A Pattern-Based J2EE Application Development Environment

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

J2EE (Java 2 Platform, Enterprise Edition) is Java's platform for building distributed enterprise applications. The platform takes advantage of a wide range of new and evolving technologies and has been enriched by proven design solutions. These solutions are formulated and documented in what is known as J2EE design patterns. Rather than applying the patterns in isolation, a complete design system can be composed of the patterns. This pattern-based system can significantly ease the development process of J2EE applications and improve the quality of the produced software. In this paper, we will show how a general architectural tool (Fred) can be used to model such a pattern system and to generate an architecture-centric environment for developing J2EE applications. The environment is a task-based wizard that guides the user through the development of the application by enforcing certain design rules.

1 Introduction

J2EE (Java 2 Platform, Enterprise Edition) is a component-based and platform-independent architecture for building enterprise applications. This architecture offers a multitiered distributed application model. Each tier is usually implemented by a different group of developers and communicates with the other tiers via a standardized interface. The advantages of n-tiered architecture are promoting software reusability, easier system maintenance and more effective use of data and networks. Most web-based enterprise applications are split into three logical tiers, Presentation Tier, Business Logic Tier, and Enterprise Information System Tier. J2EE provides standard solutions to each of these tiers.

Using J2EE, the presentation tier, which aims at presenting the business information to the user, is implemented using Servlets, JSPs and HTML/WML pages. The business tier, where core business mechanisms are implemented, is usually encapsulated in EJBs (Enterprise Java Beans). The enterprise information system tier, which represents different kinds of legacy systems, database servers, etc is usually accessed through the JDBC API and other standard interfaces provided by the J2EE Connector Architecture.

J2EE application development is still an evolving art. In order to share good design solutions, proven design and programming techniques are available and shared by J2EE developers. These solutions are known as *J2EE design patterns*. However, because the technology is moving so quickly, designers and developers are struggling to understand and apply the architecture. Recognizing the need, several commercial and non-commercial software products with J2EE tool support have been released and adopted by the industry. These systems offer a development environment for J2EE applications, providing assistance in generating code, testing the applications and deploying them within the environment. Such environments are called Application Development Environments (ADE).

Even though these ADEs provide substantial help for system designers and application developers, they open a new set of problems: the produced code hides the architectural aspects and the design decisions that have been taken, developers need to examine the generated code in order to maintain it. Another problem is that these development environments can be seen as close systems, making it hard to adapt them to new design

decisions, which contradicts with the evolving nature of J2EE patterns. In this work we show how a general architectural tool called Fred ("Framework Editor", [Hak01a], [Hak01b]) can be used to generate an architecture-centric task-based environment for developing J2EE applications. The discussed environment overcomes the mentioned problems of conventional ADE's.

The remaining of the paper is organized as follows. In the next section we discuss the main features of the J2EE patterns used in this work, and how they are organized as a small pattern system. An overview of the Fred J2EE environment and a small case study is presented in Section 3. In Section 4 we compare the work to other existing solutions. Finally, in Section 5 conclusions are drawn and possible future work is highlighted. The EJB part of the work has been previously discussed in [HaK02].

2 A J2EE Pattern System

In this section, we will just give a brief overview of the patterns we modelled using Fred to generate a J2EE ADE. More details about the patterns can be found in [ACM01]. We show how to present these cooperating patterns as an integrated pattern system that can be used by J2EE developers to cover central parts of the design.

2.1 Business-Tier Patterns

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.

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.

2.2 Integration-Tier Patterns

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

2.3 Presentation-Tier Patterns

Front Controller

View navigation is a key issue in web-based enterprise applications. Because views usually share common logic, a centralized access point for view navigation can be introduced in order to remove code duplication and improve view manageability. This pattern controls and coordinates the processing code across multiple requests. It centralizes the decision with respect how to retrieve and process the requests. A common strategy to implement this pattern is to use command pattern [Gam94].

