# JDragon: Generating J2EE applications out of database schemas and XMI metadata

Dr. Jörg Friebe
Dr. Cornelia Haber
Dr. Jörg Ritter

OSC – Information Management AG
Industriestr. 11
26121 Oldenburg
Germany
friebe@osc-im.de
haber@osc-im.de
ritter@osc-im.de

**Abstract.** JDragon is a J2EE application generator. It enables its users to generate whole applications based on database schemata. The application generated allows its users to read dates from the database, insert new database entries or to update existing entries. The whole application is generated without writing one single line of source code. Once generated, the application may be enhanced to integrate with other applications and additional business logic may be introduced. In this paper we describe the JDragon application and compare it to other approaches that generate J2EE applications.

## Introduction

Though the J2EE standards are quite young and still evolving (at the time of writing J2EE 1.4 is a Proposed Final Draft) J2EE was adopted very fast by the community of software developers. Not only research projects but big software companies like e.g. ERP vendors rely on J2EE as a strategic platform. But introducing J2EE technology is not that easy. Developers have to take J2EE design patterns into account in order to implement performant, flexible, extensible applications. Once familiar with J2EE and the J2EE best practices powerful applications integrating legacy applications can be build. Still J2EE development causes a development overhead as e.g. for each enterprise java bean at least four files have to be created: the Home and Remote Interface, the bean class itself and the deployment descriptor.Whan building applications according to the EJB 2.0 standard there may even be two more interfaces: the Local Home and the Local Interface. As these files are not independent developing each one separately is error prone and uncomfortable. Modern IDEs know about these dependencies and support developers in keeping the bean and its interfaces compatible.

Another approach is followed by projects like XDoclet[1] or EJBGen[2]. They introduce a technology where developers just need one file, the Bean file itself. With the help of special JavaDoc tags meta data is inserted into the bean file. Later on the source file is parsed and deployment descriptors as well as interface files are generated for the bean. Yet another approach introduced by OMG is the Model Driven Architecture (MDA)[3] which completely ommits technology-specific issues such as deployment in the first development step and later on generates these from platform independent metadata as far as possible.

In our JDragon project we follow a different approach. Here application skeletons are generated based on database schemata or object schemata. The application generated allows its users to read data from the database, insert new or to update existing tuples. The application is generated without writing a single line of source code. Once generated, the application may be enhanced to integrate with other applications, the graphical user interface may be customized to allow more sophisticated interactions or additional business logic may be introduced.

In this paper we present first results of the JDragon project. The paper is structured as follows. First the JDragon process is outlined. Then each phase of the JDragon process is explained in detail, discussing the artefacts generated in that phase and data used as input to each phase. The paper closes with a short comparison of JDragon to other J2EE generation frameworks.

## JDragon

Our JDragon project follows the idea that quite often software development does not start from scratch. In our experience very often legacy data and functionality needs to be rescued into new applications. Therefore our approach starts with the legacy data, which is usually stored in a relational database management system. Describing JDragon in one sentence one could say that JDragon connects to this database, extracts schema information and generates a J2EE application (EJBs, JSPs) based on the database schema. This process is shown in Figure 1. In reality this process is a bit more complex as the JDragon user (i.e. an application developer) can influence not only the beans generated but the presentation components (JSP pages) as well.

[1] http://xdoclet.sourceforge.net
[2] http://www.beust.com/cedric/ejbgen
[3] http://www.omg.org/mda

