World of Computer Science and Information Technology Journal (WCSIT)
ISSN: 2221-0741
Vol. 2, No. 4, 147-156, 2012

A Generator of MVC-based Web Applications

Strahinja Lazetic
Instituto Superior Tecnico
Universidade Tecnica de Lisboa
Lisbon, Portugal

Dusan Savic, Sinisa Vlajic, Sasa Lazarevic
The Faculty of Organizational Sciences
Belgrade University
Belgrade, Serbia

Abstract— This paper presents a generator of Spring MVC-based web applications. The generator was designed using Freemarker template and HibernateTools toolset. The generated applications are fully functional web applications based on three tiers architecture and MVC software pattern. The structure of Spring web applications is explained in brief, while designed templates and their role in the process of code generation are elaborated in more detail. The generator was designed to show a possibility for fast development of a custom generator as a solution for specific application requirements.

Keywords- applications generator; Spring; Freemarker; HibernateTools; MVC.

I. INTRODUCTION

Nowadays when information technologies represent one of the fastest developing business areas, many software companies can respond to client requirements with the same quality. Time and price (that depends on time) required for project realisation became the crucial factors for getting a job. The question is how these costs can be reduced thus improving productivity and surpassing competition.

Application generators are being imposed as a solution for the problem. They represent software components that automatically produce other software components. Every generator uses a specific input and produces an output by applying defined rules. Commonly used inputs are data model and templates. Outputs can be different kinds of text files such as HTML pages, Java source code files, SQL scripts and XML files.

Using generators has many advantages over manual coding. It significantly reduces time required for application design. A generator increases the quality of source code by producing standardized code hence reducing number of syntax errors which are common for manual coding. In addition, changing application code requires only changes in particular templates and restarting the generator. Developing a generator as a highly sophisticated software component motivates software developers and reduces monotony which would appear during manual coding of significant amount of similar or repetitive code [1].

There are many papers reporting research and practice in designing applications generators . Basically, there are two approaches to developing application generators – one based on proprietary software components, predominantely generating .NET applications (Microsoft-technology oriented) and the other one, based on integration of Java-centered free software tools. Some examples of the former approach are [2], [3], [4], [5], [6]. Examples of the later approach are JAG - Java Application Generator [7], Skyway Builder [8], etc. These generators, in general, are based on freely available components and technologies such as EJB and Hibernate [9], [10] template engines such as FreeMarker and application development frameworks, such as Spring framework [11] or JBoss Seam framework [12]. Skyway Builder, for example, is a template-based generator and one of the well known commercial solutions for applications generation. It uses templates written in XML/XSL and generates fully functional Spring applications that implement basic CRUD operations.

Further on, generators may be based on independent or a specific data model. For example, a UML specification–based source code generator [13] is an application generator which uses independent data model based on UML specification.

This paper describes fast development of a simple application generator for a specific application. Its main task is to justify development of a custom generator from the very beginning in case of short time limits and specific project requirements. The generator relies on an existing database as a starting point for code generation and produces fully functional Spring applications using designed templates. Its development is semi-automated because existing code libraries are used for data model production.

The generator presented has a lot in common with other solutions: it is template-based, it uses Spring framework, it generates applications that implement basic CRUD operations. Still, there are subtle but important differences between them: the generator presented is based on a specific database, as opposed to independent data model, and it keeps only basic customization of the code generation process. Both features make the generator presented simple, cost-effective and more suitable for specific project requirements than more general solutions.

When writing this paper, the author's previous practical experience in code generator development was used as a foundation. *The Asset Management System* for Ministry of Interior of Republic of Serbia developed within Comtrade IT Solutions and Services Ltd [14] was built using JSF/JBoss Seam application generator constructed for the system specific needs. It is a proprietary intranet system for internal use, concerned the management of vehicles and other assets of the Ministry of Interior and consisted of several hundreds of use cases. There were several hundreds of domain objects. The success of the generator was better than expected at first: about 60% of use cases were implemented using the generator – completely or with minimal manual intervention, about 20% - with moderate manual changes, and the rest we had to code manually.

The paper is organized into six sections. In the next section a common problem which is present in the process of manual coding is described. Existing solutions for code generation are also briefly described. In the section 3, the proposed solution for the problem is presented. The structure of the Spring application and specific problems connected with it are introduced as well as possible improvement using templates for code generation. Then, in the next section, the architecture of the generator is described in details. First, the basic principle of the generator functioning is presented and afterwards design of the most characteristic templates is described. At the end of the section, the procedure for starting the generator is outlined. In the section 4 a summary of the paper and possible improvements of the generator are stated. In the last section a conclusion to the work presented and a list of advantages of the generator are given along with a short statistics related to the generator performances.

II. THE PROBLEM STATEMENT

An enterprise application used by a large company is usually based on a database as its data storage. As a place where company's data are stored, the database represents a core of the business application and a starting point for its development. These applications are commonly based on three tier architecture which enables independency and easy maintenance of application layers. The structure of a three tier application that uses a database with N tables is shown in Fig. 1.

