Autom Softw Eng (2013) 20:141–184
DOI 10.1007/s10515-012-0114-7

A tool environment for quality assurance based on the Eclipse Modeling Framework

Thorsten Arendt · Gabriele Taentzer

Received: 30 March 2012 / Accepted: 28 November 2012 / Published online: 11 December 2012

Abstract The paradigm of model-based software development has become more and more popular since it promises an increase in the efficiency and quality of software development. Following this paradigm, models become primary artifacts in the software development process. Therefore, software quality and quality assurance frequently leads back to the quality and quality assurance of the involved models. In our approach, we propose a model quality assurance process that can be adapted to project-specific and domain-specific needs. This process is based on static model analysis using model metrics and model smells. Based on the outcome of the model analysis, appropriate model refactoring steps can be performed. In this paper, we present a tool environment conveniently supporting the proposed model quality assurance process. In particular, the presented tools support metrics reporting, smell detection, and refactoring for models being based on the Eclipse Modeling Framework, a widely used open source technology in model-based software development.

Keywords Modeling · Model-based software development · Model quality · Model quality assurance · Eclipse Modeling Framework

1 Introduction

In modern software development, models play an increasingly important role promising a growth in efficiency and quality of software development. In particular, this is

T. Arendt (✉) · G. Taentzer
FB 12—Mathematics and Computer Science, Philipps-Universität Marburg, Hans-Meerwein-Strasse, 35032 Marburg, Germany
e-mail: arendt@mathematik.uni-marburg.de

G. Taentzer
e-mail: taentzer@mathematik.uni-marburg.de

true for model-driven software development where models are used directly for automatic code generation. High code quality can be reached only if the quality of input models is already high.

In our approach, we concentrate on quality aspects to be checked on the model syntax. They include not only the consistency with the language syntax definition, but also e.g. the conceptual integrity in using patterns and principles in similar situations, and the conformity with modeling conventions often defined and adapted to specific software projects. In Mohagheghi et al. (2009), six classes of quality goals for software models are identified. We take them as conceptual basis for a goal-question-metrics approach (Basili et al. 1994) to our quality assurance process for software models.

In the literature, well-known quality assurance techniques for models are model metrics and refactorings, see e.g. Genero et al. (2005), Sunyé et al. (2001), Markovic and Baar (2008), Zhang et al. (2005), Porres (2003), Lange (2007). They origin from corresponding techniques for software code by lifting them to models. Especially class models are closely related to programmed class structures in object-oriented programming languages such as C++ and Java. For behavior models, the relation between models and code is less obvious. Furthermore, the concept of code smells (Fowler 1999) can be lifted to models leading to model smells (compare e.g. Lange 2007). Again, code smells for class structures can be easily adapted to model smells, but smells of behavior models cannot directly be deduced from code smells.

In Arendt et al. (2011), we present the integration of these techniques in a predefined quality assurance process that can be adapted to specific project needs. It consists of two sub-processes: Before a software project starts, project- and domain-specific quality checks and refactorings have to be defined. Quality checks are formulated using model smells which can be specified e.g. by model metrics and anti-patterns. After formulating quality checks and refactorings, the specified quality assurance process can be applied to concrete software models by computing model metrics, reporting all model smells and applying model refactorings to erase smells that indicate clear model defects.

Since the process of manual model reviews is very time consuming and error prone, the proposed project-specific model quality assurance process should be automated as effectively as possible. In this article, we present a flexible tool environment for metrics reporting, smell detection, and refactoring of models being based on the Eclipse Modeling Framework (EMF) (Steinberg et al. 2008), a widely used open source technology in model-based software development. We integrated the entire tool set into the Eclipse incubation project *EMF Refactor* (EMF Refactor 2012).

The paper is organized as follows: In the next section, we motivate the development of the presented tool environment by a discussion of selected model quality aspects and an overview on the general approach of our contribution. In Sect. 3, we present the running example of this paper whereas Sect. 4 describes the state-of-the-art in model quality assurance tooling as well as the deduced requirements for our tool environment. Thereafter, we describe the architecture of our tools in Sect. 5. The application of our quality assurance tool environment to the running example is illustrated in Sect. 6. Afterwards, we present supported specification approaches for model metrics, smells, and refactorings in Sect. 7. We evaluate the tools in Sect. 8 and finally conclude in Sect. 9.

2 Model quality and quality assurance

In this section, we present the definition and application of a structured model quality assurance process that can be used to address project-specific and domain-specific needs (compare Arendt et al. 2011). The general approach uses known model quality assurance techniques being model metrics, model smells, and model refactorings. They are combined in an overall process for structured model quality assurance focusing on syntactical model issues. We start our presentation by presenting selected model quality aspects in software development that serve as the basis for our general approach.

2.1 The 6C model quality goals presented by Mohagheghi et al.

In their article, Mohagheghi et al. (2009) present the results of a systematic review of literature discussing model quality in model-based software development. Among others, the purpose of the review was to identify what model quality means, i.e. which quality goals are defined in literature. The review was performed systematically by searching relevant publication channels for papers published from 2000 to 2007. From 40 studies covered in the review, the authors identified six classes of quality goals, called *6C goals*, in model-based software development. They state that other quality goals discussed in literature can be satisfied if the 6C goals are in place. The remainder of this section shortly introduces the identified 6C goals.

`Correctness` A model is **correct** if it includes the right elements and correct relations between them, and if it includes correct statements about the domain. Furthermore, a model must not violate rules and conventions. This definition includes *syntactic correctness* relative to the modeling language as well as *semantic correctness* related to the understanding of the domain.

`Completeness` A model is **complete** if it contains all relevant information, and if it is detailed enough according to the purpose of modeling. For example, requirement models are said to be complete when they specify all the black-box behavior of the modeled entity, and when they do not include anything that is not in the real world.

`Consistency` A model is **consistent** if does not contain contradictions. This definition covers *horizontal consistency* concerning models/diagrams on the same level of abstraction, *vertical consistency* concerning modeled aspects on different levels of abstraction as well as *semantic consistency* concerning the meaning of the same element in different models or diagrams.

`Comprehensibility` A model is **comprehensible** if it is understandable by the intended users, being humans or tools. In most of the literature, the focus is on comprehensibility by humans including aspects like aesthetics of a diagram, model simplicity or complexity, and the use of the correct type of diagram for the intended audience. Several authors also call this goal *pragmatic quality*.

`Confinement` A model is **confined** if it suits to the modeling purpose and the type of system. This definition also includes relevant diagrams on the right abstraction level. Furthermore, a confined model does not have unnecessary information and is not more complex or detailed than necessary. Developing the right model for a

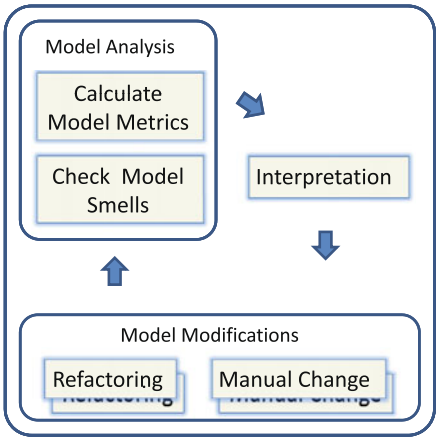

Fig. 1 Process for the application of project-specific model quality assurance techniques

system or purpose of a given kind also depends on selecting an adequate modeling language. This means that the modeler uses language concepts that are suitable for the intended purpose of the modeling activity. Further concepts should be used very sparsely or even omitted deliberately.

`Changeability` A model is **changeable** if it can be evolved rapidly and continuously. This is important since both the domain and its understanding as well as system requirements evolve over time. Furthermore, changeability should be supported by modeling languages and modeling tools as well.

2.2 Specification and application processes for customized model quality assurance techniques

The increasing use of model-based or model-driven software development processes induces the need for high-quality software models. Therefore, we propose a model quality assurance process that consists of two sub-processes: a process for the specification of project-specific model quality assurance techniques, and a process for applying them on concrete software models during a model-based software development process as shown in Fig. 1.

For a first rough model overview, a report on model metrics might be helpful. Furthermore, a model can be checked against the existence (respectively absence) of specified model smells. Each model smell found has to be interpreted in order to evaluate whether it should be eliminated by a suitable model modification (either by a manual model change or a refactoring). However, we have to take into account that also new model smells can be induced by refactorings and care should be taken to minimize this effect. This check-improve cycle should be performed as long as needed to get a reasonable model quality.

Ideally a quality assurance process is fully specified before using it within model-based software development projects. However, it is not seldom that the process has to be adapted during the model development phase. Our process allows the straight adaptation to new model checks and refactorings.

Figure 2 shows the process for specifying new model quality assurance techniques. After having identified the intended modeling purpose the most important quality

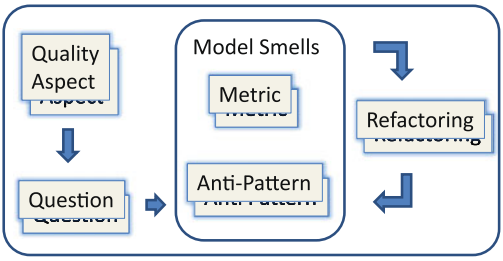

Fig. 2 Process for the specification of project-specific model quality assurance techniques

goals are selected. Here, we have to consider several conditions that influence the selection of significant quality aspects being the most important ones for modeling in a specific software project. The first issue to consider is that the selection of significant quality aspects highly depends on the modeling purpose. There is a variety of purposes for modeling in software projects. For example, models can be used for communication purposes between stakeholders, being customers and requirements engineers or project managers and software designers. In other projects, software models may be used for code generation purposes, to generate the application code and/or code that is used in tests for implemented software components. Since modeling purposes are quite different and vary in several software projects, a quality aspect that is very important in one software project might be less important in other ones. In projects that use software modeling for communication purposes the *comprehensibility* of the model might be the most relevant quality aspect whereas aspects *correctness* and *completeness* are more important for models that are used for the generation of application or test code, respectively.

Another factor that influences the significance of a model quality aspect is the corresponding application domain. This means that software models are used in various domains like web applications or embedded systems having different impacts on the significance of a certain model quality aspect. For example, models of safety-critical embedded systems need to be more *correct* than models of usual web applications.

The preceding discussions show that it is appropriate to set up a specific model quality assurance process for each software project being dependent on the modeling purpose as well as the corresponding modeling domain.

In the next step, static syntax checks for these quality aspects are defined. This is done by formulating questions that should lead to so-called model smells hinting to model parts that might violate a specific model quality aspect. Here, we adopt the goal-question-metrics approach (GQM) that is widely used for defining measurable goals for quality and has been well established in practice (Basili et al. 1994). In our approach, we consider the syntax of the model in order to give answers to these questions. Some of these answers can be based on metrics. Other questions may be better answered by considering specific patterns which can be formulated on the abstract syntax of the model. However, further static analysis techniques could be incorporated to find out additional potential model smells. Furthermore, the project-specific process can (re-)use general metrics and smells as well as special metrics and smells specific for the intended modeling purpose.

Refactoring is the technique of choice for fixing a recognized model smell. A specified smell serves as precondition of at least one model refactoring that can be used

to restructure models in order to improve model quality aspects but appreciably not influence the semantics of the model. In this context, it is also recommended to analyze the specified refactorings whether the application of a certain refactoring may cause the occurrence of a specific model smell.

Since the process of manual model reviews is very time consuming and error prone, several tasks of the proposed project-specific model quality assurance process should be automated as effectively as possible. The following tasks of the process can be automated:

- Support for the implementation of new model metrics, smells, and refactorings using several concrete specification languages.
- Calculation of implemented model metrics, detection of implemented model smells, and application of implemented model refactorings.
- User-friendly support for project-specific configurations of model metrics, smells, and refactorings.
- Generation of model metrics reports.
- Suggestion of suitable refactorings in case of specific smell occurrences.
- Provision of warnings in cases where new model smells come in by applying a certain refactoring.

The following sections present a flexible tool environment for model metrics reports, smell detection, and refactoring for models that is based on the Eclipse Modeling Framework (EMF) (Steinberg et al. 2008).

3 Running example

After giving an overview on our approach for model quality assurance, this section discusses a simple example of this process that serves as running example throughout this paper. After presenting the application of a sample model quality assurance process and the used modeling language, we describe the definition of such a process in detail.

3.1 Application of a project-specific model quality assurance process

In our example, we consider a software project for the development of an accounting system of a vehicle rental company. This company has a headquarter and owns some cars, trucks, and motorbikes which can be rented by customers via a vehicle rental service. A car is specified by its manufacturer, its registration number, its engine power, and the number of provided seats. A truck is specified by its manufacturer, its registration number, its engine power, and its weight. Finally, a motorbike is specified by its manufacturer, its registration number, its engine power, and its cylinder capacity. Each customer has a name and an email address and is related to a consultant being an employee of the company. Furthermore, the company has some subcontractors being specific employees and customers.

We assume that software models are used in the domain analysis phase in order to get an overview on real world entities in the problem domain. The modeling of this

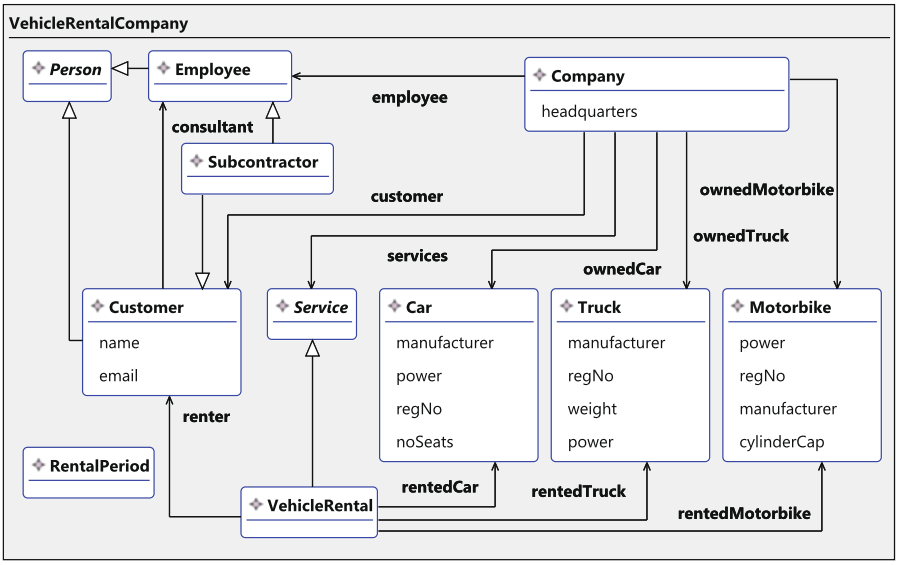

Fig. 3 Example SCM model showing the first version of domain model *Vehicle Rental Company* (before model review)

problem domain is done using SCM, a simple domain specific modeling language (DSML) being simplified UML class models (UML 2012). We discuss this language in more detail in the next section.

