*Proceedings of the 8th International Conference on Applied Informatics*
*Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 287–293.*

# Developing Web-Based Applications Using Model Driven Architecture and Domain Specific Languages

**Attila Adamkó, Csaba Bornemissza**

Department of Information Technology
Faculty of Informatics
University of Debrecen
e-mail: adamkoa|bornem@inf.unideb.hu

**Abstract**

Effectiveness and semantic correctness are key issues in modern application development. However ensuring software quality and reducing time needed for the implementation can be a contradiction. In our article we demonstrate methodological approaches, where we maintain a well-defined set of business terms and workflows, while an effective development setup ensures that the implementation is based on these definitions. Domain Specific Languages are used to formalize the domain in a particular area of interest. These languages are much more expressive in their domain and could express design goals in much deeper than general-purpose languages. Model Driven Architecture gives us the environment, where the Entities, Workflows, Data Access Objects are defined and generated based on a UML Model. The possibilities of developer activities are controlled in a strict way, thus ensuring that the semantic information defined in the Model will be maintained during the whole project lifecycle. Strict definition and control of development activities and personal roles also help us to improve effectiveness.

*Keywords:* MDA, Web application, Web modelling, XML, DSL, UML

*Categories and Subject Descriptors:* D.2.10 [Software Engineering]: Design; D.2.11 [Software Engineering]: Software Architectures; H.4.3 [Information Systems Applications] Communications Applications

## 1. Introduction

Model-driven software development is gaining more and more interest nowadays. With the evolution of technologies the web sites are changing into Web based applications. However, the reorganization and the development require knowledge

and integration of several different technologies. The underlying motivation for Model-Driven Engineering (MDE) is to improve productivity. MDE is wider in scope than MDA. MDE adds the notion of multiple modeling dimensions and a software engineering process to MDA. MDE aims to increase the return a company derives from its software development effort. In most cases when people are talking about MDA or MDE the only goal associated with these terms is to reduce the software artefacts' sensitivity for change in deployment platforms by using a PIM and a PSM.

The various dimensions at an intersection play an important role in the choice for a modeling language for that particular model. By example, the modeling language is influenced by the subject area, the stakeholders and the level of abstraction. In this case we can design a Domain Specific Language (DSL) tailored to the specific subject area, the specific stakeholder and with the right level of abstraction.

## 1.1. Short overview of DSLs and MDA

We can hear a lot about Domain Specific Languages (DSLs) nowadays. In software development a domain-specific language is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. This domain can be a problem domain (e.g. insurance, healthcare, transportation) or a system aspect (e.g. data, presentation, business logic, workflow). The idea is to have a language with limited concepts which are all focused on a specific domain. This leads to higher level languages improving developer productivity and communication with domain experts.

A domain-specific language can be either a visual diagramming language, such as those created by the Generic Eclipse Modeling System, programmatic abstractions, such as the Eclipse Modeling Framework (EMF), or textual languages. The line between domain-specific languages and scripting languages is somewhat blurred, but domain-specific languages often lack low-level functions for file system access, inter-process control, and other functions that characterize full-featured programming languages, scripting or otherwise.

In model-driven engineering many examples of domain-specific languages may be found like Object Constraint Language (OCL), a language for decorating models with assertions or Query/View/Transformation (QVT), a domain specific transformation language. However languages like UML are typically general purpose modeling languages.

A modeling paradigm for MDE is considered effective if its models make sense from the point of view of the user and can serve as a basis for implementing systems. The models are developed through extensive communication among product managers, designers, and members of the development team. As the models approach completion, they enable the development of software and systems.

In order to achieve our goals we need to find an effective way in the Model-Driven Architecture approach that defines system functionality using a platform-

independent model (PIM) using an appropriate domain-specific language (DSL).

# 2. Defining DSLs

A DSL life cycle can contain five development phases: decision, analysis, design, implementation and deployment. In practice DSL Development isn't a sequential process, the phases should be applied iteratively.

## 2.1. Decision

The development of a DSL starts with the decision to develop a DSL, to reuse an existing one, or to use a General Purpose Language (GPL). If a domain is very fresh and little knowledge is available, it doesn't make sense to start developing a DSL. In order to determine the basic concepts of the field, first the regular software engineering process should be applied and a code base supported with libraries should be developed.

