Specification of Document Structure and Code Generation for Web Content Management

Besnik Selimi[1] and Artan Luma[2]

**[1] Contemporary Sciences and Technologies, South East European University
Tetovo, 1200, Macedonia**

**[2] Contemporary Sciences and Technologies, South East European University
Tetovo, 1200, Macedonia**

Abstract

This paper presents a simple model for declarative specification of the structure of documents for web-based content management. The proposed model allows the description of the hierarchical structure of multi-lingual documents, relationships, and organization among documents of same type. We specify this model in a way that allows building specifications using drag-and-drop interfaces. Then, we use these specifications in order to automatically generate the necessary database schemas and code for managing these documents. The final goal is to provide unobtrusive automatic code generation that is strongly based on widely used design patterns and thus fits into common workflows in web application development. The usage of such models should further reduce development time in Rapid Application Development processes, especially by shortening the time from gathering requirements to having an executable application.

Keywords: *Content Management, Code Generation, Hierarchical Documents, Models, Multi-language Content, Requirement Specification.*

1. Introduction

While developing software, standard solutions of common problems are identified as design patterns [1]. These patterns represent a formalization of best practices to follow when solving application or system design problems. Reusing design patterns helps to speed up the development time and provide well tested development paradigms that help in preventing common errors. These are among the main reasons that after some time, particular design patterns become de-facto standards in application development.

Besides the advantages that design patterns provide to a programmer, following a pattern introduces a significant part of systematic work – producing the necessary code for applying the given pattern. Providing appropriate techniques, tools, and libraries for reducing such systematic work is one of the biggest challenges in software engineering [2].

One of the directions in alleviating the amount of work is to provide frameworks that include the code for the implementation of the most common design patterns, thus minimizing the code written by the developer. As an example, the Model-View-Controller (MVC) [3][4] application pattern or variants of the same are used in a multitude of domains and many frameworks implementing this pattern nowadays are available. Active record [5] is another frequent design pattern that provides a mapping of objects to relational databases. Among others, such patterns are nowadays ubiquitous in web application frameworks.

Another important direction that helps reducing the systematic work is the Model-Driven Engineering (MDE) [6] [7] approach that focuses on defining models of the system, including the business logic and other domain-specific information, in an abstract level, and base the subsequent development on these models. The ultimate goal of this approach is to provide tools that are able to automatically generate the code based on the provided formally-defined model [8] [9]. The automation allows the development team to focus on the domain and the primary reasons for creating the software system. Model-driven development methods usually require longer time from gathering requirements until obtaining executable models. Nowadays, web application oriented development teams frequently use agile methods [12], the main reason being that customers want to see and accept a finalized product instead of a specification.

Yet another model–based approach is to abstract common knowledge in a particular domain and provide Domain-Specific Languages (DSL) [10]. The primary advantage of using a DSL is in the fact that it is easier for a domain specialist to understand because it manipulates concepts

that are known to them. Creating a DSL requires expertise in both the application domain and programming for providing the underlying tools that help in the execution of such models [11].

In this paper, we present a simple domain-specific language for the specification of the structure of documents used in the domain of web applications. The main idea is to abstract the most frequently used concepts into a model, and then use such models to generate the database schema and code for managing document instances. Although such models can be dynamically interpreted, we focus on code generation since it allows developers to easily customize and extend the code using standard object-oriented techniques. Furthermore, we generate code that follows common design patterns and thus fits into current frameworks and development practices. We do not claim to provide a general model that can describe any document but rather provide a simple model that can accommodate a large range of web applications while allowing programmers to further customize it for specific requirements.

The rest of this paper is organized as follows. Section 2 provides a background on the current approaches in the development of web applications. Section 3 presents the approach discussed in this paper. Section 4 provides a succinct specification of the model we use. Section 5 discusses some of the most important issue related to the implementation of the approach. And finally, section 6 concludes and gives directions on future evolutions of the presented approach.

