Towards a human-centric development method

Robert Deckers
Atom Free IT
Robert.Deckers@atomfreeit.com

Patricia Lago
VU University Amsterdam
P.Lago@vu.nl

Wan Fokkink
VU University Amsterdam
W.J.Fokkink@vu.nl

ABSTRACT
The business success factors of software have changed from functionality to quality properties (e.g., availability, security, sustainability) and development qualities (e.g., time-to-market, development cost, project predictability). However, methods and tools for domains-specific languages and model-driven development are still focused on the delivery of software functionality instead of the guarantee of quality. This papers outlines the requirements and vision of a novel research idea that treats qualities as first class elements of a human-oriented (rather than system-oriented) development method and framework. The main subjects are specification of quality, integrating and managing multiple quality domains, and enabling automatic transformations between specifications in those domains.

Keywords
Model-driven development, domain-specific language, software architecture, quality attribute, non-functional, language theory.

1. INTRODUCTION
This paper suggests researching a method and framework for integrating domain-specific languages (DSL), building consistent specifications in those languages, and transformations between those specifications. The focus lies on quality properties (a.k.a. non-functional properties). The starting point is the natural use of domain language, with own concepts and notations, by people involved (stakeholders) in the development of a software system. A domain language can be about any aspect of the system itself, of the system environment, or of the relation between them. Figure 1 shows examples of possible stakeholders and their domain

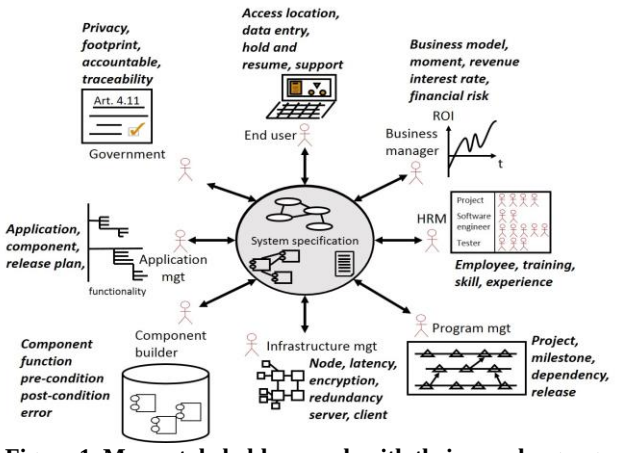

Figure 1. Many stakeholders, each with their own language

concepts. Instead of aiming for a complete system specification in source code, we suggest that all involved persons can speak their own language and record decisions in that language. A development method and framework should support the transformation between the different languages. The reasoning behind this is that people make better and more reusable decisions if they can specify them in their own domain-specific language. The integration of decisions and domain knowledge is how software development should be seen.

First, section 2 explains some issues with today's software development. Section 3 identifies solution directions for those issues, which forms the basis for the research idea. Next, section 4 discusses the novelty of the suggested research. We conclude with the envisioned research results.

2. PEOPLE ARE MORE IMPORTANT THAN MACHINES
This section elaborates on issues with today's software development and suggests some solution directions. Software development methods are focused too much on producing executable source code to feed a stupid machine. Instead, we should see software development as an accumulation of decisions made by humans with different knowledge, different opinions, and different stakes. Tools, methods, education, and frameworks should support this vision. Some of today's techniques like model-driven development (MDD), DSLs, design patterns, product line architectures, IDEs (e.g., Eclipse), and collaboration systems (e.g., Sharepoint, Wiki) solve a part of the issues, but they are not integrated and the theory to do so is lacking.

