Using Framework Introspection for a Deep Integration of Domain-Specific Models in Java Applications

Thomas Büchner and Florian Matthes
Fakultät für Informatik
Technische Universität München
{buechner,matthes}@in.tum.de

Abstract:
Domain-specific models and languages are an attractive approach to raise the level of abstraction in software engineering. In this paper, we first analyze and categorize the semantic dependencies that exist between domain-specific models and their generated implementations via frameworks and customization code in a target programming language. We then demonstrate that framework introspection allows a deeper integration of domain-specific models into statically and polymorphically typed object-oriented languages like Java. Using the example of an introspective persistence and query framework for Java, we demonstrate how programmer productivity and software quality can be improved substantially. Since the Java IDE captures the semantic dependencies between the Java application and its embedded domain-specific model(s), it is able to provide programmers with powerful consistency checking, navigation, refactoring, and auto-completion support also for the domain-specific models. Our introspective approach works for whitebox and blackbox frameworks and is particularly suited for the integration of multiple domain-specific models (e.g. data model, interaction model, deployment model) in one application. Due to space limitations, these benefits [2] are not covered in detail in this paper. The paper ends with a discussion of future work on how introspective models can improve the maintenance of long-lived business applications.

1 Introduction

To cope with the increasing complexity and constant change of business applications, new abstractions have been developed which are intended to increase programmer productivity and software quality: Statically and polymorphically typed object-oriented programming languages like Java or C# provide a powerful basic abstraction. Today, they are supported with rich IDEs that provide programmers with powerful consistency checking, navigation, refactoring, and auto-completion support. Based on these languages and tools, frameworks provide architectural abstraction. In order to solve a concrete problem, a framework has to be customized. Modern software systems usually utilize several frameworks, for example for persistence management, web-based interaction or distributed computing. Developers of a complex system have to understand both, the frameworks and their customizations.

Model-driven development tries to raise the level of abstraction of the framework customization process. Customizations are represented as models of a domain-specific lan-

guage. Model-driven hereby means, that there is a transformation between the models and the concrete customization artifacts [6].

As a consequence, there exist artifacts on two different levels of abstraction (framework core and handwritten customizations vs. models). Keeping these artifacts over the lifetime of business applications consistent is the key challenge to be met by model-driven approaches [12]. The central question of this paper is how to better realize and integrate domain-specific languages. We put special emphasis on the issue of integration. Analogous to the approach of Proof Carrying Code [13] we enable Java programs to provide model information through introspection.

This paper is organized as follows: We first review related work on DSLs and roundtrip engineering (Section 2). In Section 3 we analyze and categorize semantic dependencies that exist between domain-specific models and their generated implementations via frameworks and customization code in a target programming language and identify three integration requirements specific to model-driven development. Section 4 gives an overview of our approach to introspective model-driven development (IMDD) based on framework introspection [2] which supports introspective blackbox and whitebox frameworks. Due to space limitations, we only explain in Section 5 how our introspective modeling framework (IMF) provides introspection for whitebox frameworks. A specific example of whitebox framework introspection is the persistence and query framework described in Section 6. Using this example, we illustrate how model-core and model-code integration is achieved. Section 7 compares our approach with popular generative model-driven approaches and highlights its benefits in terms of programmer productivity and software quality. The paper ends with a discussion of future work on how introspective models can facilitate the realization and maintenance of long-lived business applications.

2 Related Work

Specifying the behavior of computer systems using domain-specific abstractions has a long tradition in computer science [3]. One way to implement a DSL is to tailor an existing base language into a DSL. These kinds of DSLs are called *embedded* or *internal* DSLs [5]. An advantage of this approach is that the whole programming environment available to the base language can be reused. The main disadvantage is, that mainstream statically typed object-oriented programming languages are not designed to be syntactically extensible and do not allow for the creation of powerful internal languages.