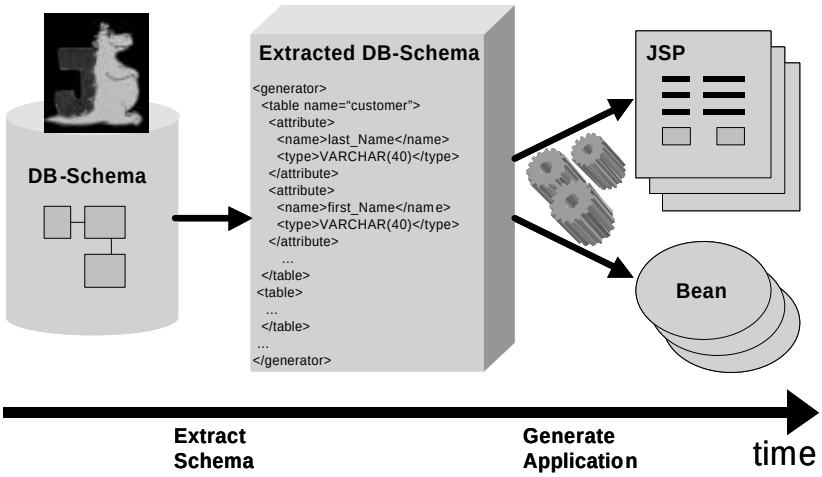



**Fig. 1.** JDragon Process

### Schema extraction

Before the database schema can be extracted information about the database needs to be configured. This configuration is held in an XML file, indicating the jdbc driver, the jdbc connection, database user, database schema and so on. JDragon takes this XML file as input, connects to the database and generates an XML file that represents the database structure.

The XML file describing the database can be manipulated in order to integrate further constraints not represented in the database schema. This manipulation may e.g. be necessary if a database like MySQL is used, where foreign key constraints may not be extracted from the database[4]. Or take e.g. a database where the database schema designer did not enforce foreign key constraints but implemented these constraints in the application. In that case there is no chance to extract the information needed from the database. With JDragon, these constraints may be added in the XML representation of the database schema. That way the J2EE application will obey the constraints even if the database does not enforce them.

Another interesting aspect of the JDragon schema extractor is that it takes previously generated and manipulated database schema information into account. A major problem of generated software is solved this way. Quite often changes of the generated code are lost when the code is regenerated. So if the database is changed later on, e.g. to add a new attribute, foreign key constraints introduced in the XML database schema representation are still there even if the XML file is newly generated.

As discussed earlier we expect the database schema to exist at the time the project starts. Therefore the database layer is not generated by JDragon. But even if there is no existing database this does not refrain you from using JDragon. One way to use

[4] MySQL in versions post 3.23 supports referential integrity using table type InnoDB though.

JDragon in a project that is implemented from scratch is to start by modeling the database schema. In the modeling process well known database modeling tools like ERWIN can be used. Once the database model is completed it is implemented in the database. This can again be done with the help of database modeling tools or manually. From there on you can start using JDragon. The whole schema extraction process is summarized in Figure 2.

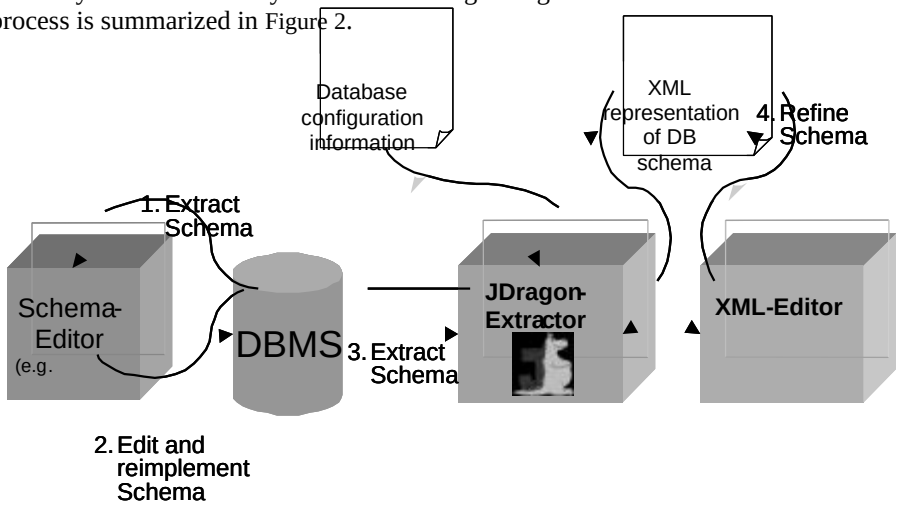



**Fig. 2.** Schema extraction

**Application generation**

More interesting than the schema extraction is generating the J2EE application itself. Before the generation process can be started information about the environment the application is to be deployed to has to be configured. The following information has to be given:

- Application Server type (BEA, WebSphere, JBoss) and version,
- Host
- Port
- Datasource
- Directory the beans are deployed into
- Directory the JSPs are deployed into

This information is needed as the JDragon process does not only generate application code, according to the users' configuration it may even compile and deploy the application.

