Generic Quality-Aware Refactoring and Co-Refactoring in Heterogeneous Model Environments

Kurzfassung der Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Jan Reimann
geboren am 16.03.1982 in Potsdam

Gutachter:
Prof. Dr. rer. nat. habil. Uwe Aßmann
(Technische Universität Dresden)

Prof. Dr. rer. nat. Ralf Reussner
(Karlsruhe Institute of Technology)

Fachreferent:
Jun.-Prof. Dr.-Ing. Thomas Schlegel
(Technische Universität Dresden)

Dresden im Mai 2015

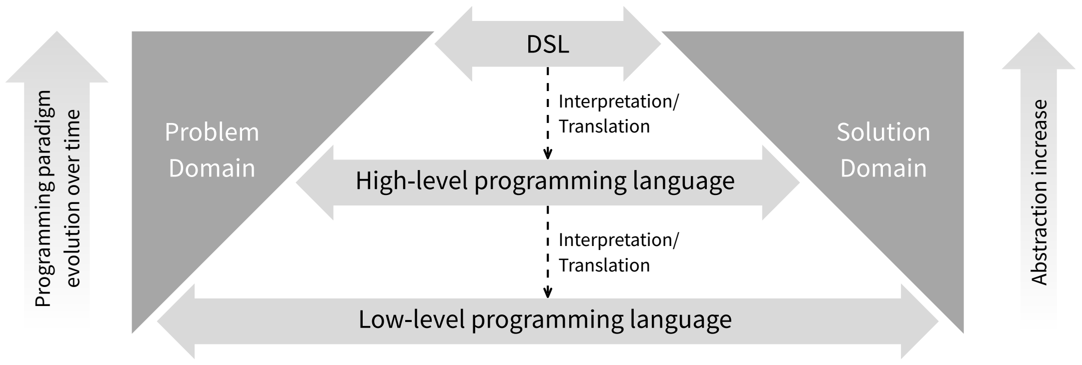

DSL
Interpretation/ Translation
Problem Domain
High-level programming language
Solution Domain
Interpretation/ Translation
Low-level programming language
Programming paradigm evolution over time
Abstraction increase

Figure 1: Abstraction Gap based on [KT08, p. 16].

1 Introduction

The techniques of *abstraction* and *specialisation* are used in almost every area of our daily lives and became apparent in languages used to solve problems in many scientific or engineering disciplines. Languages, being dedicated to a particular problem space, are called *Domain-Specific Languages (DSLs)*. Similar progress has happened in the discipline of computer science. In the beginning, instruction sets of programming languages corresponded heavily to those of the machine which the programme was intended to run on. Therefore, programmers had to bridge a big gap between the domain of the initial problem and the domain of the problem-solving programme (the solution). Between these domains, namely *problem domain* and *solution domain*, the so-called *abstraction gap* [KT08; IM10] was very big, because the used concepts of a solution idea and the particular solution implementation (machine instructions) are very different. Figure 1 illustrates this contemplation in the lower part. Realising an implementation was very complex until high-level programming languages (such as C++ or Java) came out onto the market. These languages raised the level of abstraction and increased developer's productivity by 450 % [KT08]. This progress got another impetus, when it was recognized that taking advantage of the concept of DSLs can raise the abstraction level even more. Thus, DSLs arrived in computer science and information technology (IT), enabling developers to implement solutions closer to the problem domain. Most often, these languages are delivered with some kind of compiler or interpreter to map abstract instructions to less abstract ones in an already known and executable formalism. Such a mapping specifies the translational or interpretative semantics of a DSL and defines how to resolve DSL instances to an instance of the underlying layer in Figure 1: compilation to another language, or direct execution respectively [EvSV+13].

