Evolution of a domain-specific application framework for common administrative information systems – An open standards approach –

Clemens Frei (email@cfrei.com)
Thorsten Spitta (thspitta@wiwi.uni-bielefeld.de)

University of Bielefeld
Department of Economics
33501 Bielefeld

Abstract

For many corporations, administrative information systems are the backbone of their business activities. The purpose of this article is to introduce an application framework for small and medium-sized corporations for the problem domain of administrative information systems. The framework has a long and successful history and was implemented on several environments over two decades. The latest version is based on the Java 2 Platform, Enterprise Edition. In this article, we will present the underlying theoretical principles of the framework as well as key sections of the framework's design which reflect them. Furthermore, since the concept of frameworks has a number of appealing benefits, as well as challenges, we will summarize our experiences and address potential problems.

1 Introduction

For many corporations, administrative information systems are the backbone of their business activities. Typically, such software systems are long-lived and under constant adaptation due to changing requirements. As requirements change, certain characteristics of software systems, such as maintainability and extensibility, are critical and ultimately can become a cost factor. The design and implementation of complex software meeting these requirements is difficult, expensive, and error-prone. For a reasonable time now, software architectures, especially implemented as frameworks, are a common response to the challenges of how to address these abovementioned requirements (Morris & Ferguson 1993, Fayad & Schmidt 1997, Szyperski 1999). Frameworks allow “reuse at a larger granularity than classes” (Johnson & Foote 1988).

Struts is a popular framework, based on open standards.[1] We investigated Struts with regard to the general requirements of administrative information systems. According to our experiences (see section 2.6), we consider Struts to be too flexible and in some respects too powerful for our purpose. Since certain design aspects did not comply to our requirements, reusing Struts as a basis for our higher-level framework was ruled out.

We are not content with a *reusable* framework, we want it *to be used* as easily as possible by many software developers in different organizations. This requires a framework which is easy to learn and to use. In order to define a plain, but not primitive, framework, it is necessary to confine its domain (Codenie, Hondt, Steyaert & Vercammen 1997). Nowadays, the majority of proprietarily developed applications in industrial enterprises handles tasks, such as storing data into or retrieving data from databases, aggregating data, and sometimes performing simple calculations in order to obtain derived data. The difficulties of administrative information systems are *not* the algorithms of the programs. The complexity lies in the *data structures* of the underlying conceptual data model and the herewith connected specification as well as the implementation of integrity constraints. This allows a well-standardized flow of control for the majority of such programs. The application domain comprises software that handles all kinds of administrative tasks of front-office as well as back-office activities.

Based on more than 15 years of experience with a framework for such simple administrative systems (section 2), we made an evolutionary redesign of the framework’s architecture (section 3) and the deployment process (section 4). We implemented the framework in several diploma theses and can report first experiences of reuse from developers, who were not engaged in the framework’s development (section 5). At present, we apply this framework in its first productive environment.

Our goal of the latest version – like the successful predecessor in the 1980’s and the 1990’s – was that programmers only implement their data and functional structure and are not bothered with general, application-independent features. The actual architecture defines four tiers as well as provides a set of generic standard functionalities for the application domain. The framework is solely based on open standards, the *Java 2 Platform, Enterprise Edition* (J2EE) being an important constituent. Because the framework’s foremost goal is to be reused easily, its overall complexity and size is manageable (32 classes) and application developers must only implement application-specific logic and thereby are exempt from making critical design decisions.

2 A standard architecture for administrative systems

2.1 Functional requirements

A standard architecture, implemented as a reusable framework, should contain universal and generally accepted features for the entire class of *administrative software*. The key requirements are as follows:

1 For detailed information see http://struts.apache.org

- *unified user interface* which can be customized according to institution-specific standards
- *multi-language* capabilities
- *system-independent* user interface
- administration of application-specific *access rules* (security profiles)
- dialogue *navigation* between masks of the current and other applications, realized with the framework
- user-driven *batch start* of asynchronous tasks
- *help* for applications, masks, or fields
- *error handling*
- *message handling* and maintenance. Messages should be differentiated into system- and application-specific ones. Thereby messages can contain texts that range from a single line up to several paragraphs in a separate window.
- *feedback* or messages of users *to the system administrator* in order to improve maintenance services.

A more detailed functional requirements specification can be found in Spitta (1989), p.185ff.

2.2 Organizational requirements

In order to succeed with the goal to establish an accepted framework that is *used* by developers, we further define a number of organizational requirements as follows:

- *comprehensible documentation* which teaches the framework's purpose, how to use it, as well as design details. There are details, the developer *must* know in order to develop applications and such he might only be interested in. We also think that a successful and accepted framework must be open source which fosters transparency and a climate of openness. Especially programmers tend to be suspicious against black boxes.
- *ease of use* and framework learning for regular programmers in reasonable time.
- *generator tools* for evolutionary steps of the framework. These tools should support the development process in order to avoid copies of older programs.

