Using Domain-Specific Languages to Describe the Development Viewpoint of Software Architectures

Amir Reza Yazdanshenas, Ramtin Khosravi
School of Electrical and Computer Engineering, University of Tehran, Tehran, Iran.
{a.yazdan, rkhosravi}@ece.ut.ac.ir

Abstract—**The developers of a system are accepted as one of the most important stakeholders of an Architecture Description. The Development Viewpoint is suggested to satisfy the needs of the developers throughout the development process via codeline organization descriptions, programming models, etc. However, the available models for such purposes, if any, barely cross informal natural language descriptions and checklists.**

This paper introduces the idea of enhancing the description of the Development viewpoint using lightweight Domain-Specific Languages and presents the application of such languages in two industrial case studies. This language enables the architect to provide the necessary guidelines that constrains the implementers during the development process and it is also used as a means to discover the deviation of the code from the architecture as the development goes on.

Keywords—**Architectural views; Architecture conformance; Architecture description**

I. Introduction

One method to describe the architecture of a software system is to break up the description into separate views that each address one (or more) of the concerns that stakeholders have about the software architecture. There has been considerable research conducted for finding the appropriate view set for describing software systems architecture like Kruchten's "4+1" [1], Rozanski and Woods [2], and SEI's [3] method. Detailed information on the content of the Development View, one of the suggested views, will be given in section II.

Although there are a few modeling languages for describing the Development view, they generally suffer from imprecise notations. This may lead to an unfortunate gradual process during the development process known as architecture erosion.

The sole purpose of this paper is to show that we can remedy the situation with no substantial efforts. We introduce the basic idea of using a model we currently call the "Development Metamodel" that together with a lightweight Domain-Specific Language (DSL) can enable the architect to provide the necessary guidelines that constrains the implementers during the development process.

Section III states our proposed solution and the subsections show how we can take advantage of an embedded Domain-Specific Language to both describe the architecture and check the consistency of the final product with the intended architecture. Section IV shows the successful application of the proposed approach in two industrial case studies.

II. Architecture Views and Viewpoints

A widely recognized approach to describing complex architectures is to define a set of views and describe one or more cohesive aspects of the architecture into each view. Although the idea of views is not new and dates back at least to 1970s , it has been first standardized by IEEE Standard 1471 in 2000 [4]. In essence a view is "a representation of a whole system from the perspective of a related set of concerns."

Several view sets have been suggested so far [2], [3], [5]. To tackle this multiplicity problem, IEEE Standard 1471 has tried to generalize this idea by defining the concept of a viewpoint: "A pattern or template from which to develop individual views…" This definition has the benefit of keeping the viewpoint set open, while letting the field experts gradually come up with the desired viewpoints that may eventually grow into a reusable library.

Among the many viewpoint (view) sets proposed so far, one that has gathered considerable attention in the domain of large information systems is by Rozanski and Woods [2]. They argue that previous viewpoint catalogues, like Philippe Kruchten's celebrated "4+1" set, have limitations when applied to large information systems. The viewpoint set they propose has six core viewpoints: Functional, Information, Concurrency, Development, Deployment, and Operational. The Development viewpoint is the main focus of this paper and is described in the following subsection.

A. The Development Viewpoint

The developers of a system are accepted, in academia and practice, as one of the most important stakeholders of an architecture description, as their understanding of the target architecture determines the quality of the final product. For this reason there should be at least one view, the Development View, which satisfies the needs of the developers. All suggested view sets, including SEI's view set, point out the need for such a view somehow, but in particular [1] and [2] designate an exclusive view for this purpose, naturally named the Development View. However, there is no strong agreement about the content of this viewpoint. Kruchten's Development view mainly revolves around subsystem identification and layering [1].