The prevalent way of implementing a DSL using a statically typed object-oriented language as a base language is building an *external* DSL [11]. As already introduced, we see DSLs as means to specify solutions on a higher level of abstraction in combination with frameworks [4]. Therefore, building an external DSL means building a transformation, which generates customizations from models. This is called *generative* model-driven development. In such a process a metamodel is created, which represents the extension points of the framework to be customized. Additionally, transformation rules which control how to transform the models, have to be created. Based on the metamodel, the framework user

creates a model which solves the problem at hand. This model will then be transformed automatically into customization artefacts.

As already mentioned, the lower level artifacts (framework core, handwritten customizations) have to be maintained as well as the higher level models. This leads to the wish for *roundtrip engineering*, which means that artifacts on both levels of abstraction should be editable and changes to one of them should lead to an immediate synchronization of the affected one. Realizing roundtrip engineering in a generative model-driven process is a challenging task [9].

An approach similar to the one introduced in this paper proposes the use of *Framework-Specific Modeling Languages* [14] that are defined on top of existing object-oriented frameworks. This approach tries to facilitate roundtrip engineering by defining transformations between the levels of abstraction in both directions.

The approach presented in this paper is based on *introspective* frameworks, in which the customization points are annotated explicitly and the customizations follow a constrained programming model. This enables the extraction of models as transient views and makes roundtrip engineering easily achievable.

3 Integration of Domain-Specific Models

In this paper, we call two elements *integrated*, if there exists a semantic dependency between these two, this dependency is stated explicitly [10], and it can be inferred easily by a tool.

Source code of a statically typed programming language is integrated, in that for example the connection between declarations and usages of methods and fields can be inferred at compile-time. Over the last years, the new class of post-IntelliJ-IDEs made excessive use of this property to increase developer productivity and changed the way developers perceive source code [19]. Integration enables features like navigation, search for references, code assist, and refactoring.

As we want to realize domain-specific languages in an *integrated* way, we first identify three integration requirements specific to modeling approaches. As a first requirement, all artifacts related to the DSL should be integrated with the underlying framework core. These frameworks are usually written in an object-oriented programming language. We call this requirement *model-core* integration.

In another typical scenario, concepts defined in a model of a domain-specific language have to be referenced in handwritten code. This occurs because in most cases it is not possible to completely specify a complex system using a declarative DSL. In this case we require the domain-specific language to be integrated with the code of the base language. This is called *model-code* integration in the following. There are two aspects of model-code integration, which differ regarding the location of the handwritten code. In one case, the code which references the model is part of the customizations of the framework. In this case, only parts of the framework can be customized declaratively using a DSL. In

another scenario the code which accesses the model belongs to customizations of another framework. In both cases, the model artifact should be integrated with the handwritten code to improve the consistency of the overall system.

Many problems are solved using several frameworks in cooperation. In such a case, an additional requirement is the integration of different domain-specific languages with each other, which we call *model-model* integration.

Which benefits arise from a modeling approach, which realize these three integration requirements? The main benefit is the automatic checking and assurance of consistency between the artifacts involved. Since the connection between the artifacts in an integrated scenario is stated explicity, tools can help ensuring consistency. This improves the quality of the overall system. Another benefit as already mentioned is tool support for productivity enhancements like navigation, search for references, refactoring, and input assistance.

The prevailing generative model-driven approaches lack integration, since the relationships between the modeling artifacts and the underlying system are not stated explicitly. The underlying reason for this problem is the lack of *symbolic integration* [11] between the artefacts involved. For instance it is not possible to navigate automatically from metamodel artifacts to the extension points of the framework core, reflected by them. The DSL is not integrated with the framework it is expected to customize (no model-core integration). The model-code and model-model integration requirements are not met either by generative model-driven approaches. As a consequence of this lack of integration it takes a lot of manual work to keep all artifacts consistent.