2.3 Module requirements

Other requirements are not visible to the user, but are the basis for the developer's work. We call them *module requirements*. The majority of the *application modules* which support the process of storing data have a structure that can be standardized, as illustrated in figure 1. The picture has two dimensions, *actors* (System and User), and *layers* (Presentation, Application, Data Access). Additionally, the diagram is enriched by method names (`processBeforeOutput`, `processAfterInput`, and `checkDataIntegrity`) corresponding to the activities.[2]

From the viewpoint of *one* module, the fourth layer of today's systems, *Control*, is not relevant. This layer is only visible, if the user navigates *between* modules.

2 The method names `processBeforeOutput()` and `processAfterInput()` are inspired by equal names used in SAP's R/3.

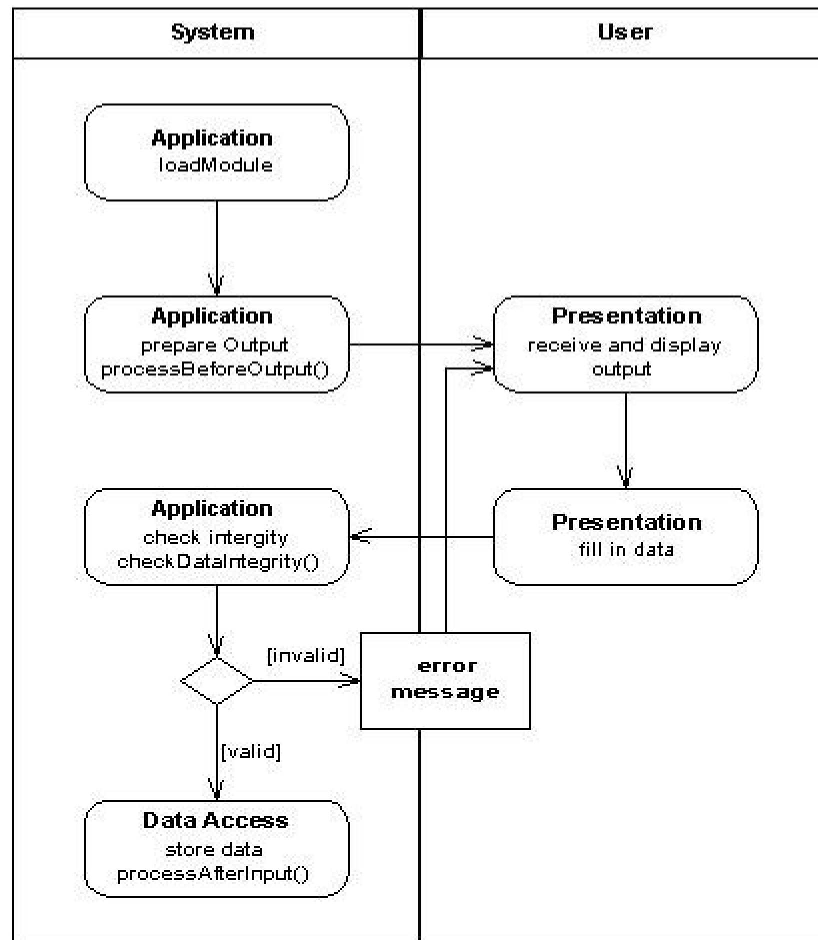

Figure 1: Activity diagram of the process to store data

The case of *information modules*, without changing the data base, has a comparable structure. The activity to `check integrity` is substituted by an activity `calculate` which might create derived data. Then, the activity `prepare output` is restricted to readonly data access.

2.4 Historical evolution of the framework

Most of the abovementioned requirements were fulfilled by a framework that was designed and implemented 1985 in the Schering AG, Berlin. The framework was build as a basis for a very large reporting system (*BW-Project*[3]), which should help to consolidate the rapidly grown enterprise, consisting of 140 single companies. The total estimated effort of all projects for that system within four years (1982-1986) added up to 300 man-years. A lot of functions were covered by standard applications of SAP R/2. However, there still remained much work left for proprietary development. Aside the backbone of the operative systems, proprietarily developed systems should be implemented efficiently as well as sustainably. Maintenance effort over time should be mastered by strict standardization.

Instead of implementing some program skeletons as standardized help – a common practice today as well as at that time – a blueprint of an architecture was outlined and implemented as a framework in a first version in NATURAL, a 4GL[4] (see Mönckemeyer & Spitta 1983). A second framework was implemented in the language RPG[5] III, at that time the only supported language on IBM AS/400. All frameworks were based on that blueprint, which was published some years later (Spitta 1989, ch.11).

3 BW = Berichtswesen (german), reporting system
4 fourth generation language
5 report program generator

As part of the BW-Project, about 20 applications originated from the NATURAL and four from the RGP-framework. An application consisted of between 10 and about 100 modules, on average about 50. Each module contained one to three separately compilable programs with an average over all applications of two programs. In total, only from the abovementioned BW-Project, this results in about 4,800 programs, most of them maintained for 15 years.

The framework was sold to several other firms, one of them also consolidating its whole information system. In this medium-sized enterprise, applications added up to entire operative information systems. SAP R/3, which was partly also implemented, revealed too many drawbacks in the case of mass production (750,000 stockings per day). The self-developed part comprised route planning, production control, inventory management as well as sales and distribution management, based on a product data administration for parts which contain many variants (Spitta 1993).