2.1 No support for your language
Software development has started as an academic specialism where software developers performed every task and needed to have all the required knowledge to do so. During the last sixty years software systems have become larger and the number of stakeholders and aspects to deal with during development has increased. Software development has become a large-scale industry that employs people in many different areas of expertise and at multiple education levels. These people each speak their own "language" with their idiom and sometimes also their own grammar and syntax. The practice is however that we still use (development) languages that are an aggregation of machine primitives. These languages have dominated software development since the beginning. They are still the main languages for development results and system specification. For example, BPMN and Java have still the same primitives as FORTRAN. The resulting disadvantage is that in many cases people have to transform the mental concepts of their own knowledge domain into machine-oriented concepts. For example, while-loops, classes, datatypes, and inheritance are typical programming language concepts that do not come natural to most people. The forced use of machine-oriented concepts hampers people in improving and applying their domain-specific knowledge and in making domain-specific decisions. *A development process must enable people to communicate and reason in their own domain language.*

2.2 An inconsistent view of a consistent world
Tools and methods, each with their own notations and artefacts, are already available and used for some of the domains in the development process. For example, requirements management tools, project management tools, modelling tools, code checkers, or predefined runtime libraries. These tools are mostly not integrated in a tool chain, although they overlap in the concepts they use and

have interdependencies. For example, a requirements engineer uses a requirements management tool and a tester uses a test tool. Although a test tool might allow you to refer to requirements, there is typically no support to guard the consistency between the requirements and the tests. The same holds for example for architectural requirements and the design patterns used to realize them. Consistency is often not even considered explicitly, let alone guaranteed. *A development method must help guarding the consistency of all products in the development process.*

2.3 Quality matters, but is hardly covered

The ratio between machine costs and development costs has lowered tremendously in sixty years. Computers have become much more powerful and much cheaper, while the salary of programmers has followed the average inflation rate. Quality properties are not the primary concepts of most development methods. In turn, the success factor of software has changed from functionality to quality in those last sixty years. In the early days of software development, a computer with software offered functionality that wasn't possible before. Nowadays, functionality of software is unique only for a short time and soon offered by more suppliers. The business success factor of software has changed to quality attributes like ease of use, availability, security, and sustainability, and to development qualities like time-to-market, development cost, or project predictability. In spite of that, development methods and tools are still focused on the delivery of software functionality instead of the guarantee of quality. There are hardly any engineering methods for quality properties and quality is often not specified at all. This clearly contradicts the just stated business importance of software- and development quality. A development method/framework should enable the engineering of quality. This includes specifying quality and transforming specifications of quality into system specifications (e.g. source code) and development specifications (e.g. a project plan).

Because needed quality is hardly specified and not clear to everyone involved, the quality properties of developed software are determined by personal beliefs and experience of the developers. In the case of many developers with different backgrounds, this results in inconsistent quality throughout a system and its development. Especially when different developers have built different system parts. Because quality is not specified explicitly, it cannot be proven, guaranteed, or engineered. *A development method must support the engineering of quality. This includes its specification, its realization, and the trade-offs between different qualities.*

2.4 Waste of knowledge

A developed software system is the result of a series of development decisions. Making good decisions may cost you significant time and money, e.g. because one must learn the domain and trade-off all kinds of concerns. Therefore, the reuse of a good decision in another context increases the return on investment of that decision. The benefits are potentially the highest if the reuse is automated (when applicable of course). To make a decision reusable in a useful way, it must be clear in what situations the decision is a good decision and what are the effects of the decision. Naturally, the more formal the decision specification, the more it is suited to be automated. Automation of decisions can be divided in two categories.

The first category is the reuse of elements that capture the result of the decision. Those elements can be part of the working system like a GUI library, a DBMS, or a rule engine. These can also be part of any intermediate product during development, like a standard set of security requirements, or a design pattern. This category of decision reuse occurs often in today's software development. Many pre-existing components are integrated in a piece of software. However, it is often unclear which decisions are reused and if they should have been in that particular case. For example, a decision might not be optimal due to a different user context or different runtime platform. It is also unclear if the integrated result is consistent. For example, functional integration of two software components does not say anything about the quality properties of the integration result.

