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

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)

Tag der Einreichung: 7. Mai 2015
Tag der Verteidigung: 9. Juli 2015

Confirmation

I confirm that I independently prepared this thesis with the title *Generic Quality-Aware Refactoring and Co-Refactoring in Heterogeneous Model Environments* and that I used only the references and auxiliary means indicated in the thesis.

Dresden, 7th May 2015

Dipl.-Inf. Jan Reimann

Abstract

Software has been subject to change, at all times, in order to make parts of it, for instance, more reusable, better to understand by humans, or to increase efficiency from a certain point of view. Restructurings of existing software can be complex. To prevent developers from doing this manually, they got tools at hand being able to apply such restructurings automatically. These automatic changes of existing software to improve quality while preserving its behaviour is called *refactoring*. Refactoring is well investigated for programming languages and mature tools exist for executing refactorings in *integrated development environments (IDEs)*.

In recent years, the development paradigm of *Model-Driven Software Development (MDSD)* became more and more popular and we experience a shift in the sense that development artefacts are considered as *models* which conform to metamodels. This can be understood as abstraction, which resulted in the trend that a plethora of new so-called model-based *Domain-Specific Languages (DSLs)* arose. DSLs have become an integral part in the MDSD and it is obvious that models are subject to change, as well. Thus, refactoring support is required for DSLs in order to prevent users from doing it manually.

The problem is that the amount of DSLs is huge and refactorings should not be implemented for new for each of them, since they are quite similar from an abstract viewing. Existing approaches abstract from the target language, which is not flexible enough because some assumptions about the languages have to be made and arbitrary DSLs are not supported. Furthermore, the relation between a strategy which finds model deficiencies that should be improved, a resolving refactoring, and the improved quality is only implicit. Focussing on a particular quality and only detecting those deficiencies deteriorating this quality is difficult, and elements of detected deficient structures cannot be referred to in the resolving refactoring. In addition, heterogeneous models in an IDE might be connected physically or logically, thus, they are dependent. Finding such connections is difficult and can hardly be achieved manually. Applying a restructuring in a model implied by a refactoring in a dependent model must also be a refactoring, in order to preserve the meaning. Thus, this kind of dependent refactorings require an appropriate abstraction mechanism, since they must be specified for dependent models of different DSLs.

The first contribution, *Role-Based Generic Model Refactoring*, uses role models to abstract from refactorings instead of the target languages. Thus, participating structures in a refactoring can be specified generically by means of role models. As a consequence, arbitrary model-based DSLs are supported, since this approach does not make any assumptions regarding the target languages.

Our second contribution, *Role-Based Quality Smells*, is a conceptual framework and correlates deficiencies, their deteriorated qualities, and resolving refactorings. Roles are used to abstract from the causing structures of a deficiency, which then are subject to resolving refactorings.

The third contribution, *Role-Based Co-Refactoring*, employs the graph-logic isomorphism to detect dependencies between models. Dependent refactorings, which we call *co-refactorings*, are specified on the basis of roles for being independent from particular target DSLs.

All introduced concepts are implemented in our tool *Refactory*. An evaluation in different scenarios complements the thesis. It shows that role models emerged as very powerful regarding the reuse of generic refactorings in arbitrary languages. Role models are suited as an interface for certain structures which are to be refactored, scanned for deficiencies, or co-refactored. All of the presented approaches benefit from it.

Acknowledgment

Starting this thesis and bringing it to an end has been a long way. A lot of people accompanied me in the one way or another. I want to express my deep gratitude to the following people. Without them, I could have never reached the goal.

First of all, I want to thank my supervisor Uwe Aßmann. After an oral exam when I was an undergraduate, he asked me to come to his group. This was the beginning of the whole process, the beginning of many interesting and demanding discussions. He pushed me into many directions and his feedback and encouragement was so valuable.

Second of all, I want to thank my parents for their great support. Especially my mother did an invaluable job for reading my thesis. Furthermore, she was the first person who gave me the chance to use a computer when I was eight years old. She showed me the game Sokoban and I was so impressed. Since then, I had no doubt that "I will do something with computers". Thank you so much.

I had a great time at the chair of software technology. I got to know many excellent researchers and friends. At first, I want to thank the DevBoost guys Mirko Seifert, Christian Wende, Florian Heidenreich, and Jendrik Johannes. I appreciate your professionalism but also your relaxed nature. The time at conferences we had, or programming in my kitchen, was always a pleasure and I pretty much enjoyed the beers with you. Mirko, that we have won the best paper award at the MoDELS 2010 was so huge, I would have never had expected such a big thing before. In addition, it was my first paper. Thank you for being part of this experience. It's so great to be at your team now.

I want to thank other guys from the software technology chair: Christoph Seidl, for his nice support, discussions and the good time at the last two trips to Wien and Karlsruhe; Birgit Demuth, for her great backup at the chair and the eTe project; Katrin Heber, for all the help in the administrative stuff. Furthermore, I want to thank the following persons for their fruitful discussions and valuable feedback: Julia Schroeter, Claas Wilke, Birgit Grammel, Sven Karol, Konrad Voigt, Thomas Kühn, Sebastian Götz, and Christian Piechnick.

In the last year of my PhD trip, I got to know the group of Ralf Reussner, my external reviewer. We had a great workshop there. I want to thank Ralf and Benjamin Klatt for their openness and their comments on my work, and not to forget Erik Burger, for giving me support in some LaTeX fonts and Ti*k*Z issues.

I want to thank Helge Pfeiffer and Andrzej Wąsowski, it was a great experience to write the Lässig-paper with you. Furthermore, my gratitude goes to Frank Furrer, it was very pleasant to work with you in the seminars which resulted in my last publication, since we had so interesting topics and good students. Stefan Pietschmann and Vincent Tietz were my supervisors in the minor thesis, and they brought me closer to research and scientific writing for the first time. Gratitude to Jens Dietrich, for our discussions about graph querying and your support regarding

GUERY, and Tricia Balfe from Nomos Software, who gave us access to their OCL constraints and her support.

Special thanks go to Cloudy. You supported me enormously during my PhD journey. I appreciate a lot what you have done! Furthermore, I want to thank Schulle for reading parts of my thesis and for just being around when I needed it. Jochen and Jakob, it is great to have you as brothers, you sometimes made me not to forget the crazy things in life.

I also want to thank my students who have contributed a lot to my thesis: Erik Tittel, Michael Muck, Christian Vonsien, Fabian Hänsel, Martin Brylski, Anja Kirste, Kristin Blawatt, and the students of my last Komplexpraktikum regarding the Quality Smells.

The first BASIC programme I got to know was from my uncle Bruno. Thank you for this, although I cannot remember what it was about, but at that time I did not understand it anyway. Finally, thanks go out to my flat mates, Dave and Jonathan, and to my former computer science teacher from school, Bernd Burmeister, who taught me programming with Pascal and aroused this passion.

Gratitude to all of you and to those I forgot to name explicitly.

Publications

This thesis is partially based on the following peer-reviewed publications:

- 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.

- 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.

- 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.

- 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.

- J. Reimann, C. Wilke, B. Demuth, M. Muck and U. Aßmann, "Tool supported OCL refactoring catalogue", in *Proceedings of the 12th Workshop on OCL and Textual Modelling*, ser. OCL '12, Innsbruck, Austria: ACM, 2012, pp. 7–12, ISBN: 978-1-4503-1799-3. DOI: 10.1145/2428516.2428518.

The following peer-reviewed publications cover work that is closely related to the content of the thesis, but not contained herein:

- 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.

- U. Aßmann, A. Bartho, C. Bürger, S. Cech, B. Demuth, F. Heidenreich, J. Johannes, S. Karol, J. Polowinski, J. Reimann, J. Schroeter, M. Seifert, M. Thiele, C. Wende and C. Wilke, "DropsBox: the Dresden Open Software Toolbox", English, *Software & Systems Modeling*, vol. 13, no. 1, pp. 133–169, 2014, ISSN: 1619-1366. DOI: 10.1007/s10270-012-0284-6.

- F. Heidenreich, J. Johannes, J. Reimann, M. Seifert, C. Wende, C. Werner, C. Wilke and U. Aßmann, "Model-driven Modernisation of Java Programs with JaMoPP", in *Joint Proceedings of the First International Workshop on Model-Driven Software Migration (MDSM 2011) and the Fifth International Workshop on System Quality and Maintainability (SQM 2011)*, A. Fuhr, W. Hasselbring, V. Riediger, M. Bruntink and K. Kontogiannis, Eds., Oldenburg, Germany: CEUR Workshop Proceedings, Mar. 2011, pp. 8–11.

- C. Wilke, S. Götz, J. Reimann and U. Aßmann, "Vision Paper: Towards Model-Based Energy Testing", in *Model Driven Engineering Languages and Systems - 14th International Conference, MoDELS 2011, Wellington, New Zealand*, J. Whittle, T. Clark and T. Kühne, Eds., ser. Lecture Notes in Computer Science, vol. 6981, Springer Berlin Heidelberg, 2011, pp. 480–489, ISBN: 978-3-642-24484-1. DOI: 10.1007/978-3-642-24485-8_35.

- S. Götz, M. Leuthäuser, J. Reimann, J. Schroeter, C. Wende, C. Wilke and U. Aßmann, "A Role-Based Language for Collaborative Robot Applications", in *Leveraging Applications of Formal Methods, Verification, and Validation*, ser. Communications in Computer and Information Science, R. Hähnle, J. Knoop, T. Margaria, D. Schreiner and B. Steffen, Eds., Springer Berlin Heidelberg, 2012, pp. 1–15, ISBN: 978-3-642-34780-1.

- S. Pietschmann, V. Tietz, J. Reimann, C. Liebing, M. Pohle and K. Meißner, "A Metamodel for Context-Aware Component-Based Mashup Applications", in *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services (iiWAS 2010)*, ACM, Nov. 2010, ISBN: 978-1-4503-0421-4.

- S. Götz, M. Leuthäuser, C. Piechnick, J. Reimann, S. Richly, J. Schroeter, C. Wilke and U. Aßmann, "Entwicklung Cyber-Physikalischer Systeme am Beispiel des NAO-Roboters", in *Chemnitzer Linux-Tage 2012 – Tagungsband –*, Team der Chemnitzer Linux-Tage, Universitätsverlag Chemnitz, Mar. 2012, pp. 45–52.

The following publication covers work that is slightly related to the content of the thesis, but not contained herein:

- F. Furrer and J. Reimann (Eds.), "Impact and Challenges of Software in 2025", Technische Universität Dresden, Software Technology Group, Tech. Rep., 2014. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-152785.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Language has been the means of humans to express themselves, at all times. Let it be the spoken or written word, drawings or body language — what all kinds of communication have in common is that there must be a mutual understanding of the meaning of the language's concepts, if the communicator wants to ensure that a message is interpreted equally. That is the reason why *syntax* and *semantics* are essential in most languages. In contrast, *painting*, as the choice of communication, is not meant to be precise and concise. Thus, painters enjoy the freedom of art and create space for interpretation for the recipients of their messages. Different people might probably reflect divergently about art.

Consider another example: raising children. When parents want to pass on their values, they have several alternatives (not being mutual exclusive). They can, for example, tell their children what to do and what not to do. Or they may hope that children learn things by rewarding or punishment. Another educationally more valuable methodology can be to lead children to literature and provide stories of characters expressing the parents' desired values. This kind of indirect teaching of children with literature is promising when strong child characters are used, as shown for example in [Mah81]. But again, these examples illustrate that people may communicate differently about the same things.

When we look back further in history and glance at the Old Testament, we can eventually say that this has not always been like this. Consider, for example, *The Tower of Babel*[1] (see Fig. 1.2) which is a biblical story telling that mankind tried to equal God by building a huge tower in Babel with its top in the sky. At this time, the story says, only one common language was spoken all over the world. But, the attempt to build the tower was interpreted by God as hubris and arrogance. So he decided to punish mankind and initiated the *confusion of tongues*[2] by eliminating the common language and creating a plethora of different ones. As a result, the

[1] cf. Book of Genesis Chapter 11 (`http://www.vatican.va/archive/bible/genesis/documents/bible_genesis_en.html#Chapter%2011` visited 10th February 2015)

[2] `http://en.wikipedia.org/wiki/Confusion_of_tongues` (visited 10th February 2015)

Figure 1.1.: Examples of cave art.[3]

humans were confronted with insuperable communication difficulties and, hence, could not continue building the tower. Furthermore, God scattered them all over the world.

By contrast to the idea of punishment through God, the story of the Tower of Babel and the confusion of tongues can be interpreted differently. It can be seen not as a curse but a blessing, since the diversity of languages corresponds with the variety of different cultural groups, which left their indelible mark of human yearning to travel, get to know other countries, to communicate and to cooperate. Without such a great diversity mankind as a whole would be rather monotone.

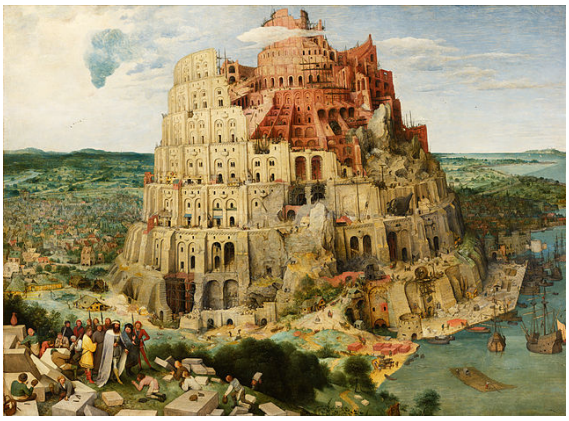


Figure 1.2.: *The Tower of Babel* by Pieter Bruegel the Elder, 1563.[4]

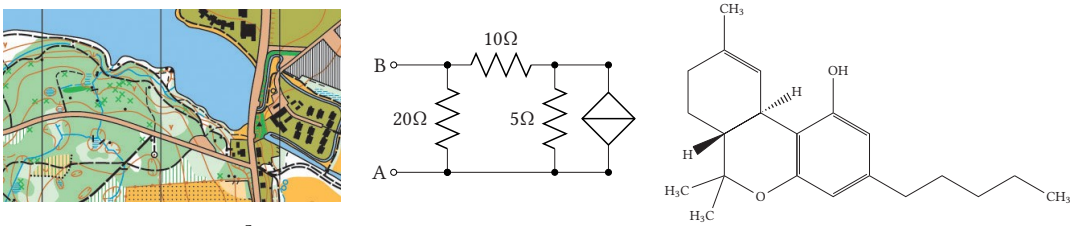
This diversity arrived in life a long time ago. Nowadays, as a tool of communication, language has many shapes. For instance, humans managed to form different kinds of communication for same domains. The most general kind is the spoken or written word, while quite often more dedicated languages are better to communicate about matters or problems, so that humans choose a suitable language for communicating about problems and solving them.

A popular example is the language of cave painting. Some of the masterpieces are supposed to be older than 40,000 years [Amo12]. The point is that the examples from Fig. 1.1 show that *abstraction* is a powerful tool for humans to express themselves. Animals or hunting scenes were painted without details, environmental surroundings were neglected. The information *where* something happened was less important to the painter in contrast to *what* happened. To this end, facts of reality were omitted, while others were emphasized.

Such 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)*. Some prominent examples can be seen in figures 1.3 (a)–(c).

Similar progress has happened in the discipline of computer science. In the beginning, instruc-

[3] `http://en.wikipedia.org/wiki/Cave_painting` (visited 10th February 2015)
[4] `http://en.wikipedia.org/wiki/Tower_of_Babel` (visited 10th February 2015)



(a) Cartography[5] (b) Electronics (c) Chemistry

Figure 1.3.: Examples of abstraction in scientific or engineering DSLs.

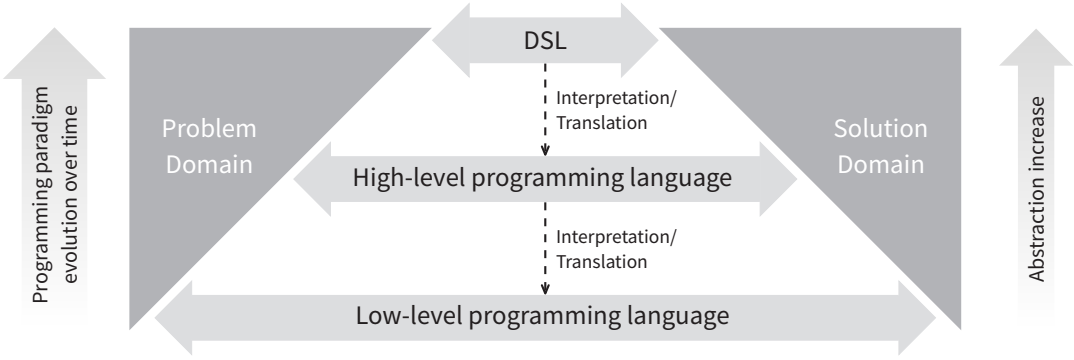


Figure 1.4.: Abstraction Gap based on [KT08, p. 16].

tion 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 1.4 illustrates this contemplation in the lower part. At that time, 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 1.4: compilation to another language, or direct execution respectively [EvSV+13]. Examples for popular DSLs are, e.g., the Unified Modeling Language [OMG11a] (UML), the Business Process Model and Notation [OMG13b] (BPMN) or Cascading Style Sheets [W3C11] (CSS). Table 1.1 shows the

[5] `http://en.wikipedia.org/wiki/File:Orienteringskort_bygholm_2005_detail.jpg` (visited 10th February 2015)

Table 1.1.: Example DSLs and their semantic contexts.

DSL	Application Context	Realisation
UML	development environment	code generation (compilation)
BPMN	running application	translation into execution language and executed in workflow engine (transformation and interpretation)
CSS	web browser	layout engine (interpretation)

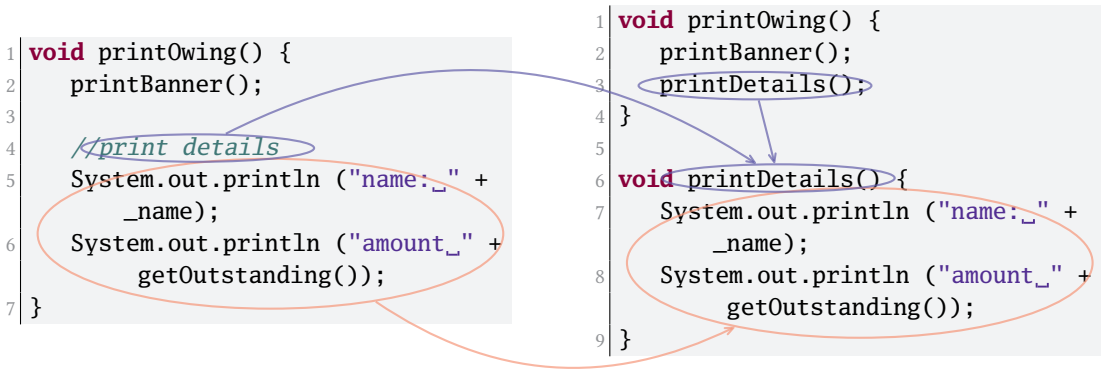
context in which these languages are applied and how they are realised. The table gives an idea about DSLs being deployed in a variety of different contexts, domains, or platforms.

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. 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. A deeper insight into MDSD is given in Section 2.2. At this point, it is important to know that 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.

There are other technical spaces suitable for developing DSLs (such as, e.g., grammars), but in this work only model-based DSLs are considered, since they allow for abstraction of languages to their constructional concepts. Their instances conform to these concepts. In the following, the strengths of such DSLs are illustrated and how the abstract concepts can be exploited to generate tools. Apart from that, the problems which are to be solved in this thesis are analysed and emphasized.

1.1. Language-Tool Generation Without Consideration Of Time And Space

As mentioned before, the increase of abstraction in programming languages fostered developers' productivity [KT08]. At the same time, 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



(a) before

(b) after

Figure 1.5.: Example of *Extract Method* refactoring in Java, based on [FBB+99].

and for the early identification of problems instead of living with them afterwards [Pfl98]. Furthermore, Lehman, Davis and Bersoff realised early that software changes: "If we do not learn to manage change, we will become its victims, not its beneficiaries" [DB91]. 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.

Let us consider the example in Fig. 1.5. In (a) a method can be seen printing a banner and some details afterwards. The comment in Line 4 suggests a different purpose of the following statements compared to the banner printing in the first statement. Thus, the last two statements can be extracted into a new method in (b) (equally named as the comment suggests) which is then invoked at the original position in the old method. As can be understood easily, the semantics of the method `printOwing()` didn't change, but from a design view `printDetails()` now can be reused, and concerns are separated better. Opdyke called this refactoring *Convert a Code Segment to a Function* [Opd92, p. 34], but today it is better known as *Extract Method* [FBB+99].

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. Representatives of LWs are MetaCase [KT08], the Eclipse Modeling Framework [SBPM08] (EMF), EMFText [HJK+09], Xtext [EV06], Spoofax [KV10], MPS [Cam14], GME [LMB+01] and many others. Some of them follow a generative approach, while others apply interpretation of a DSL's specification. What they all have in common is that they somehow produce a tool environment dedicated to the developed DSL. To distinguish between a LW

```
1  CONFERENCE
2     "International FooBar Camp"
3     ("Peter Meyers")
4
5  TRACK "Interesting Stuff" :
6     AT 09:00 : TALK "The future of Foo"
           PRESENTED BY "Peter Meyers"
7     AT 10:00 : TALK "Foo vs. FOO" PRESENTED
           BY "Andrew Bloomfield"
8     AT 11:00 : TALK "Onyl Bar-S is true"
           PRESENTED BY "Homer Simpson"
9
10 REGISTERED SPEAKERS :
11    "Peter Meyers" FROM Germany,
12    "Andrew Bloomfield" FROM USA,
13    "Homer Simpson" FROM USA
```

(a) before

```
1  CONFERENCE
2     "International FooBar Camp"
3     ("Peter Meyers")
4
5  TRACK "Interesting Stuff" :
6     AT 09:00 : TALK "The future of Foo"
           PRESENTED BY "Peter Meyers"
7
8  TRACK "Less Interesting Stuff" :
9     AT 10:00 : TALK "Foo vs. FOO" PRESENTED
           BY "Andrew Bloomfield"
10    AT 11:00 : TALK "Onyl Bar-S is true"
           PRESENTED BY "Homer Simpson"
11
12 REGISTERED SPEAKERS :
13    "Peter Meyers" FROM Germany,
14    "Andrew Bloomfield" FROM USA,
15    "Homer Simpson" FROM USA
```

(b) after

Figure 1.6.: Example of *Extract Track* refactoring in the Conference DSL.

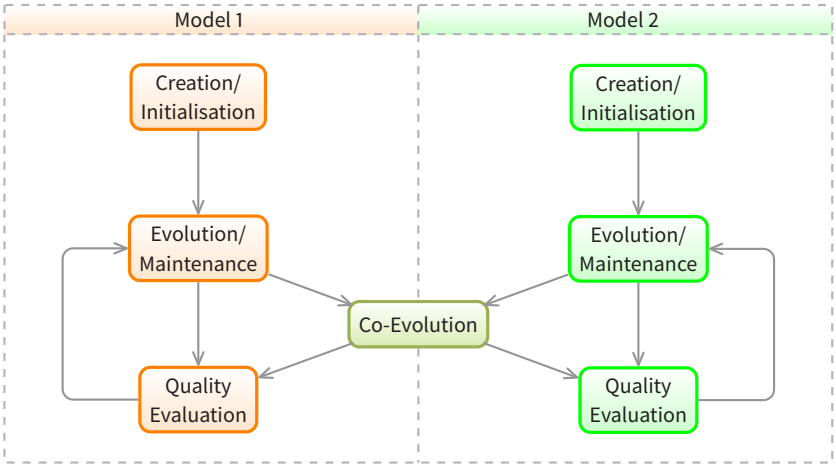
and a resulting tool environment for a developed DSL, we define a new term for the produced result. A *Domain-Specific Language Environment (DSLE)* is a tool, which is derived by a LW to interact (work/edit/debug/or many other actions) with a particular DSL. According to [EvSV+13] a resulting DSLE usually consists of an editor, syntax highlighting, a parser, language-specific refactoring, and semantic services as reference resolution or error marking. It is not relevant, if the resulting DSLE is integrated into the LW used for developing it, or if a separate tool is generated. 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.

Consider, e.g., our model-based DSL for planning conferences.[6] We used EMFText to generate a DSLE for this DSL. This is a little language and can be used to define different tracks, talks and speakers for conferences. For the talks only declared speakers can be referred. Having the *Extract Method* refactoring from Fig. 1.5 in mind, Fig. 1.6 illustrates a very similar *Extract Track* refactoring for the conference DSL. The last two talks in (a) (Lines 7 and 8) are less interesting and are therefore moved to a newly created track. The result can be seen in (b). This example shows that known code refactorings from programming languages should be made available for DSLEs.

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

[6] `http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Conference` (visited 11th February 2015)

Figure 1.7.: Overview of model life cycle in MDSD.



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. Consider, e.g., *Extract Method* in Fig. 1.5 again. From an abstract point of view, some children (the statements) of a parent (original method) are moved to a new parent (new method). But one would never declare that every parent-child relation in the syntax tree should be restructured in the sense that the child gets a new parent. It does not even make sense in every case. Thus, from a pure structural viewpoint refactorings cannot be generated automatically. For that reason, refactorings cannot be derived from the abstract syntax (the DSL's metamodel). DSLEs are not able to provide DSL-specific mature refactoring tools and they omit the aspect of evolution over *time* 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 1.7 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, e.g., by a user or by a model transformation. 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. In general, it is not even possible to estimate which other DSLs are candidates for interaction at the design or instance level. The trivial case is when a DSL is referenced by another at the design level. Obviously, models of such referring DSLs might relate to models of the referred DSL. In contrast, it is also possible that a reference of a DSL at design level is very vague or weak. This means that a referred DSL cannot be specified precisely and the connection to models is established at the instance level. Those connections then can point to instances of arbitrary DSLs and cannot be foreseen. In such a case, interdependencies between those models arise. This means they relate to each other and are dependent on each other. As a consequence, a dependent model is influenced by modifications of the other model. Figure 1.7 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*. A popular example for the described scenario is the process of aging between requirements model, design model and code. Most often, changes in one of these artefacts are not propagated to the others and thus they age. 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 its surrounding *space*.

By means of substantiating the aforementioned scenario of multiple DSLs being integrated in one IDE Pfeiffer and Wąsowski introduced a more specific term in [Pfe13; PW15]: *Multi-Language Development Environment (MLDE)*. Regarding Pfeiffer a MLDE is an IDE providing cross-language support (CLS) mechanisms. Since MLDEs "[...] integrate editors and other language specific tools across language boundaries with each other" [Pfe13], multiple DSLEs in combination with additional CLS mechanisms form a particular MLDE. Consequently, IDEs are commonly accepted for programming. For the sake of not breaking usual habits of developers in *programming*, MLDEs should be used for *modelling*. For this reason the term Multi-Language Development Environment (MLDE) corresponds to our idea of environments for modelling and it is therefore used throughout this thesis.

Another consequence of DSL isolation is that refactorings cannot be reused across different DSLs. Have a look at the *Extract Method* refactoring in Fig. 1.5 on page 5 again. This is an example for the programming language Java. The same semantics-preserving restructuring is available for other programming languages as well. 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 at the design level. Therefore, the abstract syntax must be examined and feasible relations between concepts have to be found for establishing potential candidates for refactorings. As an example, let us regard the Java programming language as a

model-based DSL. The Java Model Parser and Printer [HJSW10] (JaMoPP)[7] enables us to consider Java source code as models. A more detailed explanation about why this is possible will be provided in Sect. 2.2. Here it is sufficient to know that Java code can be considered as models. Within the metamodel of JaMoPP, there is a parent-child relation between the concepts `Method` and `Statement`. This structure, e.g., is suitable for the *Extract Method* refactoring in Fig. 1.5, because statements are extracted from an original method to a new one. Another parent-child relation can be found between `Enumeration` and `EnumConstant` expressing the fact that an enumeration can contain several constants. In this case, it does not make sense to provide an *Extract Enumeration* refactoring because referencing another enumeration as a constant in the original enumeration is not possible due to the Java specification. Such structures have to be determined by the DSL designer and the challenge is to find the suitable ones.

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. The Java language, e.g., is most probably used by engineers whereas the conference DSL for describing conferences does not need a technical background, thus, its users might not have a programming background and could be overtaxed with mature refactorings at their hands.

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.

1.2. Challenges

The aforementioned problems find expression in the following goals and challenges which will be covered in this work.

Generic Specification of Refactorings As already mentioned, there is a plethora of DSLs which is still growing. In general, 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. As argued in the previous section, 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 essential.

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 goal 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].

[7] `http://www.jamopp.org/` (visited 10th February 2015)

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 As argued in Sect. 1.1, a DSL and its models can have interdependencies to instances 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 goal 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.

In summary, an approach is missing that provides mature refactoring means in MLDEs and considers time, space and quality. The *time* aspect takes the fact into account that models might evolve through refactoring over time. The *space* aspect takes the circumstance into account that a model is never alone. Several other dependent DSL instances of the same or different metamodel may be contained in the model's surrounding. Thus, the approach must comprise the co-refactoring of those dependencies as a consequence of an initially refactored model to preserve the consistency of all models. Furthermore, the approach must take the *quality* aspect that way into account that evidence can be given about which model structures influence what quality negatively, and which refactorings can resolve those deficiencies. As a consequence refactorings can be suggested to the DSL user.

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

To establish a connection between the problems from Section 1.1 and the challenges from Section 1.2, a simplified MDSD model life cycle from Figure 1.7 is underlain. The simplified life cycle starts with the phase of *Refactoring* in which a model evolves. In the general case, two subsequent phases follow. On the one hand, the phase of *Quality Evaluation* follows, in which quality properties of the evolved model are evaluated, and, when indicated, refactoring candidates are identified and possible refactorings for resolution are suggested. On the other hand, the *Co-Refactoring* phase follows, in which dependent models are detected and possible

dependent refactorings are determined. If applicable, subsequent co-refactorings are executed on dependent models and again, the *Quality Evaluation* is the next step. Now, the quality of the dependent models is evaluated as a consequence of an applied co-refactoring. In either case, after the quality was evaluated, the next phase is *Refactoring* again when the desired quality requirements are not satisfied.

To address all three of these phases and to support the entire life cycle of heterogeneous models in MLDEs, this thesis contributes a comprehensive approach for quality-aware refactoring and co-refactoring based on role modelling. It is structured as follows.

Foundations (Chapter 2)

Before the parts of the comprehensive approach of this thesis are elaborately presented foundations are accomplished in Chapter 2.

Review of Related Work (Chapter 3)

For being able to give evidence about what other related approaches exist to tackle the overall problem, a broad review of the state-of-the-art is provided as well. For this purpose, requirements are established to compare the approach of this thesis with the related work.

Generic Role-Based Model Refactoring (Chapter 4)

To provide means for language-independent [MTM07; TMM08; MMBJ09] specification of refactorings, an appropriate abstraction mechanism is essential. On the one hand, abstraction over all potential languages is one option. The limitation of this alternative is that the DSLs which refactorings should be provided for are constrained in the sense that all must be similar to the concepts in the language abstraction. Thus, this approach would be too static, as will be discussed in Sect. 3.1. For this reason, the approach of abstraction over the desired refactorings is chosen as another option. In detail, *role modelling* is applied as abstraction technique, since a *role model* represents a dedicated view of objects in an interesting 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.

> **Hypothesis 1:** Abstraction over refactorings instead of abstraction over languages is feasible to provide means for specifying refactorings generically.

The prove of this hypothesis is twofold. First, role-based generic model refactoring is conceptualized and implemented. Second, a set of generic refactorings is elaborated which is then applied in different DSLs.

Suggestion of Refactoring Specifications (Chapter 5)

When refactorings are specified generically, it must be possible to instantiate them for particular DSLs. The process of making a generic refactoring DSL-specific is not trivial, since there might

be many possibilities to do so. Some of them are suitable while others are not. To provide support for this task, Chapter 5 presents an approach making use of graph querying in order to provide suggestions of DSL-specific refactoring specification.

Correlating Bad Smells, Qualities and Refactorings (Chapter 6)

As already illustrated in the previous section, the term *bad smell* [FBB+99] is vague and imprecise. Furthermore, the explicit connection of structures violating particular quality requirements is omitted completely. This relation is considered only informally, just the same as the relation to resolving refactorings. As a consequence of the mentioned shortcomings, the term *bad smell* lacks precise understanding. We argue that a correlation between bad smells from Fowler *et al.*, qualities and resolving refactorings exists, which must be made explicit to automate tool support in suggesting particular refactorings.

> **Hypothesis 2:** To provide a precise understanding of *bad smells* [FBB+99] and their explicit relationship to qualities and resolving refactorings the new term *Quality Smell* and a conceptualization is required. It must incorporate known approaches for quality determination, such as metrics and anti-patterns.

To prove this hypothesis the new term *Quality Smell* is defined and a conceptual framework is specified. This framework serves for sharpening the understanding of what it means when model structures do not meet particular quality requirements and how this circumstance can be eliminated. Furthermore, the conceptualization is implemented and it is shown that the suggestion of potential refactorings to resolve particular quality smells now is possible.

Quality Smell Catalogue (Chapter 7)

As we have seen in history [BC87; GHJV94; Sai03] pattern catalogues and pattern languages support designers of object-oriented software "[…] to ask (and answer) the right questions at the right time" [BC87]. This means that they provide means to declare reusable patterns for recurring problems in object-oriented design. If such a problem occurs and is detected, the appropriate pattern can be applied to solve the problem. A more rigorous approach is presented in [TM15] arguing that *Pattern First Thinking* improves software quality, in terms of more flexible software, and team communication. The authors of [BC87; GHJV94; Sai03] presented their catalogues for design problems in object-oriented software, but the same holds for quality-related problems in MDSD models.

> **Hypothesis 3:** A quality smell catalogue can be mined to improve the understanding of the term *quality smell* and its connection to qualities and refactorings. Such a catalogue enhances documentation and contributes reusable patterns for quality improvement.

To prove this hypothesis a quality smell catalogue is compiled for the domain of mobile devices' applications. Exemplarily the Android[8] operating system as target platform is used because of its open-source character.

[8] `http://www.android.com/` (visited 10th February 2015)

Co-Refactoring 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 synchronized with the initially refactored model in terms of behaviour preservation. Thus, they must be co-refactored. This task comprises, first, the specification of modifications dependent on the preceding refactoring changes, second, the detection of dependent models, and, third, the application of the specified dependent modifications on the dependent models. The first subtask demands for distinction between two different situations. On the one hand, the DSL of the initially refactored model is known, and on the other hand, the DSL is not known. In the latter case, the dependent modifications are specified with respect to the generic specification of the initial refactoring, while concrete concepts of the known language can be used in the former case.

> **Hypothesis 4:** Consistency of models, being dependent on an initially refactored model, can be maintained through the specification and execution of modifications dependent on the preceding changes of the refactored model. Such co-refactorings must be applied on those models being recognized as dependencies of an initially refactored model.

To prove this hypothesis a concept and implementation for the detection of dependent models and the specification and execution of dependent modifications are developed. Afterwards it is applied to a case study in the domain of co-refactoring Web Ontology Language [W3C12] (OWL) models dependent on changes in metamodels which the OWL models are derived from.

Refactory: An Implementation (Chapter 9)

For validation of the concepts developed in this thesis we provide an implementation. It is realised in our tool called *Refactory*.[9]

Evaluation (Chapter 10), Conclusion and Outlook (Chapter 11)

The evaluation of the aforementioned contributions is presented in Chapter 10. Finally, this thesis is closed with a conclusion and discussion of future work in Chapter 11.

[9] `http://www.modelrefactoring.org/` (visited 10th February 2015)

2
Foundations

To ensure that the reader has the same understanding of concepts and terms used throughout this thesis, this chapter provides the foundations to form a common base. First of all, the concept *refactoring* is explained in general. Since we consider only model-based DSLs, the MDSD and its abstraction layers are presented afterwards. The concept of *roles* is an essential constituent in the whole thesis. Therefore the origin and the foundations we base our work on is illustrated then.

2.1. Refactoring

In 1990 Opdyke and Johnson defined the term *refactoring* for the first time in [OJ90]. Shortly thereafter, Opdyke published his dissertation "Refactoring Object-Oriented Frameworks" and the term *refactoring* has been established. He gives the following explanation:

> "*Refactorings* are reorganization plans that support change at an intermediate level. [...] Refactorings do not change the behaviour of a program; that is, if the program is called twice (before and after a refactoring) with the same set of inputs, the resulting set of output values will be the same." [Opd92]

The main motivation of Opdyke's work was to increase reuse in object-oriented frameworks because of the rising complexity. The problem is that a refactoring can be applied after a part of a software is already reused and, thus, it can have extensive impact to existing clients. Therefore tool support must be provided in order to manage the complexity and error-proneness. As a prerequisite for the behaviour preservation Opdyke identified the satisfaction of pre-conditions as essential part of a refactoring.

In his work, he studied object-oriented frameworks and proposed a catalogue of 29 refactorings, as, e.g., *Convert a Code Segment to a Function* [Opd92, p. 34], which moves a segment of code to a new function and calls it at the previous position. As already mentioned in Sect. 1.1, this refactoring is better known as *Extract Method* nowadays [FBB+99].

Some years later, the standard work which is cited most often with respect to practical refactoring has been published by Fowler *et al.* in [FBB+99]. The authors identified the importance of testing before and after the application of refactorings in order to prove the behaviour preservation on an informal base. Furthermore, they related the refactorings to *bad smells* indicating potential candidates to be restructured. An essential result of their work is the catalogue of refactorings presented therein. It is based on the work of Opdyke, but underlies a schema similar to the one presented in [GHJV94] ("Gang of Four") regarding design patterns in object-oriented software.

With the advent of the MDSD, it was recognized that known code refactorings should be made available for the emerging amount of DSLs [TDDN00; MTM07]. As a consequence the idea of *model refactoring* was born and Van Der Straeten, Jonckers and Mens, e.g., defined it as "[...] a transformation used to improve the structure of a model while preserving its behaviour" [VJM07]. This is the tenor of most other informal definitions of the term *model refactoring*. Common sense is that the meaning is very similar to the one of code refactoring, except that instead of code existing models are restructured in order to improve design quality while the behaviour is preserved.

In this thesis, a new technique for generic model refactoring is presented. To differentiate clearly between a generic refactoring, its instantiation and the execution the following terminology is used throughout the thesis: *refactoring* denotes a concrete restructuring in a model of a particular language, *generic refactoring* indicates a reusable abstraction of similar refactorings applicable in models of one or more languages, and *refactoring execution* means the application of a refactoring on a particular model.

2.2. Model-Driven Software Development

In order to understand the need for *models*, Kleppe argued that the rise of complexity in software development can be managed with the use of frameworks and reusable patterns [Kle09]. But this is only a short-term solution which can only be mastered by a rise of abstraction to understand what software does in the core. Due to this background software development has been subject to change by means of separating concerns (separation of concerns (SoC)) in abstract representations of particular aspects of a software. Those abstractions are considered to be a *model*. Rothenberg defined a *model* as follows:

> "A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality" [Rot89].

This definition reflects the fact that a model focusses on specific concerns of reality and omits certain properties. In software development, this technique of abstraction has been taken over in the paradigm of *Model-Driven Software Development (MDSD)* [GS03; SVB+06]. In MDSD, models are used to abstract from typical technical aspects of the system under development. Those aspects are, e.g., the target platform the software is intended to be run on, or the programming language used for implementation. Furthermore, different models are used to describe solutions of different problems of the overall system. The UML is a prominent example for it, since it

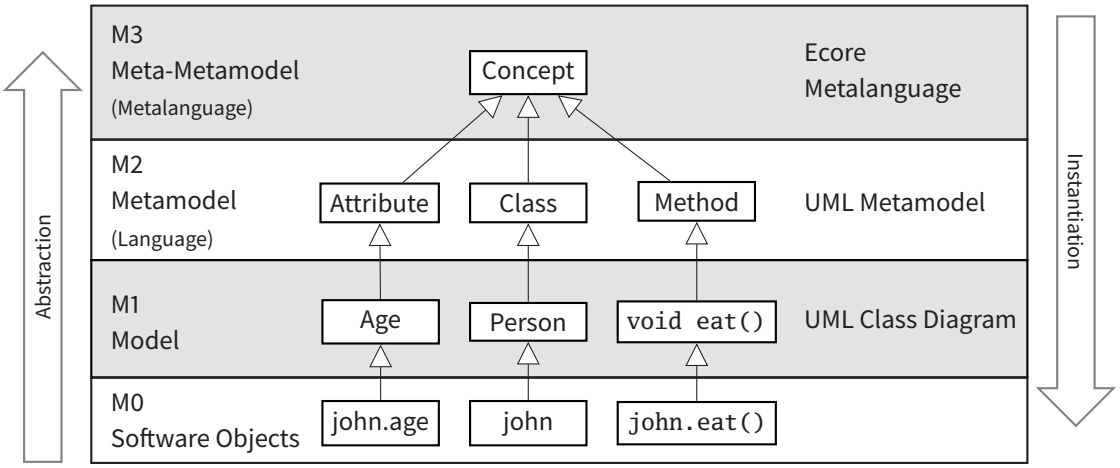


Figure 2.1.: MOF architecture.

provides various types of diagrams to specify distinct parts of a software [OMG11a]. Examples are *class diagrams* to define concepts of the system domain or *state machine diagrams* to specify a protocol a software component must satisfy.

To make use of defined models regarding the concrete realisation of a system, they are then translated into representations being meaningful for underlying executors. This conforms to the explanation of the abstraction gap in Fig. 1.4. The translation into other representations is explained in more detail in Sect. 2.2.2. But at first the question is answered how different kinds of models can be distinguished. In this sense this reflects the need for a formalisation in order to let models be processed by aforementioned translators. Thus a formal description of models is needed which provides further information about the structure of models. This is explained in the following section.

2.2.1. Levels of Abstraction and Metamodelling

As already mentioned in Chap. 1, we only consider model-based DSLs in this thesis. Consequently, it must be clarified what this means exactly and how to distinguish different DSLs in this regard.

To let models be processed by a machine, the structure of a model must be known. The reason is that the set of models is potentially infinite and thus a finite specification of the commonalities of models is needed. Therefore a formalism is required to model the structural constraints of models. The process of providing the definition of model structures by means of a model is called *metamodelling*. Thus a metamodel describes properties of models. It can be observed that we already have different abstraction layers: one for a metamodel and another for the models described by the metamodel.

The Object Management Group (OMG) provided a formal base for metamodelling in order to support the creation of concrete metamodels. This formalism is defined as *Meta Object Facility [OMG13c] (MOF)* and provides an architecture where constituents of one layer describe constituents in an underlying layer. This architecture is illustrated in Fig. 2.1.

It consists of four metalayers to express the different levels of abstraction. On top, we have the M3 layer which represents the metalanguage layer to describe languages (or metamodels)

in the underlying M2 layer. Thus, the metalanguage (or the meta-metamodel) provides means to specify the structural properties and concepts of languages. These structural properties are considered to be the abstract syntax of a DSL restricting the set of valid models regarding the DSL's required concepts. The DSL itself resides at the M2 layer of the architecture. Since the structure of valid models now is defined the DSL can be used for instantiating those structures. Instantiations of a metamodel are the models residing at the M1 metalayer. They represent the real world or software objects at the M0 layer. As can be observed there is a *conformance* relationship between all metalayers: M0 objects conform to their abstract representations on the M1 layer (the models), which in turn conform to the structural specification of their metamodels at the M2 layer (the DSLs), which itself conform to the meta-metalanguage resided at M3.

According to Fig. 2.1, an example would be (depicted at the right part) the Ecore metalanguage provided by the Eclipse Modeling Framework [SBPM08] (EMF). I It represents the quasi-standard for modelling metamodels of DSLs. Ecore is used to define the metamodel of the UML at M2 which in turn is used to define models, as, e.g., class diagrams, at M1.

Ecore itself implements the Essential MOF (EMOF) standard and therefore ensures that models can be considered as typed, attributed graphs [Roz97]. A model is composed by its containing elements which constructs a tree structure. But in contrast to trees, models can have references to other model elements and thus form a graph. This property reveals the advantage that no additional resolution mechanism is needed to navigate from one model element to another model element along a reference.

To close the gap to model-based DSLs, one can say that a DSL is defined by its *abstract* and *concrete syntax*. The abstract syntax corresponds to the metamodel—the valid structure of the DSL models. The concrete syntax defines rules about which technique is used to create concrete instances of a DSL. This is the aspect of representing models. Common techniques are, e.g., graphical syntaxes (as used by the UML) or textual syntaxes. For one and the same abstract syntax various concrete syntaxes can be defined. Thus it is possible to provide textual and graphical editors for the same DSL, as it is for example possible with EMFText or the Graphical Modeling Framework [Gro09] (GMF) respectively.

To consider programmes of a certain programming language as models, a metamodel and a concrete textual syntax, which reflects the language's syntax specification, are required. In this sense, programmes do not span an abstract syntax tree anymore but an abstract syntax graph. As stated above, resolution semantics are now explicit by means of references. Providing a model-based DSL for an existing programming language is no trivial task, but it is possible as demonstrated by the Java Model Parser and Printer [HJSW10] (JaMoPP). Therefore, especially Java programmes can be handled as models.

2.2.2. Model Transformations

Since we now have a specification of the structure of models by means of their metamodels, several tasks can be performed depending on these structures. In usual MDSD scenarios, models are translated into other *artefacts* in order to carry out additional tasks such as, for instance, refining a model in a representation being more expressive. As an artefact, we consider any kind of data that conforms to a certain schema. Thus, artefacts can be represented as models. A prominent example for such a refinement process is the Model-Driven Architecture [OMG03] (MDA) in which models of higher abstraction are transformed into more specific models, in order

to add platform-specific information stepwisely.

Thus, a model transformation translates a source model into a target model by executing the specification of the transformation. Such a specification can be defined by means of the metamodels of source and target. This means that the concepts (metaclasses) defined in a metamodel are the basis of the transformation specification. Thus a transformation maps elements from the input to elements in the output model. The particular languages of the input and output models are not restricted. Therefore, different kinds of transformations exists. On the one hand, the metamodels can be distinct. In that case the model transformation is called *exogenous* [MV06]. On the other hand, the metamodels of source and target can be the same. In this case we speak about *endogenous* model transformations [MV06]. Endogenous model transformations can be distinguished further by means of the amount of involved models. If the source and target models are the same such a transformation is considered as an *in-place* transformation. Executing a refactoring is an example for that. In contrast, an *out-place* transformation incorporates several models regardless of the fact if it is an endogenous or exogenous transformation.

While transforming one model into another model, correspondences between the transformed elements are established. These correspondences provide valuable information about which model element is the cause of the realisation of another model element. Such correspondences are called *traces* or *trace links* in order to explicitly describe the trace from the source to the target element. We make use of this concept in Sect. 8.3.

2.3. Role-Based Modelling

Software systems are subject to change, they evolve over time and the complexity rises [Leh96; Kle09]. To manage complexity, one can take advantage of a strategy humans apply in reality. Consider, e.g., a woman having a husband and some children. She plays table tennis in a club and is also a researcher at a university. In case one of her children gets sick she cares solicitously and takes the child to the medic. In another situation her table tennis team has a competition and she prepares for it mentally and physically in order to access her potential power. Then a research paper was accepted and she arranges everything for the business trip. All these tasks have to be carried out by her. In summary, one can doubt how the woman manages to organise everything. But what humans do is to separate different contexts of life. The first example shows the woman as a *mother*, in the second she is a *sportswoman* and in the third she is a *researcher*. In this sense she *plays roles* in different *contexts* of her life. Focussing one context while omitting others allows for a dedicated solution only concerning this context.

This strategy of *role playing* has been transferred to software development, as well, since it allows for the dedicated concentration on different contexts of a software. As seen in the little example, roles provide means to capture context-dependent behaviour of subjects and collaborations between them [Bac73]. In our example, the subjects are humans; and in software development subjects are objects.

A plenty of role modelling approaches exist of which each has its justification [KLG+14]. In [Ste00], Steimann provided a list of role features being essential in utilising roles for software development. This was a first step in formalising roles, which was took up and extended in [KLG+14]. The role concept we rely on in this thesis was first published in [RWL96] and later extended in [RG98]. Therein, Riehle and Gross provide an informal metamodel of roles and

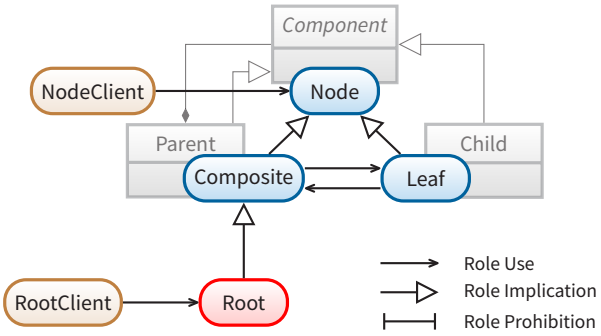


Figure 2.2.: Role model for *Composite* pattern [GHJV94] according to [RG98].

argue that role models describe patterns of collaborations between objects that constrain the intrinsic objects [RG98]. This approach is presented shortly in the following.

Riehle and Gross define the view of an object to another one as a *role type*. If an object conforms to a role type specification, it plays the *role* specified by the role type. In the following and throughout this thesis, we use the terms *role* and *role type* interchangeably since it is sufficient for this work to know that a role represents certain properties in a particular context. When an object plays that role the fact that it conforms to the role type is given implicitly. We argue that this view concerning roles and role types is sensible.

In addition to roles, Riehle and Gross allow for the specification of *collaborations* between roles. With the concepts of roles and collaborations it is now possible to specify *role models* representing the interaction of objects playing those roles in a particular context. The following collaborations are distinguished.

Role Use is directed from a role `A` to `B` and denotes that an object playing `A` must be associated to an object playing `B`.

Role Implication is directed from a role `A` to `B` and signifies that the object playing role `A` must also be able to play role `B`.

Role Prohibition is a non-directional collaboration between roles `A` and `B` and expresses that an object playing `A` must not play role `B` at the same time.

Riehle and Gross argue that a role model can be mapped to class diagrams of frameworks. A role model then is considered as structural constraints over the class diagram which can be checked statically.

Have a look at Fig. 2.2, showing a small class hierarchy in the background. It reflects the *Composite* pattern of the Gang of Four [GHJV94] (GoF). In the foreground, one can see the role model according to [RG98] representing this GoF design pattern. Thus, a class `Parent` plays the role `Composite` and knows (*Role Use*) its `Children` playing the `Leaf` role. The other way around, the `Child` knows (*Role Use*) its `Parent`. Furthermore, we have the role `Root`, which is not contained in the GoF pattern and is located in another role model (denoted by a different colour). An object playing the `Root` role must also play the `Composite` role (*Role Implication*). The roles `RootClient` and `NodeClient` provide access to either the root node of the composite or an arbitrary node.

This approach of capturing certain aspects of software systems in role models is quite promising, since it allows for distinguishing the different contexts a system can interact with. Furthermore, a role model can be specified independently from the particular model it should be mapped to. Thus, role modelling is a sophisticated abstraction mechanism to focus on the interaction between certain participants within a context by using roles and collaborations in between.

3
Related Work

In this chapter, related work including the state-of-the-art is discussed. As a preliminary consideration, we have to admit that, to the best of the author's knowledge, no comprehensive approach exists covering all three main aspects we emphasized in Section 1.1: model refactoring, quality evaluation, co-refactoring. These aspects refer to Hypotheses 1, 2 and 4. As stated earlier, quality evaluation means to detect structural deficiencies related to particular qualities. Therefore, we use the following four main categories to analyse and classify the related work and other approaches: Model Refactoring, Quality, Structural Deficiencies, Co-Refactoring. There might be approaches overlapping some of these categories, thus, the total number of possible categories is the sum of all combinations without repetition received by selecting one, two, three or four of these classes respectively: $\sum_{k=1}^{4} \binom{4}{k} = 15$. From a set-theoretic point of view we get the same result by calculation of the cardinality of the power set of these four categories. If X is the set containing the four main categories then we get the following cardinality: $|\mathcal{P}(X)| - 1 = 2^{|X|} - 1 = 2^4 - 1 = 15$ where 1 is subtracted since the empty set is not relevant for our consideration of related work. Thus, the classification scheme in Fig. 3.1 contains 15 categories being described in Table 3.1. These categories are not meant to be considered in the algebraic sense, but they are illustrated within a Venn diagram in order to recognize better how the categories relate to each other.

Obviously, category R-CoR-SD-Q in the middle of this scheme is the most interesting one. It covers all the aspects analysed previously. As will be seen, none of the presented other approaches and publications fall into this category. The objective of this work is to cover this category in the end. As a consequence of the absence of other comprehensive approaches, the related work is divided into three parts, mapped to the core chapters of this thesis: Model Refactoring (Chapter 4), Quality Evaluation (Chapter 6) and Co-Refactoring (Chapter 8). The related work discussed then is brought into context of the classification scheme and is evaluated according to the following three-valued scale: + is used if a requirement is met completely, ∘ is used if a requirement is met only partially or is not realised satisfactory, and – is used if a requirement is not realised at all.

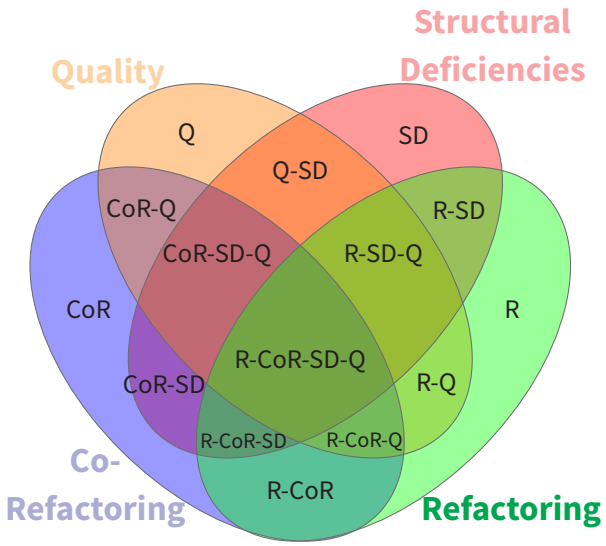


Figure 3.1.: Classification scheme for the related work.

Table 3.1.: Categories of the classification scheme.

Characterisation	Category Identifier	Description
1 element	R	Refactoring
	CoR	Co-Refactoring
	Q	Quality
	SD	Structural Deficiencies
2 elements	R-SD	Refactoring and Structural Deficiencies
	R-Q	Refactoring and Quality
	R-CoR	Refactoring and Co-Refactoring
	CoR-SD	Co-Refactoring and Structural Deficiencies
	CoR-Q	Co-Refactoring and Quality
	Q-SD	Quality and Structural Deficiencies
3 elements	R-SD-Q	Refactoring, Structural Deficiencies and Quality
	R-CoR-Q	Refactoring, Co-Refactoring and Quality
	R-CoR-SD	Refactoring, Co-Refactoring and Structural Deficiencies
	CoR-SD-Q	Co-Refactoring, Structural Deficiencies and Quality
4 elements	R-CoR-SD-Q	Refactoring, Co-Refactoring, Structural Deficiencies and Quality

3.1. Model Refactoring

As already discussed and manifested in Hypothesis 1, a suitable abstraction mechanism in the form of abstraction over the refactorings instead of abstraction over the target DSLs is needed for realising a generic model refactoring approach. In the literature, only few generic approaches can be found. All of them have their limitations, thus we take it for granted to review non-generic approaches, as well, in order to get deeper insights and learn something from them. For distinguishing the related work, we classify the existing approaches according to the MOF metalayer the refactorings are specified in. This is helpful since all approaches belonging to the same metalayer suffer from the same disadvantages or limitations and similar observations can be made.

In the following subsections, we first specify criteria enabling comparison of the different refactoring approaches. Afterwards, the related work is reviewed in detail before they are evaluated.

3.1.1. Requirements

To be able to compare the various existing approaches, comparison criteria are needed. Initial requirements regarding a refactoring tool have already been proposed in [Opd92] and [FBB+99]. Later they were extended and adapted to the MDSD context, e.g., in [MV06] and [MTM07], and tailored to the area of generic model refactoring in [Rei10]. The following requirements are based on the mentioned publications. Some are conceptual requirements while others can be realised in the implementation.

1. Genericity: An appropriate abstraction is needed that allows for metamodel-independent specification of a particular refactoring. Such an abstraction must be target of the generic transformation specification which then must be applicable in a DSL-specific context. This requirement ensures that refactorings can be reused across different DSLs, without specifying them anew.

2. Flexibility: In addition to the first requirement, the abstraction must be flexible enough not to impose restrictions with regard to the concepts of an intended target language. Thus, a generic refactoring approach must be able to support any structures of a target language.

3. Specificity: When a refactoring is specified in a generic manner, it must still be applicable in a language-specific precise context. Thus, we demand means to be provided enabling the DSL-specific execution of refactorings on the one hand, and language-specific adjustment on the other hand.

4. Behaviour Preservation: According to the definition of the term *refactoring* an approach should preserve the semantics of the refactored model.

5. Pre- and Post-conditions: For being able to describe the state of a model to qualify for a particular refactoring, it must be possible to specify pre-conditions. The circumstances under which a refactoring is valid under must be specifiable in terms of post-conditions.

6. Atomicity: A model refactoring must be executable as a whole unit. Otherwise a transformed model might be in an invalid state. Thus, a refactoring must be applied either completely or not at all.

7. Reversibility: This requirement has a direct connection to the previous one. An approach should provide means to roll back a refactoring completely. A refactoring user should have support to revise and correct a decision in case a particular refactoring was applied mistakenly.

8. Specification Suggestion: When a refactoring is generic (Requirement 1) and there is support to apply it in instances of a concrete DSL (Requirement 3), it would be reasonable to support a DSL designer in the decision which concrete refactorings to provide for her DSL users. In this sense, means are needed to help the designer in the process of making a generic refactoring DSL-specific.

9. Application Suggestion: Another suggestion is related to the DSL user. She must be able to get recommendations about which refactoring to apply in a certain context. This requirement is highly related to Requirement 3 in Sect. 3.2. But since refactoring approaches are not required to cover quality deficiency approaches (and vice versa) we separate these requirements.

10. Interoperability: This requirement is twofold. On the one hand, it means that no restrictions regarding a DSL's metamodel should be stated so that a maximum variety can be supported. On the other hand, this requirement comprises to execute refactorings independently from the model's representation (e.g. tree-based, textual, graphical). Thus, the only restriction should be that supported languages must be MOF-conform.

3.1.2. Literature Review

Depending on the MOF metalayer which refactorings are specified at, different observations can be made. In the following, related work resided at these layers is analysed.

M3

In [MMBJ09; SMM+12] an approach to specify generic refactorings is presented. Here, the authors introduce a meta-metamodel called *GenericMT*, which enables the definition of generic refactorings on the MOF layer M3. This meta-metamodel contains structural commonalities of object-oriented metamodels resided on M2 (e.g. classes, methods, attributes and parameters). Generic refactorings are then specified on a GenericMT's basis. To activate them for a specific metamodel (i.e. a model on M2) a target adaptation is needed. Once such an adaptation exists, every defined generic refactoring can be applied to instances of the adapted metamodel. The adaptation contains the specification of derived properties declared in the GenericMT which are not defined in the metamodel of interest. By this, an aspect-oriented approach is achieved and the newly defined properties are woven into each target metamodel. However, this approach is restrictive with respect to the structures of GenericMT, because it contains exclusively object-oriented elements. DSLs that expose such structure and that have a similar semantics can be treated, while other DSLs cannot. Refactorings that require other structures cannot be

implemented. Furthermore, an adaptation of a target language to the GenericMT is fix and universal for the whole language. This does not allow to map the same structure twice (e.g. if a DSL contains two concepts similar to *method*).

A very similar approach is illustrated in a series of publications realised in the MOOSE platform.[1] In [TDDN00], Tichelaar *et al.* introduced a new constituent of MOOSE: the FAMIX metamodel for a family of languages. In this sense, it is a meta-metamodel comparable to the M3-layer of MOF. Again this meta-metamodel contains object-oriented concepts like *Class* or *Method* only. To support a desired target language a language extension has to be implemented for it [DGLD05; CLMM07]. The refactorings are specified based upon MOOSE. Thus, they are independent from the target language. Nevertheless this approach suffers from the same disadvantage as the previous one because once a language extension is provided the mapping from the language's concepts to the object-oriented MOOSE concepts is fixed. Thus, this approach uses abstraction over the target languages and the claimed requirement of flexible structures in Requirement 2 in Sect. 3.1.1 cannot be assured.

Another approach was published in [ZLG05], where generic refactorings are specified on top of the custom meta-metamodel within the Generic Modeling Environment [LMB+01] (GME). The meta-metamodel of the GME is based on the UML and the authors therefore consider *generic refactoring* as refactoring UML models since UML is used to specify a DSL's metamodel in the GME. Thus, they have a different understanding in the sense that these refactorings can be applied on the metamodel level. Nevertheless, this means that they cannot be reused across different DSL instances but are specific to the meta-metamodel of the GME. With the specification on M3 a metamodel-independent solution of model refactorings is achieved, but it is not meant to be generic across different languages.

In [Läm02] Lämmel proposed an approach which describes a framework for generic refactorings enabling the specification with the functional programming language Haskell.[2] Therein generic refactorings are implemented with typed higher-order functional programming based upon an abstraction interface. Similar to [TDDN00; MMBJ09], this interface considers object-oriented structures only and is intended for specification of generic refactorings for programming languages. Lämmel's framework allows for the reuse of the transformation. But this approach is not model-based (not MOF-conform) and the generic algorithms always need to receive declared and referenced names of elements of a particular language which leads to a large manual effort. This approach could benefit from the work in [NTVW15]. There Neron *et al.* propose a generic framework for the name analysis in programming languages where the authors introduce *scope graphs* enabling the language-independent analysis of declaration and use of named concepts such as methods or variables. A language-specific mapping of the abstract syntax tree (AST) of a programme to its scope graph must be provided. Thus, this approach is generic but it can only be applied to realise rename refactorings.

In [RGdL+13; SGdL14] the authors published an approach similar to ours from [RSA13]. They propose a component model for model transformations and introduce the notion of *concept* acting as abstraction over a transformation. Then a binding is needed to map metaclasses from within the source language to the components in the *concept*. A resulting concrete transformation is generated by a higher-order transformation (HOT) which can then be persisted. This approach is

[1] `http://www.moosetechnology.org/` (visited 10th February 2015)
[2] `http://www.haskell.org/` (visited 10th February 2015)

very similar to the one we present in Chap. 4. It is a general approach for generic transformations, for refactoring they use the Epsilon Wizard Language [KPPR07] (EWL) as transformation engine from the Epsilon tool family[3] [KRGP13]. Nevertheless, there is one main shortcoming in the fact that a structural feature from a *concept* can only be bound to one structural feature in the target metamodel. This results in a lack of flexibility, since this mechanism is tightly coupled to the defined structures in the *concept*. In addition, language-specifics cannot be provided and thus the structures to be transformed and the structures in the *concept* must be almost identical.

Another interesting approach was published in [Ste11]. Steimann argues that every change to a model of a DSL without semantics is considered a refactoring. For static semantics specification he motivates to use well-formedness rules (WFRs) expressing which structures are syntactically correct but have no meaning in a particular language. Steimann's approach then transforms the WFRs to a constraint satisfaction problem (CSP), which does not use *constraint checking* anymore but *constraint solving*. This means that after applying a modification to a model a constraint solver can evaluate which other modifications have to be applied to assure that all constraints, and the WFRs respectively, are satisfied again. By this approach, every modification can be considered a refactoring if related constraints are satisfiable. Therefore it is named *constraint-based refactoring (CBR)*. On the other hand, refactorings are not specifiable explicitly and cannot be defined as a composite atomic operation. But this approach can be considered generic, since it only takes the concept of *well-formedness rule* into account. Thus, it is only dependent on the specification language of the WFRs, namely the Object Constraint Language [OMG14a] (OCL) in Steimann's publication. Since the OCL can be used to constrain any model-based DSL [WTW10], this approach is language-independent.