## 2.2. Analysis

In the analysis phase the problem domain is identified and domain knowledge is gathered. The output of formal domain analysis is a domain model consisting of:

- A domain definition, defining the scope of the domain,
- Domain terminology (vocabulary, ontology),
- Descriptions of domain concepts, and
- Feature models describing the commonalities and variability of domain concepts and their interdependencies.

The information gathered in this phase can be used to develop the actual DSL. Variability indicate what elements should be specified in the DSL, while commonalities are used to define the execution engine or domain framework.

## 2.3. Design

A DSL can be designed from scratch or it can be easier to base it on an existing language. If it is based on a language it mostly restricts and extends that language and the existing language-based rules or semantics are influencing the design procedure. If you design your DSL from scratch the basic building blocks are created in a natural language and/or examples. Fortunately there are tools which can help you to create an editor which would accept only elements in your language.

### 2.4. Implementation

For executable DSLs the most suitable implementation approach should be chosen. It could be an interpreter, a compiler/generator or a commercial off-the-shelf product. While the different approaches can make a big difference in the total effort to be invested in DSL development, the choice for a particular approach is very important.

In our research we choose the solution supported by the Eclipse Modeling Project (EMP). Xtext is a component that supports the development of a DSL grammar using an Extended Backus-Naur Form (EBNF)-like language, which can use this to generate an Ecore-based metamodel, Eclipse-based text editor, and corresponding ANTLR-based parser. This tool makes our approach very effective for the production because the created DSL can be validated against the grammar.

### 2.5. Deployment

In the deployment phase the DSLs and the applications constructed with them are used. Developers and/or domain experts use the DSLs to specify models. These models are implemented with one of the implementation patterns presented in the previous section (e.g. the models are interpreted by an engine). Such an implementation results in working software which is used by end-users.

An optional or more exactly a final step may exist in this life cycle. The maintenance. While domain experts themselves can understand, validate, and modify the software by adapting the models expressed in DSLs, sometimes changes in the software may involve altering the DSL implementation. Because it is not a new idea or decision this could not be a first stage in a life cycle, rather than a closing stage which could lead to a new cycle or only a small modification in the language.

## 3. An example

While computer languages have a syntax (e.g. ';' to terminate commands), a general semantics (e.g. expressing conditional clauses, loops) and a way of defining types for your problem domain (e.g. Customer, Order, Product) but the domain itself is not described in its natural environment.

Take the example in Figure 1. It contains a Java class but only the relevant information is highlighted. One can see that Java is too specific for our purposes. However, if we focusing only on the relevant part we can find the properties of our domain entity as we can see in Figure 2.

If we use this way of formalizing our problem domain we can reach an abstract model which focusing on only the relevant information and captures domain knowledge in a metamodel. We can communicate using an ubiquitous language and that metamodel drives the implementation. This entity can be transformed into POJOs

or DAOs. It does not matter which one is used because the model is the same only the representation is different.

```
@SuppressWarnings("serial")
@Entity
@Table(name = "CUSTOMER_INFO")
public class CustomerInfo implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "idSeq")
    @SequenceGenerator(name = "idSeq", sequenceName = "CUST_SEQ", allocationSize = 1)
    @Column(name = "CUST_ID", nullable = false)
    private String customerId;

    public void setCustomerId(String customerId) {
        this.customerId = customerId;
    }

    public String getCustomerId() {
        return customerId;
    }

    @Column(name = "EMAIL", nullable = false, length = 128)
    private String emailAddress;

    public String getEmailAddress() {
        return emailAddress;
    }

    public void setEmailAddress(String emailAddress) {
        String oldValue = emailAddress;
        this.emailAddress = emailAddress;
        firePropertyChangedEvent("emailAddress", oldValue, this.emailAddress);
    }
```

Figure 1: A Java class - Highlighting relevant information

```
entity CustomerInfo
  (id=CUST_ID, sequenceName=CUST_SEQ)
{

  String emailAddress (notNull, length = 128)

}
```

Figure 2: A domain object

Another example could be the representation of the relationship between two entities. In a GPL language (like UML) a relationship can be established between any kinds of classes. In DSL we can create the rules which are enforcing the relationship between entities and allow connecting and establishing only substantial relations.