The architecture of the generated application is a multi tier architecture, consisting of a database layer, a business layer and a presentation layer as shown in Figure 3.

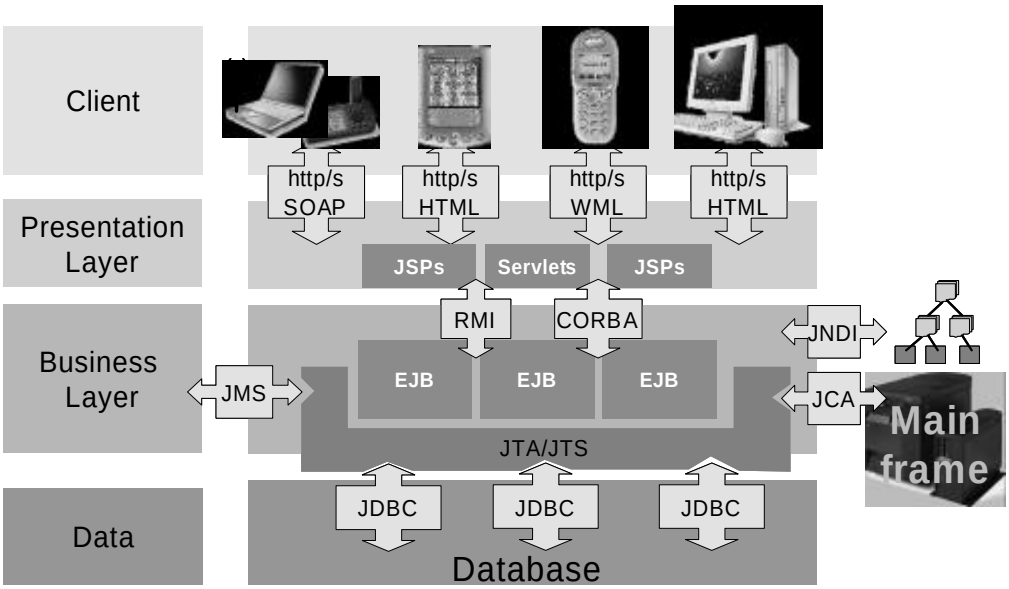



**Fig. 3.** Generated application's architecture

The figure illustrates that the generated application can be extended later on. E.g. it may be integrated with a legacy application using Java Connector Architecture (JCA). The different layers of the generated J2EE application will be discussed in more detail in the following sections with the exception of the database layer as this layer is expected to be present prior to the usage of JDragon.

**Business layer**

The business layer, which implements the application's business logic, is comprised of Enterprise Java Beans (EJB). Let's have a closer look into this layer. The code generated for the business layer is structured according to the Sun Best Practices which are published on Sun's web site[5]. The architecture emerging from these design patterns is shown in Figure 4.

[5] http://java.sun.com/blueprints/corej2eepatterns/index.html

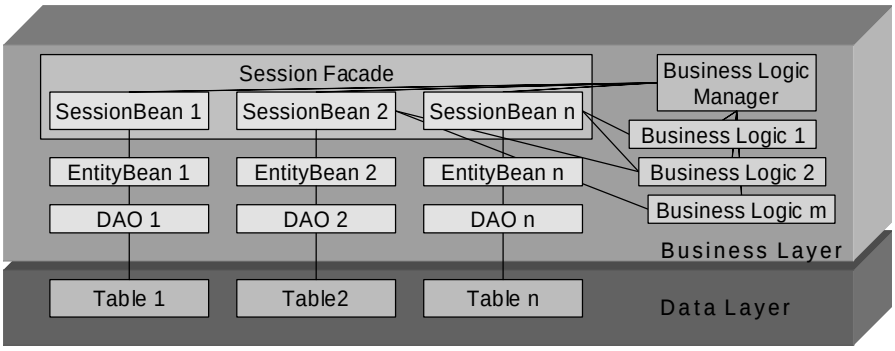



**Fig. 4.** Business Layer

The entity beans generated by JDragon are bean managed persistence beans (BMP), which access the database through data access objects (DAO). The decision to use BMP beans and not container managed persistence (CMP) beans causes JDragon to be more flexible regarding data access. If e.g. the data schema JDragon takes as input is not just one database schema but consists of two schemata implemented in two different databases JDragon is still able to generate one J2EE application which considers integrity constraints even if the constraints concern two tables in different databases.