The last discussed M3 approach is published in [RWZ11]. The authors present the new refactoring specification language *Re$\mathcal{L}$*. This language supports the generic specification of refactorings on a grammar base. The authors define the grammar of *Re$\mathcal{L}$* itself with the Backus-Naur Form [BBG+63] (BNF) and use non-terminals within this syntax as slots for extension. The desired target language must provide its syntax in BNF as well. To establish the connection between both of the mentioned, non-terminals of the core grammar then must be bound in the grammar of the target language. Thus, a new grammar and a refactoring tool for the composition of *Re$\mathcal{L}$* and the target language is generated. Obviously, this approach is not independent from the language's representation and can only be used if a BNF syntax is given. Furthermore, this approach again only abstracts over the target language. Thus, it is not very flexible once the non-terminals are bound.

In summary, one can say that the approaches targeting M3 are limited with respect to the structures refactorings operate on. As a result, they lack flexibility. This observation correlates with the fact that all approaches, except [SGdL14] and [Ste11], abstract over the target languages instead of the desired refactorings.

M2

Other approaches target the MOF layer where metamodels reside—M2. Defining refactorings on top of these metamodels implies that refactorings can work only on one specific language. One of the first publications regarding language-specific refactoring emerged in [SPLJ01]. Sunyé *et*

[3] `https://www.eclipse.org/epsilon/` (visited 10th February 2015)

al. proposed a small set of UML refactorings adapted from existing code refactorings. In this sense, their approach was generalising existing refactorings manually and instantiating them in a concrete language.

The publications in [Köh06; TMM08], [BEK+06b; BEK+06a] and [MTM08] follow the approach of language-specific refactoring and introduce their graphical definition. Here, DSL designers can graphically define a pre-condition model, a post-condition model and a model containing prohibited structures, so-called *negative application conditions*. All models can share links to express their relations and those which are not subject to modification during the transformation. By this technique, a graph is constructed and further analysis can be conducted. The actual model transformation then is executed in terms of graph rewriting with the transformation engine *Henshin*.[4] The comprehensive refactoring-related tooling then was implemented in the tool *EMF Refactor*[5] and was published in [AMT10; ABJ+10; Are14].

A conceptually different approach was published in [HMK05]. Hannemann, Murphy and Kiczales use roles to abstract over the scattered implementations of crosscutting concerns. These concerns then are to be refactored into separate aspects in terms of aspect-oriented programming (AOP). In this sense, the behaviour is preserved while the specific implementation is converted into AspectJ[6] aspects. The particular refactoring transformation is specified with respect to the defined roles. To apply such a refactoring, the roles must be mapped to concrete programme elements in the scattered implementation. The mapping process is supported in terms of suggesting possible other mapping targets while some roles are already mapped (cf. Requirement 8 in Sect. 3.1.1).

A logic- and rule-based approach was published by Van Der Straeten and D'Hondt in [VD06]. The authors propose a solution which first encodes a language's metamodel as *concepts* and *roles* in Description Logics [BMNP03]s (DLs), and the targeted models as *individuals* afterwards. This approach allows for exploiting the advantages of DLs, to reason about the models, and to specify refactorings in a rule system. Furthermore, possible inconsistencies of a model of a certain language are encoded in such a rule system as well. If a rule matches then an according refactoring is triggered automatically based on the rule system. This approach was implemented in the tool *RACOoN* for the UML environment Poseidon.[7] Since the logic-based representation of a target DSL and its models reside in a different technical space (MOF-based) the main limitation of this approach is that all artefacts have to be transformed to DLs instantly. After applying a refactoring, they have to be transformed back again, which we see as a main disadvantage of this approach. Furthermore, this approach is suitable for automatic refactorings only since the inconsistency rules trigger refactorings automatically when they match. Usually, the user wants to retain control, because if she builds up a model, it is not desired to be modified autonomously by a tool. Application of refactorings is no objective task but many subjective factors (such as, e.g., personal preferences, experiences, quality focus) play a role.

Beyond the described approaches, all other conventional model transformation languages and engines can be used to specify refactorings resided at the M2 layer. The only requirement they have to meet is that they must support endogenous in-place model transformations.

The advantage of specifying refactorings on M2 is that the target structures of the metamodel

[4] `https://www.eclipse.org/henshin/` (visited 10th February 2015)
[5] `https://www.eclipse.org/emf-refactor/` (visited 10th February 2015)
[6] `https://eclipse.org/aspectj/` (visited 10th February 2015)
[7] `http://www.gentleware.com/` (visited 10th February 2015)

can be controlled precisely. However, this is also the main disadvantage, since reuse and generic specification are sacrificed. Refactorings for specific metamodels cannot be reused for other languages, but must be defined again although the core steps are the same.

M1

The M1 approaches are motivated from the fact that conventional model transformations usually are specified on the basis of abstract syntax [KLR+12]. Thus, the refactoring designer must be aware of a DSL's metamodel, although she is more familiar with the concrete syntax (model representation) of the models to be refactored. As a consequence, some *by-example* approaches have arisen. Varró was the first researcher who established this term in a general manner, not dedicated to refactorings [Var06]. To the best of the author's knowledge only a few approaches exist regarding the refactoring specification at the M1 layer.

In [BSW+09] and [BLS+09] a refactoring-by-example approach, which was implemented in the *Operation Recorder* of the AMOR project,[8] is presented by Brosch *et al.* It enables the specification of refactorings on concrete instances of a specific metamodel. Modifications are recorded and then abstracted and propagated to the specific metamodel. Since this metamodel is situated on the M2 layer again, this approach does not allow for reuse either.

Sun, White and Gray presented a similar approach in [SWG09]. In contrast to Brosch *et al.* modifications recorded on a concrete instance model, are not propagated to the metamodel level but stored as transformation patterns which can be replayed on other instances of the same metamodel. This approach again is restricted to instances of one metamodel and thus addresses reuse of transformations in models and not across languages.

Langer, Wimmer and Kappel presented a by-example approach for model transformations in general [LWK10]. Their contribution is the *REMA* process in the first place. They support the incremental derivation of transformation rules by denoting input and output model elements, instead of establishing correspondences of the whole structure which has changed. When the source and target language is the same and the model transformation language, which the actual transformation is generated for supports in-place transformation, then this approach can be used for model refactoring as well.

3.1.3. Evaluation

The evaluation of the related work and other relevant approaches is presented in Table 3.2. Therein, the entries in the column *Approach* contain a characteristic name and the most important publication of the according approach. If no characteristic name is given, the authors are referred to. The Category-column references the overall category as introduced in Fig. 3.1 and Table 3.1. The names and figures in the other column heads identify the according elaborated requirements in Sect. 3.1.1.

As can be seen in the table, none of the discussed approaches meets every requirement. Notably, requirements 6–9 are kind of non-conceptual requirements but regard the implementation of an approach. These requirements were evaluated mainly as not fulfilled since no information regarding them was published or these features have just not been realised. The CBR approach of Steimann is the one being evaluated positively exclusively in the conceptual requirements.

[8] `http://www.modelversioning.org/` (visited 10th February 2015)

Table 3.2.: Comparison of related work regarding generic model refactoring.

MOF Layer	Approach	Genericity (1)	Flexibility (2)	Specificity (3)	Behaviour Preservation (4)	Pre- And Post-conditions (5)	Atomicity (6)	Reversibility (7)	Specification Suggestion (8)	Application Suggestion (9)	Interoperability (10)	Category
M3	GenericMT [MMBJ09]	+	○	○	○	+	−	−	−	−	+	R
	MOOSE [TDDN00]	+	○	+	○	+	−	−	−	+	○	R
	GME [ZLG05]	−	○	+	○	+	−	−	−	−	○	R
	Lämmel [Läm02]	+	○	−	○	+	−	−	−	−	−	R
	Epsilon EWL [RGdL+13]	+	○	+	○	+	−	+	○	○	+	R
	CBR [Ste11]	+	+	+	+	+	−	−	−	−	+	R
	Re$\mathcal{L}$ [RWZ11]	+	○	−	−	+	−	−	−	−	−	R
M2	EMF Refactor [Are14]	−	+	+	○	+	−	+	−	+	+	R
	Roles AOP [HMK05]	−	+	+	○	+	−	−	+	−	−	R
	RACOoN [VD06]	−	+	+	○	+	−	−	−	○	−	R
M1	AMOR [BSW+09]	−	+	+	−	+	+	+	−	−	+	R
	Sun, White and Gray [SWG09]	−	+	+	−	+	−	−	−	−	−	R
	Langer, Wimmer and Kappel [LWK10]	−	+	+	−	+	−	−	+	−	−	R

The smarter this approach is the more deficiencies regarding user expectations it has. Similar to the RACOoN approach of Van Der Straeten and D'Hondt, users do not expect to trigger automatic modifications after every model change. Furthermore, the work of Steimann is neither suitable to assemble a refactoring catalogue as argued in Hypothesis 3, nor can refactorings be recommended.

Except the GME of Zhang, Lin and Gray, all the other M3 approaches provide means to specify model refactorings in a generic manner. Nevertheless, all of them lack flexibility in the sense that no appropriate abstraction mechanism is used. Instead of abstracting over the refactorings, they abstract over the target languages and most often only object-oriented DSLs are supported. As a consequence, most of these M3 approaches use some kind of unifying meta-metamodel to capture commonalities of the potential target languages. Thus, the authors had to decide which concepts to include and how abstract these should be. The trade-off implied by this procedure would be to only use one single generic concept in the unifying meta-metamodel. Obviously this is not feasible, since different types of model elements cannot be distinguished anymore. In this sense, the discussed M3 approaches are generic in nature but are too static in their concepts. This results in a reduction of the potential languages, which refactorings could be provided for.

On the other hand, the M2 and M1 approaches lack genericity, but are powerful in flexibility and specificity. That is the nature of their metalayers. Since the M1 approaches capture the intended refactorings by examples and then propagate them to the M2 layer for reusing them in different models of the same language, the M1 approaches suffer from the same disadvantages and profit from the same advantages as the M2 approaches.

Many of the presented approaches are evaluated neutrally regarding the requirement of behaviour preservation. The reason is that the discussed publications presented some pointers about how to verify that the semantics did not change. An in-depth discussion regarding this requirement can be found in Sect. 4.3.

Based on the analysis of related works in the field, we observed that the specification of refactorings on a single MOF layer does not yield a satisfactory result. Therefore, a technique is needed that is able to combine the advantages of layer M3 and M2 specifications. Such a technique must solve the problem that M3 approaches are limited to a specific group of languages, by allowing to use multiple structural patterns rather than a single one (e.g. the object-oriented concepts). It must also address the limitation of M2 approaches, which are specific to one language. A dedicated solution should allow DSL designers to reuse individual generic refactorings for multiple different languages.

3.2. Determination of Quality-Related Deficiencies

As argued in Hypothesis 2, the bad smells of Fowler *et al.* are vague and imprecise. Furthermore, a relation to qualities is only implicit and we argue that it must be specified explicitly. Exactly the same holds for the relation to resolving refactorings, since refactorings might improve particular qualities [FBB+99; SSL01; MTM07; Als09]. Thus, we want to give an overview of related work regarding their potential for determining quality-related deficiencies. To the best of our knowledge, no approaches exist correlating all three constituents explicitly: quality, related deficiencies (bad smells), resolving refactorings. Therefore, we analyse related work in a broader sense and review other approaches regarding their abilities to resolve deficiencies in models in

general.

Again, we specify comparison criteria first and then analyse the related work and the state-of-the-art. Afterwards, the related approaches are compared by means of the criteria, before we draw a conclusion.

3.2.1. Requirements

In the following, we discuss requirements that must be fulfilled by an approach for determination of quality-related deficiencies in models of arbitrary DSLs and their resolution in terms of applying refactorings. These requirements emerged in a reflection process over the problems, goals, and solutions discussed in sections 1.1, 1.2 and 1.3 respectively.

1. Explicit Quality Relation: As already argued, the relation of deficiencies in models to qualities is only implicit. Consequently, tools implementing such an approach cannot give evidence about which particular quality requirements are violated. More precisely, this implies that the term *quality* must be conceptualised and related to a concept of *model deficiencies*.

2. Explicit Refactoring Relation: In addition to the previous requirement an explicit relation of model deficiencies to refactorings is crucial to support the fact that refactorings can improve certain qualities by removing particular bad smells. Therefore a concept of *model deficiency* must be conceptually related to refactorings.

3. Quality-dependent Refactoring Suggestion: In order to support the resolution of quality-related model deficiencies, an approach must provide means to suggest concrete refactorings being able to resolve certain deficiencies. For this, the previous two requirements are not a necessary condition. If no explicit relations to refactorings and/or qualities exist, refactorings can still be suggested by means of an implicit connection to qualities, as it was illustrated, for instance, in [FBB+99]. This requirement also corresponds to Requirement 9 in Sect. 3.1.1 regarding the application suggestion of refactorings.

4. Metrics-based Detection: Since the use of metrics is a well established technique to give evidence about certain qualities in models [Soc93; CK94; SSL01; BD02; AST10; KVGS11; SK11], an approach should support the metrics-based detection of model deficiencies.

5. Structure-based Detection: Formalisations of structures, such as anti-patterns or architectural bad smells, are also a common technique to find deficiencies in models [SW03; KE07; GPEM09; ABT10; KGH10; DMTS12]. Thus, an approach must support the structure-based detection of deficiencies in models.

6. Cause Tracing: The three previous requirements must be seen in context. On the one hand, it is not sufficient that the result of a metrics- and/or structure-based detection approach is just the information of the presence of deficiencies. On the other hand, there is no benefit in having the information that a particular refactoring is able to resolve certain deficiencies. Both the detection and the resolution must be traceable to the concrete model elements causing the deficiency. The detection's output must be the refactoring's input.

7. Language Independence: An approach should be completely independent from particular languages. It must be applicable to arbitrary DSLs and no restrictions regarding a DSL's metamodel should be stated.

8. Language Specifics: Since the concrete occurrence of a model deficiency is specific for a certain setting, as, e.g., it may be language-, platform- or framework-dependent [RBA14], an approach must provide means supporting the setting-specific definition of a model deficiency. This requirement does not doubt that in principal such deficiencies are universal from an abstract point of view. Only the concrete occurrence is specific.

9. Interoperability: Here the same holds as in Requirement 10 for generic model refactoring in Sect. 3.1.1. This requirement comprises independence from a model's representation (e.g. tree-based, textual, graphical) and that it must conform to MOF.

3.2.2. Literature Review

First, we have to point out the fact that in the upcoming presented works several similar but new terms have been established. When we introduce them, they are highlighted with an italic font shape to emphasize that they capture a specific meaning of the general term *deficiency* we used until now.

In the work from Pathak *et al.* the authors introduce the new term *energy bug* saying that it is an error of the hardware, a mobile application or firmware causing unexpected high energy consumption on a mobile phone [PHZ11]. Obviously, energy consumption is an important issue on mobile devices since even popular applications have bad energy properties and many of those energy bugs have been introduced through updates [Wil14]. In [PJHM12] the authors present an approach based on data-flow analysis to statically find the so-called *no-sleep bugs* in Android applications. Furthermore, they implemented a tool supporting the detection of these bugs. Manual resolution hints are given. Thus, Pathak *et al.* consider one specific quality, namely *energy consumption*. Since this approach is only implicitly related to a quality, but is able to detect energy bugs, we classify this work into category SD.

Gottschalk, Jelschen and Winter also focus on Android devices and introduce specific *energy code smells* in [GJJW12; GJW14]. They use a graph-based approach for defining, detecting and resolving energy deficiencies. Furthermore, they provide a catalogue of *energy refactorings* containing Android- and Java-specific energy deficiencies and refactorings to resolve them. Behind the scenes their approach is implemented using *TGraphs* and the graph query tool *GReQL* [BERS08]. By using the intermediate representation of a TGraph any target language is supported given that a translation to a TGraph exists. Since the approach of Gottschalk, Jelschen and Winter only takes the quality *energy consumption* implicitly into account but is able to detect and resolve energy code smells we classify it into category R-SD.

Similar research regarding the quality *energy consumption* or *performance* in mobile applications was elaborated, e.g. [HB10; VAPM13] or [LXC14] respectively. But the illustrated work of Gottschalk *et al.* and Pathak *et al.* is considered to be sufficient to provide insights about the research in the area of energy consumption in mobile devices and applications.

Neukirchen and Bisanz investigate *test smells* in test suites using the testing and test control notation (TTCN-3) [NB07]. They argue that test suites suffer from quality problems such as usability, maintainability, or reusability. In previous works, the authors found out that metrics

are suitable to detect either very local or very global quality problems in Java test code generated by the TTCN-3 [ZVS+07]. But in [NB07] they argue that a pattern-based approach is more powerful to tackle deficiencies distributed over the source code. Thus they make the important observation that quality issues can be cross-cutting over a whole system [TOHS99]. The authors provide a catalogue of test smells for TTCN-3 test suites and automate the detection with the tool *TRex*.[9] It also supports the resolution of those test smells by refactoring. Furthermore, Neukirchen and Bisanz argue that the notion of *metric* and *test smell* is not disjoint and a test smell can be considered as a metric by just counting the occurrences of the test smell. In this sense, the TRex approach is specific for the TTCN-3 notation, but it supports both metrics-based and structure-based detection of deficiencies. This approach is classified into category R-SD since it can detect and refactor test smells. Subsequent work supporting that direction is [GvDS13; GZvDS13].

Other test-specific work was published in [vRDDR07; BV08]. They introduce concrete test smells and detect them with the help of metrics. They implemented their approach in the tool *TestQ*[10] which currently supports the programming languages Java and C++. Unfortunately the approach of Breugelmans and Van Rompaey does not explore beyond the detection and visualisation of test smells and therefore is classified into category SD.

In [AT12], Arendt and Taentzer introduce a tool based upon the EMF. It serves for detection and resolution of deficiencies in models. They rely on a graph-based approach since they perform graph matching for detection and graph rewriting for resolution with their tool Henshin [ABJ+10]. For resolving smells they implemented the tool EMF Refactor [AMT10]. The theoretic foundation of their tools has been published by Arendt in [Are14]. The author provides a concept supporting the explicit relation of *model metrics* and *model smells* for resolving model refactorings. Model metrics must be provided by implementing a Java calculation interface or by specifying them with an OCL expression. Model smells are provided by means of graph patterns. The refactoring capabilities are already known from the discussion of their approach in Sect. 3.1.2 at the M2 layer and can be seen at a glance in Table 3.2. The examples used within Arendt's thesis illustrate the approach applied to UML- and Ecore-based models but it is independent from a particular target DSL. The only requirement a language has to fulfil is that it must be MOF-based. Their tools are mature and well-integrated into the Eclipse platform. Unfortunately, again they do not expose an explicit *quality* concept. Thus developers or modellers cannot focus model deficiencies related to particular cross-cutting qualities. Therefore, the approach of EMF Refactor is classified into category R-SD.

Another EMF-based tool suite being able to support detection and resolution of quality-aware deficiencies in models is the Epsilon tool family [KRGP13]. It provides a set of languages for, among others, validating, transforming, generating or comparing EMF-based models. Thus, the team provides the Epsilon Validation Language enabling the specification of constraints for specific modelling languages. This language conceptually extends OCL and can be applied to define structure-based deficiencies with constraints. When constraints are violated, the Epsilon Wizard Language [KPPR07] (EWL) provides means for specifying refactorings to resolve them. The Epsilon tool suite provides languages to cover the detection and resolution of model deficiencies, but again lacks an explicit relation to qualities. Thus the comprehensive Epsilon

[9] `http://www.trex.informatik.uni-goettingen.de/` (visited 10th February 2015)
[10] `http://tsmells.googlecode.com` (visited 10th February 2015)

family is classified into category R-SD.

Apart from the related work discussed so far, approaches from the area of *multi-quality architecture optimisation* exist. First to mention, Koziolek presented an approach for quality-driven optimisation of component-based architectures in terms of the Palladio component model [BKR07][11] in the domain of *self-adaptive systems*. The author mainly takes the qualities *performance*, *reliability* and *cost* into account but the approach is independent from particular qualities. Koziolek proposes a component-based development process with an explicit *quality analysis* step. Therein, the software architect annotates relevant quality criteria to an initial architecture specification serving as input for the upcoming process step of *architecture exploration*. The objective of this step is, first, to identify the design space and, second, to optimise the architecture with respect to the annotated quality criteria based on quality prediction. Optimisation is automatically conducted by application of several transformation strategies. In this sense, they can be considered as refactorings, since the meaning of the architecture is preserved, while quality properties are optimised. Koziolek's approach is implemented in the tool PerOpteryx.[12] Since all three constituents in terms of deficiencies, related qualities and refactorings (architectural optimisations) are covered by this work we classify it into the category R-SD-Q. Nevertheless, it has a different scope than this thesis since its objective is an automation process. In addition, it is specific for the Palladio component model and not intended to be language independent.

Staying in the area of software architecture, Trubiani published an approach regarding *performance anti-patterns* in [Tru11]. She provides the EMF-based Performance Antipattern Modeling Language (PAML) supporting the specification of architectural performance anti-patterns in a model-based manner. Furthermore, a catalogue of several established performance anti-patterns encoded in PAML is presented. A PAML instance always refers to a resolving refactoring, which improves a certain performance issue. The provided patterns are exemplarily applied to instances of the Palladio component model and UML+MARTE [OMG11b]. But, since it is based on MOF, arbitrary other MOF-based architecture specification languages are supported. Trubiani's approach is dedicated to the quality *performance* and the anti-patterns are explicitly related to resolving refactorings. Thus we classify this work into the category R-SD.

Another interesting approach for removing design anti-patterns in programmes was published by Dietrich *et al.* in [DMTS12; SDM13]. Here a dependency graph of a programme is constructed, containing vertices for artefacts like classes, packages or libraries, and edges for relations to other artefacts, such as, e.g., a use-relation. The whole subsequent analysis is based on this dependency graph. Thus, this approach is limited to those object-oriented languages for which a translation into a dependency graph is provided. Exemplarily Dietrich *et al.* used Java as target language. Then their graph querying tool GUERY[13] is used to query the graph for a set of anti-patterns, such as, e.g., *circular package dependencies*. The objective is to suggest a so-called *high impact refactoring* to resolve as many anti-patterns as possible. This is achieved by ranking the found dependencies by means of the amount of the different anti-patterns a single dependency is contained in. Thus, a dependency contained in more anti-pattern instances gets a higher score. The dependency with the highest score is the one having the highest impact, if it is removed because more anti-pattern instances will disappear then. They give suggestions about

[11] `https://sdqweb.ipd.kit.edu/wiki/Palladio_Component_Model` (visited 10th February 2015)
[12] `https://sdqweb.ipd.kit.edu/wiki/PerOpteryx` (visited 10th February 2015)
[13] `https://code.google.com/p/gueryframework/` (visited 10th February 2015)

resolving refactorings, but do not apply them explicitly. Furthermore, this approach is classified into category SD, since the relation to qualities is only implicit.

Beyond the approaches intended to detect and resolve deficiencies, there is further related work in a broader sense supporting only the detection of deficiencies. On the one hand, there are publications illustrating structure-based graph querying approaches [BERS08; BHH+12; USH+15]. In isolation, they cannot be used to satisfy the whole tool chain needed for quality-aware detection and resolution of model deficiencies. But they can be used in combination with other tools as e.g. Gottschalk *et al.* did with GReQL [GJJW12] and as we did with IncQuery[14][BHH+12] in [RA13; RBA14]. On the other hand, there are metrics-based approaches for deficiency detection [Mar01; SSL01; BD02; CLMM07; AST10; MGDL10; KVGS11; SK11; DPXT12]. Similar to the structure-based approaches they can support the whole tool chain only in combination with other tools. But in isolation, these approaches and tools result in no new insights or benefits and, thus, are not included in the comparison (cf. Table 3.3).

3.2.3. Evaluation

The evaluation of the related work and other relevant approaches is presented in Table 3.3. Therein, the entries in the column *Approach* contain the name of the respective approach or implemented tool realising it and the most important publication of the according approach. The column *Category* refers to the overall category as introduced in Fig. 3.1 and Table 3.1. The names and figures in the other column heads identify the according elaborated requirement from Sect. 3.2.1.

With respect to the objectives of this thesis, none of the discussed approaches fulfils every of our requirements. The reason is that only few of these approaches were proposed for providing a solution for the quality-related detection and resolution of model deficiencies as we intend.

On the one hand, we illustrated approaches targeting only specific qualities in a certain context or setup. In [PJHM12] and [GJW14] mature means are provided for detecting and/or resolving energy-related deficiencies in Java-based applications for mobile devices. Their approaches are mature and promising regarding the reduction of the energy consumption. Furthermore the PAML approach presented in [Tru11] contributes a language for the specification and refactoring-based resolution of performance anti-patterns. Then we have related work regarding the detection of deficiencies in test code [NB07; BV08]. It revealed the insight that quality concerns [TOHS99] can be cross-cutting, which is not trivial to maintain. As a representative of graph-based approaches supporting the detection of deficiencies we chose the work of Dietrich *et al.* Besides the fact that those approaches must be combined with other mechanisms to support a quality-aware engineering tool chain, nevertheless the work presented in [DMTS12] is able to recommend resolving refactorings for anti-patterns in dependency graphs.

The only discussed approach having an explicit relation to qualities is the one contributed by Koziolek in [Koz11]. Here the most important drawback is that it is dependent on the Palladio component model. Furthermore it has a different scope and aims at optimising software architectures in general.

The most promising approach was contributed in EMF Refactor by Arendt in [Are14]. It was evaluated completely as positive, except the fact that it lacks an explicit relation to a *quality*

[14] `https://www.eclipse.org/incquery/` (visited 10th February 2015)

Table 3.3.: Comparison of related work regarding quality evaluation.

Approach	Quality Relation (1)	Refactoring Relation (2)	Refactoring Suggestion (3)	Metrics-based (4)	Structure-based (5)	Cause Tracing (6)	Language Independence (7)	Language Specifics (8)	Interoperability (9)	Category
Energy Bugs [PJHM12]	○	○	○	−	+	+	−	+	−	SD
Energy Refactoring [GJW14]	○	+	+	−	+	+	−	+	○	R-SD
TRex [NB07]	○	+	+	+	+	+	−	+	−	R-SD
TestQ [BV08]	○	−	−	+	−	+	○	+	−	R-SD
EMF Refactor [Are14]	○	+	+	+	+	+	+	+	+	R-SD
Epsilon Validation [KRGP13]	○	+	○	−	+	+	+	+	+	R-SD
PerOpteryx [Koz11]	+	○	○	+	+	+	−	+	○	R-SD-Q
PAML [Tru11]	○	+	+	−	+	+	+	○	+	R-SD
GUERY [DMTS12]	○	○	+	−	+	+	○	+	○	SD

concept. This approach and the implemented tool are very mature and suitable for quality-aware engineering. The only limitation is that developers or modellers cannot focus on dedicated qualities explicitly.

Thus, to the best of our knowledge 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. It allows developers for focussing specific qualities in isolation.

3.3. Co-Refactoring

As stated in Hypothesis 4 in Sect. 1.3, the process of co-refactoring is essential to preserve consistency of models being dependent on a initially refactored model. As a consequence of the applied refactoring, dependent modifications in the subsequent models must be propagated in terms of a co-refactoring.

Having this in mind, we will derive the implied comparison criteria in terms of requirements for a co-refactoring approach in the following section before related work is analysed. Afterwards the comparison concludes this section.

3.3.1. Requirements

In the following, we discuss the requirements co-refactoring approaches for models of arbitrary DSLs must fulfil. These requirements emerged in a reflection process over the problems, goals

and solutions discussed in sections 1.1, 1.2 and 1.3 respectively.

As a preliminary consideration, we want to recapitulate the use case of co-refactoring shortly. A co-refactoring must be applied as a reaction to a preceding model refactoring. This must be the case for models being dependent on the initially refactored model. Thus, not only the models depend on each other but also the preceding and succeeding refactorings. In this regard, it might be the case that some values needed in a co-refactoring must be derived from values already given from the initial refactoring. As a consequence, the following requirements must be fulfilled by approaches for co-refactoring.

1. Dependent Models Detection: As a first step in the co-refactoring process, i.e. before a particular concrete co-refactoring is to be applied, all models that depend on the initially refactored model must be determined. For every detected dependent model a co-refactoring process has to be initiated. Different kinds of model dependencies are discussed in Sect. 8.3.1.

2. Dependent Elements Detection: In addition to the previous requirement suppose that a pair of an initially refactored model (source) and a dependent model (target) is given. Based on the concrete model elements which participated in the refactoring of source corresponding model elements in target have to be determined. Consider, e.g., the renaming of a UML class. This step then must reveal the corresponding Java class. After the initial renaming the UML and Java classes have different names.

3. Incoming Refactoring Declaration: To specify a dependent modification in terms of a co-refactoring an approach must be able to refer to an incoming refactoring the succeeding co-refactoring depends on. In this sense, an incoming refactoring must be declarable.

4. Condition Specification: To specify a dependent modification in terms of a co-refactoring, a condition must be specifiable expressing the circumstance which a co-refactoring is valid to be applied upon.

5. Outgoing Co-Refactoring Declaration: In order to specify a dependent modification in terms of a co-refactoring an approach must be able to refer to an outgoing co-refactoring being the reaction to an incoming refactoring. In this sense, an outgoing co-refactoring must be declarable.

6. Dependent Binding Specification: A co-refactoring approach must be capable of specifying the needed values in a concrete outgoing co-refactoring dependent on the values of the incoming refactoring. In this sense, the outgoing values must be bound with respect to the incoming values. Consider again the small UML-Java class renaming example mentioned above. The user should not be prompted for the name of the Java class in advance, since usually it is the same as the source UML class name. Therefore a dependent value binding must be specifiable.

7. Language Independence: Regarding the independence of the supported languages the same holds as for the generic refactoring (cf. Sect. 3.1.1) and for the model deficiencies (cf. Sect. 3.2.1). The approach of co-refactoring should be applicable for heterogeneous models.

8. Language Specifics: Nevertheless, it must be possible to take into account concrete language properties for the definition of dependent modifications. A co-refactoring designer must be enabled to reflect over her DSL's specifics which then can be subject in the concrete co-refactoring specification.

9. Interoperability: For interoperability, exactly the same holds as in Requirement 10 for generic model refactoring (cf. Sect. 3.1.1) and in Requirement 9 for quality-related model deficiencies (cf. Sect. 3.2.1). This requirement comprises independence from a model's representation (e.g. tree-based, textual, graphical) and it must conform to MOF.

Requirements 3, 4 and 5 can be considered as some kind of event-condition-action (ECA) rules for co-refactoring. The incoming refactoring is the occurring event and the outgoing co-refactoring is the resulting action if the condition is satisfied.

3.3.2. Literature Review

The term *co-refactoring* is only rarely used in literature since there are only few appropriate co-refactoring approaches targeting the same problems as are to be solved in this thesis. These approaches are highlighted in the upcoming sections. But there is a wide field in the area of *co-evolution* in MDSD in general, which serves as related work as well. A plenty of approaches have been published in literature and there are two threads of research all of them can be classified into. On the one hand, co-evolution at different MOF layers is distinguished. Precisely, this means that a metamodel evolves and its instances (models) have to co-evolve to re-establish their consistency. On the other hand, co-evolution is considered at the same MOF abstraction layer. This is also the case in our approach. In the following, we divide related work by means of these two distinctions.

Different Abstraction Layer

One of the most significant works was published by Wachsmuth in [Wac07]. The author based this work on object-oriented refactoring and grammar adaptation and investigated two main aspects. First, Wachsmuth describes the process of evolving a MOF-based metamodel in terms of separate adaptation transformations. Thus, the usually manually conducted modifications are made explicit in terms of precisely defined transformations. Second, for every metamodel adaptation transformation a corresponding co-transformation for the instances is provided being executed instantly. Consequently, a pair of a metamodel transformation and a model co-transformation is a *coupled transformation* [Läm04] and the models conform to their metamodels at any time. A co-transformation pattern is parameterised with its triggering metamodel transformation. Some of Wachsmuth's co-transformations are considered to be a co-refactoring since the corresponding initiating transformation is a refactoring, as, e.g., the renaming of a property. Therefore, this approach is classified into the category R-CoR. Nevertheless the presented approach is mainly of theoretic nature and nothing is said about the practical detection of dependent models, namely the metamodel instances. The problem is the inversion of the instance-of relationship in case a metamodel evolves. Metamodels do not know their instances and, thus, detection of dependent models is important.

In [HBJ08] Herrmannsdörfer, Benz and Juergens also present a classification of coupled changes on metamodels and models into the three groups: *model-specific*, *model-independent* (but metamodel-dependent) and *metamodel-independent* modifications. Model-specific modifications require information which varies from model to model. Model-independent modifications utilise knowledge of the application domain of the modified metamodel. Metamodel-independent changes can be reused for various analogous evolution scenarios. Therefore, the authors propose to generalise them into reusable operations. Based on that preliminary work Herrmannsdörfer, Benz and Juergens published their operation-based approach in [HBJ09] and implemented it in the tool COPE, which is now under the patronage of the Eclipse community in form of the Edapt tool.[15] The authors propose a language that allows for decomposition of a modification into manageable, modular coupled changes. Furthermore, the language provides means for metamodel-independent changes but is at the same time expressive enough to describe metamodel-specific changes. Further work was published in [HVW11] and Herrmannsdörfer, Vermolen and Wachsmuth present an extensive list of reusable coupled modification operators, which evolve a metamodel and are able to automatically migrate existing models in response. The authors argue that their coupled operators do not result in *breaking non-resolvable* changes [GKP07], since a coupled operator always provides a migration to resolve a breaking change. In this sense, only *non-breaking* and *resolvable breaking* changes can occur. The collection of coupled evolution operators can be extended by new ones. In [HK10], their approach of coupled transformation of metamodel instances is expanded to the preservation of formally specified semantics, in the sense that modifications are refactorings. Therefore, this approach is classified onto the category R-CoR. A semantics specification is then automatically adapted to the new metamodel version. The drawback of this approach is that it "only" supports adaptation. That means, clients of the evolved metamodels only see the "old" semantics and the gap gets bigger with every evolution.

Criticism concerning the previously discussed approach regards the facts that a new co-evolution language has to be learnt (such as Edapt) and a model transformation language for model migration must be used. Meyers *et al.* tackle this drawback by an operator-based approach using in-place transformations generated automatically by a HOT parameterised with a metamodel modification step [MWCS11]. The difference between two metamodel versions is regarded as a sequence of *difference operations*, each of which mapping to a corresponding *migration operation*. Thus, this is an approach of coupled operators as well. Models are then migrated after every difference operation which ensures conformance to the evolved metamodel at any time. In case no migration operation can be generated, the user is to be involved into the process and can specify manual adaptation steps. This can occur when a difference operation changes semantics and a semantics-preserving migration operation (refactoring) cannot be derived automatically. Because of this consideration, this approach is classified onto the category R-CoR.

In [RKP+14] Rose *et al.* argue that conventional model transformation and programming languages do not suit well to reflect the model migration scenario properly in the sense that they require users to specify identity transformations or to refer to underlying technical details of the modelling technology. Examples of additional manual effort are, first, that during migration all elements are copied to a new model although some of them are likely not to conform to the

[15] `https://www.eclipse.org/edapt` (visited 10th February 2015)

evolved metamodel anymore and, thus, have to be deleted afterwards. And, second, existing languages take the physical representation of models (as, e.g., representation as XML Metadata Interchange [OMG14b] (XMI)) and/or technical details of the modelling framework into account. To overcome these drawbacks, the authors propose a new model transformation language principle better suited for model migration: *conservative copy (CC)*. CC consumes the original model as input and produces a migrated model as output in the sense that it only copies those elements conforming to both the source and the evolved target metamodel. In order to be independent from the underlying model technology Rose *et al.* abstract over some technology-specific aspects as, e.g., *value conversion* which has to be done before copying in order to conform to the migrated metamodel instead of the original one. A model connectivity layer must be provided for a particular modelling technology. CC was implemented in the context of the Epsilon language family within the tool *Epsilon Flock*.[16] Since this approach takes into account only conventional model migration in general, we classify Epsilon Flock into category CoR.

Furthermore, Burger and Gruschko present an approach to create a change metamodel for MOF-based metamodels in [BG10]. Even though they do not address the problem of co-adapting models after metamodel changes, they present an elaborate classification of the impact of metamodel changes on instantiating models and propose an analysis process. With the change metamodel created from two different metamodel versions it is possible to estimate the compatibility of metamodel changes with existing model instances. Thus their approach is state-based as well. Furthermore, it supports sequences of metamodel changes. Their comprehensive approach was then published by Burger in [Bur14]. The context of Burger's approach is the *view-based* MDSD which is based on the *orthographic software modeling* approach of Atkinson, Stoll and Bostan published in [ASB10]. They assume a single underlying model (SUM) of the system and all other *convenient* models are views on the SUM. Views are created on-demand and therefore their metamodels need to be generated instantly for which Burger's approach is used. Thus, the view metamodels might quite often be subject to evolution. It is implemented in the Eclipse-based tool Vitruvius.[17]

Same Abstraction Layer

First, we want to discuss a *real* co-refactoring approach based on the constraint-based refactoring approach of Steimann already discussed in Sect. 3.1.2 with respect to generic model refactoring [Ste11; Ste15]. This approach is extended by von Pilgrim *et al.* to cover the co-refactoring between models and generated code [vPUTS13]. In this work, cross-language constraints are generated by the original model-to-code (M2C) transformation. The previous CBR approach maps the refactoring problem to a constraint-solving problem and WFRs are converted into rules upon which a constraint-solver determines which additional modifications have to be applied after an evolution step in the model. For reusing this approach for co-refactoring, von Pilgrim *et al.* argue that correspondences from model to code elements are needed. These correspondences are then represented as cross-language constraints in their CBR-based refactoring specification language *Refacola*.[18] An example would be the constraint that a generated Java class is equally named as the original UML class. Once such cross-language constraints are specified, the approach

[16] `https://www.eclipse.org/epsilon/doc/flock/` (visited 10th February 2015)
[17] `https://sdqweb.ipd.kit.edu/wiki/Vitruvius` (visited 10th February 2015)
[18] `http://www.fernuni-hagen.de/ps/prjs/refacola/` (visited 10th February 2015)

works in the same sense as their *single-artefact* CBR approach of [Ste11; Ste15]. The constraint solver then determines which constraints are violated and applies the derived co-refactoring steps needed to transit the dependent model into a consistent state. Therefore, we classify this sophisticated approach into category R-CoR. Nevertheless, this approach stands or falls on the exploitation of created trace links, from which, on the one hand, the cross-language constraints are generated, and, on the other hand, dependent models and dependent model elements are detected. If no trace links are created during code generation it demands huge effort to specify the cross-language constraints manually. Furthermore, neither dependent models nor dependent elements can be detected easily.

Another view-based approach is presented by Wimmer, Moreno and Vallecillo in [WMV12]. They argue that heterogeneous systems contain different views upon the same information, such as, for instance, the different diagram types in the UML. As a consequence, correspondences between elements in different views exist. The challenge is to preserve consistency in all views as a result of an evolution in one view. The authors criticise that other view-based approaches take evolution into account only at a coarse-grained level, resulting only in a consideration of atomic operators such as adding or removing elements. Thus, the real intent of an evolution gets lost and fine-grained consistency preservation is needed. In their approach, the authors use the logic-based rewriting tool *Maude*[19] for the specification of the system, since it has an efficient rewriting and analysis engine. In this regard, every view is encoded as an object-oriented Maude module. In order to detect changes Wimmer, Moreno and Vallecillo use a 2-phase comparison approach to, first, recognize fine-grained changes based on the element's identifiers, and, second, to derive coarse-grained changes from them. The derivation of coarse-grained changes is accomplished by means of graph transformation patterns. These patterns are not executed but matched. Maude then is able to find instances of these patterns based on the detected fine-grained changes. Subsequently coupled transformations, dependent on the determined coarse-grained pattern instances, propagate the changes to dependent views. The coupled transformations again are encoded in Maude. Similar to the previously discussed approach in [vPUTS13], Wimmer, Moreno and Vallecillo encode all models in terms of a language an engine can compute solutions upon. The difference is that this approach captures the intent of the fine-grained modification quite well as coarse-grained modifications. In this sense, it might also be possible to encode refactoring patterns, which then can be detected as coarse-grained changes. Thus, this approach is very similar to the one presented in Chap. 8, but there is a large overhead in terms of the 2-phase comparison approach. Furthermore, the user does not have support right from the beginning regarding the fact that a refactoring is to be applied explicitly, in contrast to the analysis of executed atomic changes. In addition, correspondences between models and dependent elements must be specified explicitly. This is really an interesting approach.

A very similar approach to the previously discussed one is presented in [EPRV08] by Eramo *et al.* The main difference is that a state-based approach is applied to capture the changes. Furthermore, this approach does not expose coarse-grained modifications, which can be used to detect refactorings. As the underlying logic-based formalism *Answer Set Programming* is used. Since this approach does not reveal more insights, we do not take it into account for comparison.

Another view-based approach is presented by Getir, Rindt and Kehrer in [GRK14]. They provide an analysis framework which exploits the evolution history and derives dependent

[19] `http://maude.cs.uiuc.edu/` (visited 10th February 2015)

co-evolution steps from it. These serve as a recommendation for ongoing transformations in order to keep dependent views consistent. In this approach dependencies between concrete model elements are established in terms of trace links. The authors apply a *coupling analysis* to determine intended co-evolution steps as a result of an initial evolution from the evolution history. As a result of this analysis coupled changes are derived. This approach is implemented in the EMF-based tool *SiLift*.[20] Since they also consider refactorings, we classify their work into the category R-CoR.

In [GW09], Giese and Wagner propose an incremental approach regarding the use of triple graph grammars (TGGs) to resolve model inconsistencies in a bidirectional manner instead of re-generating a whole dependent model. They also use a correspondence model for reflecting the connection between an initially evolved model and a dependent model which has to co-evolve. This correspondence model is established at the metamodel level and maps elements from the source to elements from the target metamodel. Bindings from source values to target values are encoded as constraints in the TGG. The actual transformation rules in terms of graph rewriting are derived from the TGG rules. The incremental nature of this approach is achieved by the fact that the actual graph rewriting is not initiated on the root correspondence node rather than on the correspondence node related to the node which was initially modified. In this sense, this approach does not traverse the whole correspondence graph. To achieve consistency preservation of the structures in a dependent model previously performed forward propagation steps and their dependencies are revoked. This approach is implemented in the Fujaba tool suite[21] supporting the TGG formalism for model transformation. Using this TGG approach for a co-refactoring scenario is quite similar to the recently discussed work of Wimmer, Moreno and Vallecillo [WMV12]. Source patterns in TGG rules can be considered as a coarse-grained change and the target patterns correspond to the coupled transformation of Wimmer, Moreno and Vallecillo respectively. Thus, this approach suffers from the same disadvantages as discussed for [WMV12] and consequently the evaluation in Table 3.4 is similar. We classify this approach into category CoR.

Based on the approach of incremental synchronisation from Giese and Wagner a subsequent approach was published in [HEO+15] by Hermann *et al.* It uses the formal foundations from [Dis11] and based upon a TGG a synchronisation framework for a specific context is generated. Again, the synchronisation transformations, being dependent on the initially occurred evolution, are generated from the TGG rules. Thus, we get no new insights regarding co-refactoring for our concrete setting and requirements. Therefore this approach is not part of our evaluation in Table 3.4. In addition, other TGG approaches exist we do not want to consider further in detail. For instance, Gausemeier *et al.* presented a domain-specific TGG scenario in [GSG+09]. They apply software engineering methodologies in the domain of mechatronic systems, more precisely in the area of autonomic vehicles called *RailCabs*. In this scenario, they are confronted with a system of coherent partial models, which have to be synchronised. Again, these models can be considered as views of the overall system. They do not suffer from correspondence detection problems since they have a well-defined context. Thus, correspondences can be declared statically in a TGG.

A substantial amount of work was done in the area of name analysis and consistency in

[20] `http://pi.informatik.uni-siegen.de/Projekte/SiLift/` (visited 10th February 2015)
[21] `http://www.fujaba.de/` (visited 10th February 2015)

MLDEs [Pfe13]. Pfeiffer proposes solutions for IDEs in which several different DSLs participate and models can refer to others without restriction. In this sense, a plethora of relations is established just by referring to the same names. Users of such an MLDE expect all references to change accordingly when the source of a named element is modified, otherwise many dependent models become inconsistent. Pfeiffer and Wąsowski discuss the design space of MLDEs and associated properties they must satisfy [PW15]. To maintain consistency between the names of those semantic references, they provide tool support for visualising dependencies and renaming them [PW11]. Since tracing between artefacts is an essential need in MLDEs, we jointly proposed a language-independent traceability approach in [PRW14]. The main setting of MLDEs is the same as for the co-refactoring scenario. But the difference to the approach of Pfeiffer is that this thesis goes beyond name analysis and renaming and considers all kinds of refactorings, especially we do not state that dependent refactorings must be of the same kind (as, e.g., a renaming results in a renaming).

As a next approach the work of Seifert in [Sei11] will be discussed. The author contributes a comprehensive approach from within the area of Round-Trip Engineering (RTE). He presents a conceptual framework for the design of RTE systems and argues that software development processes typically expose some redundancy which must be mastered to preserve consistency of artefacts in case of evolution. In this sense *unmanaged redundancy* [MBF11] must be avoided. In the RTE scenario consistency can be ensured either by reducing redundancy or by synchronising the shared information. Similar to the work presented in [BMMM08], Seifert creates model partitions of *skeletons* and *clothings* for those parts that need to be synchronised and those which do not respectively. One instantiation of the conceptual framework is realised in the concrete approach of *Backpropagation-based RTE* utilising change translation, traceability and fitness functions to synchronise models that are related by non-injective transformations. This concrete approach is used for synchronisation of the shared information. On the other hand, the author presents a role-based approach for tool integration to reduce redundancy. Tools are integrated by means of role bindings between role models. Our joint work in [RSA13] served as a proof-of-concept for the latter approach and is presented in detail in Chap. 4. In general, similar to the work discussed previously [Pfe13], RTE has another scope but co-refactoring. In RTE a connection between models usually can be considered as a *copy-of* relation. When the relation between two artefacts allows for it, then the synchronising modifications can be derived from the initiating modification. In contrast, co-refactoring does not state that a deduction relation between dependent modifications can be established. Since refactoring is also considered in [Sei11], we classify this approach into category R-CoR.

Another RTE-related approach exploiting model correspondences by means of model transformations encoded in the Atlas Transformation Language [JK06] (ATL) is presented by Xiong *et al.* in [XLH+07]. The authors argue that all relevant information regarding synchronisation of models can be derived from a model transformation relating two models to each other. This information must be sufficient enough to propagate changes not only from source to target model (forward), but also from target to source model (backward). In their approach, Xiong *et al.* exploit the ATL Virtual Machine (VM) by means of extending it to analyse the byte code, which every ATL transformation is compiled to prior execution. Such byte code is a sequence of instructions modifying the stack of the ATL VM. The authors extended the VM in the sense that *putting-back functions* are added to the compiled byte code for making synchronisation information explicit during transformation execution. The gathered information then can be used to propagate

synchronisation forward and backward. The authors proposed an interesting approach based on the exploitation of a common underlying representation of ATL transformations in form of byte-code being executed in the ATL VM. Similar to this, we published a language-independent traceability approach in [PRW14] that is applicable to all languages compiling to the Java VM. The synchronisation approach of Xiong *et al.* is implemented in the tool *SyncATL*[22] and we classify it into the category CoR.

As a last work, we discuss the approach of Di Ruscio, Lämmel and Pierantonio published in [DLP11]. The authors consider the pretty concrete context of defining a graphical modelling editor with the GMF. They propose a solution for the widely known problem of the violation of GMF editor models as a consequence of an evolved EMF Ecore model (*metamodel* in the following), which defines the concepts being available in the GMF editor models. There are three kinds of GMF models involved in the generation of a graphical editor: *graphical definition*, *tooling definition* and *mapping model*. The latter maps concepts from the metamodel to the graphical definitions and interrelates all models. Thus, the problem is that the GMF models might break if the metamodel evolves. From our own experiences, we can state that in practice this is a very ungrateful, complex and error-prone task if to be resolved manually. To tackle this problem, Di Ruscio, Lämmel and Pierantonio describe a catalogue of changes and required co-changes in order not to invalidate the involved models. Regarding the catalogue, the authors base their work on previously discussed publications in [Wac07; CDEP08; HBJ09]. If a metamodel evolves, the difference between both versions are determined yielding a difference model. This model serves as the input for specific adapters for every involved GMF model. These adapters are considered to be model transformations producing consistent versions of the particular GMF models dependent on the occurred atomic changes and the corresponding co-changes regarding their catalogue. This specific approach is implemented in the tool *GMFEvolution*.[23] We classify it into category CoR.

3.3.3. Evaluation

The evaluation of the related work and other relevant approaches is presented in Table 3.4. Therein, the evaluated approaches are grouped by means of their Abstraction Layer, whether synchronisation is applied at the same or between different layers. The entries in the column *Approach* denote a characteristic name and the most important publication of the respective approach. If no name is allocated, the authors are referred to. The column *Category* references the overall category as introduced in Fig. 3.1 and Table 3.1. The names and figures in the other column heads identify the related requirements from Sect. 3.3.1.

Table 3.4 shows that none of the discussed related approaches satisfies all of our requirements regarding co-refactoring. The main reason is that most of these approaches where proposed according to a different scope and objective. Co-evolution between an evolving metamodel and its instances (different abstraction layer) and model migration between dependent models (same abstraction layer) are frequent. What can be observed is that almost all presented approaches do not take the detection of dependencies between models and their elements into account. Most of them take these dependencies as granted in terms of trace links. However, we consider this a crucial drawback. One important characteristic of almost every approach is that initial

[22] `http://sei.pku.edu.cn/~xiongyf04/modelSynchronization.html` (visited 10th February 2015)
[23] `http://www.emfmigrate.org/gmf-evolution/` (visited 10th February 2015)

Table 3.4.: Comparison of related work regarding co-refactoring.

Abstraction Layer	Approach	Dependent Models (1)	Dependent Elements (2)	Incoming Refactoring (3)	Condition Specification (4)	Outgoing Co-Refactoring (5)	Dependent Binding (6)	Language Independence (7)	Language Specifics (8)	Interoperability (9)	Category
Different	Wachsmuth [Wac07]	−	−	+	+	+	+	+	−	+	R-CoR
	COPE [HVW11]	−	−	+	○	+	○	+	+	+	R-CoR
	In-Place HOT [MWCS11]	−	−	+	○	+	○	+	○	+	R-CoR
	Epsilon Flock [RKP+14]	−	−	○	+	○	+	+	+	+	CoR
	Vitruvius [Bur14]	+	+	○	○	○	○	+	+	+	R-CoR
Same	Refacola [vPUTS13]	○	○	+	+	+	+	+	+	○	R-CoR
	Coarse-Grained [WMV12]	○	○	+	+	+	−	○	+	○	R-CoR
	SiLift [GRK14]	○	○	+	−	+	○	+	+	+	R-CoR
	Fujaba TGG [GW09]	−	−	+	+	+	○	○	+	○	CoR
	MLDE [Pfe13]	+	+	+	+	+	○	+	○	+	R-CoR
	RTE [Sei11]	○	○	○	○	○	+	+	+	+	R-CoR
	SyncATL [XLH+07]	○	○	○	○	○	+	+	+	−	CoR
	GMFEvolution [DLP11]	+	+	+	−	+	○	−	+	○	CoR

evolution is not distinguished from resulting co-evolution. This is essential in such a scenario and is provided by our approach.

In detail, we can admit that the approaches applying synchronisation at different abstraction layers all are independent from particular involved languages — as expected. These approaches are generic in the sense that no assumptions regarding suitable DSLs are proposed. The Vitruvius approach of Burger [Bur14] and the MLDE approach of Pfeiffer [Pfe13] have to be emphasized with respect to their capabilities of capturing model and element dependencies. The former handles this important aspect by the use of a single underlying model (SUM). The SUM can be considered a *megamodel* as it was proposed by Bézivin, Jouault and Valduriez in [BJV04]. The advantage of such an approach is that all models belonging to a system are known and so are the dependencies between them. All information is made explicit. Consequently, the problem of dependency detection just does not exist in this approach since models (in the conventional sense being separated artefacts) are views on top of the SUM. The latter establishes relations between models in a MLDE by means of name analysis. Furthermore, this approach is very mature with respect to our requirements in general since it has almost the same main objective as we have. But we go a step further and consider refactorings in general and not only renaming.

The GMFEvolution approach of Di Ruscio, Lämmel and Pierantonio [DLP11] also evaluates positively regarding the dependency detection. The reason is that this problem just does not exist in this specific setting because it is very special and the involved models are unambiguously known beforehand. Furthermore, the Refacola approach of von Pilgrim *et al.* and Steimann [vPUTS13] is pretty strong again. The strength of applying refactorings and co-refactorings by checking and solving constraints in terms of WFRs of the static semantics still is sophisticated. But as already discussed in Sect. 3.3.2, it heavily depends on the creation of the cross-language constraints as a result of determined trace links. Then, we got to know another constituent of the Epsilon language family, Epsilon Flock [RKP+14]. Their model migration language seems very promising to better reflect evolution aspects instead of a generic model transformation language.

Summarising, to the best of our knowledge there are really interesting approaches regarding co-evolution in MDSD. But none of them can be used as is because either they have limitations regarding dependency detection or they are not generic enough to be suitable for co-refactoring between heterogeneous models.

3.4. Conclusion

To summarise the related work of this chapter, every discussed approach justifies their existence in the context of their particular domains, settings or use-cases. As we made clear beforehand, no comprehensive approach regarding the generic quality-aware model refactoring and co-refactoring to resolve deficiencies in models exists. Thus we had to partition our overall objective into the three presented parts: model refactoring, determination of quality-related deficiencies, and co-refactoring.

As we have seen, there is one tool family present in each of our three scenarios: the Epsilon language family. First, Epsilon EWL (R) contributes to the refactoring part, second, the Epsilon Validation (R-SD) contributes to the detection of model deficiencies, and, third, Epsilon Flock (CoR) substantially contributes to the synchronisation part of our big picture. Even though the Epsilon language family has some disadvantages, it must be honoured that they cover a broad

spectrum of model-driven engineering. Thus, we can cumulate their separate classifications to R-CoR-SD for the whole Epsilon framework, which now is the second tool in this category aside PerOpteryx of Koziolek.

On the other hand, we have an approach covering two aspects: Refacola. Their CBR-based approach for refactoring and co-refactoring is really novel and has great potential. Furthermore, EMF Refactor [Are14] and the MLDE approach [Pfe13] were evaluated quite good regarding our requirements. Both assume very similar prerequisites. They contribute mature means to our envisaged main objective to support refactoring and Co-Refactoring is it is known from IDEs.

In conclusion we can say that we got deeper insights into the different parts of the related work. This analysis justifies the intention of this thesis and in the following chapters we will contribute our approach of generic quality-aware model refactoring and co-refactoring to resolve deficiencies in models.

4
Role-Based Generic Model Refactoring

This chapter is based on our publications "Role-based Generic Model Refactoring" [RSA10] and "On the reuse and recommendation of model refactoring specifications" [RSA13]. It presents our role-based approach of generic model refactoring and compositions of refactorings.

4.1. Motivation

Refactorings can be used to improve the structure of software artefacts while preserving the semantics of the encapsulated information. Various types of refactorings have been proposed and implemented for programming languages (e.g. Java or C#). With the advent of MDSD, a wealth of modelling languages rises and the need for restructuring models in MLDEs similar to programmes has emerged. Since parts of these modelling languages are often very similar, we argue to reuse the core transformation steps of refactorings across languages as discussed in Sect. 1.2. Based on that, reusing the abstract transformation steps and the abstract participating elements becomes easy.

As an example, consider the *Extract Method* from Fig. 1.5 on page 5 for Java again. From an abstract point of view, the core refactoring steps comprise the sequence of selecting some statements, creating a new method, naming the new method, moving the selection to the new method, and adding a call to the new method at the origin of the moved selected statements. Now, consider our toy DSL for planning conferences as well. The following example has already been illustrated in Fig. 1.6 on page 6, but for the sake of better understanding we depict it here again. The abstract and textual concrete Syntax of the Conference DSL can be seen in Appendix F. This little language can be used to define different tracks, talks and speakers for a conference. For the talks only declared speakers can be referred to. Figure 4.1(a) depicts an example conference. Assuming the last two talks in Lines 7 and 8 are less interesting we could create a new track and move them into it. The result can be seen in Fig. 4.1(b). Except of the last *Extract Method* step (adding a method call) from above, very similar modifications have to be performed in the conference example: select some talks, create a new track, name the new track, and move the

```
CONFERENCE
   "International FooBar Camp"
   ("Peter Meyers")

TRACK "Interesting Stuff" :
   AT 09:00 : TALK "The future of Foo"
        PRESENTED BY "Peter Meyers"
   AT 10:00 : TALK "Foo vs. FOO" PRESENTED
        BY "Andrew Bloomfield"
   AT 11:00 : TALK "Onyl Bar-S is true"
        PRESENTED BY "Homer Simpson"

REGISTERED SPEAKERS :
   "Peter Meyers" FROM Germany,
   "Andrew Bloomfield" FROM USA,
   "Homer Simpson" FROM USA
```

(a) before

```
CONFERENCE
   "International FooBar Camp"
   ("Peter Meyers")

TRACK "Interesting Stuff" :
   AT 09:00 : TALK "The future of Foo"
        PRESENTED BY "Peter Meyers"

TRACK "Less Interesting Stuff" :
   AT 10:00 : TALK "Foo vs. FOO" PRESENTED
        BY "Andrew Bloomfield"
   AT 11:00 : TALK "Onyl Bar-S is true"
        PRESENTED BY "Homer Simpson"

REGISTERED SPEAKERS :
   "Peter Meyers" FROM Germany,
   "Andrew Bloomfield" FROM USA,
   "Homer Simpson" FROM USA
```

(b) after

Figure 4.1.: Example of *Extract Track* refactoring in the Conference DSL.

selected talks to the new track. These modifications can be considered as the refactoring *Extract Track* for the Conference DSL.

As can be observed in the example, both illustrated refactorings are applied to instances of completely different languages. On the one hand, we have the object-oriented programming language Java. On the other hand, we have a DSL for defining conferences not being object-oriented at all. Obviously, the executed steps in both refactorings are quite similar from an abstract point of view, and therefore a generic refactoring specification is required. As shown in Sect. 3.1, previous work in this field indicates that refactorings can be specified generically to foster their reuse. However, existing approaches can handle certain types of modelling languages only and solely reuse refactorings once per language. Especially the large variety of modelling languages demands for generic and reusable methods and tools. The effort to develop and maintain the growing number of DSLs can only be reduced by reusing tools across different languages. This does of course also apply to refactorings. If one wants to quickly establish refactoring support for new DSLs, a technology to reuse existing refactorings is needed [MTM07].

A closer look at the reuse of refactorings raises the question about what can be reused across multiple languages and what cannot. Related work in this area has shown that there is potential for reuse, but it is limited in one way or the other (cf. Sect. 3.1). In addition, this research indicated that some aspects cannot be captured in the reusable part of the refactoring. For example, a generic refactoring cannot make assumptions about the semantics of a language. Thus, reusing refactorings requires a combination of adapting existing generic parts to the modelling language of interest and the additional specification of language-specific information. This chapter presents a new technique for this combination and uses the following terminology: *refactoring* denotes a concrete restructuring in a particular language, *generic refactoring* indicates a reusable abstraction of similar refactorings of one or more languages, and *refactoring execution* means the application of a refactoring on a particular model.

In this chapter, a novel approach to specify generic refactorings is presented in the upcoming section. In order to prove Hypothesis 1, we argue that the formalism of *role models* (cf. Sect. 2.3) is a suitable abstraction over refactorings. As discussed in Sect. 3.1.2, the generic M3 approaches are too static regarding a common unified meta-metamodel for all target languages. Such a meta-metamodel represents concepts of target languages in an isolated manner and does not reflect the fact how humans think about objects. A more appropriate approach is to model structures between objects that indicate how these objects interact in specific contexts. While a metaclass only specifies properties of elements by static means, when considering an element in a concrete refactoring scenario, we notice that the interaction of elements can be substantially different in other contexts. This fact cannot be reflected with the static M3 approaches. It is obvious that the same element participating in one refactoring can play a completely different role in another one. As a consequence, every model refactoring defines another context for the involved elements. As a result of this preliminary consideration, we are convinced that role models are suitable means to address this problem, since they support declaration of roles which have to be played in a certain context. Assigned to generic refactoring, contexts are different refactorings and roles are the participating elements. We discuss how this resolves the limitations of previous works, as well as how specific refactorings can be defined as extensions to generic ones. This contributes to fulfil Requirements 1 and 3 from Sect. 3.1.1.

4.2. Specifying Generic Refactorings with Role Models

Previously, we have argued that there is a strong need to reuse generic refactorings, in particular in the context of DSLs. To enable such reuse, the parts of refactorings that can be generalised must be separated from the ones that are specific to a particular language. Consider, for example, the basic refactoring *Rename Element* for any concrete language. The steps needed to perform this refactoring are the same, no matter what kind of element needs to be renamed. After changing the value of a string-typed attribute, all occurrences which refer to this value must be kept consistent in the most general case. Since we have models, there is no need to update the references because in contrast to trees they contain references and do not establish implicit relations by using the same names. But, models can refer to other models, and, therefore, dangling references must be updated right after a refactoring is executed.

The concrete string-typed attribute may vary for different languages, but the general procedure is the same. Depending on constraints that apply to particular languages (e.g., whether unique names are required) some renamings may be valid, while others need to be rejected.

From the simple example, we can gain some initial insights. First, the structural part of a refactoring (i.e., the elements that are transformed) is a good candidate for reuse. Second, the semantics of a language can render concrete refactorings invalid. The example also shows that semantics—both static and dynamic—are language-specific and therefore cannot be part of a generic refactoring. We have also seen that the execution of a refactoring can be a composition of a generic transformation and specific steps that differ from language to language. We postpone the language specifics for a moment (see Sect. 9.1.5) and look at the structural and transformational aspects of refactorings first.

To reuse the structural part of generic refactorings, a model of this structure is needed. More precisely, the structure of models that can be handled by a generic refactoring must be specified.

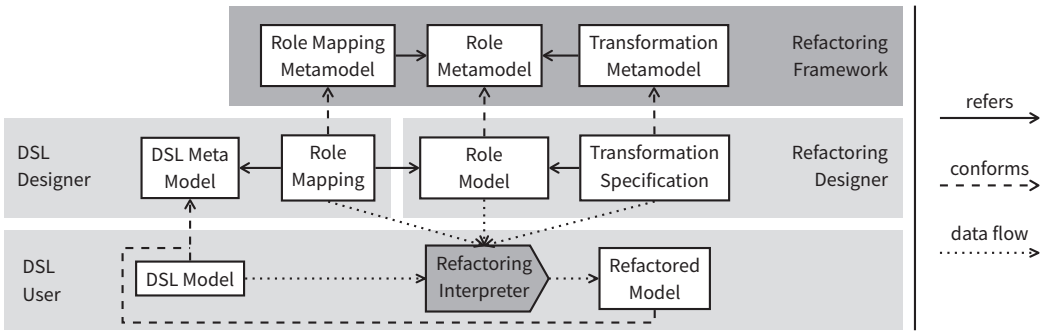


Figure 4.2.: Metamodels, relations and stakeholders.