In practice, there are heaps of DSLs supporting daily developer's and engineer's work, and their number is still growing. One reason for the quantitative increase is a development paradigm, which became more and more popular in the last decade. This paradigm is called *Model-Driven Software Development [SVB+06] (MDSD)* and takes advantage of formal artefacts to create instances of abstract concepts. The following short example illustrates the MDSD paradigm. Consider, e.g., a detached house on the one hand and a tool shed on the other hand. Both are buildings but have distinct purposes. The former is for living and the latter is for storing gardening tools. From an abstract point of view both exemplars have several things in common. Both are *buildings*, have a *door* and a *roof*. Unless a building has no roof or door it is not considered to be a building. Single parts of the houses are conceptualised (e.g. *roof* or *door*) and a rule is given under which circumstances a set of single parts is meant to be a *building*. These concepts and the rule are the formal base of both buildings. They enable us to reflect about houses and to evaluate, if something belongs to the domain *buildings* or not. This formalisation is called a *metamodel*, because it specifies properties of all its instances: the *models*. Since metamodels have a higher level of abstraction than their models, they can be used, e.g., to generate other artefacts from the models, because the generation rules can be specified on top of the metamodel's concepts. A metamodel is considered to be the abstract formal grounding of a DSL, not focussing the concrete, but on the abstract syntax of the DSL's instances, the models. Coming back to the small example, such a generation rule could automatically produce a list of all consumed materials for a building, instead of having to write the list manually.

In this work, only model-based DSLs are considered, since they allow for abstraction of languages to their constructional concepts. The abstract syntax of a DSL is specified in its metamodel and their instances (the models) conform to it.

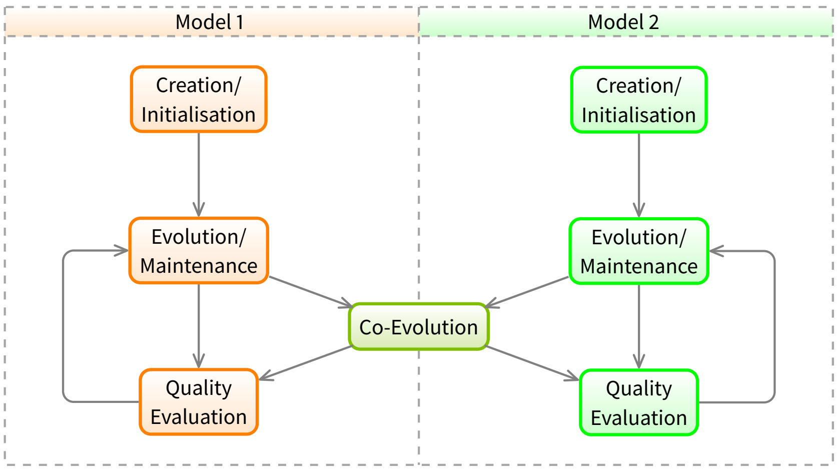
2 Problems and Objectives

2.1 Problems

Software complexity rose and still rises [Leh96; Kle09]. Software design evolved as an essential tool to cope with that complexity before developing the software itself. Therefore, the design is the base for understanding a software system and for the early identification of problems instead of living with them afterwards [Pfl98]. Opdyke and Johnson investigated on this and introduced the term *refactoring* in [OJ90], which signifies the restructuring of code while preserving its semantics to improve the design of a software. Later, Opdyke published a first catalogue of refactorings in his dissertation [Opd92] forming the foundation of refactoring tools in nowadays *integrated development environments (IDEs)*. These IDEs came into the market to support developers in managing the complexity of developed software so that qualities, such as *reusability*, *readability*, or *comprehensibility*, are improved. In the beginning, refactorings had to be applied manually after every regression, which was error-prone and required huge effort. Later, the IDEs were equipped with mature refactoring tools enabling developers to execute them (semi-)automatically. Today, all modern IDEs support refactoring in one or the other modality [XS06]. For high-level programming languages code refactoring is well investigated and can be applied easily.