2. Problem definition

Management of content has been an increasing concern since the beginning of the web. From a collection of interlinked static documents, nowadays the web has evolved into a full featured application platform. A large part of the development has shifted from conventional desktop to web and mobile applications. These applications manipulate content that needs to be precisely structured, semantically meaningful and available in an increasing number of different formats. As a consequence, web content management cannot be reduced to managing a set of pages, articles, or similar, which is already covered by a multitude of Content Management Systems (CMS) that are available.

Generally, a CMS provides a lot of commonly required features for a website. But, the rigid structure of such systems cannot easily accommodate the demand for customized data types and interfaces. Some systems provide form generators to allow users to define their data structures as a list of fields with a custom type. This handful feature is still not sufficient in many cases, because of their inherent linear structure, the fixed database schemas, and the fact that they do not allow programmatic customizations. These, together with reasons related to the distributed nature of cloud applications, are probably the main reasons that most of web applications are nowadays developed on a lower level, based on web framework libraries.

2.1 Web frameworks

The role of web application framework is to cover the usual overhead in developing websites, web applications and services. They provide a set of libraries that implement common features and help reducing the amount of code to be written. By using frameworks that are available nowadays and following suggested best practices, developers may be productive even ignoring important subtleties of web development such as the HTTP protocol, cookie and session management, security, and sometimes SQL.

Most web frameworks today are using a set of commonly accepted design patterns, such as active record and MVC. Other design patterns, such as inversion of control are making their way as standard features of web frameworks. Although these patterns standardize a significant part of the code and shift it into reusable libraries, we observe that there remains code that is produced systematically while implementing specific features.

One such situation that occurs very often is when there is a need to specify hierarchy between the attributes of a model. For example, a simple article may include more images or comments. A researcher's profile might include education, publication, work, and similar lists of records. In such cases, one needs to create separate tables in the database for each of these lists, and relate them with foreign keys. Then, a separate model class is needed for each of these tables. For editing such records, a more complex user interface needs to be created. The data sent to the server needs to be separated to corresponding tables.

Another situation that involves systematic code creation is managing hierarchies between models. Examples include categories, page hierarchies and similar. In such cases, additional columns should be added in the database, provide support for editing relationships and models should verify that these relationships form a correct hierarchy. Furthermore, by avoiding more complex solutions, the resulting implementations are often suboptimal.

A third situation that we consider does not get the required attention in web frameworks is the management of multilingual content. Most web frameworks offer the possibility of localizing the user interface. But, when it comes to actual content translations, they do not offer systematic solutions. Different known solutions can be found to the problem of storing translations, but programmers often need to manually specify the necessary additional tables or columns. Editing and storing different translations or retrieving correct translations produces a lot of code that is mere duplication.

Given such situations, we consider that there is a need for a higher level specification that includes support for describing such concerns in a succinct manner. We have considered existing standards for describing the document structure, such as XML Schema [12] or UML [13], in the perspective of using these as a specification language. In one hand, these offer a wide range of specification possibilities that makes them more complex, and require additional specification restrictions to allow efficient code generation. On the other hand, these would need to be extended for easier specification of some features we want to include.

3. A simple model based approach

In order to take into account the above situations, we have considered a model for specifying web documents and an approach for code generation based on this model. Figure 1 depicts the flow of this approach.

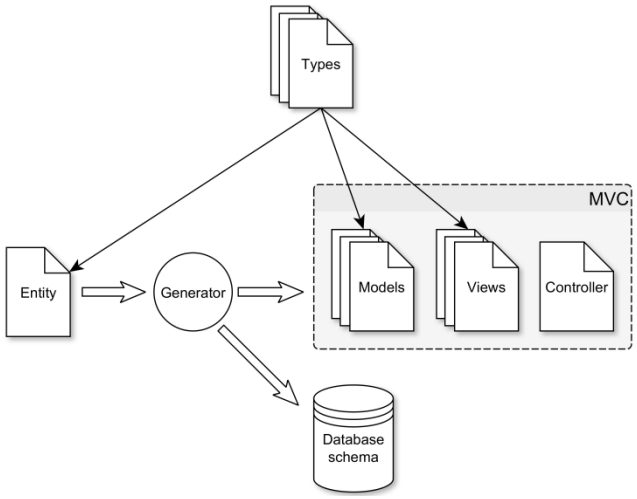

Fig. 1 Code generation based on entities.