Intercepting Filter

Requests and responses need sometimes to be processed before being passed to handlers and other clients. An example of request processing is form validation, user authentication, or data compression. The solution is to create pluggable filters to process common logic without requiring changes to core request processing which improves code reusability and decouples request handlers. Servlet specification version 2.3 comes with a standard Filter approach that can be used to apply this kind of processin g.

Dispatcher View

The system needs to control the flow of execution and the navigation between views. In particular, the system needs to know to which view to dispatch next based on the request. It is vital to separate the logic on deciding which view comes next from the view components themselves. The pattern does not perform heavy processing on the request but can be seen as a simple forwarding facility.

View Helper

In J2EE applications a view is used for content presentation, which may require the processing of dynamic business data. Business and system logic should not be placed within views. Encapsulating business logic in a helper instead of a view makes the application more modular and promotes code reuse. View Helper Pattern enforces the use of only presentation format inside the view and not the processing required to get the presentation content.

In addition to the standard J2EE 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 behaviour is observed through console output.

2.4 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 J2EE application development. Figure 1 shows a pattern system for J2EE applications. The proposed system is a modified version of the pattern system used in [ACM01]. The modifications have been considered in order to make the system easier to implement and use. For instance, each entity bean is associated with a primary key class generated by the Primary Key pattern. This practise is required in our environment whereas it is an optional feature in EJB technology. Knowing the primary key class allows the environment to generate more code, for example the type of the parameter passed to the `findByPrimaryKey` method can be deduced by the system. The same information can also be used when generating deployment descriptors for the entity beans. The architecture in Figure 1 defines an entity bean, a primary key, a value object, and a possible data access object (in case of bean managed persistence) as one block. This structure is usually used recurrently in the same application, thus it should be defined as a separate substructure. Imposing this kind of organization enhances the architecture of the generated application, which in turn improves the performance of the application and promotes its maintainability and portability as we have seen in the previous section. In addition, a Tester pattern has been added to the pattern system in order to individually check the generated beans.

The presentation tier is implemented around the MVC pattern [Gam94]. Controller, and business code is separated from the dynamic content and the view components. Client requests can be pre-processed and post-processed using intercepting filters. Front controllers further process the request by executing the right command, which in turn executes the associated task on the business-logic tier. The user is dispatched to the next view using dispatcher components. View helpers are used to present business data in separate view components.

At the business tier, 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. A Business Delegate is used as a communication point between the presentation and business tier acting as a proxy for the Session Façade.

During the development of a typical web-based enterprise application, a set of use cases is generally identified from the application requirements. A variation of the pattern combination shown in Figure 1 can be used to implement each use case. In this way, the whole application is developed around the same design rules enhancing the readability and the maintainability of the system. Besides, applying such a pattern combination for each use case promotes better separation of concerns. Each programmer can concentrate on her specific development role defining clear interfaces how to interact with the other components. The clear separation between presentation and business layers makes page designers for example independent from other programmers who are implementing the business logic.

However, other J2EE applications have very specific business domains and so they require specific design solutions. In some cases, neither the discussed pattern system nor a variation of it can be used to design the application. Nevertheless, we believe that a software pattern system is by definition an open system that is dynamic, adaptive and evolving. The system can be extended by new design rules and ideas that could make it suitable for wider range of application problem domains.