In recent years *Language Workbenches (LWs)* [Fow05] emerged enabling the development of mature tools for using DSLs. We call these tools *Domain-Specific Language Environments (DSLEs)*. According to [EvSV+13], a DSLE usually consists of an editor, syntax highlighting, a parser, language-specific refactoring, and semantic services as reference resolution or error marking. All these features are not new since they were adopted from programming language IDEs [Fow05]. Many of them can be derived from a DSL's abstract or concrete syntax. But one of the biggest problems in nowadays LWs is still the lack of adequate refactoring support in the produced DSLE [KV10; Mer10; EvSV+13; VWT+14]. As a result, developers cannot apply refactorings in DSLEs, as they are used to it from modern IDEs.

Figure 2: Overview of model life cycle in MDSD.



The main problem regarding the lack of appropriate refactoring support can be subdivided into the following three issues.

Only structural information is available in metamodel DSLs only provide structural information in their metamodels, especially knowledge about how concepts relate to each other. This information is only of static nature and is regarded as the abstract syntax. Consequently, it is not possible to establish a relation to a concept of *quality* representing information about which quality a model actually has. Without such a fact it is not possible to interrelate qualities with refactorings, which results in the fact that one cannot specify an indicator expressing *when* and *what* structure to refactor. As a consequence, refactorings would be executed *randomly* without a formal grounding. Furthermore, refactorings cannot be derived automatically for DSLs, because they do not only depend on the structure, but on the specifics of the particular language as well. DSLEs are not able to provide DSL-specific mature refactoring tools and they omit the aspect of evolution completely. As a consequence, the effort to specify refactorings for a particular DSL is huge. In the worst case, the restructurings a refactoring comprises must be applied manually. This means that the consistency of the model to be refactored may break, because this manual process is error-prone.

Figure 2 summarizes this subproblem on the left hand side for *Model 1*. This DSL instance is created before it undergoes a process of evolution. Modifications are applied. Afterwards, the quality should be evaluated for being able to give evidence which refactoring could be applied upon which structure for improving the overall quality of the model. The process of evolution and quality evaluation is a cycle which might stop when a model is not modified anymore and its quality requirements are satisfied.

DSL is regarded as isolated The second subproblem is the fact that, when DSLs are engineered, potential relations to other DSLs are not taken into account, thus, the DSL is considered as isolated. Models of a DSL might relate to models of other DSLs. Those connections then can point to instances of arbitrary DSLs and cannot be foreseen. In such a case, interdependencies between those models arise. As a consequence, a dependent model is influenced by modifications of the other model. Figure 2 illustrates this relationship for *Model 1* and *Model 2* exemplarily. If *Model 2* evolves in terms of a refactoring its quality must be evaluated again. Since *Model 1* depends on the evolved model it must co-evolve. This co-evolution is highly dependent on the concrete modifications in *Model 2* and the current state of *Model 1*. This is a problem when different DSLs are considered isolated, because an evolution of one model can violate the consistency of a dependent model. In the following, environments incorporating several different DSLs are called *Multi-Language Development Environment (MLDE)* [PW15].

Another consequence of DSL isolation is that refactorings cannot be reused across different DSLs. For instance, the same semantics-preserving restructuring is available for different programming languages. The difference between these refactorings is not the specifics of the refactoring itself but the language which it is applied to. Thus, from an abstract point of view, the same steps are executed in different languages. The same holds for DSLs. If refactorings cannot be reused they must be specified and implemented anew for every different DSL.

Appropriate refactorings are dependent on DSL designer's preferences Apart from the two subproblems above, the decision which refactorings to specify for a DSL highly depends on the preferences of the DSL designer. On the one hand, the DSL designer must determine which structures are suitable for refactoring. Therefore, the abstract syntax must be examined and feasible relations between concepts have to be found for establishing potential candidates for refactorings.

On the other hand, the technical background of the intended DSL users should be taken into consideration by the DSL designer. Depending on the target group, refactorings of different maturity could be provided. As a consequence, this subproblem again leads to the fact that DSL-specific refactorings cannot be derived automatically, since the decision which refactorings to provide is highly subjective.