4 Introspective Model-Driven Development

In [1] we proposed a bottom-up approach to model-driven development, which we call *introspective* model-driven development (IMDD). The main idea of IMDD is the construction of frameworks that can be analyzed in order to obtain the metamodel for customizations they define. The process in which the metamodel is retrieved is called *introspection*. The term introspection stems from the latin verb *introspicere*: to look within. Special emphasis should be put on the distinction between introspection and *reflection* in this context. We use both terms as they have been defined by the OMG [16] (see table 1).

In analogy to the definition of reflective, *introspective* describes something that supports introspection. An introspective framework supports introspection in that its metamodel can be examined.

The whole process of introspective model-driven development is schematically shown in Figure 1. The process is divided into the well known core development phase and the application development phase. The first result of the core development phase is an introspective framework. An introspective framework supports introspection by highlighting all declaratively customizable extension points through annotations [17]. This enables the extraction of the metamodel by *metamodel introspection*. It is important to understand, that the metamodel is not an artifact to be created by the framework developer, but rather can be retrieved at any point in time from the framework.

Table 1: Term Definitions

introspection	A style of programming in which a program is able to examine parts of its own definition.
reflection	A style of programming in which a program is able to alter its own execution model. A reflective program can create new classes and modify existing ones in its own execution. Examples of reflection technology are metaobject protocols and callable compilers.
reflective	Describes something that uses or supports reflection.

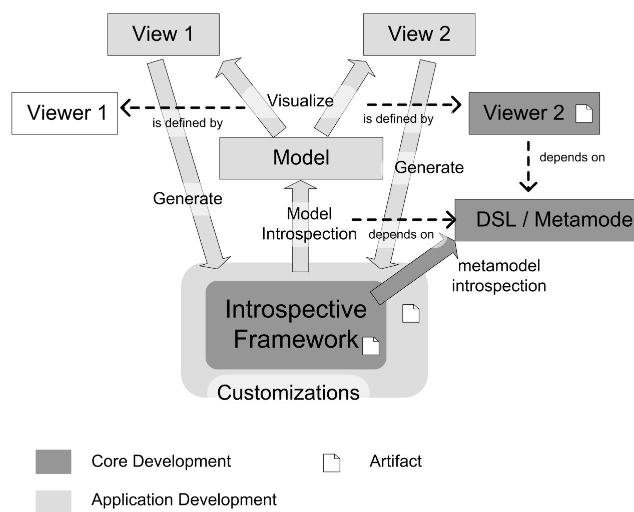

Figure 1: Introspective Model-Driven Software Development

The central artifact of the application development phase are the customizations to be made by the framework user. In IMDD it is possible to analyze these artifacts and to obtain their model representation. This is called *model introspection*. The model is an instance of the retrieved metamodel and can be visualized by different viewers (i.e. visualization tools). We implemented out of the box viewers which can visualize an introspective model in a generic way. In some cases it is desirable to develop special viewers which visualize the model in a specific way (e.g. as a UML model). This will be done by framework developers in the core development phase. The manipulation of the model can be either done by using the views or by manipulating the customization artifacts directly. In both cases an updated customization artifact leads to an updated model and subsequently to an updated view. As a result of this, the model and the views are always synchronized with the actual implementation and can never "lie". This kind of visualization is called *roundtrip visualization* [15].

Generative model-driven development and IMDD differ in the direction the transformation between the model and the customization artifacts takes place. There are similarities between our approach and that of internal DSLs. In both approaches, the models are represented in terms of the base language. The difference comes in how the user of the DSL perceives and manipulates models. Using an internal DSL, the user directly edits statements of the base language, whose syntax is tailored to the particular domain. Because of the inflexibility of statically typed object-oriented programming languages to be tailored syntactically, we have to visualize the customization artifacts on a higher level of abstraction.

The main idea of introspective model-driven development is the direct extraction of the model and the metamodel from the framework artifacts which represent them. There are two categories of frameworks: blackbox frameworks and whitebox frameworks. They differ in the way adaptation takes place. Due to space limitations we only give an overview of how introspection of whitebox frameworks works. Introspective blackbox frameworks are discussed in [1].