In conclusion, this means that the frameworks, based on a standard architecture, were successfully used and applied by about 100 programmers independently from one another. Many of the systems of that time are still in operation and have proven their sound maintainability. As main factors for the success of the framework can be seen:

- *well-documented* tools (a framework *is* a tool with modules, templates, code generators, and a dictionary)
- *strict standardization* of the vast majority of interactive programs (see figure 1)
- *concentration* of the programmer's work on the application itself
- *reduction* and focus of the framework on the domain's essentials

This approach coped with 95% of the work to be done in most of the projects. The framework supported the development of small and medium-sized administrative information systems. Applications built with it were server-centric and had a three-tier architecture. All applications consisted of one or more interacting application modules which used a number of system modules. System modules provide the framework's functionalities (see section 2.1) to simplify the development of application modules. Thus, the task for application developers was reduced to coding application-specific logic in order to build application modules. The framework enforced a uniform structure and design of all modules and applications which enabled all modules to interact with each other.

However, time and technology have changed and one severe drawback gradually turned into a burden. The time of vendor-specific environments like COBOL+CICS, NATURAL, or RPG III was over and the time of the centralized client-server-model as well. As a result, we redesigned the architecture for a fourth tier and implemented several prototypes (Grasmugg 2000, Mersch 2002, Frei 2005). They are based on a new, open technology for distributed systems which is now *Java 2 Platform, Enterprise Edition* (J2EE).

2.5 New Requirements

In the process of reviewing the first prototype (Grasmugg 2000) and preparing a redesign of the framework, we further specified a number of requirements, as well as introduced new ones:

- extend the three-tier to a four-tier architecture by incorporating a separate control layer
- application modules as well as entire applications should potentially be distributable (but also centralizable)
- the framework must be based on open standards
- the user interface should be a web interface, preferably rather a thin client than a fat client

As a result, we chose the *Java 2 Platform, Enterprise Edition* (Sun Microsystems, Inc. 2003*b*) as the basis for the current implementations of the framework.

2.6 Other J2EE frameworks – Struts

Before building versions of prototypes, we had to examine whether there already existed other open source J2EE application framework solutions we could reuse or use as a basis for our framework. A popular example for building J2EE web applications in general, based on the Model-View-Controller (MVC) design paradigm, is the *Struts* framework of *the Apache Software Foundation*[6] .

By comparing the purpose and scope of the Struts framework and the framework subject to this paper, it became apparent that both frameworks exhibit fundamental differences. According to the official website of Struts, it is an open source framework for building Java web applications, whereas we had (see section 2.4) and need a specialized and higher-level framework for building applications in the problem domain of *administrative information systems* which *can* have a web interface. Thus, the problem domain of Struts is broader.

However, this might raise the considerable question, why not reuse Struts as a basis for a more specialized framework? – In general, this is an approach which should be considered and evaluated, due to the development effort of frameworks. In the case of this particular framework, we decided against reusing another framework, such as Struts, and built this framework from scratch, based on our good experiences with a proven architecture. Subsequently, we will illustrate some considerations in regard to Struts.

Although our framework makes extensive use of J2EE technologies, it also intends to be open for using other technologies as well. The default implementation suggests and supports the use of JavaServer Pages (JSP), HTML, and JavaServlets to realize a web user interface. However, other realizations comprise Java application clients, PHP solutions, and even terminals must not be ruled out due to a too restrictive and specialized design of the control layer. Thus, the controller component itself is entirely independent from the presentation layer and the technology used to implement thereof. The implementation of the control layer which achieves this goal is described in section 3.2.

On the other hand, Struts supports the development of *web* applications, thereby enabling the use of technologies, such as "JSP, including JSTL and JSF, as well as Velocity Templates, and XSLT" (Struts' official web site). According to our experiences, Struts is less flexible in regard to applying non-J2EE technologies. The restrictive experiences with a controller component implemented as a JavaServlet in the aformentioned matter, has also been implemented in the process of developing earlier prototypes of this framework and led to revision and redesign of the control layer (Mersch 2002).

Thus, we can conclude that the core of the framework subject to this paper, is the application layer which uses enterprise beans, and a controller component, also implemented as an enterprise bean. Such an implementation of a controller component that resides in the EJB tier is sometimes referred to as *EJB-tier controller* (Singh, Stearns & Johnson 2002). This will be demonstrated in more detail in the next section.

3 Key design aspects and decisions

In this section, we describe some key features according to their tier in the architecture. For this purpose, we use design patterns to identify and describe critical sections of the framework, since they "provide a common vocabulary for design"(Gamma, Helm, Johnson & Vlissides 1993).

3.1 Presentation layer

The default implementation suggests JavaServer Pages (JSP) in combination with HTML for realizing the presentation layer. The user accesses his user interface through a web browser. Typically, the graphical elements provided by HTML are sufficient concerning the requirements of user interfaces for administrative information systems. The framework proposes and supports the use of the *Composite View* pattern (Alur, Crupi & Malks 2001) which itself is based on the classical

6 For detailed information see http://struts.apache.org

Composite design pattern (Gamma, Helm, Johnson & Vlissides 2001, p.163).