The structure of documents is defined as separate *entities*. We provide a minimal DSL for specifying these entities. Although we have experimented with specifications in different languages, we present below an XML-based description as it is easier to read.

Entities are composed of *fields* that can be repeated and in turn be composed of other fields. Fields refer to *types* that are provided as separate reusable components. These configurable types provide the information about how to display, edit, sanitize, validate, and store a particular field.

Based on these entities, we generate the database schema and the code for CRUD operations based on the MVC pattern. Several models are generated to accommodate the hierarchical structure of entities. These models call corresponding types for validating each attribute. In a similar way, views call types to generate the form elements for corresponding attributes.

The approach based on code generation instead of interpretation, allows developers further extensions and customizations in derived classes.

4. Model specification

In order to present the model specification, we will use an example of a study program. The reason of choosing such an example is that it is sufficient to present the main concepts behind our model and we can compare this to an existing implementation of the same.

4.1 Entities

Each type of document is specified as a separate entity. An entity represents the higher concept in the hierarchy and includes all the relevant definition for a particular type of document. It is defined as follows:

```
<entity id="Program" name="Study Program">
   ...
</entity>
```

The required attribute id should be a unique identifier of the entity, since it is used in other entities to refer to this entity. It will be used as a class name inside the generated code and as a name for the corresponding table in the database.

The required attribute name is a human-readable name of the entity and will be used to identify this type of documents in the user interface.

4.2 Fields

Each entity is composed of a list of fields. A simple field can be specified inside an entity using the element *field* as follows:

```
<entity ...>
   <field id="name" name="Name" type="String" />
   ...
</entity>
```

The required attributes *id* and *name* have the same meaning as in the declaration of an entity – id is used as a name for the corresponding column in the table and as a variable name inside the code, while *name* is used as a label for the corresponding input element. These two attributes will be present in every element of the specification, since we need to refer to each of them in the code or the generated user interface.

The other required *type* attribute, specifies the type of the field. A *String* in this case specifies that the field name will be edited using a standard HTML input of type text and will be stored as a textual field in the database.

4.3 Types and Configuration

The type of a field is the first and one of the most important abstractions that the model provides. Namely, instead of using low-level data types such as the types accepted by database systems, we define types at a higher level that carry with them lot more meaning than simple types. For example, we want to be able to define types that correspond to an uploaded file, an image that is automatically redimensioned, third-party hosted video and similar. Therefore, types are also the main point of extension of the model since we allow types to be plugged-in as extensions that are reusable across different applications.

For better reusability, types can be configured upon usage on a particular entity instance. The configuration is provided as a list of parameters. In our example of a study program entity, we can use such configuration to constrain the allowed HTML tags in a WYSIWYG editor and constrain the value of ECTS to be inside a particular range of integers.

```
<field id="description" name="Description"
       type="RichText">
   <param id="allowed_tags"
          value="p,em,strong,i,b,br,..." />
</field>

<field id="ects" name="ECTS" type="Integer">
   <param id="min" value="60" />
   <param id="max" value="240" />
</field>
```

It is important to state that each type can define its own configuration parameters. Furthermore, the parameters are not always related to validation of data but can be used for any purpose. A parameter may specify the editor to use for a type that provides multiple editors, choose the type of the corresponding column in the database, or specify the dimensions of an image that needs to be automatically redimensioned after upload, or even specify the possible values for an enumerated field as in the following example:

```
<field id="Cycle" name="Cycle" type="Enumerated">
   <param id="options">
      <option value="undergraduate"
              name="Undergraduate"/>
      <option value="postgraduate"
              name="Postgraduate"/>
      <option value="phd" name="PhD"/>
   </param>
</field>
```

The previous example will generate a drop-down list for selecting one of the specified options, as shown in figure 2.

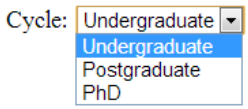

Fig. 2 Selection of values in enumerated fields.

