generation cycle and limited extensibility. The Fred-based environment allows the programmer to select an EJB pattern, and apply it in the system through a sequence of small tasks. A pattern specification can be augmented with instructions to carry out each task. The programmer understands the purpose of the pattern and the roles of its parts, seeing its construction step by step. The environment is not bound to a fixed set of architectural patterns, but new patterns can be easily introduced to the system at any time. The pattern descriptions can be generated automatically as XML files, to be further processed for documentation purposes.

In addition, the Fred-based approach offers extra benefits that are not found in conventional EJB environments:

EJB sensitive source code editor. The integrated Java editor keeps track of the application of EJB patterns and checks the constraints of the patterns during editing on the fly: after each editing action the constraints are re-checked for the changed parts of the source, and possible conflicts are notified as new mandatory tasks for the user. When the user corrects the source text, these tasks automatically disappear.

Application-oriented instructions for EJB patterns. The task lists and the associated instructions are not static but generated dynamically taking into account the decisions and application-specific names given so far. Hence the instructions are not on the abstract level of general EJB patterns, but on the concrete level of a particular application. This makes the instructions much easier to understand and follow.

All these benefits are direct consequences of using Fred. To produce this kind of EJB ADE with Fred, one only needs to specify the required EJB patterns using Fred's pattern tool, and give these specifications as input for Fred. The EJB patterns specialize the generic Fred environment into an EJB environment.

The remaining of the paper is organized as follows. In the next section we discuss the main features of the EJB patterns used in this work, and how they are organized as a small pattern system. An overview of the Fred environment is presented in Section 3. In Section 4, we show how a representative EJB pattern is specified in Fred. In Section 5 we use a case study to demonstrate our approach. Section 6 compares the work to other existing solutions. Finally, in Section 7 conclusions are drawn and possible future work is highlighted

2. An EJB Pattern System

In the past few years, the EJB community has come up with a multitude of ideas on how to optimize the architecture and deployment of distributed business applications. Since then, the term “EJB Design Pattern” has become a hot topic in the on-going discussions [2]. Some of the patterns are built on other patterns described in well-known literature such as [1] whereas others are unique to the EJB technology. Some EJB patterns deal with certain category of applications while others can be considered in the design of most distributed enterprise systems.

In this section, we will concentrate on patterns that might improve the architecture of the business logic tier, paying less attention to those of the presentation and data tier. In addition, we will discuss few integration tier patterns. Rather than going into details, we will just give a brief overview of each pattern. More details about the patterns can be found in [2]. Nevertheless, we will choose to closely study the Session Façade pattern to show how it is translated into a Fred specialization pattern in Section 4. Finally, we discuss the relationships of these patterns and present them as an integrated pattern system that can be used by an EJB developer to cover central parts of the design.

Session Façade

The communication between the presentation layer and business layer in distributed business applications often leads to tight coupling between clients and the business tier. The interaction could get so complex that maintaining the system becomes difficult. The solution to this problem is to provide a simper interface that reduces the number of business objects exposed to the client over the network and encapsulates the complexity of this interaction. At run-time, the client calls a method on a Session Façade, which in turn calls several methods on individual business objects. Figure 1 shows the UML class diagram representing the Session Façade pattern.

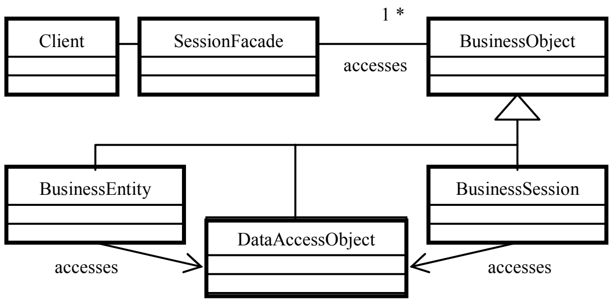


Figure 1. Structure of the Session Facade pattern

The primary benefit of Session Façade is to provide a centralized control over the business tier and ease of understanding and the maintainability of the system. In addition, the façade represents an access control layer to manage the relationships between user requests and business methods, and a transactional control layer where a transaction starts by calling a number of methods on the individual entities and commits by returning to the client. This pattern is based on the Façade pattern in [1].