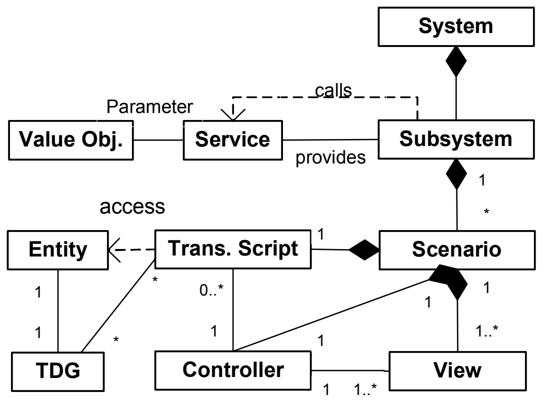

Figure 1 Development Metamodel

Rozanski and Wood's Development view is substantially broader and "supports the software development process" in general [2]. They set a highly ambitious objective for this viewpoint and say it should cover all architectural concerns of all the stakeholders involved in building, testing, maintaining, and enhancing the target system. Module organization, intelligent management of common processing, standardization of design and testing, instrumentation, and codeline organization are the top stakeholder concerns they list. However, the available models for such purposes, if any, barely cross informal natural language descriptions and checklists. From all the above, only the module organization and layering concerns are addressed fairly acceptable via module structure models. Using UML package diagrams, one can succinctly model software modules residing in each layer and show interlayer and inter-module dependencies. For addressing other concerns, currently, one has no choice but to fallback to informal models or natural language descriptions. Although Rozanski and Woods report to have tried representing these concerns via UML diagrams, they acknowledge their effort as being "not worth the bother".

In the next section, we show how we can tackle the problem using a "Development Metamodel" that together with a lightweight Domain-Specific Language can enable the architect to provide the necessary guidelines that constrains the implementers during the development process.

III. THE DEVELOPMENT METAMODEL

In this section, we introduce the idea of what we call the Development Metamodel via an example; using the architecture of a typical online store web application.

Suppose, the architect of this example system has finished the bulk of his job and he now has a clear understanding of the architecture and the intended final product. Suppose, he has clearly described the runtime components of the system and the correspondent software artifacts in the static module structure of the system. The developers can now read the architecture description, digest the models and (if the description is clear enough) all reach a single, common understanding of the system. From now on they are ready to start the development process.

But do all the developers have the same level of expertise to interpret the models in the same way? Or, what if the architect intends to set some rules and constraints that are impossible to express using only the component-and-connector style or the static module view of the system? For instance, consider the following constraints:

a) Each use case scenario needs to be implemented by a single designated controller when using the Model-View-Controller design pattern [6].

b) There could be several web pages (views) related to a use case scenario, but each view must be handled by only one controller.

c) When applying the Table Date Gateway (TDG) enterprise pattern [7] for persisting entities, each entity has one corresponding Gateway and vice versa.

d) Each scenario should be implemented in a single subsystem.

e) All controller classes must be implemented stateless and thread-safe.

f) Elements belonging to the domain layer, such as Transaction Scripts [7] and Entities, should not depend on the presentation layer as the user interface may change in the future.

g) All execution paths of the controller classes must eventually lead to a valid web page, even an error page in case the normal execution path does not end successfully.

How can the architect express these constraints in conventional, UML-compliant component-and-connector or module structure models? For example, a scenario is a way to fulfill a requirement; it is similar to a concept and does not have any runtime presentation, nor will it be manifested as an executable artifact. On the other hand, a controller has a clear runtime presentation and can clearly be modeled as a conventional component. The heterogeneity of these elements inhibits the architect from placing them into a single model, valid by the advised rules of component-and-connector or module structure models.

For such purposes we suggest encompassing all these heterogeneous "elements" into one new model called the "Development Metamodel" (Figure 1). In this example, the target system is divided into subsystems that each constitutes one or more use case scenarios. The scenarios are implemented using the Model-View-Controller design pattern. The domain logic is embedded in Transaction Scripts [7]. Transaction Scripts need to access entities, whose persistent data is mapped to database tables via a Table Date Gateway (TDG) class [7]. Subsystems communicate via published services. Each subsystem can provide some services and call services from other subsystems. Services use value objects as a means for parameter passing.

