Achieving UML and OCL Model Quality by Utilizing Metamodeling

Khanh-Hoang Doan

University of Bremen

2020

Achieving UML and OCL Model Quality by Utilizing Metamodeling

by Khanh-Hoang Doan

Dissertation
in partial fulfillment of the requirements for the degree of
Doctor of Engineering
- Dr. Ing. -

Submitted to Fachbereich 3
(Mathematics and Computer Science)
University of Bremen
June 2020

Date of Defense: 09 September 2020

Referee: Prof. Dr. Martin Gogolla
Prof. Dr. Jan Peleska

Abstract

Model-driven engineering (MDE) is a system development methodology, which helps to abstract systems architecture and provides a promising means for addressing architecture complexity and design quality. Models are the backbone of the MDE approach. Therefore, an efficient means of exploring, querying and assuring quality of models play essential roles in a successful project that applies model-based techniques.

In this thesis, we initially propose an approach for extending a two-level modeling style to a three-level one by adding a meta-model at the topmost level. Standard OCL does not support reflective constraints, e.g., constraints concerning properties of the model, such as the depth of inheritance. By adding an auto-generated metamodel instantiation that reflects the model under consideration to the middle level, we can offer an option for writing reflective constraints and queries.

Metric measurement and smell detection are efficient mechanisms for evaluating the internal quality of models. The metrics and design smells employed in this thesis are defined at a metamodel level, and their evaluation is performed in an automatically generated metamodel instantiation. The employed metrics cover both the class scope and the model scope, and software designers can define their own metrics and design smells as well. We also introduce a complete process for model quality assessment with pre-defined metrics. In addition, a metrics configuration defined by an experienced chief designer can be translated into OCL invariants and then evaluated to provide quality feedback to software designers relieving them from detailed OCL expertise.

Another contribution of this thesis is an evaluation of metric measurement and smell detection on a large dataset of Unified Modeling Language (UML) models collected from practice. This evaluation not only illustrates the feasibility and usefulness of our approaches but also helps to answer the research questions regarding the characteristics and quality of models in practice.

Acknowledgments

This thesis is the last milestone is my scientific and professional career, and I would like to thank all of those who have helped and supported me during my PhD studies. First of all, I would like to express my gratitude to my supervisor Prof.Dr. Martin Gogolla to guide me well throughout the research work, for his patience and motivation. I have learned a lot from him not only in doing research but also in daily life. I am also pleased to say thank you to Prof.Dr. Jan Peleska for his willingness to be my co-supervisor and for examining this thesis.

I would also like to thank my colleagues, who have been working at the Database Systems Group, for their support. Discussions and daily conversations with you helped me a lot to get over difficulties in completing this research.

My thanks also go to MOET, DAAD and University of Bremen as the crucial sources for my PhD studies. The support from DAAD, such as providing free German course and policies in administration procedures, also helped me a lot to get over difficulties in the early days in Germany.

In the end, I am grateful to my parents for supporting me spiritually throughout my life in general, and especially for encouraging me to go on with my research. Last but not least, my heartfelt thanks go to my wife and my son. The love and happiness from them have made me strong and able to complete this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Within software development, Model-Driven Engineering (MDE) is playing an increasingly important role. Model-Driven Engineering is a software development paradigm that considers models as central development artifacts, and other software elements, such as code, documentation and test cases, can be obtained from models through model transformations. Furthermore, MDE allows developers to describe systems at multiple levels of abstraction. Consequently, the details of implementation platforms can be disregarded during the modeling phase. Developers therefore only need to focus on verifying and validating the essentials characteristics of the system under consideration. In the early phases of software development, modeling languages such as the Unified Modeling Language (UML) [Obj15b] enriched by the Object Constraint Language (OCL) [CG12] have found their way to become the key elements throughout the development process. A study of the usage level of models and MDE in the software industry has been presented in [Tor+13]. Through a survey of 155 Italian software professionals, the authors conclude that: (1)"*software modeling is a very relevant phenomenon in the Italian industry*" and (2) "*MDE mechanisms are used in the software industry*" [Tor+13].

Therefore, the MDE paradigm presents a challenge of precise formal definitions of modeling language. Metamodeling [AK03; Béz05] is a common method within several methods for the formal definitions of modeling languages. Metamodels play a crucial role in metamodeling as they define the abstract syntax and the structure of models. A model conforms to its metamodel only if all the properties of the model fulfill the rules, constraints, and syntax of its metamodel. Moreover, as a modemodel is, in fact, a model, it might need a meta-meta model as its metamodel. This process can go further up to an

arbitrary number of metamodels. Although a metamodeling mechanism can have more than two instantiation levels, the accessibility to instantiation levels seems to be restricted to one or two lowest levels. The first part of the work in this thesis addresses this issue by extending the meta-object facility (MOF) architecture [Obj15a] to simultaneously offer accessibility to three linguistic instantiation levels. To achieve this objective, we add the full Object Management Group (OMG) UML 2.4 metamodel to the topmost level of a traditional two-level modeling approach. To offer access to the metalevel, a metamodel instantiation that reflects the user model is automatically generated and added to the middle level, (i.e., at the same level as the user model). Metamodels store the information about the model; hence, the usage of metamodels can enhance our understanding of the model and allow us to extract the important properties of the model, such as metrics and quality issues.

As models are the central artifacts, one of the major concerns in MDE is the models quality. Assuring the quality of software artifacts in the early phases of the development process has been widely accepted in software engineering as a good practice. Detecting and fixing issues occurring in the design phase, for example, can significantly prevent faults arising in later stages, (e.g., coding phase). Fixing issues reduces the cost and effort of the development process. A number of studies in the literature[BV10; Bas+16] have indicated that the quality of models has a significant influence on the outcome of software. Metrics can be potential early indicators of the model quality as many well-known and accepted software metrics had been successfully transferred from the code level to the model level. Multiple authors have proposed many sets of software metrics in the literature [CK94; LK94; AM96; BDM97; HCN98; Gen02], and a considerable number of which are applicable to the model level. Basically, these metrics can be used for measuring internal design quality characteristics, (e.g., complexity and coupling). A survey of the theoretical validation and an empirical validation of the UML class model metrics in [GPC05] has confirms the applicability of metrics in practice.

Among several languages and mechanisms that have been proposed for metric measurement, OCL seems to be a potential candidate because of its support for all the requisite mathematical operations for formulating metrics and its availability in modeling tools [Chi11]. In the literature, several works have presented approaches using OCL at the metamodel level for model metrics definition [Bar+02; MP06; Chi11]. These works introduced approaches for metrics formulated in OCL. However, the process of retrieving the value of

metrics defined by OCL on the metamodel remains unclear because the accessibility to both meta-level and user-model level is needed to calculate metrics defined by OCL. In our work, metrics are defined as operations of separated metaclasses and they can be computed thanks to the availability of metamodel accessibility. Additionally, within our three-level modeling architecture, metrics can be utilized instantly for models exploring and evaluating, (e.g., model quality assessment through metrics associated with thresholds).

Bad smells constitute another quality issue on models, which might affect the quality of software systems. Bad smells were originally applied to source code as they are particularly defined as "*certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality*". These concepts can be applicable to the context of modeling because design smells might have the same negative impacts on the software development quality as code smells. Bad design decisions (design smells) might reduce the evolution and maintainability of the system under construction. Therefore, detecting design smells and (if possible) resolving them during the design phase helps to improve the quality of the software. Multiple bad design smells have been introduced in the literature and the detection of their existence in models is far from trivial. Therefore, several approaches have been presented and integrated into tools for automating design smells detection [Fer+16]. The three-level modeling architecture introduced in this thesis allows designers to query and examine the user model. Thus, its utilization for design smells detection can be a potential approach.

1.1 Summary of the Results

The goal of this thesis is to extend a two-level modeling to support full accessibility to the metalevel and to subsequently utilize such modeling architecture to achieve UML and OCL model quality. In particular, this thesis contributes to the literature in the following aspects:

- We propose and implement an approach for extending a traditional two-level modeling architecture to a three-level modeling architecture based on the OMG metamodeling architecture. With our approach, developers now can simultaneously access three linguistic instantiation levels. Therefore, they can (1) formulate reflective queries and constraints, and (2) define level-crossing queries and constraints.

- Based on the proposed three-level modeling architecture, we present an approach for metric measurement using OCL on the metamodel. These metrics are defined as operations of separated metaclasses and can be used instantly in formulating reflexive queries and invariants. We also propose a methodology of design smell detection using the proposed three-level modeling architecture. Designers can define a library of smell definitions to check the existence of design errors of the model under consideration.

- We perform an evaluation of metrics measurement and smell detection on an extensive data set of UML models in practice. This evaluation not only illustrates the feasibility and usefulness of our metric measurement and smell detection approaches but also helps to answer the research questions about the common characteristics and quality of models in practice.

This thesis extends results that have been partly published during the years of conducting a PhD research. In particular, the work of extending the OMG architecture for metamodeling and its application for model reflective querying and level crossing invariants is introduced in [DG18b]. Approaches that utilize metamodeling for metric measurement and model quality assessment are described in [DG18a]. [DG19] presents an extension of the tool UML-based Specification Environment (USE) for metric calculation and smell detection.

1.2 Structure of the Dissertation

This dissertation is structured as follows. The foundation for the main work of this thesis is presented in Chapter 2. Some concepts of metamodeling are firstly introduced and discussed. An overview of UML and OCL, an object constraint language often used to enrich the semantics of UML models, is then presented. As most of the work in this thesis is validated by an implementation in tool USE, an overview of this tool is illustrated as well. The primary objective of this thesis is to assess and achieve the UML model quality; the succeeding sections of this chapter thus discuss the concept of model quality and model assessment. The contributions for extending a two-level modeling to a three-level modeling are outlined in Chapter 3. This chapter also presents the application for model reflective querying and level crossing invariants. Chapter 4 focuses on the work of utilizing the three-level modeling architecture for

model metrics measurement and quality assurance. An evaluation of metrics measurement and smell detection is detailed in Chapter 5. This chapter also includes some conclusions of the common characteristics and quality of models in practice derived from the evaluation data. Chapter 6 presents a summary of the additional contributions of the author, which have been published during the years of conducting a PhD research. Finally, Chapter 7 consists of the conclusion and discussion for the future work.

Chapter 2

Background

This chapter presents the fundamental concepts and artifacts important to the main work in this thesis. In particular, the modeling languages and the tool, i.e., the Unified Modeling Language (UML), the related textual Object Constraint Language (OCL) and tool UML-based Specification Environment (USE) are introduced in the first section. In the next sections, the basis about model quality and model measurement with metrics are provided with discussions.

2.1 Modeling and Metamodeling

2.1.1 Model and Metamodel

In the literature, many definitions of models, for example in [Sei03; KWB03; RJB05; Béz05], have been proposed in the context of model-driven engineering. One of them [RJB05] defined a model as "A *model* is a representation in a certain medium of something in the same or another medium. The model captures the important aspects of the thing being modeled from a certain point of view and simplifies or omits the rest." A model could be considered as an abstraction of things in the real world, which allows people to concentrate on the essentials of a (complex) system by excluding non-essential details. In the context of model-driven engineering, models are considered as central artifacts in the development of large software systems. For example, developers build different models throughout the development process in order to verify that the eventual software system will meet the requirements. *Modeling* is an activity of building representations for a model, makes it easy to understand and provides a mean to investigate and analysis it.

Metamodel, meanwhile, is a higher abstraction level since it defines the structure and semantics of models. OMG's MOF specification [Obj15a] defines metamodel as: *"A metamodel is a model that defines the language for expressing a model"*. In other words, it is considered as a 'model of a model'. A model conforms to its metamodel only if all the properties of the model fulfill the rules, constraints, and syntax of its metamodel. Generally speaking, a metamodel can be any language specification written in English, such as the W3C's OWL language specification or the OMG's UML specification. And because a modemodel in fact is a model, then it might need a meta-meta model as its metamodel. This process can go further up to an arbitrary number of metamodel. The metalmodel at the topmost level can self-defined. That means it does not need a further metamodel to define it. The process of generating such metamodels is called *metamodeling*. UML models and metamodels, which are presented in Section. 2.2, are examples of a model and a metamodel. In the following, we distinguish two type of metamodeling, i.e., *strict* metamodeling and *loose* metamodeling.

2.1.2 Strict Metamodeling

InstanceOf relationship

There exists an instance-of relationship between two modelling elements X, Y if X is instantiated from Y. X is also considered as an instance-of Y. In term of set membership theory, the instance-of relationship can be seen as "member-of". Specifically, if X is an instance-of Y, then Y refers to a set and X refers to a member of the set Y. The definition of Y defines the semantics that all member (e.g. X) must be satisfied. The instance-of relationship can be referred to as "created-by" if we see it as a technique for deriving one model element from another. Particularly, X is instantiated from Y means X is created from Y. Then the attributes of Y become slots of X, with concrete values, and the associations between Y and other modelling elements Y become the link between X and other modelling elements Xn where Xn also are instantiated from Yn. In UML, the instance-of relationship is denoted as `X : Y`, e.g., `Ibm : Company`.

Strict Metamodeling

A "strict metamodeling" [AK02; Atk97] multi-level modeling architecture is based on the principle that the adjacent layers in the architecture are related

by and only by the *instance-of* relationship. Moreover, if a model A is instantiated from model B then every element of model A must be instantiated from some element of B model. In other words, in a strict metamodeling' multi-level architecture, the instance-of relationship can only exists between the elements at one level and the elements at an immediately adjacent level. Any relationship, which is other than the instance-of, between two elements X and Y implies that X and Y must be at the same level. Strict memamodeling can be formally defined as follows [AK02]

Definition 2.1. Strict metamodeling
In an n-level modeling architecture, $M_0, M_1, ..., M_{n-1}$, every element of an M_m - level model must be an instance of exactly one element of an M_{m+1}- level model, for all $0 \leq m < n-1$, and any relationship other than the instanceOf relationship between two elements X and Y implies that $level(X) = level(Y)$.

Fig 2.1 illustrates the principle of strict metamodeling.

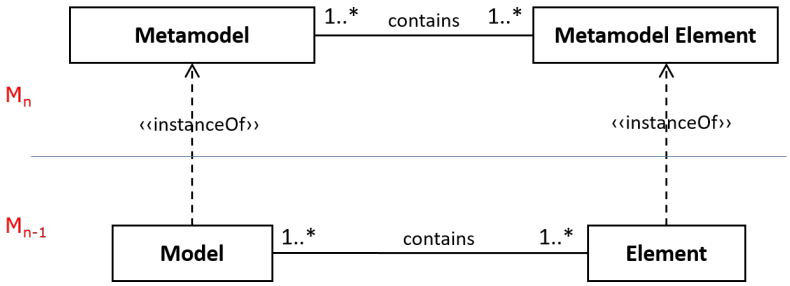


Figure 2.1: Strict Metamodeling (slightly modified from [AK02, p. 9])

Strict Metamodeling vs Loose Metamodeling

Another approach of metamodeling is 'loose metamodeling'. In loose metamodeling, a model element X is not only fully instanced from one another model element Y, but also partly instanced from some other model elements. Moreover, the models that X instantiated from can be at different levels of metamodeling. This is different from the principle of strict metamodeling where the instance-of relationship can only exist between the elements at two adjacent layers. It raises a problem that the loose metamodeling approach will blur the level boundaries between metamodels. The relationship between metamodel levels is now not only the instance-of but also can be all kind of relationships, i.e., inheritance relationships, associations and links. Strict metamodeling has a well-formed and structured meta-hierarchy connecting by

an instance-of relationship. Meanwhile, loose metamodeling ends up with a complex net crossing metamodel levels with different kinds of relationship. As a result, loose metamodeling is more flexible than strict metamodeling, but also much more complicated. One more significant problem of loose metamodeling is the need for a new instantiation mechanism to make it work. This technique must be completely different from the well-established full instance-of mechanism in object-oriented approaches. To overcome this issue, the mechanism called *"deep instantiation"* is introduced in [AK08]. *Clabjects*, a new modeling concept which has both class facet and object facet, becomes a center modeling element. With this technique, the properties of a real time object at level 0, can be instantiated from different clabject elements at different modeling levels. Each clabject might contain instantiated properties with value or the definitions of properties, which will be instantiated by clabjects or real time object at lower levels.

The work in this thesis follows the strict metamodeling architecture to extend traditional two-level modeling approaches to three-level modeling and utilize this for model quality assessment, i.e., metric measurement and smell detection.

2.2 The Unified Modeling Language (UML)

2.2.1 Overview of UML

With the flourish of object-oriented programming in the 1980s, a need for a language to specify object-oriented system design also became essential. By merging the work from Grady Booch [Boo95], James Rumbaugh [Rum+91], Ivar Jacobsen [Jac93], in 1997 such kind of modeling language has been released, which they called the Unified Modeling Language (UML). That was the first version (1.0) of UML. This unified language was later adopted as a standard by the Object Management Group (OMG) and has been managed by this organization since then. UML 1.x was accepted among the community as a well-defined, expressive, powerful, and generally applicable modeling language. In the year 2005, UML 2 has been released to address the issues that arose during the application of UML 1.X. The current version is UML 2.5, released in 2017, December. Nowadays UML is considered as a standardized modeling language, which is developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of

software systems. To do that, the current version of UML provides fourteen different diagrams; each diagram specifies a different viewpoint to a system. For example, system developers can use a *use case diagram* to illustrate an abstract view of the list of different functionality (use cases)a system offers and the corresponding actors that interact with these use cases. Structure of a system and various kinds of static relationships which exist between these structural elements could be presented by a *class diagram*. Modeling behavioral/dynamics aspects of a system can be performed by several diagrams in UML. Among them, *state machine diagrams* depict the permitted states of an object in a system and transitions between them as well as the events that trigger these transitions. It helps to visualize the entire lifecycle of objects and thus could provide a better understanding of state-based systems.