Figure 3 shows a first SCM example model that has been developed in an early stage of the problem analysis. Here, the example model is displayed in concrete model syntax using a graphical editor that was generated for SCM by means of the Graphical Modeling Project (GMP 2012).

Concerning quality issues, the model contains several suspicious parts. For example, information on the vehicles (cars, trucks, and motorbikes) are modeled redundantly (like *power*). Furthermore, class `RentalPeriod` is not associated to any other class at all (which hints for some incompleteness). During a model review (see Fig. 1) this initial model is analyzed with respect to project-specific model metrics and model smells. Several refactorings in combination with additional model changes are applied subsequently. Figure 4 shows the improved model after this review. The afore mentioned dependencies have been eliminated and incomplete model parts have been supplemented with further information. We discuss the concrete applied techniques, i.e. detected model smells and applied model refactorings, in Sect. 6.

3.2 Domain specific modeling language *SimpleClassModel* (*SCM*)

In addition to the modeling domain and the intended purpose of modeling, another important choice has to be done before initializing the proposed model quality assurance process: the selection of an appropriate modeling language. In our example, we use a domain specific modeling language (DSML) to be used for modeling in early phases of software development, e.g. for modeling the corresponding problem

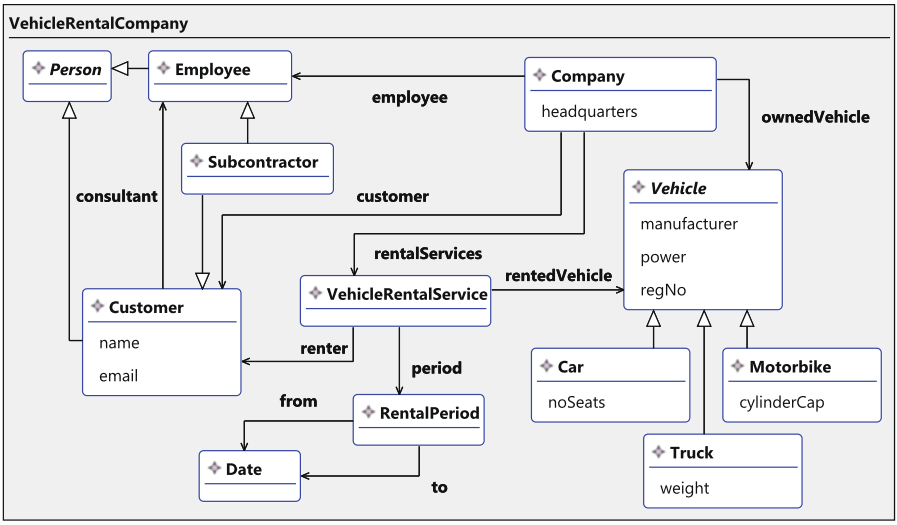

Fig. 4 Improved sample SCM model after model review

domain. The language is called SimpleClassModel (SCM) and is used to model the main static aspects of a given domain. SCM is based on the Essential MOF (MOF 2012) standard and the class diagram part of the UML but it is much simpler since it omits concepts like operations, interfaces, enumerations, and association classes. It is specified using the meta model approach and a set of well-formedness rules using OCL constraints (OCL 2012). Figure 5 shows the SCM meta model specified in Ecore, the meta-meta model of EMF. Please note that in our example we use the SCM language in order to demonstrate the application of our tool set to an arbitrary EMF-based DSML. However, our tool set comes along with a number of implemented model quality assurance techniques for the widely-used and more general languages Ecore and UML (see Sects. 8 and 9).

A `SimpleClassModel` consists of a number of `ScmPackages` where each one consists of a number of packaged elements being `Types` or `Associations`. A `Type` is either a `PrimitiveType`, e.g. *Integer* or *String*, or an `ScmClass`. A (potentially abstract) class can have an arbitrary number of parent classes realized by model element `Generalization`. The derived reference *superclasses* subsumes the total set of ancestor classes of a given class in its inheritance hierarchy. Furthermore, a class can have several (potentially constant) `Attributes`, whereas attributes inherited from ancestor classes are subsumed by a corresponding derived reference. Each attribute has a visibility and an optional type. Additionally, an attribute can redefine another attribute within the inheritance hierarchy of the owning class. Relationships between classes can be modeled using unidirectional `Associations`.

As mentioned above, valid SCM instance models must conform to some additional well-formedness constraints, e.g. unique names of owned elements in a ScmPackage, or acyclic inheritance hierarchies.

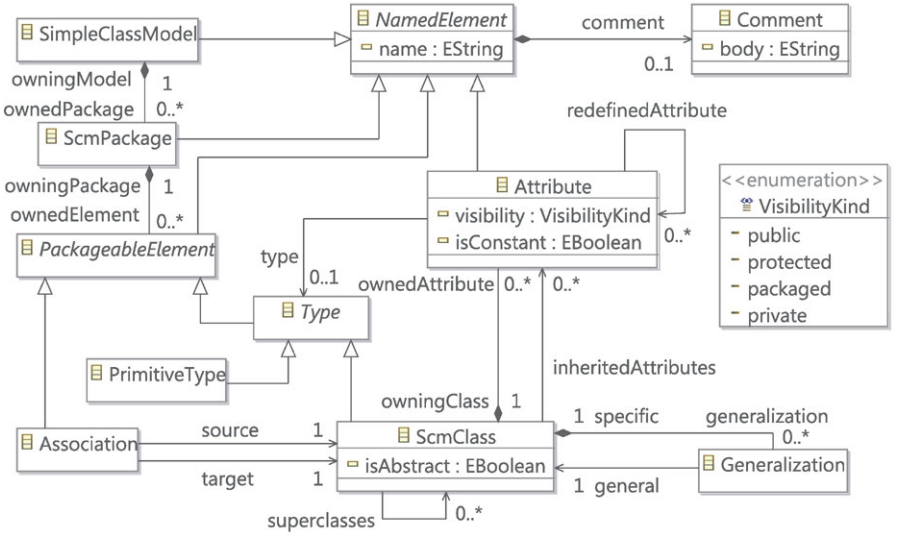

Fig. 5 Meta model of DSML *SimpleClassModel* (*SCM*)

3.3 Specification of project-specific model quality assurance techniques

In this section, we demonstrate how the specification process for the used model quality assurance techniques (see Fig. 2) is applied along our running example. Please note that this process does not need to be applied for each individual project in its full extent. Once these techniques are defined they can be reused in future projects as well.

3.3.1 Specification of relevant model quality aspects

In our example, we use the 6C quality goals described in Sect. 2.1 as quality model and determine those aspects which are most relevant as follows.

The most important property of a domain analysis model is that it models the problem domain in the right way, i.e. choosing the right elements and claiming the right statements. So, 6C goal *Correctness* is an essential quality aspect that has to be considered when applying a model quality assurance process. Since an analysis model is used for communicating with problem domain experts who are typically inexperienced in software modeling, it is also important that the model is easily understandable. This implies that the model must not allow different interpretation results. Furthermore, the analysis model must not have unnecessary information that make it more complex as necessary. So, 6C goals *Comprehensibility*, *Consistency*, and *Confinement* can be seen as essential quality aspects.

Since the modeling purpose in our example is to get an overview on the problem domain, it is rather crucial if less important information is missing. So, 6C goal *Completeness* is a less important quality aspect in our example. Furthermore, since SCM is very simple and manageable, model reviewers do not have to prioritize the quality goal *Changeability*.

Please note that we are arguing from a very specific point of view only to keep the argumentation compact. Of course, the selection of main quality aspects may vary dependent on the intended modeling purpose and demonstrates the complexities and challenges of this basic task.

3.3.2 Formulation of questions leading to static quality checks

After having specified relevant quality aspects we have to think about how to check the compliance with these aspects during the concrete modeling activity. This is done by formulating questions to lead to so-called model smells. In the following, we concentrate on one single quality aspect, namely *Confinement*, and present a selection of appropriate questions. A more detailed discussion on this task and on the complete example can be found on the complementary website of this article (Arendt 2012). Example questions are:

Q1: Are there classes being not used by any other model element? This is a typical case of unnecessarily modeled information.

Q2: Are there classes inheriting from another class several times? This would indicate that the modeler uses the inheritance concept in a too complex way, i.e. the model is more detailed than necessary.

Q3: Are there abstract classes not doing much? Again, this might be an indicator for unnecessary information within the model.

Q4: Are there at least three similar attributes staying together in more than one class? This might be a hint that the modeler does not use the inheritance concept of the SCM language which might be more suitable in this case.

Q5: Are there attributes redefining other ones within the inheritance hierarchy? Since the purpose of the model is to get an overview about the problem domain the use of this language construct might be too complex, i.e. it does not suit to the modeling purpose.

3.3.3 Specification of project-specific SCM smells

The questions formulated in the previous section lead to model smells that hint to model parts that are possibly violating the quality aspect *Confinement*. A structured definition of each smell including a name, the corresponding question, an informal description, an example, affected 6C quality goals, and ways to detect the smell can be found on the complementary website of this article. The deduced SCM smells are:

Unused Class (deduced from question *Q1*): Unused classes often stand alone in the model without any references to other classes. This smell is adapted from Riel who analyzed object-oriented design (Riel 1996) and can be detected by two different mechanisms. First, we can define the absence of child classes, associated classes, and attributes with class type as anti-patterns based on the abstract syntax of SCM and check whether they do not match on a concrete instance class. Second, we can define a constraint that uses three metrics (*Number of direct children*, *Number of associated classes*, and *Number of times the class is externally used*

as attribute type) and that checks whether each metric is evaluated to zero. Nevertheless, the former alternative seems to be the most appropriate one. We discuss this pattern in Sect. 7.

Diamond Inheritance (deduced from question *Q2*): This smell is based on the multiple inheritance concept of SCM. It occurs when the same predecessor is inherited by a class several times and is known in literature as 'diamond' inheritance problem for object-oriented techniques using multiple inheritance and was first discussed by Sakkinen (1989). An adequate mechanism to detect this smell is to specify a corresponding pattern on the abstract syntax of SCM and to find matches in concrete SCM instance models.

Speculative Generality (deduced from question *Q3*): This smell occurs if there is an abstract class inherited by one single class only. It is based on the corresponding code smell introduced by Fowler (1999) and refined by Zhang et al. (2008). To detect this smell we can use metric *Number of direct children* and check whether this metric is evaluated to 1 on an arbitrary SCM class. Of course, the corresponding constraint must check whether this class is abstract. Furthermore, it is possible to specify this smell by a corresponding pattern based on the abstract syntax of SCM and try to match this pattern on classes of a concrete SCM instance model.

Data Clumps (deduced from question *Q4*): A SCM model holds this smell if interrelated data items often occur as 'clump'. More precisely, this smell can be defined as follows:

- At least three attributes stay together in more than one class.
- These attributes should have the same signatures (same names, same types, and same visibility).
- The order of these attributes may vary.

Again, this smell is also based on the corresponding code smell introduced by Fowler (1999) and refined by Zhang et al. (2008). To detect this smell there must be a mechanism to detect similarities in SCM models. This is due to the fact that one can not predict how many attributes are involved in this smell. Furthermore, there might be variants w.r.t. similar attributes when using a more general definition of this smell than here (think of attribute names that need not to be equal but just similar or attributes with different visibilities). Another possibility to detect this smell is to define a metric for an ScmClass counting all equal attributes with other classes. Nevertheless, using a strict definition with exactly three attributes and equal signatures it is possible to define this smell as pattern based on the abstract syntax of SCM.

Redefined Attribute (deduced from question *Q5*): SCM allows for redefining attributes owned by ancestor classes. However, using this language feature could lead to misunderstandings of the modeled aspect and so might be confusing for model readers. It can be checked by matching a corresponding pattern or by evaluating metric *Number of redefined attributes* to zero.

3.3.4 Specification of project-specific SCM refactorings

After having specified appropriate model smells as done in the previous section, suitable refactorings have to be defined in order to support the handling of 'smelly' models. Table 1 gives an overview on refactoring alternatives to eliminate the SCM smells

Table 1 Suitable SCM refactorings to erase specific SCM model smells

	Unused Class	Diamond Inheritance	Speculative Generality	Data Clumps	Redefined Attribute
Extract Class				×	
Extract Superclass				×	
Extract Intermediate Superclass				×	
Extract Subclass			×		
Remove Superclass		×	×		
Remove Intermediate Superclass		×	×		
Remove Redefined Attribute					×
Remove Unused Class	×				

presented above. An entry × in cell (i, j) indicates that SCM refactoring i can be used to eliminate smell j.

To eliminate SCM smell *Unused Class* suitable refactorings can hardly be deduced since one can not determine whether this class is either useless or if there are some missing relationships. So, this smell can either be eliminated by removing the class (i.e. by using the simple refactoring *Remove Unused Class*) or by adding further information to the model not indicated as refactorings.

Smell *Diamond Inheritance* can be eliminated by applying refactorings *Remove Superclass* or *Remove Intermediate Superclass*. Both refactorings can also be used to eliminate SCM smell *Speculative Generality*. Here, the unnecessarily modeled abstract class has to be removed by one of those refactorings, depending on whether this class has a parent class or not. A further applicable refactoring addresses missing information, more precisely missing subclasses of the abstract class. This refactoring is called *Extract Subclass*. It creates a new subclass and applies refactoring *Push Down Attribute* to a set of attributes of the contextual class (which is empty in our case).

The elimination of smell *Data Clumps* can be done in two different ways, either by moving corresponding attributes to a new associated class or by moving them to a new class that is a common superclass of the owning classes. The first option uses SCM refactoring *Extract Class* that internally uses refactorings *Create Associated Class* and *Move Attribute*. The second alternative uses either refactoring *Extract Superclass* or *Extract Intermediate Superclass* if the owning classes have a common superclass already. Besides the creation of an empty (intermediate) superclass, both refactorings use refactoring *Pull Up Attribute* to move equal attributes to this newly created class.

Last but not least, SCM smell *Redefined Attribute* can be eliminated using refactoring *Remove Redefined Attribute* that removes the redefinition relationship as well as the contextual attribute if and only if the redefined attribute is visible to the owning class of the redefining attribute.