We suggest the architects put all elements they find of architectural importance during the development process into a single model, totally regardless of their type of presentation or realization in the final product. The architect can directly specify some of the structural constraints using the relationships and multiplicity between these elements. (Note that how concisely the example model in Figure 1 expresses the first four architectural (a to d) constraints of the example system.)

Note that the elements present in the Development Metamodel, indeed, belong to the metamodel level as they do not identify specific use case scenarios or specific class names. Existing notations that can be used for documenting the Development view (such as SEI's Module Viewtype), represent specific elements (e.g. module A, module B) at the model level, not metamodel.

Handed to the developers, probably accompanied by some plain text instructions, the model has some descriptive values. Yet there is no guarantee that this model is interpreted correctly and the final product actually conforms to the intended constraints and guidelines. Apart from that, the last three constraints of the mentioned list (e, f, and g), which are more complex, cannot be expressed via the proposed Development Metamodel, and the complexity of the constraints of a real industrial system could be considerably more. Hence the Development Metamodel lacks enough expressive power; the next subsection introduces our solution to this problem.

A. *The Supporting Domain-Specific Language*

Domain-Specific Languages (DSLs) are languages tailored to a specific application domain [8]. They are small, usually declarative, languages that offer expressive power focused on the target problem domain. DSLs can be divided into two subcategories according to the implementation method: external and internal (also known as embedded) [9]. External DSLs are implemented from scratch and tend to be difficult to implement, as the designer has to implement a compiler or interpreter for the language. Embedded languages are constructed by reusing the compiler or interpreter of a host language, so the implementation overhead is substantially lower than external DSLs. However, the syntactic limitations of the host language can severely impair the expressiveness power of embedded DSLs, and in many cases the optimal Domain-Specific notation has to be compromised to fit the limitations of the host language.

The main reason for choosing a DSL over other description methods is to enable automatic processing of the model. When one describes some aspect of a system via a DSL, he has considerably less implementation responsibilities when processing the description for certain purposes than when he should process graphical modeling languages, like UML, or informal descriptions.

Although “large-syntax languages”, like Java, may not be the most appropriate choice as the host language of an embedded DSL, intelligent use of simple techniques and design patterns, such as interface chaining [10], interface inheritance, Builder pattern [6], and the static import facility of Java 5+ helped us to reduce the extra “noise” syntax to negligible levels. Although the resulting language may still be far from the standards of end-user programming, it certainly serves its purpose as the users of this language (software architects and developers) are already introduced to programming languages.

As every other language, this DSL must provide enough elements for declarations and definitions. This enables the architect to introduce the architectural elements residing in the Development Metamodel into the DSL:

```
Element Scenario = element("Scenario").build();
Element View = element("View").build();
Element Ctrl = element("Controller").build();
```

Then, the architect needs to define the inter-relations of the elements. The DSL must be semantically rich enough to specify the type (e.g. aggregation) and cardinality of each relation, too. Using a simple generic “Relation” class and the interface chaining technique, we were able to incorporate this feature into the DSL with minimum implementation complexity. The final description of the architect is highly expressive and brief like this:

```
Scenario.HasMany(View)
        .HasOne(Controller)
        .HasOne(TransScript);
Controller.References(View)
          .Cardinality(one_to_many);
Controller.References(TransScript)
          .Cardinality(one_to_many);
TransScript.References(TDG)
           .Cardinality(many_to_many);
```

At this stage, we are able to express some of the constraints previously expressed in the example Development Metamodel of the last section such as: there is only a single controller in charge of each scenario or each web page (view) is handled by a single controller. However, the expression of these constraints using a text-based DSL has many benefits over using a graphical language, e.g., easy tool process-ability.