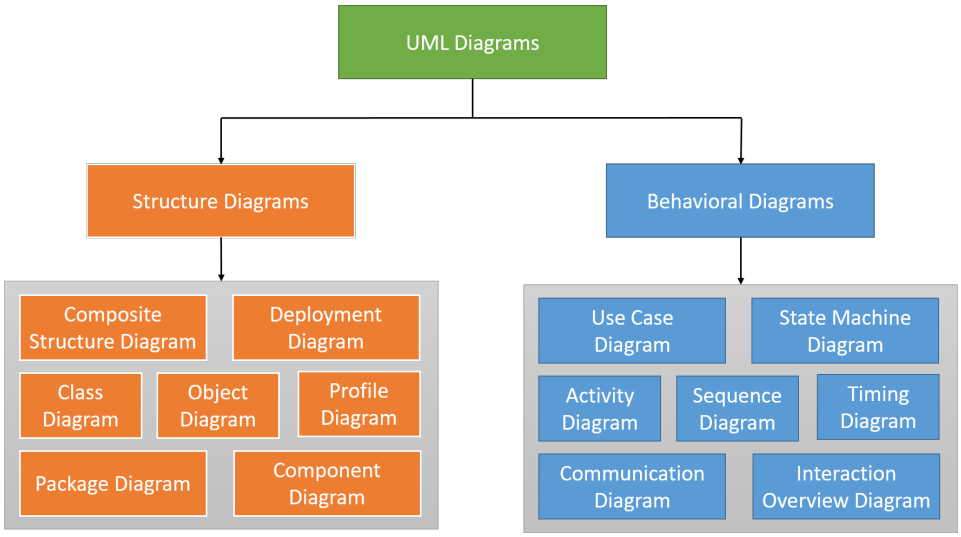


Figure 2.2: UML diagrams

Figure 2.2 lists all fourteen UML diagrams that UML 2.5 supports. These diagrams are divided into two main categories: structure diagrams and behavioral diagrams, based on the corresponding aspects that they depict. For the more detailed explanation and usage of the available UML diagrams, please refer to [Obj15b]. The work in this thesis highly relates to class diagram and object diagram; therefore in the following parts, these two diagrams will be discussed more detail.

2.2.2 Class Diagram

A class diagram is a graphical representation of a *class model* that describes the structure view of a system. In order to have a better understanding of the semantics behind a class diagram, the following part introduces an abstract syntax definition for a class model.

Class Model

The definition of a class model below extends the definition in [Dan09; Ric02] and follows the UML 2.5 specification in [Obj15b].

Definition 2.2. Abstract Syntax for Class Model
A class model $\mathcal{CM}$ is a tuple:

$$\mathcal{CM} = (\mathcal{C}, \mathcal{T}, \mathcal{P}, \mathcal{OP}, \mathcal{A}, \mathcal{G}, Mappings, Constraints)$$

where:

- $\mathcal{C}$ is a set of named elements each element representing a class in the system. Each class $c \in \mathcal{C}$ induces an object-type $t_c \in \mathcal{T}$.
- $\mathcal{T}$ is a set of types that are available in this class model for the definition of attributions, association ends, or operation parameters. The set $\mathcal{T}$ includes primitive types, i.e., Integer, Boolean, String, Real; the enumerations and set of $\{t_c\}$ for each class $c \in \mathcal{C}$.
- $\mathcal{P}$ is a set of named elements, each element representing an attribute of a class $c \in \mathcal{C}$ or an association end of an association $a \in \mathcal{A}$. A property $p \in \mathcal{P}$ can be declared as the following signatures $p : t$ or $p : t[m_1..m_2]$, where $m_1 \in \mathbb{N}$ and $m_2 \in \mathbb{N} \cup \{*\}$ ($*$ stands for the unlimited number). t is called the *type* of property p and $[m_1..m_2]$ is called the *multiplicity*.
- $\mathcal{OP}$ is a set of named elements, each element representing an operation of a class $c \in \mathcal{C}$. An operation can be declared as the following signature $t_1 \times t_2 \times ... \times t_n : t$, where $t, t_1, t_2, ..., t_n \in \mathcal{T}$.
- $\mathcal{A}$ is a set of named elements, each element representing a relation (association) between classes $\{c_i\}$ where $c_i \in \mathcal{C}$.
- $\mathcal{G}$ is a set of generalization hierarchy relationships $\prec$ between classes in $\mathcal{C}$. $\prec$ can be defined as follows: $\prec \subseteq \mathcal{C} \times \mathcal{C}$. More specifically, $c_1 \prec c_2$

indicates that c_1 is a subclass of c_2, that means c_1 inherits features from c_2 and also can add new features or can redefine existing ones. $\prec^*$ is the transitive closure of $\prec$.

- *Mappings* is a set of mappings, each mapping describes syntactic interrelationships between different class model elements, i.e., classes, properties, operations, associations. There are two kinds of mappings: $e_1 \Rightarrow e_2$ means class model element e_1 owns e_2; and $e_1 \rightarrow e_2$ means class model element e_1 uses e_2. Below are all kind of mappings in a class model:

  1. $c \Rightarrow p$, where $c \in \mathcal{C}, p \in \mathcal{P}$ indicates that class c owns attribute p.
  2. $c \Rightarrow op$, where $c \in \mathcal{C}, op \in \mathcal{OP}$ indicates that class c owns operation op.
  3. $a \rightarrow c$, where $a \in \mathcal{A}, c \in \mathcal{C}$ indicates that association a uses class c.
  4. $a \Rightarrow p$, where $a \in \mathcal{A}, p \in \mathcal{P}$ indicates that association a owns association end p.

- *Constraints* specify semantic requirements that can be enforced on the class model elements. The semantics of some constraints implies syntactic restrictions that must hold on the class model, which make the constraints to be satisfied. Constraints are usually formulated in a constraint language, OCL for example, and could be: class invariants; pre- and postconditions of operations; derived attributes; subsetting, redefinition and union properties.

Class Diagram

A class diagram is a graphic illustration of a *class model* that describes the structure view of a system. Classes and the static relationship between them are the centre of a class diagram. A class is presented as a three-compartments rectangle; name at the top, the attributes that the class owns in the middle and the operation signatures of the class at the bottom. As indicated above, classes can participate in various relations to each other, i.e., generalization and association. UML offers different graphical notations for each type of relation. In UML class diagrams, a generalization is depicted as a solid line with a hollow arrowhead ($\triangleright$) that points from the subclass to the supperclass. A general association is presented as a solid line between related classes. One can mark an end of the association as an aggregation using an unfilled diamond

(◇), or as a composition using a filled diamond (◆). Another special type of association is a *dependency*, which is displayed as a dashed line with an open arrow. An association is usually displayed with additional information: name, the name of the ends, and the multiplicities of each end.

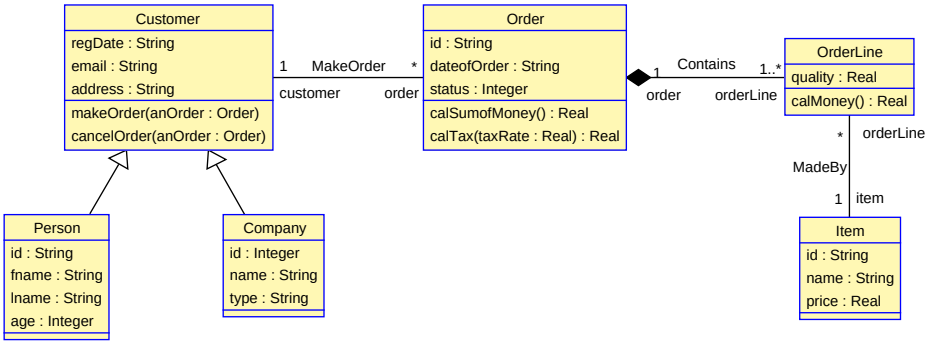


Figure 2.3: An example of UML class diagrams

Figure 2.3 presents an example of a class diagram for an online order system. The class model $\mathcal{CM}$ can be formally described using the Definition 2.2, where $\mathcal{C}$ = {`Person, Company, Customer, Order, OrderLine, Item`} and $\mathcal{G}$ = {`Person` $\prec$ `Customer, Company` $\prec$ `Customer`} for example. As can be seen from the class diagram in Figure 2.3, class `Customer` has three attributes, i.e., `email : String`, `address : String`, `regDate : String`, and two operations, i.e., `makeOrder()`, `cancelOrder()`. Classes `Person` and `Company` are subclasses of the class `Customer` and a hierarchy relationship is depicted as a solid line with a hollow arrowhead (▷). There are three associations between classes, one of them is `Contains`. This association is a composition (presented as ◆) since an order detail must be a part of an order. Two ends of the composition `Contains` are `order` and `orderdetail` with the multiplicities attached to them are `1` and `1..*`, respectively. That means one `Order` object can have many `OrderDetail` objects and an `OrderDetail` object can only be part of exactly one `Order` object.

2.2.3 Object Diagram

An object diagram is a diagrammatical presentation of an *object model*, which depicts a state of the corresponding system (a *class model*) at a point in time. An object model contains concrete instances of classes, called objects, and of associations, called links. That is the reason why an object model is also called a *system state* or a *snapshot*.

Object Model

We assume that the system, which provides the semantics for the object model $\mathcal{OM}$, is defined as $\mathcal{CM} = (\mathcal{C}, \mathcal{T}, \mathcal{P}, \mathcal{OP}, \mathcal{A}, \mathcal{G}, Mappings, Constraints)$ (see Definition 2.2 for more detail). Before defining an abstract syntax for an object model $\mathcal{OM}$, firstly the definitions of few functions on the object model $\mathcal{OM}$ will be presented.

Definition 2.3. "instanceOf" Function
An "instanceOf" function σ is depicted as:

- $\forall c \in \mathcal{C}, \sigma(c)$ is a finite set of all objects(instances) of a class c existing in the object model $\mathcal{OM}$.
- $\forall a \in \mathcal{A}, \sigma(a)$ is a finite set of all links(instances) of an association a existing in the object model $\mathcal{OM}$.

Definition 2.4. "typeOf" Function
An "typeOf" function ρ is depicted as: for each instance ins is an object or a link in the object model $\mathcal{OM}$, $\rho(ins)$ is a class or an association that the instance ins is instantiated from.

Definition 2.5. Object Model
An object model $\mathcal{OM}$ of a class model $\mathcal{CM}$ is a tuple:

$$\mathcal{OM} = \sigma(\mathcal{CM}) = (\sigma(\mathcal{C}), \sigma(\mathcal{A}), \Pi)$$

Where:

- $\sigma(\mathcal{C}) = \bigcup_{c \in \mathcal{C}} \sigma(c)$
- $\sigma(\mathcal{A}) = \bigcup_{a \in \mathcal{A}} \sigma(a)$
- For each type $t \in \mathcal{T}$, assume $I(t)$ is the domain of t. Π is a finite set of the value assignment operator $\pi : \pi(p_{ins}) := v$, where:
  1. p_{ins}, with $p \in \mathcal{P}$ and $ins \in \sigma(\mathcal{C}) \cup \sigma(\mathcal{A})$, indicates an attribute/association end p of an instance ins in $\mathcal{OM}$.
  2. $\exists (c \Rightarrow p) \in Mappings$ where $c = \rho(ins)$.

3. For each $p \in \mathcal{P}$, assuming m_p is the number of value assignment operators of p in Π. The following condition must be satisfied: $m_1 \leq m_p \leq m_2$, where $[m_1..m_2]$ is the *multiplicity* setting of p.
4. $v \in I(t)$ is a value of type t, where t is the type of property p.

Object Diagram

As mentioned above, an object diagram is a graphic visualization of an object model: $\mathcal{OM} = (\sigma(\mathcal{C}), \sigma(\mathcal{A}), \Pi)$. It shows a system state at a particular moment, which is concrete in nature. The central elements in an object diagram are objects, instances of classes, and links, instances of associations. An object $o \in \sigma(\mathcal{C})$ is presented similarly as a class in class diagrams except for several small differences. Firstly, an object rectangle does not have the third compartment for operations, since object diagrams only specify particular states of the system. The first compartment depicts the name of the object together with the name of its class ($\rho(o)$), separated by a colon. Both of them are printed underlined. The attributes of objects are presented together with their corresponding values, which are defined as value assignment operators in Π. Links connect objects in an object diagram, and they are visualized like associations in class diagrams. The objects participating in a link are defined as value assignment operators in Π.

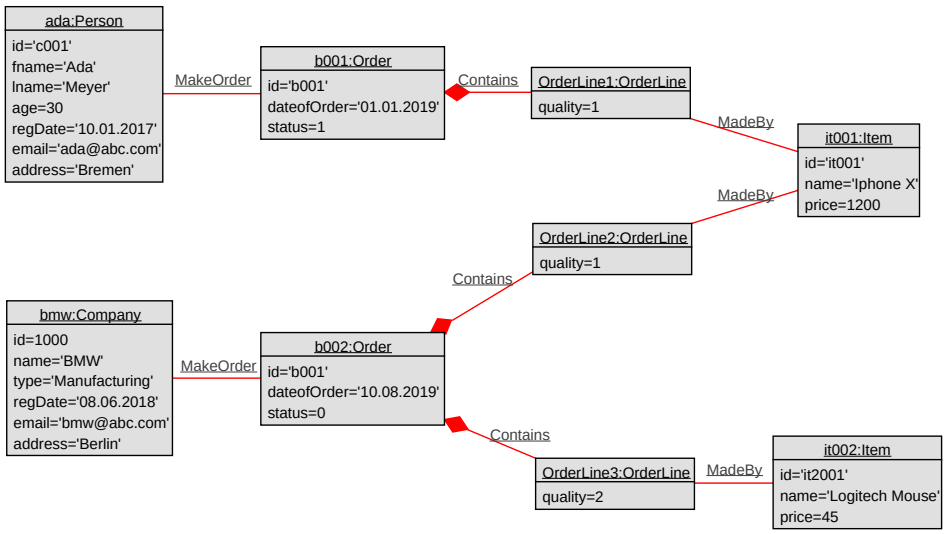


Figure 2.4: An example of UML object diagrams

Figure 2.4 shows an example of an object diagram, which specifies a state $\mathcal{OM} = (\sigma(\mathcal{C}), \sigma(\mathcal{A}), \Pi)$ of the system presented as a class diagram in Fig-

ure 2.3. As can be seen from Figure 2.4, `ada:Person` $\in$ σ`(Person)` is an example of objects in the system state; and $id_{ada} :=' c001'$ is an example of value assignment operators, which assigns value for the attribute `id` of the customer `ada`.

2.2.4 UML Metamodel

OMG has defined the Meta-Object-Facility (MOF) [Obj15a] as a fundamental standard for modeling. MOF provides a four-layer architecture for system modeling, with an expressed aim of providing an infrastructure "to support the creation, manipulation and interchange of metamodels" [Obj11a]. This architecture is showed in Figure 2.5.

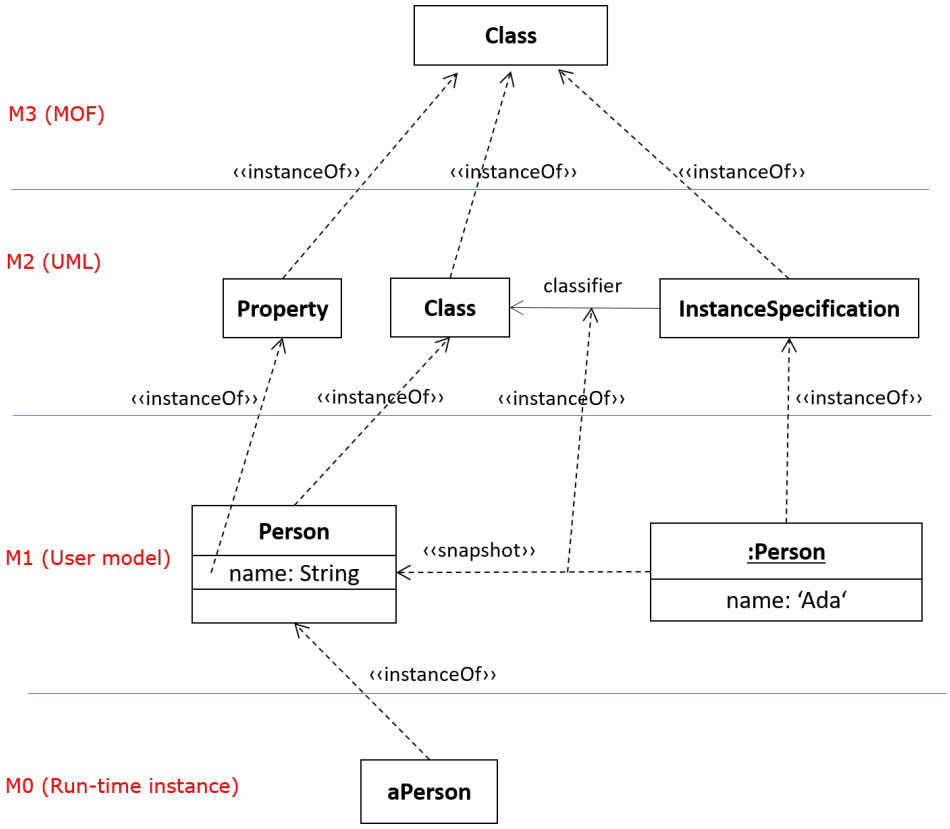


Figure 2.5: UML Four-Level Metamodel Architecture (slightly modified from [Obj11a, p. 20])

The top layer in the MOF architecture, named M3 (or MOF: Meta-Object Facility), is a meta-meta model. This meta-meta model is the language used to build the metamodels at the lower layer, called M2. The UML metamodel, which is used to describe the UML, is the most well-known example of a

model at the M2 layer. The models at layer M2 describe the elements and the structure of the models at layer M1. Models at layer M1 can be, for instance, models written in UML. The last and bottom layer in the MOF architecture is the layer M0 (also called data layer). Models at this layer describe run-time instances or objects of the model (some authors explicate 'M0' as being the 'real world').

