Web Engineering

The emerging Web engineering discipline deals with the process of developing Web-based systems and applications. This includes theoretical principles and systematic, disciplined and quantifiable approaches towards the cost-effective development and evolution of highquality, ubiquitously usable Web-based systems and applications. It fundamentally concerns the technology which enables the construction of Web applications.

Web Engineering includes the following areas:

- Web Process & Project Management Disciplines
- Web Requirements Modeling Disciplines
- Web System Design Disciplines, Tools & Methods
- Web System Implementation Disciplines • Web System Testing Disciplines
- Web Applications Categories Disciplines

Currently, Web engineering does not provide a unique and systematic approach to the development process containing process models, architectures, suitable techniques and methods with quality assurance. As a result, Web engineering is still struggling to establish itself as a reliable engineering discipline. The cost of poor reliability and effectiveness has serious consequences for the acceptability of the systems. One of the main reasons for the low acceptance of Web-based applications could be the gap between design models and the implementation model of the Web.

Web Engineering methodologies

In the past decade many design methods have been created: OOHDM, OO-H, UWE, W2000, WSDM and WebML are among the most popular ones. From a modeler's perspective, each of them offer some possibilities for modeling the levels and aspects mentioned above, and they all come with a guideline for the development process. On the other hand, today's situation is somehow similar to the well-known "object-oriented method war" of the 1990ies. That "method war" has ended with the unification of the different modelling notations which resulted in the UML so the real question is that can this strategy also work for the existing web engineering approaches or not.

Some of the existing Web application design methods (e.g., UWE, WebML) offer a metamodel, as well. This allows model-based development since one need to build models conforming to the appropriate metamodel in order to capture the structural, navigational or presentational structure of the application to be developed. However, in the most of the cases, these models mix the different levels of Web applications that results in a solution that might be appropriate for the given application domain but makes the reuse of models or model parts almost impossible.

Model-Driven Web Engineering

As the field of Model-Driven Web Engineering (MDWE) approaches follow the well-known Model-Driven Engineering principles, some methods create Computational Independent Models (CIMs, e.g., in the form of requirement models), almost all of them allows the creation of Platform Independent Models (PIMs) for structure, navigation, presentation or business processes and most of them provides a means of obtaining Platform Specific Models (PSMs) for various platforms (e.g., J2EE, .NET, Spring, Struts, etc.) that can further be transformed into code.

Model transformations are the most important operations in model engineering, describing how elements in the source model are converted into elements in the target model. This is achieved by relating the corresponding metamodel elements in the source and the target metamodels. Transformations can be classified into two categories: vertical transformations (a.k.a. refinements) are defined between models of different abstraction levels (e.g., PIM—PSM mappings), while horizontal transformations are mappings between models of the same level of abstraction (e.g., for improving or correcting a model).

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).

5.1. **MDWE and the DSL lifecycle.** 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.

5.1.1. 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.

5.1.2. 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 variabilities of domain concepts and their interdependencies.

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

5.1.3. 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.

5.1.4. 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.

One possible solution can be found in the Eclipse Modelig 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.

5.1.5. 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.

Prev
Characteristics of Web Applications and Web Engineering

Up
Home

Next
Conclusions and summary