Value Object

In J2EE applications, the client needs to exchange data with the business tier. For instance, the business components, implemented by session beans and entity beans, often need to return data to the client by invoking multiple get methods. Every method invocation is a remote call and is associated with network overhead. So the increase of these methods can significantly degrade application performance. The solution to this problem is to use a Value Object to encapsulate the business data transferred between the client and the business components. Instead of invoking multiple getters and setters for every field, a single method call is used to send and retrieve the needed data.

Business Delegate

By using the Session Façade pattern, we did not rule out all the design problems involving the interaction between the client and the business layer. We do have a centralized access to the business logic but still the session bean itself is exposed to the client. Enterprise beans are reusable components and should be easily deployable in different environments. Changes in the business services API should not affect in principle the implementation of the beans. To achieve loose coupling between clients at the presentation tier and the services implemented in the enterprise beans, Business Delegate Pattern is used. This hides the complexities of the services and acts as a simpler uniform interface to the business methods.

Service Locator

In J2EE applications, clients need to locate and interact with the business components consisting of session and entity beans. The lookup and the creation of a bean is a resource intensive operation. In order to reduce the overhead associated with establishing the communication between clients and enterprise beans (clients can be other enterprise beans), the Service Locator Pattern is used. This pattern abstracts the complexity of the lookups and act as a uniform lookup to all the clients.

Data Access Object

Enterprise Java Beans are reusable components and should be deployable in different environments with lesser effort. Implementing a so-called bean-managed persistence entity bean means that the programmer should provide all the persistent code (JDBC code). However, the API to different databases is not always identical so the bean programmer should consider different persistence code for different data sources. Depending on the data source, one specific implementation is used. To make the enterprise components transparent to the actual persistent store, the Data Access Object pattern should be used.

In addition to the standard EJB design patterns above, we have defined several custom patterns that have been translated to Fred specialization patterns.

Primary Key Pattern

Entity beans need to be occasionally stored in the database and loaded to memory to guarantee data consistency. Also some finder operations need to uniquely identify the entity bean in the underlying storage. Such operations need to have a primary key object that would allow the environment to uniquely specify the target bean. Having a separate primary key class for each entity bean is optional in EJB technology. However, in order to use the environment, we have chosen to abstract primary key data in a separate primary key class. This would enhance the readability and the maintainability of the system.

Session (Entity) Bean Pattern

EJB is a component architecture that offers a common standard for distributed applications. Session beans (as well as entity beans) share the same architectural skeleton. This makes cross-vendor, cross-platform components easy to integrate together. In our environment we have defined a programming pattern for session beans and another for entity beans. The persistence of an entity bean can be either container-managed or bean-managed. Session beans can be either stateless or stateful.

Tester Pattern

This is a simple pattern that acts as a test client for the generated enterprise beans. The client tests all possible operations on beans such as create, finder and custom business methods. The behavior is observed through console output.

Pattern system

In order to generate a working development environment, we need to put these patterns together in an integrated scenario to form a more comprehensive solution to EJB application development. Figure 2 shows a pattern system for EJB applications.

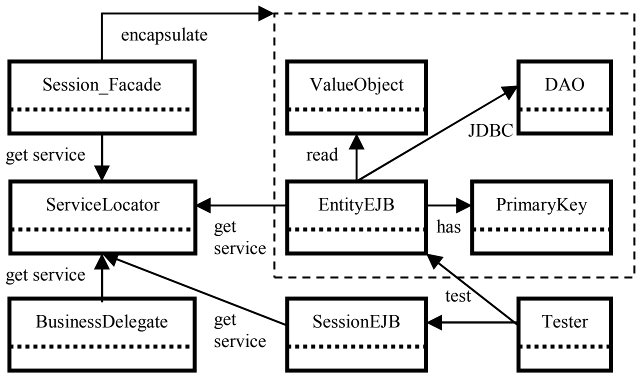


Figure 2. An EJB pattern system

The environment generates skeleton code for both entity and session beans. Every generated entity bean has one primary key class and at least one Value Object. If the entity bean has bean-managed persistence then a Data Access Object is used in order to encapsulate JDBC code. A number of Session Façades are used to encapsulate entity beans. In order to lookup and use bean instances and home objects, the Session Façade gets the service from the Service Locator pattern. Other clients as well, including session and entity beans, locate other beans using the Service Locator. Individual beans could be tested using the Tester pattern.

