Generating Java EE 6 Application Layers and Components in JBoss Environment

Gábor Antal, Ádám Zoltán Végh, Vilmos Bilicki
Department of Software Engineering
University of Szeged
Szeged, Hungary
{antalg, azvegh, bilickiv}@inf.u-szeged.hu

Abstract—**Nowadays, prototype-based development models are very important in software development projects, because of the tight deadlines and the need for fast development. Prototypes can be very efficient in the analysis, validation and clarification of the requirements during the development iterations. Model-driven development and code generation are frequently used techniques to support the fast development of application prototypes. There are many recurring tasks in the iterations of a prototype development process, which can be performed much faster using high level models and the automatic generation of application component implementations. The goal of this paper is to review model-driven development and code generation methods and tools, which can be used to create Java Enterprise Edition 6 applications. Besides, this paper introduces a new code generation tool, which uses the JBoss Forge framework for generating application layers and components based on the entities of the application.**

Keywords- prototype development; model-driven engineering; code generation.

I. INTRODUCTION

There are many software development projects at present where it is very hard to collect all the requirements at the beginning of the project. The customers can be uncertain or they cannot visualize the requirements in detail and precisely. Therefore, the classical waterfall development model does not work in these projects. The requirements specification and the system plan need more phases to validate and clarify, which need active communication with the customer. Therefore, prototype-based development models can be useful in these cases [1]. Using these models it is possible to show a functional application prototype to the customer to validate and clarify the requirements. But very fast prototype implementation is needed for effective prototype-based development processes. For this purpose, model-driven development and code generation methods are used to speed up the implementation. However, it is important to examine and use software design patterns to ensure the maintainability of the application.

The goal of Model-Driven Development (MDD) [2] is to simplify the system design process and increase development productivity by reusing standard software models and design patterns. The developer defines the components of the system with their relations, the structures and relations of the data to be managed, and the use cases and behaviors of the system using high level models. Based on these high level representations the model-driven development tools can automatically produce lower level system models, and in the end of the process they can generate implementations in specific programming language. Transitions between the different model levels and the generation of the implementation from the models are performed by predefined automatic transformations. The model-driven development process supports the regeneration of the implementation after modifying the system models.

A very important tool of model-driven development is code generation, which is used in the implementation phase [3]. The goal of code generation is to automatically create a runnable implementation of the system in a specific programming language based on the initial high level system models. Using this technique the production of prototypes can be accelerated significantly. The usage of different design patterns can cause a lot of recurring manual implementation tasks, which can be supported with code generation effectively. The generated source code is supported by expert knowledge, so the maintainability of the code can be higher compared to the manual implementation.

However, the existing code generation solutions have some main disadvantages and deficiencies, which make them unable to use effectively for generating Java EE 6 [4] applications. Our goal is to introduce a new code generation toolkit, which eliminates these drawbacks and generates application prototypes based on Java EE design patterns.

This paper provides insights on some methods, which can be used for effective development of Java EE (Enterprise Edition) 6 application prototypes. Section II examines some related work in the area of code generation and shows the disadvantages of the existing generator toolkits. Section III presents the JBoss Forge [5] framework, which can generate Java EE 6 application prototypes with its Forge Scaffold [6] extension. Section IV describes some practical methodologies for code generation using some tools from the JBoss Forge framework. Section V introduces a Forge-based code generation toolkit, which eliminates the defects of the Forge Scaffold and provides more application components to generate. Section VI presents a case study about the usage of our generator toolkit in a telemedicine project. Section VII concludes the paper and discusses some interesting problems that are targeted to be further studied and improved in the future.

II. Related Work

During model-driven development processes the high level system models are often created using Unified Modeling Language (UML) [7], mostly based on class-, sequence-, activity- and state diagrams. The programming language of the implementation is mostly Java, C# or C++. The GenCode toolkit [8] generates Java classes from UML class diagrams, which include fields, getter and setter methods, default constructors, but it can generate only stubs for non-trivial methods. Other UML-based model-driven development tools, e.g., Modelio [9], ObjectIF [10], IBM Rational Rose [11] and Rhapsody [12], have the same drawback. Some toolkits introduced in [13][14][15] papers can generate non-trivial methods based on sequence- and activity diagrams. However, these tools need these diagrams for each non-trivial method and they cannot use generic diagram templates for generating pattern-based applications.