Clients do not contact the entity beans directly, rather stateless session beans are used to access entity beans. Those session beans do not only provide access to entity beans, they may instantiate value objects (also known as transfer objects, see J2EE design patterns) representing an entity bean´s persistent data. Those value objects can be passed to the client. This way the client can read the bean´s entity data out of the value object and need not cause a remote contact each time it gets an attribute value of the bean.

Another design pattern that is used in the business layer is the session facade pattern. The session facade is a single point of access for all clients. The clients do not need not know about the internal structure of the business layer or about the beans residing in the business layer. As stated in the J2EE design patterns the session facade manages the business objects, and provides a uniform coarse-grained service access layer to clients.

One more aspect shown in Figure 4 is the Business Logic Manager together with Business Logic Beans. The Business Logic Manager is used to integrate new Business Logic Beans, that are not generated by JDragon, with the generated application. The Business Logic Manager and the way it is used is described in more detail later on.

**Presentation layer**

Not unusually, the presentation layer consists of Java Server Pages (JSPs). For each database table several JSP pages are generated: one in order to create a new database entry, one to update a table entry, one to search for database entries, one to show database entries and a last one to delete a table entry, as shown in Figure 5.

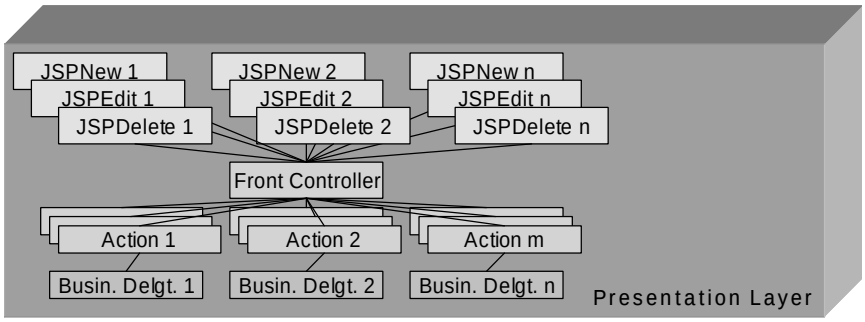



**Fig. 5.** Presentation Layer

We discovered this straight forward approach to be unsatisfactory. In fact most of the times users create new database entries not just one but several database tables are concerned. Considering the JDragon approach this would require the user to fill in several JSP forms, one for each table that is involved, which is quite cumbersome. Therefore the JDragon software had to be enhanced to support sub-forms, i.e. the possibility to combine JSP pages into one JSP page that represents data out of several database tables.

Patterns used in the presentation layer as shown in Figure 5 are the front controller pattern and the business delegate pattern as well as the Model View Controller design paradigm. The Model View Controller Paradigm is introduced by using Struts[6], an open source framework for building web applications developed within the Apache project[7]. Struts provides its own Controller component and integrates with other technologies to provide the Model and the View. In JDragon applications Struts interacts with Enterprise Java Beans for the Model and with JavaServer Pages for the View.

The front controller pattern used in JDragon appliations provides a centralized entry point that controls and manages Web request handling. By centralizing decision points and controls, the controller also helps to reduce the amount of Java code (scriptlets) embedded in the JSPs. The business delegate pattern is used to reduce coupling between presentation tier and business services. The business delegate hides the underlying implementation details of the business service, such as lookup and access details of the EJBs.

[6] http://jakarta.apache.org/struts/index.html
[7] http://www.apache.org




**Fig. 6.** Screen Dump of JDragon application.

Figure 6 shows a screen dump of an application generated by JDragon. It reveals two more JDragon features, namely the multi-language- and the multi-theme support. Multi-language-support denotes that JDragon applications support different languages and the ability to change between languages at runtime. The application developer just has to give a translation for the texts. The flag icons that switch between languages are generated into the pages by JDragon.

Multi-theme support means that different designs of a page (themes) can be generated. The application developer may even allow for the generation of different themes and leave the design decision to the users through a combo box listing the different themes.