Insertion of a new table into the database would require design of additional program classes of application logic as well as appropriate user interface pages and controllers in the presentation layer that would deal with the new table. The number of new classes may vary and depends on the table structure and relationships with other tables as well as

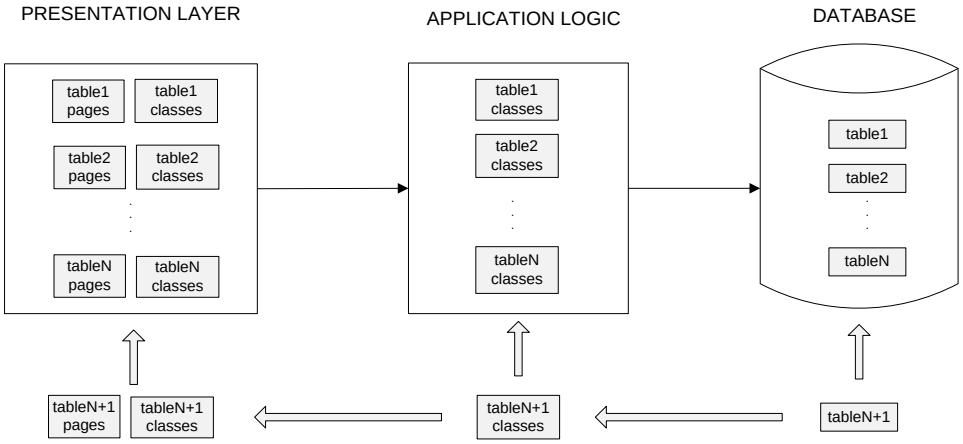

Figure 1. The structure of a three tier application using N – tables database

requirements for the program logic which would cover table's functionality. In case that a dictionary table is added, it is often only needed to implement basic CRUD operations. In this situation, application generator would significantly reduce design time of the required program code.

In [13] authors describe a generator based on UML specification and XML/XSL templates. The generator uses existing UML tools for delivery of UML functionalities. Its most important characteristics is the preserved flexibility towards the target programming language. In addition, the generator provides a wizard for customization of code generation process based on input parameters.

A good example of a commercial solution that resolves the described problem is *Skyway Builder*, which generates fully functional Spring applications that implement basic CRUD operations. Code generation process is based on an UML class model and the tool provides integration with a large scope of UML tools. The generator can be used as a plug-in for Eclipse

IDE hence enabling different wizards and visual development of Spring applications. One of important features is its modularity which provides for selection of application tiers to be generated.

Nevertheless, in real projects environment, there are cases when existing solutions for applications generation do not suite specific project requirements enough or they are not cost-effective in terms of price or time needed for their customization. Moreover, a software company's development team usually has its technology stack already defined, which causes further difficulties in using existing generators. Based on this issue, a JSF/JBoss Seam generator was developed for needs of *The Asset Management System* for Ministry of Interior of Republic of Serbia designed by author's company Comtrade IT Solutions and Services. The generator was designed to cover the wide range of specific technologies that had to be incorporated into the application, such as Hibernate ORM, EJB3 Entity classes, JBoss Seam framework, Java Server Faces and Richfaces and that was enabled using Freemarker template pages. The generator used a database as a starting point for application generation and a data model was created automatically using HibernateTools functionality. Since the database consisted of approximately 400 tables, large portion of which representing simple dictionary tables, the generator showed great efficiency in code generation, significantly reducing time required for the project completion.

The idea for this paper was founded on the described practical experience with the goal to show a possibility for fast development of application generator which would completely match the structure and technologies of the specific application. Like the *Source Code Generator Based on UML Specification*, the generator uses templates for producing source code files. Differences are that it does not use an independent data model and does not provide flexibility towards the target programming language, but generates only Java applications. The generator produces fully functional Spring applications which implement the basic CRUD operations, same as the commercial *Skyway Builder*. On the other side, Skyway Builder provides rich user interface and more advanced customization such as selection of application tiers to be generated.

Both facts - the generator being based on a specific database instead of an independent data model, as well as a limited customization and flexibility level, make the generator presented the best-suited, simple and cost-effective solution to the problem that was to be solved: fast development of an application with specific project requirements and a specific database structure, such as a data-centred web application with a large number of database tables, with simple and repetitive logic, e.g., dictionary tables.

III. PROPOSED SOLUTION

In designing our solution to the problem stated, we proceed through two steps. The starting point is the Spring framework. The second step is the development efficiency improvement by using templates.

The constructed Spring application will implement the basic CRUD operations and interact with presented database using functionalities of Hibernate Entity Manager. Domain objects will be implemented in EJB3 Entity class technology which uses annotations for mapping to database tables. DAO pattern will enable loose coupling of business layer and persistent layer. In the presentation layer Spring controllers [11] will be used for dealing with web pages which will be implemented in JSP technology [15].

Let us consider, as an example, a simple database, consisting of two tables - Employee and Company connected with the one-to-many relationship (shown in Fig. 2).

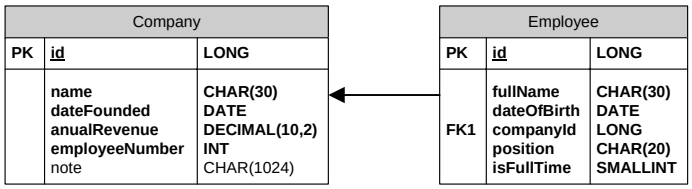

Figure 2. Database schema for the Spring web application

This schema will serve as a basis for construction of a sample Spring web application which will be in charge of the schema management.