Some MDD tools define their own modeling languages, e.g., Acceleo [16] and ActifSource [17], but they do not have any metamodels for Java EE code generation.

A part of the source code can be also a model. A great example for this statement is the entity layer implementation of an application, which defines the data structures managed by the system. The entity implementations can be used to generate Create-Read-Update-Delete (CRUD) user interfaces for the application. Entity-based code generation tools are, e.g., JBoss Forge, seam-gen [18], MyEclipse for Spring [19], and OpenXava [20]. In the case of these toolkits, the generated user interfaces are in some cases incomplete, inaccurate, or difficult to use, e.g., inheritance, association between entities.

III. The JBoss Forge Framework

One of the most important entity-based code generation toolkits is the JBoss Forge framework, which is developed by the JBoss Community. The main goal of this tool is to manage the lifecycle and configuration of Maven projects [21] and to support the automatic generation of some application components with some simple commands. Forge has a command line interface, which includes most of the important Linux file manager commands. These commands are extended to read and modify the structure of Java source files. Besides, there are commands implemented for creating and configuring projects, and creating Java Persistence Application Programming Interface (JPA) entities. The Forge commands can be run batched in a script to make the use of the framework easier.

There are many available and downloadable plug-ins for Forge, extending its functionality. It is also possible to develop new plug-ins to support new code generation capabilities. A very useful plug-in included by the framework is Forge Scaffold, which can be used to generate basic Java EE 6 prototypes with CRUD web user interfaces based on the JPA entities of the application. The Scaffold plug-in can be extended with several scaffold providers, which can be used to specify the UI component library for generating user interface layer, e.g., standard Java Server Faces (JSF), RichFaces, Primefaces. Unfortunately, the Forge Scaffold plug-in has some defects.

- If the entity layer contains inheritance between entities, the user interfaces of the child class do not contain the fields of the super class.
- If a field has a @ElementCollection annotation, the edit page does not contain any component for selecting/adding elements to the field.
- In case of associations between entities, the edit page contains a dropdown menu for selecting/adding elements with the return value of the toString() method of the related entity instances. It is not possible to select a field as label.
- List pages do not provide ordering and filtering by multiple criterions.
- The generated code is difficult to maintain. The business logic is generated into one Bean class per entity. There is no separate Data Access Object (DAO) layer.

IV. Code Generation Methodologies

The Forge Scaffold plug-in uses the combination of two main methodologies for generating source code: template-based and procedural code generation. The tools for these code generation methods are provided by the JBoss Forge framework, so new code generator plug-ins can be developed using these tools.

The concept of template-based code generation methodology is to produce source code based on predefined templates, which will be rendered depending on the generation context. These templates consist of two types of parts: static parts, which will be generated into every source code instances; and dynamic parts, which will be replaced depending on the different inputs of the generation process. For supporting template-based code generation, JBoss Community provides the Seam Render tool, which is also used by Forge Scaffold plug-in [22]. The syntax of Seam Render code templates is quite simple. The static code parts should be specified the same way as they should appear in the generated code instances. The dynamic code parts should be indicated by expressions with named objects given in the form @{indicatorExpr}. The template-based code generation process consists of three phases:

1. Compiling the template: the specified template is processed and the dynamic template parts are collected with their object names and positions.
2. Defining the rendering context: every object names in indicator expressions are mapped to a Java object (mostly a String) for replacing every occurrence of the proper object name with the value of the object.
3. Rendering the template with the given context.

This method can be efficiently used in cases when the source code to generate contains small dynamic parts, which can be rendered by simple replacements, e.g., DAO classes, Beans for business logic. But, there can also be cases when the source code to generate contains “highly dynamic” parts, which can be rendered using very complex replacements (e.g., user interfaces) or when a given existing source code file must be modified in a later phase of the generation process. In these cases the procedural code generation methodology can be useful.