2.2 Objectives

These problems find expression in the following objectives which will be covered in this thesis.

Generic Specification of Refactorings DSLs should not be regarded as isolated, thus it is not efficient to specify and implement a refactoring for different DSLs anew, although the same modifications are applied except that the target language is different. The same refactorings must be reusable in different DSLs from an abstract point of view. Consequently, the specification of refactorings should be independent from the target language which it is intended to be applied to. Thus, an approach for the generic specification of refactorings is needed.

DSL-Specific Instantiation of Refactorings When refactorings can be specified generically, an approach is needed that supports the declaration of what a generic refactoring means for a particular DSL. This is the consequence of the previous objective and comprises the DSL-specific instantiation of generic refactorings.

Explicit Relation Between Refactoring Candidates, Refactorings and Qualities In [FBB+99], Fowler *et al.* defined structures suggesting the application of a particular refactoring as *bad smells*. The presence of a bad smell is a refactoring candidate, because it deteriorates specific qualities and the execution of a refactoring might improve them [FBB+99; SSL01; MTM07; Als09]. The problem of Fowler *et al.* 's term is that it has not been defined precisely. Furthermore, the connection to qualities and refactorings is only implicit. Hence, it is not possible to give evidence about what *smelling* structures influence which quality negatively and can be resolved by which refactoring [MTM07]. That's why a precise definition of a *bad smell* in the context of MDSD is needed, explicitly relating refactoring candidates, refactorings and qualities to allow for automatic detection and resolution.

Specification of Dependent Modifications A DSL and its models can have interdependencies to models of other DSLs in MLDEs. If a model evolves, it might have effects on dependent models. To avoid violation of consistency of the dependent models, an approach for specification of dependent modifications is needed. Since the subsequent changes must not alter the dependent model semantics such modifications are considered to be refactorings. In addition, the subsequent refactorings depend on a preceding refactoring and are therefore called *co-refactoring*. Such a specification should enable the mapping of preceding modifications in a model of one DSL to succeeding modifications in a dependent model of another DSL.

Detection of Dependent Models Apart from the specification of dependent modifications, dependent models themselves must be detectable in MLDEs. This objective contains two essential parts. First, dependencies between models must be tracked. Second, it must be recognized when a tracked model evolves, what modifications occurred and which other models they have influence on. After this detection process, the dependent modifications must be applied to the dependent models.

3 Contributions in Brief

In the following, the thesis' main contributions are presented and their progress beyond the state-of-the-art is justified.

Role-Based Generic Model Refactoring (Chapter 4)

To provide means for language-independent [MTM07; TMM08; MMBJ09] specification of refactorings, an appropriate abstraction mechanism is essential. For this reason, a first approach of abstraction over the desired refactorings is contributed by means of *role modelling*. A *role model* represents a dedicated view of objects in an certain context and the collaborations of the objects within this context [RWL96; RG98]. In this thesis, a role model defines the participants of a refactoring and their collaborations, independent from the target DSL which it should be made available in. The different *context* of the same generic refactoring is considered the particular refactoring of a specific DSL. A *refactoring specification* defines the intrinsic transformation a generic refactoring should execute. To instantiate a generic refactoring for a specific DSL, only

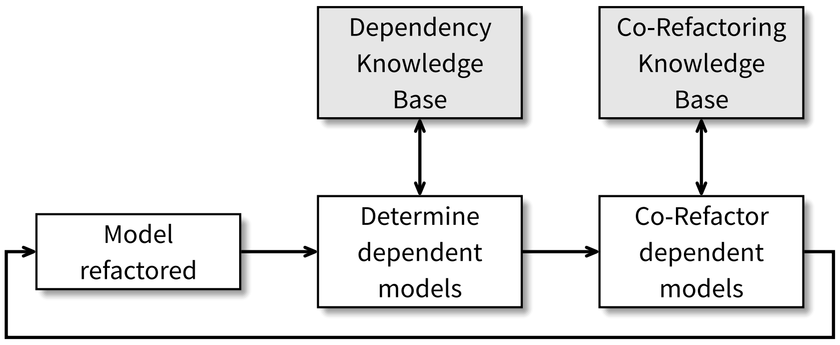

Dependency Knowledge Base
Co-Refactoring Knowledge Base
Model refactored
Determine dependent models
Co-Refactor dependent models