On the complementary website of this article, you find a structured definition of each SCM refactoring including a name, a short description, an illustrating example, the contextual meta model element for applying the refactoring, and the input parameters. Furthermore, we use a three-part specification preparing the implementation of

Table 2 Possible impacts of SCM refactorings on SCM model smells

	Unused Class	Diamond Inheritance	Speculative Generality	Data Clumps	Redefined Attribute
Extract Class				×	
Extract Superclass		⊗		×	
Extract Intermediate Superclass				×	
Extract Subclass	⊗		⊗	×	
Remove Superclass				⊗	
Remove Intermediate Superclass				⊗	
Remove Redefined Attribute	⊗				
Remove Unused Class					

refactorings in Eclipse using the Language Toolkit (LTK) technology (Frenzel 2006). The parts of a refactoring specification reflect a primary application check for a selected refactoring without input parameters, a second one with parameters, and the proper refactoring execution steps. Please note that some of the SCM refactorings are adapted from corresponding UML refactorings, for example discussed in Thongmak and Muenchaisri (2004), Zhang et al. (2005), and Markovic and Baar (2008).

As last topic in this section we discuss relations between SCM refactoring and SCM model smells. Inter-relations are presented in Table 2. An entry in cell (i, j) indicates that SCM refactoring i can cause the occurrence of SCM smell j.

Each *Extract ... Class* refactoring may cause SCM smell *Data Clumps* if appropriate attributes are moved to the newly created class. Please note that this smell already existed before the refactoring but in another context (without the newly inserted class). We mark this kind of smell with × whereas completely new smell occurrences are marked with ⊗. Furthermore, smell *Data Clumps* can also be introduced by refactorings *Remove Superclass* and *Remove Intermediate Superclass* when moved attributes complete an equivalent set of attributes in some subclasses.

The application of refactoring *Extract Superclass* can introduce smell *Diamond Inheritance* if the contextual classes have a common subclass. Refactoring *Extract Subclass* can lead to an unused class if no attribute is pushed down to the new class. Furthermore, if this refactoring is applied on an abstract class not inherited so far, SCM smell *Speculative Generality* is introduced. Refactoring *Remove Redefined Attribute* can lead to an unused class if the type class of the removed attribute has been the only use of this class. Finally, refactoring *Remove Unused Class* does not cause any smell from the analyzed list.

4 Tool environment: general approach

In this section, we present the general concepts of our tool environment for quality assurance of EMF-based models. After giving an overview on the state-of-the-art of model quality assurance tooling, we discuss the requirements on our tool set which are deduced from this survey and from the model quality assurance process presented in Sect. 2.2.

4.1 State-of-the-art: tool support for model quality assurance

The existing tool support for model quality assurance is mainly aiming at UML and EMF modeling.

4.1.1 UML modeling

Considering UML modeling, quality assurance tools are integrated in standard UML CASE tools to a certain extent. In the following, we give a rough overview on existing UML model quality assurance tools: In UML CASE tools such as the IBM Rational Software Architect (RSA 2012) and MagicDraw (MD 2012), a number of metrics and validation rules are predefined and can be configured in metrics and validation suites. MD supports class model metrics (e.g. measuring the number of classes, inheritance tree depth, and coupling), so-called system metrics such as Halstead and McCabe, and requirements metrics based on function points and use cases. Validation rules comprise completeness and correctness constraints such as all essential information fields are filled, properties have types specified, etc. Further validation rules can be specified using Java or a restricted form of OCL. RSA also supports predefined metrics. In addition, models can be checked against validation rules being based on metrics. A tool dedicated to the calculation of model metrics is SDMetrics (SDM 2012). SDMetrics analyzes the structural properties of UML models and uses object-oriented measures as well as design rule checking to automatically detect design and style problems in UML models. Measurement data is displayed in different views (e.g., tables, histograms, and kiviat diagrams) and can be exported in various formats like HTML and XML. Furthermore, SDMetrics supports custom definitions of UML metrics and design rules using XML-based configuration files.

Considering UML model refactoring, there is no mature tool support available yet. However, some research prototypes for model refactoring are discussed in the literature, e.g. in Porres (2003), Boger et al. (2003), Markovic and Baar (2008). Most of them are no longer maintained. For example, Porres (2003) describes the execution of UML model refactorings as sequence of transformation rules and guarded actions. He presents an execution algorithm for these transformation rules and constructed an experimental, meta-model driven refactoring tool that uses SMW, a scripting language based on Python, for specifying the UML model refactorings.

To summarize, UML CASE tools and further model analysis tools for UML provide model analyses by predefined metrics and validation rules and support the custom configuration of metrics and validation suites as well as the definition of further custom techniques but do not offer an integrated, custom configured quality assurance environment for UML models based on metrics, smells (validations), and refactorings.

4.1.2 EMF modeling

Since EMF has evolved to a well-known and widely used modeling technology, it is worthwhile to provide model quality assurance tools for this technology. To the best of our knowledge, explicit tool support for metrics calculation on EMF-based models

is not yet available. However, there is the EMF Model Query Framework (EMF Query 2012) to construct and execute query statements that can be used to compute metrics and to check constraints. These queries have the form of select statements similar to SQL and can also be formulated based on OCL. Specified queries are triggered from the context menu. The configuration of queries in suites as well as reports on query results in various forms are not provided. The EMF Validation Framework (EMF Validation 2012) supports the construction and assurance of well-formedness constraints for EMF models. Two modes are distinguished: batch and live. While batch validations are explicitly triggered by the client, live validations listen to change notifications to model objects to immediately check that the change does not violate any well-formedness constraint.

The Epsilon language family (Epsilon 2012) provides the Epsilon Validation Language (EVL) to validate EMF-based models with respect to constraints that are, in their simplest form, quite similar to OCL constraints. Furthermore, EVL supports dependencies between constraints, customizable error messages to be displayed to the user and the specification of fixes to be invoked by the user to repair inconsistencies. For reporting purposes, EVL supports a specific validation view reporting the identified inconsistencies in a textual way. Suitable quick fixes are formulated in the Epsilon Object Language (EOL) being the core language of Epsilon and therefore not specifically dedicated to model refactoring. Here, Epsilon provides the Epsilon Wizard Language (EWL) (Kolovos et al. 2007), a textual domain-specific language for in-place transformations of EMF. We compare our first refactoring prototype with EWL in Arendt et al. (2009). The comparison shows that refactoring EMF-based models using EWL has some strengths but also weaknesses. Refactoring specifications in EWL are very compact, each refactoring is triggered from within the context menu of a contextual model element, and redo/undo functionality is supported. Nevertheless, EWL does not follow the homogeneous refactoring execution structure used in Eclipse. For example, a refactoring is provided only if all preconditions hold (i.e., no meaningful error message is provided), and a preview of the results of a refactoring is missing. Furthermore, EWL does not support reuse of existing refactoring specifications. Finally, there are no predefined EVL inconsistency checks and EWL refactorings (for more general languages like Ecore and UML2, for example) as well as no support for custom configurations of validation suites.

Another approach for EMF model refactoring is presented in Reimann et al. (2010), Refactory (2012). Here, the authors propose the definition of EMF-based refactoring in a generic way, however do not consider the comprehensive specification of preconditions. Our experiences in refactoring specification show that it is mainly the preconditions that cannot be defined generically. (See Arendt et al. (2010b) for a more complex refactoring with elaborated precondition checks.) Furthermore, there are no attempts to analyze EMF models w.r.t. model smell detection.

Finally, the MoDisco framework (Barbier et al. 2010) provides a model-driven reverse engineering process for legacy systems in order to document, maintain, improve, or migrate them. Here, several specific models are deduced (for example, Java models are deduced from Java code) which can be analyzed in order to detect anti-patterns and then be manually improved, for example by refactorings. As the UML and EMF tooling discussed so far, MoDisco supports the specification and computation of custom metrics and queries on models as well as metrics visualization. The

main difference between MoDisco and our tool suite is the intended purpose (reverse engineering vs. modeling).

Similar as for UML modeling, there is various tool support to perform EMF model analyses and to improve EMF models by refactoring. However, there is not yet a comprehensive tool environment for specifying and applying predefined and custom metrics, smells, and refactorings to EMF models in an integrated way where metrics, smells, and refactorings are tightly inter-related. We are heading towards such a tool environment in the following.

4.2 Requirements on the tool environment for quality assurance in EMF

The analysis of existing model quality assurance tools presented in the previous section and the definition of the proposed model quality assurance process presented in Sect. 2.2 lead to the following requirements on our supporting tool set concerning model metrics, model smells, and model refactorings.

4.2.1 Requirements common to all model quality assurance tools

Generality Each tool should be based on the Eclipse Modeling Framework (EMF), i.e. the corresponding functionality should be provided on any model that is based on EMF since EMF is a well-established format for models.

Reuse The tool environment should reuse existing Eclipse respectively EMF components as far as possible, e.g. EMF Compare (EMF Compare 2012) for refactoring preview and BIRT (BIRT 2012) for metric reporting. Furthermore, quality assurance techniques implemented should be reusable since many of them recur most likely in several projects even if modeling purposes may differ.

Integration It should be possible to integrate QA plugins into EMF-based UML CASE tools like the IBM Rational Software Architect (RSA 2012).

4.2.2 Requirements on the application of specific model quality assurance tools (metrics calculation, smell detection, and refactoring execution)

Configurability The modeler (respectively model reviewer) should be provided with a project-specific configuration of model metrics, smells, and refactorings suites. For model smells being based on metrics it should be possible to specify project-specific thresholds.

Integrated application The corresponding functionality should be triggered from within several views in Eclipse like files in the project explorer (for metrics calculation and smell detection) and model elements in the standard tree-based EMF instance editor (for refactoring execution).

Reporting Calculated metric values and detected model smells should be reported in specific integrated views. Model elements being involved in a specific smell occurrence should be highlighted in the standard tree-based EMF instance editor. Furthermore, it should be possible to export metric results in various formats (e.g., HTML, PDF, and XML).

Refactoring features The application of refactorings should follow the homogeneous refactoring execution structure in Eclipse including a preview of the resulting model. This includes a transactional execution of refactorings. Furthermore, the refactoring tool should provide undo and redo functionality as well as an optional analysis of smell occurrences before and after refactoring application. Finally, smells should be related to refactorings being suitable to erase the smell, and refactorings should be related to smells potentially occurring after applying the refactoring.

Quick-fix mechanism It should be possible to invoke a suitable refactoring from within the context menu of a concrete smell occurrence in the smell results view.

4.2.3 Requirements on specification components for metrics, smells, and refactoring

Flexible specification approaches It should be possible to define custom metrics, smells, and refactorings for arbitrary EMF-based models. Here, the tools should support various concrete specification approaches like OCL, Java, and the EMF model transformation language Henshin (Arendt et al. 2010b; Henshin 2012). Furthermore, a designer should be provided with tool support for composing metrics and refactorings from existing ones.

QA tool code generation The tools should provide a comfortable input mechanism for specification-related information like the meta model, the name, and a description of an arbitrary metric, smell, or refactoring. Afterwards, each tool should generate Java code that can be used by the application component in order to provide the corresponding functionality (metrics calculation, smell detection, and refactoring execution).

5 Tool environment: architecture

This section discusses the architecture of our tool environment for EMF model quality assurance and summarizes the used components. Each tool is based on the *Eclipse Modeling Framework* (Steinberg et al. 2008; EMF 2012), i.e. each tool can be used for arbitrary models whose meta models are instances of EMF Ecore, for example domain-specific languages, common languages like UML2[1] used by Eclipse Papyrus (Papyrus 2012) and the Java EMF model used by JaMoPP (JaMoPP 2012) and MoDisco (Barbier et al. 2010; MoDisco 2012), or even Ecore instance models themselves.

Our tool environment mainly consists of two kinds of modules: For calculating model metrics, detecting smells, and executing refactorings there is an ***application module*** each. Similarly there are three ***specification modules*** for generating metrics, smell, and refactoring plugins containing Java code that can be used by the corresponding application module. For simplicity reasons, we refer to these plugins as ***custom QA plugins*** in the remainder of this section.

[1] In this article, we refer to UML2 being the standard EMF-based representation of UML2, i.e. `org.eclipse.emf.uml2.uml`.

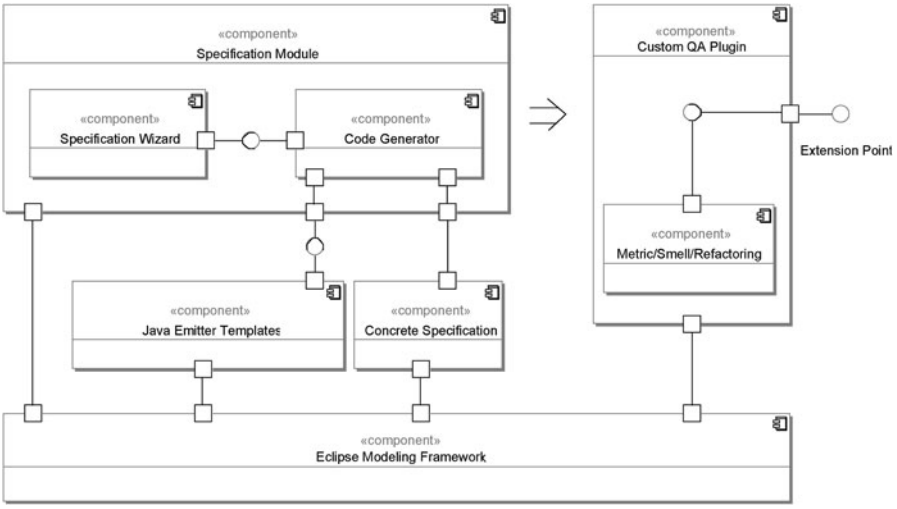

Fig. 6 UML component model of a specification module

Figure 6 shows the architecture of a *specification module* using a UML component model. The specification module provides the generation of *custom QA plugins* containing the metric-, smell-, or refactoring-specific Java code. Using the Eclipse plugin technology, libraries consisting of model quality assurance techniques can be provided. So, already implemented techniques can be reused. Currently, the following specification technologies are supported:

- Java (Java 2012); version 6.
- OCL (OCL 2012) provided by the Eclipse Modeling Project (EMP 2012).
- Henshin (Henshin 2012), a model transformation engine for the Eclipse Modeling Framework based on graph transformation concepts. Henshin uses pattern-based rules that can be structured into nested transformation units with well-defined operational semantics. For further information about Henshin we refer to Arendt et al. (2010b).
- CoMReL (Arendt and Taentzer 2012b), a model-based language for the combination of EMF model refactorings.