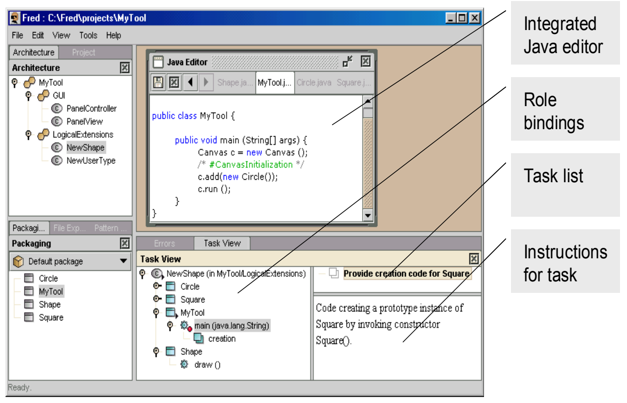
3. Fred Environment

A basic concept for defining the architectural units in Fred is a *specialization pattern*. In Fred this concept is typically used for an abstract structural description of an extension point of a framework. Basically, a specialization pattern is a specification of a recurring program structure. It can be instantiated in several contexts to get different kinds of concrete structures. A specialization pattern is given in terms of *roles*, to be played by (or bound to) structural elements of a program, such as classes or methods. The same role can be played by a varying number of program elements. This is indicated by the *multiplicity* of the role; it defines the minimum and maximum number of bindings that may be created for the role. Combinations are from one to one (denoted (1,1)), from zero to one (0,1), from one to infinity (1,n), and from zero to infinity (0,n). A single program element can participate in multiple patterns.

A role is always played by a particular kind of a program element. Consequently, we can speak of class roles, method roles, field roles etc. For each kind of role, there is a set of *properties* that can be associated with the role. For instance, for a class role there is a property inheritance specifying the required inheritance relationship of each class associated with that role. Properties like this, specifying requirements for the concrete program elements playing the role are called *constraints*. For example, a simple inheritance pattern might consist of roles Base and Derived, with a constraint stating that the class bound to Derived must inherit the class bound to Base; this is called an inheritance constraint. Another constraint might state that the program element bound to a particular role must contain the element bound to another role; we call this a containment constraint. It is the duty of the tool to keep track of broken constraints and instruct the user to correct the situation. Other properties affect code generation or user instructions; for instance, most role kinds support a property default name for specifying the (default) name of the program element used when the tool generates a default implementation for the element.

Roughly speaking, Fred generates a task for any role-element binding that can be created at that point, given the bindings made so far. A task prompting the creation of a binding is mandatory if the lower bound of the multiplicity of the corresponding role is 1, and there are no previous bindings for the role; otherwise the task is optional. Fred generates a task prompt also for an existing binding that has been broken (e.g., by editing actions).

The central part of the user interface of the Fred environment shows the current bindings of the roles for a selected pattern, structured according to the containment relationship of the roles. Since this relationship corresponds to the containment relationship of the program elements playing the roles, the given view looks very much like a conventional structural tree-view of a program. In this view, a red spot marks undone mandatory tasks, optional tasks are marked with a white spot. The actual to-do tasks are shown with respect to this view: for each bound role selected from the view, a separate task pane shows the tasks for binding the child roles, according to the containment relationship of the roles. The user interface of Fred is shown in Figure 3.



```
public class MyTool {

    public void main (String[] args) {
        Canvas c = new Canvas ();
        /* #CanvasInitialization */
        c.add(new Circle());
        c.run ();
    }
}
```

Code creating a prototype instance of Square by invoking constructor Square().

Figure 3. Fred interface

The application is built following the tasks generated by the tool. The tasks can be carried out by indicating the existing program element that plays the role, by asking the system to generate a default form of for the bound element as specified by the pattern, or simply by typing the element using the Java editor and then binding it to the role. The system checks the bound element against the constraints of the pattern and generates remedial tasks if necessary. The task list evolves dynamically as new tasks become possible after completing others. The application programmer immediately sees the effect of the actions in the source code. Each individual task can be cancelled and redone. Hence the development process becomes highly interactive and incremental, giving the application programmer full control over the process.