Fig. 3 then represents components of the Spring web application managing this simple database. Common components for domain objects Company and Employee are positioned on the central line of the figure. A short description of these components is given in Table 1.

Other components shown in Fig. 3 are domain object specific and they are located on the left and the right side of the figure. These are JSP pages for viewing and updating data of the domain objects, Spring controllers which manage these pages, classes that implement business logic, DAO classes and domain object classes. For each domain object, eight Java source code files and two JSP files had to be designed. The previous example is based on a simple database which contains only 2 tables. In real cases a database may have several hundred tables. As an example, for a database which contains 100 tables it would be necessary to design 1000 files. In addition, specific code for every domain object should be added into described common files.

This is where we proceed to the second step in designing our solution: we introduce templates.

For each component of the system, one template will be designed and will serve for its generation. These templates will represent a basis of the application generator which would merge them with specific database data thus producing output files. After processing, one output file per domain object will be produced for every domain object specific component. In addition, one output file for every common component will be generated.

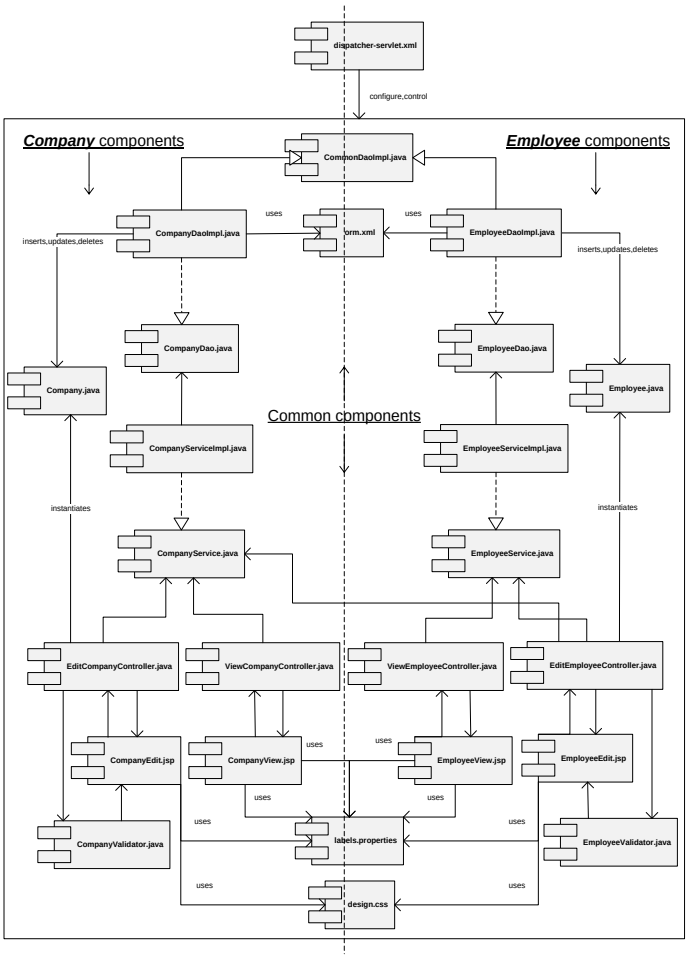

Figure 3. Spring web application components

TABLE I. DESCRIPTION OF COMMON COMPONENTS FOR ALL DOMAIN OBJECTS OF THE SPRING APPLICATION

Component name	Component description
dispatcher-servlet.xml	Main Spring configuration file. It contains configuration data about each system component which is defined as Spring bean
orm.xml	It contains Hibernate named queries for interaction with a database for each domain object
labels.properties	It is used for application internationalization. It contains values of application text labels common for all components, as well as domain object specific text labels
design.css	It contains css definitions required for application design
CommonDaoImpl.java	Dao functions common for all domain objects are put into this file

IV. THE GENERATOR

The process of application generation is based on *bottom up* principle[1] [Bauer, 07], i.e. the generator uses a database as a starting point for producing application layers code. For producing the data model it uses HibernateTools toolset and its method for reverse engineering. JDBC configuration, which is a part of the toolset, uses a configuration file to read data from the database and using reverse engineering creates an internal Hibernate meta model. Then, HibernateTools exporter, hbmtemplate, uses the meta model created and loads Freemarker templates to produce output files [16] This procedure is presented in Fig. 4.

Specific database and Freemarker template files represent inputs to the generator and outputs are Java source code files, JSP pages, XML, CSS and .properties files which constitute parts of the presented Spring application.

Since HibernateTools creates a meta model automatically, based on a specified database, the next step is to design templates for particular system components, which will serve for generation of those components.