The procedural code generation aims the production of code elements using libraries for parsing and representing the elements of the specified language. For generating Java source code procedurally, Forge provides the Forge Java Parser library, which can parse Java source code to its own high level representation, create new code elements (classes, interfaces, fields, methods, etc.) or modify existing elements in the source code [23]. For generating XHTML (eXtensible Hypertext Markup Language) elements of the web user interface, the Metawidget framework can be useful, which contains the high level representation of standard XHTML tags, and JSF components [24]. Using these tools, new generator components can be developed to produce highly dynamic code parts and extend existing source code with new elements.

V. FORGE-BASED CODE GENERATION TOOLKIT

For eliminating the defects of Forge Scaffold and extending its functionality with the generation of other application components, we aimed the development of a new code generation toolkit based on the JBoss Forge framework. We analyzed some of our software development projects to find application layers and components, which contain a large amount of repetitive work that can be replaced with code generation methods to accelerate the development process. We found nine general areas that can be effectively supported with code generation. We examined these areas and their design patterns, collected the knowledge and experience we have in relation to the development of these components and implemented a code generation toolkit for supporting the effective production of them. In the following subsections we discuss these areas in detail.

A. Entity and Validation Management

The entity generator of the Forge framework is a very useful tool but it has some defects:

- It is not possible to generate inheritance between entities.
- It cannot generate enum types with specific values.
- It is not possible to generate custom constructors, hashCode, equals and toString methods based on specific fields.

We implemented these capabilities to extend the functionality of the entity generator. Besides, Forge cannot generate validation annotations to the entity fields, so we extended our toolkit with the possibility to add annotations of Bean Validation API (Application Programming Interface) to the entities. The generated entities use the Composite Entity design pattern to implement associations between the entities.

B. DAO Layer

The DAO layer of the application is responsible for the operations, which can be made on the entities of the system: inserting, updating, deleting, querying. Using JPA-based EntityManagers it is easy to generalize the DAO layer based on the Data Access Object design pattern. We created a generic DAO superclass for providing simple EntityManager operations and the entity-dependent DAO child classes extend this superclass to provide queries and entity-specific operations. Therefore, only the child classes should be generated depending on the entity based on a DAO template. We used Hibernate as JPA provider in our generator toolkit.

C. Business Logic Layer

The business logic layer of an application is responsible for establishing connection between the user interface and the entities (using the DAO layer), and managing complex business processes and transactions. There are two main components in general, which are used by CRUD user interfaces:

- Data Models: provides the listing of specific type of entities with paging, ordering and filtering. They use the Session Façade and Value List Handler design patterns.
- Entity Action Beans: provides editing, viewing and deleting of a selected entity instance. They use the Session Façade design pattern.

These business logic components are also easy to generalize using generic superclasses. Only the entity-specific Data Models and Entity Action Beans should be generated depending on the entity based on code templates. We used Contexts and Dependency Injection (CDI) for context management and dependency injection, and Enterprise JavaBeans (EJB) 3 for transaction management in our generator toolkit.

D. CRUD User Interfaces

The CRUD user interfaces provide simple listing, creating, editing, viewing and deleting capabilities for the application user. Simple UI components (text fields, labels, dropdown menus, etc.) can be inserted using JSF 2 components. Complex UI components (list items, fields with labels, complex selector components, etc.) can be efficiently generalized using JSF 2 composite components, which can simplify the generation of complex user interfaces. These components use the Composite View design pattern. We also eliminated the defects of Forge Scaffold in our user interface generator. The generation of user interface components is performed by procedural code generation and the result is inserted into predefined XHTML page templates.

An example list user interface generated by our toolkit is shown on Figure 1. The layout and content of list items are generated using the structure of the entity classes.