5 Whitebox Introspection

The customization of whitebox frameworks is done by providing implementations of abstract classes of the framework core in the base programming language. More specifically, the framework user specifies the desired behavior by implementing methods. These methods are called *hook methods* and represent the extension points of the framework [7]. Regarding introspective whitebox frameworks there are two kinds of hook methods – introspective and non-introspective hook methods. Customization code of introspective hook methods must use a constrained subset of the expressiveness of the base language. We call this subset an *introspective programming model*. Programming using an introspective programming model is of declarative nature and enables the extraction of the model. In contrast, the implementation of a non-introspective hook method can use the full expressiveness of the imperative base language.

The main idea of whitebox introspection is to annotate introspective hook methods in the framework core and to analyze the introspective method implementations. The analysis of the structure of the introspective methods results in the metamodel of the framework core, and the analysis of the method implementations leads to a model of the provided adaptations.

The conceptual idea of whitebox introspection described so far is very generic. To verify the idea, we identified important introspective methods and programming models. In [1], we introduced some basic introspective methods and their programming models. They form a meta-metamodel of whitebox introspection, in that they enable a concrete whitebox framework to draw on these methods. We also implemented generic tool support, which creates an introspective model for introspective whitebox frameworks. Our tools are based on the Eclipse IDE, which is available under an Open Source license and is easily extensible because of its plugin architecture. Specifically, we rely heavily on the Eclipse JDT subproject [20], which provides access to an abstract syntax tree representation of the source code.

On top of Eclipse we built a framework which supports introspection in a general way. This framework is called *Introspective Modeling Framework* – IMF and provides basic abstractions for analyzing source code, representing the models, and visualizing them. Based on IMF there are tools which support blackbox and whitebox introspection with generic visualization. As previously mentioned, it is sometimes desirable to customize the way the introspective model is created and visualized (see Figure 1). This can be done easily, using the existing generic tools as a starting point.

6 An Introspective Persistence and Query Framework

A core requirement of information systems is persistent storage and efficient querying of business objects. Usually this is implemented using a relational database. There is a conceptual gap between relational data and object-oriented modeling of business objects. A *persistence framework* is used to bridge this gap with an object-relational mapping [18].

In this section we explain how to use an introspective whitebox framework for this purpose. The principal idea of our introspective whitebox framework is to explicitly represent the metamodel as introspective Java code. The framework provides abstract classes for all metamodel concepts, which have to be implemented and instantiated to model concrete business objects. For instance, there are abstract classes which can be used to specify persistent properties and persistent relationships between business objects. These abstract classes are introspective, because they have introspective methods. For example, these are value-methods which restrict the implementation to return a value literal or a reference to a final variable [1].

The schema of `Person` objects with a `firstName` property and a *one-to-many* relationship to `Group` objects is defined using inner classes that override (generic) framework classes.

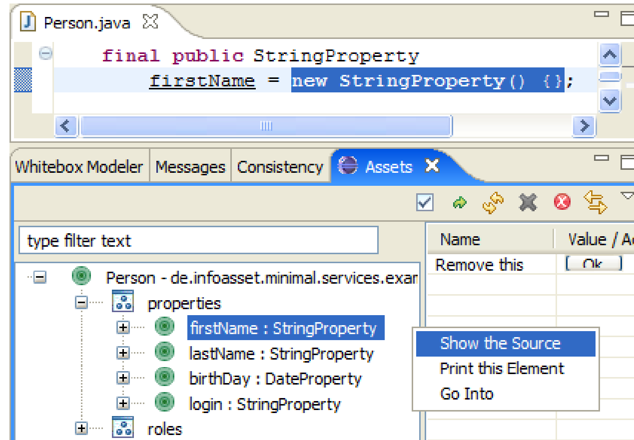

Figure 2: Visualization of the Data Model in a Tree View

```
public class Person extends Asset {
 final StringProperty firstName = new StringProperty() {
   @Override
   int getMaxLength() {
     return 100;
   }};

 final ManyRole<Group> groups = new ManyRole<Group>() {
   @Override
   Role otherRole() {
    return Group.SCHEMA.prototype().members;
   }};
 ...
```