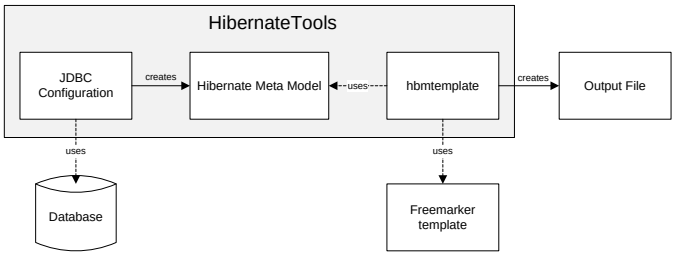

Figure 4. Procedure of code generation using HibernateTools toolset

A. *Templates design*

Within the generator, template files are implemented using Freemarker template language [16]. Hibernate meta model objects are available inside the scope of the templates with the name *pojo*. These abstract objects will get their specific values after processing the templates with a specific data model. HibernateTools offers many useful methods applied to the *pojo* object. In the following paragraphs, design of the templates for the most characteristic components of the Spring application will be presented.

1) *Domain objects*

In the presented Spring application, EJB3 Entity classes are used for the implementation of domain objects. They contain annotations which enable automatic mapping to database tables. For each domain object in the application, one Entity class is required. In the generator, one template is included that serves for generation of these classes thus significantly reducing time required for their design. This

[1] Other commonly used principles for applications generation are *top down* – a database schema is generated upon existing domain objects; *middle out* – the starting point are mapping metadeta that domain objects and database schema are generated from; *meet in the middle* – combination of existing database schema and existing domain objects

template already exists in the HibernateTools library so it is used as such. A part of the template that refers to generation of the relationships annotations is presented in the next fragment:

```
<#if c2h.isOneToOne(property)>
   ${pojo.generateOneToOneAnnotation(property, cfg)}
<#elseif c2h.isManyToOne(property)>
   ${pojo.generateManyToOneAnnotation(property)}
   ${pojo.generateJoinColumnsAnnotation(property, cfg)}
<#elseif c2h.isCollection(property)>
   ${pojo.generateCollectionAnnotation(property, cfg)}
<#else>
   ${pojo.generateBasicAnnotation(property)}
   ${pojo.generateAnnColumnAnnotation(property)}
</#if>
```

The generator uses Hibernate file hibernate.reveng.xml for adjustment of the reverse engineering process. This file allows customization of:

- Mapping between database types and types inside Entity classes
- Filtering database tables which the Entity classes and other components of the system will be generated for
- Customizations specific for database tables and related classes such as table and attribute names mapping, primary key generation strategy and relationships and foreign keys mapping
- Custom code generation which will be added into the Entity class
- Generation of the *import* instructions which will be added into the Entity class

A part of the hibernate.reveng.xml file which excludes generation of the application components for Employee database table is shown in the following fragment:

```
<table-filter match-schema="PAPER" match-name="EMPLOYEE"
 exclude="true"/>
```

2) Named queries

DAO classes of the presented Spring application call named queries located in the external xml file orm.xml. Named queries are implemented in the Hibernate query language (HQL). In the next example, a named query for SQL *select* statement for the Company domain object is presented.

```
<!— Company →
<named-query name="findAllCompanies">
   <query>
      select company from Company company
   </query>
</named-query>
```

For each domain object in the application, one such query is required. Besides *select* query, orm.xml file contains all other queries required for execution of the DAO methods of the specific domain object. The application generator resolves this problem using a template which will generate all the required queries after being processed with a specific domain model.

In the following fragment, a part of the template for generating *select* statement is presented.

```
<#foreach pojo in c2j.getPOJOIterator(cfg.classMappings)>

    <#include "../common/conf.ftl">
    <!-- ${pojoName} -->
   <named-query name="findAll${pojoName}s">
      <query>
         select ${varName} from ${pojoName} ${varName}
      </query>
   </named-query>
</#foreach>
```

3) Spring controllers

In the Spring application there are two controllers that extend Spring class SimpleFormController. They are in charge of managing web pages and calling appropriate business logic methods. Adding a new domain object to the application structure would require designing two more controllers. A solution the generator offers to this problem is designing one template for each controller, which would serve for generation of that controller for all the domain objects of a specific domain model.

The first controller manages the web page for displaying data in tabular form. The characteristic method for this controller is *handleRequest* method which loads data to be presented to an end user. The method is inserted into the template for this controller as shown in the next fragment of code:

```
public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
   ModelAndView mav = new ModelAndView("${pojoName}s");
   ${pojo.importType("java.util.List")}<${pojoName}> ${listName} =
   ${varName}Manager.findAll${pojoName}s();
   mav.addObject("${listNameList}", ${listName});
   return mav;
}
```

The second controller is in charge of a page that contains a form for adding new and updating existing objects. Two methods are required for this controller. *FromBackingBean* method instantiates a new object and *onSubmit* method saves the object after the form is submitted. These two methods are implemented into the controller template. Methods *referenceData* and *initBinder* are required only in case that domain object, the controller is in charge of, contains references to other domain objects. In that case, these methods

serve for loading reference objects and their binding with the main object, respectively. In case of the presented Spring application, controller for Employee domain object contains these methods which load and bind referenced Company objects. These methods are inserted into the template but they are generated conditionally. A part of the template that generates the method *referenceData* conditionally is shown in the following fragment:

```
<#foreach property in pojo.getAllPropertiesIterator()>
    <#if isManyToOne(property) || isOneToOne(property)>
      @Override protected ${pojo.importType("java.util.Map")}<String,
      Object>
      referenceData(HttpServletRequest request) throws Exception {
.....
```

4) User interface

The user interface of the Spring application is designed using JSP technology with embedded Spring tags for handling forms. For each domain object there are two user interface pages, one for displaying objects and another for adding and updating existing objects. For each page, one template for its generation is designed.