For the *Rename Element* example, this structure basically consists of an attribute in the particular metamodel. Any model element that has such an attribute (i.e., it is an instance of the metaclass that defines the attribute) can be renamed. Other refactorings (e.g., *Extract Method* as described in [FBB+99] for Java), have more complex structural constraints, such as requiring that the elements to extract (i.e., statements) need to be contained in a container object (i.e., a method) and must be moved to a new container object (i.e., a new method). Imagine, e.g., a refactoring *Extract CompositeState* for UML state machines in comparison to *Extract Method* for Java: some states (i.e., elements to extract) have to be selected which are intended to be moved into a new composite state (i.e., a new container). As one can see, the abstract transformation steps in this example are the same if we only consider the structural abstraction as described above. To model such structural properties—one may also consider these as structural requirements—we have chosen to use *role models* as they were proposed in [RG98; RWL96], but in a slightly adjusted manner.

Roles encapsulate the behaviour of a model element with regard to a context. Roles appear in pairs or sets, so-called collaborations. In the context of this work, roles can be considered as the types of elements that are subject to a refactoring. The role collaborations model the references between the elements required to execute a refactoring. More details about this can be found in Sect. 4.2.1.

To map the abstract structural definition of the input required by a generic refactoring to concrete languages, a relation between the structural definition (i.e., the role model) and the metamodel of the language must be established. This mapping defines which elements of the metamodel are valid substitutions for elements in the role model. We call this relation a *role mapping*. We discuss such mappings in Sect. 4.2.2. We distinguish between the static mapping of roles (beforehand) and the dynamic binding of roles (while executing a concrete refactoring).

To reuse the transformational part of generic refactorings a third model is needed—a transformation specification. This model defines the concrete steps needed to perform the generic refactoring. In the example of renaming an element, this model must contain instructions to change the value of an attribute. Of course, the transformation needs to change the value of the attribute specified by the role model (i.e., it refers to the role model). In Sect. 4.2.3, details about the transformation operators and the interpretation thereof will be found.

The models mentioned so far are depicted in upper right part of Fig. 4.2. Concrete models are related to their metamodel by *instance-of* links. Two kinds of models (i.e., role models and

transformation specifications) are provided by the refactoring designer. She creates generic refactorings, which consist of pairs of such models. DSL designers can then establish correspondences between the generic refactorings to their languages by creating role mappings. This enables DSL users to apply refactorings to concrete models.

4.2.1. Specifying Structural Constraints using Role Models

To explain role models in more detail, we want to consider *Extract Method* for Java (cf. Fig. 1.5 on page 5) and *Extract Track* for our Conference DSL (cf. Fig. 4.1 on page 52) again. As already shown in Sect. 4.1, we are convinced that this kind of refactorings (i.e., moving a set of elements to a new container and referring to the new container at the original position) is useful for other languages, too, and we want to derive a generic refactoring—*Extract X with Reference Class*.

To capture the structural properties of our generic refactoring we use a role model. For the generic refactoring *Extract X with Reference Class* (cf. Fig. 4.3), we consider elements that are extracted, their container, a new container the elements are moved to, and a moved reference referring to the new container. Elements participating in a generic refactoring form groups, where all elements within one group are handled equally. In other words, elements *play* a certain role with respect to a generic refactoring.

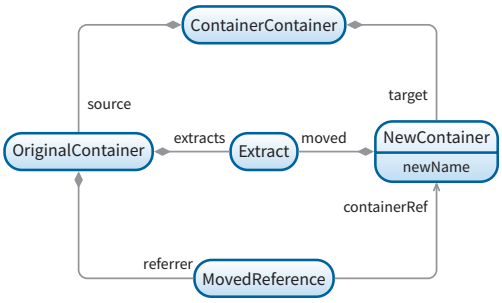
The elements that are extracted (e.g., structural features) play the `Extract` role (cf. Fig. 4.3). The object that contains the extractees plays the `OriginalContainer` role. The object to which the extractees are moved, plays the `NewContainer` role. And the element that will reference the new container afterwards plays the role `MovedReference`. In other words, roles abstract from the concrete metaclasses that are allowed as participants for a generic refactoring. Instead of referring to concrete metaclasses, the structure that can be transformed is specified by roles. Later on, when a refactoring is enabled for a concrete language, role mappings map roles to concrete metaclasses.

Between the roles that form the nodes in our structural description certain relations hold (the collaborations). For example, the `OriginalContainer` must hold a containment reference to the `Extract` elements. Also, a reference between the original container and the `MovedReference` is needed to connect the new container to the moved reference. We use collaborations to model such structural dependencies between roles.

The complete role model for the generic refactoring *Extract X with Reference Class* is depicted in Fig. 4.3 (a) in a graphical notation. Fig. 4.3 (b) shows a textual notation of the same role model. One can identify the roles mentioned above (shown as rounded boxes) as well as their collaborations, which are depicted by links between the boxes. Besides the roles mentioned above, there is a role *ContainerContainer* which models the fact that the new and the original container must be contained in a third element. For the *Extract Method* refactoring, this role is played by the class `Class`, which is specified in the role mapping (see Sect. 4.2.2).

The example gives a first impression how role models specify the structural constraints that must be met by a language's metamodel to qualify for the generic *Extract X with Reference Class* refactoring. All other concepts that can be used in role models are defined by the role metamodel shown in Fig. 4.4.

The concepts `Role` and `Collaboration` are contained in a `RoleModel`. Roles may be annotated by several `RoleModifiers`. An `optional` role is not needed to be mapped to a specific metaclass (e.g., if a DSL's metamodel does not contain such a metaclass in the desired context).



(a) Graphical representation.

```
RoleModel ExtractXwithReferenceClass {
   ROLE OriginalContainer;
   input ROLE Extract;
   ROLE NewContainer (newName);
   ROLE ContainerContainer;
   optional ROLE MovedReference;

   OriginalContainer [1..1]
      <>- Extract extracts [0..*];
   OriginalContainer [1..1]
      <>- MovedReference referrer [
          1..1];
   NewContainer [1..1]
      <>- Extract moved [0..*];
   ContainerContainer [1..1]
      <>- NewContainer target [1..1];
   ContainerContainer [1..1]
      <>- OriginalContainer source [
          1..1];
   MovedReference [1..1]
      -- NewContainer containerRef [
          1..1];
}
```

(b) Textual representation.

Figure 4.3.: Different representations of generic *Extract X with Reference Class* refactoring.

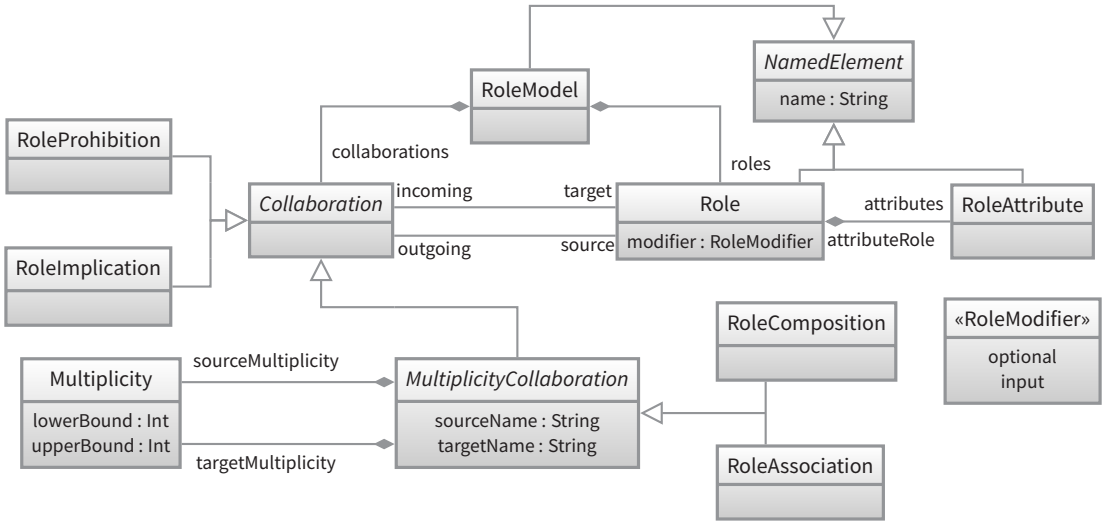


Figure 4.4.: Role metamodel.

Roles which must serve as input of a generic refactoring are marked with the `input` modifier. In the example *Extract X with Reference Class* the `Extract` must be provided as such. Furthermore, a `Role` can contain `RoleAttributes` to express that it can own observable properties (e.g., a name) which may change during the role's life cycle.

Collaborations connect two `Roles`, a `source` and a `target` `Role`. They are further distinguished into different types. A `RoleImplication` constraint can be used to express that one role must also play another role. A `RoleProhibition` constraint states that two roles are mutually exclusive—an element playing one role must not play the other role. The other types are characterised by `Multiplicity`s, expressing that one role can collaborate with a specific number of elements playing the related role. To distinguish between containment and non-containment references, we use `RoleCompositions` and `RoleAssociations` respectively.

Besides the role model of *Extract X with Reference Class* depicted in Fig. 4.3 further role models have been developed. The whole list can be seen in the Appendix A. A comparison of our role metamodel to the role feature model of [KLG+14] can be seen in Appendix B.

The next step in the development of a generic refactoring is writing a transformation specification. Such specifications do only refer to the role model of the generic refactoring, not to a concrete metamodel. However, first we have a look at role mappings, which DSL designers can use to relate role models with concrete languages. This is needed to understand the actual execution of generic refactorings, as this can only be performed in the context of a mapping.

4.2.2. Mapping Roles to Language Concepts Using Role Mappings

To control the structures intended for refactoring execution, the role model needs to be mapped to the target metamodel. According to [RWL96], applying a role to an entity in a certain domain is called *binding*. In the common sense, this term means that the role is bound dynamically at runtime. Since we propose a solution where role models are applied statically at design time, we consider this binding a *mapping* in the following. On the other hand, our approach also comprises a dynamic aspect, namely the refactoring execution at runtime. In that context, the statically mapped roles then are bound to dynamically determined model elements which we then consider a *role binding* in the common sense. But this is explained in the next section (cf. Sect. 4.2.3). Here we discuss the static mapping of roles to target metamodels.

Such a mapping between role model and target metamodel is defined by the DSL designer, which can be seen in the middle level of Fig. 4.2. Based on this mapping, refactorings can only be applied to those structures conforming to the metaclasses the roles were mapped to.

The whole metamodel to which role mappings conform is depicted in Fig. 4.5. Here, each `RoleMappingModel` refers to an `EPackage`[1] containing the `targetMetamodel`. The referenced metamodel is the language which the role mapping is defined for. Thus a concrete refactoring represented by such a mapping can be applied for instances of the target metamodel. Furthermore, a `RoleMappingModel` can refer to `importedMetamodels`. These are metamodels which might be referenced in the target metamodel, and it might be the case that those metaclasses become target of a role mapping. Thus, the related metamodels must be known here and can be referenced. To define concrete mappings, a `RoleMappingModel` owns several `RoleMappings` inheriting from `Refactoring`, thus it has a `name` indicating the identifier of the concrete refactoring,

[1] We use EMF Ecore as MOF-conform modelling technology.

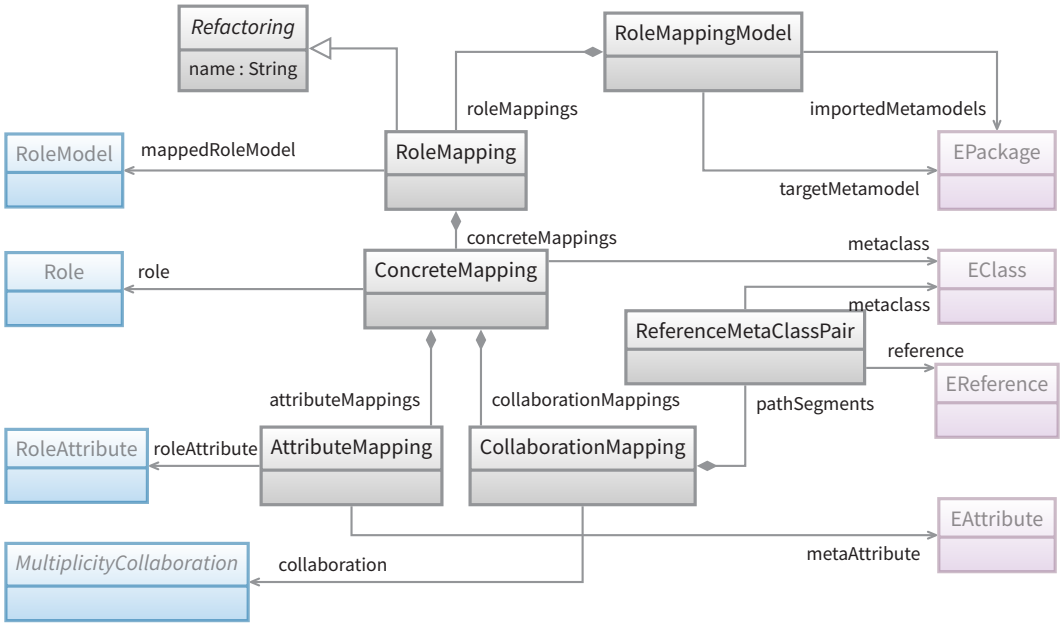


Figure 4.5.: Role Mapping metamodel. Different colours denote distinct metamodels. Since those metaclasses are only referenced here they appear slightly transparent to make clear they are defined in other metamodels.

such as *Extract Method* for Java or *Extract Track* for our Conference DSL. A `RoleMapping` references the role model of the generic refactoring. By this relation, we ensure that only roles defined in the `mappedRoleModel` can be mapped. The `name` of a `RoleMapping` expresses the name of a concrete refactoring. By means of corresponding a metaclass and a role, a `RoleMapping` owns several `concreteMappings`. Each `ConcreteMapping` specifies which `role` of the mapped role model is played by which `metaclass` from the target metamodel. Since roles can possess `RoleAttributes`, a `ConcreteMapping` can have `AttributeMappings` relating particular `RoleAttributes` to `EAttributes` owned by an `EClass`.

If roles collaborate with other roles, a `ConcreteMapping` must specify `CollaborationMappings` being used for each role `collaboration` to define which metaclass relations it corresponds to. For this purpose, we want to introduce some flexibility here. If we only allow for mapping a `collaboration` to an `EReference` from a target metamodel's metaclass, then the target metaclass must contain exactly the same structure as defined in the role model. This means that small differences in a target metamodel's design may result in the fact that a new role model is needed. Therefore, we want to loosen this restriction by introducing the possibility to map a `collaboration` not only to an `EReference` but a path of `EReferences`. For this reason, a `CollaborationMapping` can define `pathSegments` each typed by a `ReferenceMetaClassPair`. Every pair points to a concrete `EReference` (from the target metamodel), which must lead to the specified `EClass`. By defining multiple `ReferenceMetaClassPairs`, it is possible to constitute a path from one `EClass` to another. This enables the control of the structures in a more flexible way. With this mapping the possibilities of annotating a metamodel with a role model are completed—all structural features can be mapped.

```
ROLEMODELMAPPING FOR
   <http://www.emftext.org/java>

"Extract Method"
   maps <ExtractXwithReferenceClass> {

   Extract := java.statements.Statement;

   OriginalContainer := java.members.
       ClassMethod{
         extracts := statements;
         referrer := statements :
             ExpressionStatement ->
             expression : MethodCall;
   };
   NewContainer := java.members.
       ClassMethod (newName -> name){
         moved := statements;
   };
   ContainerContainer := java.classifiers.
       Class{
      source := members;
      target := members;
   };
   MovedReference := java.references.
       MethodCall{
      containerRef := target;
   };
}
```

(a) *Extract Method.*

```
ROLEMODELMAPPING FOR
   <http://www.emftext.org/language/
       conference>

"Extract Track"
   maps <ExtractXwithReferenceClass> {

   Extract := Slot;

   OriginalContainer := Track{
      extracts := slots;
   };
   NewContainer := Track (newName ->
       name){
      moved := slots;
   };
   ContainerContainer := Conference{
      source := elements:Track;
      target := elements:Track;
   };
}
```

(b) *Extract Track.*

```
ROLEMODELMAPPING FOR <http://www.eclipse.org/uml2/3.0.0/UML>

"Extract CompositeState" maps <ExtractXwithReferenceClass> {

   Extract := State;

   OriginalContainer := Region {
      extracts := subvertex:State;
   };
   NewContainer := State (newName -> name) {
      moved := region -> subvertex:State;
   };
   ContainerContainer := Region {
      target := subvertex:State;
   };
}
```

(c) *Extract CompositeState.*

Figure 4.6.: Role mappings to different DSLs for the generic *Extract X with Reference Class* refactoring.

The following example illustrates the explained theory. We come back to the refactorings *Extract Method* for Java (cf. Fig. 1.5 on page 5) and *Extract Track* for our Conference DSL (cf. Fig. 4.1 on page 52). The role mappings for these examples can be seen in Fig. 4.6. On the left hand side in (a), the mapping for *Extract Method* is depicted, whereas the mapping for *Extract Track* can be seen in (b). Both are instantiated from the generic *Extract X with Reference Class* refactoring. The mapping is written in our DSL for role mappings, which is presented in more detail in Sect. 9. This example illustrates the fact that a role collaboration can be mapped to a path of references in the target metamodel. Have a look at Line 11 in (a). One can see that the collaboration `referrer` is mapped to the following path originating from the metaclass `java.members.ClassMethod`, starting from the reference called `statements` reaching an abstract `Statement` class. Thus, a concrete subclass has to be chosen: `ExpressionStatement`. From this metaclass, the mapping continues with the reference called `expression` ending at the concrete metaclass `MethodCall` (being a subclass of the abstract `Expression` metaclass). In turn, `MethodCall` is the one playing the role `MovedReference`.

A third example is depicted in Fig. 4.6 (c) and can be used in UML state machines. This role mapping represents the refactoring *Extract CompositeState* and moves some selected states into a newly created composite state. It is intended to be applied in case a subset of a state machine should be made reusable.

These three examples show that our approach does not make any assumptions about the kind of target languages. We defined role mappings for both Java as a General Purpose Language (GPL), graphical modelling languages (UML), and DSLs (Conference language). Deeper insights about the quantity of specified role mappings and reuse of generic refactorings are presented in Sect. 10.1. The complete list of all defined role mappings can be found in Appendix C.

Our methodology of role mapping is not as rigid as the other M3 approaches in Sect. 3.1.2. One and the same role model can be mapped several times onto different structures in the same metamodel, which raises the flexibility significantly. In contrast to the work of Hannemann, Murphy and Kiczales in [HMK05] (also discussed in the M2 approaches of Sect. 3.1.2), who proposed an approach for role-based refactoring of cross-cutting aspects in Java, our approach resides an MOF layer M3 instead of M2.

Summarising, what a DSL designer only has to do to enable a concrete refactoring for her DSL users is to provide such a role mapping. The benefit is the reduced manual effort. Until now, we have an abstraction over the structure of the participating elements of a refactoring in terms of role models, and a role mapping for instantiating a role model resulting in a concrete one. In the next section, we will present our solution for specifying the particular steps of a generic refactoring.

4.2.3. Specifying Language-Independent Transformations using Refactoring Specifications

Returning to the viewpoint of the refactoring designer, the next step to consider is the particular specification of the actual generic refactoring's transformation. To abstract the transformation from concrete languages, it must refer to the corresponding role model only. Since we introduced the structural abstraction in terms of a role model a dedicated transformation language is required to take this into account. This fact is also attended by [RKP+14] since Rose *et al.* argue that an evolution-specific model transformation language can provide specific evolution operators

```
1  REFACTORING FOR <ExtractXwithReferenceClass>
2
3  STEPS {
4     object containerContainerObject := ContainerContainer from uptree(INPUT);
5     object origContainerObject := OriginalContainer from path(INPUT);
6     index extractsIndex := first(INPUT);
7
8     create new nc:NewContainer in containerContainerObject.target;
9     assign nc.newName;
10    move origContainerObject.extracts to nc;
11    create new mr:MovedReference in origContainerObject.referer at extractsIndex;
12    set use of nc in mr.containerRef;
13 }
```

Listing 4.1: Refactoring specification for *Extract X with Reference Class*.

making technical details transparent. This fact is supported by our evaluation (cf. Table 3.4 on page 47) of the model migration language Epsilon Flock [RKP+14].

For this concern, we propose a metamodel—the *Refactoring Specification (RefSpec)*—and a textual syntax for specifying generic refactoring steps based on roles. The RefSpec metamodel is extensible in terms of subclassing, for which many potential extension candidates as abstract metaclasses are available. This section presents only the conceptual aspect of the specification language, whereas the concrete textual syntax specification is presented in Chap. 9. Nevertheless we start with a concrete example RefSpec which serves for understanding the RefSpec concepts better.

Example

Listing 4.1 shows the refactoring specification for the role model of the generic *Extract X with Reference Class* refactoring (cf. Fig. 4.3). This specification is given in textual notation of our RefSpec DSL. Please consult the corresponding role model in Fig. 4.3 on page 56 to understand the used roles and collaborations in this example.

Six basic steps are sufficient to execute the generic *Extract X with Reference Class* refactoring. First, two object definitions (Lines 4 and 5) bind concrete model elements from the input (i.e., the selected elements) to symbolic names. Furthermore, the actual position of the selected elements from the input is needed and bound to a symbolic name (Line 6). Subsequently, five of the core commands can be found. The `create` operator constructs a new element playing the first given role as a child of the element playing the second given role (Line 8). In the next step, `assign` is used to pass a value to the attribute which was bound to the given role attribute (Line 9). The `move` command then relocates the elements playing the role of the collaboration with the first given role and the second given role becomes the new parent (Line 10). Afterwards, again the `create` operator is used for instantiating an element playing the role `MovedReference` (Line 11). It is created `at` the same position as the originally selected input elements where located. Finally, the `set` command arranges that the object playing the first given role is referenced in the bound collaboration of the object playing the second given role (Line 12).

This RefSpec is the transformation specification for the generic *Extract X with Reference Class*

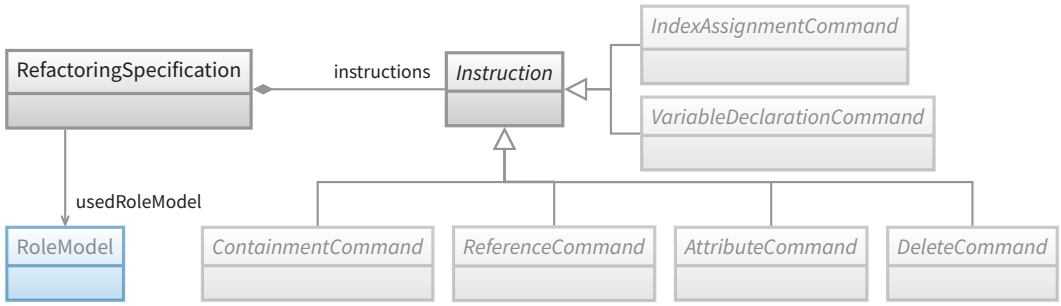


Figure 4.7.: Refactoring Specification metamodel.

refactoring. In the following, we explain the conceptual background of the RefSpec language.

Concept

As already mentioned, we designed a transformation language dedicated to specify generic refactorings based on the roles defined in the corresponding role model. The metamodel of this language is quite large and, for the sake of clarity, we divide it into several parts. Some metaclasses occur slightly transparent again. This means that their conceptual origin is located in another part of the whole metamodel.[2] Metaclasses having a different colour are originally specified in another metamodel and can also be navigated.

In Fig. 4.7, the basic structure of a `RefactoringSpecification` is illustrated. A `RoleModel` is referenced establishing the context of a generic refactoring. Only those structures defined in the referred role model can be used. A `RefactoringSpecification` contains `instructions` to be performed. This metaclass is abstract and has several subclasses. In the lower part of the figure, you can see the abstract metaclasses denoting commands to operate on structural concerns of a model: `ContainmentCommand`, `ReferenceCommand` and `AttributeCommand`. Since deletions can be applied to almost every structure, the `DeleteCommand` is located at this layer of the whole metamodel. Furthermore, commands for element binding and indexing are required. Therefore, `VariableDeclarationCommands` and `IndexAssignmentCommands` are used. These constituents of the core metamodel reflect the different parts being explained in the following.

Obviously, the particular transformation must be applied on the real model elements and not on the defined roles. Therefore, these elements are bound to the roles at runtime. This fact is reflected by the definition of variables for role bindings (Fig. 4.8). A `VariableDeclarationCommand` creates exactly one[3] `Variable` uniquely identified by its `name`. The concrete assignment of a variable is realised by one of the two metaclasses inheriting from `ObjectAssignmentCommand`. First, a variable can be assigned to the elements being bound to the target of a `Multiplicity-Collaboration` by using a `CollaborationReference`. Second, a variable can be assigned to a `RoleReference` being able to determine elements playing the given `Role` in dependence on already known model elements based on a `ObjectReference`. On the one hand, this can be the input elements reflected by the `INPUT` constant from `ConstantsReference`. On the other hand, this can be elements already assigned to another variable via a `VariableReference`. The set of

[2] These metaclasses can be considered as references and can be navigated to in the digital version of this thesis.
[3] For the sake of clarity, cardinalities are not given in the figures but denoted in the explanation.

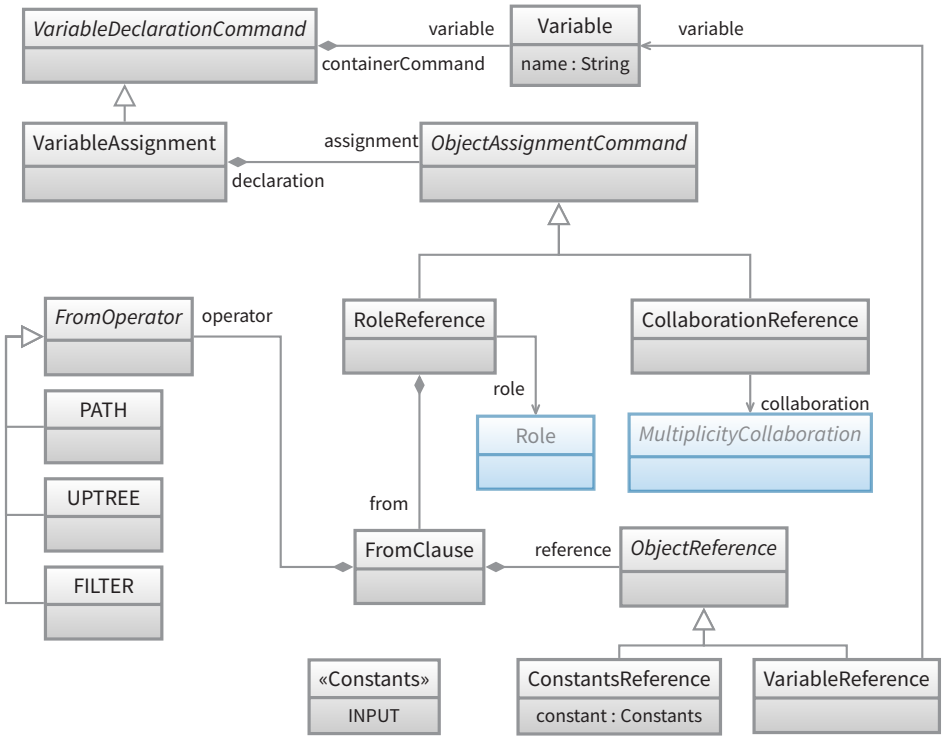


Figure 4.8.: RefSpec metamodel: Variable Declaration.

already assigned elements is the starting point for determining the desired elements playing the given `role` of a `RoleReference`. We provide the following three `FromOperators` to achieve this. They can be understood as convenience operators filtering the given elements by applying graph traversal through the model.

1. `PATH`: In contrast to a `CollaborationReference`, this operator can be used to determine elements without referring to the actual outgoing collaborations of the roles they play. It navigates from the role, played by the given model elements, along its outgoing collaborations and collects all elements playing the `role` referenced in the `RoleReference`.

2. `UPTREE`: Determines the path from each of the given elements up the tree to the root model element in which all are contained. Then the intersection of these paths is produced resulting in the path all the others have in common. This operator then returns the first element of the intersected path (starting from the leaf) playing the given `role` of the `RoleReference`.

3. `FILTER`: This operator returns those model elements from the given set playing the given `role`.

To support the possibility to specify a certain position, e.g. when moving elements, we provide an `IndexAssignmentCommand`. Similar to the `VariableDeclarationCommand`, this command creates an `IndexVariable` uniquely identified by its `name`. With the use of `ConcreteIndex`, a

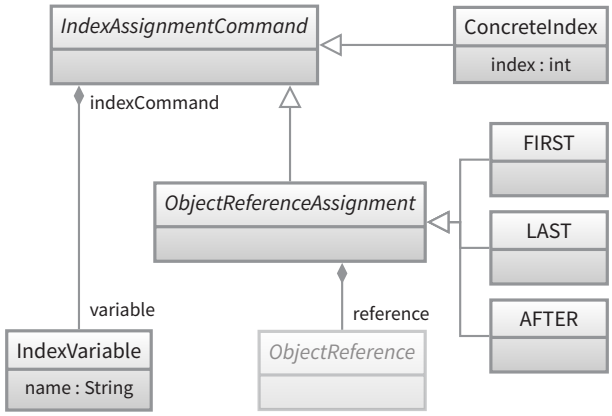


Figure 4.9.: RefSpec metamodel: Index Assignment.

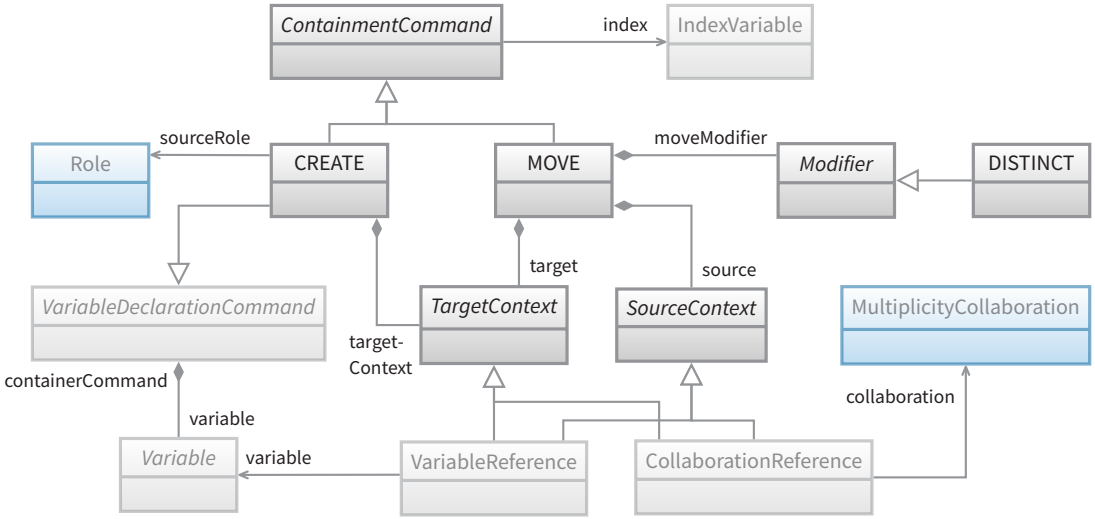
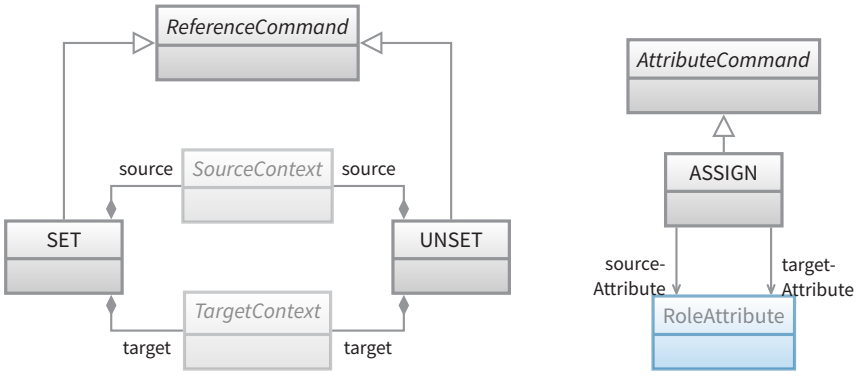


Figure 4.10.: RefSpec metamodel: Create and Move Elements.

refactoring designer is enabled to use an integer value to provide an explicit `index`. In contrast, the three metaclasses `FIRST`, `LAST` and `AFTER` refer to an already assigned set of elements (cf. `ObjectReference`) and return the concrete number of the first element in this set, the last element in this set, or the increment of the last element respectively.

Creating and moving model elements are essential operations in model transformation in general and in refactoring especially. Therefore, the part of our `RefactoringSpecification` metamodel depicted in Fig. 4.10 can be used. The abstract `ContainmentCommand` metaclass can refer to an `IndexVariable` to specify a certain index which the certain elements can be created at are moved to. To `CREATE` an element, the `Role` the element must be a player of has to be given. In addition, a `TargetContext` is needed to specify the parent of the element to be created. Here we have two possibilities. First, an already declared `Variable` can be referenced. Second, a `CollaborationReference` can be chosen to refer to a `collaboration` of a certain role. The metaclasses of the element playing the `sourceRole` must be compatible



(a) Change References (b) Change Attributes

Figure 4.11.: RefSpec metamodel: Changing Elements.

with the metaclass playing the role of the target of the given `MultiplicityCollaboration`. Furthermore `CREATE` is a `VariableDeclarationCommand` by itself. Thus a new `Variable` is created when this command is used. To move model elements around the `MOVE` command can be used. Here the `SourceContext` must be provided representing the particular elements intended to be moved. Furthermore, the mandatory `TargetContext` serves as the target of the move. Both `VariableReference` and `CollaborationReference` inherit from `TargetContext` and `SourceContext`. Since these metaclasses have already been explained, it is clear how the according model elements are resolved. Beyond that, the `modifier` `DISTINCT` can optionally be given for a `MOVE` command. If that is the case, the moved elements are compared and duplicates are not added to the new target. As an example for this, you can consider a *Pull Up Attributes* refactoring for the UML. In a class diagram several classes are contained all having a `name` attribute. These attributes are unique in their respective classes but when they should be pulled up into their common superclass they need not be unique anymore. In such a case, `DISTINCT` would be used to only result in one attribute being pulled up.

To change model elements, we distinguish between changing references and changing attributes. These two parts of the whole RefSpec metamodel can be seen in Fig. 4.11. In (a) one can see that we provide `ReferenceCommand` for `SET`ting and `UNSET`ting references. Both of them use `SourceContext` as source and `TargetContext` as target for their modification. In (b), the part of the metamodel for changing attributes is depicted. `ASSIGN` can optionally refer to a `sourceAttribute` and refers to a mandatory `targetAttribute`. If a source attribute is given, then its value is copied to the target attribute. Otherwise, a value is requested from the user who applies the refactoring. The types of source and target must be compatible.

The last part of our whole metamodel covers the removal of model elements, which is depicted in Fig. 4.12. To remove elements, the `REMOVE` command must be used. The `removal` can be specified by instances of one of these metaclasses: `ObjectReference` or `RoleRemoval`. The former has already been explained. With the latter, a `Role` can be designated which removes the elements playing this role from their container. Furthermore, a `RemoveModifier` can be given. On the one hand, `EMPTY` signifies to only remove the given elements, if they do not have any children. On the other hand, `UNUSED` can be used in case the elements are only to be removed in case they are not referenced by other elements.

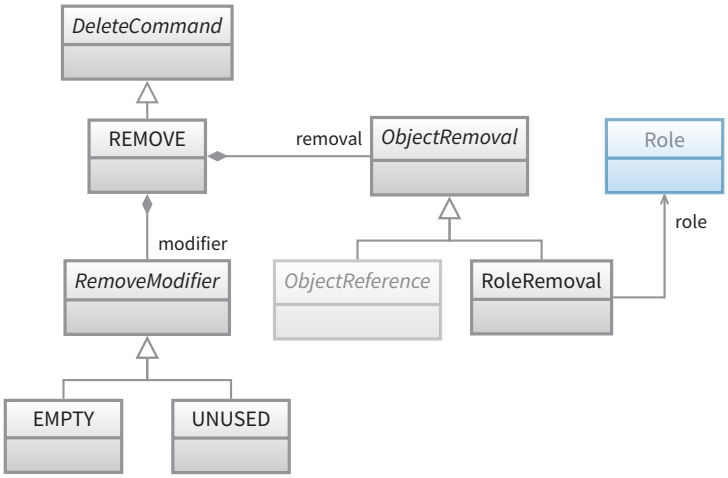


Figure 4.12.: RefSpec metamodel: Remove Elements.

This closes the presentation of our concept for the refactoring specification. The upcoming section explains how refactorings are executed.

Execution by Interpretation

Until this point we have learned three essential aspects of our generic model refactoring approach. First, a refactoring designer defines the structural requirements of the participating elements of a refactoring by means of role models (cf. Sect. 4.2.1). Second, a refactoring designer specifies the semantics of a generic refactoring by means of a role-based refactoring specification (presented in this section). Third, a DSL designer instantiates a generic refactoring to enable it as a concrete one for a particular target metamodel by means of a role mapping (cf.Sect. 4.2.2).

What is left, is the actual execution of a generic refactoring in the context of a certain model conforming to a metamodel which a role mapping was specified for. In this thesis, we decided to use an interpretation-based approach in contrast to compilation (code generation). The main advantage justifying this decision is the shorter feedback loop. A DSL designer knows her language's metamodel best and can adjust a role mapping easily. Thus, changes are taken into account directly by an interpreter when the refactoring is applied for new. Neither code generation nor deployment of the generated refactoring to the MLDE is needed. The refactoring can be executed instantly.

The input for the interpretation comprises four artefacts: role model, refactoring specification, role mapping and the set of selected model elements, which the refactoring should be applied to. During interpretation, the referenced roles and collaborations are resolved using the role mapping. In this sense, when the roles are resolved to concrete model elements at runtime, they are *bound* dynamically. Thus, as already explained, roles in our approach run through two phases: 1) they are mapped statically to the target metamodel at design time, 2) they are bound dynamically to concrete model elements while interpretation. The particular role binding then can serve as input for other MLDE tools, to preview the result of a refactoring, or to contribute to the refactoring history, or just to roll it back. We will give more insights about our implementation in Chap. 9.

As discussed by Opdyke in [Opd92],pre- and post-conditions must be met before and after refactoring execution. Otherwise, the refactoring execution must be rolled back. In our approach this fact has not been reflected so far. Pre- and post-conditions are highly domain-specific and cannot be specified generically [Are14]. Furthermore different possibilities to define pre- and post-conditions are conceivable and therefore we postpone our solution regarding this to the implementation in Sect. 9.1.6. It is important to note at this point that we provide a solution enabling the evaluation of pre- and post-conditions by means of OCL constraints related to particular role mappings.

In summary, one can say that the interpretation of generic transformation specifications in the context of role mappings, eventually bridges the gap between generic refactorings and their actual execution. It allows for defining execution steps solely based on roles and collaborations between them (i.e., independent of a concrete language metamodel), but still enables the execution of refactorings for concrete models.

Classification into the Model Transformation Feature Tree

We want to classify our concept for specifying refactoring transformations into the feature tree of model transformations published by Czarnecki and Helsen in [CH06]. Our approach covers the following most important features amongst others: it uses *generic parameterisation* for making transformation rules reusable, it uses two *intermediate structures* not being part of the transformed models (role model and role mapping model), it is a *structure-driven* model-to-model approach, and it has an *in-place source-target relationship*. In addition to the classification, our work is the first role-based approach, which results in a new feature for the classification.

Since the transformation is specified on top of a role model and not on the target metamodel itself, the feature *domain language* should have an xor-subgroup containing the following features: *fixed* and *adaptable*. The former conforms to the explanation from Czarnecki and Helsen where the domain language (being target of a transformation) is specified at design time. Whereas the latter means that the domain language, which the transformation operates on, is different from the domain of the transformed model and, thus, is adapted. In our case the transformation operates on a role model and is adapted at runtime to the metamodel of the refactored model. In this sense, the target domain is not determined until just before a generic refactoring is instantiated and executed.

4.2.4. Composition of Refactorings

As a next important aspect in model refactoring we want to provide an approach for the composition of refactorings. Composing refactorings to more complex ones has already been discussed for a long time [Rob99; VD06; MTM07; MRG09; Are14]. There is no doubt about composing refactorings, since applying several refactorings one after another yields the same disadvantages as executing the core steps of a single refactoring by hand: One has to provide information, required in a subsequent refactoring, although they might be available already from a preceding refactoring.

Let us consider the following example. Fowler *et al.* presented a discussion about the model deficiency *Feature Envy* in [FBB+99]. *Feature Envy* expresses the situation when a method interacts with the features of another class more frequently than with the features of its own

```
1 void printOwing() {
2    printBanner();
3
4    //print details
5    DetailsPrinter.print("name:␣" +
          _name);
6    DetailsPrinter.print("amount␣"
         + getOutstanding());
7 }
```

(a) before

```
1 void printOwing() {
2    printBanner();
3    printDetails();
4 }
5
6 void printDetails() {
7    DetailsPrinter.print("name:␣" +
          _name);
8    DetailsPrinter.print("amount␣"
         + getOutstanding());
9 }
```

(b) after

Figure 4.13.: Adjusted *Extract Method* refactoring example in Java.

class. Such interactions might be access to public fields or just the invocation of methods. In this sense, the method is *envious* of the other class it is not contained in. Fowler *et al.* propose to apply the *Move Method* on the envious method. Obviously, this is a pretty straightforward restructuring. Here, we extend this example in the sense that the statements within a method could be analysed more fine-grained in order to detect only groups of statements being envious. For a better understanding, we slightly adjusted the code snippet from Fig. 1.5 on page 5 in Fig. 4.13.

The printing is now carried out by the class `DetailsPrinter`. Assuming that `printBanner()` is not an envious method, the statements in Lines 5 and 6 are, because they interact with the `DetailsPrinter` class more often than with their own class. This means that we should not move the whole `printOwing()` method, but prefer to extract the envious statements to a new method and move the new (envious) method afterwards. Thus, this can be considered as a composite refactoring of *Extract Method* and *Move Method*.

The aforementioned problem of the manual provision of present information appears in this example in the following sense. First, the two statements being extracted have to be selected and a name for the new envious method must be chosen. For the second refactoring (*Move Method*) again the method intended to be moved has to be selected by the user although we already know the method from the preceding refactoring. As a consequence, composing both refactorings to *Extract and Move Method* would result in less human interaction just to provide already existing information, and further enable the reuse of the composite refactoring as a new dedicated refactoring operator.

Our formalism of `RoleMapping` helps us to provide a new approach of composite refactorings. Currently, a concrete refactoring is specified by a role mapping, which maps roles to metaclasses. At runtime these roles are bound to concrete model elements. This holds for every concrete refactoring intended to be composed. Thus, an approach for composing concrete refactorings requires means to specify which bound model elements of a preceding refactoring should be used to bind roles in a succeeding refactoring. Our concept of *roles* meets this requirement perfectly. Therefore, we provide another metamodel for composing refactorings in terms of role mappings. It is depicted in Fig. 4.14.

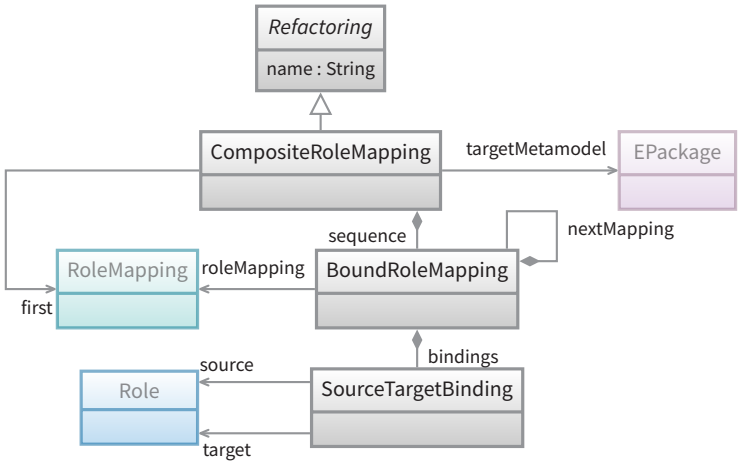


Figure 4.14.: Composite Role Mapping metamodel.

```
ROLEMODELMAPPING FOR <http://www.emftext.org/java>

"Move Method" maps <MoveX> {
   SourceContainer := java.classifiers.Class{
      sourceContainment := members;
   };
   TargetContainer := java.classifiers.Class{
      targetContainment := members;
   };
   Movee := java.members.ClassMethod;
}
```

Listing 4.2: *Move Method* role mapping.

For composing existing refactorings, the metaclass `CompositeRoleMapping` is used. Similarly as `RoleMapping`, it inherits from `Refactoring` and thus has a `name`, such as *Extract and Move Method* for Java. A `CompositeRoleMapping` references the `targetMetamodel` which the composite refactoring can be applied in. The target metamodels of the composed refactorings must match this `targetMetamodel` in order to assure that the composed refactoring is applicable in the same language as the separate ones. Furthermore, a `first` role mapping is referenced expressing the first refactoring in the `sequence`. Each subsequent refactoring is referred by an instance of `BoundRoleMapping`, which not only references the according `roleMapping`, but also establishes a `binding` by means of the `SourceTargetBinding` metaclass. Instances of this metaclass map a `source` role from the preceding refactoring to a `target` role in the succeeding refactoring. This approach benefits from the already available role mappings, since at this point it can be statically checked if the metaclasses, to which the mapped roles are already related, are compatible or not. Thus, only valid composite refactorings can be created by this approach. To complete the metamodel, every `BoundRoleMapping` instance can have a `nextMapping`, for which new `bindings` must be specified.

To return to our example, have a look at Fig. 4.6 (a) on page 59 again, with the Java *Extract*

```
1 COMPOSITE REFACTORING "Extract and Move Method"
2 FOR <http://www.emftext.org/java>
3
4 <Extract Method> -> <Move Method> {
5    <ExtractXwithReferenceClass>.NewContainer = <MoveX>.Movee;
6 }
```

Listing 4.3: Composite refactoring *Extract and Move Method*.

Method role mapping. The Listing 4.2 illustrates the *Move Method* role mapping. The role mapping of *Move X* can be found in Fig. A.2 (a). What can be seen in this example is that the `NewContainer` from *Extract Method* must be passed to `Movee` in *Move Method*. From a compatibility point of view this is a valid mapping, since both roles are mapped to the Java metaclass `ClassMethod`. The according refactoring composition is depicted in Listing 4.3 in textual notation, which is presented in more detail in Chap. 9.

As can be seen in Line 5 of Listing 4.3, the according roles are mapped to compose the two refactorings. The composite refactoring now can move the envious method to the envied class `DetailsPrinter`. When such a composite refactoring is to be executed, the interpreter uses the specified binding and passes it to the subsequent refactoring. In this way, no additional user interaction is required.

4.3. Preserving Semantics

Refactorings are required to preserve the behaviour of the programme, or in our case the model, that is subject to a refactoring execution. The behaviour of any programme (or model) is defined by its static and dynamic semantics. This immediately implies that preserving behaviour requires a formal specification of this semantics. Without formalisation, no guarantees can be given by a refactoring tool. It also implies that the meaning of a model is highly language-specific and cannot be reused [Läm02]. From our point of view, a framework for generic refactorings can solely provide extension points to check the preservation of semantics. DSL designers can use these points to attach custom components that add additional semantic constraints to enforce the correctness of a refactoring.

The need for formal semantics poses a problem for GPLs. For complex languages (such as Java), there is either no complete formal definition of its semantics, or if there is one, it is rarely used by the respective refactoring tools. However, in the context of modelling languages, there is some chance to provide refactorings that are provably correct with respect to the preservation of the model's semantics, at least from a theoretical point of view. Modelling languages often exhibit a reduced level of complexity, so that they offer the chance to have complete definitions of their formal semantics. In particular, for DSLs that have a very narrow scope, the reduced semantic complexity may allow for proving the correctness of refactorings. From a practical perspective, proving the preservation of semantics will still be difficult. Proofs require a clean specification of the semantics. Currently, popular LWs use a GPL to specify static semantics (e.g. to implement name resolution) and transformations to source code of a GPL to define dynamic (or translational) semantics. Both approaches do not allow to reason about the implied semantics.

Until today, no definite answer regarding proving preservation of dynamic semantics in DSL instances has been given, resulting from the fact that no common sense in terms of formal semantics specification exists [Var02; DJK+06; RV07; BGM+11; RASG14].

Another more radical point of view is taken by Steimann in [Ste11; Ste15]. He argues that for a language without specified semantics every modification in these DSL's models is considered a refactoring since no behaviour specification can be violated. As already known, well-formedness rules (WFRs) of a modelling language are equivalent to its static semantics [NPA91; Hax14; RASG14; Ste15]. Therefore, Steimann makes use of this correspondence as already explained in Sect. 3.1.2. WFRs are the formal base of his generic refactoring approach. WFRs are translated into constraint satisfaction problems (CSPs), which do not only check if the constraints are violated but also recommend modifications to satisfy the CSP again. Consequently, Steimann sensitizes DSL designers for the specification of WFRs in their languages, which most often is avoided or only done informally. But most often, simple invariant checking is sufficient to prove behaviour preservation [Tri07; Ste11; GL12]. We hold the same opinion that the specification of the static semantics in terms of WFRs covers a wide range of behaviour preservation in DSLs. As already mentioned, we support the specification of pre- and post-conditions. If a DSL designer uses OCL as her constraint specification language for WFRs, then these rules can be provided as pre- and post-conditions for all DSL's refactorings. Since our solution the checking of pre- and post-conditions on the one hand, and a trial run of a particular refactoring on the other hand, we can at least prove preservation of the static semantics of particular DSL instances. As such, before applying a refactoring to a model, it can be determined in our implemented tool which WFRs would be violated. But it requires DSL designers to specify these rules carefully. More technical details are presented in Sect. 9.1.6.

In summary, one can say that there is some chance to avoid the limitations observed for GPL refactoring engines, if DSL designers are willing to specify semantics formally. Assuming they do, proofs must be established to show that a refactoring's pre-conditions suffice to guarantee the preservation of semantics. These proofs must be constructed by DSL designers for each DSL, similar to the proofs that were needed for refactorings for programming languages in the past. Notably, the specification of static semantics in terms of WFRs is supported by our approach and our implemented tool. Proving the dynamic semantics of a language still is difficult and subject to further research, as already mentioned above.

4.4. Conclusion

In this chapter a novel approach to overcome the shortcomings of existing related work (cf. Sect. 3.1) has been presented. Based on role models, structural requirements for refactorings can be generically formalised (Requirement 1 on page 25). Using a role mapping, such role models can be mapped to specific based modelling languages particular concrete refactorings should be provided for. From a conceptual point of view the only "restriction" a target DSL must meet is to be MOF-based (Requirement 10 on page 26). This mapping defines which elements of a language play which role in the context of a generic refactoring (Requirement 3 on page 25). Based on the mapping, generic transformation specifications are executed to restructure models and the mapped roles are bound. Thus, generic refactorings can be reused for different languages only by providing a mapping (Requirement 3). Furthermore, the same generic refactoring can be

repeatedly applied to one language (Requirement 2).

We have discussed the extent which generic refactorings can be reused to as well as the preservation of semantics (Requirement 4). Even though the latter is highly language-specific, a generic refactoring framework can still provide extension points to be parameterised by a language's semantics. Our approach supports this by means of checking the static semantics with WFRs (Requirement 5). Further technical insights and evaluation results are shown in Sect. 10.1.

Hereby this chapter is finished. The concept of our generic model refactoring approach has been highlighted from several viewpoints. In the following chapter we will present an approach for determining possible role mappings as suggestions for a DSL designer.

5 Suggesting Role Mappings as Concrete Refactorings

This chapter is an extension to the suggestion technique published in [RSA13]. In this paper, we implemented the suggestion engine manually and had to restrict it, e.g., in terms of not considering sub-metaclasses of an abstract super-metaclass. The graph querying approach presented here avoids such restrictions.

5.1. Motivation

To instantiate a generic model refactoring, we introduced the role mapping mechanism in Sect. 4.2.2. Such a role mapping maps a role model to the metamodel of the target language. To do so, one must specify which metaclasses play which roles in the context of the generic refactoring. As depicted in Fig. 4.2, role mappings are created by DSL designers. Based on knowledge of the target metamodel, DSL designers should be able to specify a role mapping, taking into account the characteristics of the metamodel and ideas about feasible refactorings.

However, the definition of role mappings can still require substantial effort. First, metamodels can be very complex (e.g., UML or Java). Thus, it can be difficult to identify the desired metaclasses intended to be mapped to the roles. The DSL designer must abstract from the complex structure of the metamodel and find a correct subset which can be mapped to the role model. Second, DSL designers may not be familiar with the process of role mapping and it might not be obvious which metaclasses need to be mapped to which roles. Third, language designers might not be aware of all potential role mappings and, thus, forget to specify mappings even if they were useful for DSL users. Fourth, incomplete mappings can sometimes imply how to map remaining yet unmapped parts of the role model. For example, mapping two metaclasses to respective roles might uniquely determine how all other parts of the role model need to be mapped. In such cases, DSL designers could use support to automatically complete the role mapping.

As a consequence of these observations, support in the process of mapping role models to parts of the target metamodel is needed. DSL designers should get recommendations about which concrete structures in the metamodel a role model can be mapped to.

In the following, we present our approach for the automatic derivation of valid role mappings based on graph querying, in order to fulfil Requirement 8 in Sect. 3.1.1.

5.2. Automatic Derivation of Suggestions for Role Mappings with Graph Querying

As indicated above, a DSL designer can have a vision about the refactorings for her DSL. Beyond that, there can be many more possible refactorings with respect to the structures in the target metamodel which role models can be mapped to. It is therefore desired to derive all feasible refactorings (i.e. role mappings) automatically. Then, DSL designers can select the role mappings that are suitable for their languages. That means, all available role models must be mapped to all possible structures in the target metamodel.

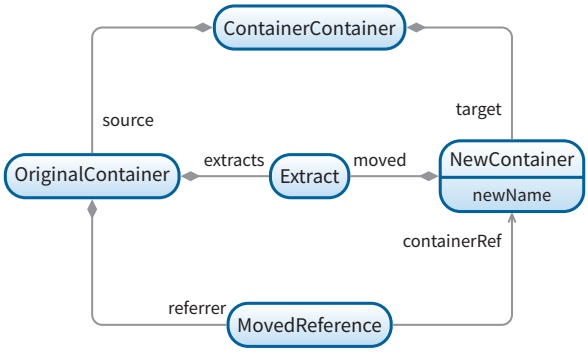
To get a valid role mapping, the respective roles and their collaborations must be mapped consistently to an applicable structure in the target metamodel. Since we use role models only to capture the structure that is required for a generic refactoring, but do not incorporate language-specific semantics, role models must be matched structurally to parts of the target metamodel. However, simply computing combinations of all pairs of roles and metaclasses quickly results in an extremely high number of role mappings due to the combinatorial explosion.

Role models are relatively small, because they only define the participating elements in the context of a generic refactoring. Thus, the quantity of valid matches of a role model can be very high depending on the metamodel's complexity. In addition, each role collaboration can be mapped to a path of references between metaclasses, which further on increases the number of potential role mappings. This can be seen later in the evaluation in Chap. 10.2.

With respect to this preliminary discussion we argue that a derivation approach based on *graph querying* is quite suitable since a MOF-conforming metamodel can be considered as a graph. It contains vertices (metaclasses) and edges (references) in between. The task of determining all valid role mappings is quite similar to *pattern matching* the role model in a given structure (the given metamodel). In the following, we illustrate this by an example.

Example 5.2.1:
Throughout this chapter, we use the graph querying engine GUERY[1] [DMTS12] (cf. Sect. 3.2.2) for the structural detection of deficiencies in models. To achieve this, we only have to convert a certain role model to a graph query accepted by GUERY. For this purpose, have a look at Fig. 5.1, where the conversion of a role model to a GUERY *motif* is illustrated. In (a) we see our generic *Extract X with Reference Class* refactoring again, whereas in (b) the corresponding GUERY motif is depicted. In Line 2, symbolic names for the vertex selection of the query are specified which reflect the roles of the role model. In the same manner, symbolic names for the edge selections can be specified in a query. The collaboration `extracts` between the roles `OriginalContainer` and `Extract` is depicted in Line 3. In the same line, one can see how the edges are qualified syntactically: After the name, the source and target vertex names are given,

[1] `https://code.google.com/p/queryframework/` (visited 10th February 2015)



(a) Role model of generic *Extract X with Reference Class* refactoring.

```
1  motif ExtractXwithReferenceClass
2  select OriginalContainer, Extract,
       NewContainer, ContainerContainer,
       MovedReference
3  connected by extracts(OriginalContainer>
       Extract)[1,1] find all
4  where "extracts.isContainment()"
5  connected by referer(OriginalContainer>
       MovedReference)[1,1] find all
6  where "referer.isContainment()"
7  connected by moved(NewContainer>Extract)
       [1,1] find all
8  where "moved.isContainment()"
9  connected by target(ContainerContainer>
       NewContainer)[1,1] find all
10 where "target.isContainment()"
11 connected by source(ContainerContainer>
       OriginalContainer)[1,1] find all
12 where "source.isContainment()"
13 connected by containerRef(MovedReference
       >NewContainer)[1,1] find all
14 where "!containerRef.isContainment()"
```

(b) Corresponding GUERY graph query.

Figure 5.1.: Example for conversion of role model to GUERY query.

as well as the allowed length of an edge. In this example, the edge in Line 3 must be of length 1. To reflect the different kinds of collaborations of a role model, `where` clauses can be given in the motif. In Line 4, one can see that the method `isContainment()` is invoked upon the given edge `extracts`. GUERY provides a light-weight adapter mechanism to support any kind of graph structure. The method `isContainment()` is provided by our adapter and returns **`true`** in case the reference of the queried metamodel is a containment reference. The role association between the roles `MovedReference` and `NewContainer` can be seen in Line 14. This small example should be self-explanatory. In the following, we will discuss some design decisions for the conversion of role models to GUERY motifs.

What we have seen until now is that the conversion of a role model to graph query can be achieved by referencing their names. The only special case we have to be aware of is when a role collaboration is to be mapped to a path in the target metamodel having a length greater than 1. For this case, consider the aggregation of `Extract` to `OriginalContainer` (`extracts`) as a subset of the role model in Fig. 5.1 (a). As explained in Sect. 4.2.2, our role mapping approach supports the mapping of collaborations to a path between metaclasses. Therefore, the type of a collaboration (e.g., `RoleComposition`) holds only for the first segment of the path it is matched to. The subsequent path segments do not need to respect the type of the mapped collaboration. To reflect this circumstance in a graph query, we introduce an intermediate vertex, for which the collaboration type is respected on the first edge segment. For the subsequent path segments another edge is generated, for which the type is not restricted.

Example 5.2.2:
Listing 5.1 shows how a multi-step path is reflected in a generated GUERY motif.

```
1 motif ExtractXwithReferenceClass
2 select OriginalContainer, _OriginalContainer_Extract_, Extract
3 connected by extracts(OriginalContainer>_OriginalContainer_Extract_)[1,1] find
      all
4 where "extracts.isContainment()"
5 connected by _extracts_(_OriginalContainer_Extract_>Extract)[0,3] find all
```

Listing 5.1: Conversion of collaboration to edge with intermediate vertex.

The vertex `_OriginalContainer_Extract_` in Line 2 is used to respect the collaboration type of `extracts` (`RoleComposition`) in Line 4. As a consequence, the remaining path segments are reflected by the edge `_extracts_` in Line 5 not being restricted. The maximum length of the path in this example is 4 and, at minimum, 1. This results from the specification of the edge lengths in Lines 3 and 5. The former must be exactly 1 while the latter is in the range 0 to 3.

This explanation of the conversion of role models to graph queries is only of informal nature. The conversion is pretty much straightforward and can be understood best by an example.

The actual querying of such motifs then is executed by GUERY. By specifying maximum path lengths, we are able to restrict the resulting set of possible structural matches. Every found match then corresponds to a valid role mapping. This is a simple, but really efficient methodology to query possible candidates for instantiating role mappings. In Sect. 10.2, we will present an evaluation with concrete numbers for different languages. Nevertheless, the resulting potential valid role mappings can be of high quantity. In the following, we will shortly discuss how to reduce such a result set further.

5.3. Reduction of the Number of Valid Matches

As explained in the previous section, the number of mappings of role models to target metamodels can be very high. The number of possible matches can be significantly reduced, if DSL designers incrementally map one role to one metaclass manually. This strategy is also applied in [HMK05] to find candidates for aspect extraction in Java programmes. We want to make use of this methodology in our approach as well.

Such a pre-selection of a valid manual mapping definitely makes sense for complex metamodels such as UML or Java. In case the DSL designer maps particular roles to concrete metaclasses beforehand, the resulting set of valid matches is reduced. Thus, this is supported in our derivation approach by restricting the vertices in the graph query according to the manual mappings. For explanation, have a look at our example mapping *Extract Track* for the Conference DSL in Fig. 4.6 on page 59. One can see that, e.g., the role `OriginalContainer` is mapped to the metaclass `Track` in the target metamodel. This circumstance is reflected in our query generation by means of restricting the type of the according vertex. This is illustrated in the following example.

Example 5.3.1:
Listing 5.2 shows that the vertex `OriginalContainer` can only be matched to the metaclass `Track`.

```
1 motif ExtractXwithReferenceClass
2 select OriginalContainer, Extract, NewContainer, ContainerContainer,
      MovedReference
```

```
3 where "OriginalContainer.getEClass().getName() == 'Track'"
```

Listing 5.2: Conversion of a role model to a graph query with a manual pre-mapping.

In line 3 it can be seen that the expression `OriginalContainer.getEClass().getName()=='Track'` does exactly this. It is checked that the actual name of the queried metaclass corresponding to the role (and the vertex in the query) `OriginalContainer` equals `Track`.

By such a manual pre-mapping the resulting set of potential valid role mappings is drastically reduced. As already mentioned, this can be an *incremental mapping* process by means of stepwise addition of a manual role-metaclass mapping after each iteration, if the resulting set still is too large. With every mapping that is applied manually, less valid role mappings, which can come next according to the manual mappings made until this point of time, are suggested.

In addition to our discussed approach, we see further potential for reducing the resulting set. On the one hand, cross-language analysis can be used to make more suitable suggestions in the context of a role model. In that case, those mappings of a particular role model must be analysed, which have already been specified by DSL designers. With this technique, e.g., the names of the mapped metaclasses can be used to find commonalities. A simple example would be the generic *Rename X* refactoring. This generic refactoring only consists of one role with a role attribute representing the new name. In most cases, this role is mapped to a metaclass called `NamedElement` or `Nameable`. Such commonalities must be found and the matched elements of the target metamodel, which correspond to the determined commonalities, can be suggested.

On the other hand, analysis of manual restructurings in instances of the target metamodel can be used to suggest suitable role mappings. With this approach, the interaction of DSL users with the concrete models must be observed. Applied changes must be recorded and then be analysed. For example, if a lot of restructurings are made for instances of a certain metaclass, this metaclass could be used as a base for the suggestions. A naive possibility for reducing the number of matches would be to use the investigated metaclass as filter and suggest only those matches containing that metaclass. A more intelligent approach would be to analyse the context of the restructurings and the metaclass in detail to find out which are the surrounding elements and which are their metaclasses. This would result in a reduced set of suggested mappings, since whole refactorings can be recognized and then be derived.

With this discussion, we are closing our approach of suggesting valid role mappings to a DSL designer. Our solution contributes to the Requirement 8 on page 8 regarding the specification suggestion of instances of generic refactorings in the sense of role mappings. Since this approach is based on graph querying, we want to bring it into context with similar work in the field in the following section.

5.4. Comparison to Model Matching

Mapping role models to metamodels is an essential task to obtain recommendations for generic refactorings if one employs our role-based refactoring approach. Thus, it is related to a certain extent to the more general task of model matching.