Because this is introspective code, the model can be analyzed and visualized. Figure 2 visualizes the model using a tree view. It is possible to edit the model in this view. On the other hand it is always possible to navigate to the corresponding code, which represents the model attribute. A graphical visualization as a UML class diagram is given in Figure 3. More complex models used in industrial projects based on our IMF are presented in [2].

The following subsections use this example to explain the benefits of our introspective approach in terms of integration.

6.1 Model-Core Integration

The way models are represented in the proposed introspective persistence framework fulfills the requirement of model-core integration as introduced in section 3. This means,

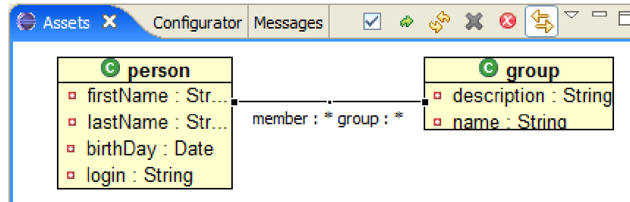

Figure 3: Visualization in UML Class Diagram Notation

that there is an explicit connection between the model attributes and the corresponding hooks of the customized framework. In the example presented above, the persistent model property `firstName` specifies its maximal length to be 100. This is done by overriding a method which is defined in the abstract super class `StringProperty`, which belongs to the framework core. The Java compiler knows about this relationship and additionally it is stated explicity through the use of the `Override` annotation. Therefore, the Java compiler already checks consistency constraints. On the other hand it is easily possible to navigate into the framework core and to find out in which context the attribute maximal length will be used. A post-IntelliJ-IDE like Eclipse can provide programmers with a list of available model attributes .

Another advantage of this model representation is the direct use of the refactoring capabilities of Eclipse to execute refactorings on both the framework and the models in one step.

6.2 Model-Code Integration

Now we can specify and analyze types of business objects at a higher level of abstraction. But how does the programming model look like? Accessing business objects is done using the explicitly represented model. Setting the first name of a person is done as follows:

```
person.firstName.set("Thomas");
```

Another important aspect concerning the programming model is navigation, also known as “traversal”. This means moving from one business object to another along existing association relationships. As already shown, in our persistence framework relationships are defined by instantiating metamodel classes. In particular, this is done specifying the association ends by instantiating objects of type `OneRole` or `ManyRole`. The instantiated model instances are used to navigate relationships in a type-safe way. Accessing all groups a person has a member association with is expressed like this:

```
Iterator<Group> groups = person.groups.getAssets();
```

Another aspect of a persistence framework is querying business objects for certain criteria. In our introspective persistence framework there is a query API to serve that purpose. The

following query retrieves all persons with the first name “Thomas”:

```
Query q = new QueryEquals
   (Person.SCHEMA.prototype().firstName, "Thomas");
Iterator<Person> persons = Person.SCHEMA.queryAssets(q);
```

These examples of the programming model show, that the handwritten customization code is integrated with the data model since it directly refers to the model definition. Also in this case type consistency is checked by the Java compiler and the IDE even for complex nested and join queries.

7 A Comparison with Model-driven Generation of POJOs

To help the reader to better understand the benefits of our approach, we now compare the introspective persistence and query framework with the prevailing approach to object-relational mapping using POJOs [8] (Plain Old Java Objects). Similar benefits arise in other modeling situations (e.g. interaction models and configuration models) as discussed in [2].

This is not exactly an adequate comparison, since representing persistent business objects using POJOs is not a model-driven approach. The model cannot be easily extracted and looked upon on a high level of abstraction. But most generative model-driven approaches using Java technologies generate POJOs [21] [22] and therefore inherit the lack of integration which we will show is inherent to this approach.