On the page for displaying objects, data are rendered in the form of an HTML table. Each row contains data for one domain object instance, which means that one table column represents one object attribute. In the next fragment a part of the template for table definition is presented:

```
<table border="1">
  <c:forEach items="<#noparse>${</#noparse>${listNameList}}"
              var="${varName}">
    <tr>
    <#foreach property in pojo.getAllPropertiesIterator()>
      <#if property.name != idField && !c2h.isCollection(property)>
        <td><@outputValue property=property
        varName=varName/></td>
      </#if>
    </#foreach>
.....
```

Another user page handles addition of a new object and update of an existing object. It contains an HTML form with HTML input elements for managing data. One element corresponds to one attribute of the domain object and it is specific to the attribute data type. For example, the attribute that represents a reference to another domain object is displayed as a drop down list.

All mappings between attribute types and HTML elements are presented in Table 2.

TABLE II. MAPPING BETWEEN ATTRIBUTE TYPES AND HTML ELEMENTS

Attribute type	HTML element	Validation
String	<input type=”text”>	
Integer, Long, Double, Float	<input type=”text”>	Number format validation
Date/Time	<input type=”text”>	Date format validation
String, length >100	<textarea>	
Boolean	<input type=”checkbox”>	
Referenced object	<select>	

A definition of the HTML element for referenced object is displayed in the following fragment of the template:

```
<spring:bind path="${property.name}">
  <#noparse>
    <select name="${status.expression}">
  </#noparse>
      <option value=""><spring:message code="global_select" /></option>
      <c:forEach items="<#noparse>${</#noparse>${parentList}}"
                  var="${parentName}">
        <option value="<#noparse>${</#noparse>${parentName}.${idField}}"
        <c:if test="<#noparse>${</#noparse>${parentName}.${idField} ==
        status.value}">selected="selected"</c:if>>
        <#noparse>${</#noparse>${propertyNameDisplay}}
        </option>
      </c:forEach>
    </select> .....
```

The generator also enables displaying dependent objects, in tabular form, on the page for adding new or updating a superordinate object. It is possible to create and update dependent objects directly from this page. This kind of display of dependent objects is conditional. When starting the generator, by setting an appropriate parameter, a user can select whether dependent objects will render this way or on the separate page.

Design of the generated user interface is quite simple but can be improved using external style sheet file which is also generated and contains the basic design elements of the user interface. Furthermore, all text labels are exported into a separate resource bundle file with default values set. This supports easy modification of text labels and an internationalization of the application. User interface page for displaying data of the Employee domain object is shown in Fig. 5.

5) Spring configuration file

For the Spring application presented, there is one configuration file (dispatcher-servlet.xml) which manages all system components. Fig. 6 shows the structure and functions of this file applied to the components of the Company domain object.

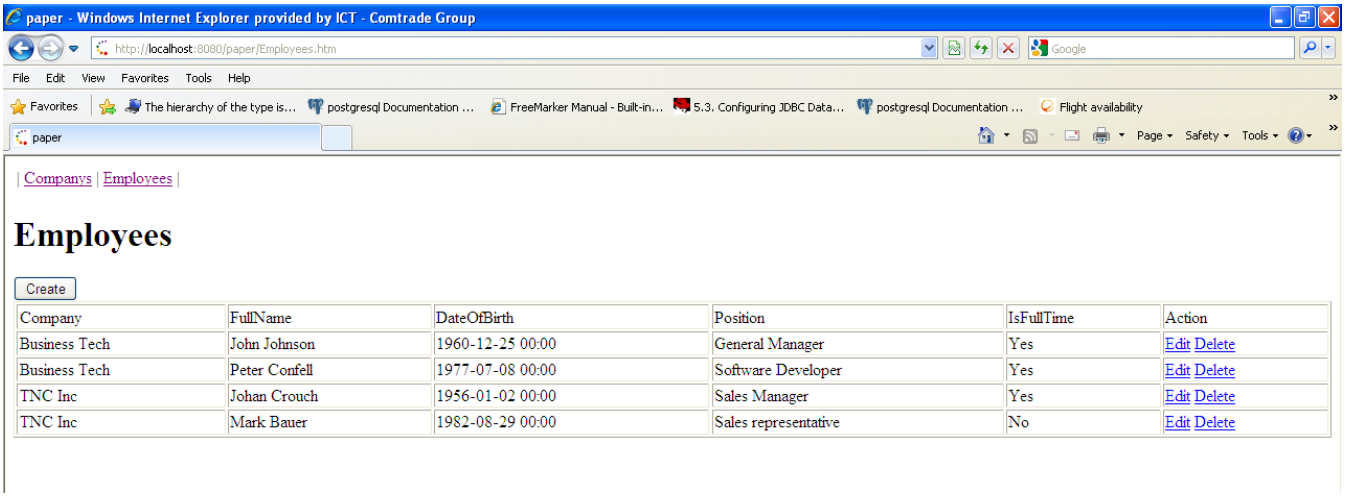

Figure 5. User interface page for displaying data of the Employee domain object

For every additional domain object in the application, six more beans in dipstchaer-servlet.xml file have to be defined for configuring and managing lifecycle of its components.