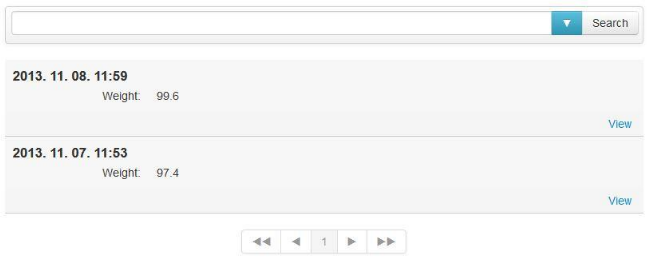

Figure 1. List user interface generated by our toolkit

E. Authentication, Authorization, Auditing

The system needs authentication and authorization modules to provide appropriate data security. Users need to be authenticated before using the system and it is very important to check the permissions of the actual user before data access or modification. These modules can be the same in most of the applications. We use the PicketLink [25] security and identity management framework in our generated security module, which provides a general entity layer pattern and JPA-based solutions for identity and role management. In our security module, we also implemented PicketLink-based registration, password change and password reminder components, which can be automatically generated for the given application.

The logging of the operations made by application users is also a very important data security aspect, which is called data auditing. Our code generation toolkit can produce an auditing module for the application, which is capable to log every data operations with the name of the performer user and the actual timestamp. The auditing module uses Hibernate Envers [26] in a combination with our auditing solutions.

F. REST Interfaces

Mobile client applications often connect with a server application for data querying or uploading. Therefore, both the client and the server application need interfaces to communicate with each other. The most frequently used methodology of client-server communication is the Representational State Transfer (REST) architecture. For transferring data between the client and the server, the JavaScript Object Notation (JSON) and eXtensible Markup Language (XML) formats are often used. Our toolkit can generate common source files for REST-based communication (both for client and server) and REST service stubs for server application based on DAO methods. Data objects transferred between server and clients are implemented using the Data Transfer Object (DTO) design pattern. The generated service implementation includes DTO classes based on entities and converters between entities and DTO classes. Our solution uses the RESTEasy library for REST implementation and Jackson library for JSON implementation.

G. Data Visualization on Charts

In server applications there can be a lot of numerical data (e.g., measurement data, quantities), which should be visualized on charts to analyze their changes. Our toolkit is capable to generate chart visualization user interfaces for specified numerical data over the changes of a specified entity field (e.g., measurement date). A generated example chart user interface is shown on Figure 2. We use the Flot [27] JavaScript-based library for visualizing charts, FlotJF [28] library for representing Flot objects in Java, and RESTEasy for the REST services, which provides data for charts.

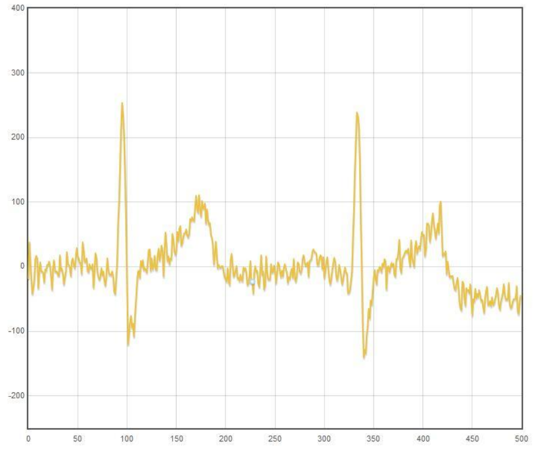

Figure 2. Chart user interface generated by our toolkit

H. Demo Data Generators

During the development the system needs a lot of demo data for testing. Demo data can also make presentation of the application to the customer easier. Therefore, demo data generator components are needed in the system, which can be very time-consuming to implement in case of large domain models. Our toolkit can generate demo data generators for the selected entities in the application. The constraints of entity fields are also taken into consideration during the generation. It is also possible to configure the number of entities to be generated before the production of data generators.

I. Dashboards

Many applications may need special, customizable user interfaces, which can show only the most important information for users, with simple and clear visualization elements. Our code generator toolkit can produce dashboard user interfaces with customizable layout (tiles, which can be moved or hidden) and content (charts, gauges, switches, status indicators, etc.). A generated example dashboard is shown on Figure 3. The dashboard is created using Gridster jQuery-based library [29].