# 4. Code generation

These models help to comprehend the problem domain, but these models would offer more complex support if we could generate from them a working prototype of the desired Web application. By using code generation techniques it is important to separate the generated code from the developer-written code. During maintenance

it is a known problem that changes in a generated code makes it impossible to apply model changes, re-generate the code without losing the manual changes in the sources. The good practice is to create three classifications of sources and separate them in clearly defined folder structures. Category one is the generated code that must not be changes manually. Category two are the generated sources that are to be edited by developers/architects. These are for instance web service skeleton classes, EJB session bean definitions. Sources belonging to this category need merging when model changes occur. When applicable, using inheritance and editing the child class makes it easier to follow model changes. In the last category are the non-generated sources created and maintained by developers.

### 4.1. A DSL to build DSLs . . . Xtext

In our research we concentrated on creating DSL-s that support the development process from two perspectives. One is obviously the Domain Model, where the DSL contains Domain related knowledge, and this is transformed into more usable source code representing Domain Objects. On the other hand there are DSL elements that represent architectural information and the generated code is rather supporting the required architectural outcome for de development process. Xtext is an open source tool we used for DSL to DSL transformations, and for DSL to effective source code transformations. Integration with the Eclipse Modeling Framework made it possible to have all the layers of the MDA approach in a single Eclipse project.

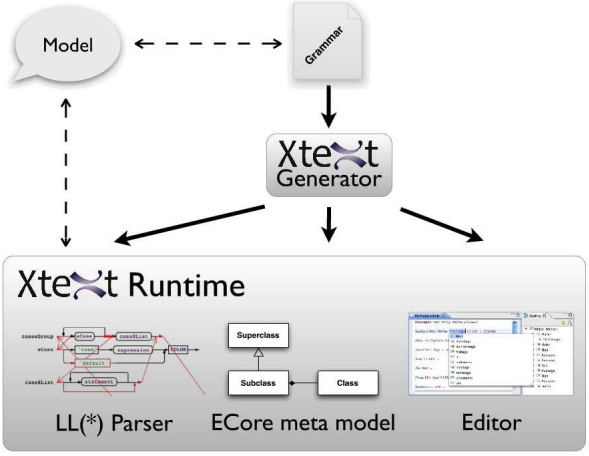


Figure 3: Integration of Xtext generator

## 5. Further possibilities

In this paper we have illustrated how data-oriented Web applications differ from traditional software, how complex and not at all systematic tasks. We have intro-

duced a new methodology to help develop Web applications rapidly and effectively based on UML and XML technologies supporting data management task in small and medium sized projects. We have added some remarks in the implementation phase utilizing XML technologies to develop modular, scalable and expandable Web based systems. Ongoing researches can go in several interesting research directions in the design and development phase. We are going to study the additional expandability of our UML based methodology.

**Acknowledgement.** This work is supported by TÁMOP 4.2.1./B-09/1/KONV-2010-0007/IK/IT project. The project is implemented through the New Hungary Development Plan co-financed by the European Social Fund, and the European Regional Development Fund.

# References

[1] Ginegi, A., Murgesan S.: The Essence of Web Engineering, in IEEE Multimedia, Vol. 8., no. 3., 2003.

[2] Schwabe D.: A Conference Review System. 1st Workshop on Web-oriented Software Technology, 2001.

[3] Eric Evans, Domain Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2004.

[4] Gnaho C.: Web-based Information Systems Development - A User Centered Engineering Approach, Lecture Notes in Computer Science, 2001.

[5] Bauer C. et al.: Matching Process Requirements with Information Technology to Assess the Efficiency of Web Information Systems, Information Technology and Management 2, 2001.

[6] Hennicker R. and Koch N.: A UML-based Methodology for Hypermedia Design., UML´2000, LNCS 1939, Springer Verlag, 2000. 410-424

[7] W3C - World Wide Web Consortium, http://www.w3.org/

[8] Eelco Visser. WebDSL: A case study in domain-specic language engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, Generative and Transformational Techniques in Software Engineering (GTTSE 2007), Lecture Notes in Computer Science. Springer, 2008.