'Strict metamodelling' is fundamental in the four-layer architecture proposed by the OMG. Generally speaking, adjacent layers in the architecture are related by the instance-of relationship, and this relationship cannot exist within any single layer. This means that a lower layer is used for instantiating the next upper layer and if a model A is an instance of another model B then every element of A must be an instance of some elements in B [Atk97]. One could also say that the same entities at a middle layer M_i can be (A) objects for the next upper layer M_{i+1} and (B) classes for the entities at the next lower layer M_{i-1}.

As can be seen from Figure 2.5, the metaclasses `Property` and `Class` are both defined as part of the UML metamodel. These metaclass are instantiated in a user model, for example, the class `Person` is an instantiation of the metaclass `Class` and the attribute `title` is an instantiation of the metaclass `Property`. `InstanceSpecification` is a special metaclass, where its instances is used simply as an illustration (a snapshot) of the classes and they will be created as a part of the user model and placed in the M1 level. The UML documentation [Obj11a, p. 19] states that run-time instances, that are created at M0 should not be confused with instances of the metaclass `InstanceSpecification`, e.g. ada:Person (or the more anonymous :Person).

The elements and semantics of M2 level (UML metamodel) and M1 level (User model) are essential to this work as will be presented in later chapters.

2.3 The Object Constraint Language (OCL)

This section presents a brief introduction of the Object Constraint Language (OCL). Readers who are already familiar with OCL might skip this section. The detailed explanation can be found in [Obj06; CG12; Ric02].

2.3.1 OCL in a Nutshell

It is clear that the modeling elements that a graphical modeling language (UML, for example) offers, are not sufficient to precisely describe all relevant details of a system. For instance, the business rules or constraints that must be satisfied with the system cannot be expressed by only using graphical notations of class diagram. That is the reason why OCL (and in general, any other textual language) is integrated into UML (or other graphical languages). It enriches UML models by precisely specifying all detailed aspects of a system design. OCL is a general-purpose (textual) formal language utilized to describe expressions on UML models. Following are several characteristics of OCL:

- OCL is a typed language. That means each valid (well-formed) OCL expression evaluates to a type. OCL types could be predefined types or user-defined types available the UML model where the OCL expression is applied. Predefined types includes basic types, i.e., `Integer`, `Real`, `String`, and `Boolean`, and collection types, i.e., `Set(t)`, `Bag(t)`, `Sequence(t)`, `OrderedSet(t)`, `Collection(t)`, and `Tuple(t)`.

- OCL is a declarative language. That means the language definition specifies what is to be done without any implementation details or implementation guidelines.

- OCL is a side effect language. This implies OCL expressions can apply to query or constrain the system state but cannot modify it.

Among many applications of OCL, we select here several kinds of expressions that can be defined by OCL within the context of UML class diagrams:

- Invariants

- Query Operations

- Operation contracts

- Query expressions

Brief introduction of these kinds of OCL expressions with examples are presenting in the following section.

2.3.2 Examples of OCL Expressions

The examples presented in this section are applicable to the class model illustrated in 2.3. In principle, OCL expressions are formulated on the level of class model(M1) and their semantics are applied to the level of run-time instances(M0).

Invariants. Each invariant states a condition that must be satisfied in all possible system state of the class model. An invariant is often defined as a boolean OCL expression within the context of a corresponding class. For example, the following expression ensures the quality of an order line must be greater than zero.

```
context OrderLine inv qualityGreaterThanZero:
   self.quality > 0
```

Derived elements. Several elements of a class model, such as attributes or association ends, can be inferred from the value of other model elements. To do that, a derivation rule is specified for the derived element. These derivation rules can be defined by OCL and the value of a derived elements is determined according to the evaluation of its derivation rule.

```
context Order:: numberOfItem: Integer derived =
   self.orderLine→size()
```

Query operations. A query operation is a class operation that does not change the state of the system. OCL could be utilized as a query language, similar to SQL on a database, to define the body of query operations. These wrapped OCL expression query data over a system state and return the desired information to the user. For example, the following OCL expression defines the operation `OrderLine::calMoney()`, which calculate the money of one order item.

```
context OrderLine::calMoney(): Real
body: self.quality * self.item.price
```

Operation contracts. A precondition of an operation contract is a condition or predicate that must hold on the system state before the execution of the operation while a postcondition is a condition or predicate that must be satisfied by the system state after the operation execution. The following example specify the pre- and post conditions of the `Customer:: cancelOrder` operation.

```
context Customer:: cancelOrder(anOrder: Order)
```

```
pre anOrderOk:
  anOrder<>null and self.order→includes(anOrder)

post linkRemoved:
  self.order→excludes(anOrder)
```

2.4 The UML-based Specification Environment (USE)

The work in this thesis is closely related to the tool USE (UML-based Specification Environment) in some aspects, i.e., by manipulating, extending it. Therefore, this section briefly introduces this tool. USE is a modeling tool, which supports the specification, visualization, validation and verification of information systems based on UML and OCL. An input is a textual specification contains the description of model elements found in UML class diagrams (classes, associations, etc.). The model is enriched by additional integrity constraints written in the Object Constraint Language (OCL). Users can create a system state on the specified model and validate the defined constraints on the created system state. The figure below shows an overview of that process in USE.

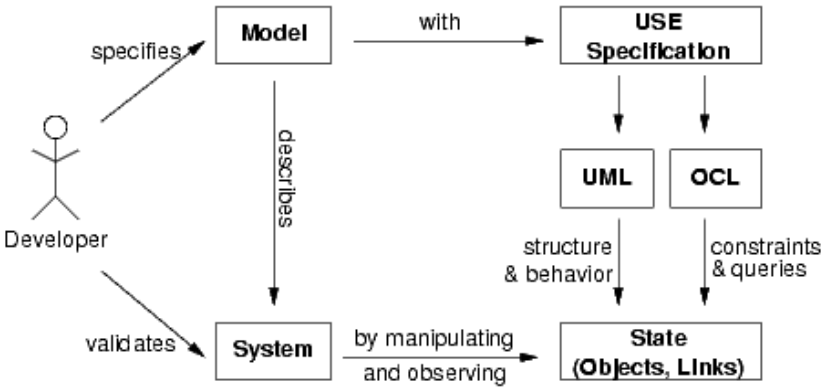


Figure 2.6: Overview of the specification workflow [Uni07]

USE offers many kinds of UML diagrams, e.g., class diagram, object diagram, sequence diagram, etc. It also provides the features for model querying, validation, verification and exploration. Details about using tool USE as well as the diagrams and features that it offers can be found in [Uni07; GBR07; GHD18]. Figure 2.7 presents a screenshot of several diagrams, features in USE.

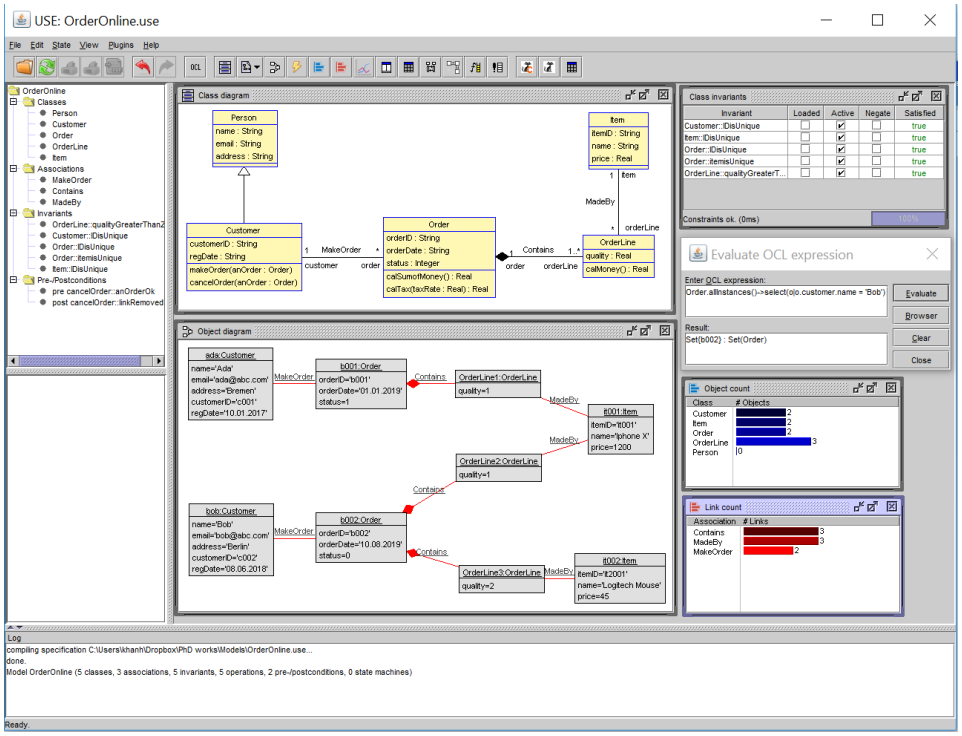


Figure 2.7: A screenshot of the tool USE

2.5 Model Quality Assurance

Model quality assurance helps modelers to detect errors or mistakes on their models, to fix bugs and to improve the models. These assessment properties might include design properties: absence of isolated classes, respecting naming conventions (e.g., the name of every element must obey the camelCase convention) or metrics properties (e.g., a generalization hierarchy is not too deep). In this section, model quality in general and the principle of model quality assurance will be presented.

2.5.1 Software Quality

The meaning of the notion "software quality" is different based on the views of different user groups. For end users, a high quality system must be simple to use, run fast and be reliable. Meanwhile, developers expect the system should be easy to maintain, reuse and adapt to requirement changes, for example.

In principle, software quality is evaluated through observation of different external attributes. **External quality attributes** are those that can be measured only with respect to how the software product relates to its envi-

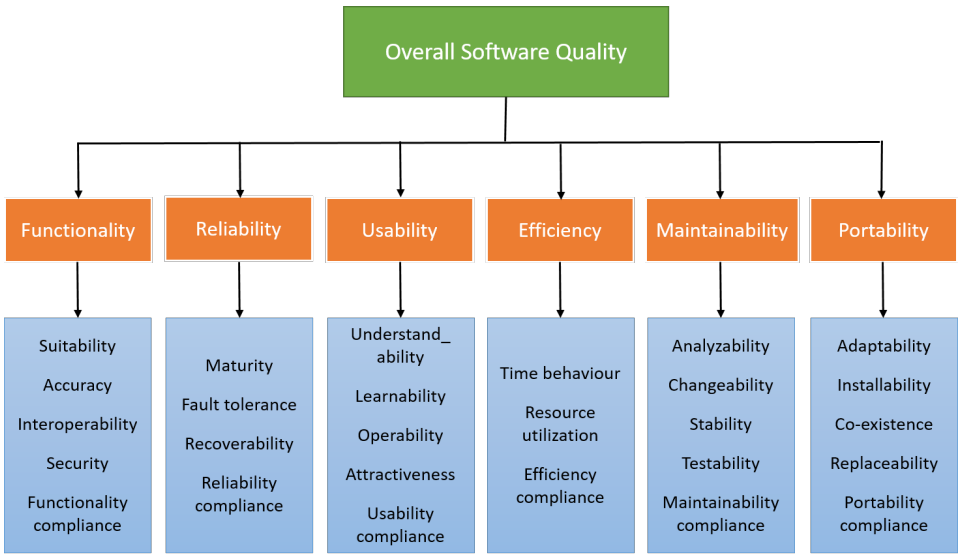


Figure 2.8: The ISO/IEC 9126 software quality characteristics

ronment. For example, a system is considered to have *poor reliability*, if it does not perform as expected. According to ISO/IEC 9126 [ISO01], software quality attributes can be stated as follows: functionality, reliability, usability, efficiency, maintainability and portability. These characteristics are decomposed into the corresponding sub-characteristics as shown in Figure 2.8. The shortly definition of these quality characteristics could be described as follows.

- **Functionality**: a set of attributes that describe the availability of functions and their specified properties in the system. The functions are those that fulfill the needs.
- **Reliability**: a set of attributes that describe the degree to which the software system maintaining its level of performance under particular conditions and time period.
- **Usability**: is the degree to which a software can be understood and use to accomplish quantified tasks with effectiveness, efficiency, and satisfaction under specified conditions of use.
- **Efficiency**: is related to a set of attributes that evaluates the performance of the software with respect to the amount of resources used and under stated conditions.
- **Maintainability**: a set of attributes that specify the effort needed to modify a part of the software.

- **Portability**: indicates the capability of software to be transferred from one environment to other.

It is clear that in order to measure external attributes, information about the software itself is not enough. Thus, external attributes can only be measured late in the software development process. However, it is costly to identify the problems and to fix them during the external quality assessment at later phases. Therefore, approximate evaluation of external qualities by assessing internal qualities could be a potential alternative.

Internal quality attributes are those that can be measured by considering the product itself. Utilizing the information about the product itself, for example, the design model, one can measure several structural properties such as size or coupling. These internal measurements, are not directly meaningful to the product qualities. However, because of the causal impact of the internal qualities on the external qualities, the measurement of internal qualities could provide a vision of the overall outcome quality of the product. Actually, a number of empirical studies indicated in the survey [GPC05] have proved that optimizing certain internal qualities (e.g., reducing the complexity of the design) can lead to several particular desirable external qualities (e.g., increasing the maintainability of the product). In the MDE paradigm, class model is one of the crucial artifacts. It can be considered as a backbone of the Object-oriented MDE development and provides a foundation for the software design and the implementation at the later phase. The quality of class model has enormous influence over the quality of the system that is eventually implemented. Extracting internal qualities of the class model, therefore, is a kind of early measurement. Since the work in this thesis deals with the quality characteristics for UML models, therefore only internal quality is considered.

2.5.2 Class Model Metrics

For a long time, metrics are utilized to gather quantitative data which can provide the answers about the quality of a software system. They are applicable to measure and therefore assess the internal quality attributes of systems. Generally, both the behavior aspects and static aspects of a software system can be measured by metrics. Since the work in this thesis only deals with the quality of UML class models, in the following, we only introduce and discuss model level metrics.

Class model metrics are the metrics which quantitatively measure the qual-

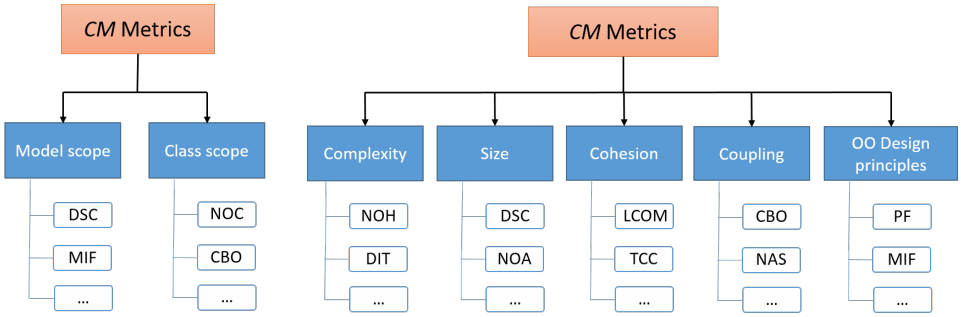


Figure 2.9: A classification of class model metrics

ity aspects of a class model. Utilizing class model metrics to assess the internal quality attributes can be an early indicator for the quality of the final output system. Based on the context that the metrics are applied, we categorize metrics into two groups, i.e., model scope and class scope, as shown in the left part of Fig. 2.9.

- **Model scope metrics** are the metrics whose context is the whole model, for example, Total number of classes in the model (DSC) [BD02], Method inheritance factor in the model (MIF) [AM96].

- **Class scope metrics** are metrics which measure an individual class in the model. In other words, class scope metrics are applicable to the context of a particular class. Number of direct children (NOC) [CK94], Coupling between object (CBO) [CK94] are examples of class scope metrics.

Metrics can also be classified by the quality aspects they measure. The right part of Fig 2.9 presents the categories under this view point.

- **Complexity metrics** measure how complex a design or class is. Number of Hierarchies (NOH) [BD02] and Depth of Inheritance (DIT) [CK94] are examples of complexity metrics.

- **Size metrics** measure how large a design or class is. In other words, these metrics give us the number of major elements in a design or class. For example, DSC metric [BD02] is the total number of classes in the design or NOA metric [BHe96] gives us the number of attributes of a class.

- **Cohesion metrics** measure how well the elements inside a class related to each other. In other words, this kind of metrics deals with

the strength of the relationship between the class's methods and data themselves. The degree of class cohesion is usually described as *high cohesion* or *low cohesion*. A class is *high cohesion* if its elements are highly correlated. Classes with high cohesion are considered better since they are more reliable, reusable, and understandable. Lack of Cohesion of Methods (LCOM) [CK94], Tight and Loose Class Cohesion (TCC and LCC) [BK95] are popular cohesion metrics.

- **Coupling metrics** generally deal with the degree of interdependence between software modules. In the context of software design, these metrics measure how strong the interconnecting between packages or classes. Low coupling is an indicator of a good design because it increases the readability and maintainability. Coupling between object (CBO) [CK94] and Number of associations (NAS) [HCN98] are examples of coupling metrics.

- **Object-oriented design principles metrics** measure the degree of object-oriented mechanisms, e.g., inheritance, encapsulation, polymorphism, applied in a design. Method inheritance factor (MIF) [AM96] and Polymorphism factor (PF) [AM96] are examples of this metric category.

2.5.3 Design Smells and Refactory

Smells were originally applied to source code as they are particularly defined as "certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality"[SSS14]. They usually indicate the weaknesses in code design that might slow down the software evolution and maintainability. Most of code smells can be resolved by applying changes to the internal structure of source code that can make it easier to understand and improve the maintainability without changing its behavior. Such technique is called code refactoring. From the above definition of smells and refactoring, it is clear that these concepts can be applicable to the context of modeling because design smells might have the same negative impacts on the software development quality as code smells. Indeed, there are many approaches have been introduced in the literature for design smells detection and refactoring [Str+16; Bas+16; LGL14; AGO12; Bet+19]. Smells can be distinguished between automatically detectable smells and manually detectable smells. In

this thesis, we only deal with the first category, which can be detected by utilizing the internal structure information of the model. Figure 2.10 shows the general model quality assessment process, which includes three main steps: model analysis, smells identification and smells resolution [Bet+19].