More concretely, the following techniques can be used in a *concrete specification* of a new EMF model metric, smell, or refactoring:

1. Model metrics can be concretely specified in Java, as OCL expressions, by Henshin pattern rules, or as a combination of existing metrics using a binary operator.
2. Model smells can be concretely specified in Java, as OCL invariants, by Henshin pattern rules, or as a combination of an existing metric and a comparator like *greater than* ($>$).
3. The three parts of a model refactoring can be concretely specified in Java, as OCL invariants (only precondition checks), in Henshin (pattern rules for precondition checks; transformations for the proper model change), or as a combination of existing refactorings using the CoMReL language.

Table 3 Extension point descriptions for EMF model metrics, smells, and refactorings

Field name	Extension point	
	`org.eclipse.emf.refactor.metrics`	
	Type	Description
metric_name	String	Name of the EMF model metric
id	String	Unique identifier of the EMF model metric
metric_description	String	Description of the EMF model metric (optional)
metric_metamodel	String	Namespace URI of the corresponding meta model
metric_context	String	Name of the context element type
metric_calculate_class	Java	Java class that implements `IMetricCalculator`

Element name	Extension point	
	`org.eclipse.emf.refactor.smells`	
	Type	Description
modelsmell_name	String	Name of the EMF model smell
id	String	Unique identifier of the EMF model smell
modelsmell_description	String	Description of the EMF model smell (optional)
modelsmell_metamodel	String	Namespace URI of the corresponding meta model
metric_finderclass	Java	Java class that implements `IModelSmellFinder`

Field name	Extension point	
	`org.eclipse.emf.refactor.refactorings`	
	Type	Description
menulabel	String	Name of the EMF model refactoring
id	String	Unique identifier of the EMF model refactoring
namespaceUri	String	Namespace URI of the corresponding meta model
controller	Java	Java class that implements `IController`
gui	Java	Java class that implements `IGuiHandler`

The specification module provides wizard-based specification processes (component *Specification Wizard* in Fig. 6). After inserting specific information (like the name of the metric, smell, or refactoring, and the corresponding meta-model) the *code generator* uses the *Java Emitter Templates* framework (JET 2012) to generate the specific Java code required by the corresponding extension point. Table 3 shows the extension point descriptions for EMF model metrics, smells, and refactorings.

Besides basic information like the name, id, or the corresponding meta model of a concrete model quality assurance technique the following interfaces have to be implemented:

`IMetricCalculator` This interface provides the calculation of the corresponding EMF model metric on a given model element. Here, two methods have to be implemented: method `void setContext(List<EObject> context)` for maintaining the model element on which the metric should be cal-

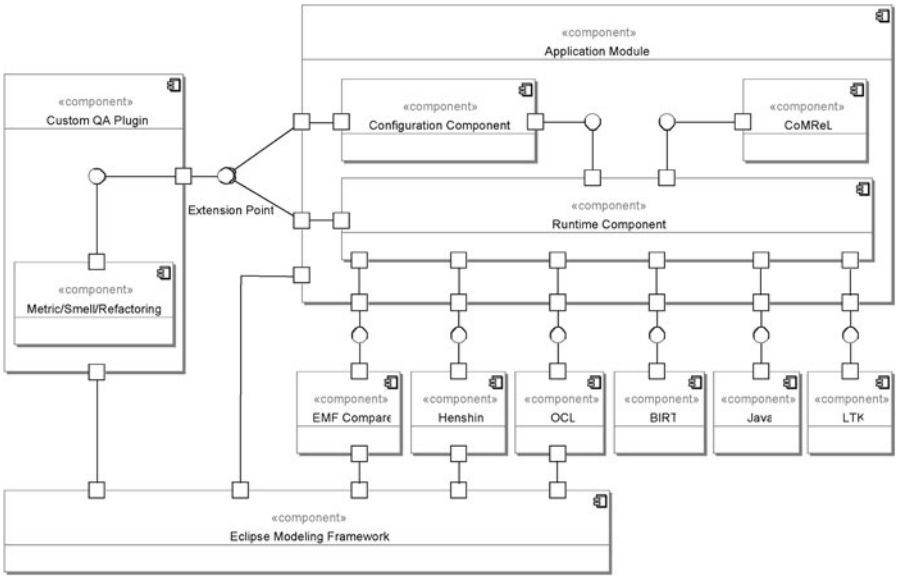

Fig. 7 UML component model of an application module

culated on, and method `double calculate()` for the proper calculation of the metric value on this element.

`IModelSmellFinder` This interface provides the detection of the corresponding model smell in a given EMF model. It has one method which must be implemented by the corresponding Java class: `LinkedList<LinkedList <EObject>> findSmell(EObject root)`. Here, the model is specified by parameter `root`. The method returns a list of detected smell occurrences where such an occurrence is given by a list of model elements which are involved in the detected smell.

`IController` This interface is responsible for executing the corresponding model refactoring. Here, the main method which has to be implemented is `RefactoringProcessor getLtkRefactoringProcessor()` that returns an instance of class `RefactoringProcessor` from the Language Toolkit (LTK) API (Frenzel 2006). Within this class, the refactoring specific preconditions are checked by methods `checkInitialConditions(...)` and `checkFinalConditions(...)` whereas the refactoring is finally executed by method `createChange(...)`.

`IGuiHandler` This interface checks whether the refactoring can be executed on the given context elements (method `boolean showInMenu(List<EObject> selection)`) and the process is started by method `RefactoringWizard show()`. As above, `RefactoringWizard` is a class of the LTK API.

Figure 7 shows the architecture of an *application module*. It uses the Java code of the *custom QA plugins* generated by the corresponding specification module (compare right-hand side of Fig. 6 and left-hand side of Fig. 7) and consists of two components. The *configuration component* maintains project-specific configurations of met-

rics, smells, and refactorings. The *runtime component* is responsible for metrics calculation, smell detection, and refactoring execution. Depending on the concrete specification approach, the runtime component uses the appropriate components *Java*, *OCL*, *Henshin*, or the internal *CoMReL* interpreter. Further languages, especially model transformation languages like EWL (Kolovos et al. 2007), may be integrated by suitable adapters (Gamma et al. 1995). For exporting calculated model metrics, the reporting engine *BIRT* (BIRT 2012) is used. Finally, the Language Toolkit (LTK, Frenzel (2006)) is used for homogeneous refactoring execution and *EMF Compare* (EMF Compare 2012), a tool that provides comparison and merge facility for any kind of EMF models, for refactoring preview.

For manually defining the relationships between model smells and model refactorings, our tool environment uses the Eclipse extension point technology again to provide information about these relationships globally. Therefore, two extension points for the manual definition of relations between model smells and model refactorings are provided. Since our tools identify smells respectively refactorings by distinct identifiers (see Table 3), these extension points require relations from *smell IDs* to a list of *refactoring IDs* (in case of providing suitable refactorings for a given smell) and relations from *refactoring IDs* to a list of *smell IDs* (in case of possible new smells when applying a given refactoring). To serve these extension points in a user-friendly way, we extend the property page of a certain Eclipse plugin project in the workspace by providing graphical user interfaces for (de-)activating appropriate relations.

In the following two sections, we present how to work with both kinds of modules. For simplicity reasons and to relate the application of our tools to the process presented in Sect. 2.2, we first present how to work with the application module and its implemented quality assurance techniques. Thereafter, Sect. 7 presents how to specify new metrics, smells, and refactorings for our example language SCM.

6 Tool environment: application of project-specific model quality assurance techniques in EMF

In this section, we present the application of the model quality assurance techniques defined in Sect. 3.3 on our example model as described in Sect. 3.1 supported by our tool environment for EMF model quality assurance.

6.1 Calculation of project-specific SCM model metrics

For the first overview on a model, a report on project-specific model metrics might be helpful. In Sect. 3.3, several metrics for SCM models have been identified that can be used for detecting corresponding smells. In the following, we do not calculate those smell-related metrics only but also other common metrics to get an overview on interesting model properties.

To calculate relevant metrics only, our tool environment supports a project-specific configuration for the metrics suite. Figure 8 shows the project-specific configuration page for our example project. Here, all existing model metrics for EMF-based models are listed. They are structured with respect to the corresponding meta-model (e.g.,

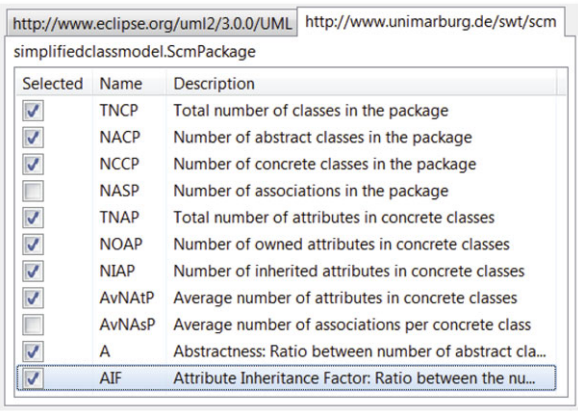

Fig. 8 Project-specific configuration dialog for selected metrics for SCM models

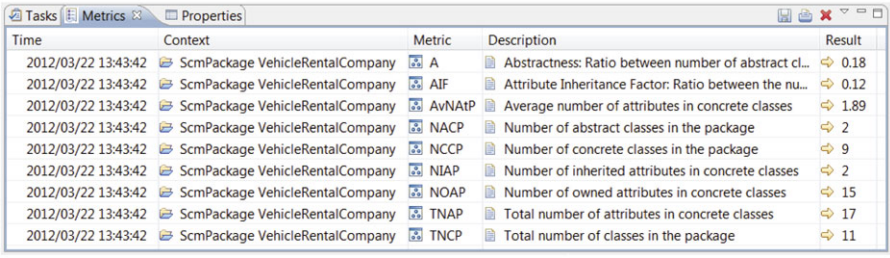

Fig. 9 Reporting of metric values calculated on SCM package `VehicleRentalCompany`

UML and SCM in Fig. 8) and to the corresponding element type the metrics are calculated on. In Fig. 8 for example, we activate model metrics for SCM packages concerning abstractness (`TNCP`, `NACP`, `NCCP`, and `A`) and inheritance issues (`TNAP`, `NOAP`, `NIAP`, `AvNAtP`, and `AIF`).

The metrics tool provides two ways for triggering the calculation of configured model metrics. On the one hand, the calculation of metrics on a specific model element is started from its context menu. On the other hand, a metrics analysis on the entire model (i.e., on each element in the model) is started from the context menu of the corresponding file in the Eclipse project explorer. Figure 9 shows the results of calculating the configured SCM metrics on package `VehicleRentalCompany` (see Fig. 3). The results view shows that there are altogether 11 classes (9 concrete and two abstract classes) in package `VehicleRentalCompany`. The concrete classes own altogether 17 attributes from which 2 are inherited from parent classes (attributes *name* and *email* of class `Subcontractor`).

The first three metrics within the results view in Fig. 9 are calculated using these 'basic' metrics. The abstractness (`A`) of the package is 0.18 (ratio between the number of abstract classes in the package and the total number of classes in the package), the attribute inheritance factor (`AIF`) is 0.12 (ratio between the number of inherited attributes in all concrete classes in the package and the total number of attributes in all concrete classes in the package), and the average number of attributes in concrete classes within the package (`AvNAtP`) is 1.89. As a first evaluation of these metrics

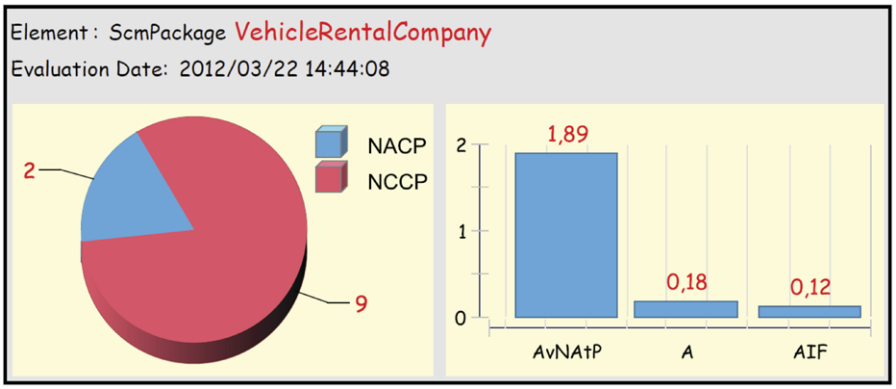

Fig. 10 Excerpt of a generated PDF report concerning calculated metrics results using a pie diagram (*left*) and a bar diagram (*right*)

results, one can state that the model might not be complete since (1) there are only 11 classes modeled for the vehicle company domain, and (2) these classes have less than two attributes on average. Furthermore, language concepts of abstractness and inheritance are not used too exhaustively. So the model is less complex and easier to understand. On the other hand, the low values of `A` and `AIF` can be interpreted as a hint that the modeling purpose is not yet achieved since the modelers use the provided language features insufficiently only.

The metrics tool provides the export of calculated results for reporting purposes. The following output formats are supported: XML (default), HTML, PDF, Postscript, MS DOC, MS PPT, MS XLS, ODP, ODS, and ODT. Furthermore, several output designs are provided but also custom designs can be imported. Figure 10 shows two PDF exports of our example metrics calculation. On the left-hand side, metrics `NACP` (number of abstract classes in the package) and `NCCP` (number of concrete classes in the package) are depicted using a pie diagram. The right-hand side of Fig. 10 shows an exported bar diagram containing the former discussed metrics `AvNAtP`, `A`, and `AIF`.

6.2 Detection of project-specific SCM model smells

The discussion of metrics results shows that a manual interpretation of metric values seems to be unsatisfactory and error-prone. So, another static model analysis technique is required, more precisely an automatic detection of specific smells for SCM models specified in Sect. 3.3.3. As for model metrics, our tool environment provides a configuration of specific model smells that are relevant for the current project. Figure 11 shows the configuration dialog listing all existing smells with respect to their meta-model. For a metric-based model smell, a corresponding threshold can be configured. In our example, SCM smell *Data Clumps* is the only metric-based smell. It relies on metric `NEAC` (number of equal attributes with further classes) and comparator $\geq$ (greater or equal). We set the limit for smell *Data Clumps* to `3`, i.e. this smell occurs if a class owns more than two attributes with same name, type, and visibility in at least one other class.