In order to be functional, we have identified the following responsibilities that each type should fulfill:

- Provide a list of configuration parameters with default values
- Parse the configuration parameters from the entity specification replacing the default values
- Return the underlying storage type, based on its current configuration
- Return a default value for each type
- Produce the HTML code for displaying its content
- Produce the HTML form elements and necessary assets for editing such a field
- Convert the data from and to the underlying storage type
- Validate and adjust the data before saving

4.4 Multilingual fields

Another important abstraction in our model is provided by the concept of multilingual field. When managing multilingual content, one needs to provide translations of the documents in different languages. Generally, the textual data needs to be translated while non-textual data such as numbers and dates may be common to all the languages. But still, in some cases textual data such as an email does not need translation while an image might need translation because it may contain textual content. In order to accommodate these situations, we provide a mean of specifying particular fields as being translatable, no matter its type. This is done by adding a simple *translate* attribute.

```
<field ... translate="true" />
```

Although simple to define, this additional specification has important consequences. These implications are discussed in more detail in the next section.

4.5 Repeatable and optional fields

Repeating a particular field is another abstraction of the model that makes a fundamental difference. The main idea is that often we need to have multiple occurrences of a particular field. In general, this implies the creation of separate tables with a one-to-many relation. We abstract this concept, by simply indicating that a field can be repeated, as in the following example for attaching multiple images to a document.

```
<field id="images" name="Images" type="Image"
       repeat="true" min-occurs="2" max-occurs="5"/>
```

The attributes *min-occurs* and *max-occurs* specify how many times a field should and can be respectively repeated.

When editing a document, the above specification should produce a form that looks similar to the example shown in figure 3.

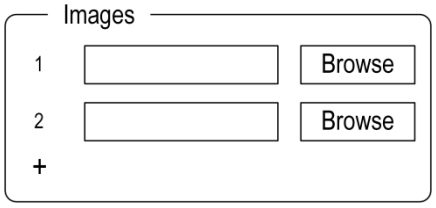

Fig. 3 Repetition of fields.

The two required occurrences are visible from the beginning, while the '+' sign will allow to dynamically add up to three more fields. When the maximum number of occurrences has been reached, no more fields can be added. Note that the same concept can be implemented using other UI approaches.

If a field is not required, we can specify it as being optional:

```
<field ... optional="true" />
```

Although optional fields may seem similar to repeating a field a minimum of zero and a maximum of one times, in essence optional fields have different meaning, they will be visible but can be left empty and no additional tables will be created.

4.6 Grouped fields

Often some fields conceptually belong together. We provide a group element to specify that some fields belong together. In a simple case, it allows us to display these fields grouped together in the user interface and will have no effect on the produced interface. In the general case, this concept is necessary when we want to repeat such a group of fields. We can consider a group equivalent to an embedded document.

Since groups can be nested into other groups, it allows us to define an arbitrary level of nested items. The following example specifies that in a study program, we can create from two up to six semesters, each containing an arbitrary number of courses.

```
<group id="semesters" name="Semesters" repeat="true"
       min-occurs="2" max-occurs="6">
   <group id="courses" name="Courses" repeat="true">
      <field id="name" name="Name" translate="true"
             type="String"/>
      ...
   </group>
</group>
```

Each course is defined as a group, since it is composed of multiple fields such as the name, description, number of credits awarded and similar.

It is clear that each time some field or group is repeated, we need to create a separate table in a relational database. The implied relationship type will be one-to-many. Additionally, such a relationship should be the equivalent of a composition, meaning that the lifetime of the records is tied to the lifetime of the document as a whole.

4.7 Relation fields

We already defined entities that specify all the parts that compose a document. We still need sometimes to refer to other types of documents defined by an external entity. This is made possible by the *belongsTo* element as shown in the following example:

```
<belongsTo id="department" name="Department"
           entity="Department" />
```

The example above defines that each program should be attached to one department. When editing, a field allowing the selection of a department should be generated. Departments will exist as separate documents since the list of departments needs to be dynamically managed and other parts of the system refer to the same list of departments.

Note that the only relationship type provided here is many-to-one. We can create many-to-many relationships by repeating the *belongsTo* element and even specify cardinalities if needed. We do not introduce one-to-one relationships, since we consider that such relationships should be avoided and the necessary fields should be embedded in a document. If absolutely necessary, such a

relationship could be provided as a special type that is used as a normal field.