In [KDPP09], an overview of current approaches to compute differences between models can be found. A particularly interesting approach can be found in [LGJ07], where models conforming to arbitrary metamodels are matched. Our recommendation approach realises a structural matching

as presented in [LGJ07]. But, we do not employ signature matching, because the names of roles and collaborations are not relevant to obtain valid mappings to a target metamodel. If one compares the names of roles and collaborations with the elements of the target model, usually similarities cannot be found.

Also, in contrast to [LGJ07], we do consider only exact matches, because this is required to obtain correct role mappings. One can consider our recommendation approach as a special case of model matching where only structural properties are relevant and exact matches are always required.

The ontology community has faced a similar matching problem, because ontologies are often developed independently, much like our role models and the target metamodels, but describe common concepts. Thus, there is a variety of approaches to match and align ontologies [ES07]. Again, all strategies that focus on the structural properties of ontologies could be reused to obtain recommendations for refactorings. Again, all algorithms that involve matching of names are not suitable for our problem.

Most probably, existing structural matching algorithms—both for models and ontologies—could be reused to realise our recommendation approach for valid role mappings. However, even though we employed a specialised strategy to gather suggestions, we were already forced to restrict the set of recommendations. The combinatorial explosion of the number of role mappings leaves no other choice. We expect other approaches to face the same problem.

Our evaluation in Sect. 10.2 will reveal concrete numbers for different languages. Stating this, we close the discussion of our approach regarding the specification, instantiation, execution and suggestion of (generic) model refactorings.

5.5. Conclusion

To provide further assistance to language designers in the context of reusing generic refactorings, we have investigated how to gather suggestions for potential valid role mappings (Requirement 8 on page 26). An approach for reducing the result set of suggestion determination is also discussed in this chapter. Concrete evaluation results regarding the suggestion are discussed in Sect. 10.2.

In the following chapter, we emphasize the quality aspect and will correlate qualities, deficiencies in models and resolving refactorings.

6

Role-Based Quality Smells as Refactoring Indicator

The foundations of this chapter are based on our publication "Quality-Aware Refactoring for Early Detection and Resolution of Energy Deficiencies" [RA13]. The initial work for this chapter was carried out by our student Christian Vonsien in his minor thesis (Großer Beleg) [Von13].

6.1. Motivation

As Fowler *et al.* explained in [FBB+99], *bad smells* are considered as structures being candidates to apply specific refactorings on.[1] Such a refactoring improves the existing artefact with regard to quality requirements [Koz11] while preserving the behaviour. Such qualities might be, e.g., requirements for *response time* (in a client-server system), *energy efficiency* (in mobile applications), or just *reusability* (of components or models). Thus, the presence of a model deficiency deteriorates specific qualities and the execution of a related refactoring might improve them [FBB+99; SSL01; MTM07; Als09]. This means that model deficiencies directly influence qualities of the developer's artefacts. As already explained in Sect. 1.1, a connection between deficiencies in models, qualities and resolving refactorings exists. The main problem is that former research (cf. Sect. 3.2.2) recognized this relationship but this connection has not been used explicitly until now, which results in the following limitations. Without such a connection, it is not possible to give evidence about which quality requirements are not fulfilled by detected deficiencies. Also, it cannot be specified which deficiencies are resolved by particular refactorings. Thus, developers are not supported in focussing on specific qualities. They cannot detect and resolve deficiencies in combination. Hence, it is required to support developers in being informed about which quality requirements are not satisfied by specific models and in getting recommendations about how to resolve such violations.

This demand is absolutely necessary because fulfilling quality requirements is usually not realised in one single model fragment (such as, e.g., a Java method). Qualities are cross-cutting

[1] Note that we do not prefer the term *bad smell* because of its impreciseness, we used *model deficiencies* in previous chapters up to now.

concerns and implementations, related to quality requirements, may occur in several different places in the whole system [NB07; TOHS99]. This cross-cutting prevents that developers have support in focussing on specific qualities in isolation and in detecting deficiencies regarding only the focussed qualities. As a consequence, the quality-dependent detection and resolution of model deficiencies without an explicit relationship is very difficult and complex [MTM07]. We argue that the existing term *bad smell* is too imprecise and that a new concept is needed reflecting the correlation between model deficiencies, qualities and resolving refactorings. We will therefore introduce the new term *Quality Smell* in the following section. Furthermore, we propose a generic architecture enabling developers to explicitly define relations between quality smells, qualities and refactorings. This architecture is extensible and enables the detection of quality smells and their resolution by refactorings. Both the conceptualisation of quality smells and the provided architecture are our main contributions in this chapter. We argue that the understanding about how model deficiencies, qualities and resolving refactorings correlate can benefit from the concept of quality smells.

6.2. Correlating Model Deficiencies, Qualities and Refactorings

Before we start the conception, we want to provide an example which we will extend step by step in this section. The example has been taken up from Sect. 4.2.4 regarding the *Feature Envy* model deficiency proposed in [FBB+99]. *Feature Envy* regards the qualities *high cohesion* and *low coupling* to achieve higher productivity and less design effort for the developer [OGB+11]. Following our example in Sect. 4.2.4, envious statements of such a method can be extracted into a new method which in turn is moved to the envied class. In the following, we will present our approach enabling the specification of such a model deficiency, relating it to the mentioned qualities and defining the composite refactoring *Extract and Move Method* for resolving it.

Continuing our previous argumentation, we explicitly correlate model deficiencies, qualities and refactorings, which results in the following definition of quality smells.

> **Definition of *Quality Smell*:** A Quality Smell is a certain structure in a model, negatively influencing specific quality requirements, which can be resolved by certain model refactorings.

More precisely, the notion of *quality smell* has two aspects. First, it can be considered from a conceptual point of view. This can be understood as the definition above and refers to the general specification of a quality smell. The specification defines how the detection of a quality smell is achieved by means of a detection procedure. This is realised at the metalayer where the according DSL's metamodel is located (M2). Here it is defined which qualities such a quality smell influences negatively and which potential refactorings can resolve it. Second, a quality smell occurs when the detection procedure is applied and matches. Then, a quality smell exists physically in a model at the model's metalayer (M1). Here, the occurrence of a particular quality smell is related to the concrete model structures which cause the appearance of the quality smell. Both of these aspects are reflected in the upcoming concept.

For being able to provide support for detecting and resolving quality smells, the following three components are essential constituents in the overall architecture. First, a central *quality smell repository* is needed in which potential quality smells are registered (M2). The specifications of

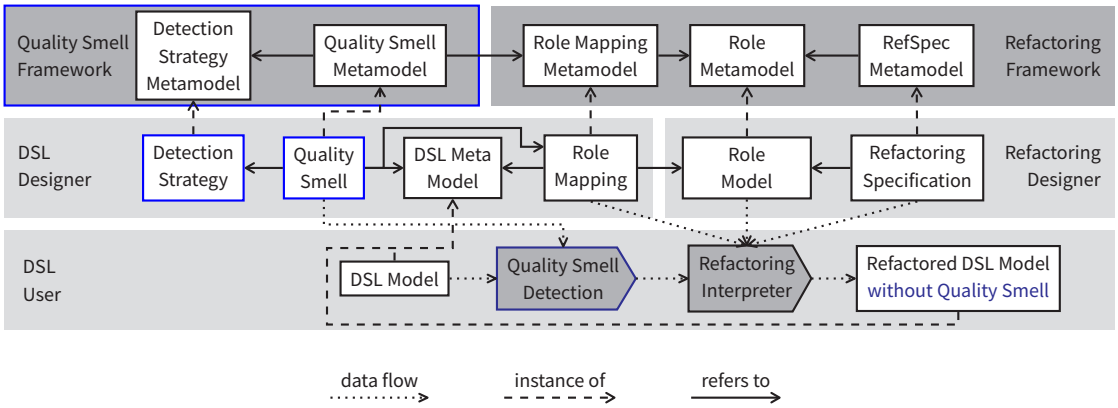


Figure 6.1.: Refactoring architecture extended by Quality Smell infrastructure. The blue-framed parts denote the quality smell additions.

these registered quality smells must be evaluated against particular models to examine whether a quality smell is present in a model or not (M1). Second, for evaluating the quality smell specifications against particular models, a flexible mechanism must be provided for making different kinds of detections available in the context of a model. Therefore, we propose a *quality smell detection repository*. Third, for resolving quality smells, the above architecture must be able to interact with an existing model refactoring architecture. For a better understanding, we adopt the architecture shown in Fig. 4.2 (cf. page 54) and extend it by the new constituents. The adjusted architecture is depicted in Fig. 6.1.

Since we have already presented our approach and architecture regarding model refactoring in Sect. 4.2, only the other two repositories are introduced in the following.

6.2.1. Quality Smell Repository

The metamodel of the quality smell repository is depicted in Fig. 6.2. As can be seen in the centre of the right hand side of the figure, the repository is represented by the metaclass `QualitySmellModel`. We assume a single instance of this metaclass and call it Quality Smell Model (QSM). For enabling the support to focus on particular qualities, the QSM contains several `qualities`. Each `Quality` has a `name` and can be considered as the abstraction of the quality concepts from literature [MRW77; BBKL78; Gra92; ISO01]. Thus, a quality plays a role in the system or is specified as a quality requirement. In addition, a quality can be marked as `active` expressing the fact that only those quality smells are detected being related to this quality.

Such qualities might be derived, e.g., from quality contracts in multi-quality aware systems [GWRA12]. Those contracts specify required and provided qualities for software components and their variants. In this sense, such contracts formally describe dependencies between artefacts with respect to qualities. As a consequence, these contracts can be used to gather the required and provided qualities and to populate the set of qualities in our quality smell repository. Furthermore, the QSM contains several generic `QualitySmells`. The reason for distinguishing this concept from `ConcreteQualitySmell` is that the latter is very specific in terms of the platform, e.g., where the model can be run, or which libraries are used. But a generalisation is essential for being able to group concrete quality smells under their common abstract meaning.

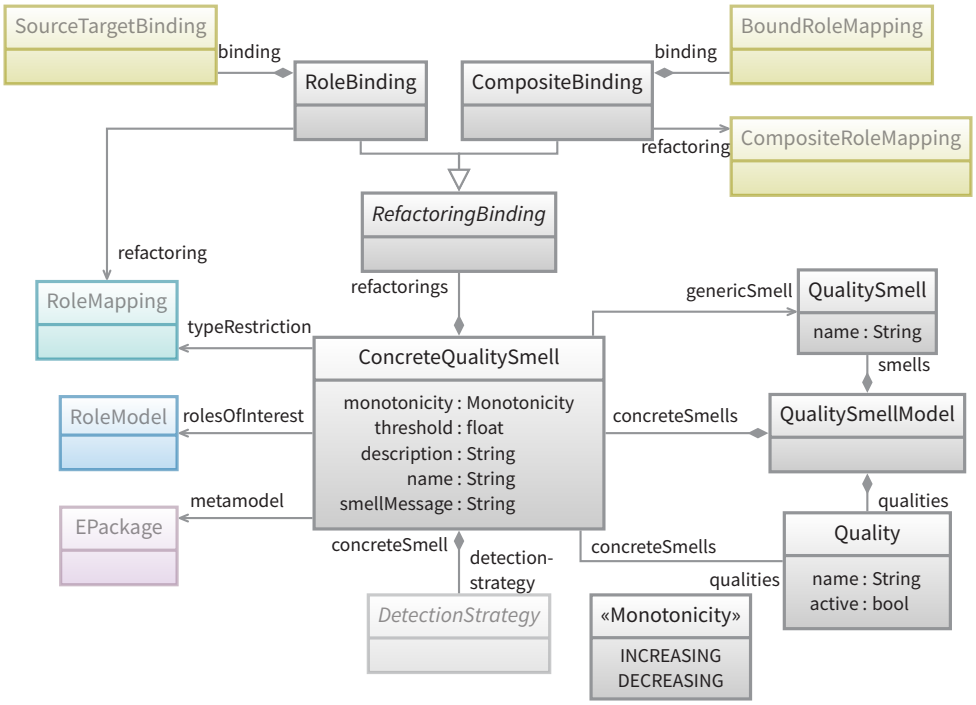


Figure 6.2.: Metamodel of the quality smell repository.

In this sense, it expresses the fact that it must be defined which meaning a particular generic quality smell has in a certain context. A project or the workspace of an MLDE, e.g., can be such a context. This generalisation enables tools to let developers just focus on those quality smells being grouped under a certain generic quality smell while the others are not considered. In our example, such a generic quality smell would be named *Low Cohesion*.

Of course, the QSM contains several instances of the metaclass `ConcreteQualitySmell`. A concrete quality smell has a `name`, a `description` and a `smellMessage`. These properties are self-explanatory and can be used for presentation in the user interface (UI). Furthermore, a concrete quality smell has the properties `monotonicity` and `threshold`. They are used to determine if a detected *candidate* is considered as a concrete occurrence of a quality smell in the particular context. If the `monotonicity` is set to `INCREASING` then the quality is satisfied better the higher a calculated value is. This means that the candidate is considered a *quality smell occurrence* if the calculated value is less than or equals the given `threshold`. Thus, an instance of the metaclass `ConcreteQualitySmell` is the specification of how its quality smell occurrences can be determined. In turn, if the `monotonicity` is `DECREASING` the quality gets worse the higher the calculated value is. The calculation of values will be presented in this section in a short while. For our example *Feature Envy*, one would set the monotonicity to `DECREASING` and the `threshold` to -1.0 expressing the difference between the count of the inner entities and the outer entities. This means that if at least one more outer entity is counted, we consider the candidate as a quality smell occurrence in the example.

Furthermore, a concrete quality smell refers to its abstract `genericSmell` and, thus, forms a more fine-grained distinction. A concrete quality smell is specific for a concrete modelling

language and therefore refers to the language's `metamodel`. In our example, this is the Java language. Java is not a modelling language per se, but JaMoPP and its metamodel are able to provide models for Java programmes. We argue that particular model elements of a detected quality smell occurrence can be of further interest in a subsequent analysis of the result or a resolving refactoring, for instance. Consequently, we use our formalism of role models to enable the specification of such `rolesOfInterest` for a particular quality smell occurrence. Those defined roles can be considered as symbolic names for the detected elements. Such a referenced `RoleModel` can be specified anew, specific for this concrete quality smell, or just be reused by an already defined generic refactoring. For our example, such a role model would contain the roles `EnviousStatements`, `EnviedClass` and `ExtractedMethod`.

In addition, in our approach we provide the possibility to restrict the elements of interest further by specifying a `RoleMapping` as `typeRestriction`. It is not mandatory to provide `rolesOfInterest` or a `typeRestriction` at all, but if they are defined, resolving refactorings can benefit from it. A type restriction, like the one in our example, would map the roles `EnviousStatements` to the metaclass `Statement`, `EnviedClass` to `Class` and `ExtractedMethod` to `Method`.

For being able to specify which potential refactorings can be applied to resolve a quality smell occurrence, one has to provide a `RefactoringBinding` for each. Here we have two alternatives. First, a `RoleBinding` can be given which refers to an existing `refactoring` by means of a `RoleMapping`, expressing that this refactoring is able to resolve the concrete quality smell. Now we come back to the `rolesOfInterest`. In case such a role model (containing the `rolesOfInterest`) was provided, a `RoleBinding` allows for the specification of a `SourceTargetBinding` we already know from Sect. 4.2.4. Thus, we can define the meaning of the `rolesOfInterest` played by elements in the detected quality smell occurrence in the context of the given resolving refactoring. Roles from the `rolesOfInterest` are mapped to roles from the referenced role model of the given `RoleMapping`. In case a `typeRestriction` is provided, static type checking of such a mapping can be realised. The second alternative allows for the specification of a resolving composite refactoring by provision of a `CompositeBinding`. Similar to the `RoleBinding`, a `CompositeRoleMapping` is referenced relating the refactoring as a resolution to this concrete quality smell. Therefore, a `BoundRoleMapping` (cf. Sect. 4.2.4) can be provided for specifying the meaning of the `rolesOfInterest` with respect to the played roles of the given composite refactoring. Again, the metaclass `BoundRoleMapping` enables the mapping of roles between concrete refactorings. In our example, we would provide an instance of `CompositeBinding` and relate it to the composite refactoring *Extract and Move Method*. In order to define the meaning of the `rolesOfInterest`, we would realise the following mapping:

1. `EnviousStatements` to `Extract` of *Extract X with Reference Class* (cf. role model in Fig. 4.3 on page 56)
2. `ExtractedMethod` to `NewContainer` of *Extract X with Reference Class*
3. `EnviedClass` to `TargetContainer` of *Move X* (cf. Fig. A.2 (a) on page 169)

According to the role mappings we have already seen for *Extract Method* in Fig. 4.6 (a) on page 59 and for *Move Method* in Listing 4.2 on page 69, it can be observed that the metaclasses

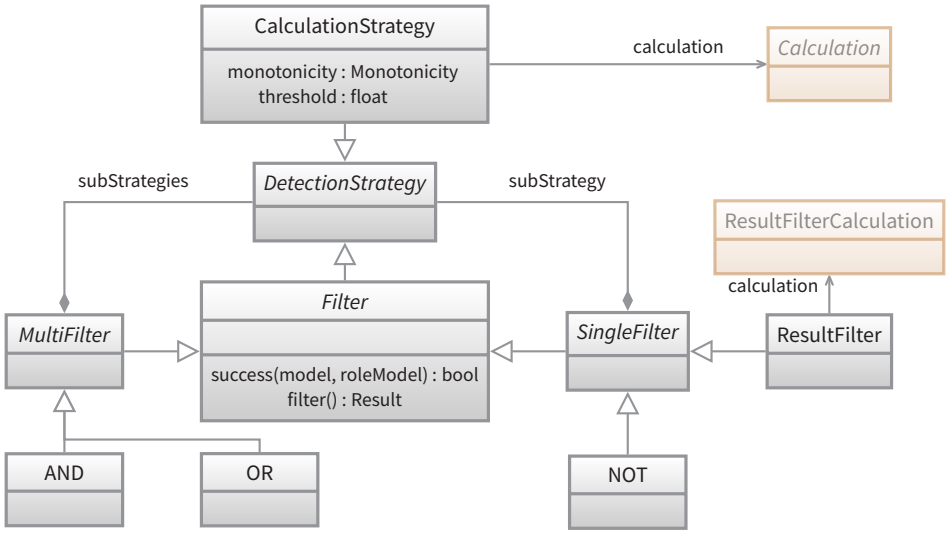


Figure 6.3.: Metamodel of the quality smell repository `DetectionStrategy` part.

playing the particular roles are compatible. By providing the `typeRestriction` of our example, this can also be checked statically.

As can be seen in Fig. 6.2, one metaclass is not yet discussed. To specify *how* quality smell occurrences are determined, a *detection strategy* must be defined. In [Mar04; LM06], it was examined that detection mechanisms, only taking one single aspect into account, are often too fine-grained and do not reflect particular deficiencies from a higher level of abstraction. Thus, detection strategies can be used for composing and filtering fine-grained detection mechanisms. Therefore we provide the metaclass `DetectionStrategy`. This part of the metamodel is presented in Fig. 6.3. In the simplest case, a concrete detection strategy can be a single `CalculationStrategy`, again having a `monotonicity` and a `threshold`. These properties must be provided for a `CalculationStrategy` again in order to reuse it as an isolated building block for other detection strategies. Therefore, an instance of this metaclass always references a `Calculation` which resides in the quality smell calculation repository because both parts might evolve independently. The calculation is presented in the following Sect. 6.2.2.

In the more complex case, `Filters` can be composed by means of propositional logics. Therefore, the operators `NOT`, to negate the contained `subStrategy`, and `OR` and `AND`, to form the disjunction or conjunction of the `subStrategies`, can be used. Thus, a detection strategy can form a tree of filters corresponding to a formula in propositional logics. Only if the root element of the tree evaluates to `true`, the concrete quality smell is considered as being physically present in a model. Hence, a quality smell candidate becomes a quality smell occurrence. A `Filter` provides the method `success(model, roleModel):bool` which succeeds if all child strategies succeed. Thus, the invocation of a `Filter` is only applied if the sub-filters succeed. If it is successful, the `filter():Result` method returns the final result. This result can also be used as input in a parent filter.

Then, the `CalculationStrategy` is considered as a reusable block for other detection strategies. Thus, it is not specific. In contrast, a `ResultFilter` is specific for the concrete quality smell and can be understood as an interpreter for an incoming preceding result. It

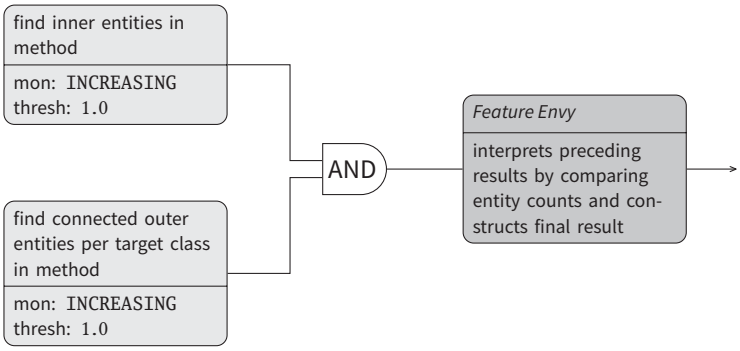


Figure 6.4.: Detection strategy of the *Feature Envy* quality smell.

represents the meaning of the previous result and references a `ResultFilterCalculation` which resides in the quality smell calculation repository. This repository is explained in the next section.

Let us clarify the way how the metamodel works with our example. As already indicated in the beginning of this section, the detection of *Feature Envy* comprises three separate steps: 1) the determination of all inner entities in a method, 2) the determination of all outer entities per envied class in a method, and 3) the interpretation of the previous steps. As can be observed, the first two steps are independent from the *Feature Envy* quality smell and can be considered as two different metrics. In contrary, the third step is specific for this quality smell, counts the inner and outer entities, and examines if the former are greater than the latter. If this is not the case a concrete quality smell takes effect and a candidate turns into a quality smell occurrence. For a better understanding have a look at Fig. 6.4. The input for the `AND` filter are the reusable calculation strategies for the determination of inner and outer entities. Both succeed if at least 1.0 entity is found. In that case, `AND` succeeds as well and passes the result to the `ResultFilter` being specific for *Feature Envy*. Here the entities are counted and the final result is created. In the example a simple subtraction of the particular entity counts was chosen, but other approaches are definitely possible, such as computing the ratio of both counts. The selection depends on the preferences of the concrete quality smell. This also makes clear that a ratio-based approach for the *Feature Envy* detection strategy can also reuse the aforementioned calculation strategies for determining the inner and outer entities. Only the `ResultFilter` is specific.

In the following section, we present deeper insights about how a particular `Calculation` is specified and which kinds we support.

6.2.2. Quality Smell Calculation Repository

As argued before, we separate the quality smell repository from the quality smell calculation and introduce an own repository for it, which is represented by the `CalculationModel` in Fig. 6.5. This model contains all `calculations` which are referenced in the QSM by a `CalculationStrategy`. The metaclass `Calculation` is abstract and has a `name` for identification purposes, since it can be reused for different detection strategies. The most important concept of a `Calculation` is the operation `calculate`. It has two incoming parameters: 1) the model which the calculation is executed on, and 2) the role model in which the

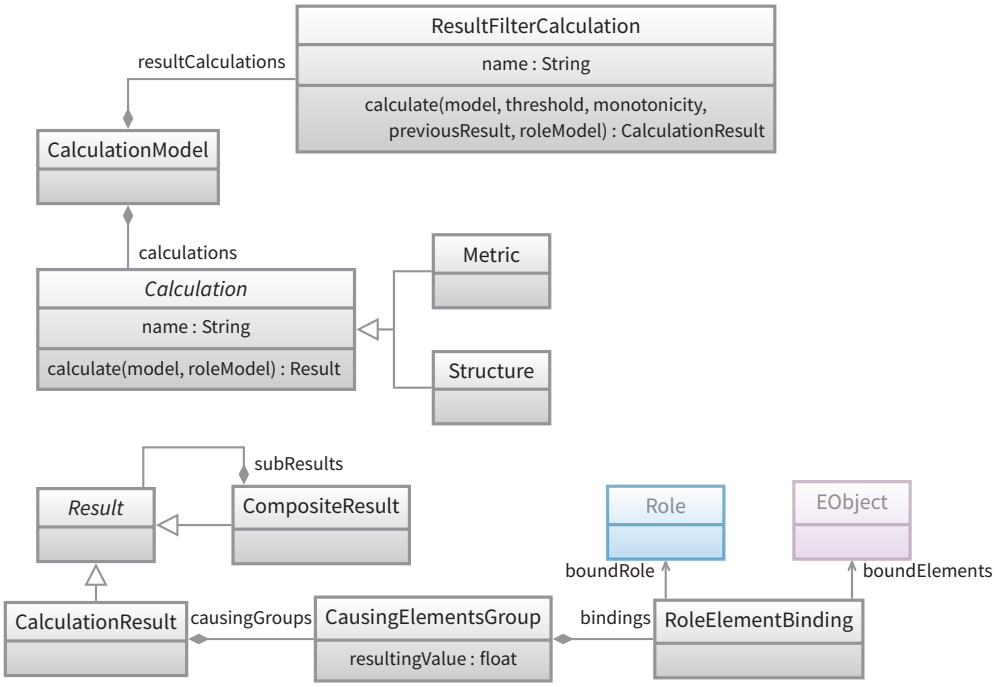


Figure 6.5.: Metamodel for different kinds of calculations.

`rolesOfInterest` are specified. The return type of this operation is `Result`, which can either be a `CalculationResult` or a `CompositeResult` being used for composing preceding results and to pass it on to the next `Filter`.

Each time an instance of `CalculationResult` is created, the concrete model elements causing the quality smell occurrence can and should be referenced to make the cause traceable. This is realised by means of `CausingElementsGroups`. Any such group has a `resulting-Value` expressing the determined characteristic of the detected model elements which cause a quality smell occurrence. This value then can be compared to the threshold specified in the `CalculationStrategy` depending in the given `Monotonicity`. A `CausingElementsGroup` can be considered as a set of model elements belonging together in the context of the quality smell occurrence. In order to trace the occurrence of a quality smell back to concrete model elements causing the quality smell (cf. Requirement 6 on page 33) such groups must provide `RoleElementBindings`. Such a binding establishes the connection to the `rolesOfInterest` defined in a `ConcreteQualitySmell`. Thus, the roles are bound to concrete model elements which then can serve for further analysis and especially as input for resolving refactorings.

Coming back to our example, this would mean that the role `EnviousStatements` is played by the last two statements of the method in Fig. 4.13 (a) on page 68. The role `ExtractedMethod` is played by the `printDetails()` method in Fig. 4.13 (b). And the role `EnviedClass` is played by the `DetailsPrinter` class to which the method is moved.

Furthermore, a `CalculationModel` contains `ResultFilterCalculations`. These calculations are referenced by instances of the `ResultFilter` metaclass. Again a `name` must be provided for identification. Since a `ResultFilter` is used to interpret results, the function `calculate (..)` contains some more parameters needed to calculate a result specific to a particular quality

smell. Thus, apart from the `model` and `roleModel`, the `threshold` and `monotonicity` given in the `CalculationStrategy` are passed. In addition, the `previousResult` in terms of a `CompositeResult` is passed in order to have all previously calculated information available and to interpret it appropriately.

Until now, we have only seen how a detection strategy can be developed but not how a concrete calculation can be realised. Therefore, we provide two concrete `Calculation` subclasses: `Metric` and `Structure`. These subclasses represent the two common approaches for detecting model deficiencies in general, as we have already discussed in Sect. 3.2.2. First, metrics are a common and well established technique for giving evidence about the quality of a software artefact [Soc93; CK94; Mar01; SSL01; BD02; AST10; KVGS11; SK11]. Due to this background, our approach supports this kind of quality calculation with the metaclass `Metric`. This means, the `calculate` operation in this subclass must encode the calculation of the metric appropriately. Implementing a metric in Java is also realised in [Are14]. Second, formalisations of structures, such as anti-patterns or architectural deficiencies, are also used to find quality smells in software artefacts [SW03; KE07; GPEM09; ABT10; KGH10; DMTS12]. As a consequence, the metaclass `Structure` provides means for supporting structure-based quality smells. This means that the determination of a structure-based quality smell must be encoded in the `calculate` method of the `Calculation` metaclass. A concrete instantiation of the conceptual framework presented in this chapter is provided in Chap. 7. There we will show especially how structure-based quality smells can be specified easily.

6.3. Discussion

Since quality smells do not concern any formal properties of the developed artefacts, we consider quality smell detection and resolution as a flexible and agile process. Developers may reject recommended refactorings for specific detected quality smell occurrences or not. Functionality will not break, but in case the refactoring is executed the specific quality requirements may be satisfied better. But it is important to realise that our presented approach is a conceptual framework and a first solution in order to correlate model deficiencies, qualities and resolving refactorings in the concept of *quality smell*. An example instantiation is presented in Sect. 7.3.

The most critical limitation of our approach is, caused by the explicit relation to qualities, that dependencies between qualities may exist [CNYM00; Koz11]. This means that different qualities can influence each other which may result in interferences or even conflicts. Consider, e.g., the qualities reusability, modularity and readability, and a software implemented in Java. Reusability can be improved with an *Extract Method* refactoring splitting a method into several and referencing the new one at the old location of the extracted code. By separating conceptually different code from one method to another both the divided code can be reused better and the modularity increases. Other opportunities would be to perform *Pull-Up Method*, *Extract Class* or *Extract Superclass* on the newly extracted method which results in a different class containing the moved method [FBB+99]. These refactorings again might increase reusability, but the readability suffers, because since the code originally located together now is spread over two different classes. This small example illustrates that qualities may be interrelated. In the worst case, resolving quality smells improves a quality but may deteriorate others.

A similar scenario is that resolving quality smells might introduce new other quality smells.

In our approach, quality dependencies are reflected only by the specification of an appropriate `DetectionStrategy`. Propositional logics can be used for combining calculations strategically. Thus, currently the DSL designer is responsible for the specification of dependencies. The DSL user can examine the impact of a resolving refactoring, since we allow for a refactoring preview. Thus, the effect of a refactoring can be investigated and it is possible to check, if a new quality smell occurrence is introduced. As a consequence, an appropriate abstraction for the specification of quality dependencies and the effects of refactorings is postponed to future work (cf. Sect. 11.3).

Another restriction is that we abstract from the whole aspect of measuring certain qualities and outsource this task. It is not verified whether a particular refactoring really resolves the related quality smell occurrence with respect to better measured values. The reason is that the relation between qualities, quality smells and refactorings is established manually. Our approach assumes that developers verified the correct relation previously before establishing it. Such a verification can only be done for a concrete quality smell and a concrete model refactoring since the generic smells and qualities do not have any formal foundation. Thus, developers must investigate if a specific relation between a quality smell and a refactoring really increases particular quality satisfactions before the relation is to be established. For the quality *energy efficiency* the work published in [Wil14] is very promising. Therein, an approach is presented being able to profile and test the energy consumption of mobile applications. The generic framework of Wilke can be used to measure energy consumption before and after resolving an energy-related quality smell. The developer then can be informed about how much energy is saved.

6.4. Conclusion

In this chapter, we motivated that an explicit relation between model deficiencies, qualities and resolving refactorings exists and is needed to detect deficiencies violating potential cross-cutting quality requirements. Therefore we introduced the new term *quality smell* and presented a conceptual framework for specifying quality smells and their concrete quality smell calculations. A quality smell establishes an explicit relation between model deficiencies, qualities and refactorings from a conceptual point of view (cf. Requirement 1 and Requirement 2 on page 33). A concrete occurrence of a quality smell can trace it back to the causing model elements (cf. Requirement 6) and suggests refactorings potentially resolving the quality smell occurrence (cf. Requirement 3). 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. In case all input roles of a resolving refactoring are mapped the detection and resolution of quality smells can be automated completely. In case several potential resolving refactorings exist a ranking mechanism could be used to determine which refactoring to apply first. As a consequence, our role-based approach of specifying quality smells allows for static analysis and automation. Resolving quality smell occurrences with refactorings now is easier than without an explicit relation between them. Furthermore, we support metrics-based and structure-based calculations of quality smells and provide an architecture being easily extensible (cf. Requirement 4 and Requirement 5). Complex combinations of concrete quality smell calculations can be specified by means of detection strategies and propositional logics. A discussion about limitations and future work closes this chapter.

7
A Quality Smell Catalogue for Android Applications

This chapter is based on a joint work with our student Martin Brylski [Bry14]. A small part has been published in our paper "A Tool-Supported Quality Smell Catalogue For Android Developers" [RBA14].

To show validity of Hypothesis 3, a quality smell catalogue is mined and compiled in this chapter. Therefore, at first a reasonable catalogue schema is explained in which the quality smells are characterised. Afterwards, we explain the process of acquiring information, which quality smells can be mined and extracted from. Then the conceptual framework from Chapter 6 is instantiated for structural-based quality smells and the catalogue is presented.

7.1. Quality Smell Catalogue Schema

The constituents of a catalogue must be comparable in order to identify particular relations between them. Furthermore, each catalogue has a specific intent about which it should inform the reader. In our case, the intent is to present quality smells according to the conceptualisation in Chap. 6. Thus, different catalogues focus various properties of its constituents. As a consequence, a common schema is needed. In the following, we will explain the schema of our quality smell catalogue. It is based on other catalogue schemas [GHJV94; BMMM98; FBB+99; Cun13].

Name: A unique and explaining name reflecting the core idea of a quality smell.

Context: A characterisation of a quality smell regarding its application.

Affected Qualities: A list of qualities being influenced negatively by a quality smell.

Roles of Interest: A list of role names that are of interest in a particular quality smell.

Description: A meaningful explanation of the problem to solve and a descriptive example. It is explained to which elements the roles of interest are bound.

Pattern: A description of the calculation strategy (cf. Sect. 6.2.1) detecting quality smell occurrences. Since this catalogue only contains structural quality smells the calculation strategy is considered to be a graph pattern. For structure-based quality smells, the roles of interest usually appear in the pattern as named concepts.

Refactorings: A list or refactorings resolving a quality smell. This represents the solution. How the roles of interests are used in a resolving refactoring is explained here.

References: A list of secondary sources about a particular quality smell.

Related Quality Smells: A list of related quality smells.

This schema serves as a basis for the catalogue. Before it is presented, the next section shows which strategy we applied to mine a quality smell catalogue.

7.2. Acquiring Quality Smells

Mining a catalogue in general and a quality smell especially is not a trivial task. Therefore, a structured procedure is necessary. In an early stage, the target domain of the catalogue was fixed to application development of mobile Android devices. The reason is that certain properties of mobile applications heavily influence the quality *energy efficiency* [Wil14], which we find a very interesting domain. But it can be seen, some other qualities emerged also being important in that domain. Furthermore, the open-source character of Android encouraged us to get deeper insights and contributions from the community.

We assume that Android developers know best about good practices in mobile application development. Furthermore, they have implicit knowledge about problems regarding satisfaction of particular quality aspects and how they can be solved. As a consequence, the challenge is how to identify and extract existing implicit knowledge from a huge community.

Therefore, we decided to restrict the search to a set of reasonable sources for mobile Android developers. Due to their relevance and popularity, we chose the following internet platforms as an information base for our mining process:

Stackoverflow [1] One of the most popular developer communities, since it uses a reputation system for questions and answers. It belongs to the StackExchange network.

Programmers StackExchange [2] It also belongs to the StackExchange network and concerns with programming in general.

Android Enthusiasts [3] Is part of the StackExchange network and is about the complete Android ecosystem.

Android Issues [4] The official bug tracker of the Android system.

[1] `http://stackoverflow.com/` (visited 4th March 2015)
[2] `http://programmers.stackexchange.com/` (visited 4th March 2015)
[3] `http://android.stackexchange.com/` (visited 4th March 2015)
[4] `https://code.google.com/p/android/issues/` (visited 4th March 2015)

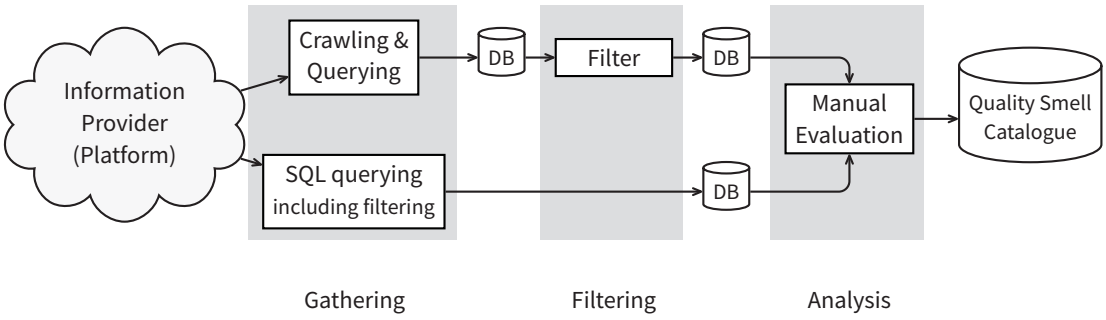


Figure 7.1.: Strategy of information extraction.

Android Developers Newsgroup [5] Is the official discussion forum for Android developers.

Android Developers Blog [6] The official blog from Google employees regarding Android development.

Android Design Patterns [7] Another popular blog covering mainly programming issues in Android.

Android Developer Documentation [8] The official documentation from Google regarding Android development.

Google I/O Talks [9] Collection of different topics from the Google I/O conferences.

To master the huge amount of available data, we applied the strategy in Fig. 7.1 to be able to mine a catalogue of quality smells. The process is divided into three parts: gathering, filtering and analysis. In the *gathering* phase, official interfaces for querying the respective provider were used to download the available data into a local database.[10] This was done for the sake of independence from the provider. Some providers do not offer a query interface. For those, we implemented a crawler collecting the data. In the phase of *filtering*, a preliminary selection of potential relevant information was realised. This was done by querying the local database by means of relevant keywords. The focus of these keywords comprised aspects such as, e.g., *energy efficiency, memory, performance*. Furthermore, those keywords were connected with terms like *slow, bad, leak, overhead.* The result of the filtering was persisted to the database again. The constituents of the StackExchange network can be queried by means of a provided SQL interface.[11] Thus, the gathering and the filtering could be applied them in one single step. The *analysis* was a completely manual phase. This could not be applied automatically anymore, since the gathered information had to be interpreted to extract knowledge from it. In this phase, the available information was read and additional references were navigated to. On that base, the

[5] `http://groups.google.com/d/forum/android-developers` (visited 4th March 2015)
[6] `http://android-developers.blogspot.de/` (visited 4th March 2015)
[7] `http://www.androiddesignpatterns.com/` (visited 4th March 2015)
[8] `http://developer.android.com/` (visited 4th March 2015)
[9] `https://developers.google.com/events/io/` (visited 4th March 2015)
[10] We restricted the period of time from 2010 until July 2013.
[11] `http://data.stackexchange.com/` (visited 4th March 2015)

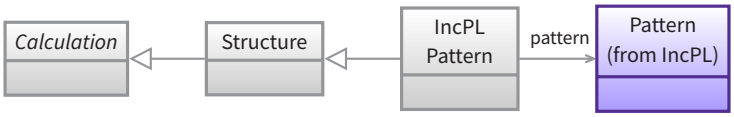


Figure 7.2.: Excerpt of quality smell calculation metamodel only showing the part for structure-based calculation with IncPL patterns. The metaclass `Pattern` is located in the metamodel of IncPL and referenced in our metamodel.

decision was made if a quality smell can be extracted from the information or not. This was the most expensive phase.

As a result, a list of Android-specific quality smells could be mined [RBA14]. Many of them are only descriptive and abstract for the time being. But we clearly defined 9 quality smells. This means that they are characterised according to our catalogue schema and they are specified precisely. The following sections present the realised quality smells.

7.3. Structure-Based Quality Smells—A Detailed Example

As already mentioned in Sect. 6.3, our quality smell approach represents a conceptual framework. In this section, we therefore want to provide a concrete instantiation for structure-based quality smells. To compile a pattern catalogue, a *pattern language* to describe the constituents of the catalogue is required [BC87; GHJV94; Sai03].

7.3.1. The Pattern Language

For being able to decide which pattern language to use, we have to anticipate that the whole quality smell framework is instantiated in the Eclipse IDE using the Eclipse Modeling Framework [SBPM08] (EMF), which provides sophisticated means for metamodelling. Further technical details are explained in Sect. 9.2. Thus, since we rely on the EMF, we will use the EMF-based query language *IncQuery*[12] [BHH+12] as the pattern language for structural quality smells. Since this pattern language has no dedicated name, we refer to it as *IncQuery Pattern Language (IncPL)* in the following. But again, we emphasize that IncPL is only an example pattern language to be able to specify the patterns precisely. Other languages, such as GReQL [BE08], Reclipse [vDMT10] or GUERY [DMTS12], could have been used, too.

IncPL reuses the concept of *graph patterns* for being able to specify complex queries on top of model structures.[13] IncPL graph patterns are reusable and therefore can be composed to more complex ones. Furthermore, IncPL can be used to specify derived features in EMF-based metamodels [SHV13]. One favourable property of IncPL we will exploit in the following is the fact that it exposes a metamodel. This is the main reason why we use IncPL because the metamodel is EMF-based and we also rely on it. Therefore, we extend the metamodel of quality smell calculations from Fig. 6.5 an page 86 with the metaclass for IncPL patterns. This extension is depicted in Fig. 7.2.

[12] `https://www.eclipse.org/incquery/` (visited 10th February 2015)
[13] `https://wiki.eclipse.org/EMFIncQuery/UserDocumentation/QueryLanguage#Language_concepts` (visited 3rd March 2015)

With this small but powerful addition, it is now possible to determine structure-based quality smells by means of IncPL patterns. With the help of JaMoPP (cf. Sect. 2.2.1), even Java programmes can be queried. In the following section, a concrete quality smell for Android applications is explained and specified. Deeper insights concerning the concrete IncPL syntax are revealed also. Afterwards the remaining catalogue of Android-related quality smells is presented in Sect. 7.4.

7.3.2. Quality Smell: Interruption from Background

In this section, our identified quality smell *Interruption from Background* is presented in detail [RBA14]. In addition to the schema explained in Sect. 7.1, the property **Pattern** is specialised to **IncPL Pattern** in order to reflect that IncPL patterns are used to specify structure-based quality smells.

Name: Interruption from Background

Context: UI

Affected Qualities: User Expectation, User Experience, User Conformance

Roles of Interest: `InterruptingStatement`

Description: According to the Android developer guide[14] users should not be interrupted when working with a mobile device, since interruption does not conform to the user's expectations. The assumption behind this demand is that when users start an application, they do it on purpose and do not want to be interrupted. In the worst case, user-supplied data could get lost. This can happen in case an `Activity`[15] is started explicitly or a `Toast`[16] is created from within background workers like `BroadcastReceiver` and `Service`. Listing 7.1 provides an example.

```
1 public class InterruptingService extends Service {
2    public IBinder onBind(Intent intent) {return new Binder();}
3    public void onCreate() {
4       super.onCreate();
5       Toast.makeText(this, "Hello World!",1000).show();
6    }
7 }
```

Listing 7.1: Interrupting service.

It shows an interrupting background service creating a `Toast` (Line 5) which pops up a dialogue and disturbs the user. This is to be avoided.

IncPL Pattern: The Listing 7.2 presents the IncPL graph pattern which detects the *Interruption from Background* quality smell.

[14] `http://developer.android.com/guide/practices/seamlessness.html#interrupt` (visited 3rd March 2015)
[15] A window in which UI elements are placed.
[16] A pop-up window in Android.

```
pattern interruptionFromBackground(InterruptingStatement:
    ExpressionStatement){
   // look if it's Service or BroadcastReceiver
   Class.^extends(actualClass, superClassRef);
   NamespaceClassifierReference.classifierReferences(superClassRef,
       classifierReference);
   ClassifierReference.target(classifierReference, superClass);
   find isServiceOrBroadcastReciever(superClass);

   // determine if interrupting methods are invoked
   find startsActivityOrToast(InterruptingStatement);

   // look of the interrupting code is executed in a method of the actual
       class
   Class.members(actualClass, method);
   find parentContainsSomething+(method, InterruptingStatement);
}

private pattern startsActivityOrToast(interruptingExpression) {
   // does the interrupting expression refer the Toast class?
   ExpressionStatement.expression(interruptingExpression, toastInstance);
   IdentifierReference.target.name(toastInstance, "Toast");
   // is the makeText method invoked?
   IdentifierReference.next(toastInstance, callsMakeText);
   MethodCall.target.name(callsMakeText, "makeText");
   // is the Toast finally shown?
   MethodCall.next(callsMakeText, showToastExpression);
   MethodCall.target.name(showToastExpression, "show");
} or {
   ExpressionStatement.expression(interruptingExpression,
       startActivitiyMethod);
   MethodCall.target.name(startActivitiyMethod, "startActivity");
}

private pattern isServiceOrBroadcastReciever(class) {
   find isClassOf(class, "Service");
} or {
   find isClassOf(class, "BroadcastReciever");
}

private pattern isClassOf(class, className) {
   Class.name(class, className);
}

private pattern parentContainsSomething(parent, child){
   LocalVariableStatement.variable(parent, child);
} or {
   StatementListContainer.statements(parent, child);
}
```

Listing 7.2: *Interruption from background* pattern.

IncPL patterns are specified by means of the metaclasses of the metamodel the queried model conforms to; in this case it is the Java metamodel provided by JaMoPP. Furthermore, patterns are specified in dot notation starting from metaclasses along their references to other metaclasses. This can be seen in Line 3. Such a *query statement* always has two parameters. The first one represents the source of the query and the second represents the target. In Line 3, `actualClass` corresponds to the `Class` instance at the beginning of the line whereas `superClassRef` corresponds to the target which is reached by navigation from the `actualClass` by means of the `extends` reference of the metamodel. As can be seen in Line 4, the parameter `superClassRef` is reused as the source parameter corresponding to an instance of `NamespaceClassifierReference`. Furthermore, IncPL supports *pattern statements* as can be seen in Line 6. Such statements allow for the reuse of other patterns which again can accept suitable parameters. This technique of passing parameters from one statement to another (regardless of being a query or pattern statement) ensures that the desired types correspond to the expectations. This was a small introduction to the syntax of IncPL. In the following the concrete pattern is explained in more detail.

Since we are interested in the statement which causes this quality smell, the only parameter (corresponding to the role of interest `InterruptingStatement`) of the pattern is of type `ExpressionStatement`. This parameter is bound as a role of interest. The detection is split into three parts. First, in Lines 3–6, it is checked if the actual class extends or `BroadcastReceiver` and `Service`. Second, it is determined in Line 9, if interrupting methods are invoked. Third, in Lines 12–13, the specification ensures that the pattern only matches if the interrupting code is executed in a method contained in the actual class.

For all of these three parts, particular sub-patterns are invoked via pattern statements. According to the example in Listing 7.1, only the first part of the sub-pattern *startsActivityOrToast* is explained in more detail. In this sub-pattern, first it is checked if the any invoked method refers to the `Toast` class. Therefore, in Line 18, we get the `expression` of the passed `interruptingExpression` and ensure in Line 18 that it is an instance of `IdentifierReference` which is named `Toast`. In Lines 21 and 22, it is checked that the `toastInstance` calls the method `makeText`. Finally, this sub-pattern checks in Lines 24 and 25 if the `show` method is really invoked on the `callsMakeText` result.

This part of the whole pattern matches the quality smell shown in Listing 7.1. The remaining part of this pattern is conceptually similar and is omitted here.

Refactorings: *Introduce Notification*

To inform the user that attention is needed the `InterruptingStatement` should be replaced by a `Notification`. The Listing 7.3 shows the result of this refactoring.

```
1 public class InterruptingService extends Service {
2    public IBinder onBind(Intent intent) {return new Binder();}
3    public void onCreate() {
4       super.onCreate();
5       Notification notification = new Notification.Builder(this).
            setContentText("Hello World!").build();
6       NotificationManager manager = (NotificationManager) getSystemService(
            Context.NOTIFICATION_SERVICE);
```

```
7         manager.notify(123, notification);
8     }
9 }
```

Listing 7.3: Notifying service.

With this resolution users can continue working without interruption since the message is only displayed in the notification area of the device. We consider this modification a refactoring since the information to be displayed remains the same.

References:
`http://developer.android.com/guide/practices/seamlessness.html#interrupt` (visited 3rd March 2015)
`http://developer.android.com/guide/topics/ui/notifiers/toasts.html` (visited 3rd March 2015)
`http://developer.android.com/guide/topics/ui/notifiers/notifications.html` (visited 3rd March 2015)

Related Quality Smells: Dropped Data (Sect. 7.4.2)

7.4. Quality Smells for Android Applications

In this section the remaining quality smell catalogue for Android applications is presented. It is an excerpt of the mined quality smells presented in [Bry14]. In contrast to [Bry14], only those quality smells are presented here for which IncPL patterns could be specified. The according IncPL patterns are omitted here and can be seen in Appendix D. The *name* property of the catalogue schema is omitted, since it corresponds to the section headings.

7.4.1. Quality Smell: Data Transmission Without Compression

Context: Implementation, Network

Affected Qualities: Energy Efficiency

Roles of Interest: `FileBodyConstructor`

Description: In [HB10], Höpfner and Bunse discussed that transmitting a file over a network infrastructure without compressing it consumes more energy than with compression. More precisely, energy efficiency is improved in case the data is compressed at least by 10 %, transmitted and decompressed at the other network node. The example in Listing 7.4 shows file transmission implemented with the Apache HTTP Client Library.[17]

```
1 public static void main(String[] args) throws Exception {
2     HttpClient httpclient = new DefaultHttpClient();
3     HttpPost httppost = new HttpPost("http://some.url:8080/servlets-examples
          /servlet/RequestInfoExample");
4     FileBody bin = new FileBody(new File(args[0]));
5     StringBody comment = new StringBody("A␣binary␣file");
```

[17] The example was taken and adapted from `http://archive.apache.org/dist/httpcomponents/httpclient/binary/httpcomponents-client-4.2.4-bin.zip` (visited 3rd March 2015).

```
6     MultipartEntity reqEntity = new MultipartEntity();
7     reqEntity.addPart("bin", bin);
8     reqEntity.addPart("comment", comment);
9     httppost.setEntity(reqEntity);
10    System.out.println("executing␣request␣" + httppost.getRequestLine());
11    HttpResponse response = httpclient.execute(httppost);
12    HttpEntity resEntity = response.getEntity();
13    EntityUtils.consume(resEntity);
14 }
```

Listing 7.4: File transmission without compression before refactoring.

In line 4 one can see that the passed `File` object in this constructor (role `FileBodyConstructor`) is transmitted without compression.

IncPL Pattern: Listing D.1 in Appendix D

Refactorings: *Add Data Compression to Apache HTTP Client based file transmission*
This refactoring adds a compression method. Then it passes the `File` parameter of the constructor (role `FileBodyConstructor`) in Line 4 of Listing 7.4 to this method. Thus, the file is transmitted with compression. In Listing 7.5 the result of this refactoring is depicted. Line 3 contains the invocation of the compression method `gzipFile(File uncompressedFile)` which is added by this refactoring.

```
1  public static void main(String[] args) throws Exception {
2     // ...
3     FileBody bin = new FileBody(gzipFile(new File(args[0])));
4     // ...
5  }
6  private static File gzipFile(File uncompressedFile){
7     File gzFile = File.createTempFile(file.getName(), "gz");
8     FileInputStream fis = new FileInputStream(file);
9     GZIPOutputStream out = new GZIPOutputStream(new FileOutputStream(gzFile)
          );
10    byte[] buffer = new byte[4096];
11    int bytesRead;
12    while ((bytesRead = fis.read(buffer)) != -1){
13       out.write(buffer,0, bytesRead);
14    }
15    fis.close();
16    out.close();
17    return gzFile;
18 }
```

Listing 7.5: File transmission with compression after refactoring

References: [HB10]

Related Quality Smells: Durable `WakeLock` (Sect. 7.4.3), Rigid `AlarmManager` (Sect. 7.4.6)

7.4.2. Quality Smell: Dropped Data

Context: UI

Affected Qualities: User Experience, User Conformity

Roles of Interest: `DataDroppingClass`

Description: The user can input or edit data in an Android `Activity` or `Fragment`. Imagine another `Activity` pops up (e.g., an incoming phone call) and interrupts the user. After returning to the former `Activity` the input is lost, but the user expects the data to be persisted. This happens if the class bound to the role of interest `DataDroppingClass` (`Activity` or `Fragment`) does not implement the methods `onSaveInstanceState(Bundle)` and `onRestoreInstanceState(Bundle)`.

IncPL Pattern: Listing D.2 in Appendix D

Refactorings: *Save and Restore Instance State*
The developer has to ensure that the state of the `Activity` or `Fragment` (role `DataDroppingClass`) is stored, when the user entered data. This is done in the `onSaveInstanceState(Bundle)` method. It can be restored by overriding `onRestoreInstanceState(Bundle)`. Therefore, this refactoring adds a skeleton for each of these methods if not present. For default widgets, this is already done by the framework. Thus, one should not miss to call `super.onSaveInstanceState(Bundle)` and `super.onRestoreInstanceState(Bundle)` respectively in the corresponding methods.

References:
`http://stackoverflow.com/questions/151777/saving-activity-state-in-android` (visited 3rd March 2015)
`http://developer.android.com/guide/practices/seamlessness.html#drop` (visited 3rd March 2015)
`http://developer.android.com/reference/android/app/Activity.html` (visited 3rd March 2015)

Related Quality Smells: Durable `WakeLock` (Sect. 7.4.3), Interruption From Background (Sect. 7.3.2)

7.4.3. Quality Smell: Durable WakeLock

Context: UI, Implementation

Affected Qualities: Energy Efficiency

Roles of Interest: `TimeoutLessAcquire`

Description: A `WakeLock` is acquired in order to indicate that the application requires the device to stay activated. This is needed, e.g., when CPU, Sensors or GPS should be interacted with explicitly. After using the resources, the application should release the `WakeLock`. If this is not done the battery power will drain. This comprises several aspects. First, it must be ensured that the `release()` method is invoked on the `WakeLock` object. Second, the method `aquire(long timeout)` should be used to acquire the `WakeLock`

instead of invoking `aquire()` (the call to this method is bound to the role of interest `TimeoutLessAcquire`). This ensures that the lock is released under all circumstances, since a time-out is specified. An example is shown in Listing 7.6. In Line 6, the `WakeLock` is acquired without time-out and without releasing it afterwards.

```
1 protected void onCreate(Bundle savedInstanceState) {
2    super.onCreate(savedInstanceState);
3    PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE)
         ;
4    WakeLock wl = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK
5          | PowerManager.ON_AFTER_RELEASE, "TAG");
6    wl.acquire();
7 }
```

Listing 7.6: Acquiring a `WakeLock` without releasing it.

IncPL Pattern: Listing D.3 in Appendix D

Refactorings: *Aquire WakeLock with time-out*

To ensure that the `WakeLock` will be released in all circumstances, the method `PowerManager.WakeLock.aquire(long timeout)` replaces the acquisition without time-out (`TimeoutLessAcquire`). Furthermore, the `release()` method is added to the end of the method which ensures that the `WakeLock` is released in case the work finishes before the time-out. The result of this refactorings can be seen in Listing 7.7.

```
1 protected void onCreate(Bundle savedInstanceState) {
2    super.onCreate(savedInstanceState);
3    PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE)
         ;
4    WakeLock wl = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK
5          | PowerManager.ON_AFTER_RELEASE, "TAG");
6    wl.acquire(60*1000*10); // auto release it in 10 minutes
7    // ... do work...
8    wl.release();
9 }
```

Listing 7.7: Acquiring a `WakeLock` with time-out.

References: [Gui12]

`http://developer.android.com/reference/android/os/PowerManager.WakeLock.html` (visited 3rd March 2015)

Related Quality Smells: Data Transmission Without Compression (Sect. 7.4.1), Dropped Data (Sect. 7.4.2), Rigid `AlarmManager` (Sect. 7.4.6)

7.4.4. Quality Smell: Internal Use of Getters/Setters

Context: Implementation

Affected Qualities: Performance

Roles of Interest: `CallOfGetter`

Description: The internal fields are accessed from the owning class via its getters and setters. But the Android Performance Tips[18] say:

> "Without a JIT, direct field access is about 3x faster than invoking a trivial getter. With the JIT (where direct field access is as cheap as accessing a local), direct field access is about 7x faster than invoking a trivial getter."

To increase performance, the use of internal getters (role of interest `CallOfGetter`) and setters should be avoided.

IncPL Pattern: For one part of this quality smell we show the according IncPL pattern here. Listing 7.8 depicts the detection of accessing an internal getter. The returned `CallOfGetter` conforming to metaclass `MethodCall` then is passed to the refactoring (Line 1).

```
1  pattern internalGetter(CallOfGetter:MethodCall) {
2     find isGetter(getter);
3     MethodCall.target(CallOfGetter,getter);
4  }
5
6  private pattern isGetter(actualMethod:ClassMethod) {
7     // return value of method is a field
8     ClassMethod.statements(actualMethod,statemets);
9     Return.returnValue(statemets,ref);
10    IdentifierReference.target(ref,variable);
11    Variable.name(variable,varName);
12    Class.members.name(actualClass,varName);
13
14    // method is contained in the same class as the field
15    Class.members(actualClass,actualMethod);
16 }
```

Listing 7.8: IncPL pattern for detecting access of internal getters.

Refactorings: *Introduce Direct Field Access*
Replaces the access to internal getters (role of interest `CallOfGetter`) and setters by a direct access.

References:
`http://stackoverflow.com/questions/6716442/android-performance-avoid-internal-getters-setters` (visited 3rd March 2015)
`http://developer.android.com/training/articles/perf-tips.html#GettersSetters` (visited 3rd March 2015)

Related Quality Smells: Unclosed `Closeable` (Sect. 7.4.7)

[18] `http://developer.android.com/training/articles/perf-tips.html#GettersSetters` (visited 3rd March 2015)

7.4.5. Quality Smell: No Low Memory Resolver

Context: Implementation

Affected Qualities: Memory Efficiency, Stability, User Experience

Roles of Interest: `ClassWithoutMemoryResolver`

Description: Mobile systems usually have little main memory and no physical space to store data in order to free some memory. Android provides a mechanism to support the system in managing memory. The method `Activity.onLowMemory()` is called when the system is running low on memory. This method should be implemented to clean caches or unnecessary resources. Subclasses of `Activity` not implementing it are bound to the role of interest `ClassWithoutMemoryResolver`.

IncPL Pattern: Listing D.5 in Appendix D

Refactorings: *Override onLowMemory()*
This refactoring adds a skeleton of the the method `onLowMemory()` to a subclass of `Activity` which is bound to the role of interest `ClassWithoutMemoryResolver` in order to override it when it is not present. The actual implementation of this method must be provided by the developer.

References: [Gui12]
`http://developer.android.com/reference/android/app/Activity.html` (visited 4th March 2015)

Related Quality Smells: Unclosed `Closeable` (Sect. 7.4.7)

7.4.6. Quality Smell: Rigid AlarmManager

Context: Implementation

Affected Qualities: Energy Efficiency, Performance

Roles of Interest: `RigidCaller`

Description: In Android, with the use of `AlarmManager` it is possible to schedule an application to be run at a certain time in future. When this time is reached the system automatically starts the application if not yet running. Such an alarm is scheduled by the method `setRepeating(int, long, long, PendingIntent)` on an `AlarmManager` object. Every application which is triggered by an `AlarmManager` wakes up the mobile device and thus the overall energy consumption and CPU might be higher, then if bundled together. In order to let the system bundle scheduled alarms the method `setRepeating(int, long, long, PendingIntent)` (method call is bound to role of interest `RigidCaller`) should be replaced by a call to `setInexactRepeating(int, long, long, PendingIntent)` Listing 7.9 shows an example.

```
1  public class RigidAlarmManagerTest extends Activity {
2     protected void onCreate(Bundle savedInstanceState) {
3        super.onCreate(savedInstanceState);
4        AlarmManager am = (AlarmManager) getSystemService(Context.
             ALARM_SERVICE);
5        Intent intent = new Intent(this,
             InterruptingFromBackgroundServiceTest.class);
6        PendingIntent pendingIntent = PendingIntent.getService(this,0, intent
             ,0);
7        long interval = DateUtils.MINUTE_IN_MILLIS * 30;
8        long firstWake = System.currentTimeMillis() + interval;
9        am.setRepeating(AlarmManager.RTC_WAKEUP, firstWake, interval,
             pendingIntent);
10    }
11 }
```

Listing 7.9: Use of an exact `AlarmManager` which results in higher energy consumption.

IncPL Pattern: Listing D.6 in Appendix D

Refactorings: *Introduce Use of Inexact Alarmmanager*
This refactoring replaces the invocation of `setRepeating(int, long, long, PendingIntent)` (role of interest `RigidCaller`) by a call to `setInexactRepeating(int, long, long, PendingIntent)` in order to ensure that the system bundles several alarms together. In Listing 7.10, the example from Listing 7.9 is shown after the refactoring.

```
1  protected void onCreate(Bundle savedInstanceState) {
2     // initialise...
3     am.setInexactRepeating(AlarmManager.RTC_WAKEUP, firstWake, interval,
          pendingIntent);
4  }
```

Listing 7.10: Use of an inexact `AlarmManager` reducing energy consumption

References:
`http://developer.android.com/reference/android/app/AlarmManager.html` (visited 4th March 2015)

Related Quality Smells: Data Transmission Without Compression (Sect. 7.4.1), Durable `WakeLock` (Sect. 7.4.3)

7.4.7. Quality Smell: Unclosed Closeable

Context: Implementation

Affected Qualities: Memory Efficiency

Roles of Interest: `UnclosedHolder`, `UnclosedParameter`

Description: An object implementing the `Closeable` interface is not closed which results in higher memory consumption. The role of interest `UnclosedHolder` is bound to a method

having a parameter of type `Closeable`. If the `close()` method is not invoked on this parameter it is bound to the role of interest `UnclosedParameter`. Listing 7.11 shows an example.

```
public static void close(Closeable closed, Closeable unclosed) {
   if (closed != null) {
      try {
         closed.close();
      } catch (Exception e) {
         System.out.println("Unable to close %s");
      }
   }
}
```

Listing 7.11: A `Closeable` object is not closed.

IncPL Pattern: Listing D.7 in Appendix D

Refactorings: *Close Closable*
This refactoring adds a call to the `close()` method of the unclosed object (`UnclosedParameter`) as can be seen in Listing 7.12. This call is added to the method which is bound to the role of interest `UnclosedHolder`.

```
public static void close(Closeable closed, Closeable unclosed) {
   // do something
   unclosed.close();
}
```

Listing 7.12: `Closeable` object is closed.

References: [Gui12]
`http://developer.android.com/reference/java/io/Closeable.html` (visited 4th March 2015)

Related Quality Smells: Internal Use of Getters/Setters (Sect. 7.4.4), No Low Memory Resolver (Sect. 7.4.5)

7.4.8. Quality Smell: Untouchable

Context: User Experience, Accessibility, User Expectation

Affected Qualities: UI

Roles of Interest: `SmallerConstructorCall`

Description: Visual elements in a mobile application, such as buttons, should be at least 48dp[19] of size in order to ensure accessibility. If they are smaller, it is hard to touch them. In case a layout of a visual element is specified programmatically, this can be checked statically. In this case smaller values are passed to the constructor of `LayoutParams` which then is bound to the role of interest `SmallerConstructorCall`. An example is depicted in Listing 7.13.

[19] Density-independent Pixels

```
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   Button myButton = new Button(this);
   RelativeLayout myLayout = new RelativeLayout(this);
   RelativeLayout.LayoutParams params = new RelativeLayout.LayoutParams(20,
        60);
   myLayout.setLayoutParams(params);
   myLayout.addView(myButton);
   setContentView(myLayout);
}
```

Listing 7.13: A button which is not touchable.