As the last example in this part, we show how we can express the constraints that are not considered structural. Consider the following code snippet:

```
Controller.Ensures(stateless)
          .Ensures(navigability);
TDG.Ensures(stateless);
TransScript.Ensures(stateless);
          .Ensures(presentationIndependent);
```

This represents the last three constraints that were too complex to be expressed in the Development Metamodel. Here, we have stated the classes we want to be implemented stateless. The navigability constraint ensures that all execution paths of the controller classes eventually lead to a valid web page, even an error page. We also require the classes Transaction Scripts, belonging to the domain layer, to be independent from the presentation layer (presentationIndependent). The meaning of each constraint in the DSL should once be documented and handed to the developers together with the Development Metamodel.

We implemented this part of the DSL, through a single “Constraint” interface containing the essential API for checking constraints on an element, like the “Ensures()” method. For adding each new constraint, the architect has to extend this interface in a new class and only provide the implementation for the “Ensures()” method.

Up to this point, the Development Metamodel and the supporting DSL are still a descriptive or documenting facility. In other words, they do not ensure the compliance of the final product to the intent of the architect. In the next subsection we provide a solution for this problem, using the suggested DSL.

B. *Checking Constraints*

The process of extracting information about a program from its source code or artifacts generated from the source code (e.g., Java byte code) using automatic tools is called static code analysis [11]. Available source code analysis tools vary from simple semi-automatic string matching tools to sophisticated automatic tools built upon the Abstract Syntax Tree (AST) of the program.

In order to check the compliance of the output of the developers during the development process with the intended constraints and guidelines set by the architect, first we need to abstract away the details that are not architecturally important. We need to raise the abstraction level up from the source code to the abstraction level of the Development Metamodel and the supporting DSL. For such purposes, one possible approach would be to construct the complete AST of the program, abstract away the undesired details, and carry out further processing on the resulting AST. A second possible approach would be to require the programmers to decorate the source code with some annotations. In this approach the architect needs to define the annotation types capable of expressing the elements present in the Development Metamodel. During the development process, the developers use these annotation types to meta-tag the elements in source code, effectively introducing the architectural elements relevant to the Development Metamodel within the source code.

Using abstract syntax trees has the benefit of staying transparent to the developers and not imposing any responsibility on the developers. However, this method imposes some implementation overheads on the architect. In order to make this approach economically feasible, the architect needs to have access to easy-to-use frameworks. On the other hand, forcing the developers to embed annotations in the source code reduces the implementation overhead greatly, but is vulnerable to human error. For preliminary experiments, we decided to use the annotation-based approach and to implement the more sophisticated non-annotation-based framework only if the initial experiments prove successful. In section IV we will show the application of both methods in two medium-scale industrial projects.

But in order to give a hint of the implementation overhead of the annotation-based approach, we show a step-by-step solution to how one of the structural constraints can be implemented. Firstly, we define the necessary annotation types supporting the desired concepts of the example Development Metamodel. Secondly, we decorate the source code of a previously built (or being built) implementation of the laptop store with the annotations. For instance in one .jsp file, which represents a view in our metamodel, we write the following annotations inside a comment:

```
@MetaModel = "view"
@Scenario = "insertLaptop"
```

This means that target artifact contains the necessary code for a web page, which is called a view in our Development Metamodel, and is responsible for the user interface of the use case scenario of adding a new laptop model to the store. Thirdly, for ensuring that each view is responsible for exactly one use case scenario, we check that any artifact containing the MetaModel tag with the "view" value contains only one Scenario tag and no Scenario has more than one associated "view".

With the help of a simple text-reader (annotation reader) class, the analysis of the source code and the implementation of checking the constraints expressed in the embedded DSL were easy and straightforward. As the last step we run the constraint checker tool against the sample implementation and analyze the feedback.

Concluding from the analysis results, it seemed that some of the constraints previously expressed to the developers via natural language descriptions had been misunderstood as all relevant artifacts to that constraint had deviated from the intent of the architect. In some other cases, it seemed as if the developers have temporarily forgotten the constraints and slipped from the architecture, as some part of the implementation were compatible with the architecture and some parts were not. In a real situation, this feedback could be immediately reported to the developers to prevent further deviation from the intended architect, and if industrialized, such a tool could be integrated to configuration management tools to make the testing process fully automated.