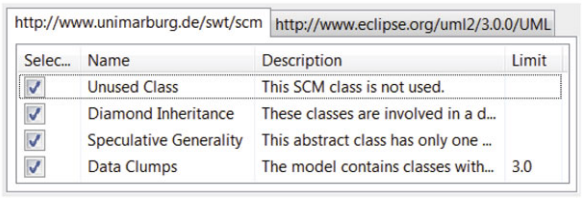

Fig. 11 Project-specific configuration dialog for selected SCM smells

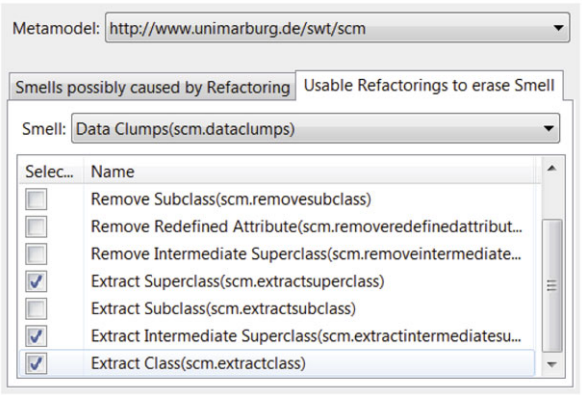

Fig. 12 Manual configuration of SCM refactorings suitable to erase a given SCM smell

Given a concrete model smell occurrence, several refactorings can be suitable to erase it. Our tool environment provides the ability to configure this relationship between model smells and model refactorings. Figure 12 shows the property page for (de-) activating appropriate relations. Here, we address smell *Data Clumps* for SCM models and select altogether three refactorings (namely *Extract Superclass*, *Extract Intermediate Superclass*, and *Extract Class*) as suitable to erase smell *Data Clumps* (according to Table 1).

Similar to the calculation process for model metrics, a smell analysis can be triggered either for the entire model or for a concrete model element. In the latter case, all smells are reported occurring within the containment hierarchy of the selected model element. Nevertheless, it has to be considered that there are model smells which might be distributed along several subtrees (like *Multiple Definition of Classes with equal Names*, looking for equally named classes in different packages). However, our framework provides smell analysis on subtrees only in order to narrow the scope of the analysis, for example on large-scale models.

Analyzing the example SCM model as shown in Fig. 3, the smell detection analysis discovers the existence of altogether six concrete smells which affect quality aspect *Confinement*. The left-hand side of Fig. 13 shows the results of this analysis in a dedicated results view. The report shows that smell *Data Clumps* occurs three times, more concretely in classes `Car`, `Truck`, and `Motorbike`. Smell *Diamond Inheritance* occurs once. Here, the involved elements are classes `Subcontractor` and `Person`. Another detected smell is *Speculative Generality* since abstract class `Service` has one single child class only. Furthermore, there is the unused class `RentalPeriod`.

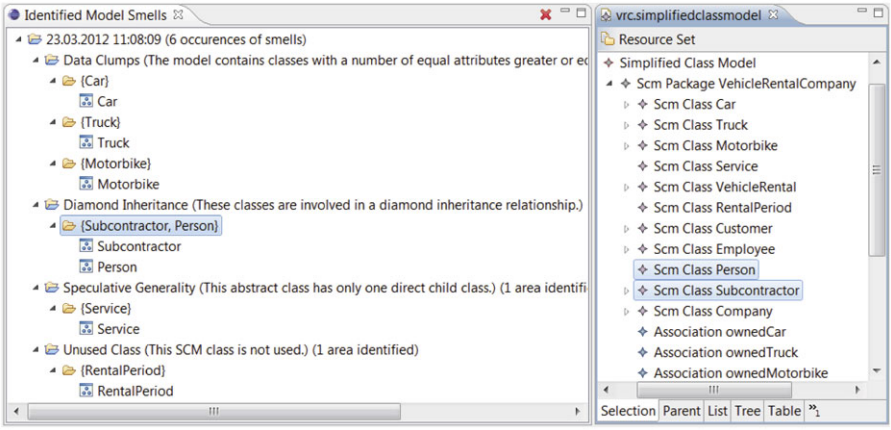

Fig. 13 Report of concrete smell occurrences in our example SCM model (*left*) and highlighting of involved elements in smell `Diamond Inheritance` within the EMF instance editor (*right*)

Concerning concrete smell occurrences, the smell detection tool provides a highlighting mechanism for involved model elements within the standard tree-based EMF instance editor. For example, selecting the occurrence of smell *Diamond Inheritance* in the smell view (compare left-hand side of Fig. 13) highlights classes `Subcontractor` and `Person` in the instance editor as shown in the right-hand side of Fig. 13.

The next step during a model review is to interpret the results of the smell detection analysis. Potential reactions on detected smells are:

- Use refactoring *Extract Superclass* on classes `Car`, `Truck`, and `Motorbike` to insert a common parent class `Vehicle` and pull up attributes *manufacturer*, *power*, and *regNo* to it.
- The diamond inheritance smell detected on class `Subcontractor` should not be eliminated since this seems to be an important detail that has to be addressed in the domain model.
- Smell *Speculative Generality* should be removed by using refactoring *Remove Superclass* on class `Service` since the company does not offer further services.
- Class `RentalPeriod` is unused up to now. It should be associated to class `VehicleRental` and shall refer a new class `Date` twice (named *from* and *to*).

6.3 Application of project-specific SCM model refactorings

Besides manual changes, model refactoring is the technique of choice to eliminate occurring smells. In our tool environment for model quality assurance, this task is provided by the primary functionality of EMF Refactor as presented in Arendt et al. (2010a). Again, this component provides a configuration mechanism to select refactorings being relevant for the given modeling project. The configuration user interface is similar to that of the metrics component (see Fig. 8) and is not shown here.

Since the application of a given refactoring poses a risk for inserting new model smell occurrences, our tool environment supports the manual configuration of this re-

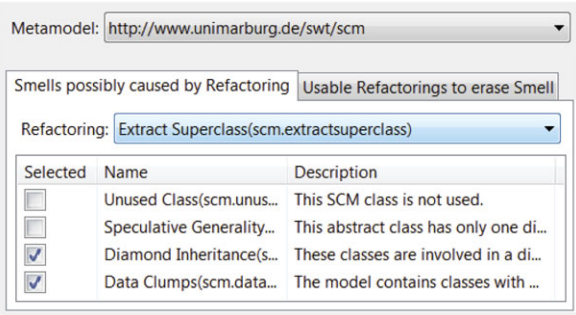

Fig. 14 Manual configuration of potentially new SCM smells after applying an SCM refactoring

lationship between model refactorings and model smells. Figure 14 shows the property page for (de-)activating appropriate relations. For example, in Fig. 14 we specify that the application of SCM refactoring *Extract Superclass* can cause an occurrence of SCM smells *Diamond Inheritance* or *Data Clumps* according to Table 2. Please note that this relationship need not be set for each project. Instead, our tool set uses again the Eclipse extension point technology to provide information about smell-refactoring relationships throughout the entire Eclipse system.

The application of a certain model refactoring can be triggered by using two alternative ways: First, it can be invoked from within the context menu of at least one model element in the standard tree-based EMF instance editor. Dependent on the selected element(s), only those refactorings are provided in the menu being defined for the corresponding model element type(s).

The second way to trigger a model refactoring is to use the quick fix mechanism of the smell results view as shown on the left-hand side of Fig. 13. Starting from this view, our tool environment provides a suggestion for potential refactorings according to pre-defined smell-refactoring relations (see Fig. 12) and a dynamic analysis of applicable model refactorings.

The suggestion dialog is started from within the context menu of a smell occurrence (e.g., occurrence {*Car*}, see Fig. 13) and consists of two tabs. The first tab (see top of Fig. 15) suggests all model refactorings that have been manually defined as being suitable to erase the corresponding model smell. Furthermore, the dialog informs about possible new smells potentially inserted when applying the refactoring (according to the manual configuration in Fig. 14). The second tab (see bottom of Fig. 15) lists all those model refactorings which have been proven to be applicable on at least one model element in the selected smell occurrence. Please note that this does not necessarily mean that each presented refactoring would improve the model quality by erasing a model smell. It simply means that the target model structure allows the application of that refactoring. Again, the dialog informs about potentially inserted smells according to the manual configuration.

After invoking a refactoring, either from within the EMF instance editor or by the provided quick fix mechanism, refactoring-specific basic conditions are checked (initial precondition check). Then, the user has to set all needed parameters. Figure 16 shows the parameter input dialog for refactoring *Extract Superclass* that is invoked on classes `Car`, `Truck`, and `Motorbike`. Here, the name for the new parent class of `Car`, `Truck`, and `Motorbike` has to be specified.

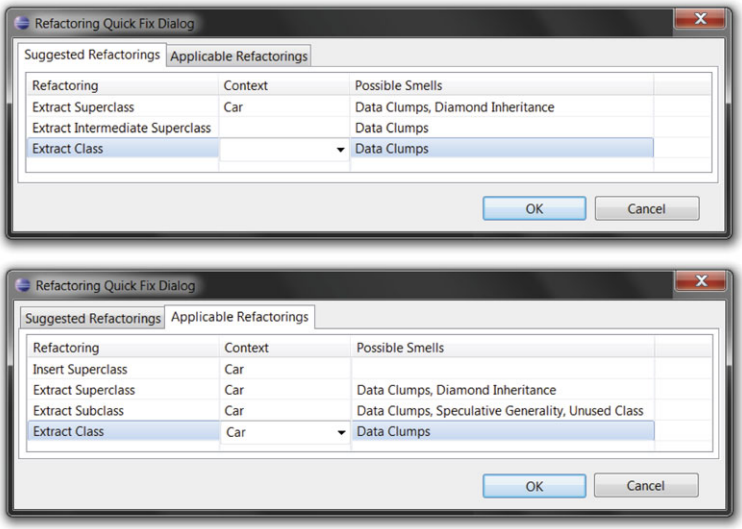

Fig. 15 Quick fix mechanism: manually defined refactorings (*top*) and actually applicable refactorings (*bottom*)

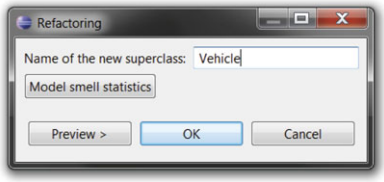

Fig. 16 Parameter input dialog of SCM refactoring `Extract Superclass`

Then, EMF Refactor checks whether the user input does not violate further conditions (final pre-condition check). In case of erroneous parameter input a detailed error message is shown. If the final check has passed, a preview of model changes to be performed by the refactoring is provided using EMF Compare (EMF Compare 2012). Figure 17 shows the resulting EMF Compare dialog using a tree-based model view. The left-hand side shows the original example model (see Fig. 3) whereas the right-hand side presents the refactored model. Model changes are highlighted by colored connections. Here, the right-hand side shows the newly created class `Vehicle` owning attributes *manufacturer*, *power*, and *regNo* being pulled up from classes `Car`, `Truck`, and `Motorbike`. These classes have a new generalization relationship to class `Vehicle` each now.

Besides the model change preview, our tool environment provides the opportunity to get a quantitative analysis on changes of model smell occurrences. In contrast to the manual configuration of potential refactoring-smell-relations, this preview provides a concrete overview on smell occurrence changes when applying the refactoring. For a detailed discussion of model refactoring-smell-relations respectively their implementation in our tool environment we refer to Arendt and Taentzer (2012a).

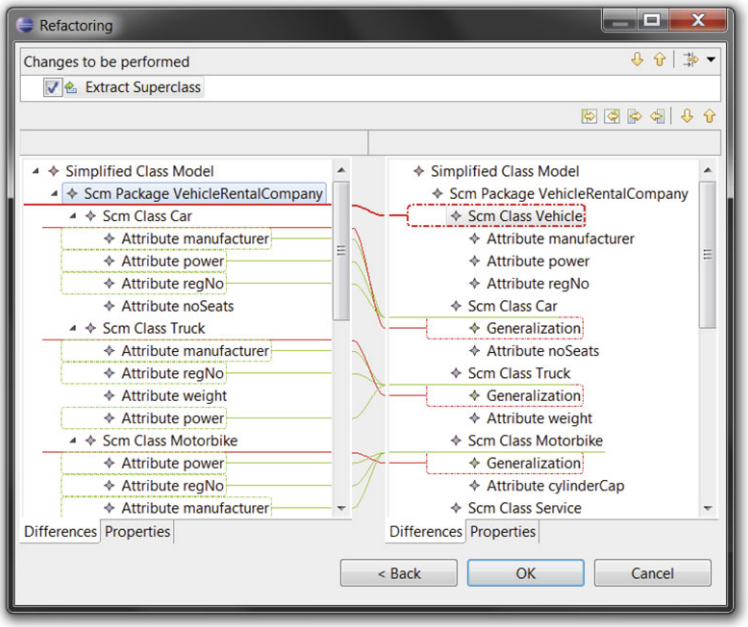

Fig. 17 Preview dialog during the application of SCM refactoring `Extract Superclass`

Fig. 18 Smell analysis during the application of SCM refactoring `Extract Superclass`

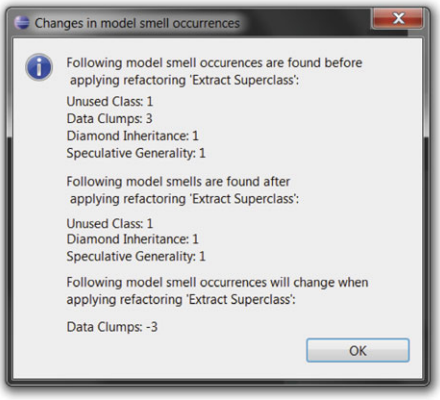

Figure 18 shows the information dialog when applying SCM refactoring *Extract Superclass* on classes `Car`, `Truck`, and `Motorbike` in our example model. Before the refactoring, SCM smell *Data Clumps* occurs three times; after the refactoring these occurrences would be eliminated and no further smell would be inserted. However, smells *Unused Class*, *Diamond Inheritance*, and *Speculative Generality* would remain. Last but not least, all model changes can be committed and the refactoring can take place.