IncPL Pattern: Listing D.8 in Appendix D

Refactorings: *Increase Touchable Size*

This refactoring asks the user for a new value of the according parameter of the layout and passes it to the constructor (role of interest `SmallerConstructorCall`). An exemplary result can be seen in Listing 7.14.

```
protected void onCreate(Bundle savedInstanceState) {
   // initialise
   RelativeLayout.LayoutParams params = new RelativeLayout.LayoutParams(60,
        60);
   // do other things
}
```

Listing 7.14: A layout with appropriate size.

References:

`http://android-developers.blogspot.de/2012/04/accessibility-are-you-serving-all-your.html` (visited 4th March 2015)
`http://developer.android.com/design/style/metrics-grids.html#48dp-rhythm` (visited 4th March 2015)

Related Quality Smells: Interruption From Background (Sect. 7.3.2)

7.5. Discussion

In this chapter, a quality smell catalogue has been presented. Therefore, an underlying schema was declared and our process of acquiring quality smells from a broad information base was illustrated. Afterwards, the conceptual framework from Sect. 6.3 was instantiated for structural-based quality smells in the domain of mobile Android applications in order to fulfil Requirement 3.

When a catalogue regarding a specific domain is provided, the question of completeness arises. We state that our catalogue is by no means complete, but in Sect. 7.2 a strategy is presented how further quality smells can be mined. This strategy is generic and thus can be applied to other domains, as well. The focus of our presented catalogue is on the potential of detection and resolution of quality smells in practice. This means that the catalogue contains only quality smells having a precisely defined pattern for detection and an implemented resolving refactoring. The catalogue can be considered as a basis for ongoing work.

8

Role-Based Co-Refactoring in Multi-Language Development Environments

In this chapter, we provide a new co-refactoring approach incorporating into our overall framework. It benefits from the concept of role models. Part of the approach presented here is covered by our joint work published in "Language-Independent Traceability with Lässig" [PRW14].

8.1. Motivation

Multi-Language Development Environments (MLDEs) are environments with a large number of heterogeneous artefacts [Pfe13]. However, all artefacts can be perceived as models provided that a suitable metamodel exists — in our case conforming to MOF. This encompasses analysis artefacts, which may be refined in a MDA process; code artefacts, which are the outcome of such a process; or other assets such as configuration or documentation files. A comprehensive example is provided in Sect. 8.2. The essence of MLDEs is that the set of metamodels to which the models conform is not fixed. Thus, the participating models can be instances of arbitrary DSLs or metamodels.

Within MLDEs, a large number of models with interdependencies exists. When one model is refactored, it is very likely that related models are influenced by this change and have to be updated in order to maintain consistency. Thus, refactorings may affect dependent models so that the changes have to be propagated as well.

We consider the combination of the initial refactoring with the consistency-preserving modification of dependent models as *co-refactoring*. Such subsequent co-refactorings may have to be performed over multiple stages and branches, which is illustrated in Fig. 8.1. In case a refactoring is initiated in one model, it might affect dependent models on the next stage. To preserve consistency, co-refactorings have to be applied in these models which, in turn, might influence adjacent models again and again. The process of propagating the initial changes to dependent models

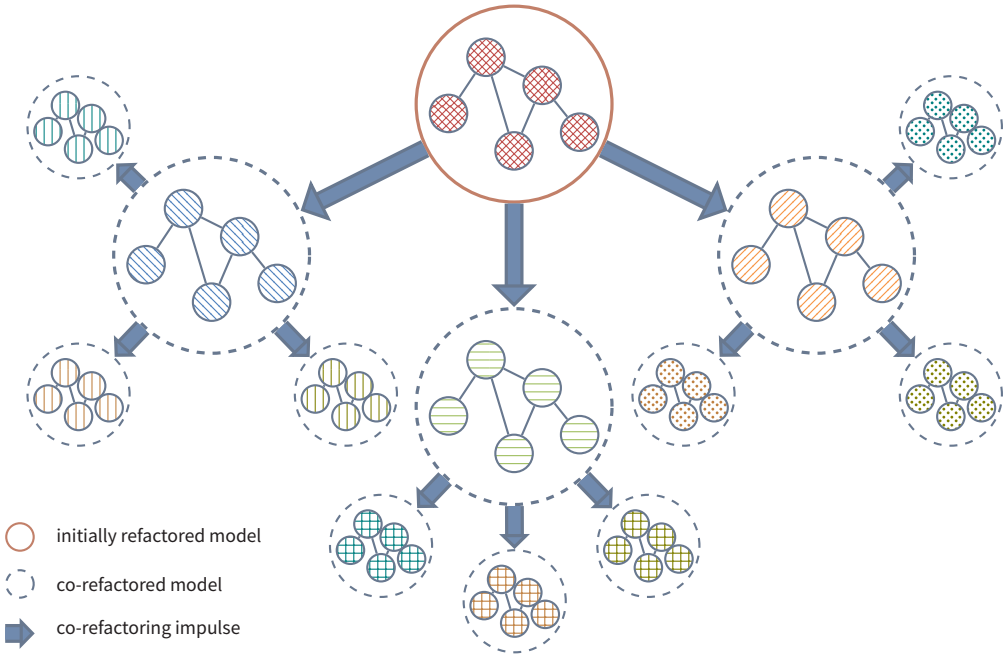


Figure 8.1.: Extract of the successive progression of model co-refactoring. Different colours and patterns in the circles representing the model elements indicate different metamodels.

over multiple stages can be very complex. In this sense, a network of dependency relations should be built providing assistance during the co-refactoring process.

In the following, we will regard the propagation of co-refactorings along the network of dependencies as *refactoring stream*. The choice of using the term *stream* has been motivated from the flow of water starting from a particular source. Initially, the path of the stream is undefined, might get split into branches which flow into the direction of the lowest resistance. The refactoring stream behaves similarly. The initially refactored model is the source of the stream. In the beginning, the path to go is undefined. A co-refactoring impulse is caused for each subsequent dependent model in the network of dependency relations. Thus, the impulse is split up, in order to reach the dependent models. Therefore, we decided to use the term *refactoring stream* for the flow of the co-refactoring impulse across the network of dependent models.

As a consequence, each stage of the refactoring stream underlies the schematic workflow in Fig. 8.2. While determining dependent models, a knowledge base providing dependency information must be accessed, since most often dependencies between models are implicit and they do not know each other. In the following, we will call this knowledge base *Dependency Knowledge Base (DK-Base)*. Furthermore, another information provider regarding the application of co-refactorings is needed. This *Co-Refactoring Knowledge Base (CoRK-Base)* is required to determine which subsequent co-refactoring should be executed as a consequence of a preceding refactoring. As a result of this observation, these constituents of the schematic co-refactoring workflow approaches will be discussed in the next two sections.

As discussed in Sect. 3.3.2, no universal approaches exist on how to derive co-refactoring steps in adjacent models in consequence to the initial refactoring of another model. As a consequence,

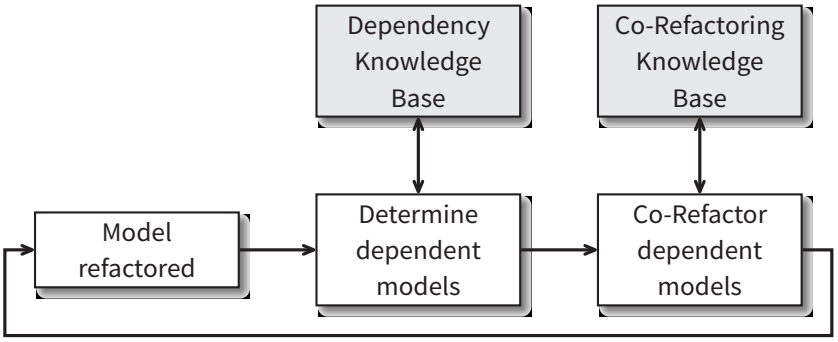


Figure 8.2.: Schematic workflow of Co-Refactoring.

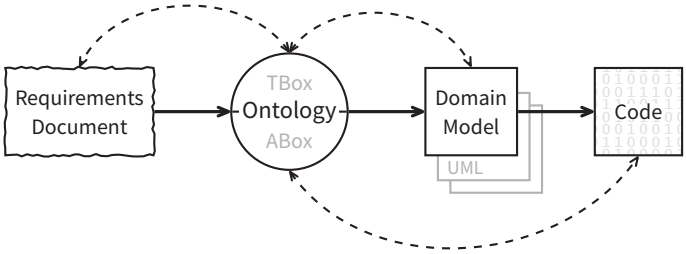


Figure 8.3.: Example of an ontology-driven requirements and software engineering process. Solid lines indicate the direction of the data flow. Dashed lines indicate interaction controlled by the ontology.

we see a strong risk that users of MLDEs have to be involved too heavily in the co-refactoring process, which can be both tedious and error-prone. Therefore, we present a co-refactoring approach for models in MLDEs in this chapter. Prior to the presentation of our approach, we will come up with a comprehensive example in the next section.

8.2. Example

As already indicated in Sect. 1.1, dependent models appear right from the beginning of software development in form of requirements documents. Furthermore, approaches exist to guide through the requirements specification process based on ontologies [STZ+11]. Herein, ontologies are used to enrich the whole process with additional information being used for consistency checking with respect to the fulfilment of requirements along the development process. Besides that, ontologies can also be used to derive other artefacts such as domain models [HS06]. In this scenario, the ontology is taken into account as some kind of intermediate artefact for generating others, although ontologies have a higher degree of expressiveness. Additional information in the ontology then is used for further reasoning [STZ+11]. Such an exemplary scenario is illustrated in Fig. 8.3. As can be seen, we have heterogeneous dependent models and changes in one model can cause changes in a dependent model. Thus, we decided to take an example from the domain of ontology-driven requirements and software engineering.

According to [STZ+11], not only the requirements specification phase is backed up by an ontology but the whole development process. Thus, the ontology in Fig. 8.3 is present right from the beginning. It defines consistency constraints according to the requirements specification, for instance, for verifying that every requirement is realised by a certain software component or that requirements do not contradict each other. As a consequence, the requirements document is the first artefact created in such a development process. Controlled by the ontology, domain information can be extracted from the requirements [HS06; KKK+06]. As a result, the gathered domain information can be made explicit in new artefacts. Such artefacts might be, e.g., generated UML models. As a next step, code artefacts then can be generated from domain models. Since the ontology plays a central role in this process, it fulfils the function of a controller. It interacts with all the other artefacts and gets populated iteratively.

New information which can be reasoned with the ontology can be, for instance, that a test case proves correctness of a certain software component, or that a state chart assures the protocol of a particular requirement [STZ+11]. Hence, the ontology is a controlling specification from which subsequent artefacts are generated. If a refactoring is applied in the ontology [BS06], subsequent co-refactorings have to be propagated to the dependent artefacts. This example serves as basis for explanation in the following. In the next section, we first clarify that different kinds of dependencies between models in MLDEs exist and categorise them.

8.3. Dependency Knowledge Base

In the process of co-refactoring, an initial refactoring of a model prompts for subsequent modifications of dependent models in the refactoring stream. As a prerequisite for this procedure, all relevant model dependencies need to be determined. In the following, we provide a classification of different types of dependencies encountered in MLDEs and emphasize a correspondence to our example from the previous section.

8.3.1. Categories of Model Dependencies

In general, dependencies between models can be *explicit* or *implicit*. We consider an explicit dependency if there is a reference from one model to another model according to the metamodel of the former. Thus, a physical connection between these models exists. An implicit dependency exists if two models are related to each other logically and have no explicit relation. The following presented categories are classified either into explicit or implicit dependencies. We distinguish four different kinds of dependencies between models as depicted in Fig. 8.4. These categories have emerged by analysis of typical connections between models from which we can benefit in the co-refactoring scenario.

Direct dependency of a model *A* to another model *B* is established if there is an explicit reference from model *A* to model *B*.

In our example, direct dependencies occur manifoldly. On the one hand, direct dependencies exist between models of the same language. A UML sequence diagram model, e.g., can reference classes or methods from a UML class diagram model. A Java class can reference other Java classes by means of a superclass relation. On the other hand, direct dependencies

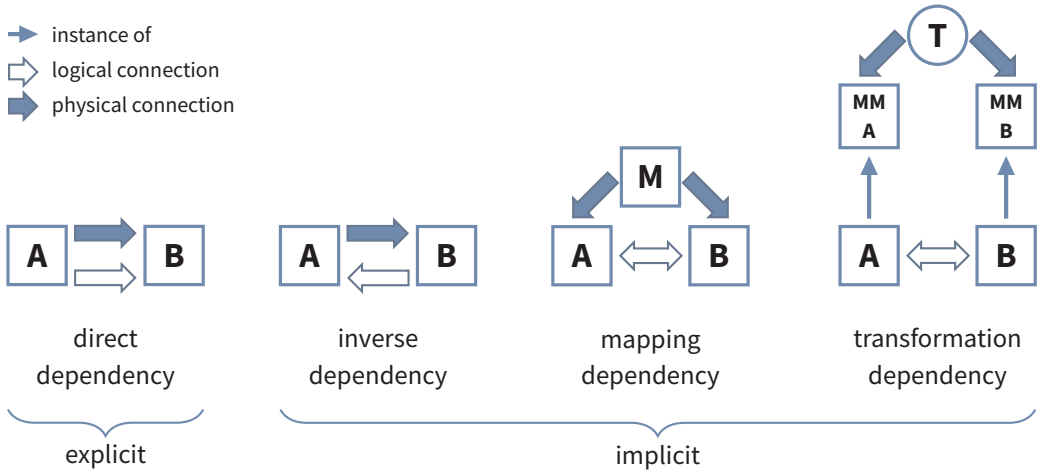


Figure 8.4.: Categories of model dependencies. In explicit dependencies, physical and logical connections are aligned.

also exist between models of different languages. A direct dependency occurs between the ontology and source code by means of a *hasSource* relation according to [Sie14].

Inverse dependency from model *A* to model *B* exists if a model *A* is referenced by another model *B* but is itself unaware of the connection. An inverse dependency can be perceived as the opposite of a direct dependency.

Thus, an inverse dependency exists, e.g., from a superclass to its subclass in Java, or from a test case to the ontology according to the opposite of the *hasSource* relation [Sie14].

Mapping dependency exists between two models *A* and *B* if an external mapping model relates elements of model *A* to those of model *B*. More than two models may participate in a mapping dependency.

In the example, the ontology serves as a mapping model. It maps, e.g., requirements to source code according to the *hasTestCase* relation. Furthermore, the ontology maps requirements to UML use cases by means of the *isDescribedBy* relation [Sie14].

Transformation dependency between two models *A* and *B* is established if specific concepts of model *B* can be obtained by applying particular rules of a transformation *T* to concepts of model *A*. The transformation *T* is specified with respect to the metamodels *MM A* and *MM B* which the models *A* and *B* respectively conform to.

Again, our example contains several of transformation dependencies. Since domain models can be generated from the ontology [HS06], a transformation dependency exists between both. Such dependencies also exist for UML models and generated code.

Direct dependencies are explicit, as physical and logical connections are aligned and the relation to other models can be established by solely inspecting the original model. On the other hand, the categories of inverse, transformation and mapping dependencies are implicit, as their physical and logical connections are not aligned. Thus, the logical connection cannot be

determined directly from the inspected models. Instead, the entirety of models in the system has to be analysed in order to establish these dependencies.

In contrast to the characterisation of MLDEs presented in [PW15], our categories have a more technical background. The reason for this decision is that these categories are directly reflected in the upcoming presentation of the detection procedure. Nevertheless, the categories presented above can be classified into the *relation types* of [PW15], although the authors consider relations between strings only. Direct dependencies can be considered as *fixed relations*, since Pfeiffer and Wąsowski define them as an established link between equal strings. A fixed relation is undirected. Hence, an inverse dependency also corresponds to a link between two elements and is covered by the fixed relation type as well. A mapping dependency from a mapping model M to the models A and B can be considered as two fixed relations between M-A and M-B respectively. The logical connection between A and B is not covered in [PW15]. Our transformation dependency corresponds to a *String-transformation relation*. Since we consider not only strings, our dependency category has a broader scope. Furthermore, all of our categories are *Domain-Specific Relations* according to [PW15], since all of them are typed. Each element participating in a dependency has a certain type in terms of a metaclass.

Before dependent models can be co-refactored, those dependent models must be detected. For this reason, we discuss the right point in time of determination in the following before the according detection procedure is illustrated.

8.3.2. When to Determine Model Dependencies

As explained previously, direct dependencies are explicit between different models and can be located by getting the information from the relation specified in the models. In contrast, implicit dependencies must be determined with dedicated mechanisms. We now discuss when model dependencies have to be determined, because this decision heavily influences performance and memory usage.

The first possibility is to analyse each model for explicit and implicit dependencies at the time it enters the MLDE. The term *entering* subsumes the creation of a new model until it is persisted and the action of moving an existing model into the MLDE. Then the acquired information can be stored in the DK-Base. The advantage is that all dependencies are readily available, when dependent models must be determined for a refactored model and can be retrieved directly from the DK-Base. However, there are some limitations: The approach requires large amounts of physical storage for the DK-Base and it stores another representation of the models and their dependencies. Furthermore, the additional representation needs to be maintained when the model dependencies change.

The second possibility is to calculate model dependencies on demand when an initial refactoring of a model was applied. The advantage of this approach is that no persistent storage is required and that no additional representation of the model dependencies must be maintained. However, traversing all models (having a graph-like structure) on demand is very expensive in terms of processing time.

As a consequence, we exploit some properties of both previously described approaches to take advantage of them. According to the schematic workflow in Fig. 8.2, our DK-Base is populated only with explicit dependencies. These dependencies are persisted for each model in the moment, when it enters the MLDE. Using this approach, explicit knowledge about the system

```
1 explicit(sourceElement, reference, targetElement).
2
3 elementtoresourcemapping(modelElement, model).
```

Listing 8.1: Prolog fact pattern to capture explicit dependencies.

is built incrementally and implicit dependencies are calculated on demand, when a model is refactored and dependent models are to be determined. Unlike in the first approach, the effort for maintaining the additional representation is acceptable because only the explicit dependencies are stored in the DK-Base. DK-Base can be updated easily when direct dependencies change, because for each model only one corresponding rule must be updated. In contrast to the second approach, at no point in time, the whole MLDE must be traversed to determine implicit dependencies, because they can be calculated using stored knowledge.

8.3.3. How to Determine Model Dependencies

The results of the previous discussion about when to determine model dependencies have implications on our approach. First, it is obvious that there is a need for storing the explicit dependencies. Second, a formalism which enables to specify rules for the categories is required, which can be interpreted at the time a refactoring occurs initially. The set of rules must be extensible at the time the MLDE is running. Therefore, the rules should be declaratively specified.

We chose an approach based on logical programming to determine inverse and mapping dependencies, since it is declarative and easily extensible. Furthermore, there exists an isomorphism between graphs and logics [Cou90; Ren04]. Thus, both formalisms can be mapped to each other bijectively. Due to this background, representing models as logical facts has already been applied successfully [PS92; HCW07; Şav10]. Therefore, we have chosen Prolog to be able to specify and present the following rules precisely.

When a model enters the MLDE, all its explicit references are added as facts to the DK-Base. For every explicit dependency, facts according to the patterns in Listing 8.1 are added. The `sourceElement` in one model `references` the `targetElement` in another model (Line 1). Furthermore both for source and target elements a fact is created to indicate in which `models` they reside (Line 3). Every time a model enters the MLDE, such facts are generated and added to the DK-Base.

For representing the category of inverse dependencies, we add the rules in Listing 8.2 to the DK-Base. The intrinsic rule which represents the meaning of the inverse dependency category can be seen in Line 1. This rule reveals an inverse logical connection between a `SourceElement` and a `TargetElement` if an explicit reference from the `TargetElement` to the `SourceElement` exists. The rule in Line 4 classifies an inverse dependency as being implicit. Thus, a `SourceElement` and a `TargetElement` are implicitly dependent via a `Reference` if there is an inverse dependency between them. The rule in Line 7 provides means to ask for all implicit references from elements in a `SourceModel` to a specific `TargetElement`. The rule in Line 11 asks for all `SourceElements` having an implicit dependency to elements in a particular `TargetModel`. The latter two rules provide different possibilities for querying the DK-Base.

The rules for determining mapping dependencies are depicted in Listing 8.3. The intrinsic rule which represents the meaning of the mapping dependency category can be seen in Line 1. The

```
inverseDependency(SourceElement,Reference,TargetElement) :-
   explicit(TargetElement,Reference,SourceElement).

implicit(SourceElement,Reference,TargetElement) :-
   inverseDependency(SourceElement,Reference,TargetElement).

implicit(SourceModel,Reference,TargetElement) :-
   elementtoresourcemapping(SourceElement,SourceModel),
   inverseDependency(SourceElement,Reference,TargetElement).

implicit(SourceElement,Reference,TargetModel) :-
   elementtoresourcemapping(TargetElement,TargetModel),
   inverseDependency(SourceElement,Reference,TargetElement).
```

Listing 8.2: Prolog rules to determine inverse dependencies.

```
mappingDependency(SourceElement,TargetElement) :-
   elementtoresourcemapping(SourceElement,SourceModel),
   elementtoresourcemapping(TargetElement,TargetModel),
   not(SourceModel = TargetModel),
   explicit(MappingElementLeft,_,SourceElement),
   explicit(MappingElementRight,_,TargetElement),
   elementtoresourcemapping(MappingElementLeft,MappingModel),
   elementtoresourcemapping(MappingElementRight,MappingModel),
   not(MappingModel = SourceModel),
   not(MappingModel = TargetModel).

implicit(SourceElement,_,TargetElement) :-
   mappingDependency(SourceElement,TargetElement).

implicit(SourceModel,_,TargetElement) :-
   elementtoresourcemapping(SourceElement,SourceModel),
   mappingDependency(SourceElement,TargetElement).

implicit(SourceElement,_,TargetModel) :-
   elementtoresourcemapping(TargetElement,TargetModel),
   mappingDependency(SourceElement,TargetElement).
```

Listing 8.3: Prolog rule to determine mapping dependencies.

goals in Lines 2–4 assure that `SourceElement` and a `TargetElement` are contained in distinct models. Then, Lines 5 and 6 ask for some elements `MappingElementLeft` and `MappingElementRight` having an explicit dependency to `SourceElement` and `TargetElement`. Lines 7 and 8 ensure that both the left and the right mapping elements are contained in the same `MappingModel`. Finally, it is checked if all three models are distinct in Lines 9 and 10. Similar to the inverse dependency category, the rules in Lines 12–19 classify a mapping dependency as being implicit and provide means for querying models and their dependent elements, or elements and their depending models, respectively.

Until now, the DK-Base is populated with explicit model dependencies and Prolog rules provide means for querying the knowledge base for implicit ones. This approach covers the categories of direct, inverse and mapping dependencies. The category of transformation dependencies cannot be realised as intuitively as the others. The basic information required for the first three categories can be provided statically by reading the explicit references between models and populating them as facts to the DK-Base. The problem in determining transformation dependencies is that the connections between source and target models are only established at transformation time, thus dynamically. This cannot be precomputed in a static manner. On the one hand, it is possible to analyse a concrete model transformation specification. At least, the mapping from the source to the target languages can be investigated. On the other hand, knowing the mapping does not yield any information about the concrete model elements being transformed. Furthermore, this task is highly dependent on the particular model transformation language.

More precisely, the determination of transformation dependencies directly corresponds to the problem of acquiring trace links. Traceability is a wide field of research which cannot be covered in this thesis. As a small contribution, we jointly published a language-independent traceability approach for transformations being compiled to byte code of the Java VM in [PRW14]. This means, that such transformations can be traced automatically. We consider the established trace links as transformation dependencies according to our classification. Since a trace link is the explicit manifestation of an implicit information they are added as explicit facts to our DK-Base. This approach allows to determine at least a small part of transformation dependencies.

8.4. Co-Refactoring Knowledge Base

Before our co-refactoring approach is presented, the architecture shown in Fig. 4.2 (cf. page 54) is extended by new constituents related to co-refactoring. The adjusted architecture is depicted in Fig. 8.5. The previously discussed detection of dependent models (cf. Fig. 8.2) is not depicted in the figure. The *Dependent DSL Model* can be considered as one result of the model detection. According to Fig. 8.1, the adjusted architecture now consists of several upright layers, one for every participating DSL and one for its users, respectively. The new stakeholder *Co-Refactoring Engineer* has been added, which can be considered as an orthogonal role, since it must communicate with several DSL Designers in order to get to know the language specifics and internals. The Co-Refactoring Engineer must have an overview of the whole MLDE and its contained DSLs to assure that adequate co-refactorings can be defined. Therefore, the relevant *Co-Refactoring Knowledge (CoRK)* is kept in the Co-Refactoring Knowledge Base (CoRK-Base) which refers to various of the DSL-specific upright layers.

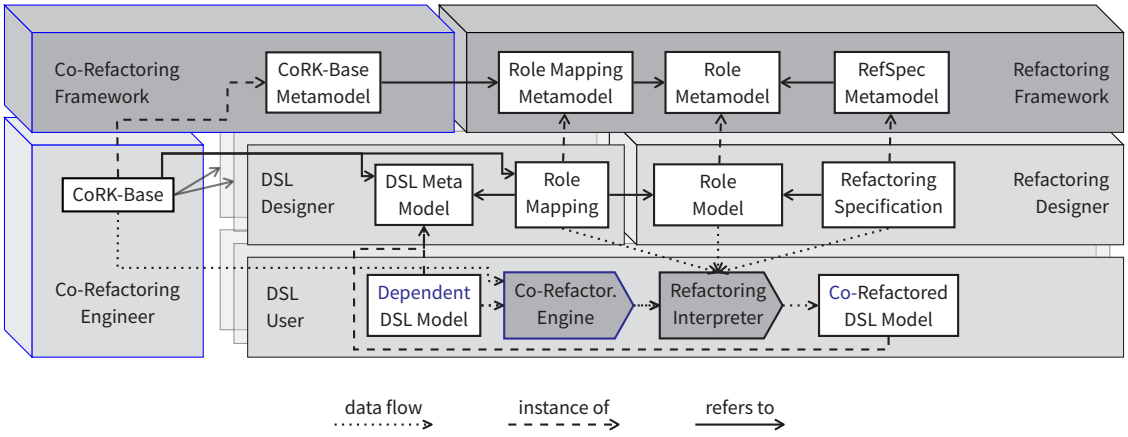


Figure 8.5.: Refactoring architecture extended by Co-Refactoring infrastructure. The blue-framed parts denote the co-refactoring additions.

8.4.1. Specifying Coupled Refactorings with Co-Refactoring Specifications

As explained before, the refactoring stream must interact with the CoRK-Base to determine which co-refactoring can be applied dependent on an initial refactoring. As a consequence, a co-refactoring can be considered as a sequence of coupled refactorings (cf. Sect. 3.3.2). In principle, the pair of a refactoring (incoming) and a co-refactoring (outgoing) is a composition of two refactorings. This means that an incoming and outgoing refactoring are referenced and a binding between both must be specified. This binding is of special significance, since potential involved target DSLs are not known beforehand and thus a generic approach is required which still provides means to enable language-specific bindings [ELF08]. The distinction to the approach of composing refactorings (cf. Sect. 4.2.4) is twofold. First, the intention for co-refactoring is different in the sense that a refactoring should be applied in a dependent model in order to maintain consistency. No new composite refactoring should be made available to users. A co-refactoring is executed transparently for the user. Second, different target languages are involved for a couple of a refactoring and a co-refactoring in general. In this sense, such a pair can be considered as a special variant of cross-language consistency rules as they are used by von Pilgrim *et al.* in [vPUTS13] or by Pfeiffer in [Pfe13]. Furthermore, we argue that a fixed set of such rules is not appropriate, but the approach must provide means to extend the rule set.

As a consequence of this preliminary consideration, we already introduced the term *CoRK-Base* which represents an extensible repository for the aforementioned coupled refactorings. In the following we present the CoRK-Base metamodel in Fig. 8.6 and explain how coupled refactorings are specified.

In the metamodel, the CoRK-Base is represented by the metaclass `CoRefactoringKnowledgeBase`. It can be extended by *Co-Refactoring Specifications (Co-RefSpecs)* (metaclass `CoRefactoringSpecification`). A Co-RefSpec is always defined for the language of the model to be co-refactored. Thus, it references a particular `metamodel`. Furthermore it can contain several `MetamodelImports` which define `shortcuts` (a symbolic name or an abbreviation) for other imported metamodels. Refactorings for imported metamodels then can be referred to when

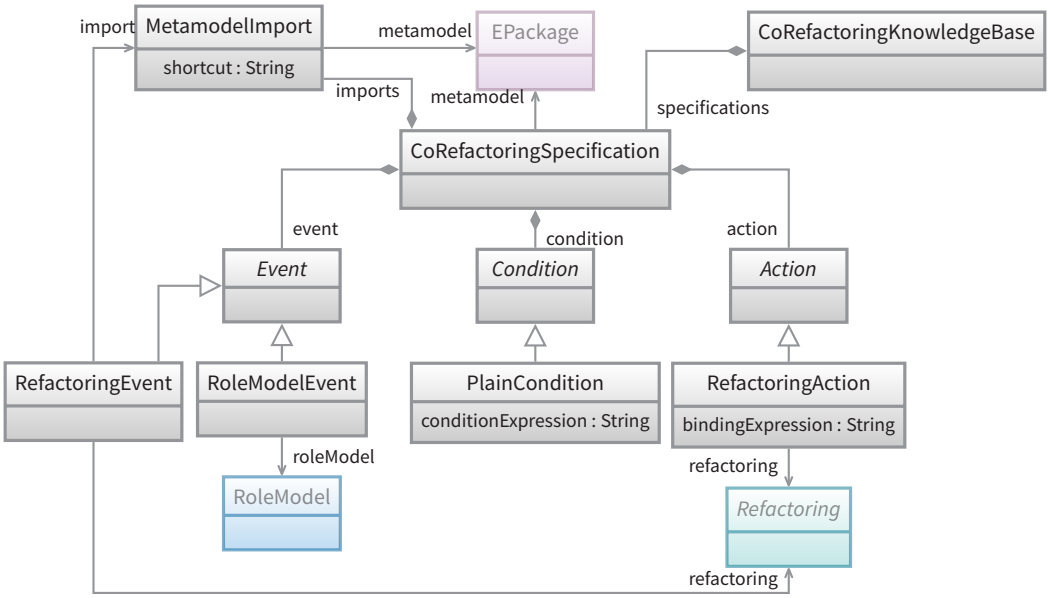


Figure 8.6.: CoRK-Base Metamodel.

the intrinsic dependent co-refactoring is specified. This is explained in Sect. 8.4.2.

Since we refer to the successive propagations of refactorings as refactoring stream, every model intended to be co-refactored can be considered as a barrier in the stream. At every barrier it must be decided *if* and, in the positive case, *how* the refactoring stream can continue. Therefore, the arrival of the stream at a barrier is considered as an incoming `Event` for the model and is reflected as such in the Co-RefSpec. To take the decision at the barrier if the refactoring stream can continue a Co-RefSpec can contain a `Condition`. The question how the refactoring stream continues is answered by an outgoing `Action`. Thus, a Co-RefSpec can be considered as an ECA rule [DGG95]. An incoming `Event` always corresponds to a refactoring on the refactoring stream which initiates the co-refactoring impulse for a dependent model. There are two alternatives to specify the Co-RefSpec: 1) the co-refactoring engineer knows the language of the potentially incoming refactoring, and 2) she does not know the language and, thus, the concrete refactoring to react on is unknown, too. For the first alternative, the co-refactoring engineer can define a `RefactoringEvent`, which means that this Co-RefSpec is intended to react on a concrete incoming refactoring. Thus, an `imported` metamodel is referenced and one `Refactoring` being defined for this imported language. Such a refactoring can be a `RoleMapping` (normal refactoring) or a `CompositeRoleMapping` (composite refactoring). For this alternative more specific actions regarding the language of the incoming refactored model can be defined.

The second alternative covers the case when the language is not known. Then the co-refactoring engineer can at least make assumptions about the kind of an incoming refactoring in terms of the mapped `RoleModel` of the refactoring. Therefore, a `RoleModelEvent` is used. This case is more generic than the first one, but the formalism of role models allows for asserting some structural facts of an incoming refactoring. In our approach, the more specific `Event` (`RefactoringEvent`) is preferred against the more generic one (`RoleModelEvent`). This means that if in the CoRK-Base a specific `RefactoringEvent` and a generic `RoleModelEvent`, which

references the same role model as is mapped in the specific `RefactoringEvent`, are declared, then the specific event is used. This can be compared with overriding methods in subclasses in object-oriented programming languages.

Especially for a `RoleModelEvent`, a `Condition` should be defined to specify the structure of an incoming refactoring more precisely. The sub-metaclass `PlainCondition` therefore contains the attribute `conditionExpression`. Provided that the `Condition` is satisfied, the `RefactoringAction` specifies what coupled `Refactoring` (the outgoing co-refactoring) must be applied and, thus, how the refactoring stream continues. Again, this co-refactoring can be a `RoleMapping` (normal refactoring) or a `CompositeRoleMapping` (composite refactoring). As already explained previously, the binding in the co-refactoring scenario is of special significance. This will be discussed in the following section.

8.4.2. Specifying Bindings for Co-Refactorings

As already explained, the objective of applying a co-refactoring is to maintain consistency in dependent models without altering the overall behaviour. As a consequence, it might not be sufficient to only define the meaning of elements bound to roles in an incoming refactoring, but to bind roles in the outgoing refactoring, as well, as it was done for composite refactorings (cf. Sect. 4.2.4). Furthermore, additional domain-specific modifications might be required to assure that a dependent model still is consistent after the co-refactoring. Therefore, coupled refactorings on a refactoring stream are also considered as cross-language consistency rules. As a consequence, we argue that the specification approach for a required binding between incoming refactoring and outgoing co-refactoring must support not only means for mapping roles, but also for specifying further modifications. Therefore, we decided to realise the specification of bindings by means of an expression language such as the Expression Language for Java [Chu13].

An expression language is a programming language usually built upon another GPL. It abstracts over language concepts and simplifies the syntax. But the main advantage which is the reason for the decision to rely on an expression language for the binding, is that it can be easily embedded into various contexts. The Expression Language for Java, e.g., is used for embedding expressions into web applications which then are interpreted on demand. This way, the amount of scripting in Java Server Pages could be reduced drastically [LBD05]. Most often, such an expression language provides extension points for specific interpretation of own domain-specific constructs, and a comfortable Application Programming Interface (API).

Thus, using an extensible expression language for the specification of bindings allows for the integration of our role concepts into the interpretation of the language. Therefore, its semantics is extended by our role concept and all other language concepts can be used to enhance the co-refactoring of the dependent model. Due to the extensible nature of expression languages, the execution semantics can be controlled. In our approach, this allows for the rejection of modifications regarding other models than the one intended to be co-refactored. Only the dependent model can be subject to modification in order to continue the natural order of the refactoring stream. This approach of using an expression language is also used for the mentioned `conditionExpression`. If the expression language evaluates the `conditionExpression` successfully the co-refactoring is applied, otherwise it is not. We chose the Java-based expression

```
1  CoRefSpec for <http://www.eclipse.org/emf/2002/Ecore>
2  import owl:<http://org.emftext/owl.ecore>
3  {
4     incoming refactoring owl:<Rename␣Element>
5     outgoing corefactoring <Rename␣EElement> $
6        oldName = OUT.Nameable.name;
7        OUT.Nameable.name = IN.Nameable.name;
8        package = OUT.Nameable.eContainer();
9        classifiers = package.getEClassifiers();
10       copy = new EClassImpl();
11       copy.name = oldName + "_COPY";
12       classifiers.add(copy);
13    $
14 }
```

Listing 8.4: Example Co-RefSpec for renaming an Ecore model dependent on an OWL model.

language MVEL[1] because it has high performance[2] and we already have good experience with it since GUERY has embedded it as well.[3]

For a better understanding, consider the example from Sect. 8.2 again. We now focus the transformation dependency between the ontology and a generated domain model. An element contained in the ontology is renamed which should trigger a dependent renaming of a concept in the domain model. In addition, a new concept should be created in the domain model having the old name of the renamed concept with a `_COPY` suffix. This realises a simple history list. Listing 8.4 shows the according Co-RefSpec. As one can see in the first two lines, the Ecore language is the target language for which the Co-RefSpec is specified. It represents the concrete DSL for domain models. OWL is declared as an imported language. In Line 4, the incoming (initiating) refactoring *Rename Element* for OWL is indicated as the `RefactoringEvent`. Line 5 shows that *Rename EElement* is defined as outgoing (triggered) `RefactoringAction`. Due to the extension facilities of the expression language we added the keywords `IN` and `OUT` to be able to explicitly refer to the incoming and outgoing refactorings. Line 6 shows that the old name is stored whereas the incoming name is bound to the outgoing name in Line 7. Lines 8–12 then determine the package of the renamed concept in the domain model, create a new one, set the suffixed name and add the copied concept to the list of the package's classifiers. This Co-RefSpec shows both the binding of an outgoing yet unbound role attribute to the same value as the bound incoming role attribute (Line 7), and the specification of additional modifications being specific for this use case (the creation of a suffixed copy).

[1] `http://mvel.codehaus.org/` (visited 26th April 2015)
[2] `http://mvel.codehaus.org/Performance+of+MVEL+2.0` (visited 26th April 2015)
[3] Unfortunately, this can only be seen in the source code of GUERY: `https://code.google.com/p/gueryframework/source/browse/src/java/nz/ac/massey/cs/guery/mvel/CompiledPropertyConstraint.java?name=1.3` (visited 26th April 2015).

8.4.3. Determination of Co-Refactoring Specifications

In the previous sections, our approaches for determining model dependencies and specifying dependent co-refactorings have been explained. The new stakeholder *Co-Refactoring Engineer* (cf. Fig. 8.5) has to populate the CoRK-Base. This process might render rather complex when the MLDE supports many DSLs. Thus, it would be helpful to give her assistance for this task. The problem is to define reasonable couples of specified refactorings in order to define Co-RefSpecs. Currently, we see two possibilities.

First, the co-refactoring engineer can get support on a role mapping basis. Thus, at first two languages must be selected for which co-refactoring support should be made available. One of them must be selected as the one from which refactorings can be initiated. This methodology assumes that related languages may have a similar degree of abstraction and, thus, potentially related refactorings are also similar (like two dependent renamings). Thus, the co-refactoring engineer could select a role mapping defined for the initiating language and all role mappings which map the same role model as the initiating role mapping can be suggested as potential candidates for a `RefactoringAction`.

Second, a more general scenario of specifying a Co-RefSpec on demand is proposed. Basically, the presented approach for the specification of Co-RefSpecs does not assume that the mapped role models of coupled refactorings must be equal. Thus, the aforementioned alternative might only work out for simpler dependent refactorings such as renamings. For the more general alternative, a user-driven approach can be applied. When an initiating refactoring occurs on the refactoring stream, dependent models are detected but for a certain model no Co-RefSpec is defined. In this case, the user could be asked which of the available refactorings of her language should be selected as the appropriate co-refactoring. Nevertheless this task still remains complex and is left open for future work.

8.5. Discussion

An aspect not discussed up to now is the occurrence of conflicts or cycles. According to [GKP07], *breaking non-resolvable* changes cannot occur within our approach since the definition of Co-RefSpecs can be considered as a strategy for maintaining consistency and, thus, is always an operation for resolving a breaking change. Hence, the co-refactoring engineer is responsible to define valid Co-RefSpecs that do not violate the semantics of a refactored or co-refactored model. Furthermore, as already discussed in Sect. 4.3, we support the preservation of static semantics on the basis of well-formedness rules. This implies that specified pre- and post-conditions are checked in the co-refactoring scenario as well and can be reported as such.

The more critical aspect of cycles can definitely occur in the sense that an already refactored model on the refactoring stream becomes subject for a co-refactoring again. This situation can be detected and reported to the user but there is no universal solution to avoid cycles. Thus, our approach supports the detection of cycles and then the user must decide how to proceed. The reason is that the refactoring stream can change dynamically in case models are removed from the MLDE or modifications are applied not being refactorings. Then, dependencies might be removed from the DK-Base and the refactoring stream changes. Thus, it can only be examined in a concrete situation which upcoming co-refactorings might be applied according to the current refactoring stream. This analysis then must take into account the defined Co-RefSpecs and the

currently available model dependencies to make assumptions about if a cycle can occur or not.

The main responsibility in our approach relies on the co-refactoring engineer. The process of defining co-refactorings must be performed in direct communication with DSL designers.

8.6. Conclusion

In this chapter, we presented our approach for co-refactoring models. It is divided into two parts implied by the general workflow depicted in Fig. 8.2. First, we discussed how dependent models and elements within them can be detected (Requirement 1 and 2 on page 39). Therefore, we defined four categories of potential model dependencies and provided a logics-based approach to reveal those dependencies. Explicit dependencies are persisted in a DK-Base and the others are determined by querying the DK-Base. Second, the intrinsic co-refactoring approach as presented. It comprises the Co-RefSpec by means of coupled refactorings, which populate the CoRK-Base. Therefore, an incoming initiating refactoring is referred (Requirement 3), a condition must be specified (Requirement 4) which validates if an outgoing triggered co-refactoring is allowed to be applied (Requirement 5). Furthermore, we discussed the binding approach of using an expression language, which not only enables the binding of elements from an initial refactoring to roles of the dependent co-refactoring, but also the execution of additional modifications in order to maintain consistency (Requirement 6). The use of an expression language assures that domain-specific modifications can be specified in connection with a co-refactoring (Requirement 8). The whole approach is supported by an example from the domain of ontology-driven requirements engineering and software development. A discussion about limitations and open issues closes this chapter.

9

Refactory: An Eclipse Tool For Quality-Aware Refactoring and Co-Refactoring

To demonstrate the feasibility of the conceptual work of this thesis, the concepts have been implemented in *Refactory*, the first tool for quality-aware refactoring and co-refactoring to resolve quality smells. Refactory is a set of plugins extending the Eclipse platform.[1] Eclipse is an open-source IDE for development in many programming languages. It is implemented in the Java GPL. Eclipse provides a *workspace* which abstracts from the particular concrete file system. Thus, developers can transparently access artefacts in the workspace. The runtime system implements the Open Service Gateway initiative [OSGi14] (OSGi) specification. Thus, a mature extension mechanism is one of the advantages of this IDE. Developers can extend existing extension points with own plugins and can contribute new features at many architectural layers. Furthermore, own extension points can be defined which provide means to extend own plugins, too. Eclipse has a huge community which contributes many different projects.[2]

One of these projects is the *Eclipse Modeling Framework [SBPM08] (EMF)* which provides rich functionality for model-based development in Eclipse. According to the MOF modelling stack, the meta-metamodel *Ecore* implements the EMOF standard which is used as the common base in the EMF [SBPM08]. Ecore allows for the definition of metamodels from which a domain-specific Java API can be generated. Furthermore, the generic reflective API of Ecore itself can be used to manipulate models domain-independently. Thus, EMF can be considered as a Language Workbench and based on a metamodel, a DSLE can be generated. Such a generated DSLE is a set of Eclipse plugins which means that Eclipse in conjunction with EMF, Ecore and certain DSLEs can be considered as a Multi-Language Development Environment (MLDE). Other projects build upon the EMF and provide modelling tools, e.g. to define graphical (GMF) or textual syntaxes (EMFText) for DSLs.

These properties have been the motivation to implement our tool based on Eclipse and the EMF.

[1] `http://eclipse.org/` (visited 26th February 2015)
[2] `https://projects.eclipse.org/list-of-projects` (visited 26th February 2015)

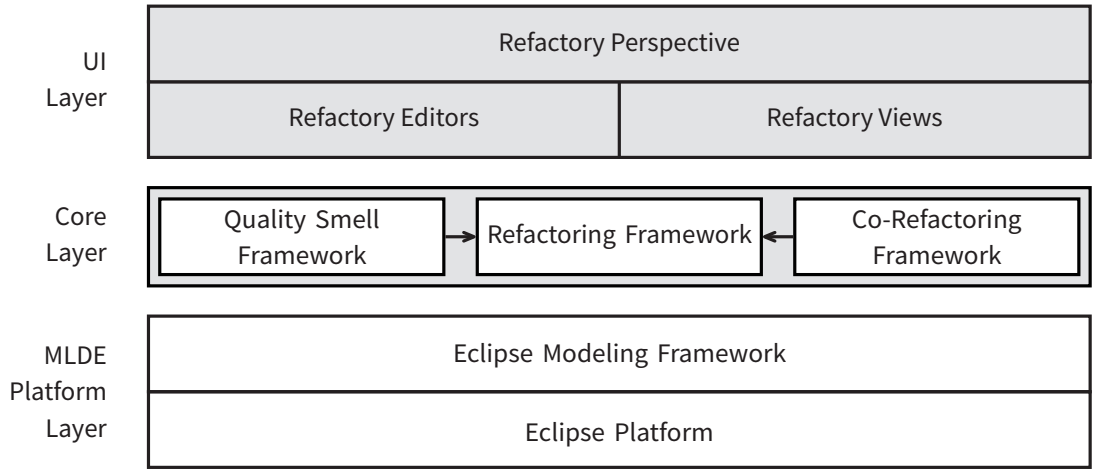


Figure 9.1.: Overall Architecture of Refactory. Grey boxes denote parts we contribute to the MLDE.

Refactory is publicly available via an Eclipse update site and further information can be found at the following address: `http://www.modelrefactoring.org/`. The overall architecture of Refactory is depicted in Fig. 9.1. The MLDE platform layer comprises the Eclipse Platform and the EMF. On top of it, the Refactory core layer contains the three frameworks covering the realisation of our approaches regarding refactoring, quality smells and co-refactoring. The latter two frameworks make use of the refactoring framework. The upper layer is the UI, which contributes several editors and views for the definition of Refactory-related artefacts. In the following sections, all three parts of Refactory are presented from an implementation point of view.

9.1. Refactoring Framework

In this section, we present the concrete realisation of the four constituents of our refactoring approach, namely role models, refactoring specifications, role mappings and refactoring compositions. To fulfil Requirement 3 on page 25, a domain-specific customisation mechanism is presented. Furthermore, the support for pre- and post-conditions based on the OCL is explained, before the integration into the Eclipse-based refactoring architecture is shown.

9.1.1. Role Model

The EMOF metamodel for the specification of role models depicted in Fig. 4.4 on page 56 has been implemented by means of Ecore. Thus, we translated the presented concepts directly to an Ecore model (which is the metamodel for the role model DSL). Furthermore, a textual syntax is defined with EMFText enabling a human readable representation of role models. This syntax definition similar to the Extended Backus-Naur Form [ISO96] (EBNF) is depicted in Listing 9.1 and is explained in the following.

```
1 SYNTAXDEF rolestext
```

```
2  FOR <http://www.emftext.org/language/roles>
3  START RoleModel
4
5  TOKENS{
6     DEFINE UPPER $('A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*$;
7     DEFINE LOWER $('a'..'z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*$;
8     DEFINE NUMBER $('0')|('-1')|('*')|(('1'..'9')('0'..'9')*)$;
9  }
10
11 RULES {
12    RoleModel ::= "RoleModel" name[UPPER] "{" roles* collaborations* "}";
13    Role ::= modifier[optional:"optional",input:"input"]* "ROLE" name[UPPER] ("("
          attributes ("," attributes)* ")")? ";";
14    RoleAttribute ::= name[LOWER];
15    RoleProhibition ::= source[UPPER] "|-|" target[UPPER] ";";
16    RoleImplication ::= source[UPPER] "->" target[UPPER] ";";
17    RoleAssociation ::= source[UPPER] sourceName[LOWER]? sourceMultiplicity "--"
         target[UPPER] targetName[LOWER]? targetMultiplicity ";";
18    RoleComposition ::= source[UPPER] sourceName[LOWER]? sourceMultiplicity "<>-"
          target[UPPER] targetName[LOWER]? targetMultiplicity ";";
19    Multiplicity ::= "[" lowerBound[NUMBER] ".." upperBound[NUMBER] "]";
20 }
```

Listing 9.1: Textual syntax for role models.

Before the definition of the syntax rules (Line 11), the metamodel for which the textual syntax is defined, the metaclass used as starting rule and some tokens are defined. Then, for each non-abstract metaclass a textual syntax is defined. Keywords are used as syntactic elements to achieve a human-readable structure of the model elements. Line 12 starts with the rule for a `RoleModel`, the container element. It is defined by the keyword *RoleModel* and is followed by its `name`. Parenthesised by curly braces the `roles` and `collaborations` follow. A `Role` (Line 13) definition starts with optional `modifiers` followed by the keyword *ROLE* and its `name`. `RoleAttributes` are optionally specified by surrounding them with brackets and separating them with a comma. They are identified by a `name` (Line 14). A `Role` definition closes with a semicolon. This is the same for the other rules for which reason this is not mentioned anymore. The rules for `RoleProhibition` (Line 15) and `RoleImplication` (Line 16) are similar. They reference the `source` role in the beginning and the `target` role in the end. In between the tokens /-/ and -> are used to denote a `RoleProhibition` and `RoleImplication` constraints, respectively. In Lines 17 and 18, the rules for `RoleAssociation` and `RoleComposition` can be seen. Again, they differ only by the tokens – and <>- respectively. What they have in common is that after the reference of the `source` role, a `sourceName` can follow to define a name for this end of the collaboration. This can be used to enable name-based navigation. Then the `sourceMultiplicity` is located in front of the token which specifies the type of collaboration. Afterwards, the referenced `target` role is followed by the optional `targetName` and the `targetMultiplicity`. The last rule in Line 19 defines that a `Multiplicity` is surrounded by square brackets and separated by two dots.

From the grammar, EMFText generates an Eclipse editor with syntax highlighting, code completion and code navigation. The resulting editor can be seen in Fig. 9.2. Role models can

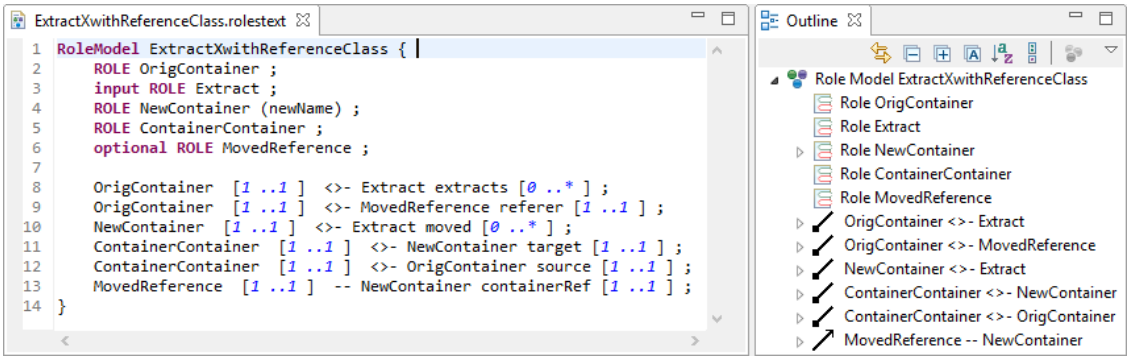


Figure 9.2.: Textual editor and tree-like outline for Role Models applied in Refactory.

now be defined in a textual manner and are understood as models by the EMF.

To allow for the flexible addition of new role models, Refactory provides an Eclipse extension point called *rolemodel*. It expects only one attribute, namely a reference to a role model resource. Every provided extension to this extension point is registered in a *role model registry* maintained by Refactory. This registry also provides an API so that it can be accessed without Eclipse, e.g., for unit testing scenarios. Refactory accesses role models via this registry exclusively.

9.1.2. Refactoring Specification

For the metamodel of the RefSpec concepts depicted in Fig. 4.7 on page 62 and following, an Ecore model has been specified. Furthermore, a textual syntax has been defined with EMFText. This syntax definition is depicted in Listing 9.2 and shortly explained in the following.

```
SYNTAXDEF refspec
FOR <http://www.emftext.org/language/refactoring_specification>
START RefactoringSpecification

TOKENS{
   DEFINE UPPER $('A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*$;
   DEFINE LOWER $('a'..'z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*$;
   DEFINE DOT $($ + UPPER + $|$ + LOWER + $)$ + $'.'$ + LOWER;
   DEFINE INTEGER$('1'..'9')('0'..'9')*|'0'$;
}

RULES{
   RefactoringSpecification ::= "REFACTORING" "FOR" usedRoleModel['<','>'] "
       STEPS" "{" (instructions ";" )* "}";

   VariableAssignment ::= "object" variable ":=" assignment;
   Variable ::= name[LOWER];
   VariableReference ::= variable[LOWER];
   CollaborationReference ::= collaboration[DOT];
   ConstantsReference ::= constant[INPUT:"INPUT"];
   RoleReference ::= role[UPPER] "from" from;
   FromClause ::= operator "(" reference ")";
```

```
22    UPTREE ::= "uptree";
23    PATH ::= "path";
24    FILTER ::= "filter";
25
26    FIRST ::= "index" variable ":=" "first" "(" reference ")";
27    LAST ::= "index" variable ":=" "last" "(" reference ")";
28    AFTER ::= "index" variable ":=" "after" "(" reference ")";
29    ConcreteIndex ::= "index" variable ":=" index[INTEGER];
30    IndexVariable ::= name[LOWER];
31
32    CREATE ::= "create" "new" variable ":" sourceRole[UPPER] "in" targetContext (
          "at" index[LOWER])?;
33    MOVE ::= "move" source "to" target ("at" index[LOWER])? (moveModifier)?;
34    DISTINCT ::= "distinct";
35
36    SET ::= "set" "use" "of" source "in" target;
37    UNSET ::= "unset" "use" "of" source "in" target;
38    ASSIGN ::= "assign" (sourceAttribute[DOT] "for" )? targetAttribute[DOT];
39
40    REMOVE ::= "remove" (modifier)? removal;
41    RoleRemoval ::= role[UPPER];
42    UNUSED ::= "unused";
43    EMPTY ::= "empty";
44 }
```

Listing 9.2: Textual syntax for refactoring specifications.

The starting rule (Line 13) is for the root container of a RefSpec—the `RefactoringSpecification`, indicated by *REFACTORING FOR* and followed by a reference to the `usedRoleModel`. A *STEPS* starts the `instructions` part, each of them separated by a semicolon. A `VariableAssignment` (Line 15) is declared with *object* followed by the `name` (Line 16) for the `variable` and the according `assignment`. An assignment can be a `VariableReference` (Line 17), a `CollaborationReference` (Line 18), a `ConstantsReference` (Line 19) or a `RoleReference` (Line 20). The first two of them are referenced by their respective names. The third is indicated by the *INPUT* keyword. The latter starts with the referenced `role` followed by *from* and a `FromClause` (Line 21). The `operator` of such a clause is denoted by its lower-case metaclass name and a variable or a constant can be `referenced`.

`IndexVariables` are declared quite similarly only that the keyword *index* is used in front of the `variable` (Lines 26–29). `FIRST`, `LAST` and `AFTER` are denoted by their lower-case name again. A `ConcreteIndex` is consequently defined by using an integer value.

To create a new model element *create new* is used, followed by the name of the new `variable` (Line 32). The referenced `sourceRole` denotes the type of the element and the `targetContext` is given after the *in* keyword. Optionally, an `index` can be denoted after an *at*. Moving a `source` *to* a `target` is prefixed by *move* (Line 33). Optionally, an *at* references an `index` and the only optional `moveModifier` is denoted by *distinct* (Line 34).

Setting and unsetting (Lines 36, 37) references to other roles, respectively, starts with *(un)set use of* and is followed by referencing `source` which should be (un)set *in* the `target`. A `targetAttribute` of a role is set by using the *assign* keyword (Line 38). If the value of another `sourceAttribute` should be passed then this must be done with a *for*-clause. While interpreting,

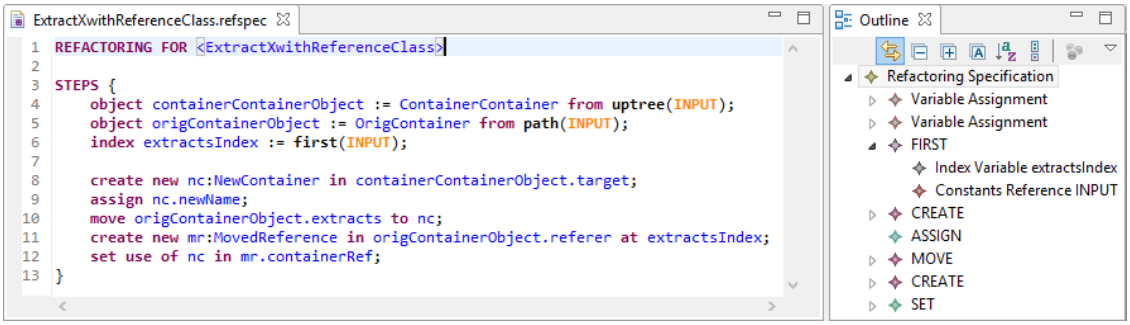


Figure 9.3.: Textual editor and tree-like outline for RefSpec models applied in Refactory.

the user will be asked if no `sourceAttribute` is given in an `ASSIGN` instruction.

To *remove* an element this keyword is optionally followed by a `modifier` which can be *unused* (Line 42) or *empty* (Line 43). Afterwards, the `removal` indicates the element to be removed.

The resulting editor can be seen in Fig. 9.3. RefSpec models can now be defined in a textual manner and are understood as models by the EMF.

Similarly to the role model registry, we also implemented a RefSpec registry. Therefore, again an Eclipse extension point is declared at which RefSpec resources can be registered. Every extension then is added to the *RefSpec registry* which can also be accessed via an API. Refactory accesses refactoring specifications via this registry exclusively.

9.1.3. Role Model Mapping

For the metamodel of the role mapping concepts depicted in Fig. 4.5 on page 58, an Ecore model has been specified. Furthermore, a textual syntax has been defined with EMFText. This syntax definition is depicted in Listing 9.3 and shortly explained in the following.

The keywords *ROLEMODELMAPPING FOR* initiates the creation of such a model (Line 11). It is followed by the desired `targetMetamodel` and the optional *IMPORTS* keyword denotes that `importedMetamodels` can be referenced. The `roleMappings` specify their names within quotation marks (Line 12) followed by *maps* to reference the `mappedRoleModel`. Curly braces enclose the `concreteMappings`, which map a `role` on the left hand side to a `metaclass` on the right hand side and := in between (Line 13). The `attributeMappings` can follow parenthesised and separated by commas. The same holds for `collaborationMappings`, except that curly braces and semicolons are used, respectively. A `collaboration` is followed by := which denotes that a sequence of `pathSegments` can be given by connecting them with -> (Line 14). Such a segment is specified by a `ReferenceMetaClassPair` giving a `reference` and optionally a colon followed by the desired `metaclass` (Line 15). A `roleAttribute` is mapped to a `metaAttribute` by pointing to it with -> (Line 16).

The resulting editor can be seen in Fig. 9.4. Role mapping models can now be defined in a textual manner and are understood as models by the EMF.

To allow for the flexible addition of new role mappings an Eclipse extension point called *rolemapping* is provided. It expects a role mapping resource. Additionally, icons can be registered which are presented to the user in the UI. Every provided extension to this extension point is registered in a *role mapping registry* maintained by Refactory. It is also accessible programmatically

```
SYNTAXDEF rolemapping
FOR <http://www.emftext.org/language/rolemapping>
START RoleMappingModel

TOKENS {
   DEFINE UPPER $('A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*$;
   DEFINE LOWER $('a'..'z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*$;
}

RULES {
   RoleMappingModel::= "ROLEMODELMAPPING" "FOR" targetMetamodel['<','>'] ("
       IMPORTS" importedMetamodels['<','>']+)? roleMappings+;
   RoleMapping ::= name['"','"'] "maps" mappedRoleModel['<','>'] "{"
       concreteMappings+ "}" ;
   ConcreteMapping ::= role[UPPER] ":=" metaclass[UPPER] ("(" attributeMappings
       ("," attributeMappings)* ")")? ("{" collaborationMappings (
       collaborationMappings)* "}")? ";";
   CollaborationMapping ::= collaboration[LOWER] ":=" pathSegments ("->"
       pathSegments)* ";";
   ReferenceMetaClassPair ::= reference[LOWER] (":" metaclass[UPPER])?;
   AttributeMapping ::= roleAttribute[LOWER] "->" metaAttribute[LOWER];
}
```

Listing 9.3: Textual syntax for role mappings.

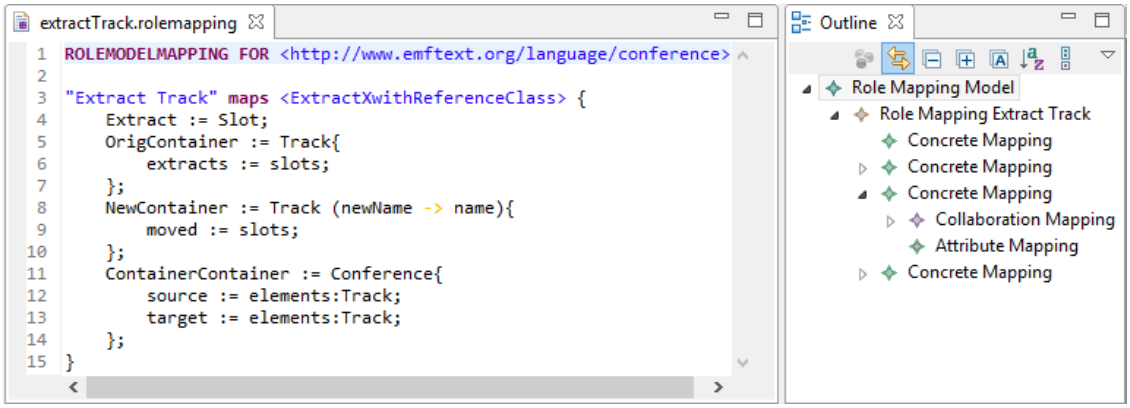


Figure 9.4.: Textual editor and tree-like outline for role mapping models applied in Refactory.

```
1  SYNTAXDEF refcomp
2  FOR <http://www.emftext.org/language/refactoringcomposition>
3  START CompositeRoleMapping
4
5  TOKENS {
6     DEFINE IDENTIFIER $('A'..'Z' | 'a'..'z' | '_')('A'..'Z' | 'a'..'z' | '0'..'9'
          | '_')*$;
7  }
8
9  RULES {
10    CompositeRoleMapping ::= "COMPOSITE" "REFACTORING" name['"','"'] "FOR"
          targetMetamodel['<','>'] first['<','>'] sequence;
11    BoundRoleMapping ::= "->" roleMapping['<','>'] ("{" bindings+ "}")?
          nextMapping?;
12    SourceTargetBinding ::= source[IDENTIFIER] "=" target[IDENTIFIER] ";";
13 }
```

Listing 9.4: Textual syntax for refactoring compositions.

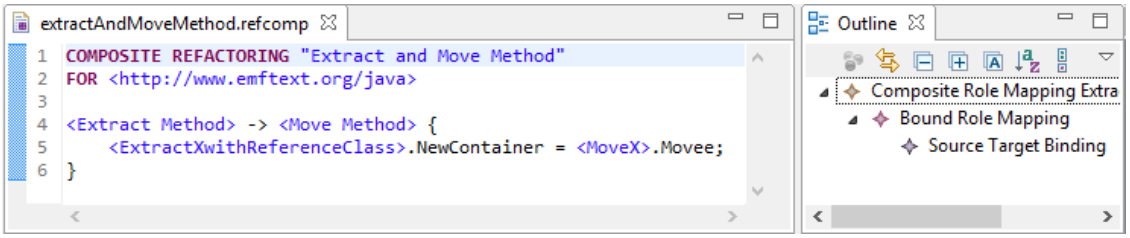


Figure 9.5.: Textual editor and tree-like outline for refactoring composition models applied in Refactory.

and Refactory accesses role mappings via this registry exclusively.