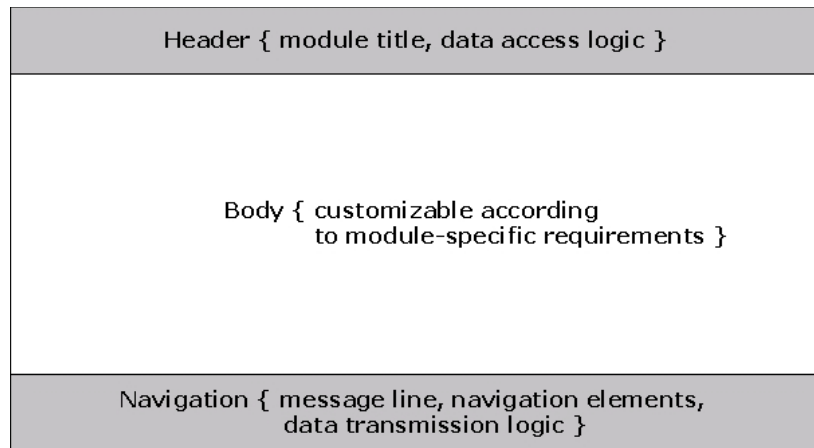

Figure 2: The composition of subview components to an entire view

As depicted in figure 2, the entire view component for each module is composed of multiple subviews. The framework supports this pattern by providing two template files, referred to in the graph as “Header” and “Navigation”. The framework reuser must only design and implement the body of each view according to the specific requirements of each module. Among other aspects, this modular structure reduces the development effort for view components. Furthermore, subview components can be reused for building the view component of each module which thus leads to an ease of maintenance for applications and avoids redundancies.

3.2 Control layer

The *Controller* encapsulates the control logic of the entire system and represents the state of each module. The Controller acts as a mediator between the presentation and the application layer. To decouple the Controller from the presentation layer and thus to provide flexibility of the technology to be used to realize the user interface, the Controller consists of two parts. The integral part, the actual Controller, is implemented as a stateful session bean and is independent from the technique used to realize the presentation layer. The second part, called the *ControllerAdapter*, is the realization of the *Adapter* design pattern (Gamma, Helm, Johnson & Vlissides 2001, p.139) and connects the Controller to the view component. Technically, the ControllerAdapter can be assigned to the presentation layer. The framework’s default implementation of the ControllerAdapter is a Java Servlet which enables the use of JSP and HTML to implement the graphical user interface. If this default implementation is an appropriate solution, neither the Controller, nor the ControllerAdapter require customization and can be interpreted by the application developer as a black-box. However, if the user interface is to be implemented using a different technology, such as PHP, only the ControllerAdapter must be replaced according to the new technology, whereas the actual Controller is exempt from such a change.

3.3 Application layer

The application layer is clearly the fundamental layer for an application developer reusing this framework. This layer consists of modules, whereas each module is implemented as an enterprise bean, i.e. a stateful session bean. In this section, we will address some aspects in regard to enterprise beans, such as the scalability of applications and potential performance limitations. Furthermore, we will briefly describe important design decisions, as well as the corresponding underlying principles which are reflected by this design.

The design of this layer aims at restricting the degrees of freedom for developers as such that these restrictions result in a uniform structure of all application modules. Furthermore, the framework provides generic functionalities for the application domain (see section 2.1). Our

framework intends to lower the barriers of entry for novices to the EJB technology, as well as allows a *separation of concerns* (Dijkstra 1976) among developers. Subsequently, we describe how these goal are achieved.

Each enterprise bean requires two interfaces that comply to the EJB-Specification (Sun Microsystems, Inc. 2003*a*). Our framework provides both interfaces for all modules in a standardized manner which ensure a correct interaction with the controller. Furthermore, the EJB-Specification (Sun Microsystems, Inc. 2003*a*) requires a number of mandatory methods.[7] These methods are declared in an abstract superclass of the actual module classes. Therefore, the source code of module classes does not contain any EJB-specific logic, and coding module classes only requires knowledge about the framework and the problem domain, whereas the creation and configuration of enterprise beans can be delegated to other developers familiar with the EJB technology (see also section 4).

Since *ease of use* is one important goal of our framework, the relevant section for application developers to implement application modules is manageable (less than 10 classes) and the design relies on proven design patterns. Both aspects also contribute to fast framework learning. Thus, the framework-specific knowledge required to code application modules is reduced to understanding a class hierarchy consisting of two abstract classes as well as a few interfaces in regard to implementing data integrity constraints and exchanging data with the framework.

Frameworks capture the control flow of applications, often referred to as *inversion of control* (Johnson & Foote 1988, Gamma, Helm, Johnson & Vlissides 2001). In general, understanding the dynamic behavior of a framework can be challenging. However, for our framework, figure 1 sufficiently describes the dynamic interactions that developers must know in order to implement application modules. Subsequently, we illustrate the mechanism that mainly enforces the uniform structure of all application modules and ensures a correct interaction with the Controller. For the sake of simplicity, we concentrate on the *structure* of application modules and abstain from the logic of how to incorporate the framework's functionalities.[8]

The template method design pattern as an integral constituent

The *template method* design pattern (Gamma, Helm, Johnson & Vlissides 2001, p.325) is a common element in object-oriented white-box application frameworks (Fayad & Schmidt 1997) and an integral part for the creation of application modules in our framework. Figure 3 illustrates in a UML class diagram a fragment of the framework's abovementioned class hierarchy which resembles the template method design pattern and complements to the dynamic behavior, illustrated in figure 1.