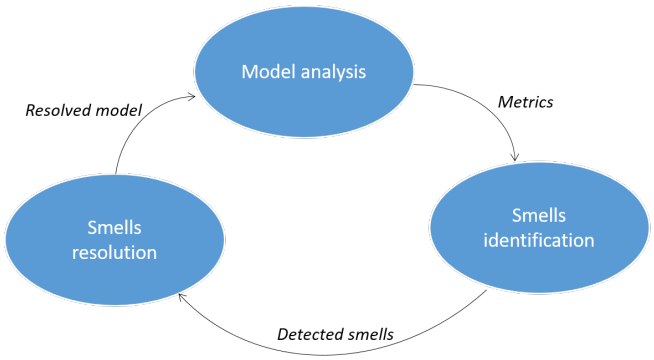


Figure 2.10: The general view of model quality assurance process

In particular, firstly the model is analyzed, for example, calculating the metrics. Next, the information achieved from the first step is utilized for smells identification. In this step, the smells and the corresponding responsible model elements are specified. The last step, i.e., smells resolution, resolves the identified smells by applying some changes to the identified responsible model elements. As a result, the smells have been resolved and therefore the quality of the model has been enhanced. The design smells, or design quality properties can be categorized into four groups depending on their nature and relevance. [LGL14]

Essential property: smells that might lead to a faulty design (an error). These kinds of smells should be definitely removed from the design through a refactory. One of them is the smell "duplicate features in sibling classes". This smell occurs when a feature (attribute or operation), is present in all sub-classes of a super-class with the same definition. The redundant features introduce the difficulty of the maintainability and reusability of the considered model. To eliminate the issue of duplicate features, a resolution can be applied by pulling up the duplicate features to the parent class. Another smell is dead class or isolated class. This smell occurs when a class in the considered model is completely disconnected from other model elements, i.e., not involved in any association or hierarchy. According to [Str+16], this smell has a negative impact on the understandability of the model and it could be resolved by removing the isolated class if the developers make sure that the classes are no

longer need.

Design guidelines: smells that might cause the model does not meet the design guidelines. The non-existence of uninstantiable classes, i.e., abstract class without concrete children, is an example of this kind of quality properties.

Naming conventions: a set of rules (quality properties) for naming the features in the model. For example, every class should be named in pascal-case, attributes and operations should be named in camel-case. The refactoring operation that can be applied in this case is simply renaming the elements following the rules.

Metrics: smells that relates to metrics on the model. Metrics measurement must be done before the detection of these smells is performed. Overloaded classes, e.g., classes that have more than ten attributes, is an example. Another example could be the existence of too deep hierarchy, e.g., inheritance hierarchy deeper than five levels.

Chapter 3

UML Models Analysis by Utilizing Metamodeling

In this chapter, the work on an extending of OMG architecture for metamodeling and its application for model reflective querying and level crossing invariants is summarized. In particular, the first section presents the contribution published in [DG18b]. In this contribution we have presented an approach to extend a two-level modeling application to a three-level modeling framework where the middle level can be seen at the same time as an object diagram, i.e., the instantiation of the upper level model, and as a class diagram, i.e., the type model for the lower level. Based on these ideas, an approach for reflective constraints and queries and the application of this approach to model quality assessment has been proposed. A proposal towards level-crossing constraints was also introduced: a proposal that offers writing more powerful and flexible constraints.

3.1 Extending Two-Level Modeling for Metamodeling

3.1.1 Methodology

A traditional two-level modeling approach offers two levels of modeling, i.e., a type model at level M1 and run-time instances of the type model at level M0. For example, in the tool USE (UML-based Specification Environment) the model at level M1 is a user UML model that describe the system (is visualized as a class diagram) and at layer M0 is a real-time snapshot of the system (is

visualized as an object diagrams). In this chapter, we introduce a methodology, in which a two-level modeling approach is extended to a three-level modeling based on the MOF architecture. This methodology is also integrated into USE as an evaluation of its usability. To make our approach more clear and self-contained, firstly several major terms which are used very often are defined as follows:

- *User model*: a UML and OCL class model that describes the structure of a system to be modeled. This model is formally specified as $\mathcal{CM} = (\mathcal{C}, \mathcal{T}, \mathcal{P}, \mathcal{OP}, \mathcal{A}, \mathcal{G}, Mappings, Constraints)$ (see Definition 2.2)
- *Metamodel*: the UML metamodel (the OMG superstructure) [Obj11b] that provides the notions to define UML class models. This *metamodel* itself is an explicit UML and OCL class model, and is formally specified as $\mathcal{CM}_m = (\mathcal{C}_m, \mathcal{T}_m, \mathcal{P}_m, \mathcal{OP}_m, \mathcal{A}_m, \mathcal{G}_m, Mappings_m, Constraints_m)$
- *Metamodel instantiation*: an automatically generated instantiation of the metamodel that reflects and is equivalent to the *user model*. It can be formally specified as $\sigma(\mathcal{CM}_m) \Leftrightarrow \mathcal{CM}$
- *Metaclass*: a class of the metamodel, $c_m \in \mathcal{C}_m$
- *Metaobject*: an instance of a metaclass, $ins_m \in \sigma(c_m)$
- *Userclass*: a class of the metamodel, $c \in \mathcal{C}$
- *Run-time object*: an instance of a user class, $ins \in \sigma(c)$

As presented in Sec. 2.2.4, the Meta-Object-Facility (MOF) [Obj15a] has been defined by the OMG as a fundamental standard for modeling. MOF provides a four level architecture for system modeling.

Based on the MOF architecture, we now extend a two-level modeling approach by making the third OMG level M2 explicitly available. Figure 3.1 shows the general schema for our three level modeling approach. The model at level M2 is the *metamodel*. This metamodel itself is an explicit UML class model formulated in USE with (currently) 63 classes and 99 associations. It is pre-loaded as a metamodel for all user models at level M1 and is fixed during the modeling process. To provide a better understanding of our approach, an excerpt of the UML metamodel which presents important elements for defining the semantics of the *user model* at level M1 is shown in Fig. 3.2.

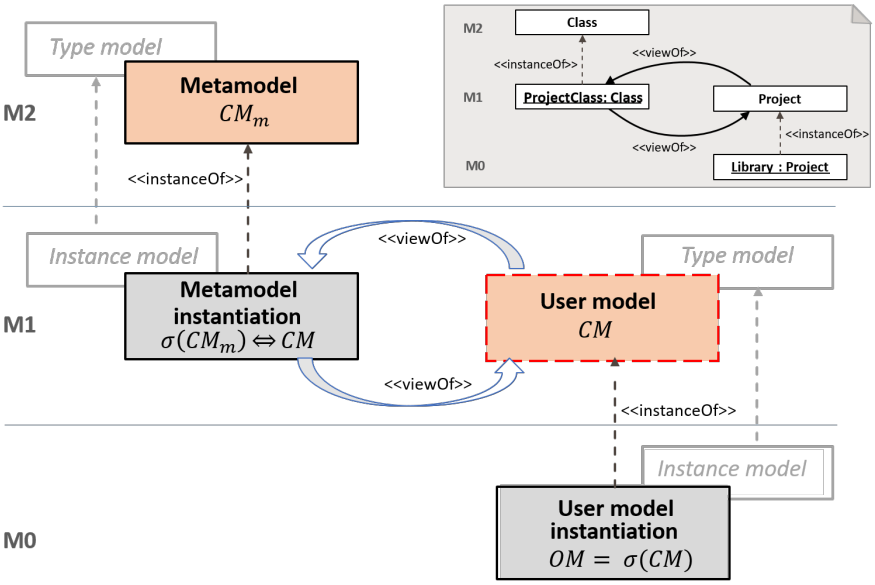


Figure 3.1: General schema for three level modeling.

The *user model* is the central artifact in our metamodeling approach as highlighted by the dashed rectangle in Fig. 3.1. However, it is clear that the user model in the form of a class model is not related to the upper level (metamodel at M2 level) by an instance-of relationship. That means it violates the *"strict metamodeling"* principle of the OMG architecture. To make our metamodeling approach following the *"strict metamodeling"* principle, a *metamodel instantiation* that reflects and is equivalent to the *user model* is generated and added to the M1 level, the same level as the *user model*. These two models are always in sync; therefore, we can say that now the M2 and M1 levels are related by the instance-of relationship. As will be presented in the following parts of this thesis, the *metamodel* and *metamodel instantiation* are the fundamentals for user model metrics measurement, evaluation and smell detection. This *metamodel instantiation* contains a number of *metaobjects*. Each *metaobject* is an instance of a *metaclass* and the character of the generated *metaobjects* and links is determined by the *user model*. For example, if there are two classes in the *user model*, then two *metaobjects* instantiated from *metaclass* `Class` are generated. Each *metaobject* is named as a combination of the name of the corresponding element from the *user model* and the corresponding *metaclass* from which it is instantiated. A simple example of our three level modeling approach is presented in the right upper corner of Fig. 3.1. The metaobject `ProjectClass` is an instance of *metaclass* `Class` and its name is the combination of 'Project', the name of the class from *user model*, and 'Class', the name

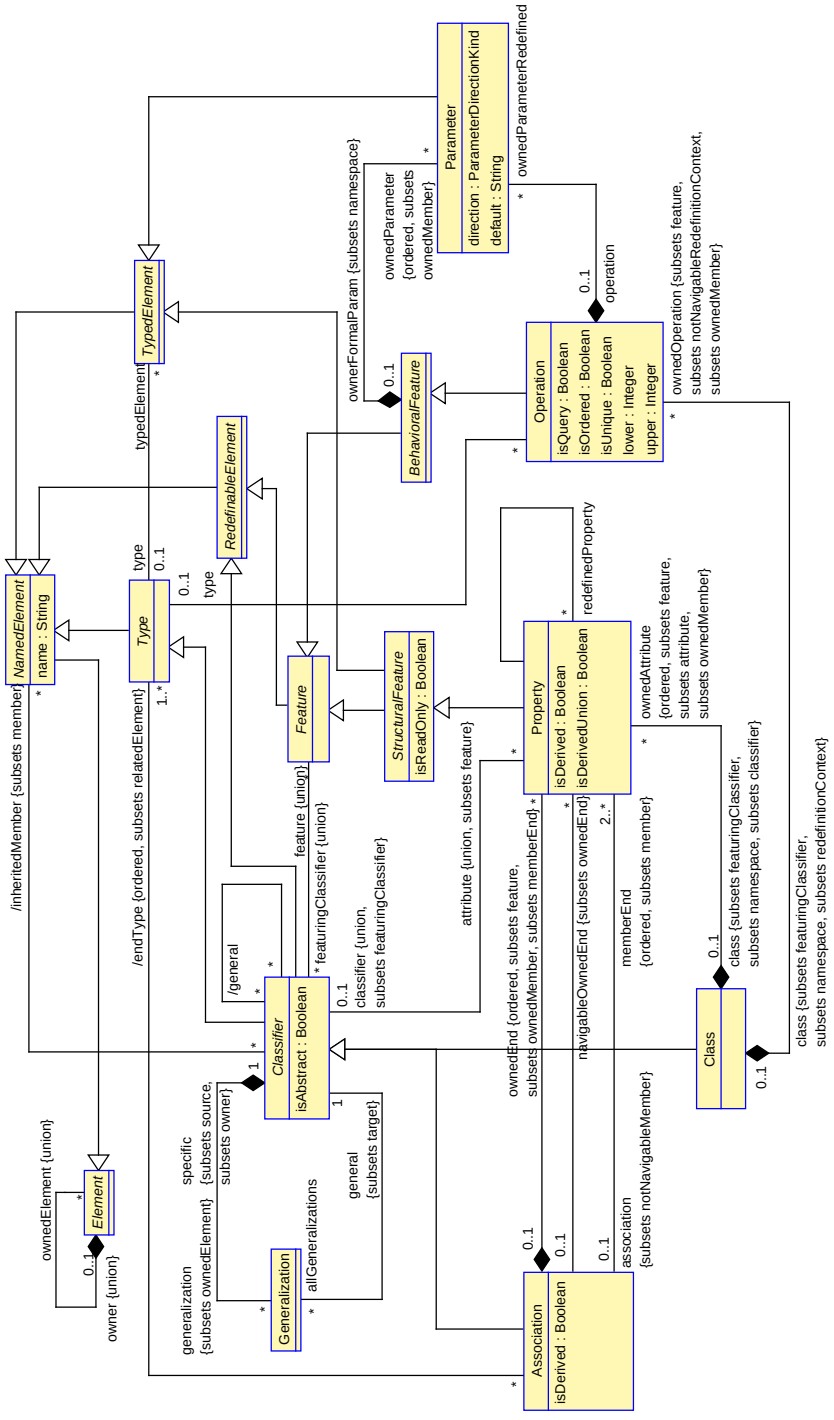


Figure 3.2: Central UML metamodel elements defining the user model.

of the *metaclass*. Our approach visits all user model elements (e.g., classes, attributes, operations, associations) and generates the corresponding *metaobjects* and links. Below is the detailed algorithm for generating the metamodel instantiation $\sigma(\mathcal{CM}_m)$ corresponding to the input user model $\mathcal{CM}$. This algorithm runs in the background when a user model is loaded to the system.

Algorithm 1 Generating the metamodel instantiation of user model

Input: A class model $\mathcal{CM} = (\mathcal{C}, \mathcal{T}, \mathcal{P}, \mathcal{OP}, \mathcal{A}, \mathcal{G}, Mappings, Constraints)$
Output: $\sigma(\mathcal{CM}_m) = (\sigma(\mathcal{C}_m), \sigma(\mathcal{A}_m), \Pi) \Leftrightarrow \mathcal{CM}$

1: $\sigma(\mathcal{C}_m) \leftarrow \emptyset$
2: $\sigma(\mathcal{A}_m) \leftarrow \emptyset$
3: Initialize the mapping $\mathcal{M}$. //from Table 3.1
4: **for all** $t \in \mathcal{T}$ **do**
5: **if** t is a primitive type **then**
6: $ins \leftarrow generateMetaobject(t,' DataType')$
7: $\sigma(\mathcal{C}_m) \leftarrow \sigma(\mathcal{C}_m) \cup ins$
8: **end if**
9: **end for**
10: $E \leftarrow Seq\{\mathcal{C}, \mathcal{A}, \mathcal{P}, \mathcal{OP}\}$
11: **for all** user model element $e \in E$ **do**
12: $Set\{c_m\} \leftarrow \mathcal{M}.getMetaclasses(e)$
13: **for all** $c_m \in Set\{c_m\}$ **do**
14: $o_m \leftarrow generateMetaobject(e, c_m)$ $//o_m \in \sigma(c_m)$
15: $\sigma(\mathcal{C}_m) \leftarrow \sigma(\mathcal{C}_m) \cup o_m$
16: **end for**
17: $Set\{a_m\} \leftarrow \mathcal{M}.getMetaassociations(e)$
18: **for all** $a_m \in Set\{a_m\}$ **do**
19: $l_m \leftarrow generateMetalink(ins, a_m)$ $//l_m \in \sigma(a_m)$
20: $\sigma(\mathcal{A}_m) \leftarrow \sigma(\mathcal{A}_m) \cup l_m$
21: **end for**
22: **end for**
23: **return** $\sigma(\mathcal{CM}_m) = (\sigma(\mathcal{C}_m), \sigma(\mathcal{A}_m), \Pi)$

Algorithm explanation:

- $generateMetaobject(e, c_m)$: the method to generate a *metaobject* $\sigma(c_m)$ as an instance of the metaclass c_m corresponding to the user model element e. Within this procedure, the set of value assignments π are also established for the generated *metaobject*. In the case implementing this algorithm in USE, we utilize the USE specific language SOIL (Simple OCL-based Imperative Language) [BG11] to create the *metaobject*.

- $generateMetalink(e, a_m)$: the method to generate a *metalink* $\sigma(a_m)$ as an instance of the metaassociation a_m corresponding to the user model element e. Within this procedure, the set of value assignments π are also established for the generated *metalink*. In the case implementing this algorithm in USE, we utilize the USE specific language SOIL (Simple OCL-based Imperative Language) [BG11] to create the *metalink*.

- $\mathcal{M}.getMetaclasses(e)$: this method returns the metaclasses, which are the type classes of the user model element e as defined in the mapping $\mathcal{M}$.

- $\mathcal{M}.getMetaassociations(e)$: this method returns the corresponding meta associations related to the user model element e as defined in the mapping $\mathcal{M}$.

Table 3.1 shows the mapping $\mathcal{M}$ between the *user model* elements and the related *metaclasses* and *metaassociations*, which are the type elements for the generated *metaobjects* and *metalinks*. Specifically, in each row, the item in the first column is a user model element, and the items in the second column are the classes that directly relate to the user model element, i.e., a typing model element. The third column contains the related meta-model associations (the subscript text includes the names of association ends corresponding to the classes).

For example, if e is an attribute of a class in the user model, then the function $\mathcal{M}.getMetaclass(e)$ will return the corresponding metamodel class, i.e., Property metaclass. The function $\mathcal{M}.getMetaassociations(e)$ will return two corresponding meta associations, i.e., $\text{Class}_{\text{class}}$ – $\text{Property}_{\text{ownedAttribute}}$ and $\text{Property}_{\text{typedElement}}$ – $\text{Type}_{\text{type}}$

3.1.2 Three-layer Modeling Representation

In principle, a two-layer modeling approach provides two levels of representation, e.g., the class model view at the level M1 and the object model view at the level M0. As discussed before, the work in this thesis mainly deals with the model quality assessment utilizing metamodeling, therefore, exploring and understanding the metamodel and the connection between modeling levels are helpful, especially when one needs to define new metrics or design smells. Thanks to the auto-generated *meta instantiation* at level M1, simultaneously representing three-level of modeling is now possible within our work.