4.8 Hierarchy

Documents belonging to a particular entity may be organized in different ways. In some cases, we need to allow manual ordering of instances. In other cases, we need to organize the documents in a hierarchical way as a tree. We provide for this a simple attribute *hierarchy* that can be added to the *entity* element.

```
<entity ... hierarchy="tree">
    ...
</entity>
```

In addition to the default arbitrary ordering, we have defined two particular hierarchies: *ordered* and *tree*.

An *ordered* hierarchy has the meaning that when displaying a list, a way for ordering the documents should be provided and a corresponding field for storing the position of each element should be provided in the corresponding table.

A *tree* hierarchy means that we need to provide a structure in the database for storing the parent-child relationships and provide a way for selecting a parent. Different approaches for representing trees are discussed in the next section.

Note that if we specified a relationship called parent within the same entity, we will obtain a tree-like structure. Such relationships still may be needed, but the hierarchy attribute provides the primary organization of documents belonging to the same entity and can be used in the user interface to show the list of documents as a tree. As an example, if we defined an entity *Page*, we can then organize these pages and automatically produce menus and URLs based on this hierarchy.

4.9 Automatic fields

We easily extend our system by creating special types that can be used for particular purpose. Such an interesting example is with the types *Created* and *Modified* that provide fields that are not visible while editing but generate the corresponding values when a document is saved.

```
<field id="modified" name="Modified"
       type="Modified" />
<field id="created" name="Created" type="Created" />
```

5. Experimentations

In order to experiment with the model, we have specified different models, among which the model of study programs presented earlier. Based on these models, we have generated the necessary database schema and MVC based code for creating and editing the records. We have experimented with integrating the code into different web application frameworks. This section discusses some of the most specific implementation aspects that we have encountered.

5.1 Types

The entities we defined are based on rich types that are provided as code. We have implemented several common types among which text input, integer input, rich-text editing, file upload and image upload and resize. These types provide code for the different responsibilities defined in the previous section. Defining such types might require a significant work, but the rationale behind is the fact that these types can be reused across applications, especially if they allow a larger amount of configuration.

5.2 Repeating fields and order

When a field is declared to be repeatable, we need to store multiple values for that field. In a relational database, a separate table with one-to-many relation needs to be created. In a previous example, we repeated semesters inside a study program. Furthermore, a semester is composed of a course that can be repeated. In order to implement such a pattern, we generate separate tables for semesters and courses as shown in the figure 4.

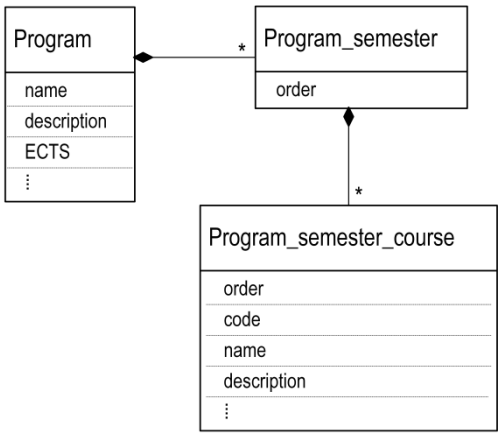

Fig. 4 Composition of repeatable fields.

Since there is an implicit order by which the fields are repeated, we also need to add a column storing the order of each repeated item. This order should be updated when adding or reordering repeated fields while editing the

document. When the corresponding records are retrieved from a database, they need to be ordered according to this field. For example, the list of courses should be returned according to the same order as edited, unless otherwise specified by customizing the generated code.

5.3 Translation

When a field is specified as translatable, additional space needs to be provided in the database for storing translations for each active language. When such a document is published, one should automatically retrieve the translated data according to the current locale. Moreover, in the user interface, when a document is translated, only the translatable fields need to be editable.

There are different systematic approaches that are used in web applications for storing translations to a relational database. We list below some of the most frequent ones. A simple approach consists in having different columns for each language. Although the rationale behind such an approach is that no table joins are necessary when retrieving a record, it suffers in terms of flexibility: adding a language requires significant changes to the database schema. Moreover, if one doesn't want to retrieve all the translations, complex queries filtering the columns need to be produced.