An important feature of Fred is its support for adaptive user guidance: the task prompts and instructions are dynamically customized for the particular application at hand. This is achieved by giving generic task title and instruction templates in the pattern specifications, with parameters that are bound at pattern instantiation time. The actual parameters can be taken, for instance, from the names of the concrete program elements bound so far to the roles of the pattern.

4. Presenting EJB Design Patterns in Fred

Each EJB pattern discussed earlier needs to be represented as a Fred specialization pattern. A typical Fred pattern is composed of several roles; each role corresponds to one program element (class, method, field…). Few other role types are used to represent certain relationships such as inheritance or alternative use between two roles. Roles may have properties like dependencies on other roles, multiplicity, constraints, and templates.

We will discuss here the Fred specification of the Session Façade Pattern in more detail. Figure 4 presents the roles and properties of the Session Façade Pattern.

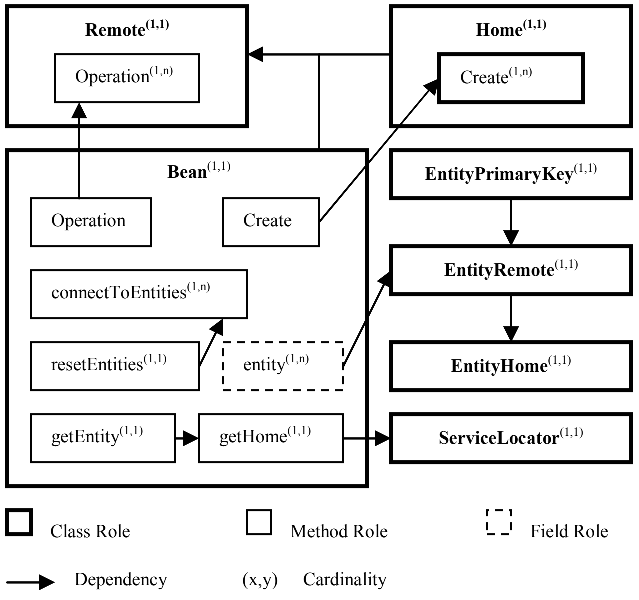
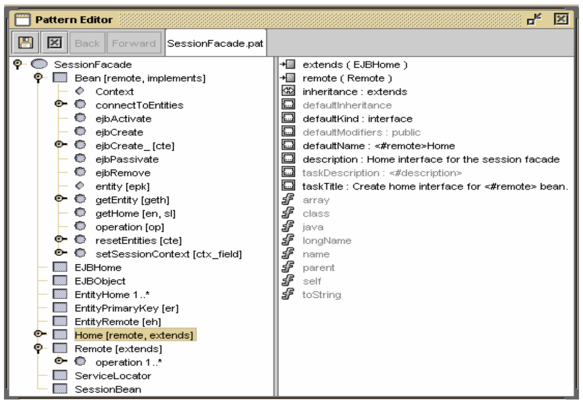


Figure 4. Structure of Session Façade pattern

Figure 4 shows the roles, their multiplicities (in the upper right corner of role names), and the constraints between the roles. Containment constraints are expressed simply by nesting, and other constraints indicating dependencies between roles are denoted by arrows. For readability we have omitted some roles and other details. Roles Remote, Bean, and Home should be bound to the remote interface, bean class and home interface of the Session Façade, respectively. Roles EntityHome, EntityRemote, and EntityPrimaryKey should be bound to home interface, remote interface, and the primary key class (instance of the Primary Key Pattern) of the encapsulated entity bean, respectively. The field bound to entity role holds a remote reference of the encapsulated entity bean. There can be any number of actual fields bound to entity since one Session Façade can encapsulate several entity beans. We can also see that there is a dependency between resetEntities method role and connectToEntities method role. The reason is that resetEntities reconnects the Session Facade to its entity beans by calling connectToEntities.

In Section 2 we showed how the patterns constitute an integrated pattern system. This is reflected in pattern specifications as well. For example, the method bound to Session Façade's getHome role uses the ServiceLocator class role, which belongs to the Service Locator Pattern.