Table 3.1: Mapping between user model elements and metamodel elements.

User model element	**Related metamodel class**	**Related metamodel association**
Primitive Type	DataType	
Class	Class	
Attribute	Property	$\text{Class}_{\text{class}}$ – $\text{Property}_{\text{ownedAttribute}}$ $\text{Property}_{\text{typedElement}}$ – $\text{Type}_{\text{type}}$
Association	Association	$\text{Class}_{\text{endType}}$ – $\text{Association}_{\text{association}}$
Association End	Property	$\text{Association}_{\text{association}}$ – $\text{Property}_{\text{memberEnd}}$ $\text{Association}_{\text{association}}$ – $\text{Property}_{\text{ownedEnd}}$
Operation	Operation	$\text{Class}_{\text{class}}$ – $\text{Operation}_{\text{ownedOperation}}$ $\text{Operation}_{\text{typedElement}}$ – $\text{Type}_{\text{type}}$
Parameter	Parameter	$\text{Operation}_{\text{operation}}$ – $\text{Parameter}_{\text{ownedParameter}}$ $\text{Parameter}_{\text{typedElement}}$ – $\text{Type}_{\text{type}}$
Generalization	Generalization	$\text{Class}_{\text{subClass}}$ – $\text{Class}_{\text{superClass}}$ $\text{Generalization}_{\text{Generalization}}$ – $\text{Class}_{\text{specific}}$ $\text{Generalization}_{\text{Generalization}}$ – $\text{Class}_{\text{general}}$
Redefined Attribute/ Redefined Association End		$\text{Property}_{\text{property}}$ – $\text{Property}_{\text{redefinedProperty}}$
Subsetted Attribute/ Subsetted Association End		$\text{Property}_{\text{property}}$ – $\text{Property}_{\text{subsettedProperty}}$
Class Invariant	Constraint	$\text{Class}_{\text{context}}$ – $\text{Constraint}_{\text{ownedRule}}$
Pre-condition	Constraint	$\text{Operation}_{\text{preContext}}$ – $\text{Constraint}_{\text{precondition}}$ $\text{Operation}_{\text{context}}$ – $\text{Constraint}_{\text{ownedRule}}$
Post-condition	Constraint	$\text{Operation}_{\text{postContext}}$ – $\text{Constraint}_{\text{postcondition}}$ $\text{Operation}_{\text{context}}$ – $\text{Constraint}_{\text{ownedRule}}$

Particularly, in the M1 layer, we provide two views on the user model. The first view is a class diagram view, which can be seen as a type model for the object model at the lower layer M0. The second view is an object diagram view that is an instantiation of the UML meta-model at layer M2 and that corresponds to the loaded UML user model. These two views represent the same information and are always in sync.

Fig. 3.3 contains an example of a three-level modeling representation has been integrated into the tool USE. The user model in this example is a simple model, with two classes, Employee and Department, and an association WorksIn as shown in the right middle part of Fig. 3.3. As mentioned above, the full meta-model includes 63 classes and 99 associations. Therefore, viewing the full meta-model is not practical and sometimes not necessary, because many of the meta-model elements might not be used to describe the current user model. For example, the meta-model class Operation is not used to describe any element in the Employee-Department model. Starting from these observations, we provide a simplified view for the meta-model, as shown in the upper part of Fig. 3.3. To construct the simplified view from the full meta-model, we drop all unnecessary classes and associations, which are not needed for any element of the current user model. In the simplified view, we only show the meta-model

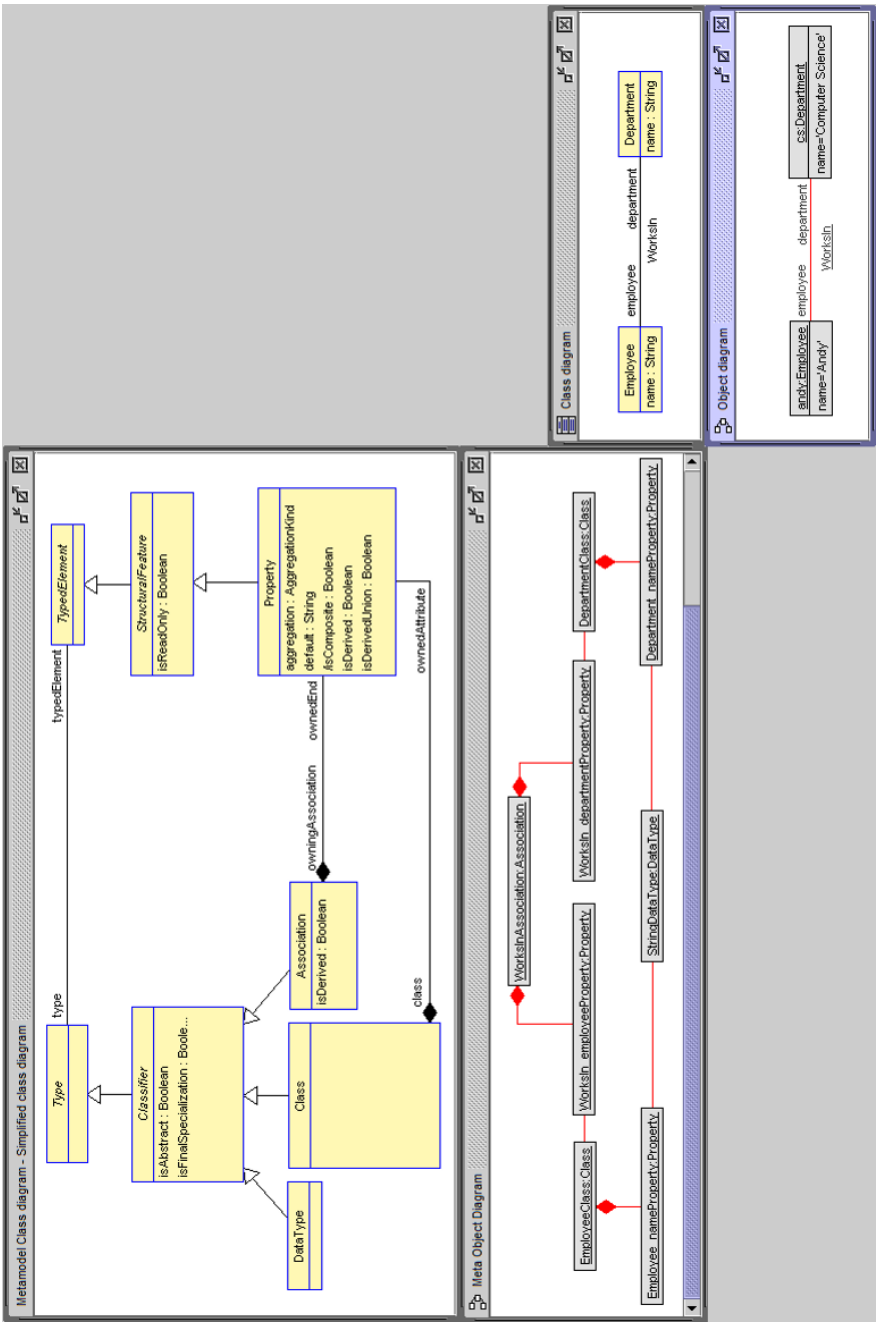


Figure 3.3: Three-layer model representation in USE.

elements, i.e., classes and associations, that the user model needs as type elements to instantiate model elements. Table 3.1 shows the mapping between the user model elements and the related meta-model classes and associations.

Based on this mapping, we can detect which meta-model elements will be displayed in the simplified view. For example, if a class in the user model contains attributes then the Property metaclass and two metaassociations, i.e.,

$\text{Class}_{\text{class}}$ – $\text{Property}_{\text{ownedAttribute}}$ and $\text{Property}_{\text{typedElement}}$ – $\text{Type}_{\text{type}}$ will be shown. Concerning the example and as the result of the described mapping, only three classes, Class, Property, Association, and the corresponding associations are shown in the simplified meta-model view for the Department-Employee user model. Additionally, we still provide a full meta-model view, in case the developer wants to explore it.

The left middle part in Fig. 3.3 is the user model represented as an instance of the meta-model. As can be seen, every element of the user model is an instance of a class from the meta model. Each instance is named as a combination of the name of the corresponding element from the user model and the meta-model class from which it is instantiated. For example, the object WorksInAssociation is an instance of metaclass Association and its name combines 'WorksIn', the name of the association from user model, and 'Association', the name of the meta-model class. The object diagram shown in the lower part of Fig. 3.3 is a run-time instance of the user model. It is the model at layer M0. 'CS' and 'Andy' are instances of the Department class and the Employee class, respectively.

There is a number of derived links between objects in the meta-instance view. However, to make the meta-instance view at layer M1 more focussing on the instanceOf aspect, we only show the direct links and do not show these derived links. The two views in layer M1 describe one model. They are equivalent and kept in sync. Each element in the user model class diagram view, e.g., a class, an attribute or an association, is presented as a meta-instance in the meta-instance view. If there is any change in the user model, e.g., a name change, an addition or a deletion of an element, the object diagram in meta-instance view will be updated. An example of a synchronous change on the views at layers M1 and M2 is presented and highlighted in Fig. 3.4. The change on the user model is made by (A) adding the operation 'numberOfEmp(): Integer' into class Department. Consequently, two corresponding meta-instances are synchronously (B) added to the meta-instance view, i.e., the Department_numberOfEmp:Operation instance and the IntegerDataType:DataType instance for the return data type 'Integer'. And synchronously, the metaclass Operation and related associations are (C) added to the simplified meta-model view (at layer M2).

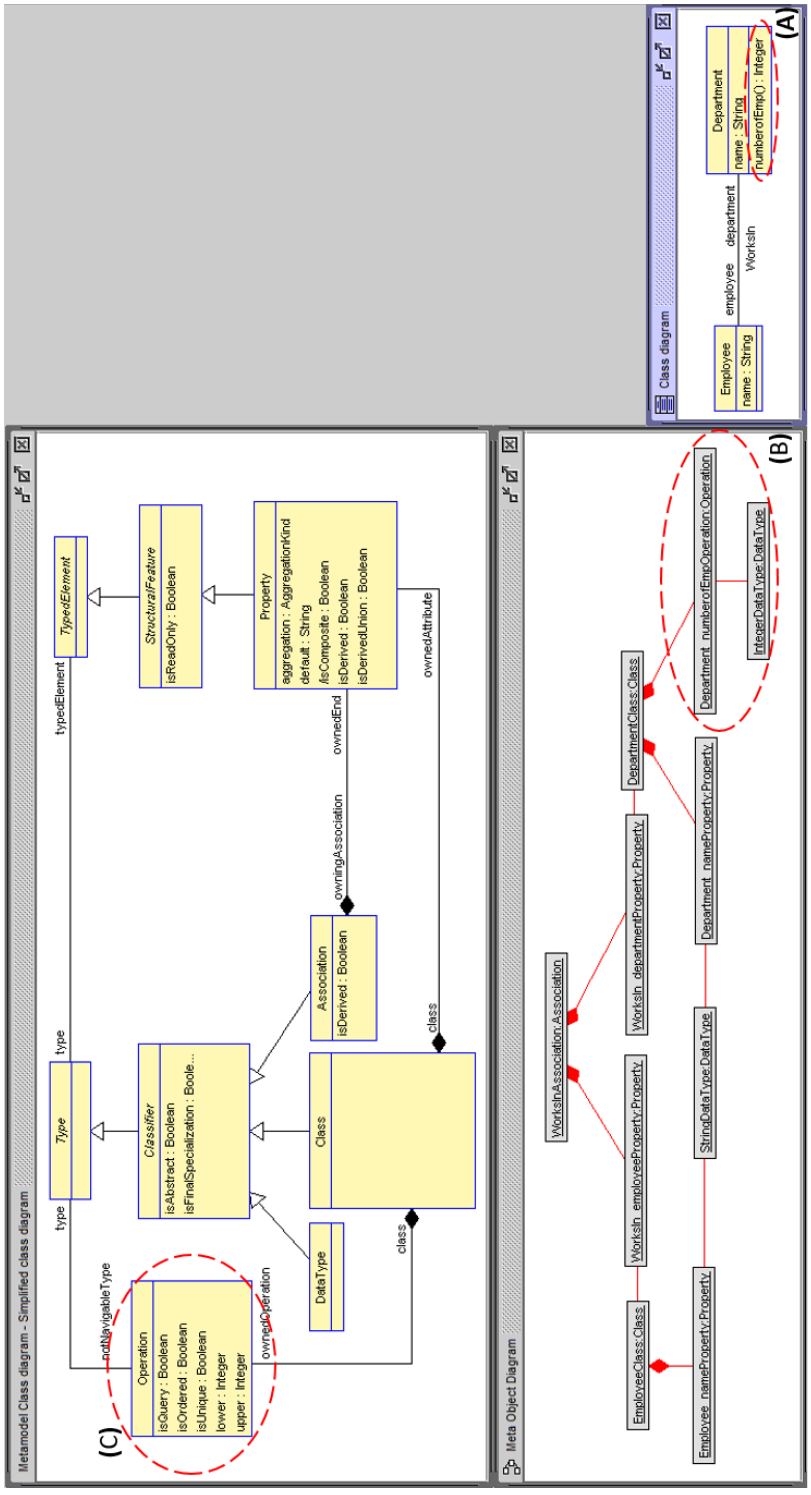


Figure 3.4: Synchronous changes on the views of layer M1 and M2.

3.1.3 Tool-based Reflective Querying

The access to the meta-level supported by standard OCL [Obj06] is limited, therefore writing reflective queries, e.g., "find the classes related to a given class c and their relevant roles", is impossible. In this section, we will introduce how our approach supports more meta-level access capabilities for writing reflective queries within the extended tool.

Meta-level Accessibility in OCL

Standard OCL is a formal language for formulating constraints and queries on UML models. OCL expressions are formulated on the level of classes (M1) and their semantics is applied on the level ofobjects (M0).

With the limited meta-level access capabilities, standard OCL cannot express a number of reflective queries and constraints. The following list presents several reflective queries that cannot be expressed with the standard OCL.

1. Find all classes related to a given class
2. Find names of all subclasses of a given class
3. Find all abstract classes
4. Find all classes that have more than 10 attributes
5. Find all classes that have more than 5 subclasses
6. Calculate the number of classes in a user model
7. Check for the setter and getter methods of all attributes
8. Find all classes of a user model that have no subclass

These queries, however, can be expressed with our three-level modeling approach thanks to the auto-generated metamodel instantiation $\sigma(\mathcal{CM}_m) \Leftrightarrow \mathcal{CM}$. In the next section, we will show how to formulate and execute reflective queries in the extended tool.

Writing Reflective OCL Querying in Tool USE

As introduced in the previous section, our approach supports a three-layer UML and OCL specification: instances, model, and metamodel. Through the

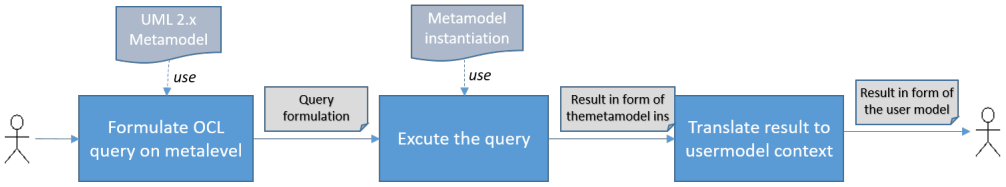


Figure 3.5: UML model reflexive querying.

tool support, one can access the metamodel and create OCL queries for the user model by considering it as an instance of the metamodel. Model querying using the metamodel approach provides possibilities for considering the elements contained in a model, for example, by accessing the attributes, operations, and referenced elements of a given model element, by executing comprehensions and quantified expressions. As depicted in Fig. 3.5, a reflexive query is an OCL expression on the metamodel layer, and the final result of executing the query is a Boolean value, scalar value or a set of user model elements. To achieve that, firstly the reflexive query is executed on the metamodel level and the result from executing the query (in form of the metamodel instantiation) is translated to user model context to display to the designer. These two steps are run in the background.

We have integrated the process depicted in Fig. 3.5 in to tool USE to show its usability. It is clear that the model queries such as "find the classes related to a class and their relevant roles" cannot be formulated in standard OCL directly. However, the extended tool USE now can deal with this kind of model query. For example, the query "find classes related to class Department via an association" on the Employee-Department example can be formulated by the following OCL expression and executed by our tool as shown in Fig. 3.6.

```
Association.allInstances()→select(a|a.endType→
  includes(DepartmentClass)).endType→excluding(DepartmentClass)
```

The Association meta class is the type element for associations in the user model and endType is an end of a derived association that can be used to navigate from the Association meta class to the Class metaclass. DepartmentClass is the meta-instance of the Class metaclass; its name is a combination of the user model element and the corresponding metaclass. The result of executing this query, i.e., Bag{EmployeeClass}, is also shown in Fig. 3.6.

Another example for model querying is presented in Fig. 3.7. This example is an Employee hierarchy model, a typical subclass-superclass generalization

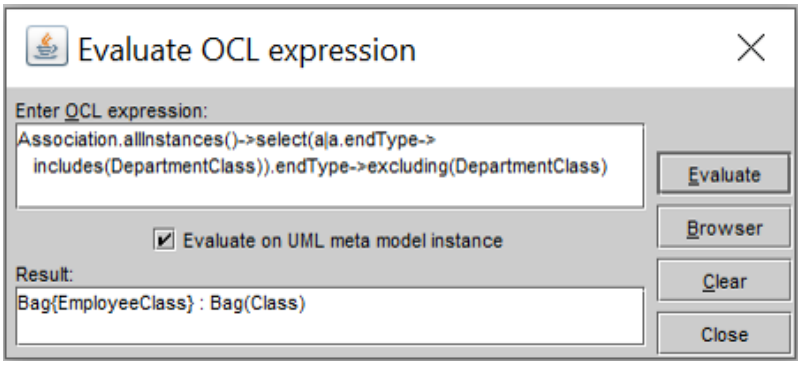


Figure 3.6: Model query example: Find related classes.

model with a four-level inheritance structure. For example, one might want to find all abstract classes within this model. The OCL query to perform this task is stated in Fig. 3.7. In particular, isAbstract is a Boolean attribute of the metaclass Class in order to define whether a class at level M1 (in a user model) is abstract or not.

```
Class.allInstances()→select(c|c.isAbstract)
```

To see more examples for model queries, we refer to the paper [Bal+16].