Figure 19 shows our example SCM model after performing several model changes, being refactorings and manual changes, as described at the end of Sect. 6.2. Now,

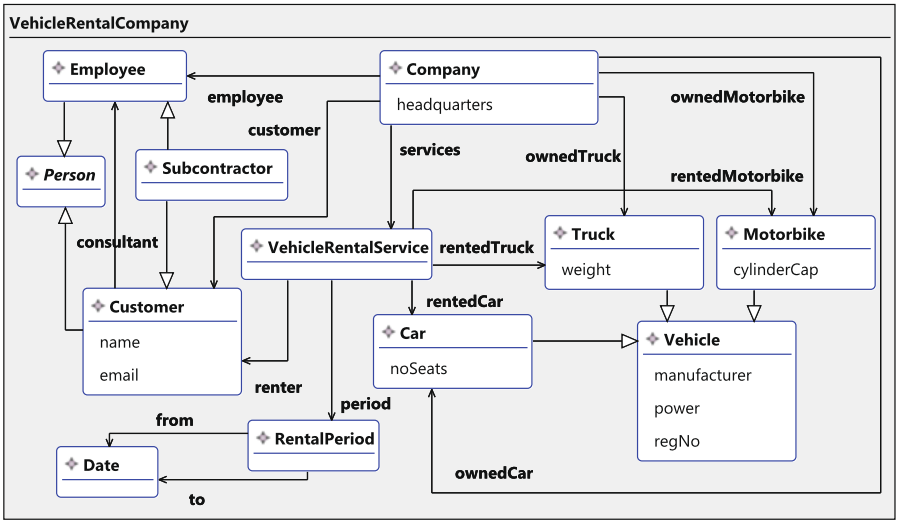

Fig. 19 Example SCM model after several model changes during a first model review

classes `Car`, `Truck`, and `Motorbike` have a common superclass `Vehicle` owning the afore redundant attributes *manufacturer*, *power*, and *regNo*. Class `Service` has been removed so that `VehicleRentalService` is the only offered service left. Finally, class `RentalPeriod` has been completed by additional information, i.e. class `Date` and associations *period*, *from*, and *to*.

From the detected smell occurrences only one is left (smell *Diamond Inheritance* in class hierarchy `Subcontractor` $\Rightarrow$ `Person`). Nevertheless, there are model parts remaining suspicious with respect to several model quality aspects. For example, there are two elements indicating incorrect modeling. First, class `Vehicle` is concrete even though it should represent a generic term for concrete vehicle kinds, hence should be abstract. Moreover, the association between classes `Company` and `VehicleRentalService` has a too general name and should be named *vehicleRentalService* instead. Furthermore, there are associations from class `Company` to classes `Car`, `Truck` and `Motorbike` respectively from class `VehicleRentalService` to these classes hinting to some kind of redundant modeling.

The former discussion shows that project-specific model quality assurance techniques need not be completely defined before a project starts. In our example, the quality assurance process should be adapted during the model development phase in order to be steadily improved. SCM smells *Concrete Superclass* and *Association Clumps* as well as SCM refactorings *Rename Association* and *Pull Up Association* would extend the suite of project-specific model quality assurance techniques in a meaningful way.

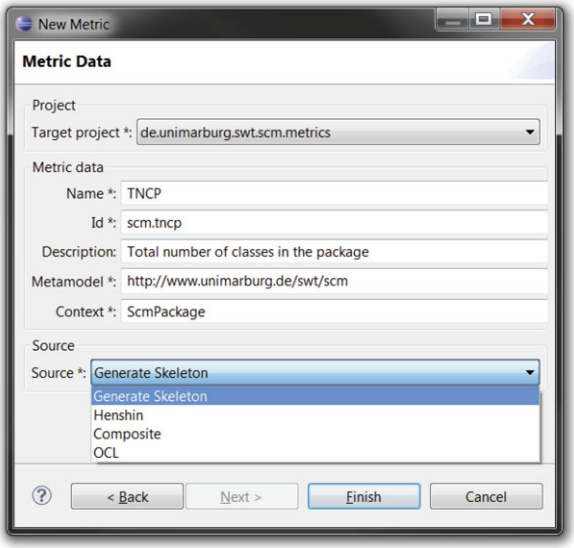

Fig. 20 Wizard dialog for the specification of metrics for EMF-based models

7 Tool environment: specification of project-specific model quality assurance techniques in EMF

Our tool environment for EMF model quality assurance provides a wizard-based specification process for each supported quality assurance technique model metrics, model smells, and model refactorings. In this section, we present several supported concrete specification mechanisms for model quality assurance techniques discussed along the SCM example.

7.1 Specification of project-specific model metrics

For the specification of model metrics, our tool environment currently supports four concrete techniques. As basic approaches, pure Java code using the modeling language API generated by EMF and OCL expressions can be used. Another approach is to define a pattern using the abstract model syntax first and to count its occurrences in a concrete model thereafter. These patterns are formulated as rules in a language included in the EMF model transformation tool Henshin (Arendt et al. 2010b; Henshin 2012). To define compositional metrics, our tool environment supports a combination of existing ones. Here, the involved metrics as well as appropriate arithmetic operations have to be specified.

Figure 20 shows an example wizard dialog concerning the specification of SCM metric `TNCP` (total number of classes in a package). After inserting metric-specific information like the name or the corresponding meta-model, our tool environment generates Java code to be completed by the actual metrics calculation. After this completion, we obtain a module with all metrics features as described in Sect. 6.1. The list of supported model metrics is extended using the extension point technology of Eclipse.

```
ScmPackage p = (ScmPackage) context;
int numberOfClasses = 0;
for (PackageableElement pe : p.getOwnedElement())
 if (pe instanceof ScmClass) numberOfClasses++;
return numberOfClasses;
```

Listing 1 Java specification for SCM metric TNCP (total number of classes in a package)

```
context ScmPackage
def: getNACP(): Integer = self.ownedElement
 ->select(e.oclAsType(ScmClass)|e.oclIsTypeOf(ScmClass))
 ->select(cl|cl.isAbstract = true) -> size()
```

Listing 2 OCL specification for SCM metric NACP (number of abstract classes in a package)

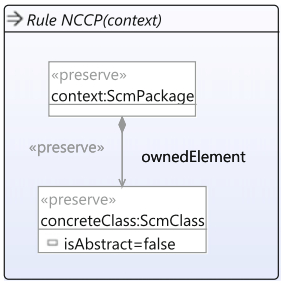

Fig. 21 Henshin pattern rule specifying metric NCCP

Listing 1 shows a Java code snippet specifying SCM metric TNCP. Starting from the contextual package element, the Java API of SCM generated by EMF is used. According to the SCM meta-model in Fig. 5, a package owns several elements, either of type Association, PrimitiveType, or ScmClass. The code in Listing 1 filters the owned elements and returns the number of occurrences of type ScmClass. This code snippet can be inserted into the generated Java code when selecting specification mode *Generate Skeleton* (see Fig. 20).

Since EMF models can be queried well using the Object Constraints Language (OCL 2012), our tool environment supports model metrics specifications formulated as OCL queries. Listing 2 shows an OCL specification of SCM metric NACP (number of abstract classes in a package). Similar to the Java specification of metric TNCP in Listing 1, the owned elements of the contextual package are filtered on type ScmClass. Then, only abstract classes are selected and their number is returned. To insert the OCL query, the specification wizard provides a dedicated page after selecting this specification mode.

Figure 21 shows a Henshin pattern rule specifying SCM metric NCCP (number of concrete classes in a package) using the graphical syntax of Henshin. The upper node *context* of type ScmPackage represents the contextual model element for calculating metric NCCP whereas the remaining rule elements represent the pattern that has to be found in the model. Here, this pattern defines an owned element of the package with type ScmClass whose meta-attribute *isAbstract* has value *false* (i.e., the class

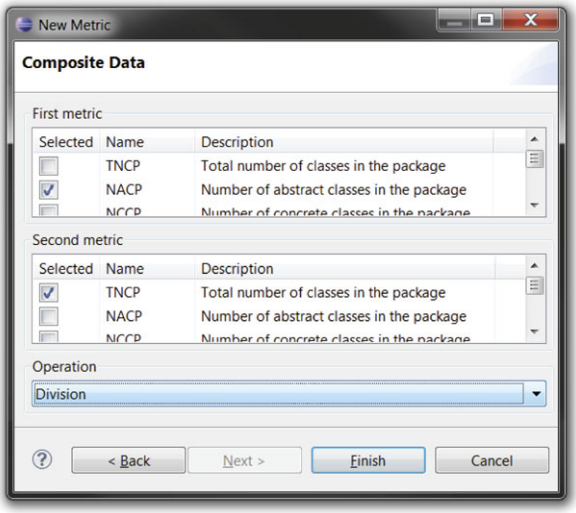

Fig. 22 Compositional specification for SCM metric A

is concrete). To calculate metric NCCP, our metrics tool uses the Henshin interpreter to find and count matches of this pattern rule on concrete SCM instance models.

For defining compositional metrics, the specification wizard provides a dedicated page after selecting specification mode *Composite*. Here, the metric designer simply has to select the involved existing metrics as well as the appropriate arithmetic operation. Currently, binary arithmetic operations sum, subtraction, multiplication, and division are supported. For specifying e.g. SCM metric A (Abstractness), metrics NACP (number of abstract classes in a package) and TNCP (total number of classes in a package) are combined using the binary arithmetic operation division (see Fig. 22). Of course, only those metrics are presented whose contextual elements correspond to the contextual element of the new compositional metric (ScmPackage in our example).

7.2 Specification of project-specific model smells

Our tool environment currently supports four concrete mechanisms for model smell specification. Again, pure Java code and OCL expressions can be used as basic approaches. Some smells can be detected well by metric benchmarks. Here, appropriate model metrics are used together with suitable benchmarks being set by project-specific configurations. Pattern-based smells (i.e., smells that are detectable by the existence of specific anti-patterns) can be specified by Henshin rules. The specification process for model smells is similar to that for metrics specification as shown in Fig. 20. After inserting smell-specific information like the name or the corresponding meta-model, our tool environment generates Java code to be completed. Again, the list of supported model smells is extended using the extension point technology of Eclipse.

Listing 3 shows the core Java specification of SCM smell Speculative Generality. According to Sect. 3.3.3, this smell occurs if there is an abstract class

```
List<ScmClass> classes = getAllClasses(scmModel);
for (ScmClass cl : classes)
 if (cl.isAbstract() && getChildren(cl).size() == 1)
  addToSmellOccurrences(cl);
```

Listing 3 Excerpt of Java specification for SCM smell `Speculative Generality`

```
context ScmClass
def: hasDiamond(): Boolean =
 self.generalization
 -> select(gen1, gen2| gen1.general.superclasses ->
          intersection(gen2.general.superclasses))
 -> notEmpty()
```

Listing 4 Excerpt of OCL specification for SCM smell `Diamond Inheritance`

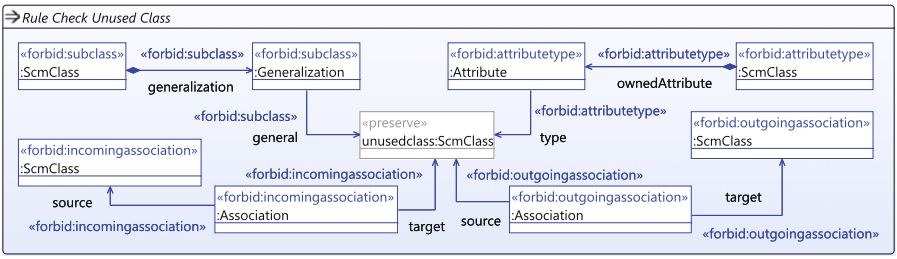

Fig. 23 Henshin pattern rule specification for SCM smell `Unused Class`

inherited by one single class only. The condition in the if-clause exactly checks these features. Please note that the code snippet is not complete since we use auxiliary methods *getAllClasses*(), *getChildren*(), and *addToSmellOccurrences*() that are not discussed in detail here.

The use of OCL is also an adequate approach to specify model smells. Listing 4 shows an excerpt of the OCL specification for SCM smell `Diamond Inheritance`: It defines a boolean operation that checks whether two direct parent classes of the contextual class have at least one common ancestor class.

As described in Sect. 3.3.3, SCM smell *Unused Class* can be detected by matching an appropriate anti-pattern to a concrete SCM class. Figure 23 shows a Henshin pattern rule defining this smell. The pattern specifies an SCM class (tagged by ⟨⟨preserve⟩⟩) and altogether four different application condition patterns that must **not** be found in the model (tagged by ⟨⟨forbid:...⟩⟩).

The first negative application condition (NAC) ⟨⟨*forbid:subclass*⟩⟩ looks for a direct subclass of the contextual class; the second NAC ⟨⟨*forbid:attributetype*⟩⟩ looks for an attribute owned by another class that has the contextual class as type; NAC ⟨⟨*forbid:outgoingassociation*⟩⟩ looks for an outgoing association of the contextual class that is targeted in another class; the last NAC ⟨⟨*forbid:incomingassociation*⟩⟩ looks for an incoming association of the contextual class originating from another class. All NACs have to hold, e.g. if one of the specified relations is found the SCM

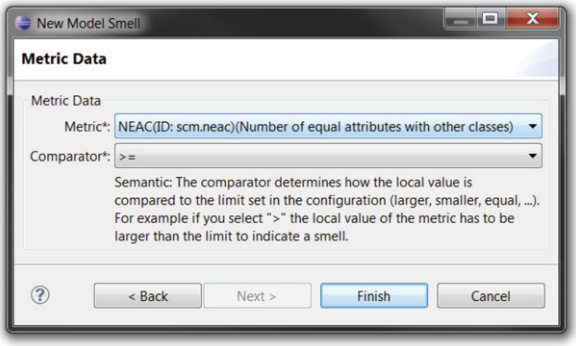

Fig. 24 Specification of SCM smell `Data Clumps` using metric `NEAC`

class is not unused, i.e. the Henshin rule is not applicable on that class. Our smell detection tool uses Henshin's pattern matching algorithm to detect rule matches. The matches found represent the existence of model smells in the model.

For the specification of metric-based model smells, our tool environment provides a dedicated wizard page. The metric designer simply has to select the corresponding metrics as well as the appropriate comparator. For specifying e.g. SCM smell *Data Clumps*, metric `NEAC` (number of equal attributes with further classes) is combined with comparator $\geq$ as discussed in Sects. 3.3.3 and 6.2. Figure 24 shows the corresponding wizard. Please note that the threshold value is not pre-set. This is done in the project-specific configuration as described in Sect. 6.2.

7.3 Specification of project-specific model refactorings

The specification process for model refactorings is started from the context menu of an arbitrary model element. Doing this, several required information like the metamodel and the type of the contextual element is obtained automatically. Its wizard is similar to that for metrics specification as shown in Fig. 20.