If an application developer does not like the standard JDragon design he may design a totally different page layout. How this is realized with JDragon is described in the section about template based code generation.

**Integrating business logic**

A major problem of generated appliations is caused by changes of the information the application is generated from. Once this basic information changes, the application has to be rebuilt. Quite often the newly generated code just replaces the code generated before and that way changes manually made in the generated code are lost.

One may now argue that the code generated should not be manipulated rather the basic data on which the generator operates is to be changed. That is true but in reality – if time runs short – it's more feasible to submit minor changes to the generated code than to change the basic information and generate the code again or even to adapt the code generator. Therefore we decided to introduce a concept that allows changes to the source code to prevail even if the whole application is regenerated.

Our concept is based on the idea that changes to the source code like new functions that check the validity of input data are not inserted directly into the generated entity or session beans. Rather special business logic classes are introduced. These business logic classes implement additional functionality (not generated by JDragon). Business logic classes as implemented by an application developer are just ordinary classes. For JDragon to identify them as business logic classes they are subclassed from a special JDragon class called *BusinessLogicBean*. The name suggests that the business logic classes are EJBs, a fact which is hidden from the implementor. The implementor just writes an ordinary java class. The generator takes this class and generates the appropriate bean and interface classes.

With JDragon business logic beans are dynamic, i.e. they can be included in the application at runtime. Naturally they can be inserted at deployment time as well but the more interesting notion is insertion at runtime which is especially useful wile developing the application. A special component, the business logic manager, administers all business beans. At runtime a new business logic bean is registered with the business logic manager and from then on the bean is taking part in the application. On the other hand beans may be deregistered with the business logic manager and are ignored from that time on. Using the registering and deregistering of business logic beans at runtime allows for updates of individual business logic beans at runtime without restarting the whole application.

**Template based Code generation**

JDragon utilizes template based code generation, i.e. templates are used to specify structure and content of source code files. This way application developers may easily affect the code generated by editing the templates. JDragon templates are written in a special JDragon template language. The concept of the JDragon template language is quite similar to JSPs. The main difference is that it does not only allow for plain HTML pages to be generated, the JDragon template language may generate source code files in any language like e.g. XML, HTML, java, C++ etc. A server based runtime system is not required for the generation process.

For each kind of file that is generated by JDragon, a template exists that determines structure and content. There is a template for each JSP page type (create, edit, search, etc.), for each kind of bean (entity bean, session bean), for additional classes according to the J2EE design patterns (data access object, value object, ….) and so on.

The template that is used for the generation of remote interfaces for entity beans is shown in Figure 7. This way JSP code outside of tags is taken over into web pages text outside of template tags (*{#*) is copied into the generated files without modification. In between the template tags there is java code that is translated and executed by the generator. A special tag (*{#=*) is used to mark code expressions. The value of these expressions is inserted into the files generated.These new tags had to be introduced in

favour of the equivalent standard jsp tage (<%, <%=) to simplify generation of JSP code.

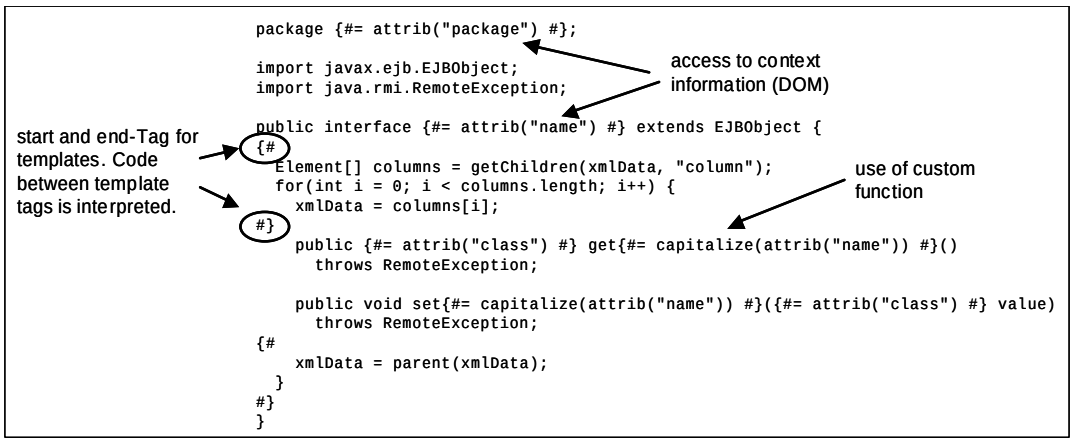



**Fig. 7.** Remote interface template

The generator uses templates in combination with XML files representing the database schema, configuration information, etc. Whenever a template is called it is given an XML node as input. In order to access the XML node, its elements, attributes, the parent node, and so on. special methods are introduced. For instance *{#= attrib("name") #}* is used to access the attribute named *name* of the current node in the example above.