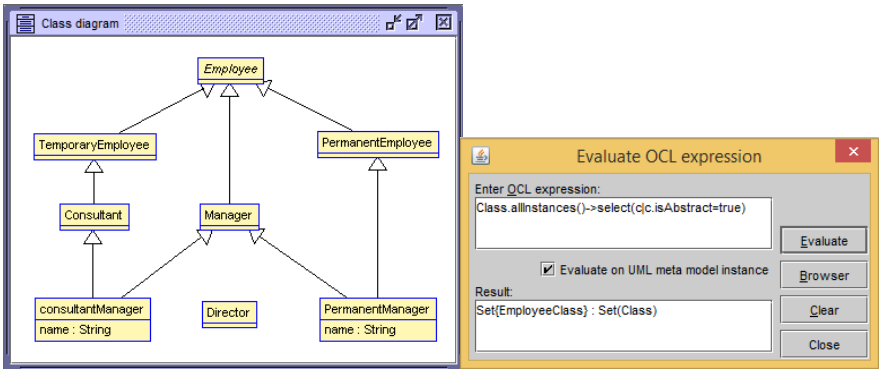


Figure 3.7: Model query example: Find abstract classes.

3.2 An Approach towards Level-Crossing OCL Expressions

Level-crossing invariants plays an important role in multi-level modeling. Standard OCL, however, only supports formulating constraints on the level of

classes (M1) and their semantics concerns the level of objects (M0). That means it is impossible to write expressions on objects at different linguistic levels, e.g., the expression "`Class.allInstances().allInstances()`" is syntactically invalid in OCL. Looking back to our approach on three-level modeling, we can see that designers now can access meta objects as well as run-time objects at the same time at the different level, i.e., the M1 level and the M0 level, respectively. In principle, our three-level modeling approach offers enough semantics for extending OCL for level-crossing constraints and queries, for example, formulating OCL expression on the metamodel level (M2) and their semantics is applied on the run-time instance level (M0). To achieve this feature, one possibility is introducing an additional notation in OCL, e.g., `#...#`. Specifically, the sub-expression within the `#...#` notation is an OCL expression formulated on the M1 level, and the super-expression outside the `#...#` is an OCL expression formulated on the M2 level which provides the context for the sub-expression. The `#...#` notation can be considered as a bridge to go from M2 level to M1 level.

For example, one could formulate a constraint to ensures that the value of the **age attribute** of all instances of all classes is over 18 (assuming every class in the model has an age attribute with Integer type) as follows:

```
Class.allInstances()→forAll(c |
 #c.allInstances()→forAll(age > 18)#)
```

Generally speaking, using this approach, one can write constraints that go from the M2 to the M0 level through the M1 level. This capability supports writing more powerful constraints. However, it is inflexible and it has a few restrictions: (a) the meta-level expression must return the type `Set(Class)` (to provide the context for the sub-expression), (b) an extension of OCL for a new notation, i.e., the `#...#` notation, is needed.

3.2.1 Methodology

As mentioned above, in order to offer an ability to write the level-crossing constraints or queries, in principle, introducing a new notation to travel from the upper level of modeling to the direct lower level is needed. In this section we present an approach which allows designers formulating and evaluating level-crossing invariants without introducing new OCL notations.

In order to do that, we make the semantics of the models at different levels available in only one level. Thus, we do not need to transfer between levels

in order to formulate and evaluate a level-crossing invariant. As presented in Section 3.1.1, the semantics of the usermodel $\mathcal{CM}$ is now already available at M1 level as a meta instantiation of the metamodel at level M2. We now make the semantics of the object model at level M0 available at M1 level. Particularly, we extend the architecture presented in Fig. 3.1 by adding to the M1 level metamodel instantiation $\sigma(\mathcal{CM}_m) \Leftrightarrow \mathcal{OM}$, which corresponds to the object model at level M0. Figure 3.8 shows the overview of the extended architecture.

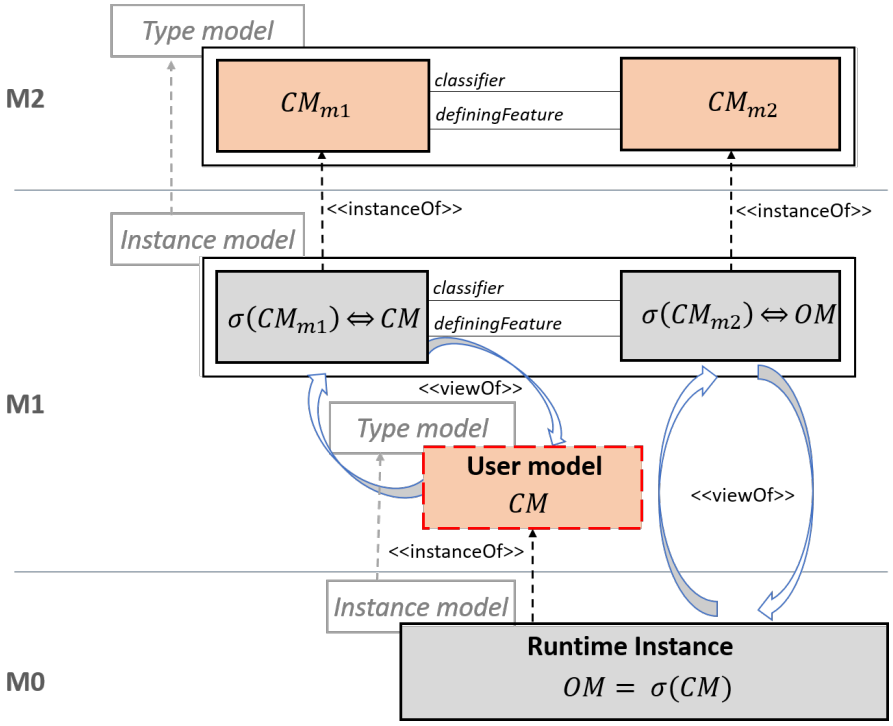


Figure 3.8: Extended architecture to support level-crossing invariant.

The UML metamodel provided by OMG [Obj11b] can be separated in to two parts. The first part(called CM_{m1}), whose main elements were described previously in Fig. 3.2, is used to described the user model. The second part (called CM_{m2}), which is presented in the lower part of Fig. 3.9, is originally used to describe an illustration (a snapshot) of classes' instances. In other words, for a set of instances of class $c \in \mathcal{C}$, only one of them need to be represented as an instance of the CM_{m2} with the connecting to class c. In this work, we extend that idea to utilize CM_{m2} not only describing a snapshot of classes but also describing all instances in the object model at the M0 level.

As can be seen in Fig. 3.9, there are two associations which connect two parts of the metamodel, one association connects the metaclasses `Classifier`

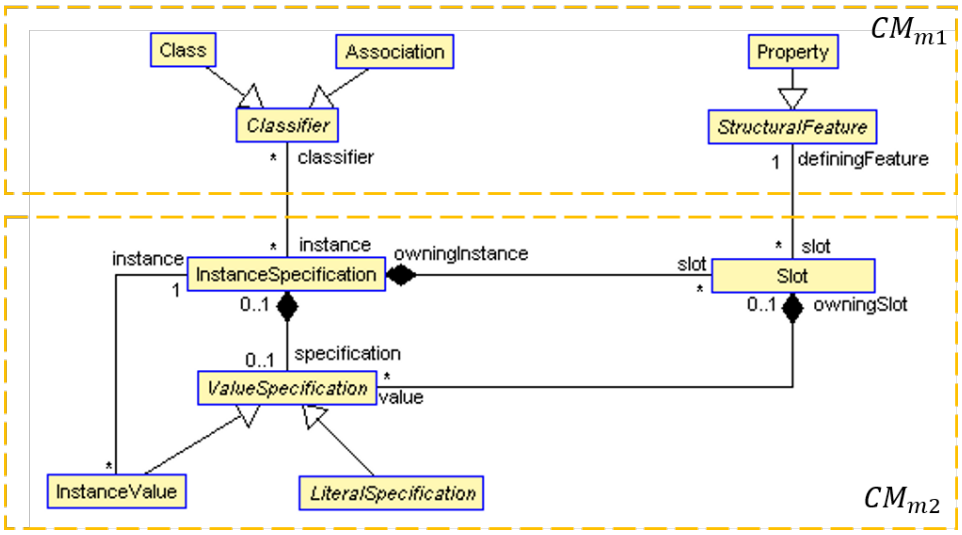


Figure 3.9: Cental UML metamodel elements defining an illustration of the user classes' instances

and `InstanceSpecification`, and the other connects `StructureFeature` and `Slot` metaclasses. These two associations play as a bridge to go from user model elements (upper-level elements) to the corresponding object model elements (lower-level elements) when formulating the level-crossing invariants or queries within our approach.

A metamodel instantiation $\sigma(\mathcal{CM}_m) \Leftrightarrow \mathcal{OM}$ corresponding to the object model at level M0 is automatically generated applying the Algorithm 2. Particularly, our algorithm first visits all objects in the object model. For each object, the algorithm also visits its attributes and values of these attributes. It then visits all the links in the object model. For each link, the algorithm also visits its roles. When the algorithm visits an object model elements e (e can be an object, an attribute, a value, a link or a role), a function, i.e., $o_m, l_m \leftarrow visit(e)$, is called to generate and returns the corresponding metaobject o_m and metalink l_m. Table 3.2 shows the mapping between the object model elements and the related metaclasses and metaassociations, which are the type elements for the generated metaobjects and metalinks

Algorithm 2 Generating the metamodel instantiation corresponding to the object model

Input: A run-time object model $\mathcal{OM} = \sigma(\mathcal{CM}) = (\sigma(\mathcal{C}), \sigma(\mathcal{A}), \Pi)$
Output: $\sigma(\mathcal{CM}_m) = (\sigma(\mathcal{C}_m), \sigma(\mathcal{A}_m), \Pi) \Leftrightarrow \mathcal{CM}$

1: $\sigma(\mathcal{C}_m) \leftarrow \emptyset$
2: $\sigma(\mathcal{A}_m) \leftarrow \emptyset$
3: Initialize the mapping $\mathcal{M}$. //from Table 3.2
4: $E \leftarrow Seq\{\sigma(\mathcal{C}), \sigma(\mathcal{A})\}$
5: **for all** run-time instance $ins \in E$ **do**
6: $Set\{c_m\} \leftarrow \mathcal{M}.getMetaclasses(ins)$
7: $\sigma(\mathcal{C}_m) \leftarrow$ generate metaobjects corresponding to ins
8: $\sigma(\mathcal{A}_m) \leftarrow$ generate metalinks corresponding to ins
9: **for all** $\pi(p_{ins}) := v \in \Pi$ **do**
10: $\sigma(\mathcal{C}_m) \leftarrow$ generate metaobjects corresponding to p_{ins} and v
11: $\sigma(\mathcal{A}_m) \leftarrow$ generate metalinks corresponding to p_{ins} and v
12: **end for**
13: **end for**
14: **return** $\sigma(\mathcal{CM}_m) = (\sigma(\mathcal{C}_m), \sigma(\mathcal{A}_m), \Pi)$

Algorithm explanation:

- In order to generate metaobjects corresponding to an object model element e (a run-time instance, a link or a value assignment), the methods $\mathcal{M}.getMetaclasses(e)$ and $generateMetaobject(e, c_m)$ (see Algorithm 1) are called.
- In order to generate metalinks corresponding to an object model element e(a run-time instance, a link or a value assignment), the methods $\mathcal{M}.getMetaassociations(e)$ and $generateMetalink(e, c_m)$ (see Algorithm 1) are called.

Table 3.2: Mapping between object model elements and metamodel elements.

Object model element	**Related metamodel class**	**Related metamodel association**
Object	InstanceSpecification	$\text{InstanceSpecification}_{\text{instanceSpecification}} - \text{Classifier}_{\text{classifier}}$ $\text{InstanceValue}_{\text{instanceValue}} - \text{IS}_{\text{instance}}$
Link	InstanceSpecification	$\text{InstanceSpecification}_{\text{instanceSpecification}} - \text{Classifier}_{\text{classifier}}$
Attribute	Slot	$\text{InstanceSpecification}_{\text{owningInstance}} - \text{Slot}_{\text{ownedElement}}$ $\text{Slot}_{\text{slot}} - \text{StructuralFeature}_{\text{definingFeature}}$
Value (primary type)	LiteralBoolean/ LiteralInteger/ LiteralReal/ LiteralString/ LiteralNull	$\text{Slot}_{\text{owningSlot}} - \text{ValueSpecification}_{\text{value}}$
Value (user-defined type)	InstanceValue	$\text{Slot}_{\text{owningSlot}} - \text{ValueSpecification}_{\text{value}}$
Role	Slot InstanceValue	$\text{InstanceSpecification}_{\text{owningInstance}} - \text{Slot}_{\text{ownedElement}}$

The lower part of Fig. 3.10 shows a metamodel instantiation corresponding to an object model of the `Employee-Department` class model (see Fig. 3.3). Because of the inherent complexity of the UML metamodel, the object diagram presenting the metamodel instantiation initially includes so many dashed lines and compositions. However, in order to reduce the complexity and also focus on the instance specification, the object diagram has been simplified as shown in Fig. 3.10. This simplification task is automatically tackled by using our tool USE.

3.2.2 Formulating Level-Crossing OCL Expressions

Within our approach, level-crossing OCL expressions are formulated at the M2 level. In principle, every standard two-level OCL expression formulating at the level of user model (M1 level) can be transformed into an equivalent level-crossing OCl expression. Let us have a look at two basic invariants on the use model depicted in Fig 3.11.

Example 3.2.1. *The age of a person must be greater than 18.*

In order to express this constraint with two-level standard OCL, we only need to access the attribute 'age' of the class `Person` as expressed in the following invariant.

```
context Person inv:
    age >= 18
```

This constraint can be formulated with the following level-crossing expression that appropriately effects instance specification.

```
Context Foo inv:
Class.allInstances()→select(
    name= 'Person').instance→forAll(ins|
        ins.slot→select(definingFeature.name='age').value
            →asSequence()→first().integerValue() >=18
)
```

With the intention to keep the original metaclasses unchanged, we add a new class, i.e., `Foo` to the metamodel at the M2 level, and all new level-crossing invariants will be formulated within the context of metaclass `Foo`. Now let us consider another constraint which contains a navigation expression, i.e., navigate from a company to employee who works for that company.

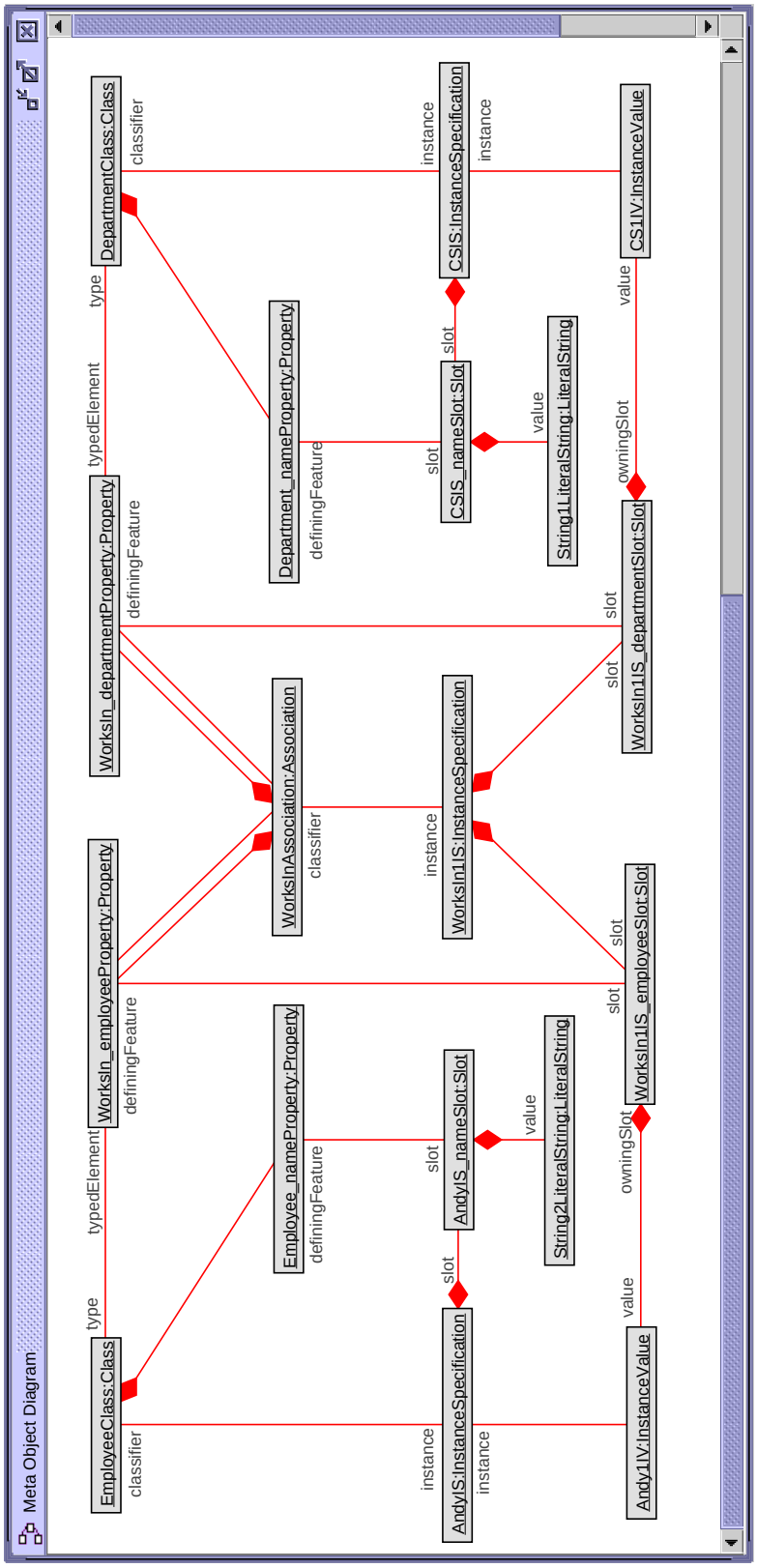


Figure 3.10: An example of meta instantiation reflecting object model

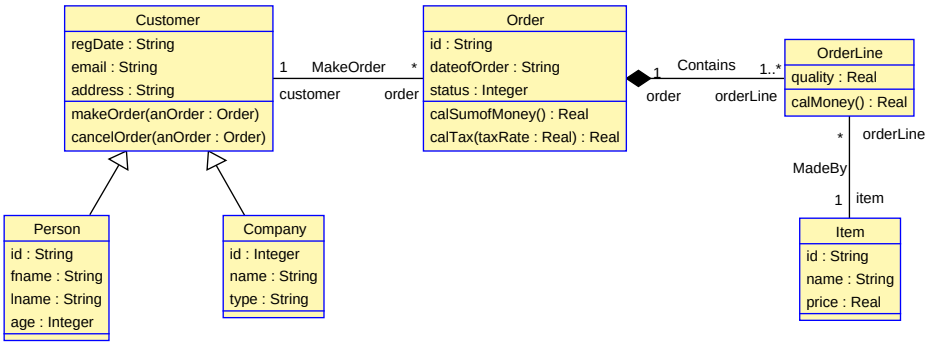


Figure 3.11: Class diagram of an online order management model

Example 3.2.2. *An order can have no more than 10 items.*

This invariant can be simply formulated as the following two-level standard OCL expression:

```
context Order inv:
    orderLine→size() <= 10
```