The second category to automatically reuse decisions is via transformations between development products, which reuse the decision each time the transformation is executed. Examples are code generators, (pre-)compilers, or model-to-model transformations. This category is often used in current MDD approaches: a transformer turns an input specification into source code (or another intermediate format). The input model rarely explicitly defines quality properties. Rather, the transformer adds them each time it performs a transformation. This makes the transformer applicable only for systems that target the same quality properties. To judge the applicability of a transformer requires an explicit specification of the desired quality of the system, and the quality provided by the transformer.

The lack of awareness of development decisions that concern quality makes it difficult to explicitly exploit people's knowledge and investments in earlier developments. The problem with unawareness of quality and development decisions is also noticeable when you want to integrate software components made by different people. The chances are high that those components do not have matching quality properties. In that case you might want to reuse only a part of the development decisions. But, if all development decisions are only captured in the source code, then reusing only a part of the development decisions, or trying to reverse part of the decisions, becomes practically impossible. *Development methods must support the reuse and automation of any decision that can be made in a system's life-time: from idea, to design, to test, usage, and system deprecation.*

3. EXPLOIT AND INTEGRATE KNOWLEDGE

A formalism based on human thinking concepts instead of machine primitives should be the basis for a system's lifecycle that does not suffer from the issues stated in the previous section. This section expresses the starting point of this formalism and its practical applicability via a method and framework. The research builds upon existing theories and practices. Earlier literature research has shown that many partial elements already exist, but no integral solution seems to be available.

3.1 Integral specification of quality

There are standards like ISO25010 [14] that define a limited set of quality attributes to address quality properties. ISO25010 however, does not help in making specifications for those properties, nor in building a complete and consistent specification of the design problem, the development, or the system itself. Domain models

exist for some of the quality attributes, which can serve as a basis for a DSL, e.g., the security domain model in [6]. Also design patterns are available for some of the quality attributes. How to integrate specifications and design patterns for quality in a generic way is not clear yet. *The research result should contain requirements and specifications for a method and framework that enable integration and transformation of specifications of quality via DSLs and inter-domain rules.* Figure 2 illustrates this for some example domains. The research result must also contain examples of specifications of quality and explain how quality attributes can be added to the framework. Because there is no predefined standard way to divide the world into domains, the method should be applicable to any concept that a human can grasp, reason with, or specify. The result should ideally provide a method for specifying DSLs for any domain. Whether this is possible and what the limitations are is also part of our research.

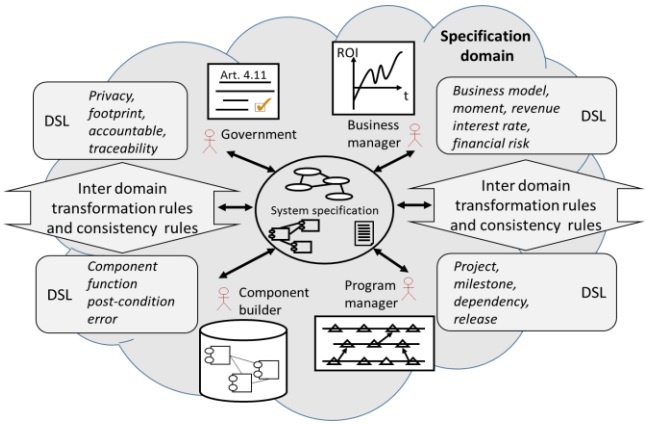

Figure 2: DSLs, specifications, and their relations.

3.2 Extend MDD to anything formal

MDD approaches have gained attention in the last years. They range from meta-modeling tools and meta-languages in which you build your own DSL and transformations (e.g., MetaEdit+, MPS), to more 4GL like platforms with a predefined modeling language and predefined application architecture (e.g., Mendix, Thinkwise). The models address mostly the application domain, data structure, and/or software functionality. Other system properties or stakeholder concerns are hardly modeled or even configurable. The result is that software systems built with such an MDD approach have fixed and implicit quality properties. This might be ok for some cases, but variable quality and trade-offs between quality attributes are desirable when an MDD approach aims at a large application domain. Also, quality should be predictable and explicitly known in many cases, for example when legislation demands it. *The research result should contain requirements and specifications of how a model of a quality can be automatically transformed into software specifications and/or software development specifications.* The result should also show examples of such a transformation.