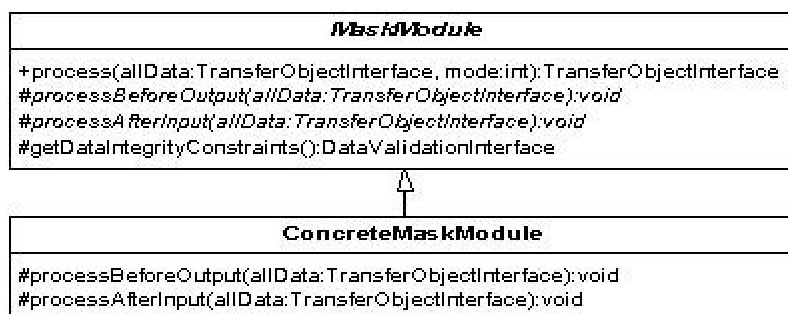

Figure 3: The template method design pattern in this framework

The method `process()` is referred to as *template method* (Gamma, Helm, Johnson &

7 These methods are declared in the interface javax.ejb.SessionBean and comprise the methods ejbActivate(), ejbPassivate(), ejbRemove(), and setSessionContext() (Sun Microsystems, Inc. 2003*a*, p.79). Furthermore, each stateful session bean class must have at least one ejbCreate()-method (Sun Microsystems, Inc. 2003*a*, p.82).

8 The direct superclass to the class `MaskModule`, depicted in figure 3, contains the methods providing the framework's functionalities as well as the mandatory EJB-specific methods. Thus, the framework's functionalities are made available by white-box reuse.

Vlissides 2001) which manages the control flow. It is not solely responsible for the control flow of a module, but is dependent on the state of the module or on the action executed by the user it receives from the Controller. The remaining methods can be distinguished according to whether they *must* be implemented by an subclass or whether it is *optional* to do so.

The methods `processBeforeOutput()` and `processAfterInput()` are declared abstract and must be implemented in a subclass. They are executed immediately before and after the corresponding view is displayed to the user (see also figure 1). Therefore, application-specific logic can be realized in these two methods. Alternatively, if the application logic is organized in a separate class library, the aforementioned methods can also contain the appropriate method calls of external sources.

Important for this class of software is to ensure data integrity. Thus, our framework provides a customizable mechanism to support the implementation of data integrity constraints. The method `getDataIntegrityConstraints()` represents a default implementation which can be overridden if the application module requires data integrity constraints. Such methods are often referred to as *hook methods* (Gamma, Helm, Johnson & Vlissides 2001). All data integrity constraints must be implemented in a separate class and comply to the contract defined in the interface `DataValidationInterface`. The framework's method `checkDataIntegrity()` (see figure 1) invokes this validation mechanism.

In conclusion, the development effort for each module is reduced to coding application-specific logic or method calls thereof in a module class whose structure is bound by an abstract superclass. Module classes are exempt from EJB-specific logic. However, each module must be configured in the deployment descriptor.

Inheritance-based reuse for creating modules

Creating new modules is based on inheritance which identifies this part of the framework as a white-box (part of the overall) framework. Since white-box frameworks and inheritance in general are not uncritical, we will address some objections in reference to our framework. Johnson & Foote (1988) note that due to the number of subclasses which need to be created in order to reuse a white-box framework, the task for a new programmer to learn the design of the application can become difficult. Furthermore, they remark that learning to use a framework is the same than learning its design (Johnson & Foote 1988). Both problems cannot be dismissed, but alleviated due to the size and complexity of a framework. Our framework consists of 32 classes in 5 packages, and subclassing from an abstract class is only relevant for a framework reuser by creating application modules. For this purpose, only a class hierarchy comprising two abstract classes of 32 existing ones must be understood.

After all, each module is also an enterprise bean...

As mentioned above, each module is in fact a stateful session bean. Although the EJB technology is not uncritical for a number of reasons (Schätzle, Seifert & Kleine-Gung 2002), the current version of the framework relies on EJB, since EJB containers provide a number of useful low-level services (Monson-Haefel 2002, p.69ff). The framework provides two sets of interfaces, each contains a standardized home and component interface that differ in the programming model (local or remote) they support. Reusing either set of interfaces ensures the correct interaction with the Controller. However, the application developer must make the design decision which programming model should be applied. For the purpose of providing location independence and flexibility with regard to the distribution of application modules, the remote programming model should be used. This means that both interfaces are Java RMI interfaces (Sun Microsystems, Inc. 2003*a*, p.54f). However, there is a trade-off between location independence and flexibility on the one hand and potential performance limitations due to remote method invocations on the other hand (Singh,

Stearns & Johnson 2002, p.140). Ceteris paribus, a local client view which is not location independent dominates a remote client view in regard to performance (Schätzle, Seifert & Kleine-Gung 2002, p.222), however, this cannot be generalized entirely, since the distribution of components among different servers might improve the application's overall performance (Sun Microsystems, Inc. 2004, p.869).