In order to express the pattern structure as a Fred specialization pattern we need to use Fred's pattern editor tool. The pattern editor is opened in Figure 5 for the annotated Session Façade Pattern. The Home role is selected for detailed editing, showing the role editor in the right-hand side pane.



```
SessionFacade                                  extends ( EJBHome )
  Bean [remote, implements]                    remote ( Remote )
    Context                                    inheritance : extends
    connectToEntities                          defaultInheritance
    ejbActivate                                defaultKind : interface
    ejbCreate                                  defaultModifiers : public
    ejbCreate_ [cte]                           defaultName : <#remote>Home
    ejbPassivate                               description : Home interface for the session facade
    ejbRemove                                  taskDescription : <#description>
    entity [epk]                               taskTitle : Create home interface for <#remote> bean.
    getEntity [geth]                           array
    getHome [en, sl]                           class
    operation [op]                             java
    resetEntities [cte]                        longName
    setSessionContext [ctx_field]              name
  EJBHome                                      parent
  EJBObject                                    self
  EntityHome 1..*                              toString
  EntityPrimaryKey [er]
  EntityRemote [eh]
  Home [remote, extends]
  Remote [extends]
    operation 1..*
  ServiceLocator
  SessionBean
```

Figure 5. Fred pattern editor

Instead of going through the pattern specification as it is given using the pattern editor, we will give below a textual specification of the pattern, with some clarifying comments. This representation follows roughly the structure of the XML description generated automatically by the tool, but we have transformed it to a more readable form. This description gives an idea of the kind of information that has to be given as input for Fred. For simplicity, Table 1 shows the textual representation of a small part of the specialization pattern illustrated in Figure 5. Bound roles are common to every pattern instance and should appear in the framework with the specified name. Unbound roles are specific to every pattern instance; this allows the pattern to be used in different contexts. For instance, the unbound role Home, home interface of the session bean, extends the bound role EJBHome, a built-in Java interface that every EJB home interface should extend. Every role has a set of properties; the defaultModifiers property of the getEntity role for example, has the value "private" to say that the method should be private. The definitions of properties may refer to other roles; such references are of the form <#r>, where r is the identification of a role. This is used for producing adaptable textual specialization instructions. For example, the description of getHome method role contains a reference to EntityRemote class role. In constraints, references to other roles imply relationships that must be satisfied by the program elements playing the roles. For example, the type of the field playing the role entity should be the class playing the role EntityRemote. The multiplicity symbol "+" that comes with EntityHome means that there can be more than one program element playing the role EntityHome.

SessionFacade		
Bound roles	Properties	
EJBHome : class	*description*	Java's built-in EJBHome interface
SessionBean : class	*description*	Java's built-in SessionBean interface
EJBObject : class	*description*	Java's built-in EJBObject interface
Unbound roles	Properties	
Remote : class		
operation : method		
exc : Exception	*type*	java("javax.ejb.RemoteException")
EntityRemote: class	*description*	Remote interface of the encapsulated bean
EntityHome + : class	*description*	Home interface of the encapsulated bean
Bean : class		
getHome: method	*description*	Private method to get Home for <#EntityRemote>.
entity : field	*type*	EntityRemote
	defaultInitializer	null
	description	A field that holds the remote reference for the <#EntityRemote> entity bean.
	taskTitle	Provide <#EntityRemote> remote reference
getEntity :method	*defaultImplementation*	<#EntityHome> home = <#Bean.geHome>(); return (<#EntityRemote>) home.findByPrimaryKey(pk);
	defaultModifiers	private
	defaultName	get<%Bean.entity>Entity
	description	Private method to get <#EntityRemote> entity.