Figure 3: Schematic workflow of Co-Refactoring.

a mapping of the role model to a certain structure of the DSL's metamodel must be provided. Thus, the transformation is reused. This contribution is extends our work published in [Rei10; RSA10; RSA13].

The limitations of related approaches of generic refactorings is that they abstract from the DSLs which refactorings should be provided for. Thus, they are constrained in the sense that all DSLs must be similar to the concepts of the language abstraction. These approaches are too static and arbitrary DSLs are not supported.

Role-Based Quality Smells as Refactoring Indicator (Chapter 6)

The term *bad smell* [FBB+99] is too imprecise, and an explicit connection of structures violating particular quality requirements is omitted completely. Therefore, the concept of *quality smell* is contributed. A quality smell establishes an explicit relation between model deficiencies, their deteriorating qualities and resolving refactorings. A concrete occurrence of a quality smell can trace it back to the causing elements and suggests refactorings potentially resolving the occurrence. Such a suggestion is achieved by defining roles of interest for a concrete quality smell which then are mapped to the roles used in a resolving refactoring. This approach allows developers for focussing specific qualities in isolation. This contribution is extends our work published in [RA13; RBA14].

The limitations of related work is that there are approaches enabling detection and resolution of deficiencies in models but none of them correlates qualities, model deficiencies and resolving refactorings explicitly. We argue that this relation is essential for a quality-aware development and engineering life cycle.

Role-Based Co-Refactoring of Dependent Models (Chapter 8)

Usually models do not occur isolated, but have dependencies on other models or other models are dependent on them. To ensure consistency of dependent models, they must be co-refactored. Our contribution is divided into two parts according to the workflow depicted in Fig. 3.

First, dependent models and elements within them can be detected by means of a *Dependency Knowledge Base (DK-Base).* Therefore, we defined four categories of potential model dependencies and provide a logics-based approach to reveal them. Explicit dependencies are persisted in the DK-Base and the others are determined by querying the DK-Base. A part of determining dependencies is published in [PRW14].

Second, the intrinsic co-refactoring approach is realised by means of *Co-Refactoring Specifications (Co-RefSpecs)*, which populate a *Co-Refactoring Knowledge Base (CoRK-Base).* A Co-RefSpec refers an incoming initiating refactoring, a condition, and an outgoing triggered co-refactoring. Again, the roles of an initiating refactoring are bound to roles of a triggered refactoring. As a consequence, the mapping of roles prevents the user from providing required input for an outgoing refactoring which is dependent from an incoming refactoring.

The limitations of existing approaches is that none of them can be used as is for co-refactoring of models. Either they have limitations regarding dependency detection, or they are not generic enough to apply them in heterogeneous model environments such as MLDEs.

4 Conclusion

This thesis shows,that the concept of role models is beneficial for all of our contributions. For the generic refactoring approach, we use role models to capture structural constraints of participants of a refactoring in a language-independent manner. A role mapping must be provided to map roles to metaclasses of a particular target metamodel in order to enable a generic refactoring in a concrete language. For quality smells, one can define a role model to declare certain participants of a quality smell that might be of interest. The elements which are bound to the roles in a present quality smell then are passed to a resolving refactoring. To co-refactor models, the used roles from an incoming refactoring are related to the used roles of an outgoing refactoring. Thus, role models rendered as a very powerful abstraction mechanism in the scenario of quality-aware model refactoring and co-refactoring in MLDEs. As a consequence, role models can be considered as some kind of *interfaces* for models which can be used for loosely coupled interaction. The interaction then can be specified just depending on the roles, regardless the context of interaction such as a refactoring, the detection of a quality smell or a co-refactoring.