9.1.4. Refactoring Composition

For the metamodel of the refactoring composition concepts depicted in Fig. 4.14 on page 69, an Ecore model has been specified. Furthermore, a textual syntax has been defined with EMFText. This syntax definition is depicted in Listing 9.3 and shortly explained in the following.

The keywords *COMPOSITE REFACTORING* denote the root container of such a model. The `name` is surrounded by quotation marks followed by *FOR* to reference the `targetMetamodel` (Line 10). Then the `first RoleMapping` follows and subsequent ones are connected by -> (Line 11). The `bindings` are surrounded by curly braces and separated by a semicolon (Line 12). The `source` and `target` roles are mapped with the equals sign.

The resulting editor can be seen in Fig. 9.5. Refactoring composition models can now be defined in a textual manner and are understood as models by the EMF. Again, new refactoring composition models can be registered via an Eclipse extension point which belongs to the role mapping registry explained in the previous section. Refactory takes into account only refactoring compositions added to this registry.

With the presentation of these four refactoring languages the core concepts of the framework are realised. In the following section, the language-specific additive customisation of generic refactorings is illustrated.

9.1.5. Custom Refactoring Extensions

Not all transformation steps specific to a concrete metamodel can be captured in the reusable part of a generic refactoring. For example, recall the *Extract Method* Java refactoring in Fig. 1.5 on page 5. The RefSpec and role model for the used generic *Extract X with Reference Class* refactoring only captures the core language-independent structures of this refactoring. Thus, in the concrete case of *Extract Method* for Java no local variables or method parameters are taken into account, since these are specific for the Java language and cannot be generalised for arbitrary languages. Besides the restructurings from the according RefSpec, the statements to be moved must be analysed with respect to the usage of variables. In case local variables declared outside the selected statements are accessed, additional parameters have to be added to the extracted method. This cannot be handled by the generic *Extract X with Reference Class* refactoring as it is. Therefore, Requirement 3 states to support the specification of language-specific adjustments. For that reason, we introduce *Post-Processors* which execute additional steps after the core refactoring execution. They can be registered for specific metamodels in combination with a role mapping via an Eclipse extension point. Post-processors can obtain the runtime bindings from roles to the particular model elements and can then invoke further transformation steps implemented in Java. The post-processor for the refactoring *Extract CompositeState* used in UML state machines, is presented in Listing E.9 in Appendix E on page 183. It computes the incoming and outgoing transitions of the extracted composite state. In Sect. 10.1, we present further refactorings which require custom extensions.

9.1.6. Pre- and Post-conditions

As already mentioned in Sect. 4.2.3, the specification and evaluation of pre- and post-conditions has not been reflected in our approach so far. We postponed this aspect to this section and will explain in the following how to fulfil Requirement 5 from page 25.

To support pre- and post-conditions, two Eclipse extension points have been declared in Refactory. The first one is called *conditions* and is part of the role mapping extension point and the corresponding registry. At this extension point, a condition can be added to a registered role mapping. It has two attributes: *preConditions* and *postConditions*, each requiring a file. The given file can contain pre- and post-conditions in arbitrary format or language. This extension point is independent from the particular evaluation mechanism used to interpret certain conditions. Therefore, the second extension point is provided which is named *constraint interpreter*. This extension point requires an implementation of the `IConstraintInterpreter` interface provided by Refactory. Implementors of this interface must return **`true`** if they support the interpretation of a certain constraint. As a consequence, Refactory is independent from concrete constraint languages or formats but provides an interface to support arbitrary mechanisms.

As already discussed, the continuous satisfaction of WFRs is important and is considered as the preservation of at least the static semantics of models. Therefore, we provide one concrete implementation for the constraint interpreter interface and support constraints by means of

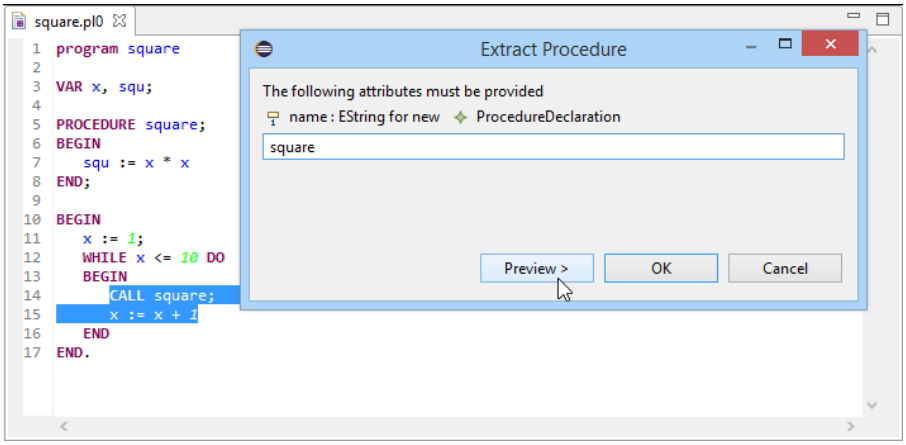


Figure 9.6.: PL/0 example programme before refactoring should be applied.

the OCL. More precisely, our implementation uses *Dresden OCL*[3] since it supports arbitrary metamodels upon which OCL constraints can be defined [WTW10]. As a consequence of using Dresden OCL, Refactory is able to support the detection of static semantics violation by means of WFRs encoded in OCL.

As an example, consider the academic language PL/0 being a small subset of Pascal [Wir86]. We created a EMF-based DSLE with EMFText for it. A PL/0 programme can have procedures for which reason we defined the *Extract Procedure* for it (more details are given in Sect. 10.1.2 and Appendix C). Figure 9.6 shows the editor and a small example programme. Here, one can see that the selected lines should be moved to a new procedure for which the desired name *square* is provided by the user.

The problem is that the programme already contains a procedure with the same name. This is forbidden and, therefore, we defined and registered the constraints in Fig. 9.7 (a). The constraint *uniqueProcedureName* ensures that no two procedure with the same name can exist. If the refactoring still should be applied with the non-unique name, this constraint is violated and the error dialogue in Fig. 9.7 (b) occurs.

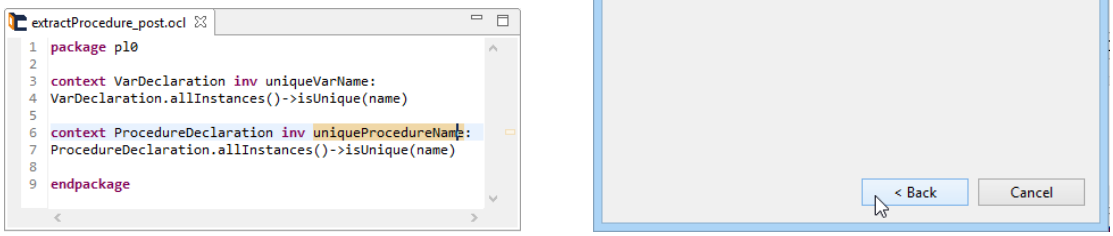
In this small example a first impression of the Eclipse Refactoring Framework has been given. In the following section, the integration therein is presented.

9.1.7. Integration Into the Eclipse Refactoring Framework

So far, only the single parts of the refactoring framework have been presented. As explained in Sect. 1.1, a main drawback of DSLEs is that they do not offer adequate refactoring support. Since Refactory has been implemented as extension for the Eclipse IDE, we claim to seamlessly integrate into existing refactoring facilities of Eclipse in order to not break the user experience and expectations when refactoring models.

Therefore, Refactory supports the following features. The first important aspect is that the presented implementation makes use of the Eclipse Language Toolkit (LTK).[4] The LTK provides

[3] `http://www.dresden-ocl.org/` (visited 27th February 2015)
[4] `https://www.eclipse.org/articles/Article-LTK/ltk.html` (visited 27th February 2015)



(a) Unique names constraints

(b) Constraint violation

Figure 9.7.: OCL constraints and representation of the violation in the refactoring dialogue.

a language-neutral API for the definition of new refactorings. Several well-known refactoring wizards are provided by the LTK including a preview of refactorings. To keep this set of widely accepted refactoring features, Refactory registers the class `ModelRefactoring` (being a subclass of `org.eclipse.ltk.core.refactoring.Refactoring`) in Eclipse. Within this class, the pre- and post-conditions are checked and the intrinsic refactoring of the model must be returned as an instance of `org.eclipse.ltk.core.refactoring.Change` being called `ModelRefactoringChange` in Refactory. This is needed to provide means to roll back a refactoring in case something fails (Requirement 6) or in case a whole refactoring is to be reversed when the user revises her decision (Requirement 7). For this purpose, the `ModelRefactoringChange` class encapsulates the execution of a particular refactoring in a `org.eclipse.emf.transaction.RecordingCommand`, which captures all modifications made to a model. Such a recording command can be undone. An instance of this class is added to the `org.eclipse.core.commands.operations.IOperationHistory` in order to make the undo operation of a refactoring available to the accustomed *Edit* menu in the Eclipse IDE. In addition, our `ModelRefactoringChange` class returns an instance of `org.eclipse.ltk.core.refactoring.ChangeDescriptor` tailored to our generic refactoring approach, namely a `ModelRefactoringDescriptor`. By doing this, it is assured that each instance of a generic refactoring is also added to the Eclipse refactoring history. Thus, the properties of a refactoring are persisted in the workspace and can be re-applied, or a refactoring script can be created out of several refactorings. Furthermore, Refactory registers an extension for the `org.eclipse.ltk.ui.refactoring.IChangePreviewViewer`, interface which can be implemented to provide a preview of a refactoring. The implementation of this interface realised in Refactory makes use of the *EMF Compare*[5] framework for determining differences between models. Therefore, we initiate a dry run of the intended refactoring and determine all inputs which have to be made. EMF Compare can then calculate a difference model and it is presented to the user in the refactoring wizard.

So far, the integration into the Eclipse-specific refactoring workflow was presented. A last

[5] `https://www.eclipse.org/emf/compare/` (visited 27th February 2015)

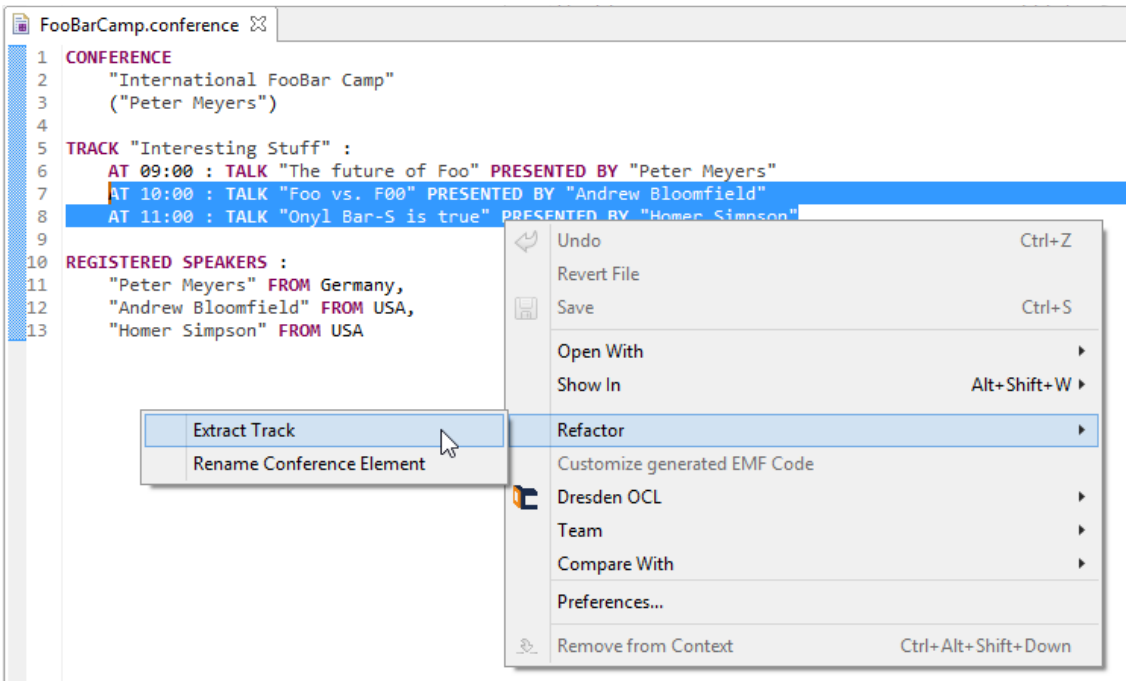


Figure 9.8.: Selection of talks to be refactored.

open aspect concerns the question, which editors used for modelling are supported by Refactory. Therefore, we make use of the adapter mechanism used in Eclipse, which allows for easy declarative registration of additional classes which other objects of interest can be adapted to. For this purpose, Refactory tries to adapt the editor in which a refactoring should be initiated to the interface `IEditorConnector`. This interface is provided by Refactory and is used to get the model elements from a selection and to select modified elements after an applied refactoring. Currently, four different editor connectors are supported: 1) textual EMFText editors, 2) textual Xtext[6] editors, 3) graphical editors based on the GMF, and 4) the intrinsic editor of the Java Development Tools provided by Eclipse. The latter is the editor for Java source files, thus, refactorings provided by Refactory can also be applied for Java models in the accustomed Java editor.

Leaving the explanation of the implementation behind, finally an example should be provided. Consider the *Extract Track* refactoring for the conference DSL depicted in Fig. 4.1 on page 52. To accomplish this refactoring, the according talks have to be selected in the editor and *Extract Track* must be invoked from the context menu as can be seen in Fig. 9.8. After invocation, the refactoring wizard opens and the user is asked for the name of the new track which is intended to be extracted from the selection. This wizard page can be seen in Fig. 9.9. If the user now presses the *Preview* button in the wizard EMF Compare calculates the difference and presents it as can be seen in Fig. 9.10. After pressing the *OK* button in the preview the refactoring is applied and we get the same result as in Fig. 4.1 (b).

With this example, the presentation of the first part of Refactory is finished. In the following section the Quality Smell Framework is illustrated.

[6] `http://xtext.org/` (visited 27th February 2015)

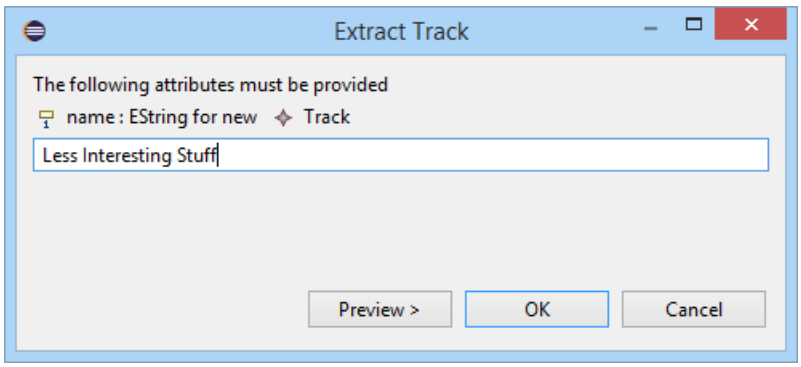


Figure 9.9.: Refactoring wizard for providing the name of the new Track.

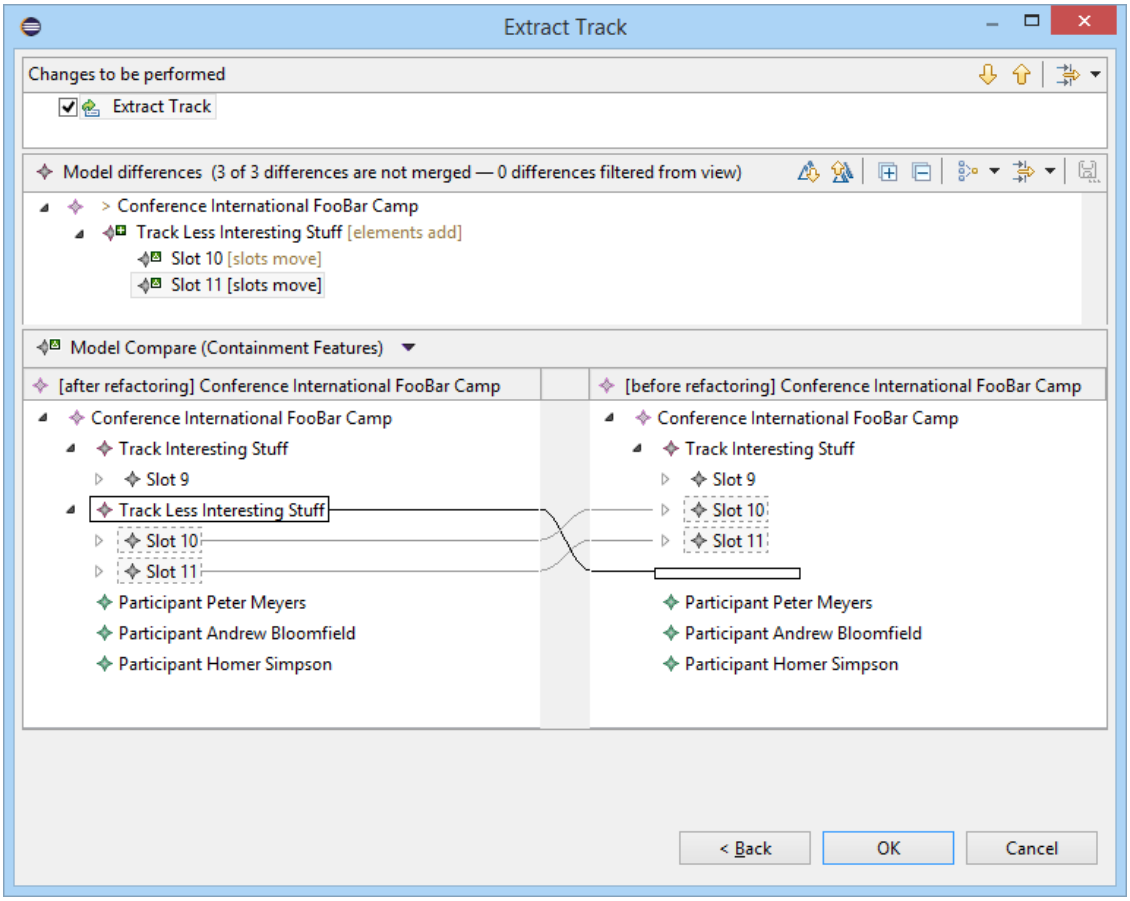


Figure 9.10.: Preview of the *Extract Track* refactoring in a conference model.

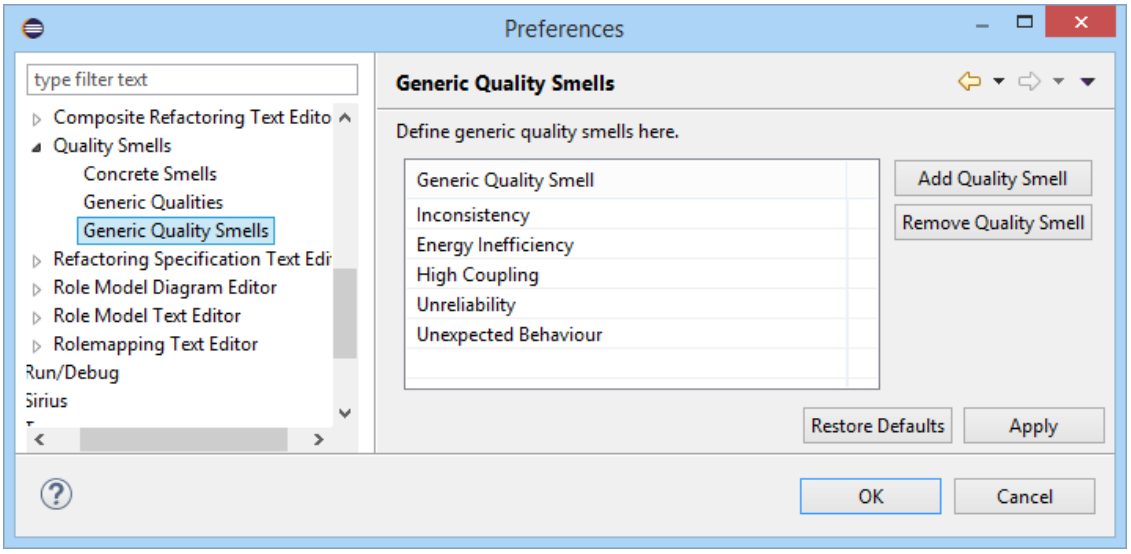


Figure 9.11.: Preference page for the definitin of generic quality smells.

9.2. Quality Smell Framework

In this section, the first implementation of a quality smell framework proposed in Chap. 6 is presented. The metamodels of the quality smell repository (cf. Fig. 6.2 on page 82) and the quality smell calculation repository (cf. Fig. 6.5 on page 86) are realised as EMF Ecore models again. Both of them are considered to be a singleton and, thus, Refactory persists the particular single model in the workspace transparent to the user. This also means that each developer having its own workspace can define her own specific quality smells.

To populate the quality smell repository, three preference pages have been implemented in Refactory. Thus, developers can use the accustomed preference mechanism of Eclipse to carry out the specification of qualities, generic and concrete quality smells. Since all of the three preference pages inherit from an abstract preference page, only the one for the definition of generic quality smells is shown in Fig. 9.11.

To populate the quality smell calculation repository, currently, two Eclipse extension points are provided: one for metrics-based and one for structure-based calculations. For the former, the defined extension point only requires to refer to a Java class being a subclass of `Metric`. Thus, the metric has to be implemented in the according `calculate(model, roleModel)` method. For structure-based calculations it has already been shown in Sect. 7.3 that IncQuery is used to query patterns in EMF-based models. Therefore, the metaclass `IncPLPattern` in Fig. 7.2 on page 92 references the metaclass `Pattern` provided by IncQuery. Consequently, the resource such a pattern is contained in can be registered at the above mentioned extension point which then will be added to the quality smell calculation repository. This extension point also requires the provision of a pattern name, a description and a smell message. The former is used to identify the pattern from the given pattern resource since it can contain various patterns. The other attributes are used for presentation in the UI in case the pattern matches and the according quality smell takes effect. In the following, we want to illustrate how the invocation of IncQuery

```
1  private Result queryPattern(Pattern pattern, ResourceSet resourceSet, RoleModel
       rolesOfInterest) throws IncQueryException {
2
3     BaseIndexOptions options = new BaseIndexOptions();
4     EMFScope scope = new EMFScope(resourceSet, options);
5     IncQueryEngine engine = IncQueryEngine.on(scope);
6     SpecificationBuilder builder = new SpecificationBuilder();
7     IQuerySpecification<? extends IncQueryMatcher<? extends IPatternMatch>>
          querySpecification = builder.getOrCreateSpecification(pattern);
8     IncQueryMatcher<? extends IPatternMatch> matcher = engine.getMatcher(
          querySpecification);
9
10    Result result = null;
11    if (matcher != null) {
12       Collection<? extends IPatternMatch> matches = matcher.getAllMatches();
13       result = createResultFromMatches(matches, rolesOfInterest);
14    }
15    return result;
16 }
```

Listing 9.5: Invocation of IncQuery engine to query structure-based quality smells.

patterns works in detail.

For an uncomplicated workflow, a default implementation of `IncPLPattern` is realised in order to provide means for relieving DSL designers (cf. Fig. 6.1 on page 81) from implementing the invocation of such a pattern in Java. Furthermore, such a programmatic invocation would always be the same according to the generic API provided by IncQuery. Thus, the default implementation of Refactory avoids redundancy: The metod `calculate(model, roleModel)` loads the referenced IncQuery pattern and then invokes the `queryPattern` method which can be seen in Listing 9.5. The initialisation is realised in Lines 3–7. Among the other things, the query scope is set and the engine is created on the scope. In Line 8, the intrinsic matcher is created with respect to the engine. Then, the matcher is invoked in Line 12 by requesting all matches. The determined matches then are passed to the method `createResultFromMatches` which is depicted in Listing 9.6. This method iterates over each match and creates a new `CausingElementsGroup` for it (Lines 3 and 4). Thus, each match is considered to be an instance of the according structure-based quality smell. The names of the parameters of a given pattern correspond to the names of the `rolesOfInterest`. Therefore, in Lines 11–16 the corresponding roles are determined. The intrinsic `RoleElementBinding` then is created in Lines 18–22. By provision of this implementation, DSL designers only have to register an IncQuery pattern in order to define a quality smell for her language. Unfortunately, Refactory currently does not yet support the composition of detection strategies according to Fig. 6.3 on page 84. It is only possible to use a single `CalculationStrategy` to specify how a quality smell should be detected. This is postponed to future work.

In addition to the described extension points, Refactory also provides two views in order to manage the defined qualities and quality smells, and to resolve detected quality smells. The first view can be seen in Fig. 9.12. It is divided into three parts, whereas the left part shows all

```
private Result createResultFromMatches(Collection<? extends IPatternMatch>
    matches, RoleModel rolesOfInterest) {
   CalculationResult result = CalculationFactory.eINSTANCE.
       createCalculationResult();
   for (IPatternMatch match : matches) {
      CausingElementsGroup causingElementsGroup = CalculationFactory.eINSTANCE.
          createCausingElementsGroup();
      List<String> parameterNames = match.parameterNames();
      for (String parameterName : parameterNames) {
         Object matchedElement = match.get(parameterName);
         if(matchedElement != null && matchedElement instanceof EObject){
            EObject boundElement = (EObject) matchedElement;
            Role boundRole = null;
            for (Role role : roleModel.getRoles()) {
               if(role.getName().equals(parameterName)){
                  boundRole = role;
                  break;
               }
            }
            if(boundRole != null){
               RoleElementBinding roleElementBinding = CalculationFactory.
                   eINSTANCE.createRoleElementBinding();
               roleElementBinding.setRole(boundRole);
               roleElementBinding.setBoundElements(Arrays.asList(new EObject[]{
                   boundElement}));
               causingElementsGroup.getBindings().add(roleElementBinding);
               causingElementsGroup.setResultingValue(1);
            }
         }
      }
      result.getCausingGroups().add(causingElementsGroup);
   }
   return result;
}
```

Listing 9.6: Creation of a result with respect to the matches determined by IncQuery.

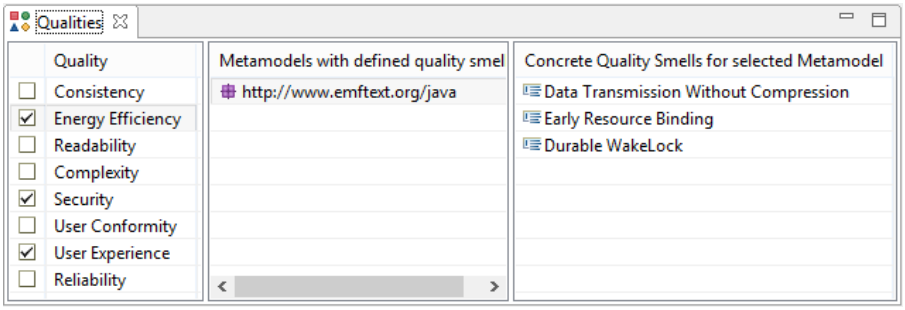


Figure 9.12.: *Qualities* view of Refactory.

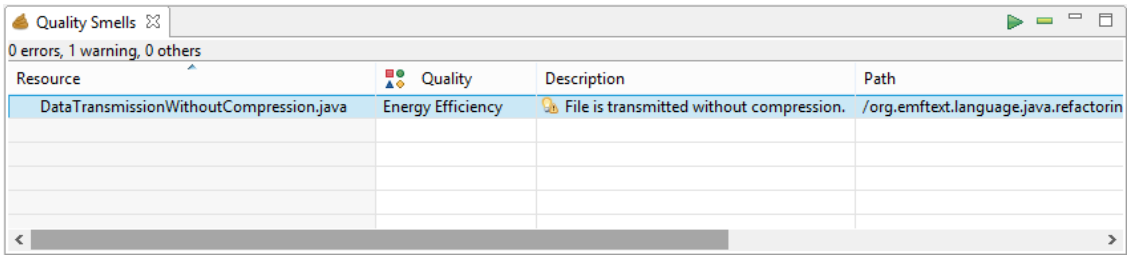


Figure 9.13.: *Quality Smells* view of Refactory.

defined qualities which can be explicitly activated or deactivated. The middle part shows all metamodels for which concrete quality smells have been defined influencing the selected quality in the left part. The right part then shows the concrete quality smells for the language selected in the middle.

The second view is the *Quality Smells* view depicted in Fig. 9.13. It subclasses the `org.eclipse.ui.views.markers.MarkerSupportView` of Eclipse in order to specify that this view presents markers. The Eclipse concept of *markers* is used to indicate that certain elements in a model have specific properties. Most often, markers are used to indicate errors or warnings. Therefore, Refactory exploits the marker concept to denote that a concrete quality smell occurred. Thus, the single line selected in the view in Fig. 9.13 is the visual representation of a quality smell marker. Furthermore *quick-fixes* can be specified to resolve problematic markers in general. In particular, Refactory also makes use of this concept to connect a particular quality smell marker with the resolving refactorings defined for the quality smell. That a quick-fix is available is denoted by the small bulb in the *Description* column in Fig. 9.13. In addition, markers are also visually indicated in the editor of the particular resource as can be seen in Fig. 9.14. The invocation of a resolving refactoring can also be achieved via the *Quality Smells* view by pressing `Ctrl + 1` as accustomed by the ordinary problems view of Eclipse.

The combination of the *Qualities* view of Refactory for activating certain qualities of interest, the indication of detected quality smell occurrences as markers, and the provision of resolving refactorings as quick-fixes has several advantages. The former is completely new, since no known tool can focus on qualities explicitly, since they do not support a relation to qualities. Thus, developers can focus various qualities dependent from different contexts, use cases or situations. The indication of quality smells as markers seamlessly integrates into the accustomed working method of Eclipse. If problems occur markers are created, thus, no new visual metaphor has been introduced and the curve of understanding remains low. Consequently, providing refactorings as quick-fixes fosters the automation to resolve quality smell occurrences. Developers can remove them easily. Summarising, this part of Refactory is the first framework providing support for detecting and resolving deficiencies in models with regard to specific qualities.

The presentation of the second part of Refactory is finished. In the following section, the Co-Refactoring Framework is illustrated.

9.3. Co-Refactoring Framework

In this section the third constituent of Refactory is presented: the Co-Refactoring Framework.

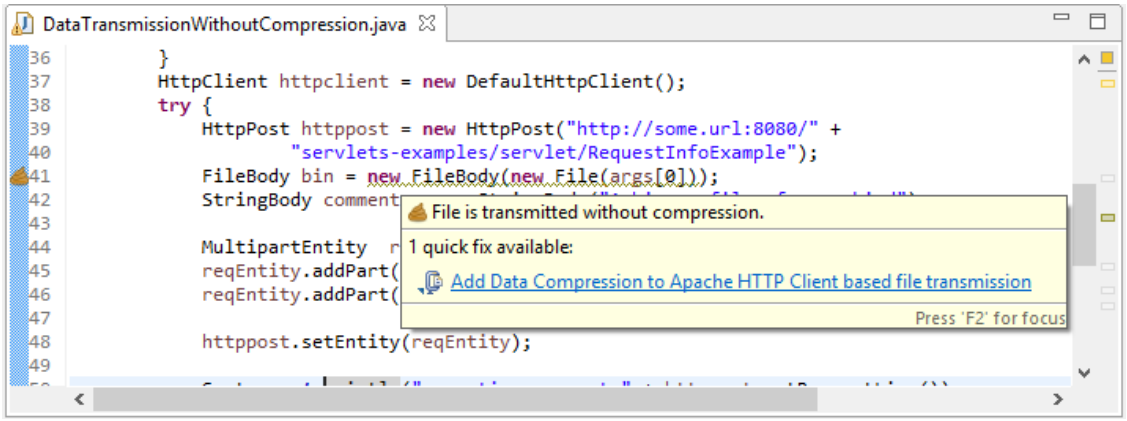


Figure 9.14.: Resolving refactoring invokable by a quick-fix in Refactory.

9.3.1. Concrete Syntax of a Co-RefSpec

For the CoRK-Base metamodel depicted in Fig. 8.6 on page 115, an Ecore model has been created again. As could be already seen in Listing 8.4 on page 117, a textual syntax has been declared for the Co-RefSpec. We realised this with EMFText and the syntax definition is depicted in Listing 9.7.

As can be seen in Line 10, a `CoRefactoringSpecification` starts with the keywords *Co-RefSpec for* and references the target `metamodel`. Optional imports are introduced with *import*. The particular definition of the coupled refactorings is surrounded by curly braces, followed by *incoming* to denote the `event`. Subsequently, a `condition` can be specified before *outgoing* denotes the `action`. A `MetamodelImport` is defined by a `shortcut` followed by a colon and the referenced `metamodel` (Line 11). A `PlainCondition` is introduced by the keyword *condition* and the `conditionExpression` is surrounded by dollar signs (Line 12). An incoming `RefactoringEvent` is commenced with *refactoring* followed by a reference to the `shortcut` of the `imported` metamodel. A colon connects the particular `refactoring` surrounded by < and > (Line 13). The language-independent possibility of specifying an incoming event by means of a `RoleModelEvent` starts with *rolemodel* and is followed by a reference to the particular `roleModel` (Line 14). The outgoing `RefactoringAction` is denoted by *corefactoring* and followed by a reference to the particular `refactoring`, where only those are offered which have been defined for the `metamodel` referenced in the beginning (Line 15). Again, the `refactoring` is enclosed in < and >. The `bindingExpression` is surrounded by dollar signs.

9.3.2. Expression Evaluation by Using an Expression Language

As already explained in Sect. 8.4.2, both the `conditionExpression` and the `bindingExpression` are processed with the expression language MVEL. The main reasons for choosing MVEL have been that it is of high performance, has a clean syntax and a huge open-source test class can be accessed[7].

[7] `https://fisheye.codehaus.org/browse/mvel/trunk/src/test/java/org/mvel2/tests/core/CoreConfidenceTests.java?r=trunk` (visited 27th February 2015)

```
SYNTAXDEF corefspec
FOR <http://www.modelrefactoring.org/corefspec>
START CoRefactoringSpecification

TOKENS {
   DEFINE IDENTIFIER $('A'..'Z' | 'a'..'z' | '-'| '_')('A'..'Z' | 'a'..'z' |
       '0'..'9' | '-'| '_')*$;
}

RULES {
   CoRefactoringSpecification ::= "CoRefSpec" "for" metamodel['<','>'] ("import"
        imports)* "{" "incoming" event condition? "outgoing" action "}";
   MetamodelImport ::= shortcut[IDENTIFIER] ":" metamodel['<','>'];
   PlainCondition ::= "condition" conditionExpression['$','$'];
   RefactoringEvent ::= "refactoring" import[IDENTIFIER] ":" refactoring
       ['<','>'];
   RoleModelEvent ::= "rolemodel" roleModel['<','>'];
   RefactoringAction ::= "corefactoring" refactoring['<','>'] bindingExpression['
       $','$'];
}
```

Listing 9.7: Textual syntax for co-refactoring specifications.

As already seen in Listing 8.4 on page 117, the keywords `IN` and `OUT` are used similar to variables in the binding expression. For being able to realise custom variable resolution, MVEL offers the possibility to evaluate expressions by using an own implementation of the `org.mvel2.integration.VariableResolverFactory` interface. Refactory's implementation is named `GenericBindingResolverFactory` and resolves both of the above keywords to the incoming or outgoing bound elements, respectively. More precisely, for `IN` the subsequent syntactic constructs are resolved against the incoming binding. Consider, e.g., the line `OUT.Nameable.name = IN.Nameable.name` from Listing 8.4. Both the left hand side and the right hand side refer to the same string `Nameable`. The question how to distinguish it is answered by the `GenericBindingResolverFactory` which resolves these strings either against the incoming or the outgoing element bindings according to the used keyword. Thus, the first string in such a chain always denotes a variable which Refactory resolves to the incoming or outgoing bound elements in case the above keywords are used.

Furthermore, MVEL supports the access to properties of an object just by referring to its name instead of using getters or setters (Lines 6–12 in Listing 8.4). Properties can also be chained by using a dot notation. This facility is called *property expression* in MVEL. As can be seen in Listing 8.4, property expressions are frequently used in a binding expression. Let us have a look at this expression: `OUT.Nameable.name`. It is already known that the used variable is resolved to the outgoing bound elements. In addition to custom variable resolvers, MVEL also provides means to register custom property resolvers. Therefore, an implementation of the interface `org.mvel2.integration.PropertyHandler` has to be provided. In Refactory, this is realised by the class `GenericBindingResolver`. It always takes over the resolution in case a role, role attribute or collaboration name is denoted as a supposed property. Thus, in the case of

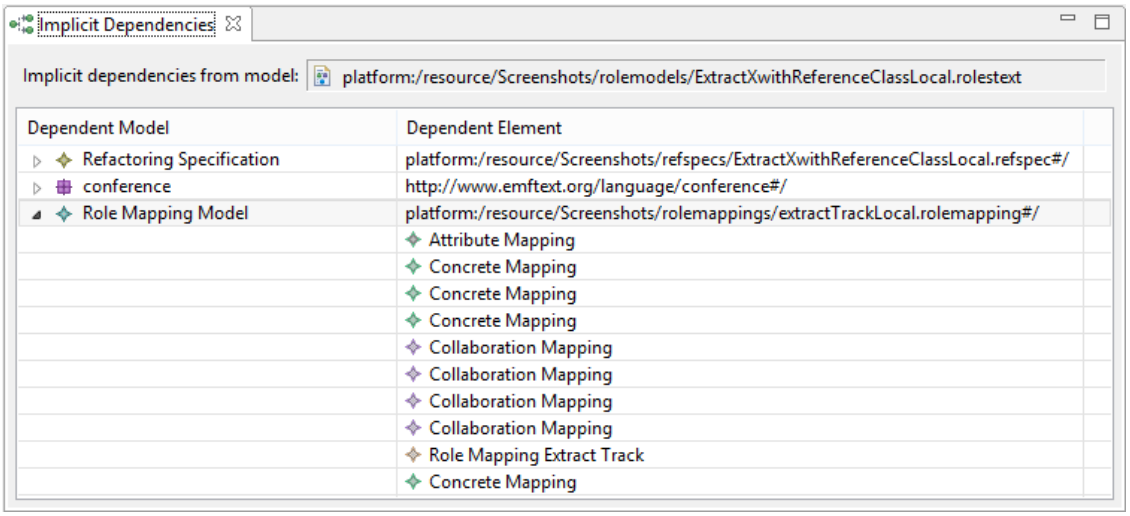


Figure 9.15.: *Implicit Dependencies* view in Refactory.

our example above, the `GenericBindingResolver` takes over when `Nameable` and `name` are to be resolved. According to the current context, these names are resolved against the incoming or outgoing bound elements. In case strings are to be resolved which cannot be handled by Refactory's variable or property resolvers, the default MVEL resolving takes effect. This also means that language-specific properties can be referenced in such a binding expression, since MVEL uses reflection for resolving them. Hence, this approach and implementation provides powerful means to execute bindings or to apply additional modifications.

9.3.3. UI and Integration

To examine implicit dependencies of a model, Refactory provides the *Implicit Dependencies* view. It is depicted in Fig. 9.15. As can be seen in the upper part of the view, the implicit dependencies of the *Extract X with Reference Class* role model are shown. In the lower part of the table, one can see three dependent models on the left and the concrete dependent elements on the right. Exemplarily, the dependent elements of the *Extract Track* role mapping are expanded. Figure 9.15 shows examples for the inverse and mapping dependencies: The dependencies to the RefSpec and to the role mapping model are inverse, whereas the dependency to the conference metamodel is a mapping dependency, since the role mapping model of *Extract Track* maps roles to conference metaclasses.

One final question remained open: How to detect an initial refactoring and how to initiate the refactoring stream? Transparently to the MLDE user, an implementing class of the `org.eclipse.ltk.core.refactoring.history.IRefactoringHistoryListener` interface is registered to the Eclipse refactoring history. This listener always gets notified in case a new refactoring was applied in Eclipse. Then it checks if the refactoring descriptor is an instance of the `ModelRefactoringDescriptor` already described in Sect. 9.1.7. Amongst others, it is used to add an executed refactoring to the refactoring history. In case the notification was initiated on a `ModelRefactoringDescriptor`, Refactory can be sure that it is a model refactoring of its own. Thus, the point in time to start the refactoring stream is detected. A `CoRefactorerFactory`

determines all dependent models from the DK-Base. Furthermore, the according Co-RefSpecs for each of the dependent models are collected from the CoRK-Base. For every combination a `CoRefactorer` is created. The `CoRefactorer` contains the according `IRefactorer` for the initiating and the dependent model, respectively. Prior application of the co-refactoring, the condition expression is checked and, if satisfied, the binding expression is evaluated by MVEL as explained previously. After successful application of a co-refactoring, it is again persisted in the refactoring history. As a consequence, this workflow starts again until no co-refactorings can be determined anymore or a cycle is detected. If an error occurs it will be communicated to the user, but only the failing co-refactoring is rolled back. Thus, not the whole refactoring stream is reverted and the user has a more fine-grained focus on the failures instead of starting the whole refactoring stream from new.

9.4. Conclusion

In this chapter, the feasibility of the proposed approaches regarding the generic refactoring, quality smells and co-refactoring has been demonstrated. As a result, the Eclipse-based tool Refactory has been developed which is open-source and publicly available. As a main objective, the user expectations regarding the application of refactorings have been in focus. At all costs, it should be possible that refactorings can be applied in a accustomed manner: A refactoring must be applicable by a selection of the desired elements and a click in the *Refactor* context menu, as it is known from the Eclipse platform. It has been shown that Refactory seamlessly integrates into the existing refactoring workflow of the platform and, therefore, uses existing extension points and methodologies. Co-Refactorings are executed transparently to the user.

Furthermore, means for extensions are provided, such as the registration of new editor connectors in order to support other kinds of editors, or constraint interpreters to enable the evaluation of pre- and post-conditions other than OCL-based constraints. In addition, structure-based quality smells can be registered by means of IncQuery patterns which emerged as a mature mechanism for detecting structural quality smells in EMF-based models. Metrics-based quality smells can be registered by means of providing a class which implements the `Metric` interface of Refactory. The extension mechanisms of the expression language MVEL could be exploited to implement the evaluation of condition and binding expressions for the co-refactoring scenario. Hence, language-specific modifications can be easily declared in a fashion similar to the Java programming language.

In summary, the implementation of Refactory fulfilled some last requirements, such as the support for pre- and post-conditions (Requirement 5 on page 25), the comfortable specification of structure-based quality smells (Requirement 5 on page 33) or the language-specific binding definition for co-refactorings (Requirement 6 and Requirement 8 on page 39). Refactory contributes the first tool which incorporates quality smells, resolving model refactorings and co-refactorings. With the help of our tool, quality-related model deficiencies can now be detected and resolved easier. If all required information for applying refactorings and co-refactorings are provided beforehand, quality smells can now be resolved automatically.

10 Evaluation

10.1. Case Study: Reuse of Generic Refactorings in many DSLs

To evaluate the feasibility of our generic refactoring approach, we collected refactorings for different modelling languages and implemented them according to the procedure presented in Sect. 4.2 and with the help of the Refactory editors illustrated in Sect. 9.1. A test suite has been realised to validate the execution of refactorings automatically. Therefore, a model intended to be refactored and an expected model have to be provided. A particular refactoring then is applied to the former and the result is compared to the expected model with the help of EMF Compare. Thus, the test suite can easily be extended just by providing more models and a new entry in the configuration file regarding which refactoring has to be applied.

The goal of this evaluation is to collect information about the number of specific refactorings that can benefit from reusing a generic refactoring. In addition, the question how many refactorings would require specific extensions to the generic refactoring should be answered.

10.1.1. Threats to validity

The validity of our evaluation is influenced by multiple factors. First, the selection of metamodels heavily accounts for the applicability of generic refactorings. Thus, we used languages of different complexity and maturity for our evaluation. The metamodels for UML, BPMN, Java and Timed Automata exposed the highest number of classes and structural features. Other languages (e.g. OWL, Concrete Syntax and Ecore) have a medium complexity. Some metamodels (e.g. PL/0, AppFlow, Role Models or Feature Models) were rather small and taken from the EMFText Syntax Zoo.[1] We are aware that this selection of metamodels is by no means exhaustive, but we are still convinced that it supports the idea of generic refactorings.

Second, the selection of refactorings (both generic and concrete) has an impact on our evaluation. To obtain a representative result, we tried to cover as many refactorings as possible.

[1] `http://www.emftext.org/zoo/`

Also, we collected concrete refactorings from catalogues to make sure that the evaluation is not biased towards our approach. However, the list of refactorings can never be complete, which is why the results of our evaluation must always be considered with respect to the languages and refactorings under study.

Third, our evaluation is quantitative in its nature. We count the number of mappings that were established, but we did not measure the time or the skills required to do so in comparison to the direct creation of concrete refactorings. From a practical perspective, this is an important point, because DSL designers are not only concerned about the applicability of our approach, but also about the effort that is required to use it.

10.1.2. Results

The concrete results of our evaluation can be found in Table 10.1. The complete list of all defined refactorings is depicted in Table C.1 in the Appendix C. The metamodels to which refactorings were applied are depicted as columns, the generic refactorings form the rows. The numbers in the cells denote how often the generic refactoring was mapped to the metamodel. Underlined numbers indicate that the according mappings required post-processors to augment the generic refactoring with additional transformation steps. The table is divided into two parts. The upper part indicates those role mappings which benefit from reusing generic refactorings. The graphical representations of the according role models can be seen in Appendix A. The role mappings in the lower part denote role mappings being defined specifically for particular target metamodels. In the following, we will discuss the upper part first and the lower part afterwards.

In total, we created 96 mappings by applying 27 generic refactorings to 18 metamodels. The generic refactorings that were reused most often are *Rename X* and *Extract X with Reference Class*. The latter is an extended version of *Extract X*.

In Sect. 3.1, we have identified two types of limitations of previous works. The first limitation—no reuse of refactorings across languages—applies to approaches that define refactorings on MOF layer M2. When looking at the upper part of Table 10.1, one can see that each generic refactoring was applied to at least two metamodels. Some of them were even applicable to the majority of the languages under study.

The second limitation—having a single fixed mapping for each metamodel—was observed for approaches that reside on the MOF layer M3. All cells from Table 10.1 that contain a value higher than 1 indicate cases where our approach is superior to existing approaches. Here, the same generic refactoring was mapped multiple times to a metamodel to obtain different specific refactorings.

The lower part of Table 10.1 shows two kinds of refactorings. The first one is covered by the generic *Select X* refactoring. It contains only one role which is meant to be used as an entry point so that Refactory can take effect upon an editor selection. The intrinsic modifications then must be implemented in Java with respect to a domain-specific API. This is the reason why all eight concrete refactorings in this row are realised by means of post-processors. The seven refactorings of Java are those presented in the quality smell catalogue in Sect. 7.4. The second kind can be observed in the other rows of the lower part. As can be seen, each generic refactoring is instantiated only once. These refactorings can be considered as specific for the particular target language they have been defined for. This means that the role models are a structural abstraction of the particular refactorings, which have not been reused for other languages. But

Table 10.1.: Refactorings applied to metamodels.

	OWL	Java	UML	AppFlow	Conference	Forms	PL/0	SimpleGUI	TextAdventure	ConcreteSyntax	Company	Office	Role Models	Sandwich	TimedAutomata	Ecore	BPMN 2	Feature Models
Rename X	2	1	1	1	1	1	3		2	3	3	2	1	1	1	1	1	1
Extract X with Reference Class	2	1	1	1	1	1	1	1	1	1								
Move X	1	1	4				1									1		
Extract X			2													5		
Introduce Reference Class			2													1		1
Extract Sub X			1								1					1	1	
Remove Empty Contained X	1	1	1															
Move X loosely	1					1						1						
Remove Contained X	2															1		
Remove Unused Contained X		1	1															
Duplicate With Reference	1															1		
Select X	1	7																
Create Referenced Elements	1																	
Introduce Inverse Reference In Container	1																	
Introduce Simple Reference Class				1														
Introduce Inverse Reference																1		
Convert X																1		
Replace Feature In Container																1		
Inline X																1		
Re-reference X	1																	
Introduce Class And Reference	1																	
Extract X Loosely	1																	
Introduce Referrer To All X	1																	
Replace Feature	1																	
Simple Move X	1																	
Extract X from Children			1															
Slow For Loop		1																

by using our approach, it is possible to reuse them, e.g., for the specification of quality smells. Thus, they can be declared as resolving refactorings of quality smells and the finer grained such a role model is the more specific the element bindings of the roles of interest of a quality smell can be mapped to them (cf. Sect. 6.2.1).

Even with respect to the restricted number of languages and refactorings that were evaluated, we think that the results support our approach. First, we were able to instantiate many specific refactorings from few generic ones, which shows that reusing generic refactoring specifications is beneficial. Second, the low amount of customisation that was needed (16 out of 96 refactorings), indicates that most refactorings can be instantiated without additional effort, besides specifying a role mapping. Furthermore, we have seen that not only reusing generic refactoring specifications is beneficial. On the one hand, mapping one rudimentary role model (such as *Select X*) can be considered as entry point to the whole Refactory infrastructure. Thus, DSL designers using this alternative only need to implement the particular modifications by means of their accustomed language-specific API. All the other features of Refactory are then delivered automatically. On the other hand, realising domain-specific refactorings by specifying them with a role model and a RefSpec, results in the fact that it can be integrated into the whole quality smell and co-refactoring frameworks and mechanics. Thus, it can be defined which qualities they improve by resolving particular quality smells, or it can be declared as part of coupled refactorings in the co-refactoring scenario.

However, the conclusions drawn above apply first of all to the selection of languages and refactorings of our evaluation. That means we only evaluated a subset of all possible refactorings (cf. Sect. 5) in the respective DSLs and do not present a complete list. Especially, in complex languages, such as the UML or Java, we expect more refactorings which have to be customised with post-processors. Reasons for this assumption are that language-specific semantics, such as type inference for Java, cannot be generalised structurally. Nonetheless, our approach fosters the reuse of generic refactorings and its specifications across different languages as it was claimed in [MTM07].

An additional discussion about the effort of instantiating a generic refactoring can be found in the following section.

10.1.3. Experience Report

As shown above, our approach is suitable to reuse generic refactorings for various different DSLs. Since we are using a model-based approach, one might wonder how much effort is needed to instantiate a generic refactoring for the metamodel of choice in contrast to implementing an appropriate transformation in a common transformation language. Based on our experiences, we can say that specifying a role mapping to a certain structure of a DSL's metamodel is not very difficult, when the DSL designer has a good overview about the available generic refactorings and their purposes. In a situation when she identified a refactoring being useful for her language, defining the role mapping is straight forward. As an example, we consider the refactoring *Extract CompositeState* for UML again (cf. Fig. 4.6 (c) on page 59). It moves some selected states into a newly created composite state. At a point when it is clear that some states are intended to be moved to a new extracted composite state, the role mapping in Fig. 4.6 (c) is easy to achieve since the DSL designer knows the metamodel and has identified a generic refactoring which can be used for it.

This mapping is sufficient to get a new entry in the context menu of an Eclipse environment which then invokes this refactoring. Beyond the generic transformation steps in this special case of state machines, some more steps have to be executed. Namely, besides moving the original states into a new composite state, the incoming and outgoing transitions must be calculated. Those steps cannot be modelled generically and, thus, must be implemented in a post-processor. As mentioned in Sect. 9.1.5, such a post-processor must be implemented in Java and accesses the EMF-generated API of the particular metamodel. The concrete post-processor of this particular refactoring is presented in Listing E.9 in Appendix E on page 183. It can be seen that the whole class contains 73 lines of code. From our experience, we can say that this is a rather complex post-processor.

Another example for a complex post-processor is the refactoring *Extract Method* for Java, since the type inference of local variables of the original methods and depending resulting parameters in the new extracted method cannot be handled generically. In such cases, the effort is relatively large. But, since a post processor is implemented in Java, every common transformation language exposing a Java interface can be invoked. In particular, this means that refactorings for which very language-specific transformation steps are needed, such as type inference, we recommend to implement the transformation as post-processor in a transformation language of choice, as we did for the resolving refactorings presented in Sect. 7.4. In this case, one can still benefit from our approach and our tool Refactory, because refactorings are seamlessly integrated into the Eclipse IDE and respective model editors. Once the role models are mapped to the target metamodel and the post-processors are implemented, Refactory provides commands (one for each refactoring) in the context menu of the particular model editor. These commands are displayed context-sensitively depending on the metaclasses of the currently selected elements. As roles are mapped to metaclasses, such a selection identifies which refactorings are applicable in a certain context or not.

According to our observations, for larger models most time is spent for printing the refactored model back to the editor's presentation mode. This results from the fact that converting textual models into graphs and vice versa is very time consuming. Since this is not a question of the refactoring execution itself, we do not provide any detailed evaluation numbers.

10.2. Case Study: Suggestion of Valid Role Mappings

In Chap. 5, our approach regarding the suggestion of valid role mappings has been presented. As already mentioned, the task of identifying potential valid role mappings is not trivial, especially if one is not used to it. For this purpose, we explained how it is possible to find all potential role mappings of a role model in a particular metamodel based on graph querying with GUERY.

The goal of this evaluation is to investigate, how often a role model can be mapped in different target metamodels of different complexity. Since the revealed results show that the amount of potential valid mappings can be huge, further numbers are provided illustrating the needed effort to still get feasible suggestions.

10.2.1. Implementation

At first, we want to provide some numbers that show the complexity of the different target metamodels under study. For this purpose, consider Table 10.2. As can be seen, the same

18 DSLs are used as in Table 10.1 from Sect. 10.1.2. Remember that a metamodel is translated on demand to a graph representation the GUERY engine can operate on to find valid matches. Therefore, we implemented a graph adapter which basically creates a vertex for each metaclass and determines the needed edges which then are added to the graph as well. Since metamodels make use of the inheritance semantics between metaclasses, edges have to be created dependent on the references of super-metaclasses. Thus, if an incoming edge of a vertex is created and the corresponding metaclass has sub-metaclasses, then this edge has to be propagated to the corresponding sub-vertices, too.

Having this in mind the table columns have the following meaning.

MC denotes the number of metaclasses of the particular DSL.

SF stands for the total number of structural features in a metamodel.

SF/MC provides the ratio of structural features per metaclass.

Vertices shows the number of vertices of the graph to be queried. If these numbers differed from the according numbers in the MC column, something would be wrong.

Edges denotes the total number of edges in the graph. As explained before, edges must be propagated along the inheritance hierarchy to cover all possible paths.

E/V then expresses the ratio of edges per vertex.

Table 10.2.: DSL Complexity.

Metamodel	MC	SF	SF/MC	Vertices	Edges	E/V
UML	242	594	2.45	242	229,821	949.67
Java	237	120	0.51	237	5629	23.75
BPMN 2	138	458	3.32	138	3170	22.97
TimedAutomata	77	124	1.61	77	864	11.22
ConcreteSyntax	47	65	1.38	47	468	9.96
OWL	67	97	1.45	67	648	9.67
Ecore	20	81	4.05	20	133	6.65
SimpleGUI	7	4	0.57	7	42	6.00
AppFlow	20	34	1.70	20	63	3.15
Role Models	13	21	1.62	13	39	3.00
PL/0	25	36	1.44	25	54	2.16
Feature Models	7	24	3.43	7	15	2.14
Company	3	18	6.00	3	6	2.00
TextAdventure	8	9	1.13	8	14	1.75
Forms	10	13	1.30	10	11	1.10
Office	5	4	0.80	5	5	1.00
Conference	9	10	1.11	9	8	0.89
Sandwich	12	4	0.33	12	9	0.75

Especially the ratios *SF/MC* and *E/V* should give information about the complexity of the used languages. Therefore, the table is sorted by this column. A high *E/V* value signifies, e.g., a deep inheritance hierarchy, which the edges are propagated along. Of course, the numbers are only relative since a certain metaclass can have no references to others while a different metaclass can have a very large number of references. But these ratios illustrate the characteristic properties and will help in the following to interpret the results.

For this study, we focussed only some of our developed role models which we consider most reasonable. The role model for the generic *Rename X* refactoring is not part of this set, since it consists only of one role having a role attribute. It would not make sense to let this role model be subject of the graph querying, since it matches every metaclass having an attribute. The following role models are used: *Extract X with Reference Class* (Fig. A.1 (b) on page 169), *Move X* (Fig. A.2 (a) on page 169), *Extract Sub X* (Fig. A.3 (b) on page 170), *Extract X* (Fig. A.2 (b) on page 169), *Introduce Reference Class* (Fig. A.3 (a) on page 170) and *Remove Unused Contained X* (Fig. A.5 (a) on page 170). As explained in Chap. 5, these role models are automatically converted to GUERY queries and then tried to be matched. We restricted the matching process in the sense that only paths of length 1 are to be matched because longer paths can be very time consuming especially for languages of higher complexity. The Table 10.3 shows the result of this matching process.

The numbers in the cells denote the amount of determined matches corresponding to valid role mappings. The underlined numbers do not represent the final result but a previously maximum value which we set for being able to interrupt the process. Furthermore, these numbers are by far not feasible anymore to provide the found matches as suggestions to the DSL designer. As can be seen in the table, the determined amounts are really high for complex languages such as UML, Java or BPMN. UML is the language with the highest values which stems from the fact that it is most complex, since its *E/V* ratio is 949.67. Obviously, these numbers are not suitable to present the determined role mappings as suggestions. To still obtain some practical useful recommendations for role mappings, two initial restrictions on the computation of role mappings have been placed. First, we constitute that each collaboration is mapped to a single reference in the metamodel only. We do not allow mappings to paths of references, to prevent matching explosion. This was already applied to the found mappings in Table 10.3, since the maximum path length was set to 1.

Second, the number of role mappings is reduced by restricting the mapping between roles and metaclasses. If a role is mapped to a metaclass having subclasses, we omit the mappings to these. Thus, we exclude separate mappings for each of the subclasses. Since subclasses are specialisations of their superclass, mapping a role to the superclass yields a more general refactoring (role mapping) compared to mapping the role to one of the subclasses. However, one must admit that potential relevant matches are lost due to this restriction. For example, if a subclass provides a feature (e.g., a reference) that is required to map a collaboration from the role model, the respective role mapping is not found.

By these restrictions the complexity of the metamodels under study are reduced partly drastically. Therefore Table 10.4 provides adjusted numbers with respect to the restrictions. It is again sorted by the *E/V* column and it can be observed that the order changed compared to Table 10.2. Especially, the extremely high *E/V* ratio for the UML was reduced from 949.67 to 19.37. Furthermore, the value of Java decreased to 1.35 which will hopefully result in better numbers for the determined valid role mappings. Based on the above restrictions, the queries which represent

Table 10.3.: Automatically determined role mappings with maximum path length of 1.

	Extract X With Reference Class	Move X	Extract Sub X	Extract X	Introduce Reference Class	Remove Unused Contained X
UML	1,000,000	100,000	49,376	1,000,000	100,000	5794
Java	1,000,000	1022	1924	0	0	5517
BPMN 2	1,000,000	5506	19,107	78,782	11,124	1874
TimedAutomata	3	0	0	0	1	5
ConcreteSyntax	132,068	17	2555	0	112	399
OWL	88,234	294	2223	92	10	561
Ecore	893	146	35	87	345	47
SimpleGUI	1512	63	0	0	18	21
AppFlow	0	0	0	0	0	0
Role Models	38	0	0	0	17	14
PL/0	137	0	5	0	0	43
Feature Models	35	6	3	6	8	7
Company	0	0	0	0	0	0
TextAdventure	2	0	0	0	2	7
Forms	11	0	0	0	1	10
Office	0	0	0	0	0	3
Conference	2	0	0	0	0	6
Sandwich	0	0	0	0	0	5

the role models are tried to be matched to the adjusted graphs of the reduced metamodels again.

To explain the results gained using the recommendation engine, consider the generic *Extract X* refactoring (cf. Fig. A.2 (b) on page 169) and the UML metamodel explained in [OMG11a]. Our suggestion engine calculated 16,866 valid matches. That means that *Extract X* can be mapped at least 16,866 times to the UML metamodel, with respect to the restrictions stated before. This is in fact a very high number, which makes it impossible for a DSL designer to pick suitable mappings. As a consequence, we extended the process of suggesting role mappings in such a way that the possible results are reduced, if the DSL designer maps one role to one metaclass manually. By this extension, only the matches which contain the manually provided mapping remain. This strategy reduces the search space significantly. To give an example, let us consider the generic *Extract X* refactoring and the UML metamodel again.

If a DSL designer wants to map this role model and manually maps role `Extractee` to the metaclass `Region`, only one valid match remains. All the other roles and collaborations can be mapped automatically. The complete role mapping which can be derived from this manual mapping can be seen in Listing 10.1. The concrete resulting refactoring can be named *Extract StateMachine in Interface* because a `Region` in a `ProtocolStateMachine`, which defines the

Table 10.4.: DSL Complexity reduced due to omitting sub-metaclasses.

Metamodel	MC	SF	SF/MC	Vertices	Edges	E/V
UML	242	594	2.45	242	4687	19.37
BPMN 2	138	458	3.32	138	1228	8.90
Ecore	20	81	4.05	20	82	4.10
Feature Models	7	24	3.43	7	15	2.14
Company	3	18	6.00	3	6	2.00
OWL	67	97	1.45	67	130	1.94
Role Models	13	21	1.62	13	24	1.85
TimedAutomata	77	124	1.61	77	111	1.44
SimpleGUI	7	4	0.57	7	10	1.43
Java	237	120	0.51	237	320	1.35
AppFlow	20	34	1.70	20	26	1.30
ConcreteSyntax	47	65	1.38	47	57	1.21
PL/0	25	36	1.44	25	29	1.16
TextAdventure	8	9	1.13	8	9	1.13
Conference	9	10	1.11	9	6	0.67
Forms	10	13	1.30	10	6	0.60
Office	5	4	0.80	5	3	0.60
Sandwich	12	4	0.33	12	6	0.50

protocol of an `Interface`, is extracted to a new `StateMachine` which then is referenced as `extendedStateMachine` in the original one. By mapping a single metaclass to a role, the complete mapping can be derived.

On the contrary, there are also manual mappings which lead to reduced result sets that are still quite large. For example, mapping role `OriginalContainer` to the metaclass `Behavior` [OMG11a, p. 356] leads to 4581 remaining valid matches. As shown by this example, manually mapping a role to a metaclass can reduce the resulting possible mappings significantly but they may still be of too much quantity to make suitable suggestions. In such a case, we assume that the DSL designer wants to restrict the valid matches further and, thus, continues with a next manual mapping of a role to a metaclass. As a consequence, the previously filtered result set is restricted by another manual mapping and the resulting set of valid matches decreases incrementally. We applied this strategy to the valid matches as they are calculated by the graph query engine. The results of a reasonable subset of the languages from Table 10.1 can be found in Table 10.5 and are evaluated and discussed in the following section.

10.2.2. Evaluation and Discussion

Table 10.5 contains pairs of numbers in each cell. The first number in a cell designates the quantity of all valid matches, i.e. all possible complete role mappings with respect to the restrictions explained previously. As far as our experience goes, we think that tool support is only useful for the DSL designer if at most 20 suggestions can be provided by our engine. A larger number of valid matches would not be perceived to be feasible, because too many suggestions have to be

Table 10.5.: Number of possible matches (first number in pair) and average needed manual mappings to get a result set of at most 20 matches (second number in pair).