3.3 A method based on multiple formalisms

Any specification, formal or informal, can only be (partially) understood if the language in which the specification is stated, is (partially) understood. If a specification must be processed by a machine, then that specification must be formal for at least the part that must be processed. Otherwise, a machine can simply not parse it. Some well-known formalisms exist that are used for specification languages, like Petri Nets, state machines, process algebra, set theory, and predicate logic. These mathematical mechanisms are applied in various specification languages that allow formal specification of software structures and software behavior. However, the authors have not found a language and method that allows to formalize anything that a human can understand and that could be useful for the realization of a software system. If we want to specify anything a human is able to communicate and reason about it in a systematic way, then we need at least a language formalism that allows extension to any human perceivable idea. How this can be achieved is an important challenge for this research.

The idea is that a useful formalism for the specification of any quality is based on any of the above mentioned formalisms combined with a method to give clear semantics to the terms/words in a DSL. This could be realized via the Universal Grammar theory by Noam Chomsky [3]. Chomsky's theory unifies the real world, the mental concepts in a human's brain, and the language we use to communicate about it. It also states that a person is born with the mental ability to create and use language. *The research result should explain the basic formalism and show how it can be used for specifications of qualities that are made by a human and processed by a machine.* It should also explain how a development language and method could be formalized gradually.

4. A NEW PERSPECTIVE ON SOFTWARE DEVELOPMENT

This section summarizes the vision behind the proposed research and its novelty. It concludes with a discussion on related work.

4.1 The vision

Software development is a human task. Software development methods must serve a process where different people with different expertise work together to achieve a result instead of focusing on the realization of a machine-readable formula. A development method should support the involved people in making the right decisions, and once a (difficult and expensive) development decision has been made it must be possible to reuse it in other developments. What the ingredients and structure of such a method are is the topic of this research.

4.2 The new idea

The core of this research is to investigate what are the ingredients for a method and framework that enables the guarantee of quality properties of a software system and its development. The method must support the creation of DSLs for quality attributes and system aspects, and the specification of requirements and models for those attributes and design patterns for the system aspects. Via MDD techniques, software systems with the specified quality can be achieved in an automatic way. The basic notion is that any statement about something (under development) is at least based on a human notion/thought. That notion is a concept perceived by one or more people involved in the development. This can be any concept from the notion of a whole system, to a single line of code or one requirement, to a particular need that is independent from the developed system. Any language or specification is based on the prerequisite that at least one person is able to perceive or handle it. This idea should be the basis of languages and methods for software development that support human reasoning and human cooperation.

The idea for this research is the result of over twenty years of experience and education in software engineering by the authors. Experience with software architecture and system architecture has taught that quality properties and the corresponding design decisions are stable throughout an application domain, but are hardly treated as such. Experience in MDD for business information systems, medical systems, and embedded software has

shown that only a few of the people involved in a development project are allowed to reason and communicate in their own domain language. This should be possible for everyone involved.

4.3 Why is it new?

The research subject combines engineering, software architecture, DSLs, MDD. There are no methods and tools that simultaneously:

- Apply models of quality properties as input for transformations into working software.
- Allow the specification of any quality in a formal language.
- Provide mechanisms to extend a specification language with any domain, and for quality domains in particular.

4.4 Related work

The idea to integrate and unify all specifications in a development process and let all stakeholders have their own view on it, is not new. It is at least described by Deckers and Steeghs [5] and roughly by Evans [6]. Business applications use several mechanisms (e.g. MVC, Data Vault) to provide users and management with a consistent view on a diverse set of data. We aim to apply this principle also to software development and quality in particular.

In literature on software architecture the importance of quality and the use of specific viewpoints for quality is emphasized (e.g., [2][4] [17]). But, a clear method on how to engineer and guarantee quality in a systematic way (via DSLs) is not found.