Table1. Textual representation of part of the Session Façade pattern

The EJB design patterns discussed in Section 2 are typically deployed in certain combinations. For example, an EJB application often requires several Session Façades. Each façade encapsulates a set of entity beans and for each façade there is a corresponding Business Delegate. To handle this kind of composition of the EJB design patterns, we have applied the *composite pattern* approach used in [5]. Composite patterns are combinations of other patterns. Such highly adaptable (at deployment time) components are important in situations where a similar functionality is needed in several places in an application, with slight variations. Currently our environment distinguishes between the following substructures that serve as either simple or composite patterns:

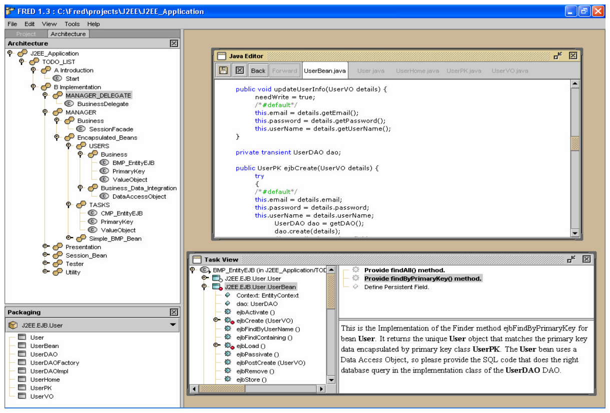
- Session Façade that encapsulates a set of entity beans.
- Business Delegate that forwards client requests to Session Façades.
- Bean managed entity bean that has a primary key class and a Value Object, and uses Data Access Objects to implement the JDBC code
- Bean managed entity bean implemented as a session bean and a Data Access Object
- Container managed entity bean that has a primary key class and a Value Object
- Session bean (stateful and stateless)
- Tester client

A typical scenario is to start creating a set of strongly coupled entity beans that can take any form discussed above, encapsulate them with a façade, and then map a Business Delegate to that Session Façade. If needed, the user could generate a number of session beans by specializing the right component in different ways. In order to encapsulate a different set of entity beans, the composite pattern responsible for the "delegate-façade-entity beans" substructure should be considered once again. The user, with the help of the environment, can integrate custom business logic code with the design patterns. She can define any number of business methods for entity and session beans. The environment ensures that every method defined in the remote and home interface is implemented in the bean implementation class. If the user chooses to implement methods that access data from an Enterprise Information System, the required tasks for using the Common Client Interface (CCI) are generated.

5. A Case Study

We applied the environment produced by Fred to implement the business logic of a simple to-do list application. The application accesses a list of users (administrators and normal users) and associated tasks stored in a relational database. Operations on the lists include adding, updating, deleting, and searching database entries. We created a bean-managed persistence entity bean to encapsulate the table of users and a container-managed persistence entity bean to represent the table of all tasks. A session bean is then used as a façade to both entity beans. The application is implemented according to the architecture proposed by the design patterns of the environment. Since the specifications of the design patterns are associated with various default implementations, a substantial amount of code will be automatically generated during the development process. However, this code is produced in small, understandable pieces to fulfill certain limited tasks.

Figure 6 shows a sample view of the environment at the specialization phase. The Architecture View to the left gives an idea about the overall application architecture. The Session Façade composite pattern Manager for example encapsulates two entity beans. Users composite pattern represents a bean-managed persistent entity bean whereas Tasks composite pattern encapsulates a container-managed entity bean. Users has a business logic part and a business-data integration part. In the left pane of the Task View we can see the list of the tasks that have been carried out by the user. In the other half, the environment suggests several other tasks, finder methods in this case. Besides, an optional task to define other bean persistent fields is shown.



```
public void updateUserInfo(UserVO details) {
        needWrite = true;
        /*#default*/
        this.email = details.getEmail();
        this.password = details.getPassword();
        this.userName = details.getUserName();
}

private transient UserDAO dao;

public UserPK ejbCreate(UserVO details) {
        try
        {
        /*#default*/
        this.email = details.email;
        this.password = details.password;
        this.userName = details.userName;
                UserDAO dao = getDAO();
                dao.create(details);
```

Provide findAll() method.
Provide findByPrimaryKey() method.
Define Persistent Field.

This is the Implementation of the Finder method ejbFindByPrimaryKey for bean **User**. It returns the unique **User** object that matches the primary key data encapsulated by primary key class **UserPK**. The **User** bean uses a Data Access Object, so please provide the SQL code that does the right database query in the implementation class of the **UserDAO** DAO.