TABLE I.
AN INSURANCE COMPANY CASE STUDY RESULTS

Constraint	level	No.
No direct access to "implementation" classes	high	0
"Action" classes cannot call other actions' methods	high	3
"Action" and "service" classes must be stateless and thread-safe	high	0
Avoid `java.io.PrintStream.println()` in certain classes	low	14
Avoid `java.lang.Throwable.printStackTrace()`	low	61
Avoid leaving commented java code	low	12
No finalize method that only calls `super.finalize()`	low	1

In this approach, for introducing each new constraint, the architect needs to implement the "Ensures()" method of the Constraint interface, and our preliminary experiments show that this amount of implementation overhead is economically feasible for the architect. In other words, we believe that the amount of time and effort once spent by the architect implementing the "descriptive" DSL (handed to the developers as a guide map) and checking of the constraints to ensure the conformance is "worth the bother". This has made us confident enough to start the implementation of a framework based on the concept of the AST for the use of architects.

It is also worth mentioning that, the expression of such constraints, even simpler ones, is not only beneficial for novice or sub-professional programmers. As the experiment shows, developers experienced enough to appreciate and apply the constraints of the architect may also make occasional mistakes and deviate from the architecture, too. It is certainly better to have a tool that ensures the correctness and consistency of the source code, be it merely a backup procedure in highly professional teams.

IV. CASE STUDIES

In this section we discuss the application of the suggested approach in two real-world industrial projects.

A. *An Insurance Company Case Study*

This web-based system is used by one of the major insurance companies in middle-east Asia, containing approximately 100,000 lines of Java code containing about 500 classes.

Since this system had already been developed, we applied the suggested approach as a post-development analysis and evaluation mechanism, needed for long-term maintenance purposes. First, we developed a thorough understanding of the system's architecture by studying the few available software architecture documents (basically a Functional viewpoint document), reading source code, and communicating with some of the original architects and developers of the system. Second, we expressed our understanding of the system through some Development Metamodel diagrams and completed this expression via an internal java-based DSL. Third, we confirmed this expression of the system with the original architects of the system and carried out the constraint checking process to find inconsistencies between the final product and the intended architecture. We used annotations to meta-tag source code elements and used the Eclipse Test and Performance Tools Platform (TPTP 4.5) [12] to implement the constraints of the internal DSL.

The important constraints identified by the original architects, regarding the development process, were both low-level and high-level constraints. Low-level constraints were less architecturally important and we generally needed to investigate only one source code element to check the conformance of each constraint. Some of them were even characterized as team-specific coding conventions. However, the architects insisted on checking all low-level constraints as they believed any mismatch could result in either bugs or more expensive maintenance processes, due to less comprehensible source code. The high-level constraints were more architecturally significant and although fewer in number, they needed the investigation of several source code elements at the same time for each constraint.

Considering the relative maturity of this system, and the fact that it had already passed rigorous testing and evaluation processes, we, as well as the architects of the system, were stunned to see 91 instances of inconsistency belonging to 6 different constraints. As only 3 of the 91 instances were considered high-level (and all 3 belonging to one constraint), we believe the architects had been fairly successful in conveying the intended architecture via ordinary component-and-connector diagrams, and most developers had interpreted and implemented their intention correctly. Nevertheless, this case study demonstrates that the expression of the Development Metamodel and the constraints using a DSL can be beneficial even in successful projects and can point out inconsistencies that have not been caught during system testing. The remainig 89 inconsistencies were low-level but some of them were serious coding mistakes that could lead to definite bugs that had escaped unit tests (like the presence of e.printstackTrace() in several catch clauses as the default code template of the Eclipse framework). Table 1 shows a snippet of the final result.