3.4 Persistence layer

The framework does not make assumptions regarding the use of a particular persistent storage mechanism or the access thereof. Due to the number of persistent storage mechanisms, such as relational databases, object-oriented databases, and flat files, the decision of which mechanism and access strategy to choose, or how to integrate legacy database systems is ultimately application-specific and beyond the scope of this framework and article. In the context of our framework, modules, i.e. stateful session beans, need to store or retrieve data from a persistent storage mechanism. We will focus on relational database management systems (RDBMS) and briefly refer to a few aspects in regard to the access of persistent storage mechanisms. Typically, J2EE applications use entity beans to represent persistent data (Alur, Crupi & Malks 2001). Entity beans are distinguished in regard to how their persistence is handled between entity beans with bean-managed (BMP) and container-managed persistence (CMP). Using entity beans with container-managed persistence means "that the EJB container handles all database access required by the entity bean" (Sun Microsystems, Inc. 2004, p.861).

The *Data Access Object* pattern (Alur, Crupi & Malks 2001) defines a strategy which can be applied if the container does not handle the access of a persistent storage mechanism. Thus, an object, called the *Data Access Object* abstracts and encapsulates all access to the persistent storage mechanism and hides the data source implementation from its clients (Alur, Crupi & Malks 2001). Related patterns are *Transfer Object* (Alur, Crupi & Malks 2001), *Factory Method* (Gamma, Helm, Johnson & Vlissides 2001, p.107), and *Abstract Factory* (Gamma, Helm, Johnson & Vlissides 2001, p.87).

Although we have successfully applied the implementation of a data access layer, based on the aforementioned design patterns, we can also support the argument made by Alur, Crupi & Malks (2001) that it is worth considering to use third-party tools if the requirements for a Data Access Object are sufficiently complex. There are a number of third-party tools that provide object-to-relational mapping for RDBMS and can also be used in combination with this framework. Those products include open source solutions, such as *Hibernate* and *iBatis*.[9] Using and integrating such a solution into a framework also reinforces the paradigm to rely on proven solutions, thereby significantly decreasing the overall development effort.