Since EMF Refactor uses the LTK technology (Frenzel 2006) as described in Sect. 6.3, a concrete refactoring specification requires up to three parts (i.e., specifications for initial checks, final checks, and the proper model changes). EMF Refactor currently supports four concrete mechanisms for EMF model refactoring specification. As for metrics and smells, refactorings can be specified using Java and OCL. A way to specify a model refactoring straight forwardly is to use Henshin. Here, EMF Refactor uses Henshin's model transformation engine for executing the refactoring as well as Henshin's pattern matching algorithm to detect violated preconditions. Finally, our current work concentrates on a combination of existing refactorings to more complex ones by using a domain-specific language, called CoMReL (Arendt and Taentzer 2012b).

In Sect. 6.3, SCM model refactoring *Extract Superclass* is applied to eliminate smell *Data Clumps*. In the following, we demonstrate the use of each specification approach mentioned above for specifying this refactoring.

Refactoring *Extract Superclass* only makes sense, if each contextual class owns at least one attribute that can be pulled up to a new parent class. The Java specification of this initial precondition check is shown in Listing 5. Here, reference *ownedAttribute*

```
boolean checkEachClassHasAttributes(List<ScmClass> cls) {
 for (ScmClass cl : cls)
  if (cl.getOwnedAttribute().isEmpty())
   return false;
 return true;
}
```

Listing 5 Excerpt of Java specification for the initial precondition check of SCM refactoring `Extract Superclass`

```
def: classWithNameExists(superclassName:String): Boolean =
 ScmClass::allInstances()
 -> select(cl| cl.name = superclassName)
 -> notEmpty()
```

Listing 6 Excerpt of OCL specification for the final precondition check of SCM refactoring `Extract Superclass`

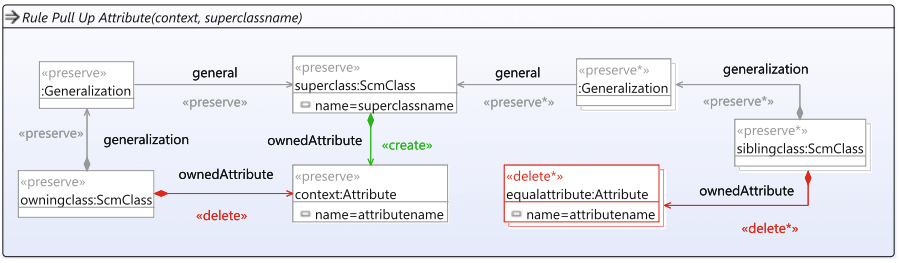

Fig. 25 Henshin amalgamation rule specification for the model change part of SCM refactoring `Pull Up Attribute`

of each contextual class is checked to be empty. If a class does not have any owned attributes, the Java method returns *false* (otherwise *true*).

In the final precondition check, our refactoring tool checks whether the user input does not violate further conditions. SCM refactoring *Extract Superclass* has one parameter *superclassName* that defines the name of the new parent class. One final check is to ensure that there is no class in the model with the same name. Listing 6 shows an OCL query operation that can be used to specify this check.

SCM refactoring *Extract Superclass* internally uses refactoring *Pull Up Attribute* to move equal attributes to the newly created parent class (compare Sect. 3.3.4). The model change part of *Pull Up Attribute* moves the contextual attribute to the specified superclass and removes all equal attributes from their corresponding sibling classes. To specify these changes we can use the amalgamation concept provided by Henshin. This concept contains an interaction scheme consisting of one rule acting as a kernel rule and multiple rules acting as multi-rules. The effect is that the modification defined in the kernel rule is applied exactly once while modifications defined in the multi-rules are applied as often as matches are found.

Figure 25 shows both, the kernel rule as well as a multi-rule of *Pull Up Attribute*, in an integrated way. Here, kernel rule nodes have a single-lined border whereas

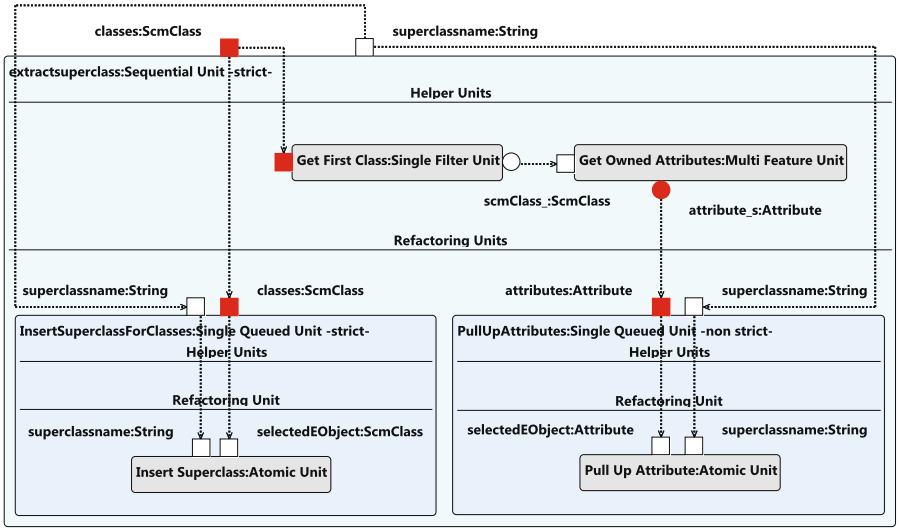

Fig. 26 CoMReL specification for the model change part of composite SCM refactoring `Extract Superclass`

nodes of the multi-rule are double-bordered. In the kernel rule, the contextual attribute is moved to the superclass specified by parameter *superclassName*. The multi-rule deletes the equal attributes from the sibling classes. Equal attributes are determined using an internal parameter *attributeName* being set by the match of the contextual attribute. Please note that this check for equality is sufficient since other checks have been performed before the model change.

Refactorings like *Pull Up Attribute* are rarely applied in isolation. Instead, they are part of refactoring groups needed to perform a larger change in design. For example, refactoring *Extract Superclass* uses simpler refactorings *Insert Superclass* and *Pull Up Attribute*. Therefore, our current work concentrates on a combination of existing refactorings to more complex ones by using a dedicated domain-specific language, called CoMReL, and a tool set for editing and interpreting CoMReL models. Figure 26 shows the CoMReL model specification for the model change part of composite SCM refactoring `Extract Superclass`. Existing refactorings *Insert Superclass* and *Pull Up Attribute* are executed in so-called `QueuedUnits` each representing a looping execution. These units are executed in a sequential order (specified by CoMReL element `SequentialUnit`). Furthermore, there are two auxiliary units that are needed to obtain the owned attributes of the first contextual class. Queued unit *InsertSuperclassForClasses* must be successfully executed for each contextual class (*Insert Superclass* either creates a new superclass for a given class, or a generalization relationship to an existing but empty class). Instead, queued unit *PullUpAttributes* needs to be executed successfully for those attributes of the first contextual class having equal attributes in sibling classes. These facts are specified by unit attributes *strict* respectively *non-strict*. For a more detailed presentation of CoMReL we refer to Arendt and Taentzer (2012b).

Table 4 Number of proof-of-concept implementations of metrics, smells, and refactorings for Ecore, UML2, and SCM models

	Model Metrics	Model Smells	Model Refactorings
Ecore	17	10	22
UML2	106	30	33
SCM	19	11	14

Table 5 Used specification approaches for UML2 metrics, smells, and refactorings

	Java	OCL	Henshin	Combin.	Metric	CoMReL
UML2 Metrics	10	4	52	40	–	–
UML2 Smells	9	3	14	–	4	–
UML2 Refactorings	24	2	11	–	–	16

8 Tool environment: evaluation

In this section, we evaluate our tool environment for EMF model quality assurance along two different perspectives: suitability and performance (resp. scalability). More information about the test design and results can be found on the complementary website of this article.

8.1 Suitability

To evaluate whether our tool environment is suitable to support the techniques of our proposed model quality assurance process as discussed in Sect. 2.2, we implemented a comprehensive catalog of model metrics, smells, and refactorings. In this proof-of-concept implementation, we consider the EMF core meta-model (Ecore), a commonly used meta-model (UML2), and a domain-specific meta-model (SCM). Each metric, smell, and refactoring has been tested extensively. Table 4 summarizes the implemented techniques. As can be seen, the vast majority of implemented QA techniques are for UML2 models. For example, we provide metrics concerning coupling or inheritance issues like *afferent coupling*, *efferent coupling*, and *MaxDIT*. Example UMl2 smells and refactorings are *Multiple Definition of Classes with equal Names* and *Primitive Obsession* respectively *Introduce Parameter Object* and *Remove Isolated State*. Lists of implemented metrics, smells, and refactorings for UML2 models can be found of the complementary website of this article.

Furthermore, we implemented the afore mentioned quality assurance techniques using different specification approaches. Table 5 summarizes the used approaches for concrete specifications of metrics, smells, and refactorings for UML2 models. For comparison purposes, we implemented some refactorings using alternative approaches. Please note that we did not evaluate the suitability of the supported approaches for each technique, i.e. the used approach has been selected freely by the designer.

Java specifications of UML2 metrics use 15.2 LoC (Lines of Code) on average (min. 1 LoC; max. 36 LoC), whereas UML2 smells are implemented in 20.5 LoC on average (min. 13 LoC; max. 74 LoC). Refactoring specifications require 99.7 LoC

on average (min. 8 LoC; max. 269 LoC). Here, about 20 % (20.2 LoC on average) are used for specifying the model change part only, but almost 80 % (79.5 LoC on average) for specifying the initial and final precondition checks. This shows that the complexity of refactoring specifications is particularly hidden in checking the corresponding preconditions.

As a last topic in our proof-of-concept implementation we have related altogether 16 UML2 smells to 18 potentially suitable refactorings and 14 refactorings to 6 potentially occurring smells.

In summary, our implementations show that our tool environment supports metrics calculation, smell detection, and refactoring of EMF-based models to a high extent. Specifications are compact and concentrate purely on the QA technique to be specified. All further functionalities such as metrics reports, etc. are provided by the framework. Furthermore, it is shown that each supported specification approach is suited to specify metrics, smells, and refactorings which can be used by our tool environment. Our experiences in using the various specification approaches show that using Java has been the favorite approach for implementing specifications, especially for implementing refactoring specifications. In fact, this may be due to the preferences of the designer and the progress of supported approaches by the corresponding tool. Independent of the preferred specification language, we feel confident that OCL is particularly suited for specifying metrics which can be directly deduced from the contextual model element using adequate meta attributes respectively references. Henshin transformations have been proven well-suited especially for specifying the model change part of a refactoring. The specification of new metrics, smells, and refactorings is a straightforward task since it is highly supported by comfortable wizards and several concrete specification languages.

8.2 Performance and scalability

To evaluate the scalability of our tool environment, we implemented several performance tests of all three application modules. We performed our tests on a Lenovo ThinkPad W500, Intel Centrino vPro 2.8 GHz, 4 MB RAM.

8.2.1 Metrics calculation

For evaluating the metrics calculation module we calculated a selected set of ten UML2 metrics on model instances with 100, 500, 1 000, 5 000, 10 000, 50 000 and 100 000 elements and measured the time needed for metrics calculation. The selected metrics are:

1. `TNME`—Total number of elements in the model.
2. `MaxDIT`—Maximum of all depths of inheritance trees (context: model).
3. `MaxHAgg`—Maximum of aggregation trees (context: model).
4. `DNH`—Depth in the nesting hierarchy (context: package).
5. `NATIP`—Number of inherited attributes in classes within the package.
6. `NOPIP`—Number of inherited operations in classes within the package.
7. `HAgg`—Length of the longest path to the leaves in the aggregation hierarchy (context: class).

Table 6 Results of the performance tests for calculating 10 UML2 metrics on model instances with 100 to 100 000 elements

Elements	Calculated UML2 metrics	Average time needed
100	42	0.365 sec
500	201	0.563 sec
1 000	399	1.472 sec
5 000	2 008	8.494 sec
10 000	4 016	37.705 sec
50 000	20 068	8 min 36 sec
100 000	40 136	33 min 54 sec

8. `MaxDITC`—Depth of Inheritance Tree (maximum due to multiple inheritance; context: class).
9. `NSUBC2`—Number of all children of the class.
10. `NSUPC2`—Total number of ancestors of the class.

We selected these metrics to cover inheritance and nesting issues. Furthermore, they are calculated on different context types (model, package, and class). We consider UML2 models only due to the variety of implemented metrics and similarities between SCM metrics and those for UML2.

The model instances are created by using a basic model similar to our running example model in Sect. 3 (see Fig. 3) and duplicating respectively nesting the root package. Doing this, we assure that the number of calculated metrics grows nearly linearly compared to the model size. For each case, we repeated the metrics calculation ten times. Table 6 shows the results of these performance tests.

8.2.2 Smell detection

For evaluating the smell detection module we analyzed UML2 models with 100, 500, 1 000, 5 000, 10 000, 50 000 and 100 000 elements with respect to a set of seven selected smells for UML2 models and measured the time needed for smell detection. The selected smells are:

1. `Concrete Superclass`—The model contains an abstract class with a concrete superclass.
2. `Equal Attributes in Sibling Classes`—Each sibling class of the owning class of an attribute contains an equal attribute.
3. `Specialization Aggregation`—The model contains a generalization hierarchy between associations.
4. `Speculative Generality (Abstract Class)`—The model contains an abstract class that is inherited by one single class only.
5. `Speculative Generality (Interface)`—The model contains an interface that is implemented by one single class only.
6. `Unused Class`—The model contains a class that has no child or parent classes, that is not associated to any other classes, and that is not used as attribute or parameter type.
7. `Unused Interface`—The model contains an interface that is not specialized by another interface, and not realized or used by any classes.

Table 7 Results of the performance tests for the detection of 7 UML2 model smells on model instances with 100 to 100 000 elements

Elements	Detected UML2 smells	Average time needed
100	12	0.475 sec
500	60	0.550 sec
1 000	120	0.607 sec
5 000	600	2.834 sec
10 000	1 200	10.716 sec
50 000	6 000	5 min 05 sec
100 000	12 000	20 min 50 sec

We selected the model smells with respect to their influence on quality aspect *confinement*. Smells 1 to 3 use consistent language concepts being more complex than necessary. Smells 4 and 6 are known from corresponding smells for SCM in our running example (see Sect. 3.3.3). Finally, smells 5 and 7 are similar to smells 4 and 6 but consider interfaces instead of classes. Again, we consider UML2 model smells only due to the same reasons mentioned above.