Another approach consists of providing a translation table where each row refers to the table, the column and the row of the original translated data. Such approach allows partial translation of some documents, but produces a huge translation table.

A third approach that we are using here consists in creating a parallel table for translations containing the columns that can be translated. A *lang* column separates columns for each language. The figure 5 shows the example of a course where the translation table contains the fields *name* and *description* that are indicate as translatable, while the field *code* is missing. By slight modifications to the ActiveRecord pattern, this approach can be used to automatically retrieve the translation for the current locale, if present.

5.4 Naming of repeated fields

In order to avoid naming conflicts, we prefix tables by the name of the containing group. Hence, the name for the table corresponding to courses contained in semesters which in turn are contained in a program becomes "Program_semester_course". We use the same systematic naming for the corresponding models in the generated code.

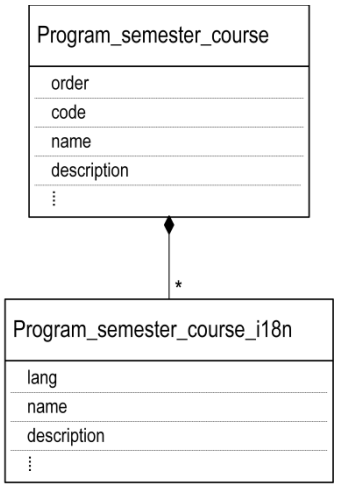

Fig. 5 Representation of translations.

5.5 Managing hierarchies

Another issue to consider during implementation is the hierarchical organization among entities. In a way similar to repeated fields, we generate an order field when the specified hierarchy is *ordered*. When documents are listed in the user interface, we allow users to reorder them.

In the case of a *tree* hierarchy, different approaches can be used: parent-child, explicit-path or nested-set [13]. The first approach consists of a simple one-to-many reflexive relation. The explicit path approach consists in storing the list of ancestors for each record. In our implementations, we use the nested-set model which has the advantages of fast retrieval of all the descendants or ancestors of a given node. In this model, we add *left*, *right* and *level* fields to encode the hierarchy, and provide an extension of models with methods for managing the hierarchy. Note that, other generation strategies may also be easily added.

5.6 Querying for records

Our approach focuses on generating code that follows the common design patterns, Active Record being one of them. In this sense, the generated models are fully compliant with the underlying Object-Relational Mapping libraries that also provide full-fledged relational query possibilities. Although we could have considered a higher-level query language designed upon this model, it will imply breaking the usual design patterns which in turn might be counterproductive.

5.7 Designing entities

In this work, our objective was to provide a well-defined model for specification of entities. This is the reason why we provide a domain-specific language. But, we have

designed our model to allow the definition of these models using a drag-and-drop interface. Combined with template possibilities, such an approach might enable non-technical users to define and use their own models.

5.8 Entity evolution and data migration

One of the main problems in software development is the evolution of the software after deployment. In most cases, when changing the underlying data models one wants to migrate existing data to the new model. In such a scenario, when the definition of an entity changes between versions, it would be desirable to provide code for automatic migration. By using difference between two versions of an entity, some of these changes could be detected automatically. Unfortunately, not all the changes can be identified in the general case, even in a simpler model. If we were to use an interactive drag-and-drop interface, we could capture these changes and progressively apply corresponding transformations in the database. Even then, in order to reapply the same changes in a different environment, we would need a well-defined sequence of changes. Providing the means to define a sequence of changes between two versions remains a future challenge.

5.9 Using NoSQL storage

In our work, we mainly covered relational databases that require the generation of a schema and relationships for repeated fields. In this sense, the generation is more complex than that for NoSQL databases which generally don't require explicit schemas and/or provide native support for lists and embedded documents. Nevertheless, our models provide important information on how these documents should be edited and validated before being saved.

6. Conclusions and future work

In this work, we have tried to identify the systematic and repetitive tasks while programming web applications, based on today's best practices. Then, we abstract these into a descriptive model that includes a specification for defining these as entities. These models, fully take into account some of the most important concerns that arise while developing web applications: managing multi-lingual content; specifying the hierarchy of the internal structure of the content; using rich, extensible and reusable higher-level data types.