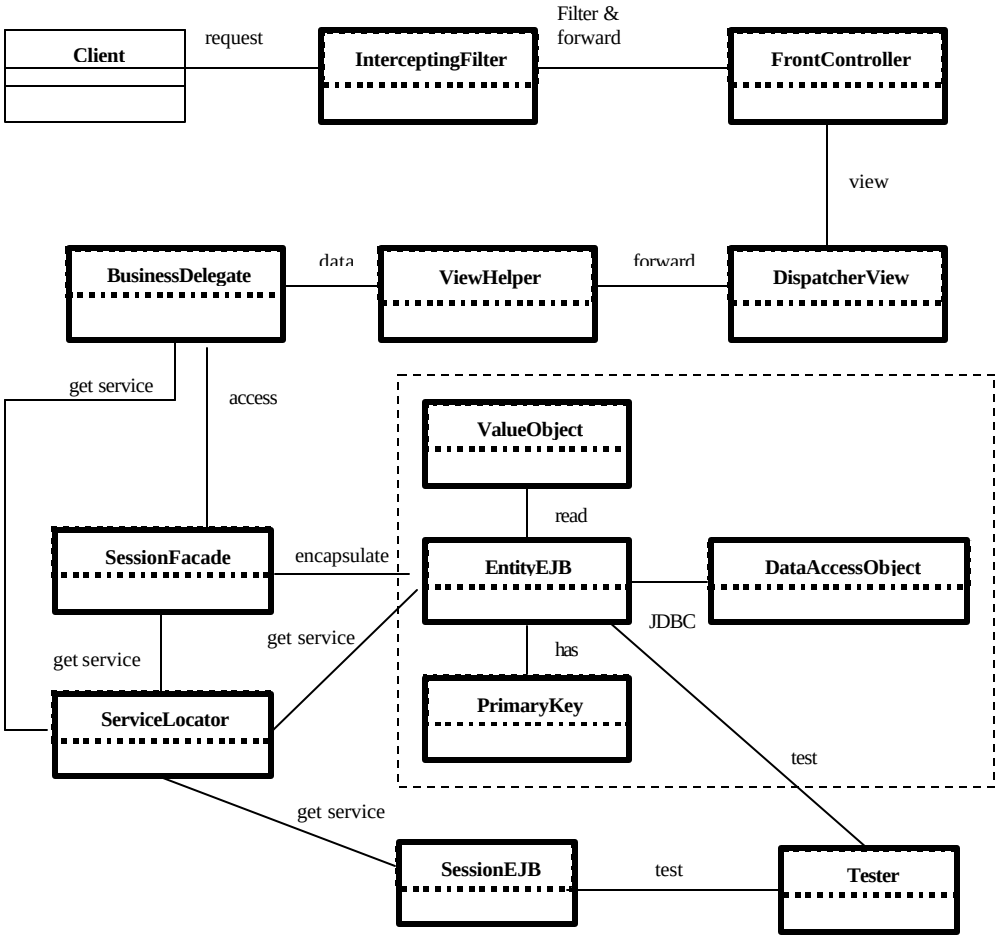

Figure 1. A J2EE pattern system

3 Fred-based J2EE Environment

Fred is a prototype tool for enforcing pattern-based software development in Java. The software architect specifies the architectural decisions of a system as a set of patterns. A Fred pattern is composed of several roles; each role corresponds to one program element (class, method, field…). Roles may have properties like dependencies on other roles, cardinality, constraints, and templates. The task of the software developer is to bind the roles of the patterns to actual program elements following the dependencies of the roles. In this way the tool can guarantee that the given source code conforms to the architecture. More information about Fred tool can be found in [Hak01a] and [Hak01b]. Specialization patterns represent the core of the tool. The framework developer creates the patterns for the framework and puts them together in a unified scenario. The application developer uses the framework to generate a specific instance. Each pattern we discussed in the previous section is modelled as a separate Fred specialization pattern.

It is a relatively easy task to model the specialization pattern in Fred once they are specified in terms of roles and their properties. Figure 2 shows an annotation example of the Session Façade pattern. For example, it took a couple of hours to model the Session Façade pattern shown in Figure 2. Documenting the pattern (role and task description) takes up to half of the annotation time. Equivalently, Fred is used in a simple visual mode to put the patterns in a specific combination forming the pattern framework. At this level, roles that represent the commonalities between all the applications are bound to concrete program elements whereas roles that represent the variabilities are left for the application developer. The time spent to represent the whole framework in Fred is proportional to the number of used patterns. In our case, it took a dozen of hours to generate the whole J2EE environment.