The application generator simplifies this procedure by introducing a template for this xml file that abstract beans for each component type of the system are defined in. After the template is processed with specific data model, appropriate bean definitions for each domain object will be generated. In the next fragment of the template for dispatcher-servlet.xml file, an abstract definition for bean that manages Spring 'edit' controllers is presented.

```
<#foreach pojo in c2j.getPOJOIterator(cfg.classMappings)>
    <bean id="edit${pojoName}Controller"
          class="${packageController}.Edit${pojoName}Controller"
          p:sessionForm="true" p:commandName="${varName}"
          p:commandClass="${packageEntity}.${pojoName}"
          p:formView="edit${pojoName}"
          p:successView="redirect:${pojoName}s.htm"
          p:${varName}Manager-ref="${varName}ManagerImpl" p:validator-
          ref="${varName}Validator"                    .....
```

All designed templates and their relationships that are included in the generator are displayed in Fig. 7.

B. Starting the generator

Functionalities of the HibernateTools toolset are used through Apache Ant build.xml file [17]. In this file, *hibernatetool* task for generation of different artefacts is defined and realized with the class org.hibernate.tool.ant.HibernateToolsTask.

For each template of the generator, one <hbmtemplate> exporter is designed. HibernateToolsTask creates JDBC configuration based on configuration data loaded from hibernate.cfg.xml file. In this file the connection to the desired database is set. JDBC configuration then creates a meta model from the database and every <hbmtemplate> exporter supplies its template with this meta model. As a result of the execution of one <hbmtemlate> exporter, one file per domain object is created. A part of the build.xml file used for generation of DAO classes is shown in the following fragment:

```
.....
<taskdef name="hibernatetool"
            classname="org.hibernate.tool.ant.HibernateToolTask"
            classpathref="toolslib" />
<target name="generate" depends="init">
   <hibernatetool templatepath="${template_path}">
   <jdbcconfiguration configurationfile="conf/hibernate.cfg.xml"/>
   <hbmtemplate template="java/ObjectDaoImpl.java.ftl"
                     filepattern="{class-name}DaoImpl.java"
                     destdir="${folder.app.src.java.db}">
      <property key="jdk5" value="true" />
      <property key="ejb3" value="true" />
      <property key="packageDao" value="${packageDao}" />
      <property key="packageEntity" value="${packageObject}" />
      <property key="packageDaoCommon"
                  value="${packageDaoCommon}" />
   </hbmtemplate>
.....
```