We use these models to automatically create concrete database schemas and generate code for user interfaces for content management. The generated code follows common design patterns and allows further customizations using standard OOP techniques. The generation of executable code largely simplifies the gathering and specification of requirements since it provides an advanced prototype out of the box, allowing faster iteration cycles in rapid application development approaches.

We consider that this approach can accommodate a large number of information systems. Additional types can be provided to handle storing and editing of more complex data types and, since we generate code, further customizations are possible for specific requirements.

The specification we provided can be conceptually extended in different directions. One such direction we are working on is to provide a simplified workflow model that will be attached to entities. Based on the information provided by the model, we associate change permissions to different users, check that each change to the document respects the workflow and automatically store a complete history of changes to a given document.

Another direction for future development is to use the information provided by entities for generation of RESTful services [13] [14], customizable by additional specifications.

Finally, we can use this model as a starting point for adding semantic annotations to content. For a simpler and straightforward example, microformat [16] annotations can be added as attributes for entity fields and groups and then reproduced when the content belonging to that entity is displayed in a webpage.

References

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns: elements of reusable object-oriented software, Pearson Education, 1994.

[2] F. J. Budinsky, M. A. Finnie, J. M. Vlissides and P. S. Yu, "Automatic code generation from design patterns," *IBM systems Journal*, vol. 35, no. 2, pp. 151-171, 1996.

[3] G. E. Krasner and S. T. Pope, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26-49, 1988.

[4] A. Leff and J. T. Rayfield, "Web-application development using the model/view/controller design pattern," in *Proceedings of the Fifth IEEE International Enterprise Distributed Object Computing Conference (EDOC'01)*, 2001.

[5] M. Fowler, Patterns of enterprise application architecture, Addison-Wesley, 2003 (ISBN 978-0-321-12742-6).

[6] S. Kent, "Model driven engineering," *Lecture Notes in Computer Science*, vol. 2335, pp. 286-298, 2002.

[7] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19-25, 2003.

[8] I. A. Niaz and J. Tanaka, "Code generation from UML

statecharts," *Intl. Conf. on Software Engineering and Application (SEA*, pp. 315-321, 2003.

[9] S. Kelly and J. P. Tolvanen, Domain-specific modeling: enabling full code generation, John Wiley & Sons, 2008.

[10] J. Bentley, "Programming pearls: little languages," *Communications of the ACM*, vol. 29, no. 8, pp. 711-721, 1986.

[11] J. H. a. A. M. S. Marjan Mernik, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316-344, 2005.

[12] D. C. Fallside and P. Walmsley, "XML schema part 0: primer second edition," W3C recommendation, 2004.

[13] J. Rumbaugh, I. Jacobson and G. Booch, "Unified Modeling Language Reference Manual," The. Pearson Higher Education, 2004.

[14] M. J. Kamfonas, "Recursive hierarchies: The relational taboo," *The Relational Journal*, vol. 27, 1992.

[15] R. T. Fielding, "Architectural styles and the design of network-based software architectures," University of California, Irvine, 2000.

[16] L. Richardson and S. Ruby, RESTful web services, Sebastopol: O'Reilly, 2007.

[17] R. Khare and T. Çelik, "Microformats: a pragmatic path to the semantic web," in *Proceedings of the 15th international conference on World Wide Web*, ACM, 2006.

[18] P. Abrahamsson, J. Warsta, M. T. Siponen and J. Ronkainen, "New directions on agile methods: a comparative analysis," in *Proceedings of the 25th International Conference on Software Engineering*, 2003.

Besnik Selimi received a Master degree in software engineering (2004) and a Ph.D. degree in computer science from Joseph Fourier University, Grenoble, France, in 2009. He is currently assistant professor with South East European University. His current research interests are in the fields of software engineering, software testing, web applications and services, etc. He is a member of ACM.

Artan Luma received a PhD degree in computer sciences from South East European University, in 2010. He is currently assistant professor with South East European University. His current research interests are in cryptography, security, semantic web, etc.