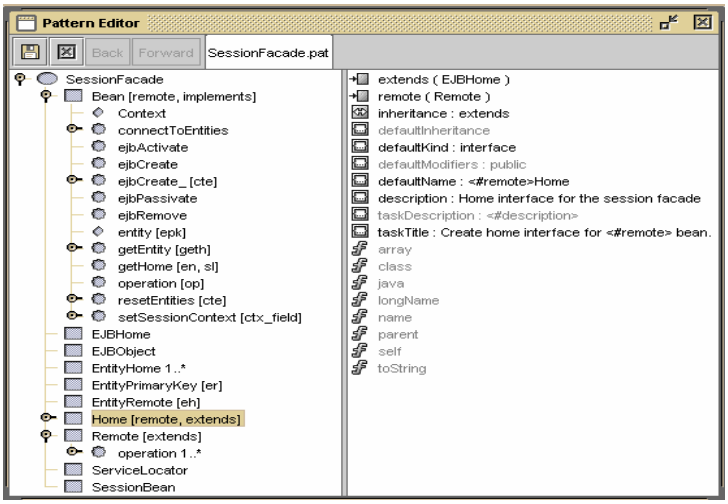

Figure 2. Fred pattern editor

We have applied the *composite pattern* approach used in [Mar98] to create macro patterns. Composite patterns are combinations of other patterns. For example a BMP entity bean is used together with a Primary Key, a Value Object, and a Data Access Object.

The environment generated adds a number of qualities that we find desirable and essential for J2EE application development. These include:

- *Task-based stepwise framework specialization.* The developer can select which pattern to use and can eventually switch to other patterns during the specialization process. The programmer executes a list of simple tasks. This gives her better understanding of the pattern
- *Extensible architecture.* 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.
- *Sensitive source code editor*. The environment can check the source code against of the pattern constraints. The user is informed about possible conflicts every time a new task is provided.
- *Adaptive documentation*. When generating tasks, the instructions given to the user takes into account previous decisions and concrete names used for the program elements. This makes it easier to follow the specialization process.

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. At the presentation layer, the application developer creates different command classes. Each command invokes the right methods on the business delegate, dispatches to the next view using the dispatcher component.

Example:

Figure 3 shows a sample view of the environment during the development phase of a sample J2EE application. The developer is implementing a web-based to-do list where a list of users and their associated tasks can be accessed, manipulated and stored in a relational database. The application is implemented according to the architecture proposed by the design patterns of the environment. The Architecture View to the left gives an idea about the overall application architecture in terms of patterns. The Session Façade composite pattern Facade for example encapsulates two entity beans. User composite pattern represents a bean-managed persistent entity bean whereas Task composite pattern encapsulates a container-managed entity bean. User 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 asks the user to provide findByPrimaryKey method. An optional task is to define a new persistent field.