Our experience with this case study has demonstrated that understanding every class file from an architectural point of view and the decoration of the code with correct annotations accounts for a major portion of implementation overhead of this method. In the next case study we will show how one could overcome this problem using abstract syntax trees, the notion of packages and naming conventions.

B. *A Financial Group CRM Case Study*

At the time of the initial rollout of this case study, this web-based Customer Relationship Management system was under construction for one major financial group. We deliberately decided to choose this case study to test the applicability of the suggested approach during the early stages of the development process and to see how amenable this approach is for agile development teams. Being at the early stages of the development phase, the system comprised of approximately 10,000 lines of Java code containing about 200 classes.

TABLE II.
A FINANCIAL GROUP CASE STUDY RESULTS

Constraint	level	No.
Controller classes in the presentation layer should not have dependency to UI classes	high	23
If more than one repositories are accessed in a single use case, a single Hibernate session object should be passed to all data access methods	high	5
Service implementation classes cannot directly access entity classes (only through repositories)	high	3
GWT UI elements named via `setName()` method for unit test	high	35
Only classes in server.rpc package can access `client.DTO` classes	high	11
Only controller classes in the presentation layer can call services (UI classes cannot call services directly)	high	18
No finalize method that only calls `super.finalize()`	low	0
Avoid leaving commented java code	low	30
Avoid `java.io.PrintStream.println()` in certain classes	low	16

As the system was under construction and the architects were actively involved in the day-to-day activities of the development, we did not have to go through time-consuming code reading and architecture discovery phases but rather, we were able to convey the Development Metamodel and the intended constraints of the architects through our DSL within a few days.

In such an early stage of development, especially with no conventional software architecture document (as in many agile development teams), the team were concerned about the developers' correct and uniform understanding of the intended architecture. Thus, for the first constraint checking iteration the team asked for 6 high-level architectural constraints as well as 4 low-level constraints. Fortunately, the developers had been mandated to conform to strict packaging rules. Thus, by introducing the notion of packages into the DSL, usage of naming conventions, and sophisticated AST manipulation facilities of Eclipse TPTP significant and JDT frameworks, we were able to implement the constraint checking functionalities with no need for annotations. In other words, we deduced each source code element's role in the architecture instead of meta-tagging each element manually. This method later proved to be extremely advantageous as it enabled us to further the progress the constraint checking process in parallel to the mainstream development process of the agile development team.

Since the system was still far from maturity and had not passed any testing phases, the staggering number of inconsistencies (148) in such small amount of code was not surprising to us. However, since 95 instances belonged to high-level or architectural constraints, this immediately alerted the architects of the flawed communication and interpretation of the intended architecture among the developers. The results urged the team to provide more detailed architecture descriptions and more frequent briefing sessions to prevent further architecture divergence. This early constraint checking phase were encouraging enough to make the team ask for repeating this constraint "introduction and checking" phase on a weekly basis. A summary of the results is shown in Table 2.

V. Related Work

There has been considerable research and practical work carried out in the field of software architecture description and documentation. For example, Kruchten's "4+1" method of the 1990s, has since evolved to form an important part of the Rational Unified Process and the basis of many modeling tools, including those of the Rational Corporation. Both [1] and [2] urge the architect to include the Development view in his architecture description. This paper seeks the same goals in documenting the Development View, but differs significantly from the mentioned methods in two respects:

- The Development Metamodel and the supporting DSL belong to a level best described as a "meta level", while the two methods either do not suggest any description language or their models are explained at the concrete model level, not "meta level". This generic description facility makes our approach promising for ultra-large scale information systems.
- The models they suggest is by no means tool-processable, as it can only be used by humans, while the model in this paper is amenable for processing and integrating into development tools.

As the suggested language in the current paper belongs to a meta level, it bears a close resemblance to the models and languages used in the field of Model Driven Development (MDD). However, the holy grail of the MDD is automatic code generation from models, while our main purpose is architecture description for use of the developers.