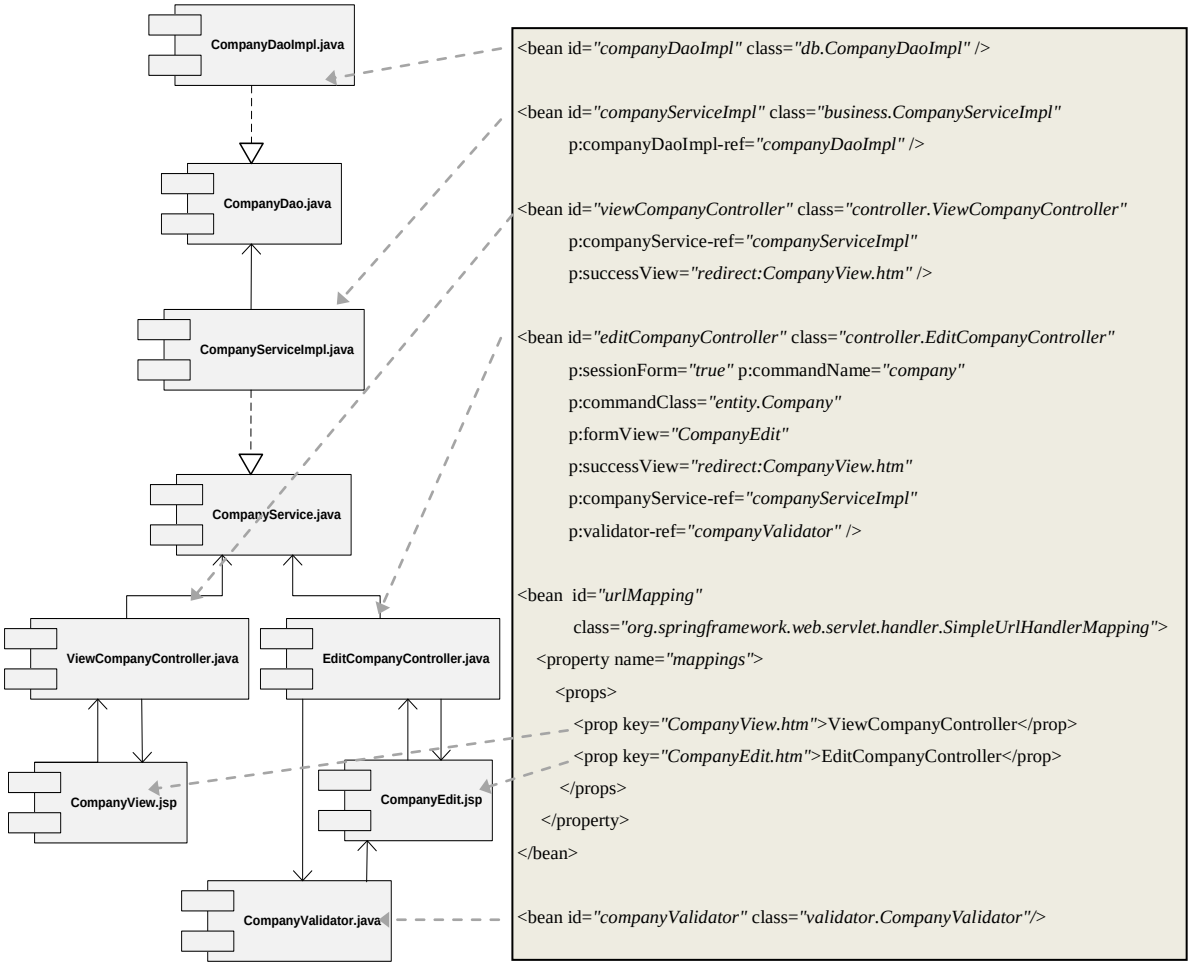

Figure 6. Structure and functions of the dispatcher-servlet.xml file

Besides code generation task, some other tasks are also defined in the build.xml file:

- creation of appropriate directory structure where generated source code files will be located
- compilation of Java source code files and creation of directory structure suitable for web application
- packing of the created structure into a war file

The generator may be started by invoking the described Ant file from the command line as shown bellow:

```
> ant build.xml
```

After the generator is executed, source files are generated and can be imported in an appropriate IDE and further improved. Moreover, a fully functional application is generated in the form of a war file which is ready for deployment on an application server.

Generated directory structure suitable for packing into a war file is shown in Fig. 8 [18].

V. SUMMARY AND FUTURE WORK

In this paper, development of a Spring application generator for a specific application was presented. Problems that occur when coding an application manually are described, as well as a possible solution that the generator would offer. Then the generator architecture was presented along with the process of designing templates. At the end, the procedure of generator starting was presented.

The generator presented uses a specific database as a variable input. It was constructed using two main components, HibernateTools toolset, which is responsible for producing data model using the database and Freemarker templates which are in charge of generating appropriate application components. The configuration and connection of the main

generator components are integrated into an Ant file which is also responsible for starting the generator.

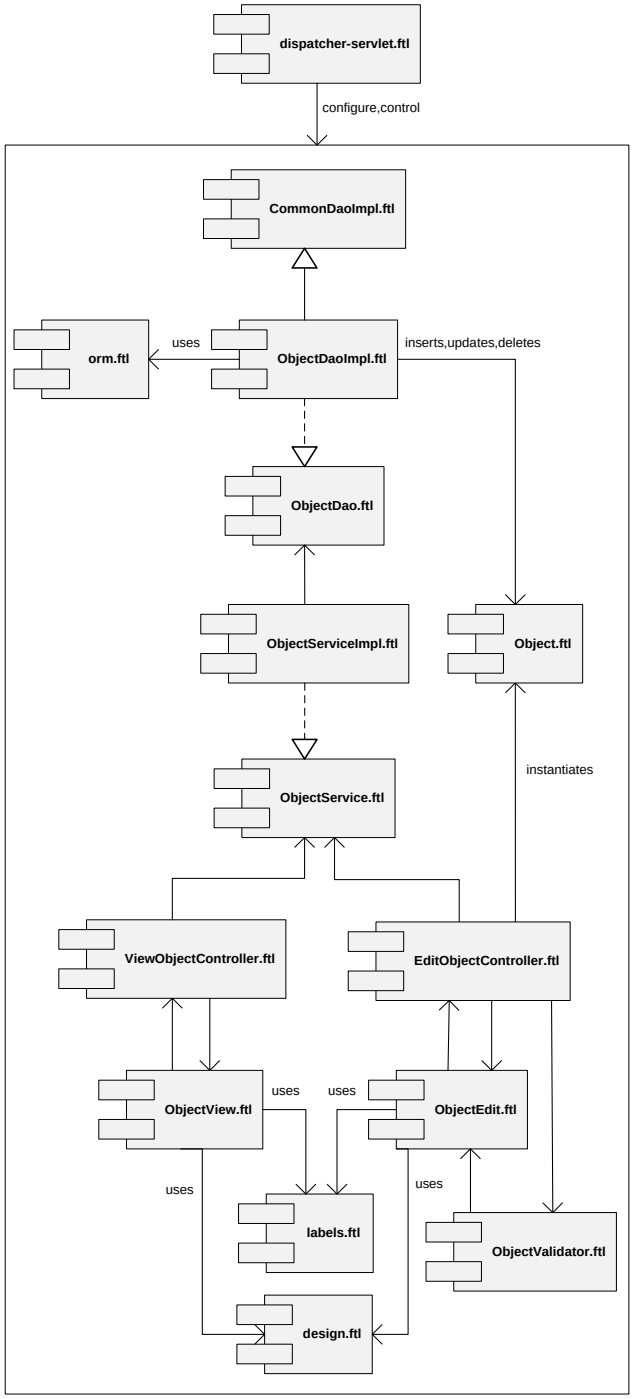

Figure 7. Designed templates for generation of a Spring web application

The generator does not have a rich user interface; it can be started from the command line, by calling Ant instruction. Nonetheless, it is still in accordance to its main purpose – code generation for a specific application development, not commercial exploitation of the generator. Moreover, some advanced customization of the process of code generation was omitted in favour of the generator simplicity. This leaves space for the future improvements.