An interface generated with the template is shown in Figure 8.

```
package application;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface customer extends EJBObject {
            public java.lang.Integer getId()
                        throws RemoteException;
            public void setId(java.lang.Integer value)
                        throws RemoteException;
            public java.lang.Integer getCustomerid()
                        throws RemoteException;
            public void setCustomerid(java.lang.Integer value)
                        throws RemoteException;
            public java.lang.String getLastName()
                        throws RemoteException;
            public void setLastName(java.lang.String value)
                        throws RemoteException;
            public java.lang.String getFirstName()
                        throws RemoteException;
            public void setFirstName(java.lang.String value)
                        throws RemoteException;
}
```

**Fig. 8.** Interface generated by JDragon

If an application developer wants the generated files to be structured in another way he can bring this change about by modifying the correspronding templates. If e.g. a developer prefers another application layout he may change the JSP templates, regenerate the application and redeploy it. It is not necessary to change every single JSP generated by JDragon manually. If an application developer is not familiar with the JDragon template language, there is an even easier way to change the layout of an application.

**Determine the application's layout**

The JDragon generator makes solid use of tag libraries. Using the standard JDragon templates the JSPs themselves return pure HTML pages without any use of JavaScript or other non HTML features, so that they may be used with any WWW browser. If an application developer wants a richer user interface using JavaScript he just has to change the templates accordingly.

A cascading style sheet contains layout information for all HTML pages issued by the JSP engine. Small changes to the application layout may therefore be realized by changing the style sheet. As style sheets can only determine the presentation of tables, text, form elements, … but not the structure of the whole document, changes to the layout through changes in the style sheet are quite restricted.

JDragon does not only generate JSPs, for each JSP an intermediary file is generated as well as shown in Figure 9.

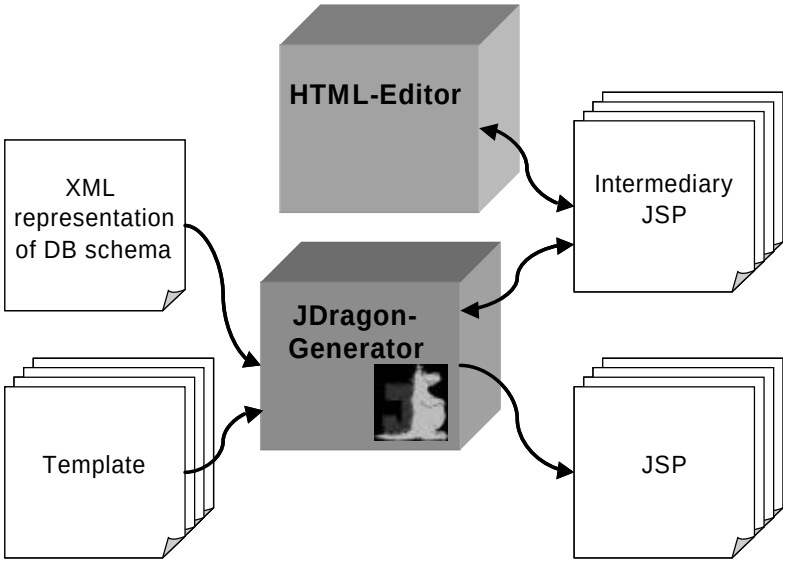



**Fig. 9.** Generation of intermediary JSPs

This intermediary file shows the HTML structure of the output file generated from the JSP engine. Data inserted by the JSP scriptlets is marked through a special syntax. Therefore intermediary files are not pure HTML but they are similar enough to be edited manually or with the help of special HTML editors like Frontpage. Using Frontpage an application developer may e.g. rearrange the HTML form elements. If the application is regenerated these changes are taken into account and the JSP file will follow the rearranged form. This method for changing layouts is appropriate if only a few JSP pages are redesigned. It should not be used if the layout of the whole application is to be redesigned. In that case the templates should be changed.