In a POJO-based approach persistent business objects are represented as JavaBeans. Properties are represented as private fields with a getter and setter method. Associations are represented using collection fields and getter and setter methods. In both cases, additional metainformation might be provided with annotations or through an external configuration file. The introduced business object `Person` will be represented as follows:

```
public class Person {
  private String firstName;
  @Column(length=100)
  String getFirstName() { return firstName; }
  void setFirstName(String s) { this.firstName = s; }

  private Set<Group> groups;
  Set<Group> getGroups() { return groups; }
  void setGroups(Set<Group> s) { this.groups = s; }
  ...
```

First lets have a look at model-core integration. Model attributes are represented here using annotations [17]. Java annotations are syntactically elegant but only provide very

limited automatic consistency checking capabilities. The scope of an annotation only can be restricted to very generic Java constructs as fields, types, methods, and constructors. The `Column` annotation used in the example could also be applied to the `setGroups` method, which would be an inconsistent modeling. This inconsistency cannot be checked by the Java compiler. Post-IntelliJ-IDEs cannot provide help answering questions about which modeling attributes are available in a specific situation.

Now lets focus on model-code integration. The programming model for accessing properties and navigating associations is straight forward and uses the getter and setter methods. Apart from aesthetic arguments of taste and style both programming models are equivalent and integrated, in that they provide a type-safe way of accessing properties and navigating associations.

But accessing fields and navigating associations is only one part of the overall usage scenarios. Another one is querying, as already introduced. In our introspective persistence framework, the metamodel instances can be referenced integrated to specify queries. This is not possible with a POJO-based persistence framework, because the metamodel instances are represented through Java fields and methods, which are not referenceable. This leads to an unsafe way of defining queries:

```
Criteria crit = session.createCriteria(Person.class);
crit.add(Expression.eq("firstName", "Thomas"));
List<Person> result = crit.list();
```

Unsafe means hereby, that the properties are identified using strings. This is an implicit binding which is not accessible for the compiler and the IDE. Renaming a property or association with a simple refactoring may lead to broken queries, which cannot be found automatically. This is not an issue with our introspective persistence framework.

Another advantage of our introspective modeling approach is, that it is very easy to access metadata at runtime. This enables generic services on business objects, e.g. generic visualization and manipulation. An asset can be asked directly for the model instances which define its schema, which in turn provide access to all available metadata. In a POJO-based persistence framework accessing metadata at runtime has to be done using Java reflection, which is cumbersome. Accessing more sophisticated metadata, like length restrictions as applied via annotations in our example, is even more complicated.

For the sake of completeness, we want to mention the high level modeling and visualization capabilities of our introspective persistence framework again. They come out of the box as an integral part of our framework. For POJO-based frameworks the same kind of tool support is theoretically possible, but the development of the framework core and the tooling do not go hand in hand.

The points mentioned so far concern the experience of the framework user. We believe, that also the framework developer benefits from introspection, because all metadata is directly accessible to the framework core.

8 Conclusions and Future Research

Integration of domain-specific models with the underlying system is a desirable goal in model-driven development because it improves consistency and enables productivity features developers are used to nowadays. Existing generative approaches lack integration because of their character as an external DSL. We have shown, that the proposed approach of introspective model-driven development using whitebox frameworks integrates domain-specific models with the framework core and with handwritten customization code. Referring to the integration requirements proposed in section 3, we have shown that introspective whitebox frameworks enable model-core and model-code integration and solve a key maintenance problem in model-driven software engineering [12].

Furthermore, an introspective software engineering approach increases the maintainability of long-lived business application by explicitely stating adaptations in the system. These adaptations are made on a high level of abstraction using a domain specific language and a programming model that provides abstractions tailored to the problem domain.