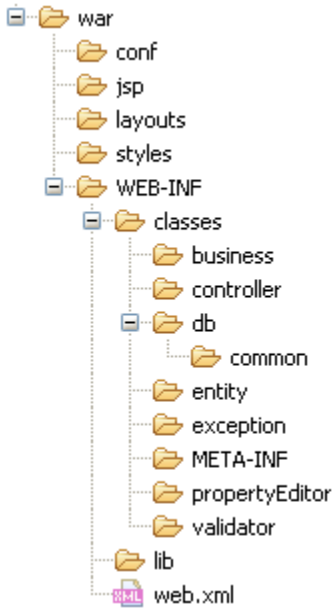

Figure 8. Directory structure suitable for packing into a war file

After the generator successfully finishes its job, the task of user interface design can be approached, thus enabling visual settings and control of a code generation process. In addition, a more advanced customization can be added which will allow better flexibility of the generator. Due to incomplete knowledge of all application details at the moment of designing the generator, many functionalities were not covered by the designed templates and they were implemented in the application manually. It will be possible to generalize these functionalities and insert them into existing templates. This will enable the generator to satisfy needs of future projects with fewer changes.

VI. Conclusion

On the software market there are many generators which provide for plenty of parameters and customizations in order to respond to specific requirements of an application to be generated. Nevertheless, there are cases when they do not suite project requirements sufficiently or they are not cost-effective in terms of the price or time needed for their customization. In those cases, design of an own generator should be considered.

The generator presented is developed for that purpose. Based on a practical experience in designing generator for a commercial application, it presents a possibility for fast development of a custom generator as a solution for a specific problem. The generator relies on the HibernateTools toolset which is responsible for a meta model creation. Thus most of the time was spent on designing templates which implement specific technologies and structure of an application.

Development of the templates lasted for about 3 weeks. Smaller part of the work included design of the Ant file which managed the process of application generation. It took 3 more days.

The main characteristics of the presented generator are simplicity of its design, short time spent on its development and full suitability to application specific requirements. Despite its simplicity, the generator allows some customizations and controls of the code generation process. It is primarily enabled by using functionality of the hibernate.reveng.xml file.

Testing the generator on *The Asset Management System* database which contains 400 tables, 200 functionalities based on 200 dictionary tables were fully generated without need for additional customization. Coding these functionalities manually would require approximately 800 hours, assuming that 4 hours are required for coding and testing one simple functionality. Since the enterprise applications have (or should have) more or less uniform code structure and visual appearance, the generator was able to produce basis for more complex functionalities, which were available for later easy upgrading. Some of the specific functionalities such as log in page and supporting program logic could not be produced by the generator and had to be developed manually.

ACKNOWLEDGEMENT

The authors are thankful to MNTRS for financial support grant number 174031.

REFERENCES

[1] J. Herrington, Code Generation in Action, Manning Publications Co., 2003.

[2] Iron Speed Designer, http://www.ironspeed.com/, accessed 2011.

[3] AppGini, http://www.bigprof.com/appgini/, accessed 2011.

[4] NConstruct - Intelligent Software Factory, http://www.nconstruct.com/, accessed 2011.

[5] Code On Time, http://codeontime.com/, accessed 2011.

[6] Rad Software: NextGeneration, http://www.radsoftware.com.au/codegenerator/benefits.aspx, accessed 2011.

[7] JAG - Java Application Generator, http://jag.sourceforge.net/, accessed 2011.

[8] Skyway builder Overview, http://www.skywaysoftware.com/products/builder, accessed 2010.

[9] C. Bauer and G. King, Java persistence with Hibernate, Manning Publications Co., 2007.

[10] M. Andersen, O. Chikvina, and S. Mukhina, Hibernate Tools Reference Guide, JBoss.org Community Documentation, 2008.

[11] C. Walls and R. Breidenbach, Spring in Action, Manning Publications Co., 2005.

[12] Jboss Enterprise Middleware, www.jboss.com, accessed 2011.

[13] K. Fertalj, M. Brcic, "A Source Code Generator Based on UMLSpecification", International journal of computers and communications, Issue 1, Volume 2, 2008

[14] ComTrade, http://www.comtrade.com, accessed 2011.

[15] S. Lazetic, Development of a Spring applications generator using Freemarker templates and Hibernate tools (Razvoj generatora Spring aplikacija primenom Freemarker sablona i Hibernate okvira), Master thesis, 2010. (in Serbian)

[16] Revusky, J., Szegedi, A., Dékány, D.,: Freemarker manual, http://freemarker.sourceforge.net, 2009.

[17] Loughran, S., Hatcher,E.,: Ant in Action, Manning Publications Co., 2007

[18] Yoshida, Y., Coward, D.,: Java Servlet Specification Version 3.0, Sun Microsystems, Inc. , 2003.