## JDragon versus other approaches

When we started the JDragon project there were hardly any other J2EE generation tools to be found. Now the situation has changed. On one hand tools coming from the field of modeling are starting to offer code generation mechanisms. On the other hand there are tools where only the bean class itself is to be coded and all interfaces and the deployment descriptor are generated by a code generator. In the following sections both kinds of tools and approaches will be discussed.

### JDragon and the model driven architecture

With the Model Driven Architecture (MDA) the Object Management Group (OMG) addresses the complete life cycle of designing, deploying, integrating, and

managing applications as well as data using open standards. The MDA makes heavy use of other OMG standards: UML (Unified Modeling Language), MOF (Meta Object Facility), XMI (XML Metadata Interchange) and CWM (Common Warehouse Metamodel).

MDA starts development with the creation of a Platform Independent Model (PIM). The PIM represents the functionality of the application regardless of platform or technology. A PIM is specified in terms of UML diagrams. Pervasive services like persistence, transactions, etc. may be used in a PIM.

Later on the PIM is transformed into a Platform Specific Model (PSM) once again expressed in UML. The PSM e.g. specifying that the application is an J2EE application and showing Entity and Session Beans makes use of the UML Profile concept. For the transformation between PIM and PSM the MDA definition document lists four ways with increasing level of sophistication and automation from completely hand coded to completely automated.

The source code is then generated out of the Platform Specific Model, eventually modified or enhanced manually and deployed. The main MDA development artefacts are shown in Figure 10.

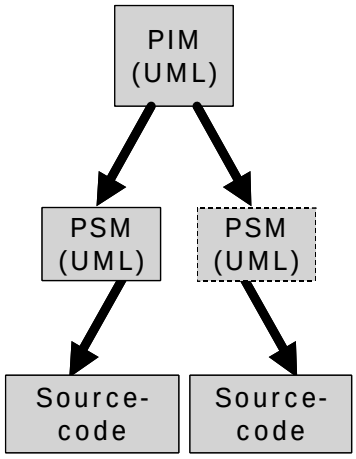



**Fig. 10.** MDA artefacts

Now let's compare the MDA approach to our JDragon. JDragon starts with a database schema that is extracted and represented in an XML file. Now if this XML file is not a proprietary JDragon DTD but follows the XMI DTD, it expresses an UML data model. This XMI representation may be imported into a MDA tool where a PIM based on this database schema may be created as shown in Figure 11. More interesting is the second exchange arrow showing that a PIM (in XMI format) may be used as input to the JDragon generator which generates the source code.