9 For more information see http://www.hibernate.org and http://www.ibatis.com

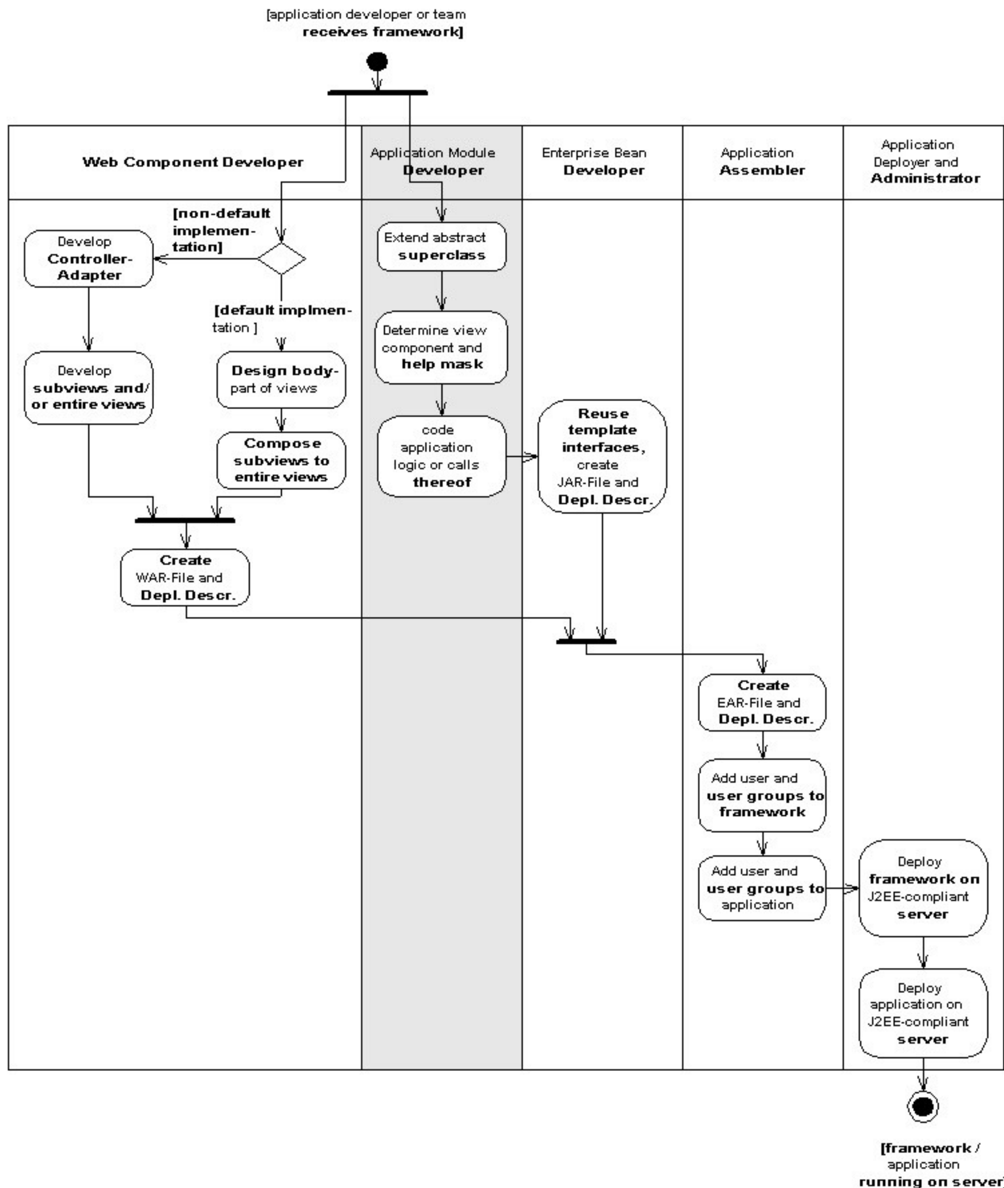

Figure 4: Activity diagram of the development process

4 Development and deployment process

Figure 4 shows a UML activity diagram which outlines the process how to reuse the framework in order to build applications with it. Each swim lane represents a different role developers perform. However, these roles are not always executed by different people. This categorization of roles is based on (Sun Microsystems, Inc. 2003*b*, p.15ff). Additionally, we incorporated the role of an *Application Module Developer*, since no expertise in the EJB technology is required to code the application module class; only knowledge about the framework and the application domain is mandatory to code modules (see section 3.3). Thus, the EJB-specific aspects of a module are handled by developers performing the role of an *Enterprise Bean Developer* who delivers an EJB JAR file containing the enterprise beans. The *Web Component Developer* develops the user interface of an application and encapsulates it in one or more WAR files. *Application Assemblers* combine these application components into an J2EE application (EAR file). For the sake of simplicity in this diagram, we combined the roles of an *Deployer* and *System Administrator* (Sun Microsystems, Inc. 2003*b*) to the single role of an *Application Deployer and Administrator*. The tasks of this role comprise deploying applications and the framework on a J2EE-compliant application server, as well as maintaining this environment during the productive use.

As mentioned in section 2.2, a comprehensible documentation of a framework is a critical success factor. Johnson proposes to use *patterns* as an effective documentation technique. Thereby, patterns are a “structured essay” (Johnson 1992), consisting of a number of patterns, where each pattern describes a solution to a particular problem and each pattern leads into the next.

We think that an UML activity diagram like figure 4 can act as kind of a roadmap for the development and deployment process of applications and lead through this process. The idea of a roadmap is related to the idea expressed in Johnson’s patterns.[10] Thus, we think that such an UML diagram, describing the development process, can lead through such a set of patterns or a documentation in another representation. Since the activities, referred to in figure 4, are rather coarse-grained, each activity could thereby be refined by a number of patterns. Each pattern could use further documentation techniques, as well as references to corresponding sections in sample applications.

A weakness of this documentation technique is that it is based on the assumption that a person who creates the documentation is able “to anticipate future uses of the framework adequately to provide enough patterns in sufficient detail” (Shull, Lanubile & Basili 2000). However, we think and know from our experiences with the documentation of the NATURAL framework (Mönckemeyer 1988) that for a framework like this such a set of patterns might be an effective structure for documenting a framework. Primarily this is due to the fact that the framework defines rather restrictive and binding constraints, as well as provides a clearly defined and manageable set of functionalities which we think can be covered by a reasonable number of patterns.

5 Experiences and concluding remarks

Although frameworks undoubtedly have a number of appealing benefits, they are neither a panacea nor “free-lunch”. We will subsequently summarize few experiences made during these development processes. At first, we will address a few aspects, benefits as well as obstacles, the literature, such as Fayad & Schmidt (1997) and Johnson (1997), mentions in regard to object-oriented frameworks.

The development effort of frameworks is higher compared to “normal” applications (Fayad & Schmidt 1997). Thus, an investment in developing a framework cannot be justified by a one-time reuse of the framework, but amortizes only over a number of projects in which the framework is reused. Also, validating and debugging applications built using frameworks can be difficult, for instance, since “it is usually hard to distinguish bugs in the framework from bugs in the application code” (Fayad & Schmidt 1997).

As mentioned in previous sections, applications based on the same framework typically exhibit a uniform structure. This reduces maintenance costs, since “maintenance programmers can move from one application to the next without having to learn a new design” (Johnson 1997). Also, such a uniform structure of different applications built with this framework fosters the communication among developers. As a result, developers are more productive.

Another important aspect is that frameworks have a large learning curve (Fayad & Schmidt 1997), i.e. it is time-consuming and takes considerable effort to learn how to reuse a framework. Understanding a framework is essential for reusing it, thus, the documentation of frameworks is crucial to the success of frameworks in general, and even more for large-scale frameworks (Fayad & Schmidt 1997). A reuse project with this framework (Frei 2005) when the documentation was incomplete and insufficient respectively indicated that the framework’s characteristics alleviated the importance of an appropriate documentation. To a large extent, this can be attributed to the simplicity of the framework in general, and to the manageable size of the section relevant for a framework reuser. As mentioned, the relevant section of the framework for a application developer contains less than ten classes of 32 classes for the entire framework and class hierarchies do not exceed derivations from two abstract classes. However, we can strongly support the argument often mentioned in the literature that examples are important to learn how to reuse a framework (Johnson

10 The first pattern of Johnson’s patterns documents the purpose of a framework, introduces the other patterns, and determines the sequence in which these patterns should be applied.