Again, we can turn the above constraint into an appropriate level-crossing constraint within the context of instance specification.

```
Context Foo inv:
Class.allInstances()→select(
    name= 'Order').instance→forAll(ins|
        ins.notNavigableInstance.owningSlot.owningInstance.slot
        →select(definingFeature.name='orderLine').value
        →collect(iv|iv.oclAsType(InstanceValue).instance)
        →excluding(ins)→size() <= 10
)
```

It can be seen from the two simple examples above, the level-crossing expressions are substantially more complex than the original constraints formulated by standard two-level OCL expression. However, level-crossing OCL is more flexible and it allows developers to formulate a number of constraints and queries that can not be formulated with two-level standard OCL. These kinds of constraints and queries will be discussed later in this section. First of all, we extend the UML meta model at M2 level by adding a few new operations. These operations can simplify level-crossing expressions and therefore increase the understandability and applicability of the proposed level-crossing OCL expression approach.

Access an attribute of an object (instance specification)

```
InstanceSpecification ::
    attribute(attrName: String): ValueSpecification =
    self.slot→select(definingFeature.name=attrName).value
       →asSequence()→first().getValue()
```

This operation returns a value (`ValueSpecification`) of an attribute through the name of the attribute. Note that this operation currently can only work with a single value attribute. By applying this operation, the level-crossing corresponding to the constraint in Example. 3.2.1 can be rewritten as follows:

```
Context Foo inv:
Class.allInstances()→select(
    name= 'Person').instance→forAll(ins|
        ins.attribute('age').oclAsType(Integer) >= 18
)
```

Navigate from an object (instance specification) through a role name

```
InstanceSpecification ::
  navigate(roleName: String): Bag(InstanceSpecification) =
  self.notNavigableInstance.owningSlot.owningInstance.slot
  →select(definingFeature.name=roleName).value
      →collect(iv|iv.oclAsType(InstanceValue).instance)
         →excluding(self)
```

This operation returns a `Bag(InstanceSpecification)` connecting to a instance specification through the role `roleName`. Below is the simplified version of the level-crossing expression specifying the constraint in Example. 3.2.2

```
Context Foo inv:
Class.allInstances()→select(
    name= 'Order').instance→forAll(ins|
        ins.navigate('orderLine')→size() <= 10
)
```

It is clear that standard two-level OCL only supports specifying constraint within the context of one class. Therefore, if one wants to define the same constraint on two different classes, for example, he has to create two different invariants and copy the OCL expression from the first invariant to the second invariant. One advantage of level-crossing OCL compares to two-level standard

OCL is that it allows developers to defined the constraints which are applied to more than one class. In the following, we would like to provide some templates (patterns) to do that.

1. Constrain is applied to a set of classes: `Set{class1,class2, . . .}`

```
let context = Set{class1, class2, . . .} in
Class.allInstances()→select(c|
   context→includes(c.name)).instance→forAll(ins|
       <the OCL expression specifying the constraint>
   )
)
```

2. Constrain is applied to all classes except a class `className`

```
Class.allInstances()→select(c|
   name <> className).instance→forAll(ins|
      <the OCL expression specifying the constraint>
   )
)
```

3. Constrain is applied to all sub-classes of a class `className`

```
Class.allInstances()→select(c|
   superClass.name→includes('className')).instance→forAll(ins|
      <the OCL expression specifying the constraint>
   )
)
```

In the following section, we will utilize level-crossing OCL to formulate a number of queries and invariants which can not be expressed with standard two-level OCL.

3.2.3 Level-crossing OCL Expression Examples

In order to illustrate the expressiveness and usefulness of the proposed approach on level-crossing OCL expression, in this section, we present some examples of level-crossing expression, which are impossible to be formulated with standard two-level OCL. These examples are formulated within the context of the class model depicted in Fig 3.11.

Example 3.2.3. *Query: find the class(es) that has most number of instances.*

```
let max = Class.allInstances()→collect(instance→size())→max()
  in Class.allInstances()→select(c|c.instance→size() = max)
```

Example 3.2.4. *Invariant: values of all Real attributes of instances, e.g.,* `Item::price`*,* `Orderline::quality`*, must have at most 2 positions after decimal point.*

```
Context Foo inv:
let realProperties = Property.allInstances()
    →select(att|att.type.oclAsType(DataType).name = 'Real')
    in realProperties.slot.value→forAll(v|
        let s = v.oclAsType(LiteralReal).value
        in (s*100).floor()/100 = s)
```

Example 3.2.5. *Invariant: the number of customers in each categories, e.g.,* `Person`*,* `Company`*, must be less than 1000.*

```
Context Foo inv:
Class.allInstances()→select(
    superClass.name→includes('Customer')).instance
    →size() < 1000
```

Example 3.2.6. *Invariant: the* `id` *attributes in all classes, e.g.,* `Person::id`*,* `Company::id`*,* `Order::id`*,* `Item::id`*, must be unique.*

```
Context Foo inv:
Class.allInstances()→forAll(c|
    c.ownedAttributes→exists(name='id')
         implies c.instance→isUnique(attribute('id')))
```

3.2.4 Discussion

In the following we would like to point out and discuss some major points of the above level-crossing OCL approach.

1. **Does our approach introduce an actual level-crossing OCL expression formulation?**
   Indeed, our mechanism does not support formulating OCL expressions which actually travel through modeling levels, i.e., from M2 level through

M1 level to M0 level. Therefore, in the technical view, one can say that it is not a real level-crossing expression. However, our approach allows developers formulate the expressions at the M2 level and its semantics is applied for the object model at level M0 thanks to the availability of the semantics of M0 object model at M1 level as a meta instantiation. Thus, in the semantics view, we can say that our approach supports formulating level-crossing expressions.

2. **The redundancy of information**
   The existence of both the object model at the M0 level and a meta instantiation reflecting the object model introduces redundancy. One question can be raised that why we still keep the object model at the M0 level since it can be modeled as a meta instantiation at the M1 level. Dropping the object model at the M0 level can exclude the redundancy. Our intention is extending the original two-level modeling for metamodeling to support more flexible and powerful level-crossing OCL expressions. We want to keep the fundamental of the original two-level modeling in new metamodeling approach, i.e., one can also see the object model as an instance of the user model and formulating OCL expressions at user model level (M1 level) and those semantics are applied for the object model at level M0. Because of the redundancy, keeping the object model and the meta instantiation reflecting the object model always in sync is very important. In particular, when any change on the object model, e.g., additions, updates, deletions any object model element, occurs, we detect and do the instant updates on the corresponding metaobjects or/and metalinks of the added/updated/deleted object model element.

3.3 Related work

There is a number of other proposals, which are related to our work on metamodeling, have been introduced in recent years. The tools Melanee [AG12; AG16] and MetaDepth [LG10] support multi-level metamodeling and facilitates general-purpose languages as well as domain-specific languages. These tools implement "*deep instantiation*" and "*orthogonal classification architecture (OCA)*" [AKG10] mechanisms to support the creation of models with arbitrary numbers of classification levels. They, therefore, allow developers to build models with both linguistic as well as ontological instantiation with

an arbitrary number of meta-levels supporting the potency concept. *Clabject* and *Feature* are two major elements of the linguistic metamodel at L2 level, which define all constructs available in the lower level for populating the ontological instanceOf. Within a linguistic metamodel level, the properties of a real time object at ontological level 0 can be instantiated from different *Clabject* elements at different ontological modeling levels. Each *Clabject* might contain instantiated properties with value or the definitions of properties, which will be instantiated by *Clabjects* or real-time object at lower levels. One significant difference between Melanee and MetaDepth is regarding the metamodeling mechanisms applying in the tools. While Melanee requires strict-metamodeling, MetaDepth offers a special feature exclusive called "*leap potency*", which allows loose metamodeling. Therefore, in MetaDepth, a type element with leap potency n can be instantiated n ontological levels below.

The metamodeling framework Modelverse introduced in [Mie+14] allows for multi-level linguistic modelling hierarchies, as well as modular language design. Deep instantiation and deep characterization are supported in Modelverse using potency.

The work in [BKK16; IGS14] uses F-Logic(FOML) as an implementation basis for multi-level models and model reasoning (querying and validation). A model is defined in FOML using UML class diagram concepts, such as classes, associations, properties, generalization and multiplicities. Rules and constraints on these concepts are written in a logic rule language called *PathLP*. This work also utilizes OCA to support multi-level modelling. That means it supports two instantiation dimension, i.e., linguistic instantiation and ontological instantiation. It also uses several multi-level notations, such as *Clabjects* and *Potency*.

As can be seen, in most multi-level modeling approaches, developers can only work with models in the ontological instantiation dimension. The levels in the linguistic instantiation dimension seem to be fixed and only one or two lowest levels are accessible to developers. These approaches only support defining invariants or queries across levels in ontological instantiation dimension. In contrast to these approaches, our contribution deals with traditional two-level UML/OCL modeling approaches by extending them for meta-modeling in the linguistic instantiation dimension. By adding a metamodel instantiation that reflects and is equivalent to the user model at M1 level, our approach now supports the accessibility to three modeling levels. Therefore, one can exploit the added meta-data for model analysis and exploration with reflective con-

straints. Our approach also supports to define level-crossing OCL expression through linguistic instantiation levels. One commonality between our work and these above-mentioned approaches is the introduction of elements in the middle level that have both type and instance facets.

The idea of copying the M2-model instance to lower levels in the MOF meta-model architecture and exploiting it for reflective constraint writing is also presented in [Dra16]. In that paper, the meta instance is added to the M0 level, together with the run-time instances, through instantiation and reification processes. Adding elements to the M0 level is a major difference between the work in [Dra16] and our approach because, in our work, the metamodel instantiation of the user model is generated and added to the M1 level. Therefore, in order to write a reflective constraint or query with our approach, we only need to go from the M2 level to the M1 level. This means we do not need to extend the OCL for writing reflective constraints or queries.

One attempt to produce a multi-level modeling representation within a two-level modeling tool is in [Gog15]. In this paper, the authors introduce a method to represent the connection between levels by using two equivalent models at the middle level, one class model and one object model instantiated from the upper level. This approach only shows the experimentation of visualizing multi-level models in a two-tevel modeling tool without actually extending two-level modeling to multi-level modeling.

Chapter 4

Quality Assurance through Metrics Definition and Bad Smells

This chapter extends the work in [DG18a], which utilizes the meta modeling architecture presented in Chapter. 3 for model metric measurement and quality assurance. Particularly, Section 4.1 represents an approach for metric definitions. UML class model metrics are defined by OCL as the operations of the classes in a new and separate package at the topmost level. A further contribution of our work is to propose a complete process for model quality assessment with pre-defined metrics, which is demonstrated in Section 4.2. In this assessment process, developers can achieve information about the quality of their design based on a metrics threshold configuration, and they can detect problems in their model. Finally, in Section 4.3 we present a work on smell definition and detection. UML model design smells are specified by meta-level OCL expressions and stored in a separated file. The existence of these smells will be inspected within our three-level modeling architecture.

4.1 Metrics Measurement

In the previous chapter, we have shown that our approach for metamodeling now supports full accessibilities to the metamodel. As a result, extracting the internal information for computing the metrics of the user model is possible with our three-level modeling approach. In this section, we will present how the *metamodel* is exploited for metrics measurement. Particularly, the metrics

are defined separately from the metamodel as operations of two newly added classes, one for class scope metrics and one for model/package scope metrics. These two classes are encapsulated in a new package, named `Metrics`, at the same level of the *metamodel*, i.e., M2 level. In Chapter 5, we present an evaluation to demonstrate the applicability and feasibility of the metric measurement approach.

4.1.1 Class Scope Metrics

Class scope metrics are the metrics which measure the structural properties of classes in a model. In other words, the context of these metrics is class. Some examples of class scope metrics are NOM (Number of Local Methods) [LH93], NOC (Number of Children) [CK94], DIT (Depth of Inheritance Tree) [CK94]. In order to measure class scope metrics, first a class, named `ClassMetrics`, has been added to the `Metrics` package. This class is connected to the *metaclass* `Class` by an association as shown in Fig. 4.1. Therefore, from a class, one can access the metrics through the role name of this navigable association, i.e., `metrics`. Following this approach and in order to measure metrics for other elements of the *metamodel*, one can extend this option by creating a corresponding metric class in the Metrics package and an association between the created metric class and the element to be measured. Each metric is declared as an operation of the class `ClassMetrics` and the body is defined as an OCL expression (relying on the class diagram in Fig. 3.2).

These operations can access the *metamodel* in order to collect data for metrics calculation through the navigable association to the *metaclass* `Class`. One additional step must be performed in order to make these metrics ready for use. For each class, one *metaobject* of the corresponding *metaclass* `ClassMetrics` and a link are automatically added to the M1 level. Consequently, for every class in the *user model*, a pair of *metaobjects* (class, classmetrics) and a link between them are now available for metrics handling.

To demonstrate the idea, we take the DAC (Data Abstraction Coupling) metric as an example. This metric is defined in [LH93] as follows:

Definition 4.1.1. *Data Abstraction Coupling (DAC) is the number of attributes in a class having another class as their type.*

The DAC metric is a coupling metric. It shows how this class is coupled with other classes in the model through data declaration. A higher value of

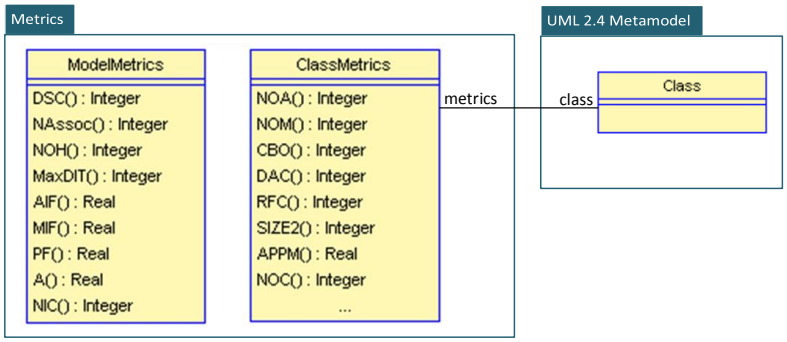


Figure 4.1: Metrics package and relationship to UML 2.4 metamodel.

DAC indicates more complexity in data structures, and increase the dependency of the class under consideration and other classes in the model. Note that from the original definition, DAC does not include the attributes that are inherited from superclasses, only the attributes declared within the class itself are considered. The listing below shows the OCL expression definition of the DAC metric in the context of the metaclass `ClassMetrics` as an operation.

```
DAC(): Integer =
    userdefinedTypeAttributes()→size()
```

where `userdefinedTypeAttributes()` is an auxiliary function that selects an OrderedSet of user-defined type attributes of the corresponding class. We have defined a list of auxiliary functions and these functions can be called within the definition of a metric.

```
userdefinedTypeAttributes(): OrderedSet(Property) =
    self.class.ownedAttribute
      →select(att|att.type.oclIsTypeOf(Class))
```

Within our approach, retrieving a metric of a class from the user class diagram can be done by the following expression template:

<*Name of class*>.**metrics**.<*Name of the metric*>.

This is a natural way of retrieving a class metric, because a metric can be seen as a property of a class. Fig. 4.2 presents an example of a UML class metric measurement.