Figure 6. Sample EJB application development session

Moreover, adaptive documentation helps in assisting the user with explanations on what tasks should be provided and which missing program elements should be added. This is illustrated in the right bottom pane of the task view in Figure 6: the specializer is asked to provide the findByPrimaryKey method. The documentation explains the user the purpose of the method, its parameter list and return value. It then asks the user to provide the SQL code of the method in the implementation class of the bean's Data Access Object. Note the use of application-specific names (like Users). The integrated Java editor can be used to add custom business logic in addition to viewing and maintaining the code that has been generated. Recall that Fred keeps track of any broken dependencies or inconsistencies between the code and the given architecture.

The environment acts as a tool for compile time checking of an EJB application. For instance, the system checks if for every method declared in the remote interface of a bean, there is a corresponding method definition in the bean implementation class. Hence in a sense the environment acts itself as a Business Interface Pattern, a common practice used to avoid inconsistencies between remote interfaces and bean implementation. This

-pattern assumes an interface that the remote interface extends and the bean class implements. This design pattern is very specific to EJB.

The Value Object specialization pattern can be considered as a factory for Value Objects. Complex applications need to use several custom Value Objects for the same entity bean. For simplicity, the environment we propose uses by default the domain Value Object, a Value Object that has all the persistent fields of the bean.

If the user decides to switch from container-managed persistence to bean-managed persistence for example, the right actions and modifications are generated and displayed to the user in the form of tasks. In this respect the environment supports also an evolutionary mode of work. Another important feature is flexibility; for example, the user is free to choose whether to put the JDBC code in a separate Data Access Object or in the bean implementation class itself. As the business tier gets larger, the proposed architecture makes it easier to understand, control and maintain the system.

Deployment descriptors are text files describing various properties of the beans in XML format, required for the deployment of an EJB application. Generating deployment descriptors can be supported by Fred; the user selects the pattern instance that represents the bean and applies an XSLT transformation using an XSL file that comes together with the environment. Fred parses the pattern instance, extracts the data needed for the bean deployment such as the name of the bean's remote interface, and puts it between the opening and closing tags of the corresponding fields in an XML file.

The environment has been tested against several other similar, simple applications. The experiences showed that the environment can improve the quality of the software and reduce the development effort. The improved quality is a consequence of making desirable EJB solutions available for the application developer and guiding her to use these solutions. For example, using Data Access Objects makes the persistence code of entity beans independent from the used data source, increasing the flexibility of the system design. Using Value Objects reduces the network overhead when accessing persistent fields, optimizing the performance of the application. Much of the work behind coding Enterprise Java Beans was reduced to mouse clicking. Although it is hard to define specific metrics to compare the development process with more conventional ones, it seems obvious that this approach improves the quality of the software and reduces the development effort considerably. The benefits apply both for experienced EJB programmers and novices: for the former, the environment releases the developer from the burden of writing a lot of straightforward but complicated code and remembering all the details of the EJB conventions; for the latter, the environment helps the developer to learn the EJB patterns and successfully produce an application even with limited understanding of EJB. In the next section we discuss the advantages and limitations of the Fred EJB environment in comparison with some other tools.

6. Related Work

Several commercial and non-commercial tools providing support for the EJB technology is being used by the industry. Some tools require users to provide a specific representation of the enterprise beans such as XML files. An example of such products is the realMethods tool [7]. The tool is capable of generating Java code, SQL code and deployment descriptors on the basis of the input XML file that represents the object model. The user can select a set of design patterns, which are used to build the application infrastructure. The advantage of this technique is that once you have your bean specifications ready, generating the equivalent Java code becomes a simple and fast operation.

Other widely used application development environments offer a more standard way of support for EJB technology. An example of such environments is JBuilder [8]. The tool is for example capable of generating code for enterprise beans in a visual mode. It can also construct entity beans out of data models and create relationships between entity beans using drag-and-drop. Changes can be made in both the generated beans code and in the visual tool, keeping the two versions in synchronization. A big advantage of such a tool is the built-in application server where the generated beans could be deployed and tested.

However, the two environments fall short in many features available in the Fred EJB development environment.