1992, Shull, Lanubile & Basili 2000), although they do not explicitly show how certain features of the framework are provided (Shull, Lanubile & Basili 2000).

We conclude that, once it was understood how to reuse the framework, it was possible to rapidly develop prototypical applications with this framework (Echterhoff 2002, Frei 2005). Thereby, the design of the framework which significantly reduces the degrees of freedom for application developers was perceived as being helpful and supporting, instead of constraining. The relevant sections of the framework's design and its functionalities could be learned within a reasonable period of time. Furthermore, the set of functionalities could be applied easily and proved to be appropriate for this application domain. Thus, we think that mainly the manageable size and the design of this framework, i.e. the limited degrees of freedom for application developers, as well as the clearly and well-defined set of functionalities contributed to successful reuse of this framework.

References

1. Alur, D., Crupi, J. & Malks, D. (2001), Core J2EE Pattern – Best Practices and Design *Strategies*, Prentice Hall / Sun Microsystems Press, Upper Saddle River, NJ.

Codenie, W., Hondt, K. D., Steyaert, P. & Vercammen, A. (1997), 'From custom applications to domain-specific frameworks', *Communications of the ACM* **40**(10), 70–77.

Dijkstra, E. W. (1976), *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ.

Echterhoff, D. (2002), Java und Wiederverwendung – Review und Verbesserung einer Fallstudie, Diploma Thesis, Universität Bielefeld.

Fayad, M. & Schmidt, D. (1997), 'Object-oriented application frameworks', *Communications of the ACM* **40**(10), 32–38.

Frei, C. (2005), Über die Weitergabe von Wissen bei der Software-Entwicklung – Eine Fallstudie, Diploma Thesis, Universität Bielefeld.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1993), 'Design patterns: Abstraction and reuse of object-oriented design', *Lecture Notes in Computer Science* **707**, 406–431.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (2001), *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Grasmugg, S. (2000), Entwurf und Implementierung eines Standarddialogwerkzeugs für administrative Softwaresysteme in Java 2, Diploma Thesis, Universität Bielefeld.

Johnson, R. E. & Foote, B. (1988), 'Designing reuseable classes', *Journal of Object-Oriented Programming* **1**(2), 22–35.

Johnson, R. E. (1992), 'Documenting frameworks using patterns', *SIGPLAN Not.* **27**(10), 63–76.

Johnson, R. E. (1997), 'Frameworks = (components + patterns)', *Communications of the ACM* **40**(10), 39–42.

Mersch, D. (2002), Java und Software-Architektur – Review und Verbesserung einer Muster-Architektur, Diploma Thesis, Universität Bielefeld.

Mönckemeyer, M. & Spitta, T. (1983), Concept and experiences of prototyping in a software-engineering-environment with natural, *in* 'Proceedings of the Working Conference on Prototyping-Namur', Springer.

Mönckemeyer, M. (1988), *SPASS – TEBAS 16.0*, Schering AG, Berlin/Bergkamen.

Monson-Haefel, R. (2002), *Enterprise JavaBeans*, 2 edn, O'Reilly, Köln.

Morris, C. & Ferguson, C. (1993), 'How architecture wins technology wars', *Harvard Business Review* **71**(2), 86–96.

Schätzle, R., Seifert, T. & Kleine-Gung, J. (2002), 'Enterprise JavaBeans – Kritische Betrachtungen zu einer modernen Software-Architektur', *Wirtschaftsinformatik* **3**(44), 217–224.

Shull, F., Lanubile, F. & Basili, V. (2000), 'Investigating reading techniques for object-oriented framework learning', *IEEE Transactions on Software Engineering* **26**(11), 1101–1118.

Singh, I., Stearns, B. & Johnson, M. (2002), *Designing enterprise applications with the J2EE platform*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Spitta, T. (1989), *Software Engineering und Prototyping*, Springer, Berlin, Heidelberg.

Spitta, T. (1993), Sechs Jahre Anwendungsentwicklung mit Prototyping, *in* 'Requirements Engineering '93:', Teubner, pp. 49–66.

Sun Microsystems, Inc. (2003*a*), *Enterprise JavaBeans Specification, Version 2.1*. Available from: http://java.sun.com/products/ejb/docs.html.

Sun Microsystems, Inc. (2003*b*), *Java 2 Platform Enterprise Edition Specification, v1.4*. Available from: http://java.sun.com/j2ee/download.html.

Sun Microsystems, Inc. (2004), *The J2EE 1.4 Tutorial*. Available from: http://java.sun.com/j2ee/download.html.

Szyperski, C. (1999), *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Wrede, S. (2002), *Das Standard Dialog Werkzeug – User Manual*, Universität Bielefeld.