References

[Als09] M. Alshayeb, “Empirical investigation of refactoring effect on software quality”, *Information and Software Technology*, vol. 51, no. 9, pp. 1319–1326, 2009, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.04.002 (cit. on p. 6).

[EvSV+13] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth and J. van der Woning, “The State of the Art in Language Workbenches”, in *Software Language Engineering*, ser. Lecture Notes in Computer Science, M. Erwig, R. F. Paige and E. Van Wyk, Eds., vol. 8225, Springer International Publishing, 2013, pp. 197–217, ISBN: 978-3-319-02653-4. DOI: 10.1007/978-3-319-02654-1_11 (cit. on pp. 2, 3).

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999 (cit. on pp. 6, 7).

[Fow05] M. Fowler, "Language Workbenches: The Killer-App for Domain Specific Languages?", 2005, Available: `http://www.martinfowler.com/articles/language-Workbench.html` (cit. on p. 3).

[IM10] J. L. C. Izquierdo and J. G. Molina, "An Architecture-Driven Modernization Tool for Calculating Metrics", *Software, IEEE*, vol. 27, no. 4, pp. 37–43, Jul. 2010, ISSN: 0740-7459. DOI: `10.1109/MS.2010.61` (cit. on p. 2).

[Kle09] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels.* Pearson Education, 2009, ISBN: 0321553454 (cit. on p. 3).

[KT08] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation.* John Wiley & Sons, 2008 (cit. on p. 2).

[KV10] L. C. L. Kats and E. Visser, "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs", in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, M. C. Rinard, Ed., Reno/Tahoe, Nevada: ACM, 2010, pp. 444–463, ISBN: 978-1-4503-0203-6. DOI: `10.1145/1869459.1869497` (cit. on p. 3).

[Leh96] M. M. Lehman, "Laws of Software Evolution Revisited", in *Lecture Notes in Computer Science*, C. Montangero, Ed., vol. 1149, Springer Berlin Heidelberg, 1996, pp. 108–124, ISBN: 978-3-540-61771-6. DOI: `10.1007/BFb0017737` (cit. on p. 3).

[Mer10] B. Merkle, "Textual Modeling Tools: Overview and Comparison of Language Workbenches", in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '10, Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 139–148, ISBN: 978-1-4503-0240-1. DOI: `10.1145/1869542.1869564` (cit. on p. 3).

[MMBJ09] N. Moha, V. Mahé, O. Barais and J.-M. Jézéquel, "Generic Model Refactorings", in *MODELS*, A. Schürr and B. Selic, Eds., ser. Lecture Notes in Computer Science, vol. 5795, Denver, USA: Springer, Oct. 2009, pp. 628–643, ISBN: 978-3-642-04424-3. DOI: `10.1007/978-3-642-04425-0_50` (cit. on p. 6).

[MTM07] T. Mens, G. Taentzer and D. Müller, "Challenges in Model Refactoring", in *Proceedings of the 1st Workshop on Refactoring Tools*, University of Berlin, 2007 (cit. on p. 6).

[OJ90] W. F. Opdyke and R. E. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems", in *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, Sep. 1990 (cit. on p. 3).

[Opd92] W. F. Opdyke, "Refactoring Object-Oriented Frameworks", PhD thesis, University of Illinois at Urbana-Champaign, 1992 (cit. on p. 3).

[Pfl98] S. L. Pfleeger, *Software Engineering: Theory and Practice.* Prentice Hall, 1998 (cit. on p. 3).

[PRW14] R.-H. Pfeiffer, J. Reimann and A. Wąsowski, "Language-Independent Traceability with Lässig", in, ser. Lecture Notes in Computer Science, J. Cabot and J. Rubin, Eds., vol. 8569, Springer International Publishing, 2014, pp. 148–163, ISBN: 978-3-319-09194-5. DOI: 10.1007/978-3-319-09195-2_10 (cit. on p. 8).