	Move X (3)	Introduce Reference Class (3)	Extract X (4)	Extract X With Reference Class (5)	Extract Sub X (3)	Remove Unused Contained X (2)
UML (MC: 247, SF: 586, E/V: 19.37)	10,793;2.69	11,944;2.68	16,866;3.51	89,068;4.33	4197;2.57	1440;1.82
Ecore (MC: 20, SF: 81, E/V: 4.10)	74;1.66	45;1.38	13;0	208;3.13	41;1.45	42;1.0
Feature Models (MC: 6, SF: 26, E/V: 2.14)	6;0	9;0	6;0	35;1.98	3;0	7;0
OWL (MC: 67, SF: 97, E/V: 1.94)	30;1.0	2;0	2;0	21;1.0	6;0	99;1.35
Timed-Automata (MC: 77, SF: 124, E/V: 1.44)	0;0	13;0	0;0	3;0	1;0	5;0
Java (MC: 233, SF: 121, E/V: 1.35)	0;0	5;0	0;0	407;2.61	158;1.97	309;1.38
Concrete-Syntax (MC: 47, SF: 65, E/V: 1.21)	4;0	5;0	0;0	10;0	2;0	40;1.0
PL/0 (MC: 25, SF: 36, E/V: 1.16)	0;0	0;0	0;0	3;0	2;0	24;1.0

```
ROLEMODELMAPPING FOR
<http://www.eclipse.org/uml2/3.0.0/UML>

"Extract State Machine in Interface" maps <ExtractX> {

  ContainerContainer := Interface{
    source := protocol;
    target := nestedClassifier;
  };

  OriginalContainer := ProtocolStateMachine{
    extracts := region;
    reference := extendedStateMachine;
  };

  NewContainer := StateMachine(newName -> name){
    moved := region;
  };

  Extractee := Region;
}
```

Listing 10.1: Derived role mapping *Extract StateMachine in Interface*.

examined by the DSL designer. This procedure is time consuming and not very supportive in finding a feasible role mapping. For this reason, the second number in each cell represents the average count of manual mappings of a role to a metaclass a DSL designer has to specify, in order to reach a result set containing at most 20 valid matches. To better compare the given quantities, we expose the *MC*, *SF* and *E/V* values of each metamodel in the row headings of Table 10.5 again. The values of the reduced complexity in Table 10.4 are used according to the made restrictions. Furthermore, besides the name of the role model, each heading in a column contains the number of roles in the particular role model.

As can be seen in the table, there are a lot of very different numbers, although the complexity of some metamodels is similar. Consider, e.g., the numbers of the TimedAutomata and the OWL languages. The metaclasses and structural features count of the former is slightly higher than these from the latter. However, the amount for the role model *Remove Unused Contained X* is much higher for OWL. In contrast to this observation, the quantity of valid matches for *Introduce Reference Class* is higher for TimedAutomata. Such controversial observations can be made throughout the whole table.

For example, consider the metamodels of Ecore and Java. Although Ecore has fewer metaclasses and structural features than Java, it has much higher numbers for the valid matches of the role models *Move X*, *Introduce Reference Class* and *Extract X*. This would speak in favour of the higher *E/V* ratio of Ecore, but the numbers for the other three role models are the other way around, which cannot be explained with the *E/V* value anymore. It can be also noticed that the average manual mapping count for those two metamodels for the role model *Extract X with Reference Class* is worse (i.e., higher) for Ecore (3.13) than for Java (2.61), even though the number of all

possible role mappings is with 208 almost one half of 407 for Java.

Another interesting fact is that the quantity of many valid matches for the Feature metamodel is much higher compared to ConcreteSyntax, TimedAutomata, PL/0 or OWL, although it is the metamodel with the lowest complexity since it has only 6 metaclasses and 26 structural features. The main reason for this is that the Feature metamodel has a relatively high *E/V* ratio compared to the others. This means that not only the number of structural features per metaclass but also the edges per vertex are quite high. This results in more possibilities for larger role models such as the one for the generic refactoring *Extract X with Reference Class*. The same can be observed for Ecore and UML. The ratio of the edges per vertex for UML is still very high with 19.37. Java has many valid matches for this role model, too, but they result from the high number of metaclasses in general.

The objective of our incremental matching strategy is to reduce all possible valid matches for supporting the DSL designer in specifying a role mapping. The contents of Table 10.5 can be interpreted as follows. The best results are achieved for languages, where the overall number of matches (i.e., the first number) is greater than 0 and less than 20. Such cases indicate that no manual mappings need to be specified to get full suggestion support. The DSL designer simply needs to select one of the suggested role mappings. This holds for every role model for TimedAutomata (excluding the role models that cannot be matched). Some other languages almost achieve such results: OWL, Concrete Syntax and PL/0. In these cases, only few role models require to map one role in order to get less than 20 valid matches.

In total, there are 48 cells in Table 10.5, each representing a metamodel/rolemodel pair. 20 of these pairs have less than 20 possible matches, 5 cells do indicate more possible matches, but these can be reduced to under 20 solely by mapping one role manually. All other cells represent cases where more than 1 role must be mapped manually. But only in 7 cases more than 2 mappings have to be established manually to reduce the set of valid matches appropriately. Thus, we can summarise that suggestion support is feasible here.

Compared to the number of existent roles in the particular role models, the average numbers of manual mappings for the metamodels of Java and especially UML are quite bad. Consider, e.g., UML and *Extract X with Reference Class* where we find an average of 4.33 manual mappings to yield a result set of at most 20 valid matches. This value is relatively poor because it almost correlates with the number of 5 roles in the role model. That means that, in average, one must almost map five roles to get a role mapping derived which contains five roles which is not supportive for the DSL designer. At least, the role collaborations are mapped automatically. The average numbers for the other role models are analogous. At first sight, those numbers really seem not supportive. But, on the other hand, matching every role model to every metamodel always yields cases in which manually mapping a role to a metaclass once results in a set of less than 20 valid matches. That means, there are always situations in which the DSL designer only needs to do one manual mapping and the suggestion engine can provide support by calculating the few resulting valid matches. This is also the case for complex metamodels such as UML or Java.

Furthermore, it is obvious that one concrete target role mapping can be derived from different possibilities of manual mappings, let it be only one or incrementally more than one which yield the same role mapping. One interesting fact here is that some variants of manual mappings need less manual mappings than others to produce a manageable set of valid matches containing the target role mapping. In other words, it depends on the order of mapping roles to metaclasses.

Remember, for example, the previously derived role mapping in Fig. 10.1 from manually mapping the role `Extractee` from the role model *Extract X* to the metaclass `Region` from the UML. The same role mapping can be obtained if the incremental matching strategy is processed in the following order of manual mappings:

1. `NewContainer := StateMachine`
2. `ContainerContainer := Interface`
3. `Extractee := Region`

As one can see from this example, the same results can be achieved in different ways. The reason for this is that the metaclass `Region` is referenced (in terms of a containment reference) less times from other metaclasses (4) than the metaclass `StateMachine` references other metaclasses (23). Thus, more possibilities remain when starting with the mapping from `NewContainer` to `StateMachine` than with the mapping from `Extractee` to `Region`. This is an important fact which must be taken into consideration by DSL designers. More supportive results can be achieved by mapping roles to metaclasses holding relations to other metaclasses which correspond to the role collaborations and, more importantly, which are few in quantity. Thus, it is obvious that if a role model is to be matched onto a structure of a certain metamodel which reflects the same structure as the role model, only a single role mapping is possible for this structure. The high number of structural features per class in the UML metamodel, hence, is the reason for the extremely high quantities of possible valid matches.

To summarise, we can say that suggestion extremely depends on the complexity of the target metamodel. If there is a high number of metaclasses and a high number of structural features per metaclass the DSL designer only gets constructive support if she maps some roles manually to reduce the resulting set of valid matches. Furthermore, it can be useful to map roles to metaclasses having, first, relations to other metaclasses which correspond to the role collaborations and, second, having a number of relations which correlate approximately to the number of role collaborations. But, we are aware that the latter is not always possible. If a DSL designer exactly knows which role to manually map to which metaclass it might be the case that she does not need tool support at all. In this case she can map the whole role model by herself.

Beyond that, it must be emphasized that the determination of all valid role mappings needs to be performed only once per metamodel version. The result then can be persisted into a database, e.g., and the suggestion engine than can query the database instead of determining potential mappings on demand. This must be avoided at all cost, especially for complex metamodels.

10.3. Proof of Concept: Co-Refactoring OWL and Ecore Models

As a last evaluation scenario, a proof of our co-refactoring concept and implementation is provided in this section. Here, we want to return to the running example of Sect. 8.2: ontology-driven requirements engineering and development. Consider again the process illustrated in Fig. 8.3 on page 107. We argued that the ontology serves as base for successive transformations. Hence, domain models in terms of Ecore can be generated from the ontology. We restrict the presentation of this proof of concept to the situation where an Ecore model is generated from an

ontology encoded in the OWL syntax. For being able to achieve this on the base of EMF, we use the textual DSL for OWL specified with EMFText[2] in order to consider OWL artefacts as models.

In the following, we describe seven coupled refactorings between OWL and Ecore models. Afterwards, the concrete realisation of the respective Co-RefSpecs is shown before we close with a discussion.

10.3.1. Coupled OWL-Ecore Refactorings

The following coupled refactorings between OWL and Ecore models have been jointly discovered and defined with Tittel in [Tit11].

Ontologies are used in the IT to define concepts and relations between them in a particular domain. In contrast to domain models, an ontology also contains the instances of such concepts which can hold specific relations to other constituents of the ontology again. Thus, ontologies are used to represent knowledge and to reason about it.

The main concepts of ontologies are *classes*, *individuals* and *properties* [Hor08; Tit11]. Classes denote the concepts of a domain, whereas individuals represent instances of them. Relations between all constituents can be specified by properties. Thus, ontologies are related to domain models, but are more expressive.

When ontologies are the base of an ontology-driven development process, they are most likely subject to evolution; especially, it must be supported to refactor them. As a consequence, domain models which were generated from an ontology must stay consistent in such a scenario. Therefore, coupled refactorings between OWL ontologies and Ecore models are presented in the following.

Renamings

Renaming ontology elements yields the renaming of the according elements in the Ecore model. Since the *name* of an ontology itself has a slightly different meaning than the name of ontology classes or properties, these cases are distinguished. The name of an ontology is considered to be the identifier for the outer world. Thus, an ontology is not meant to be just a local artefact. Therefore, the name can be considered as a unified identifier (which must be unique) in contrast to the names of classes and properties. Their names must be unique locally in the ontology, but are distinguishable from the outside with respect to the ontology name. Hence, on the one hand, the renaming of an ontology corresponds to the renaming of the metamodel contained in an Ecore model. In contrast, renamings of classes in an ontology yield renamings of metaclasses in an Ecore model.

Extracting an OWL Class

As already mentioned, relations between ontology concepts are represented by properties. Properties are first class constituents in ontologies, but are reflected by references in metamodels. In OWL, such relations are represented by *object properties* having a *domain* and *range* referring to other classes. Such object properties can be generalised by providing a new class which

[2] `http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_OWL2_Manchester` (visited 1st March 2015)

becomes the new domain of the object property. Then, the previous domain class must subclass the new one. Thus, such an extraction of a new OWL class corresponds to the extraction of a super-metaclass in a metamodel.

Pulling up a Property

Similar to the extraction, an ontology property can be pulled up to an existing class. The difference to the extraction is that the class already exists. Pulling up a feature in the metamodel to an existing metaclass is the corresponding refactoring.

Introducing an Inverse

When classes in an ontology are related via an object property, this property is directed from the domain to the range. To realise the other direction, a new object property is created having the reversed domain and range set. This modification is reflected in metamodels by adding a new reference in the target metaclass of the reference belonging to another metaclass. Both references then must be declared as their inverse.

Class Duplication

To introduce a new ontology class which has the same properties as another, but which should be refined further, the other class can be duplicated and a new name must be set. The *subClassOf* relations must be incorporated as well. The same procedure should be applied for metamodels.

Conversion of Data to Object Properties

Ontology classes can also be related to simple data types such as strings. Therefore, a *data property* defines the class as domain and the simple type as range. In case this relation to the simple data type is not expressive enough anymore, it should be converted into an object property which then can be enriched by other properties. This refactoring is similar to the *Replace Data Value with Object* refactoring proposed in [FBB+99], for which reason it is the coupled refactoring in the metamodel.

10.3.2. Realisation

In the following, the Co-RefSpecs of the presented coupled refactorings are shown. They are pretty much straightforward and most often, domain-specific role models were used for the role mapping. We denote the mapped role models in the captions of the following listings for the purpose to be looked up in the Table 10.1 in Sect. 10.1.2. For the captions the following syntax pattern is used: *<OWL refactoring>* (<mapped role model for OWL >) ⇒ *<Ecore refactoring>* (<mapped role model for Ecore>).

The depicted listings represent the Co-RefSpecs of the defined coupled refactorings from the previous section. As can be seen in Listing 10.5, the Co-RefSpec for the pull-up co-refactoring has no binding expression specified. The reason is that ontologies can be more expressive than domain models specified with Ecore. Therefore, no assumptions about the potential target of the

```
CoRefSpec for <http://www.eclipse.org/emf/2002/Ecore>
import owl:<http://org.emftext/owl.ecore>
{
   incoming refactoring owl:<Rename_Ontology>
   outgoing corefactoring <Rename_EElement> $
      OUT.Nameable.name = IN.Nameable.name
   $
}
```

Listing 10.2: *Rename Ontology* (Rename X) ⇒ *Rename EElement* (Rename X).

```
CoRefSpec for <http://www.eclipse.org/emf/2002/Ecore>
import owl:<http://org.emftext/owl.ecore>
{
   incoming refactoring owl:<Rename_Element>
   outgoing corefactoring <Rename_EElement> $
      OUT.Nameable.name = IN.Nameable.name
   $
}
```

Listing 10.3: *Rename Element* (Rename X) ⇒ *Rename EElement* (Rename X).

```
CoRefSpec for <http://www.eclipse.org/emf/2002/Ecore>
import owl:<http://org.emftext/owl.ecore>
{
   incoming refactoring owl:<Extract_Superclass>
   outgoing corefactoring <Extract_Super_Class>$
      OUT.NewContainer.name = IN.NewContainer.name;
   $
}
```

Listing 10.4: *Extract Superclass* (Extract Loosely X) ⇒ *Extract Super Class* (Extract X).

```
CoRefSpec for <http://www.eclipse.org/emf/2002/Ecore>
import owl:<http://org.emftext/owl.ecore>
{
   incoming refactoring owl:<Pull_Up_Property>
   outgoing corefactoring <Pull_Up_Feature>
}
```

Listing 10.5: *Pull Up Property* (Re-reference X) ⇒ *Pull Up Feature* (Move X).

```
CoRefSpec for <http://www.eclipse.org/emf/2002/Ecore>
import owl:<http://org.emftext/owl.ecore>
{
   incoming refactoring owl:<Introduce_Inverse_Property>
   outgoing corefactoring <Introduce_Inverse_Reference>$
      OUT.InverseReference.name = IN.InverseReference.name;
   $
}
```

Listing 10.6: *Introduce Inverse Property* (Introduce Inverse Reference In Container) ⇒ *Introduce Inverse Reference* (Introduce Inverse Reference).

```
CoRefSpec for <http://www.eclipse.org/emf/2002/Ecore>
import owl:<http://org.emftext/owl.ecore>
{
   incoming refactoring owl:<Duplicate_Class>
   outgoing corefactoring <Duplicate_Class> $
      OUT.Duplicate.name = IN.Duplicate.name
   $
}
```

Listing 10.7: *Duplicate Class* (Duplicate With Reference) ⇒ *Duplicate Class* (Duplicate With Reference).

```
CoRefSpec for <http://www.eclipse.org/emf/2002/Ecore>
import owl:<http://org.emftext/owl.ecore>
{
   incoming refactoring owl:<Convert_Data_Property_To_Object_Property>
   outgoing corefactoring <Replace_Data_Value_with_Object> $
      OUT.NewFeature.name = IN.NewFeature.name;
      OUT.Range.name = IN.Range.name;
   $
}
```

Listing 10.8: *Convert Data Property To Object Property* (Replace Feature) ⇒ *Replace Data Value with Object* (Replace Feature In Container).

pulled up property can be made. In case no binding expression was specified, the user is asked on-demand.

For being able to test co-refactoring scenarios automatically, we implemented an extensible and highly flexible test suite. Similar to the one mentioned in Sect. 10.1, the desired models to run the tests on are defined in a configuration file. Thus, new use cases can be easily appended. One must provide an input model for both the initiating refactoring and the co-refactoring. Furthermore, for each of them an expected model must be provided, which then can be compared to the result of the refactoring and co-refactoring.

10.3.3. Discussion

In this section, a proof of the co-refactoring concepts (cf. Chap. 8) and implementation in Refactory (cf. Sect. 9.3) were presented. We have explained seven coupled refactorings regarding the ontology-driven development scenario from Sect. 8.2. The according Co-RefSpecs were realised within an extensible test suite, so that it can be validated automatically. As can be observed, these Co-RefSpecs are not complex and the creation is quite feasible for a co-refactoring engineer (cf. Fig. 8.5 on page 114).

Nevertheless, we do not claim that the presented proof of concept is all-encompassing. It covers only a small dedicated scenario, but nevertheless the described use case is fully supported by the presented solution. But we are convinced that it can be applied to other domains with little effort. An extensive evaluation is left open for future work.

11
Summary, Conclusion and Outlook

In this final chapter, the contributions of this thesis are summarised. Afterwards, a conclusion is drawn and the contributions are related with the requirements stated before. A discussion of open issues and future work closes this thesis.

11.1. Summary

In this thesis, we have presented a comprehensive approach for generic quality-aware model refactoring and co-refactoring to resolve quality smells. It is divided into three parts: 1) *Generic Model Refactoring*, 2) *Quality Smells* and 3) *Co-Refactoring*. Since no approach existed before that covers all of these parts, the discussion of related work is distinguished according to these three aspects, as well.

Chapter 3 revealed deeper insights regarding the related work in the three areas. First, we discussed other model refactoring approaches and classified them depending on the MOF layer refactorings are specified at. It became clear that the M3 approaches are generic but lack flexibility. They abstract from the potential target languages and expose a common meta-metamodel for them upon which refactorings are specified. Concepts of the target languages then are mapped to it. Since such a mapping is fix, the defined generic refactorings cannot be reused as different refactoring in the same language. Thus, the use of one dedicated common meta-metamodel compares to our approach in the sense that this meta-metamodel can be considered as one single role model, which can only be mapped once. Thus, the M3 approaches are too static. Refactorings specified at the M1 layer are defined by recording respective modifications in example models. These modifications are generalised to the M2 layer. Thus, M2 and M1 approaches share the same advantages and disadvantages. Refactorings are defined for a specific target language. Thus these approaches are flexible, but not generic. The result of this analysis was that the specification of model refactorings at one MOF layer is not satisfactory.

Second, related work regarding quality smells was discussed. We have shown that only some approaches support the detection and resolution of model deficiencies in general, while others

realise it for specific domains or qualities yet. On the other hand, approaches exist that cover the resolution only. Thus, they can only be applied in combination with other approaches. The main drawback of the related work in this area is that the relation to qualities is not explicit.

Third, the presentation of co-refactoring approaches revealed that almost none of them takes into account the detection of model dependencies. We consider this an essential constituent of a co-refactoring approach. Furthermore, many presented approaches in this area are specific for a dedicated domain or scenario. Thus, they cannot be applied for co-refactoring in general.

The first main contribution of this thesis is presented in Chap. 4: Role-Based Generic Model Refactoring. To overcome the limitations of the M3 approaches, a more flexible abstraction mechanism was found by the use of role models. In our approach, role models define potential participants of a refactoring with roles. Required structures the participants must expose are defined by collaborations between roles. Thus, a role model provides an explicit structure of the participating elements of a refactoring and, therefore, abstracts from the refactoring in contrast to the target language. The other two main parts of the thesis will heavily benefit from the concept of role models. The intrinsic transformation is specified with our refactoring specification language. It is independent of any target DSL and defines the desired modifications on top of role models. To make a generic role model available in a certain target language, a role mapping must be provided. This is to map the role model of a refactoring to a dedicated structure in the metamodel. In addition, our approach also contains a solution for the composition of refactorings. This is achieved by defining a sequence of role mappings and correlate the roles of the mapped role models. This must be applied in order to express which elements from a preceding refactoring correspond to which elements in a succeeding refactoring. Preservation of static semantics is covered by means of checking well-formedness rules as pre- and post-conditions.

In Chap. 5, we have stressed the fact that DSL designers need support for specifying role mappings, since the set of all possible valid role mappings might be huge. Therefore, a suggestion approach is presented. Therein, role models are translated into graph queries and, thus, can determine all possible role mappings in a target language. A manual mapping of a role to a certain metaclass can drastically reduce the resulting set of potentially possible role mappings and the DSL designer can get automatic support.

The second main contribution is presented in Chap. 6: Role-Based Quality Smells as Refactoring Indicator. We argue that the term *bad smell* from Fowler *et al.* [FBB+99] is too imprecise and the assumptions regarding qualities are only implicit. Therefore, we defined the new term *quality smell* and provided a conceptual framework for it. The concept of quality smell explicitly correlates model deficiencies with the qualities they deteriorate and refactorings which can potentially resolve these deficiencies. Our quality smell approach separates quality smells from their particular detection strategy to allow for independent development of both. For a quality smell, further roles can be defined which represent relevant participants of interest in a detected quality smell. The elements bound to these roles (if the quality smell is detected) can be related to roles of a resolving refactoring again. Thus, model elements causing the quality smell can be directly defined as subjects of a resolving refactoring. Detection strategies are developed by specifying calculations which can be considered as reusable calculation components. Such components then can be composed by means of propositional logics to form complex detection strategies. As components metrics-based and structure-based calculations are supported. As a concrete instantiation of this conceptual quality smell framework we present a quality smell catalogue in Chap. 7. It contains 9 structure-based quality smells which can be applied in the

domain of mobile Android development.

The third main contribution is presented in Chap. 8: Role-Based Co-Refactoring in Multi-Language Development Environments. The principal workflow of co-refactoring is that a model is refactored initially and dependent model elements have to be determined which will be co-refactored then. Regarding the model dependencies, we identified four different categories and explained how logic programming is used to detect them. We argued that for both the determination of dependent models and the application of co-refactorings dedicated knowledge bases are needed. The dependency knowledge base stores explicit model dependencies and can be queried for implicit ones. The co-refactoring knowledge base contains coupled refactorings expressing which outgoing co-refactoring must be applied as an implication of a preceding incoming refactoring. Such couplings can be specified on a generic or a specific basis. For the former, only a role model is declared as incoming refactoring. This means that for specification time it is not yet known which language the initially refactored model conforms to. For the latter, a role mapping or a composite role mapping is declared as incoming refactoring and, thus, the language is known. For the problem of defining which model elements from an incoming refactored model have to be processed in the outgoing refactoring, a binding expression can be specified. The binding expression is then evaluated by an expression language interpreter at runtime. The integration of an expression language suits perfectly our role based approach since expression languages usually can be extended. We realised this in order to support the use of our role concepts within the binding expression.

In the last two parts of this thesis we provided a reference implementation of the proposed concepts in our EMF-based tool Refactory (Chap. 9). This is used to evaluate the application of the concepts (Chap. 10).

11.2. Conclusion

In this thesis, we have shown that the concept of role models is beneficial for all of our contributions. For the generic refactoring approach, we used 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 or reference an existing one 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 can be passed to a resolving refactoring. To co-refactor models, again the used roles from an incoming refactoring are related to the used roles of an outgoing refactoring to prevent the user from having to provide any further input which is already present. Thus, we can say that role models rendered as a very powerful abstraction mechanism in the scenario of quality-aware model refactoring and co-refactoring. 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.

The evaluation in Chap. 10 supports our concepts and their realisation within our tool Refactory (cf. Chap. 9). We have shown that a huge potential of reusing generic refactorings emerged since 96 concrete refactorings could be instantiated based on 27 generic ones in 18 target languages.

Table 11.1.: Evaluation of the generic model refactoring approach regarding the fulfilment of the requirements.

Genericity (1)	Flexibility (2)	Specificity (3)	Behaviour Preservation (4)	Pre- And Post-conditions (5)	Atomicity (6)	Reversibility (7)	Specification Suggestion (8)	Application Suggestion (9)	Interoperability (10)
+	+	+	○	+	+	+	○	+	+

Therefore Hypothesis 1 could be shown to be true. Furthermore, the new term *quality smell* has been defined and conceptualised in Chap. 6. Our concept abstracts from existing quality models and supports known approaches for the determination of certain qualities by using metrics-based or structure-based approaches. The presented implementation in Sect. 9.2 shows that it is now possible to focus specific qualities to detect according quality smells and let them resolve by refactorings. This proves Hypothesis 2. In addition, Chap. 7 provided a quality smell catalogue for mobile Android applications. The catalogue explicitly correlates a model deficiency with the deteriorating qualities and resolving refactorings according to our quality smell concept. This shows the validity of Hypothesis 3. The proof-of-concept presented in Sect. 10.3 has shown that our co-refactoring approach is applicable in the domain of ontology-driven requirements and software engineering. We have to admit that the case study is quite small since we only defined seven co-refactoring specifications between OWL and Ecore models. But we are convinced that the applicability of our co-refactoring approach could be demonstrated, last but not least, by the detection mechanism of model dependencies, the specification of coupled refactorings, and the definition of bindings which are interpreted by an expression language. Thus, Hypothesis 4 could be shown to be valid.

To support the comparison of our presented approaches with the related work, we now evaluate it regarding the requirements stated in Chap. 3. First, Table 11.1 shows the fulfilment regarding the generic model refactoring requirements from Sect. 3.1.1. The *genericity* could be achieved by using role models to abstract from refactorings and specify the according transformation based on roles. The concept of role mappings then ensures *flexibility* since a role model can be mapped to whatever structure in a target metamodel, provided that it respects the structure of the role model of a generic refactoring. The concept of post-processors supports the *specificity*, since language-specific modifications can be implemented with them. The *behaviour preservation* is evaluated as neutral, since we support the detection of the violation of static semantics based on WFRs (cf. Sect. 4.3), but no final general answer could be given. For the dynamic semantics of a language, more research has to be invested. *Pre- and post-conditions*, *atomicity* and *reversibility* could be ensured by means of the implementation presented in Sect. 9.1. The *specification suggestion* was evaluated as neutral, since we provided an approach based an graph querying

Table 11.2.: Evaluation of the quality smells approach regarding the fulfilment of the requirements.

Quality Relation (1)	Refactoring Relation (2)	Refactoring Suggestion (3)	Metrics-based (4)	Structure-based (5)	Cause Tracing (6)	Language Independence (7)	Language Specifics (8)	Interoperability (9)
+	+	+	+	+	+	+	+	+

(cf. Chap. 5), but the evaluation in Sect. 10.2 has shown that the main problem is that a huge amount of potentially valid role mappings might be calculated. Thus, the DSL designer has to map roles manually to reduce the amount. The *application suggestion* conforms to the refactoring suggestion regarding quality smells and is evaluated as fulfilled, since our concept of quality smells allows for the automatic suggestion of refactorings. The *interoperability* is ensured by the fact that we rely on MOF-based languages.

Second, Table 11.2 shows the fulfilment regarding the quality smells requirements from Sect. 3.2.1. The *quality relation* and the *refactoring relation* is ensured by the new term and concept of *quality smells*. It explicitly correlates deficiencies with deteriorating qualities and resolving refactorings. Because of this correlation, the *refactoring suggestion* is also ensured. Refactorings can be recommended in case quality smells occur. *Metrics-based* and *structure-based* quality smells are supported by calculation strategies. As a proof-of-concept, we have shown how IncPL is used to specify structure-based quality smells. The result of a detection strategy references the causing elements and, thus, *cause tracing* is ensured. The approach itself is *language independent*, since we make no assumptions regarding possible target languages. Nevertheless, *language specifics* can be taken into account in the detection strategies. The *interoperability* is ensured by the fact that we rely on MOF-based languages.

Third, Table 11.3 shows the fulfilment regarding the co-refactoring requirements from Sect. 3.3.1. Our concept of Dependency Knowledge Base ensures that *dependent models* and according *dependent elements* can be determined. The requirements of *incoming refactoring*, *condition specification* and *outgoing co-refactoring* is supported by the Co-Refactoring Specification by means of ECA rules. *Dependent binding*s can be specified as a binding expression in a `RefactoringAction`. Again, *language independent* and *interoperability* is supported by the use of MOF and *language specifics* can be encoded in the aforementioned binding expressions.

As a conclusion, we can classify our comprehensive approach of *generic quality-aware refactoring and co-refactoring to resolve quality smells* into the category R-CoR-SD-Q. We cover all required aspects of refactoring, co-refactoring, deficiencies and the relation to qualities.

Table 11.3.: Evaluation of the co-refactoring approach regarding the fulfilment of the requirements.

Dependent Models (1)	Dependent Elements (2)	Incoming Refactoring (3)	Condition Specification (4)	Outgoing Co-Refactoring (5)	Dependent Binding (6)	Language Independence (7)	Language Specifics (8)	Interoperability (9)
+	+	+	+	+	+	+	+	+

11.3. Outlook

This thesis raised further questions and some issues have not been addressed yet. In the following, the remaining open aspects of the thesis are explained shortly. Afterwards, general open questions and ongoing future work are discussed.

Impact of Refactorings A first open issue is the investigation of refactoring impact from different points of view. As we discussed in Sect. 6.3, a refactoring which resolves a quality smell can in turn evoke other quality smells. This is due to the fact that qualities might be interrelated, depend on each other or are rather contradictory [CNYM00; Sai03; Koz11; Are14]. The important point is that the relations between qualities cannot be generalised. It heavily depends on the system under development, the platform it should be run on and other factors like the interaction with external components. Thus, no generic solution can be proposed. In our approach, this problem is faced in two ways. First, the responsibility is shifted to the DSL designer and we sensitize her to carefully develop the quality smells. Second, we allow for previewing a refactoring. This means that the direct impact can be investigated for the model which is to be refactored. As a consequence, the detection of quality smells can be applied on the preview and newly evoked quality smells can be revealed on-demand. The drawback of this practice is that it is a runtime approach and can produce huge overhead because of the additional execution of detection strategies on the preview. It might be better to have a development time approach, which, e.g., takes into account the structure of the refactoring's role model, the roles of interest of the quality smells, and the target metamodels. We are convinced that research into this direction is promising, since role models expose relevant structures of refactorings and quality smells beforehand and should be used for further analysis. In [Are14], Arendt faces this problem by explicitly relating resolving refactorings with deficiencies they can cause in turn. But we argue that his approach is too rigid and can cause huge overhead, because it requires a DSL designer to always investigate every specified model refactoring in case a new quality smell was defined. She must decide if existing refactorings may evoke a newly defined quality smell.

A similar problem arises in the co-refactoring scenario, as discussed in Sect. 8.5. A refactoring

can cause a co-refactoring, which again may initiate another co-refactoring, and so on. Thus, branches of the refactoring stream can result in cycles. The problem is not a cycle itself but how to avoid endless loops. Currently, our approach only supports the preview of a refactoring by which the direct impact can be investigated. Theoretically, this preview can be propagated along the refactoring stream to investigate if loops occur. But this only shifts the problem to the virtual models of the preview. Endless loops cannot be detected this way. Furthermore, it is not trivial to analyse this in a static manner because it heavily depends on the current refactoring stream, which can change dynamically. In practice, we can only detect if a model which is to be co-refactored already was subject of another (co-)refactoring earlier on the refactoring stream. In such a case, the user must decide how to proceed and if the refactoring stream should be stopped.

Impact of Quality Smells As already discussed in Sect. 6.3 and mentioned before, the resolution of quality smells increases certain qualities but can deteriorate others since they can be interrelated [CNYM00; Sai03; Koz11; Are14]. Since our quality smell approach abstracts from measuring certain qualities by using a generic concept of *quality*, the intrinsic satisfaction of particular quality requirements must be investigated. Thus, existing approaches for measuring qualities must be integrated. For the quality *energy efficiency* the work presented in [Wil14] is promising, since it allows for measuring the energy consumption of mobile Android applications before and after a refactoring. Thus, the quality smells presented in Chap. 7 should be subject of this further research. Furthermore, the catalogue should be extended and more quality smells be added. To accomplish this task, at least a strategy for acquiring quality smells has been presented in Sect. 7.2. In this sense, an intensive evaluation of the triggering of quality smells must be conducted in future work in which the impact of resolving refactorings regarding the particular qualities is measured.

Suggestion of Co-Refactoring Specifications In Sect. 8.4.3, we have illustrated two possibilities to support the co-refactoring engineer in determining appropriate coupled refactorings for the specification of co-refactorings. The first alternative is to suggest role mappings, which map the same role model is an initial refactoring. This is only feasible in case the dependent refactorings are really similar, which is not the case in general. The second alternative is to integrate the user and ask her how to proceed in case no appropriate co-refactoring was defined. As a consequence, other approaches should be investigated. As a first research direction, we propose to learn from the field of the composition of semantic web services [PKPS02; SPAS03]. Therein, a semantic description of web services is exposed and a matching engine tries to match suitable semantic capabilities of web services. Nevertheless, such an approach could not offer suggestions regarding the binding expression. It can only recommend matching refactorings based on their role mappings.

Evaluation in Industrial Context In addition to the presented evaluation in Chap. 10, the whole concept and the implementation within Refactory should be evaluated in a broader setting. Therefore, our co-refactoring approach must be evaluated more extensively. Furthermore, an evaluation in an industrial context should be conducted. We recommend to realise such an evaluation in a context, which contains a huge tool chain like in the automotive area. For instance,

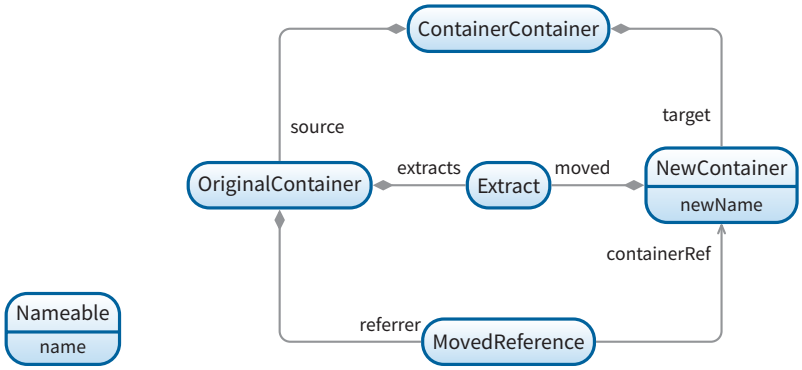
the industrial product *PREEVision*[1] provides a tool chain for the model-based development of distributed systems for the automotive sector [Vector13]. Standards like AUTOSAR[2] [FMB+09] or ReqIF [OMG13a] are incorporated in this product. We are convinced that providing refactoring, quality smell and co-refactoring facilities in an automotive design tool can raise productivity and quality tremendously. Thus, not only the evaluation in an industrial context should be realised in future but also an empirical evaluation regarding cost and utility.

Back-Propagation of Structural Quality Smells As a last issue, we consider worthwhile for future research, the back-propagation of structural quality smells along a transformation chain. Consider, e.g., the quality smells within our catalogue in Chap. 7. They describe quality smells for mobile Android applications. Thus, they regard the implementation phase in MDSD processes. Usually, chains of transformations are used in the MDSD to derive models from other models and finally generate code (which is also a model). As can be observed, the generated code is placed at the end of such a transformation chain. We argue that the root cause, which results in the presence of a quality smell may be situated in earlier phases and quality smells should be detected as early as possible [Sai03]. Thus, we propose to investigate if and how structure-based quality smells can be analysed in order to determine the point in time of a transformation chain, which causes a particular quality smell. Since we have a precisely defined pattern by means of IncPL, the reflected structure could be compared with preceding transformations in the chain. Thus, this investigation is threefold. First, it must be analysed how a transformation chain can be executed symbolically in order to avoid overhead and to reveal information about the structures a transformation produces. Thus, symbolic transformations must be forward-propagated up to the point where the quality smell is situated. Second, the structure of the quality smell and the determined structure of a symbolic transformation must be compared in order to find out if the quality smell was caused in earlier phases of the transformation chain. Third, if the comparison rendered successful the quality smell must be back-propagated to the beginning of the transformation chain to find the distinct phase where the quality smell was originally caused.

[1] `https://vector.com/vi_preevision_en.html` (visited 5th March 2015)
[2] `http://www.autosar.org/` (visited 5th March 2015)

Appendix

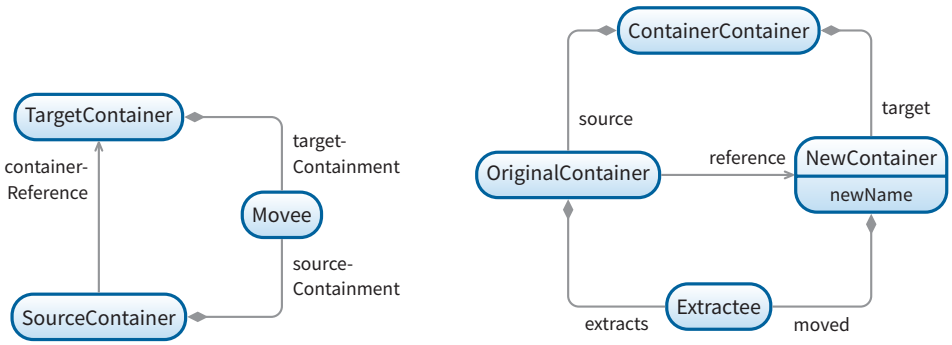
A. List of Role Models



(a) Rename X

(b) Extract X With Reference Class

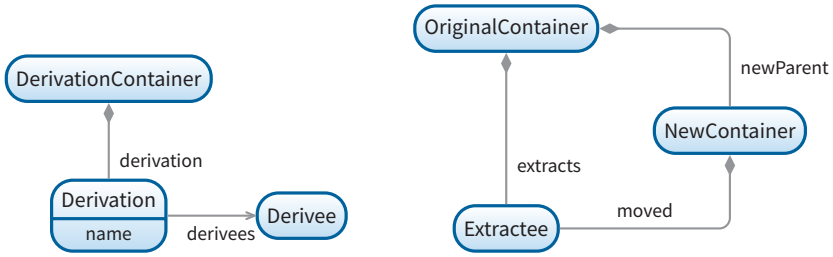
Figure A.1.: Role Models Part 1.



(a) Move X

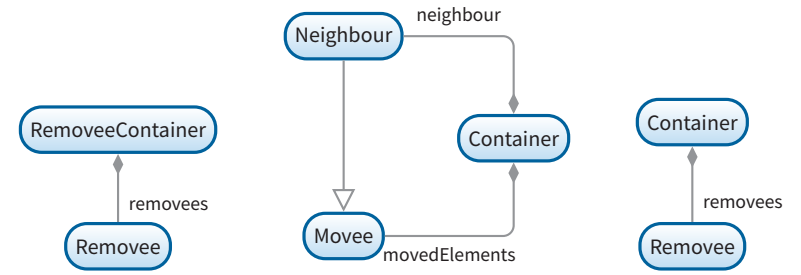
(b) Extract X

Figure A.2.: Role Models Part 2.



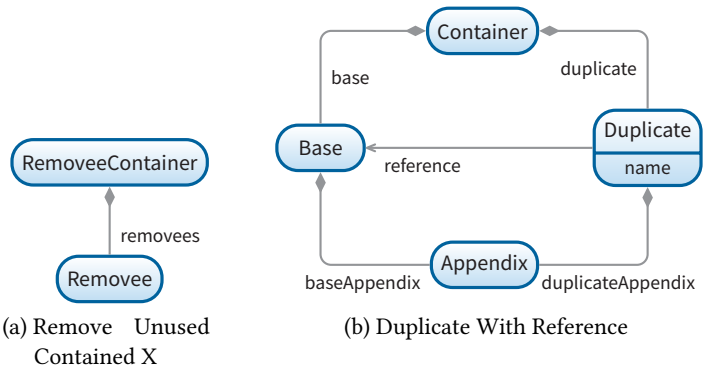
(a) Introduce Reference Class (b) Extract Sub X

Figure A.3.: Role Models Part 3.



(a) Remove Empty Contained X (b) Move X loosely (c) Remove Contained X

Figure A.4.: Role Models Part 4.



(a) Remove Unused Contained X (b) Duplicate With Reference

Figure A.5.: Role Models Part 5.

B. Comparison to Role Feature Model

Based on the role metamodel presented in Sect. 4.2.1, we are able to model the structures required by all refactorings presented in Sect. 10.1. But we want to compare it with the proposal of Kühn *et al.* in [KLG+14], since the authors recognized that there is no clear common understanding of the *role* concept. In this publication, they analyse existing role modelling approaches since the year 2000 and identify commonalities and differences which they derive a feature model from extending the proposed role features of Steimann in [Ste00]. This feature model represents a language family for role models. The discussed features for role models have a pretty much more formal grounding than we have in our role metamodel. With the help of the tools *FeatureIDE*[3] [TKB+14] and *DeltaEcore*[4] [SSA14] it can then be used to generate a new metamodel for role models dependent on the selected desired features.[5]

According to [KLG+14], we would only need the *Role Properties* feature regarding the structure. As valid role *Player*s we only need the mandatory feature *Objects* since we do not allow that roles or other role models can play roles again (because in our approach a role model corresponds to the context). Regarding the *Playable* concepts, we only select *Role Dependent Player Features* and *Different Roles Simultaneously*. We omitted the whole optional *Compartment Types* feature group. A *compartment type* reflects the *context* concept of role models. It is true that we make use of a context but only implicitly. In our approach different refactorings (not a refactoring execution) are considered to be the *refactoring context* for a particular target DSL. In this sense, a role model corresponds directly to the context in our approach and no additional sub-features of *Compartment Types* are needed. That is also the reason why the *Dependent* sub-feature *On Compartments* is deactivated. Furthermore, in our approach only the constraints *Role Implication* and *Role Prohibition* are needed. These *Role Constraints* correspond to our `Collaboration` concept. In addition, our role modelling approach assumes a *Shared Identity* for a role and its playing object. As a last feature we select *Relationship Cardinality* as valid *Relationship Constraints*.

Based on a role feature selection and the specified cross-tree constraints from [KLG+14] a metamodel for roles can be generated. Nevertheless, our proposed role metamodel from the previous section contains two additional metaclasses not having correspondences in the role feature model of Kühn *et al.* at all, namely `RoleComposition` and `RoleAssociation`. According to [Gui05], those are called *parthood* relations. The reason for this design decision is that our metamodel is an abstraction for the participants of refactorings. Thus, it must be capable of reflecting over the structures of participating elements. Since the MOF metalanguage [OMG13c] provides means to model composite and associative relations, these have to be reflected in our metamodel. As a consequence, the role metamodel presented in Sect. 4.2.1 is specific for abstracting over structures in a specific context. In contrast to this, a role metamodel perceived by a feature selection according to [KLG+14] targets the behavioural aspect of role models and explicit contexts, whereas in our approach the context is a generic refactoring itself and thus implicit. Nevertheless, Kühn *et al.* allow for modelling parthood relations by means of annotating them to relations as constraints.

In addition, both Steimann and Kühn *et al.* argue that a role is only valid if it shares a context

[3] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/` (visited 12th February 2015)
[4] `http://deltaecore.org/` (visited 12th February 2015)
[5] `http://st.inf.tu-dresden.de/RML/` (visited 12th February 2015)

with a relation to another role. This means that no single role can occur as the only element in a role model. In contrast to this, our approach allows for this scenario since we abstract over structural properties of refactoring participants. Thus, it is definitely valid if a role model contains only one role which in turn owns a role attribute as the structural property of interest. In the terminology of Steimann and Kühn *et al.* this means that the role shares a context with a relation to its attribute within the specific generic refactoring the role model corresponds to. An example for this is a generic *Rename* refactoring (cf. Fig. A.1 (a)). The role model contains only one role owning one attribute which expresses a property that is to be renamed in the refactoring.

Furthermore, our metamodel was developed in an iterative process and was published in 2010 already [RSA10]. It was subject to further evolution stages [RSA13; ABB+14] but it is stable now with respect to our requirements. Therefore, it would be too complex to replace the existing role metamodel with a completely different one which most likely affects existing clients and their generic refactorings. Besides, a generated role metamodel in terms of a feature selection according to Kühn *et al.* would not exactly meet our requirements and intention. Summarising one can say that for future role metamodel developers we definitely recommend the approach and the tool presented in [KLG+14].

C. Complete List of Role Mappings

The following Table C.1 shows the complete list of all role mappings we considered useful. In the first column, the metamodel is denoted whereas the second column contains the name of the concrete refactoring. The third column contains a check symbol in case a concrete refactoring required a post-processor. Furthermore, the table contains rows covering all three columns and filled grey. These rows denote a generic refactoring and the subsequent refactorings are instances of it.

Table C.1.: Complete list of refactorings.

Metamodel	**Specific Name**	**PP**
Extract X with Reference Class		
AppFlow	Encapsulate In Panel	
Conference	Extract Track	
Forms	Introduce Item Group	
Java	Extract Method	✓
OWL	Extract to new super class	
	Extract to new defined class	
PL/0	Extract Procedure	
SimpleGUI	Encapsulate In Panel	
TextAdventure	Move To New Room	
UML	Extract CompositeState	✓
ConcreteSyntax	Extract Rule	✓
Rename X		
AppFlow	Rename Element	
Company	Rename Company	
	Rename Department	
	Rename Employee	
Conference	Rename Conference Element	
Feature Models	Rename Feature	
Forms	Rename Option	
Java	Rename Element	
Office	Rename Employee	
	Rename Office	
OWL	Rename Element	
	Rename Ontology	
PL/0	Rename Procedure	
	Rename Declaration	
	Rename Program	
Roles	Rename Role	
Sandwich	Rename Ingredient	
TextAdventure	Rename Room	
	Rename Door	
TimedAutomata	Rename EElement	
Ecore	Rename EElement	
BPMN 2	Rename Flow Element	
UML	Rename Element	
ConcreteSyntax	Rename Token	
	Rename Token Redefinition	
	Rename Partial Token	
Introduce Reference Class		
Ecore	Derive Composite Interface	

Feature Models	Introduce Constraint	
UML	Create Sub Interface	
	Create Subclass	
Remove Empty Contained X		
Java	Remove Empty Methods	
OWL	Remove Empty Classes	
UML	Remove Empty Superclass	
Remove Unused Contained X		
Java	Remove Unused Parameters	
UML	Remove Unused Classes	
Duplicate With Reference		
OWL	Duplicate Class	
Ecore	Duplicate Class	
Remove Contained X		
OWL	Remove Disjoint Axioms	
	Remove Inverse Property	
Ecore	Remove Inverse Reference	
Move X		
Java	Move Method	
OWL	Pull up Axiom	
PL/0	Pull Up Constant	
Ecore	Pull Up Feature	
UML	Pull Up Property	
	Pull Up Operation	
	Pull Up Operation To Interface	
	Pull Up Operation To Super Interface	
Extract X		
Ecore	Extract EEnum	
	Introduce Parameter Object	
	Extract Super Class	
	Extract Interface From Features	
	Extract Interface From Operations	
UML	Extract Super Class	
	Extract Interface	
Move X loosely		
Forms	Move Group Next To	
Office	Move Employee Next To	
OWL	Move Element loosely	
Extract Sub X		
Company	Extract Sub-Department	✓
Ecore	Extract Sub EPackage	
BPMN 2	Extract Sub Process	✓
UML	Extract Sub Package	
Select X		
Java	Move GPS resource request to visible state method	✓
	Add Data Compression to Apache HTTP Client based file transmission	✓
	Transform acquire statement to use timeout	✓
	Let activity class override onLowMemory() method	✓
	Introduce Notification	✓
	Replace exact with inexact Alarm Manager	✓
	Use unique generated id for tracking hardware id	✓
OWL	Add Covering Axiom	✓
Introduce Referrer To All X		

OWL	Make All Individuals Distinct	
Inline X		
Ecore	Inline Class	
Replace Feature		
OWL	Convert Data Property To Object Property	
Simple Move X		
OWL	Move Element simple	
Create Referenced Elements		
OWL	Create Enumerated Class	
Introduce Inverse Reference In Container		
OWL	Introduce Inverse Property	
Introduce Simple Reference Class		
AppFlow	Create Initial State	✓
Slow For Loop		
Java	Replace slow for loop with extended for loop	✓
Re-reference X		
OWL	Pull Up Property	
Introduce Inverse Reference		
Ecore	Introduce Inverse Reference	
Introduce Class And Reference		
OWL	Create Individuals to Class	
Extract X Loosely		
OWL	Extract Superclass	
Convert X		
Ecore	Convert EClass to EData Type	
Replace Feature In Container		
Ecore	Replace Data Value with Object	
Extract X from Children		
UML	Introduce Parameter Object	✓

D. List of all IncPL Patterns for Detecting Quality Smells

```
pattern dataTransmissionWithoutCompressionOptimised(FileBodyConstructor:
    NewConstructorCall) {
   find fileBodyConstructorWithFileConstructorParameter(fileBodyVar,
       FileBodyConstructor, _);
   find transmissionCoreMethodCalls(fileBodyVar, method);
} or {
   find fileBodyConstructorWithFileParameter(fileBodyVar, _, FileBodyConstructor)
       ;
   find transmissionCoreMethodCalls(fileBodyVar, method);
}

// the first possibility for instantiating a FileBody:
// FileBody bin = new FileBody(new File(args[0]));
private pattern fileBodyConstructorWithFileConstructorParameter(
   fileBodyVar: LocalVariable,
   fileBodyConstructor: NewConstructorCall,
   fileArgumentConstructor: NewConstructorCall
){
   LocalVariable.initialValue(fileBodyVar, fileBodyConstructor);

   // FileBody constructor
   NewConstructorCall.typeReference(fileBodyConstructor, fileBodyConstructorType)
       ;
   NamespaceClassifierReference.classifierReferences(fileBodyConstructorType,
       fileBodyConstructorTypeReference);
   ClassifierReference.target(fileBodyConstructorTypeReference,
       fileBodyConstructorTypeReferenceTarget);
   Class.name(fileBodyConstructorTypeReferenceTarget, "FileBody");

   // File constructor
   NewConstructorCall.arguments(fileBodyConstructor, fileArgumentConstructor);
   NewConstructorCall.typeReference(fileArgumentConstructor,
       fileArgumentConstructorType);
   NamespaceClassifierReference.classifierReferences(fileArgumentConstructorType,
        fileArgumentConstructorTypeRef);
   ClassifierReference.target(fileArgumentConstructorTypeRef,
       fileArgumentConstructorTypeRefTarget);
   Class.name(fileArgumentConstructorTypeRefTarget, "File");
}

// the second possibility for instantiating a FileBody:
// File file = new File(args[0]);
// FileBody bin = new FileBody(file);
private pattern fileBodyConstructorWithFileParameter(
   fileBodyVar: LocalVariable,
   fileVar: LocalVariable,
   fileBodyConstructor: NewConstructorCall
){
```

```
40      // File variable
41      LocalVariableStatement.variable(_, fileVar);
42      LocalVariable.initialValue(fileVar, fileConstructor);
43
44      // File constructor
45      NewConstructorCall.typeReference(fileConstructor, fileConstructorType);
46      NamespaceClassifierReference.classifierReferences(fileConstructorType,
            fileConstructorTypeReference);
47      ClassifierReference.target(fileConstructorTypeReference,
            fileConstructorTypeReferenceTarget);
48      Class.name(fileConstructorTypeReferenceTarget, "File");
49
50      // FileBody variable
51      LocalVariableStatement.variable(_, fileBodyVar);
52      LocalVariable.initialValue(fileBodyVar, fileBodyConstructor);
53
54      // FileBody constructor
55      NewConstructorCall.typeReference(fileBodyConstructor, fileBodyConstructorType)
            ;
56      NamespaceClassifierReference.classifierReferences(fileBodyConstructorType,
            fileBodyConstructorTypeReference);
57      ClassifierReference.target(fileBodyConstructorTypeReference,
            fileBodyConstructorTypeReferenceTarget);
58      Class.name(fileBodyConstructorTypeReferenceTarget, "FileBody");
59
60      // File variable parameter
61      NewConstructorCall.arguments(fileBodyConstructor, fileVarRef);
62      IdentifierReference.target(fileVarRef,fileVar);
63  }
64
65  private pattern transmissionCoreMethodCalls(fileBodyVar : LocalVariable,
        classmethod: ClassMethod){
66      IdentifierReference.target(entityVarRef, entityVar);
67      IdentifierReference.next(entityVarRef, addPartCaller);
68      MethodCall.target(addPartCaller, addPartCallee);
69      Method.name(addPartCallee, "addPart");
70      MethodCall.arguments(addPartCaller, fileBodyRef);
71      IdentifierReference.target(fileBodyRef, fileBodyVar);
72
73      IdentifierReference.target(httppostRef, httppostVar);
74      IdentifierReference.next(httppostRef, setEntityCaller);
75      MethodCall.target(setEntityCaller, setEntityCallee);
76      Method.name(setEntityCallee, "setEntity");
77      MethodCall.arguments(setEntityCaller, setEntityEntityVarArg);
78      IdentifierReference.target(setEntityEntityVarArg, entityVar);
79
80      MethodCall.target(executeCaller, executeCallee);
81      Method.name(executeCallee, "execute");
82      MethodCall.arguments(executeCaller, executeHttpPostVarArg);
83      IdentifierReference.target(executeHttpPostVarArg, httppostVar);
```

```
84
85    // give me the method in which everything is contained
86    find parentContainsSomething+(classmethod, fileBodyVar);
87 }
88
89 private pattern parentContainsSomething(parent, child){
90    LocalVariableStatement.variable(parent, child);
91 } or {
92    StatementListContainer.statements(parent, child);
93 }
```

Listing D.1: Data Transmission Without Compression.

```
1  pattern droppedData(DataDroppingClass:Class) {
2     find isActivityOrFragment(DataDroppingClass);
3     find classOfField(DataDroppingClass,"EditText");
4     neg find hasMethod(DataDroppingClass, "onRestoreInstanceState");
5     neg find hasMethod(DataDroppingClass, "onSaveInstanceState");
6  }
7
8  private pattern hasMethod(class, method) {
9     Class.members.name(class, method);
10 }
11
12 private pattern classOfField(class,field) {
13    Class.members.name(class, anyFieldName);
14    Variable.name(actualField,anyFieldName);
15    Variable.typeReference(actualField,fieldTypeRef);
16    NamespaceClassifierReference.classifierReferences.target.name(fieldTypeRef,
          field);
17 }
18 private pattern isActivityOrFragment(actualClass:Class) {
19    Class.^extends(actualClass, superClassRef);
20    NamespaceClassifierReference.classifierReferences(superClassRef,
          classifierReference);
21    ClassifierReference.target(classifierReference, superClass);
22    Class.name(superClass, "Activity");
23 } or {
24    Class.^extends(actualClass, superClassRef);
25    NamespaceClassifierReference.classifierReferences(superClassRef,
          classifierReference);
26    ClassifierReference.target(classifierReference, superClass);
27    Class.name(superClass, "Fragment");
28 }
```

Listing D.2: Dropped Data.

```
1  pattern durableWakeLock(TimeoutLessAcquire:MethodCall) {
2     Class.^extends(actualClass, superClassRef);
3     NamespaceClassifierReference.classifierReferences(superClassRef,
          classifierReference);
```

```
    ClassifierReference.target(classifierReference, superClass);
    Class.name(superClass, "Activity");
    Class.members(actualClass, anyMethod);

    ExpressionStatement.expression(bindingStatement, wakeLockIdentifier);
    IdentifierReference.target(wakeLockIdentifier, wakeLockIdentifierReference);
    IdentifierReference.next(wakeLockIdentifier, TimeoutLessAcquire);
    MethodCall.target.name(TimeoutLessAcquire, "acquire");
    neg find hasArguments(TimeoutLessAcquire);

    LocalVariable.typeReference(wakeLockIdentifierReference,
        wakeLockIdentifierReferenceClassfier);
    NamespaceClassifierReference.classifierReferences.target.name(
        wakeLockIdentifierReferenceClassfier, "WakeLock");

    find parentContainsBindingExpression+(anyMethod, bindingStatement);
}

private pattern parentContainsBindingExpression(parent, child) {
    StatementListContainer.statements(parent, child);
} or {
    ExpressionStatement.expression(parent, child);
} or {
    StatementContainer.statement(parent, child);
}

private pattern hasArguments(args) {
    MethodCall.arguments(args, _);
}
```

Listing D.3: Durable WakeLock.

```
pattern internalGetter(CallOfGetter:MethodCall) {
    find isGetter(getter);
    MethodCall.target(CallOfGetter,getter);
}

private pattern isGetter(actualMethod:ClassMethod) {
    // return value of method is a field
    ClassMethod.statements(actualMethod,statemets);
    Return.returnValue(statemets,ref);
    IdentifierReference.target(ref,variable);
    Variable.name(variable,varName);
    Class.members.name(actualClass,varName);

    // method is contained in the same class as the field
    Class.members(actualClass,actualMethod);
}
```

Listing D.4: Internal Use of Getters/Setters.

```
pattern noMemoryResolver(ClassWithoutMemoryResolver:Class) {
   Class.^extends(ClassWithoutMemoryResolver, superClassRef);
   NamespaceClassifierReference.classifierReferences(superClassRef,
       classifierReference);
   ClassifierReference.target(classifierReference, superClass);
   Class.name(superClass, "Activity");

   neg find hasMethod_mom(ClassWithoutMemoryResolver, "onLowMemoryResolver");
}

private pattern hasMethod_mom(class, method) {
   Class.members.name(class, method);
}
```

Listing D.5: No Low Memory Resolver.

```
pattern rigidAlarmManager(RigidCaller:MethodCall) {
   Class.^extends(actualClass, superClassRef);
   NamespaceClassifierReference.classifierReferences(superClassRef,
       classifierReference);
   ClassifierReference.target(classifierReference, superClass);
   Class.name(superClass, "Activity");
   Class.members(actualClass, anyMethod);

   ExpressionStatement.expression(bindingStatement, alarmIdentifier);
   IdentifierReference.target(alarmIdentifier, alarmIdentifierReference);
   IdentifierReference.next(alarmIdentifier, RigidCaller);
   MethodCall.target.name(RigidCaller, "setRepeating");

   LocalVariable.typeReference(alarmIdentifierReference, alarmReferenceClassfier)
       ;
   NamespaceClassifierReference.classifierReferences.target.name(
       alarmReferenceClassfier, "AlarmManager");

   find parentContainsBindingExpression_rig+(anyMethod, bindingStatement);
}

private pattern parentContainsBindingExpression_rig(parent, child) = {
   StatementListContainer.statements(parent, child);
} or {
   ExpressionStatement.expression(parent, child);
} or {
   StatementContainer.statement(parent, child);
}
```

Listing D.6: Rigid `AlarmManager`.

```
pattern UnclosedCloseable(UnclosedHolder:ClassMethod, UnclosedParameter:
    OrdinaryParameter) {
   neg find findClassParam(s);
   OrdinaryParameter.name(UnclosedParameter,s);
```

```
    find findClass(UnclosedHolder,UnclosedParameter);
    find findCloseableParam(UnclosedParameter);
}

pattern findClassParam(s) {
    find findClose(s);
    ClassMethod.name(c,r);
    ClassMethod.parameters(c,t);
    ClassMethod.statements(c,h);
}

pattern findClass(c,t) {
    ClassMethod.name(c,r);
    ClassMethod.parameters(c,t);
    ClassMethod.statements(c,h);
}


pattern findClose(t) {
    ExpressionStatement.expression(a,exp);
    IdentifierReference.target.name(exp, t);
    IdentifierReference.next(exp, exp2);
    MethodCall.target.name(exp2, "close");
}

pattern findCloseable(dev:NamespaceClassifierReference) {
    NamespaceClassifierReference.classifierReferences(dev, a);
    ClassifierReference.target.name(a, "Closeable");
}

pattern findCloseableParam(t){
    OrdinaryParameter.typeReference(t,s);
    NamespaceClassifierReference.classifierReferences(s,r);
    ClassifierReference.target(r,a);
    Interface.name(a,"Closeable");
    find implementsOrExtends(a,b);
}
```

Listing D.7: Unclosed `Closeable`.

```
pattern UnTouchable(SmallerConstructorCall:NewConstructorCall){
    Class.^extends(actualClass, superClassRef);
    NamespaceClassifierReference.classifierReferences(superClassRef,
        classifierReference);
    ClassifierReference.target(classifierReference, superClass);
    find isActivity(superClass);

    find decimalLessThan(SmallerConstructorCall, argumentExpression);
}

pattern decimalLessThan(constructor, decimalParameter){
```

```
11     NewConstructorCall.typeReference(constructor,g);
12     NamespaceClassifierReference.classifierReferences(g, c);
13     ClassifierReference.target.name(c, "LayoutParams");
14     NewConstructorCall.arguments(constructor,x);
15     DecimalIntegerLiteral.decimalValue(x,decimalParameter);
16
17     check(new BigInteger("48") >= decimalParameter);
18 }
19
20 private pattern isActivity(class) {
21     find isClassOf(class, "Activity");
22 }
23
24 private pattern isClassOf(class, className) {
25     Class.name(class, className);
26 }
```

Listing D.8: Untouchable.

E. Post-Processor of the *Extract CompositeState* refactoring for UML State Machines

Listing E.9 shows the post-processor for the refactoring *Extract CompositeState*. The method `process(..)` in Line 6 belongs to the API of Refactory. It can be seen, that the `roleBindings` to the concrete model elements is passed as first parameter. The second parameter `resourceSet` is specific to the EMF and provides the set of all directly related models of the refactored model. The third parameter change is specific to EMF Compare and contains a description of the modifications made so far, recorded while executing the generic part of the refactoring.

In this post-processor, the bound elements are determined in Lines 7 and 8. In Line 9, the method `moveRelevantTransitionsToCompositeState()` is invoked implementing the analysis, which transitions to inner states of the newly created composite state now have to end at the composite state instead of the inner state. Analogously, the outgoing transitions must be computed. To produce a correct composite state with respect to the UML metamodel, an `Activity` must be added to the composite state, which is realised with the method call in Line 10.

```
1  public class UMLExtractCompositeStatePostProcessor extends
       AbstractRefactoringPostProcessor {
2
3     private List<State> movedStates;
4     private State newCompositeState;
5
6     public IStatus process(Map<Role, List<EObject>> roleBindings, ResourceSet
          resourceSet, ChangeDescription change) {
7        movedStates = RoleUtil.getObjectsForRole("Extract", State.class,
             roleBindings);
8        newCompositeState = RoleUtil.getFirstObjectForRole("NewContainer", State.
             class, roleBindings);
9        moveRelevantTransitionsToCompositeState();
10       createActivity();
11       return Status.OK_STATUS;
12    }
13
14    private Boolean moveRelevantTransitionsToCompositeState(){
15       Set<Transition> removees = new HashSet<Transition>();
16       Set<Transition> inComposites = new HashSet<Transition>();
17       Set<Transition> outComposites = new HashSet<Transition>();
18       for (State movedState : movedStates) {
19          List<State> otherStates = new ArrayList<State>(movedStates);
20          otherStates.remove(movedState);
21          List<Transition> incomings = movedState.getIncomings();
22          handleTransitions(removees, inComposites, otherStates, incomings, true);
23          List<Transition> outgoings = movedState.getOutgoings();
24          handleTransitions(removees, outComposites, otherStates, outgoings,
                false);
25       }
26       handleInternalTransitions(removees);
27       for (Transition transition : inComposites) {
```

```
28          transition.setTarget(newCompositeState);
29       }
30       for (Transition transition : outComposites) {
31          transition.setSource(newCompositeState);
32       }
33       return true;
34    }
35
36    private Boolean createActivity(){
37       Activity activity = UMLFactory.eINSTANCE.createActivity();
38       activity.setName(newCompositeState.getName() + "Activity");
39       newCompositeState.setDoActivity(activity);
40       return true;
41    }
42
43    private Boolean handleInternalTransitions(Set<Transition> removees) {
44       List<Region> regions = newCompositeState.getRegions();
45       for (Region region : regions) {
46          if(region.getSubvertices().containsAll(movedStates)){
47             for (Transition transition : removees) {
48                EcoreUtil.remove(transition);
49                if(!region.getTransitions().add(transition)){
50                   return false;
51                }
52             }
53          }
54       }
55       return true;
56    }
57
58    private void handleTransitions(Set<Transition> removees, Set<Transition>
          outsides, List<State> others, List<Transition> transitions, boolean source
          ) {
59       for (Transition transition : transitions) {
60          Vertex vertex = null;
61          if(source){
62             vertex = transition.getSource();
63          } else {
64             vertex = transition.getTarget();
65          }
66          if(others.contains(vertex)){
67             removees.add(transition);
68          } else {
69             outsides.add(transition);
70          }
71       }
72    }
73 }
```

Listing E.9: UML-specific post-processor for determining incoming and outgoing transitions of the extracted composite state.