Apart from that, the fact that our Development Metamodel is in the architectural level makes it similar to Stahl and Völter's special flavor of MDD called Architecture-Centric MDSD [13]. In AC-MDSD, they show how one can leverage the abilities of the Functional viewpoint to automatically generate what they call the "infrastructure code" of a software system, which is The infrastructure code is the code corresponding to third-party infrastructural components used heavily in software systems and contains skeleton of the code. What makes AC-MDSD of special interest to us is the use of Domain-Specific Languages to describe the system's architecture in addition to the code generation process. They also provide the necessary modeling and code generation tools for the AC-MDSD approach based on the Eclipse framework, called openArchitectureWare [13]. In [14], Völter suggests using a programming model in seek of:

- Ensuring correct use of the architecture by the developers.
- Being able to review application code regularly and easily.
- Correct management of developers with different qualification with a single architecture.

A programming model describes how architecture is used from a developer's perspective. It is a How-To Guide that walks developers through the process of building an application. This is, indeed, the exact purpose of the Development Metamodel and the supporting DSL in the current paper. For such purposes Völter suggests the use of external XML-based DSLs but avoids specifying the DSL giving further information on it.

VI. Conclusion and Future Work

In this paper, we introduced the idea of the Development Metamodel as a means to guide the developers throughout the development process. We suggested switching the focus from the *software* to the *developers* by introducing a new expressive model; keeping a distance from conventional rules of component-and-connector or module structure models and instead describe anything (any element) related to the daily activities of the developers, be it an artifact, a concept or a constraint.

We showed how a lightweight (Java-based) embedded Domain-Specific Language can help the architect to describe his/her intended guidelines and constraints clearly and concisely. We also showed how this DSL, with the help of some annotations embedded manually in the source code by the developers, or with the help of Abstract Syntax Trees, can be a used to ensure the constraints in the final product.

However, the internal Java-based DSL has limitations in expressing the development view. We intend to overcome this problem and minimize the "noise code" of the DSL by designing a special-purpose external DSL, tailored to describe the development view of the software architecture clearly and concisely.

References

[1] P. Kruchten, "Architectural Blueprints - The "4+1" View Model of Software Architecture". *IEEE Software*, vol. 12, no. 6, November 1995, pages 42–50.

[2] N. Rozanski, E. Woods, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives,* Addison-Wesley, Boston, MA, USA: Addison-Wesley, 2005.

[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice,* 2nd ed., Addison Wesley, Boston, MA, USA, 2003.

[4] IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Standard 1471, 2000.

[5] C. Hofmeister, P.Kruchten, R. L. Nord, J. H. Obbink, A. Ran, P. America. "A general model of software architecture design derived from five industrial approaches", *Journal of Systems and Software*, vol. 80, no. 1, January 2007, pages 106-126.

[6] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995.

[7] M. Fowler, *Patterns of Enterprise Application Architecture,* Addison-Wesley Professional, 2002.

[8] A. Deursen, P. Klint, J. Visser, "Domain-specific languages: An annotated bibliography", *ACM SIGPLAN Notices*, vol. 35, issue 6, June 2000, pages 26–36.

[9] M. Mernik, J. Heering, A. M. Sloane, "When and how to develop domain-specific languages", *ACM Computing Surveys*, vol. 37, issue 4, December 2005, pages 316-344.

[10] S. Freeman, N. Pryce, "Evolving an embedded domain-specific language in Java", *OOPSLA Companion*, 2006, pages 855-865.

[11] D. Binkley, "Source Code Analysis: A Road Map. Future of Software Engineering". *FOSE* apos; 07, vol. 25, issue 23, May 2007, page 104 - 119.

[12] Available at: http://www.eclipse.org/tptp

[13] T. Stahl, M. Völter, *Model-Driven Software Development: Technology, Engineering, Management*, John Wiley & Sons, Ltd., Hoboken, NJ, USA, 2006.

[14] M. Völter, "Software Architecture: a Pattern Language for Building sustainable software architectures", Unpublished. Available: www.voelter.de/data/pub/ArchitecturePatterns.pdf