[PW15] R.-H. Pfeiffer and A. Wąsowski, "The design space of multi-language development environments", English, *Software & Systems Modeling*, vol. 14, no. 1, pp. 383–411, 2015, ISSN: 1619-1366. DOI: 10.1007/s10270-013-0376-y (cit. on p. 5).

[RA13] J. Reimann and U. Aßmann, "Quality-Aware Refactoring for Early Detection and Resolution of Energy Deficiencies", in *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, ser. UCC '13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 321–326, ISBN: 978-0-7695-5152-4. DOI: 10.1109/UCC.2013.70 (cit. on p. 7).

[RBA14] J. Reimann, M. Brylski and U. Aßmann, "A Tool-Supported Quality Smell Catalogue For Android Developers", in *Proceedings of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung – MMSM 2014*, 2014 (cit. on p. 7).

[Rei10] J. Reimann, "Generisches Modellrefactoring für EMFText", Diploma Thesis, Technische Universität Dresden, 2010. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-67762 (cit. on p. 7).

[RG98] D. Riehle and T. Gross, "Role Model Based Framework Design and Integration", in *Proc. of OOPSLA '98*, Vancouver, British Columbia, Canada: ACM, 1998, pp. 117–133, ISBN: 1-58113-005-8. DOI: 10.1145/286936.286951 (cit. on p. 6).

[RSA10] J. Reimann, M. Seifert and U. Aßmann, "Role-based Generic Model Refactoring", in *Model Driven Engineering Languages and Systems - 13th International Conference, MoDELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II*, D. C. Petriu, N. Rouquette and Ø. Haugen, Eds., ser. Lecture Notes in Computer Science, vol. 6395, Springer, 2010, pp. 78–92. DOI: 10.1007/978-3-642-16129-2_7 (cit. on p. 7).

[RSA13] J. Reimann, M. Seifert and U. Aßmann, "On the reuse and recommendation of model refactoring specifications", English, *Software & Systems Modeling*, vol. 12, no. 3, pp. 579–596, 2013, ISSN: 1619-1366. DOI: 10.1007/s10270-012-0243-2 (cit. on p. 7).

[RWL96] T. Reenskaug, P. Wold and O. A. Lehne, *Working with objects – The OOram Software Engineering Method.* 1996. [Online]. Available: http://heim.ifi.uio.no/~trygver/1996/book/WorkingWithObjects (cit. on p. 6).

[SSL01] F. Simon, F. Steinbrückner and C. Lewerentz, "Metrics Based Refactoring", in *Proceedings of Fifth European Conference on Software Maintenance and Reengineering*, CSMR 2001, 2001, pp. 30–38 (cit. on p. 6).

[SVB+06] T. Stahl, M. Völter, J. Bettin, A. Haase and S. Helsen, *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006 (cit. on p. 2).

[TMM08] G. Taentzer, D. Müller and T. Mens, "Specifying Domain-Specific Refactorings for AndroMDA Based on Graph Transformation", in *Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*, Berlin, Heidelberg: Springer, 2008, pp. 104–119, ISBN: 978-3-540-89019-5. DOI: 10.1007/978-3-540-89020-1_9 (cit. on p. 6).

[VWT+14] E. Visser, G. Wachsmuth, A. Tolmach, P. Neron, V. Vergu, A. Passalaqua and G. Konat, “A Language Designer’s Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs”, in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2014, Portland, Oregon, USA: ACM, 2014, pp. 95–111, isbn: 978-1-4503-3210-1. doi: `10.1145/2661136.2661149` (cit. on p. 3).

[XS06] Z. Xing and E. Stroulia, “Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study”, in *22nd IEEE International Conference on Software Maintenance (ICSM) 2006*, Sep. 2006, pp. 458–468. doi: `10.1109/ICSM.2006.52` (cit. on p. 3).