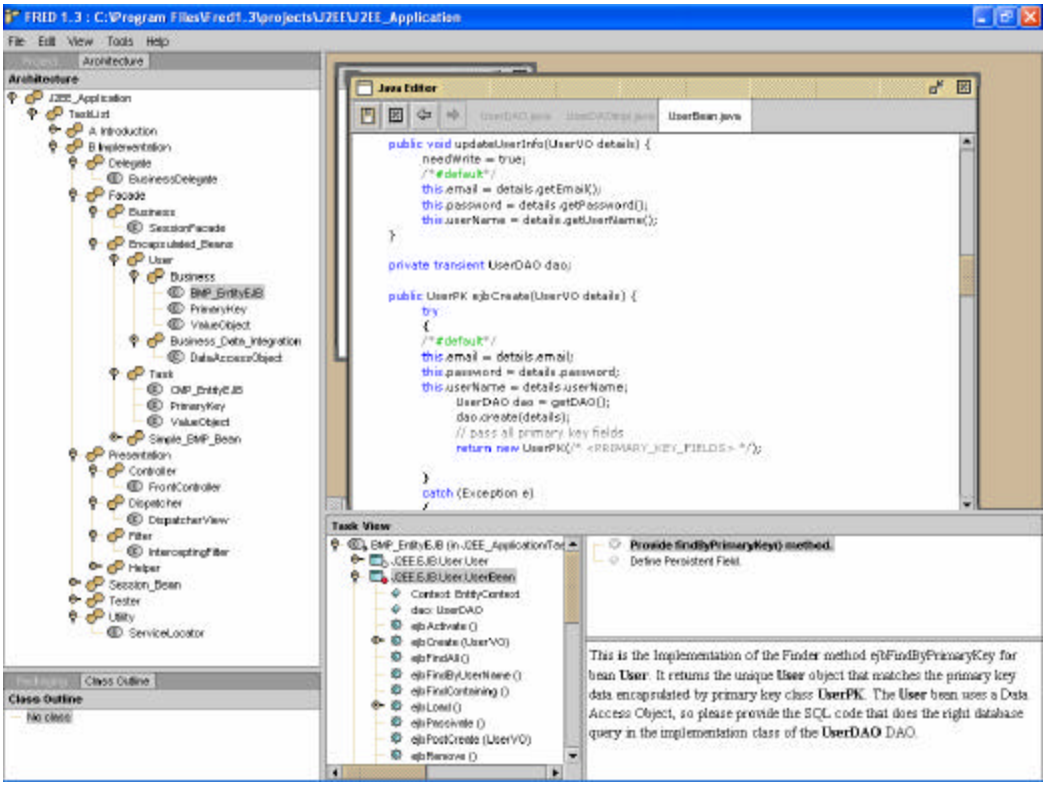

Figure 3. Sample J2EE application development session

Fred supports adaptive documentation that help 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. A substantial amount of code, shown in the Java Editor, is automatically produced as instructed by the pattern specification.

The environment has been tested against several other similar, simple applications. The experiences showed that the environment could improve the quality of the software and reduce the development effort. The improved quality is a consequence of making desirable J2EE design solutions available for the application developer and guiding her to use these solutions. Although it is hard to define specific metrics to compare the development process with more conventional ones, it can be seen that this approach reduces the development effort considerably. Methods for example are usually generated with default implementations. A big part of

the programming tasks is reduced to mouse clicks, the user is often asked to provide a program element with the name proposed by the environment. The benefits apply both for experienced J2EE programmers and novices. In the case of our example application, up to 60% of the total lines of code was automatically generated by the environment. Manually, it could take up to 20 hours to build the discussed application code from scratch. Using Fred J2EE environment the same code has been generated within a couple of hours. Approximately, one third of the development time is spent with using the environment for automatic code generation, whereas the rest was the programmer's part to provide custom business code and implement the user interface. However, these figures should not hide the improvement in the design quality of the generated application that cannot be quantified.

4 Related Work

There are several commercial and non-commercial environments that enable rapid J2EE application development. RealMethods [ReMe02] for example is a tool that provides design pattern based implementation on all J2EE tiers (presentation, business, and integration). The tool requires the user to import in the system a standard XML file that represents the object model. The XML file is parsed and represented as a tree before generating the required code by a single mouse click. 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 J2EE technology. An example of such environments is JBuilder [Bor02]. The tool for example is capable of generating code for enterprise beans in a visual mode. It can also construct entity beans out of data models. 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 J2EE development environment.