In the future, we plan to continue our research on introspective models and address the following issues:

- How can introspective models be enriched to capture more business logic, for example through cascading deletes, derived attributes, declarative constraints on classes and relationships, temporal constraints, security constraints and other business rules?
- How can domain-specific models be visualized and navigated including their links with implementation artifacts?
- How can introspective models be accessed by standard tools used in industry, like configuration management databases (CMDBs), enterprise architecture management tools?

References

[1] Thomas Büchner and Florian Matthes, *Introspective Model-Driven Development*. In Proc. of Third European Workshop on Software Architectures (EWSA 2006), pages 33-49, LNCS 4344, Springer, Nantes, France, 2006.

[2] Thomas Büchner, *Introspektive modellgetriebene Softwareentwicklung*. Dissertation, TU-München, Lehrstuhl für Informatik 19, August 2007

[3] Arie van Deursen, Paul Klint, and Joost Visser *Domain-specific languages: An Annotated Bibliography*. ACM SIGPLAN Notices, vol. 35, pp. 26–36, 2000.

[4] Arie van Deursen *Domain-specific languages versus object-oriented frameworks: A financial engineering case study.*. In Smalltalk and Java in Industry and Academia, STJA'97, pages 35-39. Ilmenau Technical University, 1997.

[5] Paul Hudak *Building Domain-Specific Embedded Languages*. ACM Computing Surveys, vol. 28, pp. 196, 1996.

[6] Markus Völter and Thomas Stahl, *Model-Driven Software Development*. John Wiley & Sons, 2006.

[7] Wolfgang Pree, *Essential Framework Design Patterns*. Object Magazine, vol. 7, pp. 34-37, 1997.

[8] Chris Richardson, *Untangling Enterprise Java*. ACM Queue, vol. 4, pp. 36 - 44, 2006.

[9] Shane Sendall and Jochen Küster, *Taming Model Round-Trip Engineering*. Proceedings of Workshop on Best Practices for Model-Driven Software Development (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications), Vancouver, Canada, 2004.

[10] Martin Fowler, *To Be Explicit*. IEEE Software, vol. 16, pp. 10-15, 2001.

[11] Martin Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?* http://www.martinfowler.com/ articles/languageWorkbench.html

[12] Gregor Engels, Michael Goedicke, Ursula Goltz, Andreas Rausch, Ralf Reussner, *Design for Future - Legacy-Probleme von morgen vermeidbar?* Informatik-Spektrum, Springer Verlag 2009, DOI 10.1007/s00287-009-0356-3

[13] George C. Necula, *Proof-carrying code: design and implementation* PPDP'00: Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming, ACM 2000, New York, pp 175-177

[14] Michal Antkiewicz, *Round-Trip Engineering of Framework-Based Software using Framework-Specific Modeling Languages*. Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06), 2006.

[15] Stuart M. Charters, Nigel Thomas, and Malcolm Munro, *The end of the line for Software Visualization?*. VISSOFT 2003: 2nd Annual "DESIGNFEST" on Visualizing Software for Understanding and Analysis, Amsterdam, September 2003.

[16] OMG – Object Management Group, *Common Warehouse Metamodel (CWM), v1.1 – Glossary*. http://www.omg.org/docs /formal/03-03-44.pdf

[17] Joshua Bloch, *JSR 175: A Metadata Facility for the Java Programming Language*. http://www.jcp.org/en/jsr/detail?id=175

[18] Wolfgang Keller, *Mapping Objects to Tables - A Pattern Language*. Conference on Pattern Languages of Programming (EuroPLoP), Irsee, Germany, 1997

[19] Gail C. Murphy, Mik Kersten, and Leah Findlater, *How Are Java Software Developers Using the Eclipse IDE?*. IEEE Software, vol. 23, pp. 76-83, 2006.

[20] Eclipse Foundation, *Eclipse Java Development Tools (JDT) Subproject*. http://www.eclipse.org/jdt/

[21] Witchcraft, http://witchcraft.sourceforge.net

[22] AndroMDA, http://www.andromda.org