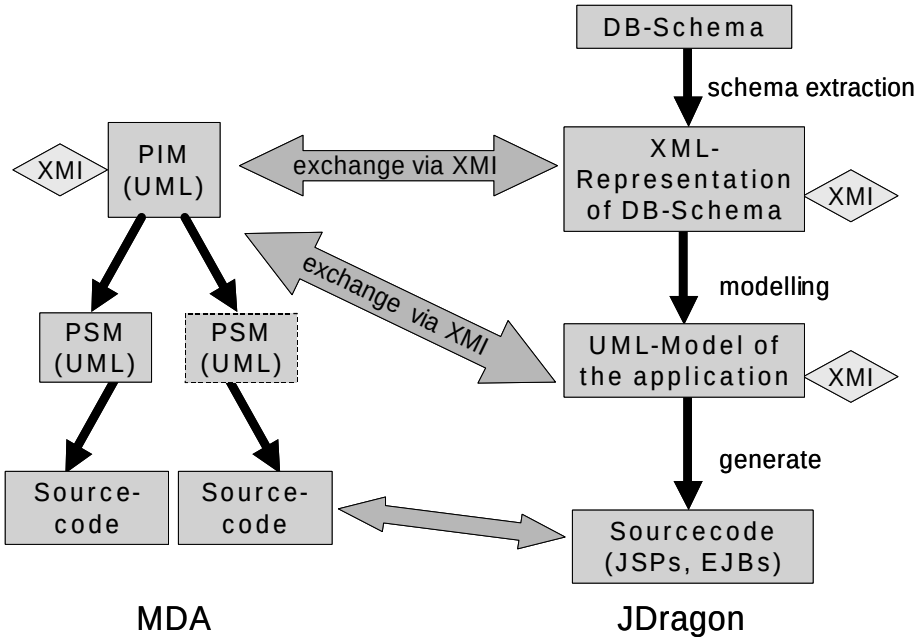



**Fig. 11.** MDA and JDragon

Right now the XML JDragon uses for database schema representation and as input to the generator is not XMI conform but we intend to change this representation to XMI. Once this change is implemented, JDragon may be seen as an MDA-light-tool. Light because no PSM-UML model is generated and right now only J2EE applications are supported.

**Other J2EE generators**

The development of J2EE applications relies on the developer to code for each bean at least four files that have to be kept in tune. This is a cumbersome and error prone task. Therefore quite soon projects trying to reduce coding overhead arose. Examples for those projects are XDoclet or EJBGen. Both projects use special JavaDoc tags to insert meta data into the bean file. Using these tags a developer may specify all information needed in order to generate the interfaces and the deployment descriptor. There are tags to define bean type, jndi-name, cmp-version, transaction-type, and so on. XDoclet as well as EJBGen allow the developer to use J2EE design patterns, allowing e.g. tags for the generation of value objects or in the case of XDoclet data access objects.

Later on the source files containing the JavaDoc tags are parsed and artefacts such as XML descriptors and/or source code are generated from it. Again the generation is based on templates, but with XDoclet and EJBGen the templates use the information provided in the source code and its JavaDoc tags.

Both projects – EJBGen and XDoclet – started as code generators for EJBs. However XDoclet has evolved into a general-purpose code generation engine. Another difference between XDoclet and EJBGen lies in the application servers they support. Whereas XDoclet generates code for different application servers (BEA WebLogic, IBM WebSphere, JBoss, …) EJBGen only supports BEA WebLogic.

How do EJBGen and XDoclet compare to JDragon? EJBGen generates components for the business layer only. It does not consider presentation layer or database layer. Therefore it aims at a different usage scenario than JDragon. While JDragon generates whole applications, EJBGen eases the implementation of Enterprise Java Beans.

XDoclet offers tags for presentation layer as well (e.g. it has tags for usage with struts). Still XDoclet will be used in another scenario than JDragon. JDragon is used to automatically generate applications from database schemata. The application generated can be used to read or update database data or to insert data into the database. If you need such an application JDragon is more suitable than XDoclet because you do not have to write any code yourself and therefore it's easier to use. Another scenario where JDragon should be used is to generate a base application. This base application (entity beans, data access objects, value objects, …) may then be used to build complex business logic on top of it (e.g. optimization algorithms, …). In order to build the beans and classes on top, XDoclet is probably the right tool.

Another tool supplementary to XDoclet is Middlegen[8]. Middlegen is a database-driven code generation engine based on XDoclet. The graphical interface to Middlegen can be used to extract a database schema and generate EJBs, JDO classes or JSP pages for the database tables. In fact, Middlegen does not generate the EJBs, it generates XDoclet classes that can then be used to generate the classes and interfaces. Taking Middlegen and XDoclet together the functionality sounds quite similar to JDragon. Still the generation of JSP pages out of Middlegen is not fully supported right now. Another main difference lies in the fact that JDragon allows the user to manipulate the extracted database schema, i.e. to insert additional constraints like foreign key relationships. Middlegen just takes the database as it is.

A further difference to point out between XDoclet, EJBGen and JDragon is the integration of business logic into an application at runtime. While XDoclet and EJBGen allow for enhancements at deploy time only, JDragon is able to integrate business logic classes at runtime.

## Summary

Over the years we have developed quite a few J2EE applications ranging from small applications to big critical applications with high demands on performance, availability and reliability. Implementing these applications the JDragon idea came up. Our experiences with J2EE caused quite a few changes to JDragon especially regarding the structure of the applications generated by JDragon. We've been using JDragon in our projects for some time now and we found it to be very useful.

[8] http://boss.bekk.no/boss/middlegen/index.html