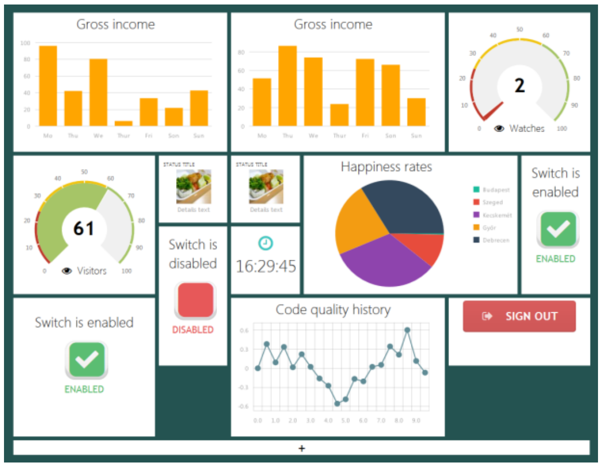

Figure 3. Dashboard user interface generated by our toolkit

VI. Case Study: Development of a Telemedicine Application

We used our code generation toolkit successfully in a telemedicine project where we had to develop a data collector application for supporting the long distance monitoring of patients after heart surgery. The task of server application was the collection, storage, and the visualization of the incoming data from different sensors, e.g., weight scales, ECG sensors, blood pressure monitors.

When planning the mentioned application we analysed the components to be developed and the results showed that a significant part of the development can be supported with code generating methods. Relying on this, the prototype development of the patient monitoring system with our code generating tool was started. The following components of the application were generated:

- DAO (Data Access Object) and business level layer.
- Listing, editing, visualizing user interfaces.
- Charts for the visualization of ECG measurements and for the representation of the temporal changes of blood pressure and weight.
- Data uploading REST service stubs on the server side and REST client stubs in the mobile client application.

Above this, only the refinement of user interfaces and the implementation of REST interfaces were required. Furthermore, for integrating sensors, Android mobile prototypes were developed, whose task was to receive and process the data sent by a single channel ECG sensor, a blood pressure monitor, and a weight scale.

We measured the time requirements at this application with our productivity measurement plug-in for Eclipse IDE, and compared them with the time needed by developing this application manually [30][31]. The measurement results can be examined in Table I. t_m means the manual implementation time (in person days), and t_{cg} means the implementation time with code generation support (in person days). The application layers needed about 80% less time in average to implement with code generation support, compared to the manual development.

TABLE I. Comparison of development time in manual- and code generation supported development

Layer	t_m	t_{cg}	t_{cg}/t_m (%)	Total time spent (%)
Domain model	0.1	0.02	20	5
DAO layer	3	1	33	10
CRUD user interfaces	10	0.2	2	50
Authentication, authorization, auditing	20	0.2	1	15
Data visualization	3	1	33	10
System integration interfaces	3	1	33	10

VII. Conclusion and Future Work

This paper reviewed model-driven development and code generation methods and tools for producing Java EE 6 applications, and introduced a code generation toolkit based on the JBoss Forge framework, which can generate many application components based on Java EE design patterns to accelerate the prototype development process. We also used this toolkit in the development of a telemedicine application and measured the efficiency of the code generation supported development compared to the manual development. The measurement results show that the development process with code generation needed significantly less time than the manual development.

In the future, we would like to continue the development of our generator toolkit and extend its functionality. We identified the following goals, which we would like to reach in the further development of the toolkit.

- Now, the generator works based on Forge 1.4.3. We would like to upgrade our toolkit to Forge 2.
- During the test phase of the development, unit tests provide the correctness of small, low level system units. The development of correctly functioning system units can be more effective with implementing unit tests for given units. We would like to extend our toolkit with the capability to generate unit tests for selected classes.
- An entity-based code generator toolkit can be easier to use if it can process UML class diagrams, which contain the entities of the application. These UML diagrams should be extended with some additional properties, such as JPA and Bean Validation annotations for entity classes and fields. Using these diagrams the Java implementation of entities can be automatically generated. We would like to develop an extension for our code generator toolkit to process extended UML class diagrams of entities as the input of application prototype generation.
- The main goal of our toolkit is to generate web-based server application components. In the future we would like to identify some areas in Android mobile application development, which can be supported with code generation and implement a generator toolkit for Android applications.
- Many entity fields can have special meanings that imply special criterions during the demo data generation and special validation processes. For example, an entity field with String type can be a first name of a person, an address or a country name. This semantic aspect should be taken into consideration during the data generation and validation of values in entity fields. Our goal is to provide an opportunity to add semantic annotations to entity fields, which are related to ontology definitions of special data structures, and to generate special data generators and validators for the annotated fields.