- Almost all tools supporting the EJB technology focus on code generation but pay less attention to the design and architecture of the application. Design patterns (as well as other best practices) are used in isolation. Fred environment pays special attention to the architectural aspect of reuse. Fred EJB framework is a collection of collaborating patterns and several other programming rules that could be easily extended and updated.
- Tools that require prior representation of the business logic and data such as the realMethods tool comes at the cost of spending time and effort in preparing correctly the input specifications. Any error in the specifications could lead to serious problems. The maintainability of the system can become as hard as implementing the whole application from scratch. Fred environment gently guides the user in a step-by-step basis to do the job.
- The JBuilder environment does not provide much help to users on what tasks should be done next. The documentation that comes with the environment is rather static and same for all applications. Fred provides adaptive help, documentation are specific to every framework specialization instance.

- Using Fred, the user could build a better understanding of the problem domain. The pattern editor and the architecture view of the framework make it possible to obtain a quick and deep view of the overall structure of the application.
- Fred could automatically detect and locate violations in the design rules as presented by the framework. The environment enforces the cardinalities, the naming and the dependencies of the EJB pattern elements and complains if the contract between the architecture and the developer is broken.

There are few other experimental tools that resemble the pattern-based approach of Fred (see e.g. [6]). In principle, such tools could be used to produce support for EJB application development environment based on a selected set of EJB patterns. However, Fred is unique in its support for fine-grained task-driven development and adaptive on-line guidance. Hence the resulting environment would lose much of the interactive flavor of a Fred-based environment.

7. Conclusions

We demonstrated that it is possible to automatically generate a pattern-based EJB application development environment that provides stronger support than existing EJB environments and removes some of their typical problems. To produce such an environment, a coherent collection of EJB patterns was developed and specified using the notation of the Fred tool. On the basis of such specifications, Fred was effectively turned into an EJB environment. Early experiences with the produced environment confirm our expectations of the benefits of the approach. When compared to conventional EJB environments, the Fred-based ADE allows the generation of large amounts of code without loosing the programmer's control of the code, and the environment can be easily extended with new patterns when needed. However, further evaluations are still necessary to validate our approach.

We are currently integrating Fred with a commercial Java ADE. In this way the rich (but somewhat ad hoc) functionality of a modern ADE can be combined with the more systematic pattern-based software development paradigm of Fred. An even better solution would be to implement the characteristics of the Fred-based ADE directly within a traditional ADE.

There are also some limitations and challenges that were recognized during this work. Some programmers may feel that the task-based wizard limits the freedom of the programmer. In principle this is not true, because the programmer can always resort to the integrated Java-editor and write arbitrary code; however, the benefits of the system would be then lost. Hence the programmer has full freedom to decide which parts of the application are supported by the Fred patterns. The environment could allow more freedom by offering different alternatives on how to implement the design patterns, instead of sticking to one implementation strategy. Another limitation is the strict step-by-step working mode enforced by the tool; this appears to be too tedious in cases where user input is actually not needed. For example, several EJB callback methods come with default empty implementations and could be generated without user involvement. Finally, Fred provides no support for specifying how the method bodies should be written when the default implementation is not sufficient.

So far we have used a fairly limited collection of EJB patterns, and the environment should be extended with other J2EE patterns. The Fred environment could then be used to generate an environment for developing complete J2EE compliant applications. Another important limitation is the lack of support for the new EJB 2.0 specification. Currently the environment does not support message beans, local interfaces and the notion of a query language. Finally, a serious challenge would be to generate custom deployment descriptors for different application servers.

References

[1] Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley 1994.

[2] Alur D., Crupi J., Malks D., Core J2EE Patterns: Best Practices and Design Strategies, Prentice Hall PTR, 2001.

[3] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Annotating Reusable Software Architectures with Specialization Patterns. In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), Amsterdam, August 2001, 171-180.*

[4] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Generating application development environments for Java frameworks. In: *Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE'01), Erfurt, Germany, September 2001, Springer, LNCS2186, 163-176.*

[5] Martin R., Riehle D, Buschmann F., Pattern Languages of Program Design, Addison-Wesley 1998, *163-185.*

[6] Riehle R., Framework Design — A Role Modeling Approach. Ph.D. thesis, ETH Zürich, Institute of Computer Systems, February 2000.

[7] The realMethods: http://www.realmethods.com, April 2002.

[8] Borland JBuilder: http://www.borland.com/jbuilder/, January 2002.