Books on DSLs and MDD (e.g., [1][6][9][13][15][18]) present methods and examples of DSLs and MDD techniques. These examples all consider the application domain or a software design aspect. System quality is determined by the transformation rules, but is not explicitly addressed. A nice exception is the domain model for security by Firesmith [8]. This model offers concepts to reason and communicate about security. These concepts form (implicitly) a DSL for security. We will investigate what type of elements should be in a domain model that is useful for the specification of a particular quality and transformations to other domain.

Some issues with today's MDD techniques and exploitation of development knowledge are for example addressed in [7],[11] [12], and [16]. These issues are an input for our research.

Well known books on design patterns (e.g., [10],[11]) have in common that they explain patterns in the software domain. But, they offer no traceable and consistent solution for the relation between a design pattern and its targeted quality.

Besides the technology the authors would also like to investigate the psychological and language aspect of taking development decisions. This aspects seems to been neglected by literature in the field of software engineering.

5. THE TARGETED RESULT

The targeted research result is threefold: a method for defining DSLs and specifications for quality attribute and their integration, a framework for specifying and performing transformations between DSLs, and complete examples for a set of – not yet chosen - quality attributes.

The basis of the suggested method is a basic language and method to enable the specification of quality and transformations between specifications. The starting point of this theory will be the analysis of existing meta-models and formalisms of development languages. Solution directions given by the language theory from Chomsky [3] and existing metamodels, like ECore of Eclipse, will be analyzed for usefulness. This part will probably lead to requirements and principles for DSLs and modeling tools.

A framework for specification and transformation will be investigated. The usefulness of existing IDEs and software engineering tools (e.g., Eclipse, UML modelling tools) will be investigated on extendibility, consistency, and manageability of development knowledge. Possible results will be new requirements and principles for those.

The research result should also contain a complete example in which several qualities are specified via DSLs and integrated via MDD techniques.

6. REFERENCES

[1] Brambilla, M., Cabot, J., Wimmer M., Model-Driven Software Engineering in Practice, Morgan & Claypool, 2012

[2] Buschmann et al. Pattern oriented software architecture, John Wiley & Sons Inc., 1995

[3] Chomsky, N., Reflections on Language, Pantheon books, New York, 1975

[4] Clements, P., et al., Documenting Software Architecture: Views and Beyond, Pearson Education Inc., 2002

[5] Deckers, R., Steeghs, R., DYA|Software, architectuuraanpak voor bedrijfskritische applicaties. Sogeti, Vianen, 2010

[6] Evans, E., Domain-Driven Design, Prentice hall, 2003

[7] Farenhorst, R., Boer, R. de, Architectural Knowledge Management: Supporting Architects and Auditors, VU Amsterdam 2009

[8] Firesmith, D., Specifying Reusable Security Requirements, in Journal of Object Technology, vol. 3, no. 1, January-February 2004, pp. 61-75

[9] Fowler, M., Domain-Specific Languages, Addison-Wesley, 2010

[10] Fowler, M., Patterns of Enterprise Application Architecture, Addison-Wesley, 2002

[11] Gamma E., et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994

[12] Gonzalez-Perez, C., Henderson-Sellers, B., Metamodelling for software engineering, John Wiley & Sons Inc, 2008

[13] Hemel, Z. Methods and Techniques for the Design and implementation of Domain-Specific Languages. TU Delft, 2012

[14] ISO/IEC 25010, International Standard. Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models

[15] Kelly, S., Tolvanen J., Domain Specific Modelling, John Wiley & Sons Inc, 2008

[16] Malavolta I., et al. Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies, IEEE Transactions on software engineering, Vol. 36, No 1, Jan 2010, pag 119-140.

[17] Rozanski, N., Woods, E., Software Systems Architecture, Addison-Wesley Professional, 2005

[18] Voelter, M., DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. 2013. www.dslbook.org