ACKNOWLEDGMENT

The publication is supported by the European Union and co-funded by the European Social Fund.

Project title: "Telemedicine-focused research activities on the field of Mathematics, Informatics and Medical sciences"

Project number: TÁMOP-4.2.2.A-11/1/KONV-2012-0073

REFERENCES

[1] M. F. Smith, Software prototyping: adoption, practice, and management. McGraw-Hill, 1991.

[2] B. Sami, Model-Driven Software Development. Springer, 2005.

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

[4] http://docs.oracle.com/javaee/, [retrieved: 2014.04.11].

[5] http://forge.jboss.org/, [retrieved: 2014.04.11].

[6] http://forge.jboss.org/docs/important_plugins/ui-scaffolding.html, [retrieved: 2014.04.11].

[7] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Guide, Second Edition, Addison-Wesley Professional, 2004.

[8] A. G. Parada, E. Siegert, and L. B. de Brisolara, "GenCode: A tool for generation of Java code from UML class models", 26th South Symposium on Microelectronics (SIM 2011), Novo Hamburgo, Brazil, pp. 173-176.

[9] http://www.modeliosoft.com/, [retrieved: 2014.04.11].

[10] http://www.microtool.de/objectif/en/, [retrieved: 2014.04.11].

[11] http://www-03.ibm.com/software/products/us/en/ratirosefami/, [retrieved: 2014.04.11].

[12] http://www-03.ibm.com/software/products/us/en/ratirhapfami/, [retrieved: 2014.04.11].

[13] A. G. Parada, E. Siegert, and L. B. de Brisolara, "Generating Java code from UML Class and Sequence Diagrams", Computing System Engineering (SBESC), 2011 Brazilian Symposium, Florianopolis, Brazil, pp. 99-101.

[14] M. Usman and A. Nadeem, "Automatic Generation of Java Code from UML Diagrams using UJECTOR", in International Journal of Software Engineering and Its Applications vol. 3, no. 2, 2009, pp. 21-38.

[15] E. B. Oma, B. Brahim, and G. Taoufiq, "Automatic code generation by model transformation from sequence diagram of system's internal behavior", in International Journal of Computer and Information Technology, vol. 1, issue 2, 2012, pp. 129-146.

[16] http://www.eclipse.org/acceleo/, [retrieved: 2014.04.11].

[17] http://www.actifsource.com/, [retrieved: 2014.04.11].

[18] http://docs.jboss.org/seam/2.3.1.Final/reference/html/gettingstarted.html, [retrieved: 2014.04.11].

[19] http://www.myeclipseide.com/me4s/, [retrieved: 2014.04.11].

[20] http://openxava.org/home, [retrieved: 2014.04.11].

[21] http://maven.apache.org/, [retrieved: 2014.04.11].

[22] https://github.com/seam/render, [retrieved: 2014.04.11].

[23] https://github.com/forge/java-parser, [retrieved: 2014.04.11].

[24] http://metawidget.org/, [retrieved: 2014.04.11].

[25] http://picketlink.org/, [retrieved: 2014.04.11].

[26] http://envers.jboss.org/, [retrieved: 2014.04.11].

[27] http://www.flotcharts.org/, [retrieved: 2014.04.11].

[28] https://github.com/dunse/FlotJF, [retrieved: 2014.04.11].

[29] http://gridster.net/, [retrieved: 2014.04.11].

[30] G. Kakuja-Toth, A. Z. Vegh, A. Beszedes, and T. Gyimothy, "Adding process metrics to enhance modification complexity prediction", Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension (ICPC 2011). Kingston (ON), pp. 201-204.

[31] G. Kakuja-Toth, A. Z. Vegh, A. Beszedes, L. Schrettner, T. Gergely, and T. Gyimothy, "Adjusting effort estimation using micro-productivity profiles", 12th Symposium on Programming Languages and Software Tools. SPLST'11. Tallinn, Estonia, 5-7 October 2011, pp. 207-218.