The model instances are constructed in the same way as in the metrics calculation case. For each model size, we repeated the smell detection ten times. Table 7 shows the results of these performance tests.

8.2.3 Refactoring execution

For evaluating the refactoring execution module we applied 7 pretty complex UML2 refactorings on models with a larger refactoring context (e.g., the application of UML2 refactoring *Extract Superclass* on 10 classes having altogether 10 equal attributes and 10 equal operations each) instead of large-scale models. We measured the time in-between committing the refactoring (i.e., after parameter input) and finishing the corresponding model change. Moreover, we repeated each refactoring application ten times to address potential side effects. The maximum time needed to apply a refactoring (without parameter input) has been 236 ms. Table 8 summarizes the results of these performance tests.

8.2.4 Interpretation of results

The results show that the application modules for metrics calculation and smell detection are well-suited for small and mid-sized EMF-based models. For large-scale models, reporting of a high number of calculated metrics (respectively detected smells) is provided in a satisfying time only. However, since static analyses normally do not need to be performed time-critically, this is no crucial limitation of our tool set. Furthermore, the configuration mechanism of our tools can be used even to deal with large-scale models efficiently. For example, the configuration of only a small number of relevant metrics and smells reduces the overall execution time. Moreover, a smell search can be performed on a subtree of the model only, again reducing the overall execution time. Concerning model refactoring, the results show that the refactoring execution module is well-suited for applying refactorings even on large-scale refactoring contexts.

Table 8 Results of the performance tests for the application of 7 UML2 model refactorings on model instances having a larger refactoring context

Refactoring	Context	Min. time	Max. time	Aver. time
Extract Class	Refactoring application on a class having 10 attributes and 10 operations	43 ms	110 ms	66 ms
Extract Subclass	Refactoring application on a class having 10 attributes and 10 operations. The selected class has 10 child classes already. Each child class has 10 attributes and 10 operations	178 ms	236 ms	196 ms
Extract Superclass	Refactoring application on 10 classes having 10 equal attributes and 10 equal operations	91 ms	119 ms	105 ms
Inline Class	Refactoring application on a class having 10 attributes and 10 operations	17 ms	47 ms	34 ms
Introduce Parameter Object	Refactoring application on 9 parameters of an operation with 10 input parameters. The owning class has altogether 10 operations with 10 parameters each. Each operation has parameters equal to the selected ones	85 ms	101 ms	93 ms
Merge States	Refactoring application on a state with 5 incoming transitions. The parameter state has entry, doAction, and exit behaviour. The parameter state has 5 incoming transitions equal to the selected state. The owning region has 20 further states	78 ms	107 ms	88 ms
Remove Superclass	Refactoring application on a class having 10 attributes and 10 operations. The selected class has 10 child classes already. Each child class has 10 attributes and 10 operations	143 ms	231 ms	182 ms

9 Conclusion

In this article, we present a tool environment for model quality assurance based on the Eclipse Modeling Framework (EMF), a common open source technology in model-based software development. It has been designed to support a syntax-oriented model quality assurance process that can be easily adapted to specific needs in model-based projects. This means that dependent on the modeling language and the modeling purpose, specific quality goals, and hence specific metrics, smells, and refactorings may be defined. In such a tailored process, smell detection and model refactoring can be iterated as long as a reasonable model quality has not been reached.

Our tool environment supports the model designer respectively reviewer by obtaining metrics reports, by checking for potential model deficiencies (called model smells) and by systematically restructuring models using refactorings. Automatically proposed refactorings as quick fixes for occurring smells and information on implications of a selected refactoring concerning new model smells widen the provided functionality and support an integrated use of the quality assurance tools.

Model checks and refactorings can be specified by several specification mechanisms. In this paper, we present Java, OCL, and the model transformation language Henshin as possible specification approaches. However, other model transformation approaches such as EWL (Kolovos et al. 2007) are interesting alternatives to be used. In our tool environment, metrics can be composed to more complex metrics and refactorings can be composed by using a dedicated language named CoMReL. It is up to future work to analyze the preconditions of component refactorings w.r.t. to their execution order and to deduce a composite precondition therefrom. A first approach for in-depth composition of refactorings is available for Henshin-specified ones using algebraic graph transformations and critical pair analysis (Ehrig et al. 2006).

As a next step, we plan to evaluate the proposed model quality assurance process in larger case studies using UML models. To do so, we intend to use the UML2 model as language definition and to provide a set of well-known smells and refactorings for class models. A comprehensive catalog of UML metrics, smells and refactorings that have been extracted from literature has already been implemented. A list of implemented techniques can be found on the complementary website of this article (Arendt 2012).

The entire tool set presented belongs to the Eclipse incubation project *EMF Refactor* (EMF Refactor 2012) and is available under the Eclipse public license. Furthermore, we integrated our tool environment into the widely used EMF-based UML CASE tool IBM Rational Software Architect. Here, each version additionally provides a highlighting of model elements for smells in the graphical model view. It is up to future work, to present the preview of refactoring effects also graphically. Both the version for the open source UML tool Eclipse Papyrus and the version for the commercial tool IBM RSA can be installed from the download area of the EMF Refactor homepage. Further information about the integration in Papyrus and RSA can be found at Arendt and Taentzer (2012c).

In future releases, we will continue with making our quality assurance tools still more user-friendly. Besides support for further available QA techniques and further specification languages, performance and scalability shall be further optimized. Here, potential inefficiencies in the framework need to be analyzed and performance-oriented technologies for metric computation and smell detection need to be discussed. Another open issue is how to deal with false positives during model smell detection. These are concrete smell occurrences being actually non-issues to be ignored. Here, we think of using mechanisms like `@SupressWarnings` in Java to indicate areas to be elided during a specific smell search. In the context of EMF, `EAnnotations` might be useful.

We are convinced that performing quality assurance processes is an essential task to obtain software products of high quality. Using the structured model quality assurance process and the corresponding tools presented in this article, model-based and model-driven development can be made more mature yielding software of higher quality.

Acknowledgements This work has been partially funded by Siemens Corporate Technology, Germany. Furthermore, we thank the students Jan Baart, Matthias Burhenne, Gerrit H. Freise, Florian Mantz, Pawel Stepien, and Alexander Weber for their work on our tools. Last but not least, we like to thank the anonymous reviewers for their valuable comments on the previous version of this article.

References

Arendt, T.: A tool environment for quality assurance based on the Eclipse Modeling Framework: additional material. http://www.mathematik.uni-marburg.de/~arendt/mqa/ (2012). Accessed 29 Aug 2012

Arendt, T., Taentzer, G.: Integration of smells and refactorings within the Eclipse Modeling Framework. In: Proceedings of the 5th Workshop on Refactoring Tools Co-Located with ICSE 2012 (2012a). To appear in ACM Digital Library 2012

Arendt, T., Taentzer, G.: Composite refactorings for EMF Models. Technical report. http://www.uni-marburg.de/fb12/forschung/berichte/berichteinformtk (2012b). Accessed 29 Aug 2012

Arendt, T., Taentzer, G.: Besser modellieren: Qualitätssicherung von UML-Modellen. Objektspektrum **06** (2012c). http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/os/2012/06/arendt_taentzer_OS_06_12_lo66.pdf

Arendt, T., Mantz, F., Schneider, L., Taentzer, G.: Model refactoring in Eclipse by LTK, EWL, and EMF refactor: a case study. In: Model-Driven Software Evolution, Workshop Models and Evolution (2009). http://www.modse.fr/modsemccm09/doku.php?id=Proceedings. Accessed 29 Aug 2012

Arendt, T., Mantz, F., Taentzer, G.: EMF refactor: specification and application of model refactorings within the Eclipse Modeling Framework. In: 9th Edition of BENEVOL Workshop (2010a). http://rmod.lille.inria.fr/benevol/pier. Accessed 29 Aug 2012

Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformation. In: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2010. LNCS, pp. 121–135. Springer, Berlin (2010b)

Arendt, T., Kranz, S., Mantz, F., Regnat, N., Taentzer, G.: Towards syntactical model quality assurance in industrial software development: process definition and tool support. In: Software Engineering. LNI, vol. 183, pp. 63–74 (2011). GI

Barbier, G., Brunelière, H., Jouault, F., Lennon, Y., Madiot, F.: MoDisco, a model-driven platform to support real legacy modernization use cases. In: Information Systems Transformation: Architecture-Driven Modernization Case Studies, pp. 365–400. Morgan Kaufmann, San Mateo (2010)

Basili, V., Caldiera, G., Rombach, D.H.: The goal question metric approach. In: Marciniak, J. (ed.) Encyclopedia of Software Engineering. Wiley, New York (1994)

BIRT: BIRT Project. http://www.eclipse.org/birt/ (2012). Accessed 29 Aug 2012

Boger, M., Sturm, T., Fragemann, P.: Refactoring browser for UML. In: Aksit, M., Mezini, M., Unland, R. (eds.) Objects, Components, Architectures, Services, and Applications for a Networked World. LNCS, vol. 2591, pp. 366–377. Springer, Berlin (2003)

Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Berlin (2006)

EMF: Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf/ (2012). Accessed 29 Aug 2012

EMF Compare: EMF Compare Project. http://www.eclipse.org/emf/compare/ (2012). Accessed 29 Aug 2012

EMF Query: EMF Query. http://www.eclipse.org/projects/project.php?id=modeling.emf.query (2012). Accessed 29 Aug 2012

EMF Refactor: EMF Refactor. http://www.eclipse.org/modeling/emft/refactor/ (2012). Accessed 29 Aug 2012

EMF Validation: EMF Validation. http://www.eclipse.org/projects/project.php?id=modeling.emf.validation (2012). Accessed 29 Aug 2012

EMP: Eclipse Modeling Project (EMP). http://www.eclipse.org/modeling/ (2012). Accessed 29 Aug 2012

Epsilon: Epsilon. http://www.eclipse.org/epsilon/ (2012). Accessed 29 Aug 2012

Fowler, M.: Refactoring—Improving the Design of Existing Code. Addison-Wesley, Reading (1999)

Frenzel, L.: The Language Toolkit: an API for automated refactorings in Eclipse-based IDEs. Eclipse-Mag. **5** (2006)

Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Reading (1995)

Genero, M., Piattini, M., Calero, C.: A survey of metrics for UML class diagrams. J. Object Technol. **4**(9), 59–92 (2005)

GMP: Graphical Modeling Project (GMP). http://eclipse.org/modeling/gmp (2012). Accessed 29 Aug 2012

Henshin: EMF Henshin. http://www.eclipse.org/modeling/emft/henshin/ (2012). Accessed 29 Aug 2012

JaMoPP: JaMoPP. http://www.jamopp.org (2012). Accessed 29 Aug 2012

Java: Oracle. Java. http://www.java.com (2012). Accessed 29 Aug 2012
JET: JET. http://www.eclipse.org/modeling/m2t/ (2012). Accessed 29 Aug 2012
Kolovos, D.S., Paige, R.F., Polack, F., Rose, L.M.: Update transformations in the Small with the Epsilon Wizard Language. J. Object Technol. **6**(9), 53–69 (2007)
Lange, C.F.J.: Assessing and improving the quality of modeling: a series of empirical studies about the UML. Ph.D. thesis, Department of Mathematics and Computing Science, Technical University Eindhoven, The Netherlands (2007). Accessed 29 Aug 2012
Markovic, S., Baar, T.: Refactoring OCL annotated UML class diagrams. Softw. Syst. Model. **7**, 25–47 (2008)
MD: No Magic. MagicDraw. http://www.nomagic.com/products/magicdraw.html (2012). Accessed 29 Aug 2012
MoDisco: MoDisco. http://www.eclipse.org/MoDisco/ (2012). Accessed 29 Aug 2012
MOF: Meta Object Facility (MOF). http://www.omg.org/spec/MOF/2.4.1/ (2012). Accessed 29 Aug 2012
Mohagheghi, P., Dehlen, V., Neple, T.: Definitions and approaches to model quality in model-based software development—a review of literature. Inf. Softw. Technol. **51**(12), 1646–1669 (2009)
OCL: Object Constraint Language (OCL). http://www.omg.org/spec/OCL/ (2012). Accessed 29 Aug 2012
Papyrus: Papyrus. http://www.eclipse.org/modeling/mdt/papyrus/ (2012). Accessed 29 Aug 2012
Porres, I.: Model refactorings as rule-based update transformations. In: Stevens, P., Whittle, J., Booch, G. (eds.) Proc. UML 2003: 6th Intern. Conference on the Unified Modeling Language. LNCS, pp. 159–174. Springer, Berlin (2003)
Refactory: Refactory. http://www.modelrefactoring.org/index.php/Refactoring (2012). Accessed 29 Aug 2012
Reimann, J., Seifert, M., Aßmann, U.: Role-based generic model refactoring. In: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2010. LNCS, pp. 78–92. Springer, Berlin (2010)
Riel, A.J.: Object-Oriented Design Heuristics. Addison-Wesley, Boston (1996)
RSA: IBM Rational Software Architect. http://www-01.ibm.com/software/awdtools/swarchitect/ (2012). Accessed 29 Aug 2012
Sakkinen, M.: Disciplined inheritance. In: Cook, S. (ed.) Proceedings of ECOOP'89, pp. 39–56. Cambridge University Press, Nottingham (1989)
SDM: SDMetrics. http://www.sdmetrics.com/ (2012). Accessed 29 Aug 2012
Steinberg, D., Budinsky, F., Patenostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Reading (2008)
Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.-M.: Refactoring UML models. In: Gogolla, M., Kobryn, C. (eds.) Proc. UML 2001—the Unified Modeling Language. Modeling Languages, Concepts, and Tools. LNCS, vol. 2185, pp. 134–148. Springer, Berlin (2001)
Thongmak, M., Muenchaisri, P.: Using UML metamodel to specify patterns of design refactorings. In: Proceedings of the 8th National Computer Science and Engineering Conference (NCSEC 2004) (2004)
UML: Unified Modeling Language (UML). http://www.uml.org (2012). Accessed 29 Aug 2012
Zhang, J., Lin, Y., Gray, J.: Generic and domain-specific model refactoring using a model transformation engine. In: Model-Driven Software Development, pp. 199–217. Springer, Berlin (2005)
Zhang, M., Baddoo, N., Wernick, P., Hall, T.: Improving the precision of Fowler's definitions of bad smells. In: Software Engineering Workshop, Annual IEEE/NASA Goddard, pp. 161–166 (2008)