- Some tools, JBuilder for example, pay more attention to code generation but give less care to design and architecture solutions. Fred environment pays special attention to the architectural aspect of reuse. The environment constitutes of a collection of collaborating patterns and several other programming rules that could be easily extended and updated.
- Tools similar to realMethods require the preparation of standard representation of the object model. This is a sensitive and time consuming operation. Any errors in the specifications can affect the produced code. Every time the specification changes, new code should be generated. Fred offers a smooth evolution of the application that goes in parallel with the specification.
- Adaptive documentation, supported by Fred, is an essential factor to architecture-oriented software development. Documentation in JBuilder environment is rather static. The environment does not record the history the developer's tasks and does not inform her what to do next.
- The architecture view of Fred environment makes it easy for the developer to understand the problem domain. The task view reflects the status of the application.
- Using Fred environment, it is easy to see how the different tiers of the J2EE application communicate with each other. The user does not need to refer to the generated code as in the case of realMethods.
- 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 J2EE pattern elements and complains if the contract between the architecture and the developer is broken.

5 Conclusions

In this paper, we have presented the main features of J2EE Enterprise Java Beans component architecture. We have briefly studied some proven design solutions to J2EE application known as "J2EE design patterns". These solutions could significantly improve the design and architecture of distributed enterprise applications. Using a general architectural tool called Fred, we have developed a pattern based framework for J2EE applications. The generated environment can improve the quality of J2EE software and minimize development time and effort. We compared the framework to some other products that offer tool support for the technology. Fred environment looked very promising and contributed with many useful new features to conventional application development environments for J2EE systems.

Nevertheless, we are aware of several limitations that we recognized in our approach. First, rather than considering one implementation strategy, the environment could present the user with different solutions, at different levels of abstractions. Second, the environment comes with no control on what developers add to the generated code. For instance, it would be very beneficial to check if the user has implemented the methods as intended by the framework developer. In addition, because Fred cannot generate non-Java code, view components at the presentation tier have to be written in Servlets. View helpers implemented with Servlets are hard to understand and maintain, it is more convenient in implement them using JSP's for example. Moreover, the environment could be enriched by new J2EE patterns and should provide support for EJB 2.0 specification.

However, by adding new patterns to the system, the framework could become more and more complex, which may lead to a longer learning curve. Also, the application development phase may become more complicated and error-prone. As the framework gets larger, using a standard generic architecture may require a bigger coding effort and so there will be no control over the overhead associated with the application development. It is essential to keep the size of the framework as small as possible. In this way, the framework remains simple and understandable, and therefore can be quickly adopted by novice developers. Also smaller frameworks tend to be more reusable and can be applied to wider problem domains than complex frameworks. Instead of continually adding new patterns to the existing framework, it is more convenient to create separate framelets [PrK99]. Framelets are small frameworks consisting in a small number of classes that represent a clear substructure of a component. Framelets could either be used by the framework developer to build more complex frameworks or by the framework specializer to generate application code. A future plan of this work is to develop a collection of framelets that can be used either discretely or in combination to cover a wider range of problem domains.

References

[Gam94] Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley 1994.

[ACM01] Alur D., Crupi J., Malks D., Core J2EE Patterns: Best Practises and Design Strategies, Prentice Hall PTR, 2001.

[Hak01a] 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.*

[Hak01b] 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.*

[Mar98] Martin R., Riehle D, Buschmann F., Pattern Languages of Program Design, Addison-Wesley 1998, *163-185* .

[reMe02] The realMethods: **http://www.realmethods.com** , April 2002.

[Bor02] Borland JBuilder: **http://www.borland.com/jbuilder/**, April 2002.

[HaK02] Hammouda I., Koskimies K., Generating a Pattern-Based Application Development Environment for Enterprise JavaBeans. *Computer Software and Applications Conference (COMPSAC'02) Oxford, August 2002*, accepted

[PrK99] Pree W., Koskimies K.: Framelets - Small is Beautiful. In: *Building Application Frameworks: Object-Oriented Foundations of Framework Design (M.E. Fayad, D.C. Schmidt, R.E. Johnson, ed.), Wiley 1999, 411-414.*