F. Specification of the Conference Language

Abstract Syntax

```
package conference conference "http://www.emftext.org/language/conference" {

   abstract class NamedElement {
      attribute EString name (1..1);
   }

   abstract class ConferenceElement { }

   class Conference extends NamedElement {
         containment reference ConferenceElement elements (0..-1);
         reference Participant organizers (1..-1);
         containment reference Participant speakers (1..-1);
   }
   class Track extends ConferenceElement, NamedElement {
         containment reference Slot slots (0..-1);
   }
   class TimedElement {
         attribute EInt hour (1..1);
         attribute EInt minute (1..1);
   }
   class Slot extends TimedElement {
         containment reference Talk talk (1..1);
   }
   class Talk extends NamedElement {
         reference Participant presenter (1..1);
   }
   class Participant extends NamedElement {
         attribute EString country (1..1);
   }
   class Lunch extends ConferenceElement, TimedElement { }
}
```

Concrete Textual Syntax

```
SYNTAXDEF conference
FOR <http://www.emftext.org/language/conference>

START Conference

RULES {
   Conference ::=
      "CONFERENCE" #1 name['"','"'] #1
      "(" organizers['"','"'] ("," #1 organizers['"','"'])* ")"
      !0 ( !0 elements )*
      !0 "REGISTERED" "SPEAKERS" ":" !0 speakers ("," !0 speakers)*;

   Participant ::= name['"','"'] #1 "FROM" #1 country [];
```

```
14
15    Talk ::= "TALK" #1 name['"','"'] #1 "PRESENTED" "BY" presenter['"','"'] !0;
16
17    Track ::= "TRACK" #1 name['"','"'] ":" !0 (slots)*;
18
19    Slot ::= "AT" #1 hour[] #0 ":" #0 minute[] ":" #1 talk;
20
21    Lunch ::= "AT" hour[] #0 ":" #0 minute[] #1 "LUNCH" !0;
22 }
```

List of Abbreviations

AOP	aspect-oriented programming 29, 31
API	Application Programming Interface 116, 121, 124, 126, 131, 135, 144, 146, 147, 183
AST	abstract syntax tree 27
ATL	Atlas Transformation Language [JK06] 45, 46
BNF	Backus-Naur Form [BBG+63] 28
BPMN	Business Process Model and Notation [OMG13b] 3, 4, 143, 145, 148–151, 173, 174
CBR	constraint-based refactoring 28, 30, 31, 42, 43, 49
CC	conservative copy 42
CLS	cross-language support 8
Co-RefSpec	Co-Refactoring Specification 114, 115, 117–119, 138, 141, 156, 157, 160
CoRK	Co-Refactoring Knowledge 113
CoRK-Base	Co-Refactoring Knowledge Base 106, 114, 115, 118, 119, 138, 141
CSP	constraint satisfaction problem 28, 71
CSS	Cascading Style Sheets [W3C11] 3, 4
DK-Base	Dependency Knowledge Base 106, 110, 111, 113, 118, 119, 141
DL	Description Logics [BMNP03] 29
DSL	Domain-Specific Language v, 2–13, 15–18, 25–30, 32–35, 38, 40, 45, 48, 51–55, 57, 58, 60, 61, 66, 70–74, 76, 77, 80, 81, 88, 105, 113, 114, 117–119, 121, 122, 132, 135, 143, 144, 146, 148–151, 153–156, 162, 165, 166, 171
DSLE	Domain-Specific Language Environment 6–8, 121, 130
EBNF	Extended Backus-Naur Form [ISO96] 122
ECA	event-condition-action 40, 115, 165
EMF	Eclipse Modeling Framework [SBPM08] 5, 35–38, 44, 46, 49, 57, 92, 121, 122, 124, 126, 128, 130–132, 134, 141, 143, 147, 156, 163, 183
EMOF	Essential MOF 18, 121, 122

EWL	Epsilon Wizard Language [KPPR07] 28, 31, 48
GME	Generic Modeling Environment [LMB+01] 27, 31, 32
GMF	Graphical Modeling Framework [Gro09] 18, 46, 121, 132
GoF	Gang of Four [GHJV94] 20
GPL	General Purpose Language 60, 70, 71, 116, 121
HOT	higher-order transformation 27, 41, 47
IDE	integrated development environment v, 5, 6, 8, 45, 49, 92, 121, 130, 131, 147
IncPL	IncQuery Pattern Language 92, 93, 95–104, 165, 168, 176
IT	information technology 3, 156
JaMoPP	Java Model Parser and Printer [HJSW10] 9, 83, 93, 95
LTK	Eclipse Language Toolkit 130, 131
LW	Language Workbench 5, 6, 70
M2C	model-to-code 42
MDA	Model-Driven Architecture [OMG03] 18, 105
MDSD	Model-Driven Software Development [SVB+06] 4, 7, 10, 12, 15, 16, 18, 25, 40, 42, 48, 51, 168
MLDE	Multi-Language Development Environment 8, 10, 11, 45, 47–49, 51, 66, 82, 105, 107, 108, 110, 111, 113, 118, 122, 140
MOF	Meta Object Facility [OMG13c] 17, 25–29, 31, 32, 34–36, 40, 42, 57, 60, 71, 74, 105, 121, 144, 161, 165, 171
OCL	Object Constraint Language [OMG14a] 28, 35, 67, 71, 122, 130, 131, 141
OMG	Object Management Group 17
OSGi	Open Service Gateway initiative [OSGi14] 121
OWL	Web Ontology Language [W3C12] 13, 117, 143, 145, 148, 150–157, 164, 173–175
PAML	Performance Antipattern Modeling Language 36–38
QSM	Quality Smell Model 81, 82, 85
RefSpec	Refactoring Specification 61–66, 81, 114, 124–126, 129, 140, 146
RTE	Round-Trip Engineering 45, 47
SoC	separation of concerns 16
SUM	single underlying model 42, 48

TGG	triple graph grammar 44, 47
TTCN-3	testing and test control notation 34, 35
UI	user interface 82, 93, 98, 103, 122, 126, 134, 140
UML	Unified Modeling Language [OMG11a] 3, 4, 16–18, 27, 29, 35, 36, 39, 42, 43, 54, 60, 65, 73, 76, 107–109, 129, 143, 145, 146, 148–152, 154, 155, 173–175, 183
VM	Virtual Machine 45, 46, 113
WFR	well-formedness rule 28, 42, 48, 71, 72, 129, 130, 164
XMI	XML Metadata Interchange [OMG14b] 42

Bibliography

[ABB+14] U. Aßmann, A. Bartho, C. Bürger, S. Cech, B. Demuth, F. Heidenreich, J. Johannes, S. Karol, J. Polowinski, J. Reimann, J. Schroeter, M. Seifert, M. Thiele, C. Wende and C. Wilke, "DropsBox: the Dresden Open Software Toolbox", English, *Software & Systems Modeling*, vol. 13, no. 1, pp. 133–169, 2014, ISSN: 1619-1366. DOI: 10.1007/s10270-012-0284-6 (cit. on p. 172).

[ABJ+10] T. Arendt, E. Biermann, S. Jurack, C. Krause and G. Taentzer, "Henshin. Advanced Concepts and Tools for In-Place EMF Model Transformation", in *Model Driven Engineering Languages and Systems - 13th International Conference, MoDELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, D. C. Petriu, N. Rouquette and Ø. Haugen, Eds., ser. Lecture Notes in Computer Science, vol. 6394, Springer, 2010, pp. 121–135, ISBN: 978-3-642-16144-5. DOI: 10.1007/978-3-642-16145-2_9 (cit. on pp. 29, 35).

[ABT10] T. Arendt, M. Burhenne and G. Taentzer, "Defining and Checking Model Smells. A Quality Assurance Task for Models based on the Eclipse Modeling Framework", in *9th edition of the BENEVOL workshop*, 2010 (cit. on pp. 33, 87).

[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 pp. 9, 32, 79).

[Amo12] J. Amos. (Jun. 2012). Red dot becomes 'oldest cave art'. Accessed: 2014-09-30 (Archived by WebCite® at http://www.webcitation.org/6SyZIW65G), [Online]. Available: http://www.bbc.com/news/science-environment-18449711 (cit. on p. 2).

[AMT10] T. Arendt, F. Mantz and G. Taentzer, "EMF Refactor. Specification and Application of Model Refactorings within the Eclipse Modeling Framework", in *9th edition of the BENEVOL workshop*, 2010 (cit. on pp. 29, 35).

[Are14] T. Arendt, "Quality Assurance of Software Models - A Structured Quality Assurance Process Supported by a Flexible Tool Environment in the Eclipse Modeling Project", Dissertation, Philipps-Universität Marburg, 2014. [Online]. Available: http://archiv.ub.uni-marburg.de/diss/z2014/0357 (cit. on pp. 29, 31, 35, 37, 38, 49, 67, 87, 166, 167).

[ASB10] C. Atkinson, D. Stoll and P. Bostan, "Orthographic Software Modeling: A Practical Approach to View-Based Development", English, in *Evaluation of Novel Approaches to Software Engineering*, ser. Communications in Computer and Information Science, L. Maciaszek, C. González-Pérez and S. Jablonski, Eds., vol. 69, Springer Berlin Heidelberg, 2010, pp. 206–219, ISBN: 978-3-642-14818-7. DOI: 10.1007/978-3-642-14819-4_15 (cit. on p. 42).

[AST10] T. Arendt, P. Stepien and G. Taentzer, "EMF Metrics. Specification and Calculation of Model Metrics within the Eclipse Modeling Framework", in *9th edition of the BENEVOL workshop*, 2010 (cit. on pp. 33, 37, 87).

[AT12] T. Arendt and G. Taentzer, "Integration of Smells and Refactorings within the Eclipse Modeling Framework", in *Proceedings of the Fifth Workshop on Refactoring Tools*, 2012, pp. 8–15 (cit. on p. 35).

[Bac73] C. W. Bachman, "The Programmer As Navigator", *Communications of the ACM*, vol. 16, no. 11, pp. 653–658, Nov. 1973, ISSN: 0001-0782. DOI: 10.1145/355611.362534 (cit. on p. 19).

[BBG+63] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden and M. Woodger, "Revised report on the algorithmic language ALGOL 60", *The Computer Journal*, vol. 5, no. 4, pp. 349–367, 1963. DOI: 10.1093/comjnl/5.4.349 (cit. on pp. 28, 187).

[BBKL78] B. W. Boehm, J. R. Brown, H. Kaspar and M. Lipow, *Characteristics of software quality*, ser. TRW Softw. Technol. Amsterdam: North-Holland, 1978 (cit. on p. 81).

[BC87] K. Beck and W. Cunningham, "Using Pattern Languages for Object-Oriented Programs", in *Proceedings of OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming*, 1987. [Online]. Available: http://c2.com/doc/oopsla87.html (cit. on pp. 12, 92).

[BD02] J. Bansiya and C. G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment", *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002 (cit. on pp. 33, 37, 87).

[BE08] D. Bildhauer and J. Ebert, "Querying Software Abstraction Graphs", in *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC), collocated with ICPC*, 2008 (cit. on p. 92).

[BEK+06a] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer and E. Weiss, "EMF Model Refactoring based on Graph Transformation Concepts", *ECEASST*, vol. 3, 2006 (cit. on p. 29).

[BEK+06b] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer and E. Weiss, "Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework", in, ser. Lecture Notes in Computer Science. Springer, 2006, vol. 4199/2006, pp. 425–439. DOI: 10.1007/11880240_30 (cit. on p. 29).

[BERS08] D. Bildhauer, J. Ebert, V. Riediger and H. Schwarz, "Using the TGraph Approach for Model Fact Repositories", in *Proceedings of the Second International Workshop MoRSe*, 2008, pp. 9–18 (cit. on pp. 34, 37).

[BG10] E. Burger and B. Gruschko, "A Change Metamodel for the Evolution of MOF-Based Metamodels", in *Proceedings of Modellierung 2010*, G. Engels, D. Karagiannis and H. C. Mayr, Eds., ser. GI-LNI, vol. P-161, Klagenfurt, Austria, –2624th Mar. 2010, pp. 285–300 (cit. on p. 42).

[BGM+11] B. R. Bryant, J. Gray, M. Mernik, P. J. Clarke, R. B. France and G. Karsai, "Challenges and Directions in Formalizing the Semantics of Modeling Languages", *Computer Science and Information Systems*, vol. 8, no. 2, pp. 225–253, 2011. DOI: 10.2298/CSIS110114012B (cit. on p. 71).

[BHH+12] G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi and D. Varró, "Integrating Efficient Model Queries in State-of-the-Art EMF Tools", in *Objects, Models, Components, Patterns*, ser. Lecture Notes in Computer Science, C. Furia and S. Nanz, Eds., vol. 7304, Springer Berlin Heidelberg, 2012, pp. 1–8, ISBN: 978-3-642-30560-3. DOI: 10.1007/978-3-642-30561-0_1 (cit. on pp. 37, 92).

[BJV04] J. Bézivin, F. Jouault and P. Valduriez, "On the need for megamodels", in *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004 (cit. on p. 48).

[BKR07] S. Becker, H. Koziolek and R. Reussner, "Model-Based Performance Prediction with the Palladio Component Model", in *Proceedings of the 6th International Workshop on Software and Performance*, ser. WOSP '07, Buenes Aires, Argentina: ACM, 2007, pp. 54–65, ISBN: 1-59593-297-6. DOI: 10.1145/1216993.1217006. [Online]. Available: http://doi.acm.org/10.1145/1216993.1217006 (cit. on p. 36).

[BLS+09] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger and W. Schwinger, "An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example", in *Model Driven Engineering Languages and Systems – 12th International Conference, MoDELS 2009*, A. Schürr and B. Selic, Eds., ser. Lecture Notes in Computer Science, vol. 5795, Denver, CO: Springer, Oct. 2009, pp. 271–285, ISBN: 978-3-642-04424-3. DOI: 10.1007/978-3-642-04425-0_20 (cit. on p. 30).

[BMMM08] X. Blanc, I. Mounier, A. Mougenot and T. Mens, "Detecting Model Inconsistency through Operation-Based Model Construction", in *ACM/IEEE 30th International Conference on Software Engineering (ICSE '08)*, May 2008, pp. 511–520. DOI: 10.1145/1368088.1368158 (cit. on p. 45).

[BMMM98] W. J. Brown, R. C. Malveau, S. McCormick and T. J. Mowbray, *Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998 (cit. on p. 89).

[BMNP03] F. Baader, D. McGuinness, D. Nardi and P. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003 (cit. on pp. 29, 187).

[Bry14] M. Brylski, "Durchführung einer Entwicklerstudie zum Ermitteln von Quality Smells und deren Beseitigung auf Android-Systemen", Minor Thesis (Großer Beleg), Technische Universität Dresden, 2014 (cit. on pp. 89, 96).

[BS06] J. Baumeister and D. Seipel, "Verification and Refactoring of Ontologies with Rules", English, in *Managing Knowledge in a World of Networks*, ser. Lecture Notes in Computer Science, S. Staab and V. Svátek, Eds., vol. 4248, Springer Berlin Heidelberg, 2006, pp. 82–95, ISBN: 978-3-540-46363-4. DOI: 10.1007/11891451_11 (cit. on p. 108).

[BSW+09] P. Brosch, M. Seidl, K. Wieland, M. Wimmer and P. Langer, "The Operation Recorder: Specifying Model Refactorings By-Example", in *OOPSLA Companion*, S. Arora and G. T. Leavens, Eds., ACM, 2009, pp. 791–792, ISBN: 978-1-60558-768-4 (cit. on pp. 30, 31).

[Bur14] E. Burger, "Flexible Views for View-based Model-driven Development", PhD thesis, Institut für Programmstrukturen und Datenorganisation (IPD), Karlsruher Institut für Technologie, 2014. DOI: 10.5445/KSP/1000043437 (cit. on pp. 42, 47, 48).

[BV08] M. Breugelmans and B. Van Rompaey, "TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites", in *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*, 2008 (cit. on pp. 35, 37, 38).

[Cam14] F. Campagne, *The MPS Language Workbench*. Mar. 2014, vol. I (cit. on p. 5).

[CDEP08] A. Cicchetti, D. Di Ruscio, R. Eramo and A. Pierantonio, "Automating Co-evolution in Model-Driven Engineering", in *12th International IEEE Enterprise Distributed Object Computing Conference (EDOC '08)*, Sep. 2008, pp. 222–231. DOI: 10.1109/EDOC.2008.44 (cit. on p. 46).

[CH06] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches", *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006, ISSN: 0018-8670. DOI: 10.1147/sj.453.0621 (cit. on p. 67).

[Chu13] K.-m. Chung, *Expression Language Specification – Version 3.0 Final Release*, Apr. 2013 (cit. on p. 116).

[CK94] S. Chidamber and C. Kemerer, "A Metrics Suite for Object Oriented Design", *Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994 (cit. on pp. 33, 87).

[CLMM07] Y. Crespo, C. López, M. E. M. Martínez and R. Marticorena, "From Bad Smells to Refactoring: Metrics Smoothing the Way", in *Object-Oriented Design Knowledge: Principles, Heuristics and Best Practices*. Idea Group Publishing, 2007, ch. VII, pp. 193–249 (cit. on pp. 27, 37).

[CNYM00] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers Boston/Dordrecht/London, 2000 (cit. on pp. 87, 166, 167).

[Cou90] B. Courcelle, "Graph Rewriting: An Algebraic and Logic Approach", in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., Cambridge, MA, USA: MIT Press, 1990, ch. 5, pp. 193–242. [Online]. Available: http://dl.acm.org/citation.cfm?id=114891.114896 (cit. on p. 111).

[Cun13] W. Cunningham, *Anti Patterns Catalog*, visited 4th March 2015, 2013. [Online]. Available: http://c2.com/cgi/wiki?AntiPatternsCatalog (cit. on p. 89).

[DB91] A. M. Davis and E. H. Bersoff, "Impacts of Life Cycle Models on Software Configuration Management", *Communications of the ACM*, vol. 34, no. 8, pp. 104–118, Aug. 1991, ISSN: 0001-0782. DOI: 10.1145/108515.108537 (cit. on p. 5).

[DGG95] K. R. Dittrich, S. Gatziu and A. Geppert, "The Active Database Management System Manifesto: A Rulebase of ADBMS Features", English, in *Rules in Database Systems*, ser. Lecture Notes in Computer Science, T. Sellis, Ed., vol. 985, Springer Berlin Heidelberg, 1995, pp. 1–17, ISBN: 978-3-540-60365-8. DOI: 10.1007/3-540-60365-4_116 (cit. on p. 115).

[DGLD05] S. Ducasse, T. Gırba, M. Lanza and S. Demeyer, "Moose: a Collaborative and Extensible Reengineering Environment", *Tools for Software Maintenance and Reengineering, RCOST/Software Technology Series*, vol. 71, 2005 (cit. on p. 27).

[Dis11] Z. Diskin, "Model Synchronization: Mappings, Tiles, and Categories", English, in *Generative and Transformational Techniques in Software Engineering III*, ser. Lecture Notes in Computer Science, J. M. Fernandes, R. Lämmel, J. Visser and J. Saraiva, Eds.,

vol. 6491, Springer Berlin Heidelberg, 2011, pp. 92–165, ISBN: 978-3-642-18022-4. DOI: 10.1007/978-3-642-18023-1_3 (cit. on p. 44).

[DJK+06] D. Di Ruscio, F. Jouault, I. Kurtev, J. Bézivin and A. Pierantonio, "Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs", Apr. 2006, [Online]. Available: https://hal.archives-ouvertes.fr/hal-00023008 (cit. on p. 71).

[DLP11] D. Di Ruscio, R. Lämmel and A. Pierantonio, "Automated Co-evolution of GMF Editor Models", English, in *Software Language Engineering*, ser. Lecture Notes in Computer Science, B. Malloy, S. Staab and M. van den Brand, Eds., vol. 6563, Springer Berlin Heidelberg, 2011, pp. 143–162, ISBN: 978-3-642-19439-9. DOI: 10.1007/978-3-642-19440-5_9 (cit. on pp. 46–48).

[DMTS12] J. Dietrich, C. McCartin, E. Tempero and S. M. A. Shah, "On the Existence of High-impact Refactoring Opportunities in Programs", in *Proceedings of the 35th Australasian Computer Science Conference*, ser. ACSC '12, vol. 122, Melbourne, Australia: Australian Computer Society, Inc., 2012, pp. 37–48, ISBN: 978-1-921770-03-6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2483654.2483659 (cit. on pp. 33, 36–38, 74, 87, 92).

[DPXT12] J. Dexun, M. Peijun, S. Xiaohong and W. Tiantian, "Detecting Bad Smells with Weight Based Distance Metrics Theory", in *Second International Conference on Instrumentation, Measurement, Computer, Communication and Control (IMCCC), 2012*, 2012, pp. 299–304. DOI: 10.1109/IMCCC.2012.74 (cit. on p. 37).

[ELF08] A. Egyed, E. Letier and A. Finkelstein, "Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models", in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, Sep. 2008, pp. 99–108. DOI: 10.1109/ASE.2008.20 (cit. on p. 114).

[EPRV08] R. Eramo, A. Pierantonio, J. R. Romero and A. Vallecillo, "Change Management in Multi-Viewpoint System Using ASP", in *Enterprise Distributed Object Computing Conference Workshops, 2008 12th*, Sep. 2008, pp. 433–440. DOI: 10.1109/EDOCW.2008.22 (cit. on p. 43).

[ES07] J. Euzenat and P. Shvaiko, *Ontology Matching*. Heidelberg: Springer, 2007, ISBN: 3-540-49611-4 (cit. on p. 78).

[EV06] S. Efftinge and M. Völter, "oAW xText: A framework for textual DSLs", in *Workshop on Modeling Symposium at Eclipse Summit*, 2006 (cit. on p. 5).

[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. 3, 6).

[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. 5, 9, 10, 12, 15, 16, 25, 32, 33, 54, 67, 68, 79, 80, 87, 89, 157, 162).

[FMB+09] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa and K. Lange, "AUTOSAR – A Worldwide Standard is on the Road", in *14th International VDI Congress Electronic Systems for Vehicles*, 2009 (cit. on p. 168).

[Fow05] M. Fowler, "Language Workbenches: The Killer-App for Domain Specific Languages?", 2005, Available: `http://www.martinfowler.com/articles/language-Workbench.html` (cit. on pp. 5, 6).

[GHJV94] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994 (cit. on pp. 12, 16, 20, 89, 92, 188).

[GJJW12] M. Gottschalk, M. Josefiok, J. Jelschen and A. Winter, "Removing Energy Code Smells with Reengineering Services", in *Beitragsband der 42. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, vol. 208, Bonner Köllen Verlag, 2012, pp. 441–455 (cit. on pp. 34, 37).

[GJW14] M. Gottschalk, J. Jelschen and A. Winter, "Saving Energy on Mobile Devices by Refactoring", in *28th International Conference on Informatics for Environmental Protection: ICT for Energy Effieciency, EnviroInfo 2014*, J. Marx Gómez, M. Sonnenschein, U. Vogel, A. Winter, B. Rapp and N. Giesen, Eds., Oldenburg, Germany: BIS-Verlag, Sep. 2014, pp. 437–444, ISBN: 978-3-8142-2317-9. [Online]. Available: `http://www.enviroinfo2014.org/` (cit. on pp. 34, 37, 38).

[GKP07] B. Gruschko, D. S. Kolovos and R. F. Paige, "Towards Synchronizing Models with Evolving Metamodels", in *In Proceedings of the International Workshop on Model-Driven Software Evolution held with the ECSMR*, 2007 (cit. on pp. 41, 118).

[GL12] H. Giese and L. Lambers, "Towards Automatic Verification of Behavior Preservation for Model Transformation via Invariant Checking", English, in *Graph Transformations*, ser. Lecture Notes in Computer Science, H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, Eds., vol. 7562, Springer Berlin Heidelberg, 2012, pp. 249–263, ISBN: 978-3-642-33653-9. DOI: `10.1007/978-3-642-33654-6_17` (cit. on p. 71).

[GPEM09] J. Garcia, D. Popescu, G. Edwards and N. Medvidovic, "Identifying Architectural Bad Smells", in *13th European Conference on Software Maintenance and Reengineering (CSMR '09)*, 2009, pp. 255–258. DOI: `10.1109/CSMR.2009.59` (cit. on pp. 33, 87).

[Gra92] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992, ISBN: 0-13-720384-5 (cit. on p. 81).

[GRK14] S. Getir, M. Rindt and T. Kehrer, "A Generic Framework for Analyzing Model Co-Evolution", in *Proceedings of the MoDELS 2014 conference workshop Models and Evolution (ME 2014)*, 2014, pp. 12–21 (cit. on pp. 43, 47).

[Gro09] R. C. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Pearson Education, Apr. 2009, ISBN: 0321534077 (cit. on pp. 18, 188).

[GS03] J. Greenfield and K. Short, "Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools", in *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Anaheim, CA, USA: ACM, 2003, pp. 16–27, ISBN: 1-58113-751-6. DOI: `10.1145/949344.949348` (cit. on p. 16).

[GSG+09] J. Gausemeier, W. Schäfer, J. Greenyer, S. Kahl, S. Pook and J. Rieke, "Management of Cross-Domain Model Consistency during the Development of Advanced Mechatronic Systems", in *Proceedings of the 17th International Conference on Engineering Design (ICED'09)*, M. Norell Bergendahl, M. Grimheden, L. Leifer, P. Skogstad and U. Lindemann, Eds., ser. Design Methods and Tools (pt. 2), vol. 6, Palo Alto, CA, USA: Design Society, 2009, pp. 1–12 (cit. on p. 44).

[Gui05] G. Guizzardi, "Ontological foundations for structural conceptual models", PhD thesis, Centre for Telematics and Information Technology, Enschede, Netherlands, 2005. [Online]. Available: `http://doc.utwente.nl/50826/` (cit. on p. 171).

[Gui12] H. Guihot, *Pro Android Apps Performance Optimization*. Apress, 2012. DOI: `10.1007/978-1-4302-4000-6` (cit. on pp. 99, 101, 103).

[GvDS13] M. Greiler, A. van Deursen and M.-A. Storey, "Automated Detection of Test Fixture Strategies and Smells", in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, Mar. 2013, pp. 322–331. DOI: `10.1109/ICST.2013.45` (cit. on p. 35).

[GW09] H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronization", English, *Software & Systems Modeling*, vol. 8, no. 1, pp. 21–43, 2009, ISSN: 1619-1366. DOI: `10.1007/s10270-008-0089-9` (cit. on pp. 44, 47).

[GWRA12] S. Götz, C. Wilke, S. Richly and U. Aßmann, "Approximating quality contracts for energy auto-tuning software", in *Green and Sustainable Software (GREENS), 2012 First International Workshop on*, 2012, pp. 8–14. DOI: `10.1109/GREENS.2012.6224264` (cit. on p. 81).

[GZvDS13] M. Greiler, A. Zaidman, A. van Deursen and M.-A. Storey, "Strategies for Avoiding Text Fixture Smells during Software Evolution", in *10th IEEE Working Conference on Mining Software Repositories (MSR)*, May 2013, pp. 387–396. DOI: `10.1109/MSR.2013.6624053` (cit. on p. 35).

[Hax14] A. E. Haxthausen, "An Institution for Imperative RSL Specifications", English, in *Specification, Algebra, and Software*, ser. Lecture Notes in Computer Science, S. Iida, J. Meseguer and K. Ogata, Eds., vol. 8373, Springer Berlin Heidelberg, 2014, pp. 441–464, ISBN: 978-3-642-54623-5. DOI: `10.1007/978-3-642-54624-2_22` (cit. on p. 71).

[HB10] H. Höpfner and C. Bunse, "Towards an Energy-Consumption Based Complexity Classification for Resource Substitution Strategies", in *Proceedings of the 22nd Workshop "Grundlagen von Datenbanken 2010", CEUR Workshop Proceedings*, W.-T. Balke and C. Lofi, Eds., vol. 581, 2010 (cit. on pp. 34, 96, 97).

[HBJ08] M. Herrmannsdörfer, S. Benz and E. Juergens, "Automatability of Coupled Evolution of Metamodels and Models in Practice", in *Model Driven Engineering Languages and Systems*, ser. LNCS, K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl and M. Völter, Eds., vol. 5301, Springer, 2008, pp. 645–659, ISBN: 978-3-540-87874-2 (cit. on p. 41).

[HBJ09] M. Herrmannsdörfer, S. Benz and E. Juergens, "COPE - Automating Coupled Evolution of Metamodels and Models", in *ECOOP 2009*, ser. LNCS, S. Drossopoulou, Ed., vol. 5653, Springer, 2009, pp. 52–76, ISBN: 978-3-642-03012-3 (cit. on pp. 41, 46).

[HCW07] A. Hessellund, K. Czarnecki and A. Wąsowski, "Guided Development with Multiple Domain-Specific Languages", English, in *Model Driven Engineering Languages and*

Systems, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. C. Schmidt and F. Weil, Eds., vol. 4735, Springer Berlin Heidelberg, 2007, pp. 46–60, ISBN: 978-3-540-75208-0. DOI: 10.1007/978-3-540-75209-7_4 (cit. on p. 111).

[HEO+15] F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, Y. Xiong, S. Gottmann and T. Engel, "Model synchronization based on triple graph grammars: correctness, completeness and invertibility", English, *Software & Systems Modeling*, vol. 14, no. 1, pp. 241–269, 2015, ISSN: 1619-1366. DOI: 10.1007/s10270-012-0309-1 (cit. on p. 44).

[HJK+09] F. Heidenreich, J. Johannes, S. Karol, M. Seifert and C. Wende, "Derivation and Refinement of Textual Syntax for Models", in *Proc. of the 5th Europ. Conf. on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009)*, ser. LNCS, vol. 5562, Enschede, The Netherlands: Springer, 2009, pp. 114–129, ISBN: 978-3-642-02673-7 (cit. on p. 5).

[HJSW10] F. Heidenreich, J. Johannes, M. Seifert and C. Wende, "Closing the Gap between Modelling and Java", English, in *Software Language Engineering*, ser. Lecture Notes in Computer Science, M. van den Brand, D. Gašević and J. Gray, Eds., vol. 5969, Springer Berlin Heidelberg, 2010, pp. 374–383, ISBN: 978-3-642-12106-7. DOI: 10.1007/978-3-642-12107-4_25 (cit. on pp. 9, 18, 188).

[HK10] M. Herrmannsdoerfer and M. Koegel, "Towards Semantics-Preserving Model Migration", in *International Workshop on Models and Evolutions*, Oslo, Norway, Oct. 2010 (cit. on p. 41).

[HMK05] J. Hannemann, G. C. Murphy and G. Kiczales, "Role-based Refactoring of Crosscutting Concerns", in *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, ser. AOSD '05, Chicago, Illinois: ACM, 2005, pp. 135–146, ISBN: 1-59593-042-6. DOI: 10.1145/1052898.1052910 (cit. on pp. 29, 31, 60, 76).

[Hor08] I. Horrocks, "Ontologies and the Semantic Web", *Communications of the ACM*, vol. 51, no. 12, pp. 58–67, Dec. 2008, ISSN: 0001-0782. DOI: 10.1145/1409360.1409377 (cit. on p. 156).

[HS06] H.-J. Happel and S. Seedorf, "Applications of Ontologies in Software Engineering", in *Proceedings of the 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE'06)*, 2006 (cit. on pp. 107–109).

[HVW11] M. Herrmannsdörfer, S. Vermolen and G. Wachsmuth, "An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models", in *Software Language Engineering*, ser. LNCS, B. Malloy, S. Staab and M. van den Brand, Eds., vol. 6563, Springer, 2011, pp. 163–182, ISBN: 978-3-642-19439-9 (cit. on pp. 41, 47).

[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. 3).

[ISO01] *Software engineering – Product quality – Part 1: Quality model*, ISO/IEC, 2001. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749 (cit. on p. 81).

[ISO96] *Information technology – Syntactic metalanguage – Extended BNF*, ISO/IEC, 1996. [Online]. Available: http://www.iso.org/iso/catalogue_detail?csnumber=26153 (cit. on pp. 122, 187).

[JK06] F. Jouault and I. Kurtev, "Transforming Models with ATL", English, in *Satellite Events at the MoDELS 2005 Conference*, ser. Lecture Notes in Computer Science, J.-M. Bruel, Ed., vol. 3844, Springer Berlin Heidelberg, 2006, pp. 128–138, ISBN: 978-3-540-31780-7. DOI: 10.1007/11663430_14 (cit. on pp. 45, 187).

[KDPP09] D. Kolovos, D. Di Ruscio, A. Pierantonio and R. F. Paige, "Different Models for Model Matching: An analysis of approaches to support model differencing", in *International Workshop on Comparison and Versioning of Software Models, MCVS'09 at ICSE'09*, IEEE Computer Society, 2009 (cit. on p. 77).

[KE07] D.-K. Kim and C. El Khawand, "An Approach to Precisely Specifying the Problem Domain of Design Patterns", *Journal of Visual Languages and Computing*, vol. 18, no. 6, pp. 560–591, 2007 (cit. on pp. 33, 87).

[KGH10] O. Kaczor, Y.-G. Guéhéneuc and S. Hamel, "Identification of design motifs with pattern matching algorithms", *Information and Software Technology*, vol. 52, no. 2, pp. 152–168, 2010, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.08.006 (cit. on pp. 33, 87).

[KKK+06] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger and M. Wimmer, "On Models and Ontologies - A Layered Approach for Model-based Tool Integration", in *Proceedings of Modellierung 2006*, H. C. Mayr and R. Breu, Eds., ser. Lecture Notes in Informatics, vol. GI-Edition, Innsbruck, Austria, 2006 (cit. on p. 108).

[Kle09] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Pearson Education, 2009, ISBN: 0321553454 (cit. on pp. 4, 16, 19).

[KLG+14] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl and U. Aßmann, "A Metamodel Family for Role-Based Modeling and Programming Languages", in *Software Language Engineering*, Springer, 2014, pp. 141–160 (cit. on pp. 19, 57, 171, 172).

[KLR+12] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger and M. Wimmer, "Model Transformation By-Example: A Survey of the First Wave", English, in *Conceptual Modelling and Its Theoretical Foundations*, ser. Lecture Notes in Computer Science, A. Düsterhöft, M. Klettke and K.-D. Schewe, Eds., vol. 7260, Springer Berlin Heidelberg, 2012, pp. 197–215, ISBN: 978-3-642-28278-2. DOI: 10.1007/978-3-642-28279-9_15 (cit. on p. 30).

[Köh06] C. Köhler, "A Visual Model Transformation Environment for the Eclipse Modeling Framework", Diploma Thesis, TU Berlin, Oct. 2006 (cit. on p. 29).

[Koz11] A. Koziolek, "Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes", PhD thesis, Institut für Programmstrukturen und Datenorganisation (IPD), Karlsruher Institut für Technologie, Karlsruhe, 2011. [Online]. Available: http://nbn-resolving.org/urn:nbn:de:swb:90-249552 (cit. on pp. 36–38, 49, 79, 87, 166, 167).

[KPPR07] D. Kolovos, R. F. Paige, F. Polack and L. M. Rose, "Update transformations in the small with the Epsilon Wizard Language", *Journal of Object Technology*, vol. 6, no. 9, pp. 53–69, 2007 (cit. on pp. 28, 35, 188).

[KRGP13] D. Kolovos, L. Rose, A. García-Domínguez and R. Paige. (Apr. 2013). The Epsilon Book (cit. on pp. 28, 35, 38).

[KT08] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008 (cit. on pp. 3–5).

[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 pp. 5, 6).

[KVGS11] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc and H. Sahraoui, "BDTEX: A GQM-based Bayesian approach for the detection of antipatterns", *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011, ISSN: 0164-1212. DOI: 10.1016/j.jss.2010.11.921 (cit. on pp. 33, 37, 87).

[Läm02] R. Lämmel, "Towards Generic Refactoring", in *Proceedings of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA: ACM Press, Oct. 2002 (cit. on pp. 27, 31, 70).

[Läm04] R. Lämmel, "Coupled Software Transformations – Extended Abstract –", in *1st International Workshop on Software Evolution Transformations*, Y. Zou and J. R. Cordy, Eds., 2004, pp. 31–35 (cit. on p. 40).

[LBD05] R. Lubke, J. Ball and P. Delisle. (Aug. 2005). Unified Expression Language. Accessed: 2015-04-26 (Archived by WebCite® at http://www.webcitation.org/6Y54zCW8u), [Online]. Available: http://www.oracle.com/technetwork/java/unifiedel-139263.html (cit. on p. 116).

[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 pp. 4, 5, 19).

[LGJ07] Y. Lin, J. Gray and F. Jouault, "DSMDiff: a differentiation tool for domain-specific models", *European Journal of Information Systems*, vol. 16, pp. 349–361, 2007 (cit. on pp. 77, 78).

[LM06] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice – Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Berlin Heidelberg, 2006 (cit. on p. 84).

[LMB+01] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. G. Nordstrom, J. Sprinkle and P. Volgyesi, "The Generic Modeling Environment", in *Workshop on Intelligent Signal Processing*, Budapest, Hungary, 2001 (cit. on pp. 5, 27, 188).

[LWK10] P. Langer, M. Wimmer and G. Kappel, "Model-to-Model Transformations By Demonstration", in *Theory and Practice of Model Transformations*, ser. Lecture Notes in Computer Science, L. Tratt and M. Gogolla, Eds., vol. 6142, Springer Berlin Heidelberg, 2010, pp. 153–167, ISBN: 978-3-642-13687-0. DOI: 10.1007/978-3-642-13688-7_11 (cit. on pp. 30, 31).

[LXC14] Y. Liu, C. Xu and S.-C. Cheung, "Characterizing and Detecting Performance Bugs for Smartphone Applications", in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: ACM, 2014, pp. 1013–1024, ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568229 (cit. on p. 34).

[Mah81] H. Mahler (today Heike Reimann), "Der Beitrag Vladislav Krapivins zur zeitgenössischen sowjetischen Prosa für Kinder", Dissertation, Pädagogische Hochschule Potsdam, 1981 (cit. on p. 1).

[Mar01] R. Marinescu, "Detecting Design Flaws via Metrics in Object-Oriented Systems", in *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS'01)*, 2001, pp. 173–182. DOI: 10.1109/TOOLS.2001.941671 (cit. on pp. 37, 87).

[Mar04] R. Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws", in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, Sep. 2004, pp. 350–359. DOI: 10.1109/ICSM.2004.1357820 (cit. on p. 84).

[MBF11] S. Murer, B. Bonati and F. J. Furrer, *Managed Evolution: A Strategy for Very Large Information Systems*. Springer, 2011, ISBN: 978-3-642-01632-5 (cit. on p. 45).

[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. 6).

[MGDL10] N. Moha, Y. Guéhéneuc, L. Duchien and A. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells", *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, 2010, ISSN: 0098-5589. DOI: 10.1109/TSE.2009.50 (cit. on p. 37).

[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 pp. 11, 26, 27, 31).

[MRG09] M. Mohamed, M. Romdhani and K. Ghedira, "Classification of model refactoring approaches", *Journal of Object Technology (JOT)*, vol. 8, no. 6, pp. 143–158, 2009 (cit. on p. 67).

[MRW77] J. A. McCall, P. K. Richards and G. F. Walters, "Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality", General Electric Co. Sunnyvale California, Tech. Rep. ADA 049014, 1977 (cit. on p. 81).

[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 pp. 9–11, 16, 25, 32, 52, 67, 79, 80, 146).

[MTM08] T. Mens, G. Taentzer and D. Müller, "Model-Driven Software Refactoring. Integrating Quality Assurance", in. IGI Global, 2008, ch. 8, pp. 170–203 (cit. on p. 29).

[MV06] T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation", in *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)*, vol. 152, 2006, pp. 125–142. DOI: 10.1016/j.entcs.2005.10.021 (cit. on pp. 19, 25).

[MWCS11] B. Meyers, M. Wimmer, A. Cicchetti and J. Sprinkle, "A generic in-place transformation-based approach to structured model co-evolution", *Electronic Communications of the European Association of Software Science and Technology (ECEASST)*, vol. 42, p. 13, 2011 (cit. on pp. 41, 47).

[NB07] H. Neukirchen and M. Bisanz, "Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites", in *Testing of Software and Communicating Systems*, ser. Lecture Notes in Computer Science, A. Petrenko, M. Veanes, J. Tretmans and W.

Grieskamp, Eds., vol. 4581, Springer Berlin Heidelberg, 2007, pp. 228–243, ISBN: 978-3-540-73065-1. DOI: 10.1007/978-3-540-73066-8_16 (cit. on pp. 34, 35, 37, 38, 80).

[NPA91] E. J. Neuhold, M. Paul and K. R. Apt, *Formal Description of Programming Concepts*. Springer Science & Business Media, 1991 (cit. on p. 71).

[NTVW15] P. Neron, A. P. Tolmach, E. Visser and G. Wachsmuth, "A Theory of Name Resolution", in *24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015*, J. Vitek, Ed., London, UK, Apr. 2015 (cit. on p. 27).

[OGB+11] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk and A. De Lucia, "Identifying Method Friendships to Remove the Feature Envy Bad Smell (NIER Track)", in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 820–823, ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985913 (cit. on p. 80).

[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 pp. 5, 15).

[OMG03] The Object Management Group, *MDA Guide Version 1.0.1*, Object Management Group, 2003. [Online]. Available: http://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf (cit. on pp. 18, 188).

[OMG11a] The Object Management Group, *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*, Aug. 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF (cit. on pp. 3, 17, 150, 151, 189).

[OMG11b] The Object Management Group, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1*, Jul. 2011. [Online]. Available: http://www.omg.org/spec/MARTE/1.1/PDF (cit. on p. 36).

[OMG13a] The Object Management Group, *Requirements Interchange Format (ReqIF) – Version 1.1*, 2013. [Online]. Available: http://www.omg.org/spec/ReqIF/1.1/PDF (cit. on p. 168).

[OMG13b] The Object Management Group, *Business Process Model and Notation (BPMN), Version 2.0.2*, Dec. 2013. [Online]. Available: http://www.omg.org/spec/BPMN/2.0.2/PDF (cit. on pp. 3, 187).

[OMG13c] The Object Management Group, *OMG Meta Object Facility (MOF) Core Specification – Version 2.4.1*, Jun. 2013. [Online]. Available: http://www.omg.org/spec/MOF/2.4.1/PDF (cit. on pp. 17, 171, 188).

[OMG14a] The Object Management Group, *Object Constraint Language, Version 2.4*, 2014. [Online]. Available: http://www.omg.org/spec/OCL/2.4/PDF (cit. on pp. 28, 188).

[OMG14b] The Object Management Group, *XML Metadata Interchange (XMI) Specification – Version 2.4.2*, Apr. 2014. [Online]. Available: http://www.omg.org/spec/XMI/2.4.2/PDF (cit. on pp. 42, 189).

[Opd92] W. F. Opdyke, "Refactoring Object-Oriented Frameworks", PhD thesis, University of Illinois at Urbana-Champaign, 1992 (cit. on pp. 5, 15, 16, 25, 67).

[OSGi14] The OSGi Alliance, *OSGi Core – Release 6*, 2014. [Online]. Available: https://osgi.org/download/r6/osgi.core-6.0.0.pdf (cit. on pp. 121, 188).

[Pfe13] R.-H. Pfeiffer, "Multi-language Development Environments - Design Space, Models, Prototypes, Experiences", PhD thesis, IT University of Copenhagen, Software Development Group, 2013 (cit. on pp. 8, 45, 47–49, 105, 114).

[Pfl98] S. L. Pfleeger, *Software Engineering: Theory and Practice*. Prentice Hall, 1998 (cit. on p. 5).

[PHZ11] A. Pathak, Y. C. Hu and M. Zhang, "Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices", in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, ser. HotNets-X, Cambridge, Massachusetts: ACM, 2011, 5:1–5:6, ISBN: 978-1-4503-1059-8. DOI: 10.1145/2070562.2070567 (cit. on p. 34).

[PJHM12] A. Pathak, A. Jindal, Y. C. Hu and S. P. Midkiff, "What is keeping my phone awake? Characterizing and Detecting No-Sleep Energy Bugs in Smartphone Apps", in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, ser. MobiSys '12, Low Wood Bay, Lake District, UK: ACM, 2012, pp. 267–280, ISBN: 978-1-4503-1301-8. DOI: 10.1145/2307636.2307661 (cit. on pp. 34, 37, 38).

[PKPS02] M. Paolucci, T. Kawamura, T. Payne and K. Sycara, "Semantic Matching of Web Services Capabilities", English, in *The Semantic Web — ISWC 2002*, ser. Lecture Notes in Computer Science, I. Horrocks and J. Hendler, Eds., vol. 2342, Springer Berlin Heidelberg, 2002, pp. 333–347, ISBN: 978-3-540-43760-4. DOI: 10.1007/3-540-48005-6_26 (cit. on p. 167).

[PRH10] D. C. Petriu, N. Rouquette and Ø. Haugen, Eds., *Model Driven Engineering Languages and Systems - 13th International Conference, MoDELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, vol. 6394, ser. Lecture Notes in Computer Science, Springer, 2010, ISBN: 978-3-642-16144-5.

[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 pp. 45, 46, 105, 113).

[PS92] B. Peuschel and W. Schäfer, "Concepts and Implementation of a Rule-based Process Engine", in *Proceedings of the 14th International Conference on Software Engineering (ICSE)*, ser. ICSE '92, Melbourne, Australia: ACM, 1992, pp. 262–279, ISBN: 0-89791-504-6. DOI: 10.1145/143062.143126 (cit. on p. 111).

[PW11] R.-H. Pfeiffer and A. Wąsowski, "Taming the Confusion of Languages", in *7th European Conference on Modelling Foundations and Applications*, R. France, J. Kuester, B. Bordbar and R. Paige, Eds., ser. LNCS, vol. 6698, Springer, 2011, pp. 312–328, ISBN: 978-3-642-21469-1 (cit. on p. 45).

[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 pp. 8, 45, 110).

[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 In-*

ternational 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 pp. 37, 79).

[RASG14] E. Riccobene, P. Arcaini, P. Scandurra and A. Gargantini, "Formal Semantics for Metamodel-Based Domain Specific Languages", in. IGI Global, 2014, ch. 15 (cit. on p. 71).

[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 pp. 34, 37, 89, 92, 93).

[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. 25).

[Ren04] A. Rensink, "Representing First-Order Logic Using Graphs", English, in *Graph Transformations*, ser. Lecture Notes in Computer Science, H. Ehrig, G. Engels, F. Parisi-Presicce and G. Rozenberg, Eds., vol. 3256, Springer Berlin Heidelberg, 2004, pp. 319–335, ISBN: 978-3-540-23207-0. DOI: 10.1007/978-3-540-30203-2_23 (cit. on p. 111).

[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 pp. 11, 19, 20, 54).

[RGdL+13] L. Rose, E. Guerra, J. de Lara, A. Etien, D. Kolovos and R. Paige, "Genericity for model management operations", English, *Software & Systems Modeling*, vol. 12, no. 1, pp. 201–219, 2013, ISSN: 1619-1366. DOI: 10.1007/s10270-011-0203-2 (cit. on pp. 27, 31).

[RKP+14] L. M. Rose, D. S. Kolovos, R. F. Paige, F. A. C. Polack and S. Poulding, "Epsilon Flock: a model migration language", *Software & Systems Modeling*, vol. 13, no. 2, pp. 735–755, 2014, ISSN: 1619-1366. DOI: 10.1007/s10270-012-0296-2 (cit. on pp. 41, 42, 47, 48, 60, 61).

[Rob99] D. B. Roberts, "Practical Analysis for Refactoring", PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1999, ISBN: 0-599-46857-2 (cit. on p. 67).

[Rot89] J. Rothenberg, "The Nature of Modeling", pp. 75–92, 1989 (cit. on p. 16).

[Roz97] G. Rozenberg, Ed., *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997 (cit. on p. 18).

[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 pp. 51, 172).

[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 pp. 27, 45, 51, 73, 172).

[RV07] J. E. Rivera and A. Vallecillo, "Adding Behavior to Models", in *11th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2007*, Oct. 2007, pp. 169–169. DOI: 10.1109/EDOC.2007.40 (cit. on p. 71).

[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 pp. 11, 19, 54, 57).

[RWZ11] T. Ruhroth, H. Wehrheim and S. Ziegert, "ReL: A Generic Refactoring Language for Specification and Execution", in *37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA) 2011*, Aug. 2011, pp. 83–90. DOI: 10.1109/SEAA.2011.22 (cit. on pp. 28, 31).

[Sai03] J. Said, "Pattern-Based Approach for Object Oriented Software Design", PhD thesis, KU Leuven, 2003. [Online]. Available: http://www.cs.kuleuven.be/publicaties/doctoraten/cw/CW2003_04.pdf (cit. on pp. 12, 92, 166–168).

[Şav10] I. Şavga, "A Refactoring-Based Approach to Support Binary Backward-Compatible Framework Upgrades", PhD thesis, Technische Universität Dresden, 2010. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-38533 (cit. on p. 111).

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro and E. Merks, *Eclipse Modeling Framework*, 2nd. Addison-Wesley, Pearson Education, 2008, ISBN: 0321331885 (cit. on pp. 5, 18, 92, 121, 187).

[SDM13] S. M. A. Shah, J. Dietrich and C. McCartin, "On the Automation of Dependency-Breaking Refactorings in Java", in *29th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2013, pp. 160–169. DOI: 10.1109/ICSM.2013.27 (cit. on p. 36).

[Sei11] M. Seifert, "Designing Round-Trip Systems by Change Propagation and Model Partitioning", PhD thesis, Technische Universität Dresden, 2011. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-71098 (cit. on pp. 45, 47).

[SGdL14] J. Sánchez Cuadrado, E. Guerra and J. de Lara, "A Component Model for Model Transformations", *IEEE Transactions on Software Engineering*, vol. 40, no. 11, pp. 1042–1060, Nov. 2014, ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2339852 (cit. on pp. 27, 28).

[SHV13] O. Semeráth, Á. Horváth and D. Varró, "Validation of Derived Features and Well-Formedness Constraints in DSLs", English, in *Model-Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, A. Moreira, B. Schätz, J. Gray, A. Vallecillo and P. Clarke, Eds., vol. 8107, Springer Berlin Heidelberg, 2013, pp. 538–554, ISBN: 978-3-642-41532-6. DOI: 10.1007/978-3-642-41533-3_33 (cit. on p. 92).

[Sie14] K. Siegemund, "Contributions To Ontology-Driven Requirements Engineering", PhD thesis, Technische Universität Dresden, 2014 (cit. on p. 109).

[SK11] S. Singh and K. S. Kahlon, "Effectiveness of Encapsulation and Object-oriented Metrics to Refactor Code and Identify Error Prone Classes Using Bad Smells",

SIGSOFT Software Engineering Notes, vol. 36, no. 5, pp. 1–10, Sep. 2011, ISSN: 0163-5948. DOI: 10.1145/2020976.2020994 (cit. on pp. 33, 37, 87).

[SMM+12] S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry and J.-M. Jézéquel, "Reusable model transformations", English, *Software & Systems Modeling*, vol. 11, no. 1, pp. 111–125, 2012, ISSN: 1619-1366. DOI: 10.1007/s10270-010-0181-9 (cit. on p. 26).

[Soc93] I. C. Society, "IEEE Standard for a Software Quality Metrics Methodology", *IEEE Std 1061-1992*, 1993. DOI: 10.1109/IEEESTD.1993.115124 (cit. on pp. 33, 87).

[SPAS03] K. Sycara, M. Paolucci, A. Ankolekar and N. Srinivasan, "Automated discovery, interaction and composition of Semantic Web services", *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 1, pp. 27–46, 2003, ISSN: 1570-8268. DOI: 10.1016/j.websem.2003.07.002 (cit. on p. 167).

[SPLJ01] G. Sunyé, D. Pollet, Y. Le Traon and J.-M. Jézéquel, "Refactoring UML Models", in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, Springer, 2001, pp. 134–148 (cit. on p. 28).

[SS09] A. Schürr and B. Selic, Eds., *Model Driven Engineering Languages and Systems – 12th International Conference, MoDELS 2009*, vol. 5795, ser. Lecture Notes in Computer Science, Denver, USA: Springer, Oct. 2009, ISBN: 978-3-642-04424-3. DOI: 10.1007/978-3-642-04425-0.

[SSA14] C. Seidl, I. Schaefer and U. Aßmann, "DeltaEcore—A Model-Based Delta Language Generation Framework", in *Modellierung 2014*, ser. Lecture Notes in Informatics, H.-G. Fill, D. Karagiannis and U. Reimer, Eds., vol. P-225, Bonn: Gesellschaft für Informatik, 2014, pp. 81–96 (cit. on p. 171).

[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 pp. 9, 32, 33, 37, 79, 87).

[Ste00] F. Steimann, "On the representation of roles in object-oriented and conceptual modelling", *Data & Knowledge Engineering*, vol. 35, no. 1, pp. 83–106, 2000, ISSN: 0169-023X. DOI: 10.1016/S0169-023X(00)00023-9 (cit. on pp. 19, 171, 172).

[Ste11] F. Steimann, "Constraint-Based Model Refactoring", English, in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, J. Whittle, T. Clark and T. Kühne, Eds., vol. 6981, Springer Berlin Heidelberg, 2011, pp. 440–454, ISBN: 978-3-642-24484-1. DOI: 10.1007/978-3-642-24485-8_32 (cit. on pp. 28, 30–32, 42, 43, 48, 71).

[Ste15] F. Steimann, "From well-formedness to meaning preservation: model refactoring for almost free", English, *Software & Systems Modeling*, vol. 14, no. 1, pp. 307–320, 2015, ISSN: 1619-1366. DOI: 10.1007/s10270-013-0314-z (cit. on pp. 42, 43, 71).

[STZ+11] K. Siegemund, E. J. Thomas, Y. Zhao, J. Z. Pan and U. Aßmann, "Towards Ontology-driven Requirements Engineering", in *SWESE2011 – The 7th International Workshop on Semantic Web Enabled Software Engineering Co-located with ISWC2011*, K. Bontcheva, J. Z. Pan and Y. Zhao, Eds., Bonn, Germany, 2011 (cit. on pp. 107, 108).

[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 pp. 4, 16, 188).

[SW03] C. U. Smith and L. G. Williams, "More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot", in *Computer Measurement Group Conference 2003*, Computer Measurement Group, 2003, pp. 717–725 (cit. on pp. 33, 87).

[SWG09] Y. Sun, J. White and J. Gray, "Model Transformation by Demonstration", English, in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, A. Schürr and B. Selic, Eds., vol. 5795, Springer Berlin Heidelberg, 2009, pp. 712–726, ISBN: 978-3-642-04424-3. DOI: 10.1007/978-3-642-04425-0_58 (cit. on pp. 30, 31).

[TDDN00] S. Tichelaar, S. Ducasse, S. Demeyer and O. Nierstrasz, "A Meta-model for Language-Independent Refactoring", *International Symposium on Principles of Software Evolution*, pp. 157–167, Nov. 2000. DOI: 10.1109/ISPSE.2000.913233 (cit. on pp. 16, 27, 31).

[Tit11] E. Tittel, "Refactoring in der Ontologiegetriebenen Softwareentwicklung", Diploma Thesis, Technische Universität Dresden, Dresden, Germany, 2011. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-69119 (cit. on p. 156).

[TKB+14] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake and T. Leich, "FeatureIDE: An extensible framework for feature-oriented software development", *Science of Computer Programming*, vol. 79, pp. 70–85, 2014, Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010), ISSN: 0167-6423. DOI: 10.1016/j.scico.2012.06.002 (cit. on p. 171).

[TM15] M. Toll and W. R. Minto. (2015). Principles of Pattern Enabled Java Development, [Online]. Available: http://gtcgroup.com/ped/toll.pdf (cit. on p. 12).

[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 pp. 11, 29).

[TOHS99] P. Tarr, H. Ossher, W. Harrison and S. M. Sutton Jr., "N degrees of separation: multi-dimensional separation of concerns", in *Proceedings of the 21st international conference on Software engineering (ICSE '99)*, ser. ICSE '99, Los Angeles, California, USA: ACM, 1999, pp. 107–119, ISBN: 1-58113-074-0. DOI: 10.1145/302405.302457 (cit. on pp. 35, 37, 80).

[Tri07] T. Triebsees, "Constraint-based Model Transformation: Tracing the Preservation of Semantic Properties", *Journal of Software*, vol. 2, no. 3, 2007. DOI: 10.4304/jsw.2.3.19-29 (cit. on p. 71).

[Tru11] C. Trubiani, "Automated generation of architectural feedback from software performance analysis results", PhD thesis, University of L'Aquila, Italy, 2011 (cit. on pp. 36–38).

[USH+15] Z. Ujhelyi, G. Szőke, Á. Horváth, N. I. Csiszár, D. Vidács László Varró and R. Ferenc, "Performance Comparison of Query-based Techniques for Anti-Pattern Detection",

Information and Software Technology, 2015. DOI: 10.1016/j.infsof.2015.01.003 (cit. on p. 37).

[VAPM13] A. Vetro', L. Ardito, G. Procaccianti and M. Morisio, "Definition, Implementation and Validation of Energy Code Smells: an Exploratory Study on an Embedded System", in *ENERGY 2013, The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, S. Fries and P. Dini, Eds., Lisbon, Portugal, Mar. 2013 (cit. on p. 34).

[Var02] D. Varró, "A Formal Semantics of UML Statecharts by Model Transition Systems", English, in *Graph Transformation*, ser. Lecture Notes in Computer Science, A. Corradini, H. Ehrig, H.-J. Kreowski and G. Rozenberg, Eds., vol. 2505, Springer Berlin Heidelberg, 2002, pp. 378–392, ISBN: 978-3-540-44310-0. DOI: 10.1007/3-540-45832-8_28 (cit. on p. 71).

[Var06] D. Varró, "Model Transformation by Example", English, in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, O. Nierstrasz, J. Whittle, D. Harel and G. Reggio, Eds., vol. 4199, Springer Berlin Heidelberg, 2006, pp. 410–424, ISBN: 978-3-540-45772-5. DOI: 10.1007/11880240_29 (cit. on p. 30).

[VD06] R. Van Der Straeten and M. D'Hondt, "Model Refactorings Through Rule-based Inconsistency Resolution", in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ser. SAC '06, Dijon, France: ACM, 2006, pp. 1210–1217, ISBN: 1-59593-108-2. DOI: 10.1145/1141277.1141564 (cit. on pp. 29, 31, 32, 67).

[vDMT10] M. von Detten, M. Meyer and D. Travkin, "Reverse Engineering with the Reclipse Tool Suite", in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE '10, vol. 2, Cape Town, South Africa: ACM, 2010, pp. 299–300, ISBN: 978-1-60558-719-6. DOI: 10.1145/1810295.1810360 (cit. on p. 92).

[Vector13] Vector Informatik GmbH. (2013). Modellbasierte Elektrik-/Elektronik-Entwicklung vom Architekturentwurf bis zur Serienreife. visited 5th March 2015, [Online]. Available: http://vector.com/portal/medien/cmc/marketing%5C_items/web/91106.pdf (cit. on p. 168).

[VJM07] R. Van Der Straeten, V. Jonckers and T. Mens, "A formal approach to model refactoring and model refinement", *Software & System Modeling*, vol. 6, no. 2, pp. 139–162, 2007. DOI: 10.1007/s10270-006-0025-9 (cit. on p. 16).

[Von13] C. Vonsien, "Spezifikation von Model Smells zum Vorschlagen von Modell- Refactorings", Minor Thesis (Großer Beleg), Technische Universität Dresden, 2013 (cit. on p. 79).

[vPUTS13] J. von Pilgrim, B. Ulke, A. Thies and F. Steimann, "Model/code co-refactoring: An MDE approach", in *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 2013*, Nov. 2013, pp. 682–687. DOI: 10.1109/ASE.2013.6693133 (cit. on pp. 42, 43, 47, 48, 114).

[vRDDR07] B. van Rompaey, B. Du Bois, S. Demeyer and M. Rieger, "On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test", *Software Engineering, IEEE Transactions on*, vol. 33, no. 12, pp. 800–817, 2007, ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70745 (cit. on p. 35).

[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. 6).

[W3C11] The World Wide Web Consortium, *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*, Jun. 2011. [Online]. Available: http://www.w3.org/TR/2011/REC-CSS2-20110607/ (cit. on pp. 3, 187).

[W3C12] The World Wide Web Consortium, *OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax*, Dec. 2012. [Online]. Available: http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/ (cit. on pp. 13, 188).

[Wac07] G. Wachsmuth, "Metamodel Adaptation and Model Co-adaptation", in *ECOOP*, E. Ernst, Ed., ser. Lecture Notes in Computer Science, vol. 4609, Springer, 2007, pp. 600–624, ISBN: 978-3-540-73588-5 (cit. on pp. 40, 46, 47).

[Wil14] C. Wilke, "Energy-Aware Development and Labeling for Mobile Applications", PhD thesis, Technische Universität Dresden, 2014. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-139391 (cit. on pp. 34, 88, 90, 167).

[Wir86] N. Wirth, *Compilerbau - Eine Einführung*. Teubner, 1986, ISBN: 3519323389 (cit. on p. 130).

[WMV12] M. Wimmer, N. Moreno and A. Vallecillo, "Viewpoint Co-evolution through Coarse-Grained Changes and Coupled Transformations", English, in *Objects, Models, Components, Patterns*, ser. Lecture Notes in Computer Science, C. Furia and S. Nanz, Eds., vol. 7304, Springer Berlin Heidelberg, 2012, pp. 336–352, ISBN: 978-3-642-30560-3. DOI: 10.1007/978-3-642-30561-0_23 (cit. on pp. 43, 44, 47).

[WTW10] C. Wilke, M. Thiele and C. Wende, "Extending Variability for OCL Interpretation", in *Model Driven Engineering Languages and Systems - 13th International Conference, MoDELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, D. C. Petriu, N. Rouquette and Ø. Haugen, Eds., ser. Lecture Notes in Computer Science, vol. 6394, Springer, 2010, pp. 361–375, ISBN: 978-3-642-16144-5 (cit. on pp. 28, 130).

[XLH+07] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi and H. Mei, "Towards Automatic Model Synchronization from Model Transformations", in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, Atlanta, Georgia, USA: ACM, 2007, pp. 164–173, ISBN: 978-1-59593-882-4. DOI: 10.1145/1321631.1321657 (cit. on pp. 45–47).

[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. 5).

[ZLG05] J. Zhang, Y. Lin and J. Gray, "Generic and Domain-Specific Model Refactoring using a Model Transformation Engine", in *Volume II of Research and Practice in Software Engineering*, Springer, 2005, pp. 199–218 (cit. on pp. 27, 31, 32).

[ZVS+07] B. Zeiß, D. Vega, I. Schieferdecker, H. Neukirchen and J. Grabowski, "Applying the ISO 9126 Quality Model to Test Specifications – Exemplified for TTCN-3 Test Specifications", in *Software Engineering 2007 (SE 2007)*, W.-G. Bleek, J. Raasch and H. Züllighoven, Eds., ser. Lecture Notes in Informatics (LNI), Gesellschaft für

Informatik, vol. 105, Bonn, Germany: Köllen Verlag, Mar. 2007, pp. 231–242 (cit. on p. 35).