The left part of Fig. 4.2 is a simple class diagram containing major features of an object-oriented design, such as classes, attributes, operations, inheritance, and polymorphism. In the right upper part, the expression to retrieve

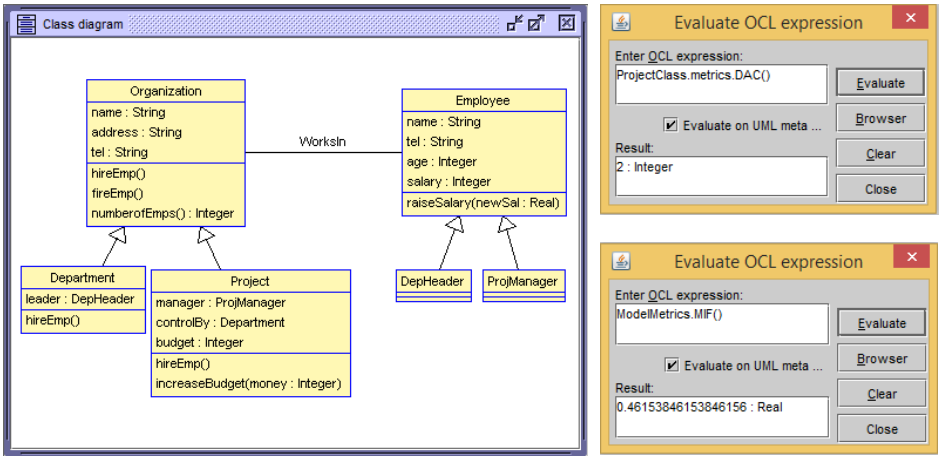


Figure 4.2: Example for metrics evaluation.

the DAC() metric of the class `Project`, i.e., `ProjectClass.metrics.DAC()`, and the value, i.e., 2, are shown. The value 2 is caused by two attributes with user-defined type in the class `Project`, i.e., *manager* and *controlBy*. Note that Fig. 4.2 is a screenshot from the tool USE since our approach has been successfully integrated into USE. The value of the DAC metric can be used to analyze and predict the external quality of the software product to be implemented. DAC is a coupling metric: the higher the value of DAC, the less independent the class. Therefore, the DAC metric should not be too high, in order to improve the modularity and encapsulation of the design. When the coupling of a class is high, changing other classes is much more sensitive to the class under consideration, and therefore, the cost for maintenance in later phases is increased.

4.1.2 Model Scope Metrics

Model scope metrics are the metrics which measure the structural properties of whole model. In other words, the applicable context of the these metrics is model. DSC (Total number of classes in the design) [BD02], AHF (Attribute Hiding Factor) [AM96], MaxDIT (Maximum depth of inheritance) [Gen02] are some examples of model scope metrics. As can be seen in Fig. 4.1, all the model scope metrics are encapsulated in an isolated metaclass, i.e., `ModelMetrics`, in the Metrics package. Each metric is defined as an operation of the class `ModelMetrics` to measure the overall quality in the context of a class model, such as metrics for complexity, coupling, inheritance, polymorphism, or size. They are also formulated as OCL expressions at the meta level, and, within

expressions, the pre-defined class scope metrics and the auxiliary functions can be called as well. In the following, an example of the definition and the use of a class model scope metric is presented, i.e., the MIF metric. The MIF metric is defined as follows [AM96]:

Definition 4.1.2. *The Method Inheritance Factor (MIF) is defined as a ratio between the total number of inherited methods in all classes of the system under consideration and the total number of available methods (locally defined within the class and inherited from other classes) for all classes.*

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

where:

- TC = *total number of classes*
- $M_i(C_i)$ = *number of inherited methods in class* C_i
- $M_a(C_i) = M_d(C_i) + M_i(C_i)$ = *number of locally defined methods + number of inherited methods in class* C_i

The following OCL expression is the formal definition of the MIF metric. It is defined as the body of an operation of the class *ModelMetrics*.

```
MIF(): Real =
    (Class.allInstances()→collect(c|c.metrics.NMI())→sum())/
    (Class.allInstances()→collect(c|c.metrics.NOM() +
   c.metrics.NMI())→sum())
```

In the above expression, two class scope metrics, i.e., NMI(Number of Methods Inherited) and NOM(Number of Local Methods), are called to yield the number of inherited methods and the number of locally defined methods in a class, respectively. The definition of these two class metrics can be found in the Appendix. The right lower part of Fig. 4.2 shows how the MIF metric can be achieved with our approach. The expression `ModelMetrics.MIF()` give us the corresponding value of the MIF metric of the class diagram on the left, i.e., 0.4615. This metric can be used to measure the level of reuse. A high value indicates a high level of reuse. According to [AM96] the value should not be too high. They proposed the value of the MIF metric should have an upper bound and should be smaller than 0.7 resp. 0.8.

3. Definition of new metrics: It could be that designers may find it difficult to define new metrics as OCL expressions over the UML metamodel on their own, and it could be an error-prone task for them. To overcome this drawback, we offer an interactive process for developing new metrics thanks to the availability of the *metamodel*, the *metamodel instantiation*, and the option for interactive OCL expression evaluation on the *metamodel* (as shown in the right-hand side of Fig. 4.2). A new metric can be the technical realization of what is typically called a "design smell".

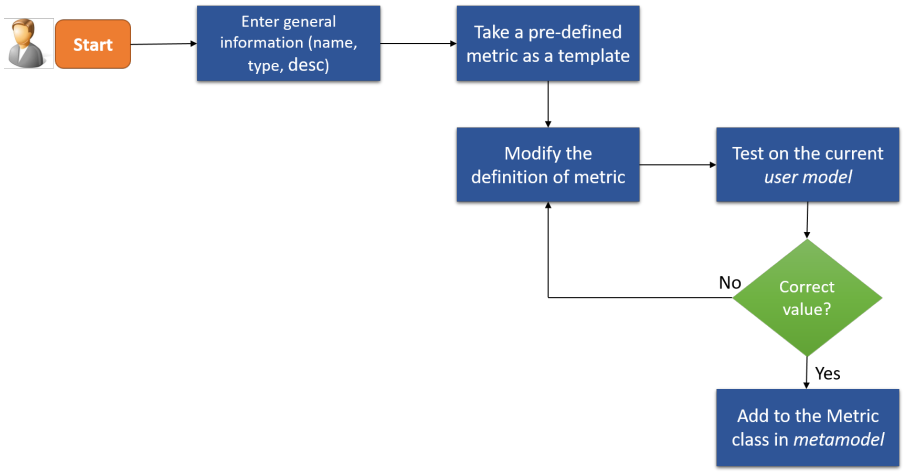


Figure 4.3: Defining a new metric.

Fig. 4.3 shows the interactive process for a new metric definition. In particular, one can take an already defined metric as a template, and then modify the corresponding OCL expression (for example, one could change `Class` to `AssociationClass` in the definition of the DAC metric). The new metric, i.e, the new OCL expression over the UML metamodel, can be checked for syntax and tested on the generated *metamodel instantiation* of the current *user model*.

This process can be iterated until a desired new metric has been developed. Finally, the successful new metric is added to the Metrics package as an operation of the class `ClassMetrics` or the class `ModelMetrics` after giving it a name and then this metric is ready to be used in the future.

For example, if one wants to define a new metric, namely MIF1, that calculates the average number of the inherited methods per class in a *user model*. It can be done by taking the OCL expression definition of the MIF metric as a template, and then alter it by replacing the later part by `ModelMetrics.DSC()` to get the total number of classes in the user model. This new OCL expression can be tested on the user model, e.g., the model in Fig. 4.2, as shown in

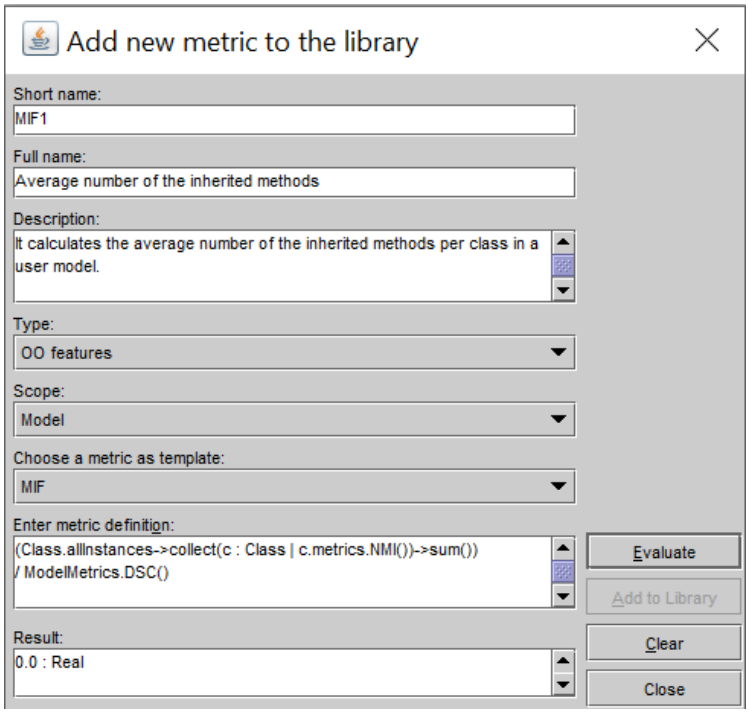


Figure 4.4: Example of defining a new metric.

Fig. 4.4.

4.2 Quality Assurance with Metrics Threshold

As discussed in Sect. 2.5, metric measurement can be used as an early indicator for software quality. Beside that, metrics can be exploited for design assessment or evaluation by setting different *thresholds* for different metrics. For example, one could set the possible maximum value of the DAC metric as 4, i.e., every class in the model should not have more than 4 attributes that have another class as their type. Or setting the value of the MaxDIT metric of the overall design between 1 and 5 means the model should have at least one inheritance tree longer than 1 and no inheritance tree longer than 5. Actually, the proposed metric threshold values may come from empirical studies (some of them have been indicated in the survey in [GPC05]) or can be set by an experienced chief designer based on the requirement of a project. This assessment could not only help designers to control the quality of their work, but could help them to detect problems in the model as well. In this section

we present an approach that supports designers (a) in the definition of a list of metrics, both class scope and model scope metrics, (b) in the specification of upper and lower thresholds, and (c) in the automatic evaluation of the model along the stated threshold settings.

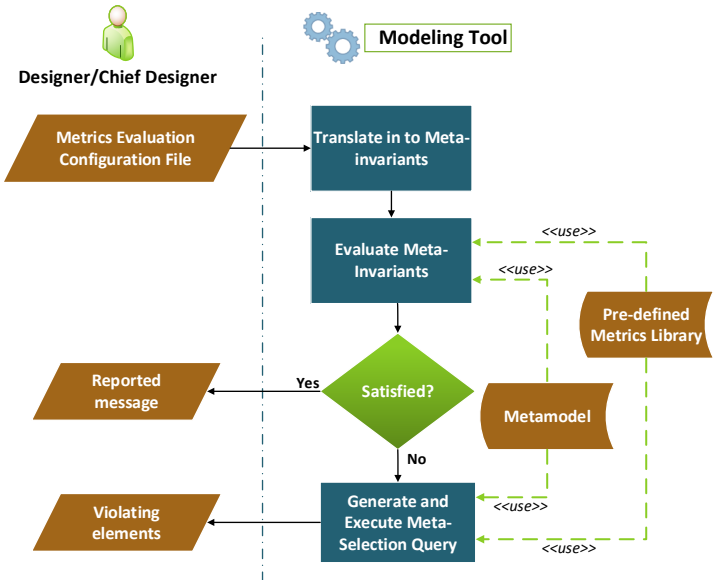


Figure 4.5: Workflow of model quality assessment with metrics.

Fig. 4.5 shows the workflow of the model quality assessment using a metric interval configuration. As can be seen from the workflow, the input of the assessment is a *metric evaluation configuration*. This configuration contains a set of desired metric threshold settings. These settings should be stated by an experienced chief designer. Each setting must contain the following information: metric scope, name of the metric as well as the lower and upper thresholds. Because the values of metrics are always either integer or real, the value of the lower and upper thresholds must be integer or real as well. The following listing is an example of a metric evaluation configuration, and it will be utilized for assessing the models with the combination of the metrics values.

```
0 NOM 4 15
0 DIT -1 5
0 NOA -1 10
0 NOC -1 4
1 DSC 5 20
1 MIF 0.2 0.8
```

Please note that, each line in this configuration contains the setting for one metric. In each line, from the left to the right, the corresponding values of the metric scope, the name of the metric, the lower threshold and upper threshold are stated. The first value 0 indicates that the corresponding metric is a class scope metric. The -1 value for lower or upper threshold settings means the corresponding metric is only upper bounded or lower bounded, respectively.

In the next step, the metric settings will be translated into a set of *meta-invariants*, one meta-invariant for each metric setting. We use the notion 'meta-invariant' because they are formulated on the metamodel (M2) and their evaluation is applied on the level of the user model (M1) (this rule is also applied to the later introduced type of query in the workflow, i.e., *meta-selection query*). The translation is based on the following strategy:

- **For class scope metrics**: we have to check the setting on all classes in the model. To achieve that, we propose the following template for evaluating the meta-invariant:

  ```
  Class.allInstances()→forAll( c|
    <the metric condition for class c>)
  ```

  where the metric condition is an expression for the comparison between the metric value and the lower and upper thresholds. If the lower or upper threshold value is missing (set as -1), the metric condition expression will not contain the comparison expression of the missing value. In the condition, the corresponding metric operation of class c is called. For example, the translation of the first setting in the above configuration file example, i.e., `0 NOM 4 15`, gives us the following meta-invariant:

  ```
  Class.allInstances()→forAll(c|
    4 <= c.metrics.NOM() and c.metrics.NOM() <= 15)
  ```

- **For model scope metrics**: because the context of these metrics is the complete model, the translated meta-invariants are simply basic comparison expressions between the metric value and the lower and upper thresholds. For example, the following listing is the meta-invariant which is translated from the last setting in the above configuration example, i.e., `1 MIF 0.2 0.8`

  ```
  0.2 <= ModelMetrics.MIF() and ModelMetrics.MIF() <= 0.8
  ```

Next, these meta-invariants must be evaluated on the metamodel level. This step can be performed by a supporting tool, since many modeling tools now support parsing and evaluating OCL expressions, for example, the tool USE. The result of this step is either an indication for satisfaction or violation of the metric threshold settings (meta-invariants), i.e., *true* or *false*. In case the evaluation for a class scope metric is unsatisfied (the evaluation is *false*), the designer may want to know which elements (classes) violate the setting. To find the violating classes in the model, we introduce a template for auto-generating a corresponding meta-selection query. The result of executing this meta-query is a set of classes that violate the threshold setting. The template for this kind of query is as follows:

```
Class.allInstances()→select(c |
  not <the metric condition for class c>)
```

The metric condition expression is constructed as mentioned above. For example, if the evaluation of the setting `0 NOM 4 15` shows the result *false*, the following OCL meta-query is automatically generated in order to find the violating classes, i.e., classes that have a number of local methods less than 4 or more than 15:

```
Class.allInstances()→select(c|
  not (4 <= c.metrics.NOM() and c.metrics.NOM() <= 15))
```

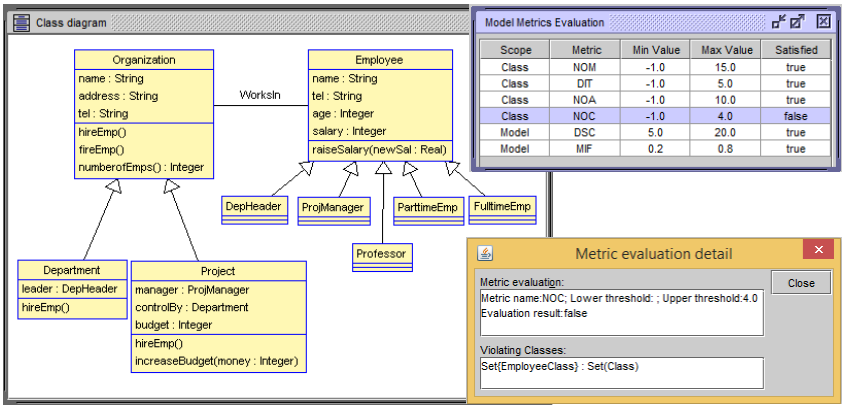


Figure 4.6: Example of model quality assessment with metrics.

In order to show how the above proposed approach for model quality assessment with metrics can be successfully integrated into a modeling tool, Fig. 4.6 presents a simple example of applying this approach in the tool USE.

The class diagram of the model under consideration is presented on the left. Following the process shown in Fig. 4.5, using the metric configuration which is

presented at the beginning of this section, the settings in the configuration are first translated into meta-invariants and then evaluated within USE. The right upper window shows the evaluation result. In particular, only the setting of the NOC metric, i.e., `NOC -1, 4`, is false. To explore which elements cause the NOC metric setting to be unsatisfied, the last step of the process is performed by double clicking on the row of the NOC metric setting. As a result, the violating class, i.e., the *Employee* class, is found and presented in the right lower part of Fig. 4.6. The *Employee* class has five direct sub-classes, which is out of the bounds of the NOC metric threshold setting.

4.2.1 Model Level Metrics Threshold: Preliminary Review

It is clear that knowing reference values of metrics increases the usefulness of metrics. Additionally, in Section 4.2, we have presented an approach to assess UML models with a set of metric thresholds. The result of the assessment relies on the input metric thresholds. As mentioned before, the input metric thresholds may come from empirical studies or can be set by an experienced chief designer based on the requirement of the project under consideration. Therefore, the knowledge of the common value of software metrics using in practice is essential. In this section, we would like to gather and present model level object-oriented software metric thresholds proposed from the literature. For each study, the methodology applying to extract metrics threshold as well as the data-source being used for the study are also represented. We believe this catalog can be a reliable reference source for metrics threshold. Actually, there are many approaches that have been presented to compute software metrics threshold values. We select studies to present in this section base on the following requirements.

- It must be conducted based on empirical research and not on opinions. Our aim is studying the common/good metric intervals from practice, therefore, we only consider the result from empirical studies.

- The result of the study is presented as a numeric or interval. With this preliminary review, we would like to provide a reliable reference for setting the input of the quality assessment process presented in the previous section. Because the input is a numerical interval of metric values, thus only researches which show the result as a numeric or interval

are examined.

- Threshold values are extracted by a statistic analysis. Using statistic analysis from a large data set to compute the metric threshold values is our consideration, since we want to study the common/good metric intervals from practice.

- It contains at least one model level metric. We concern about the metrics which are applicable to the model level. Therefore, the studies which deal with only code level metrics, e.g., LOC (Line of Code), LCOM (Lack of Cohesion of Methods), RFC (Response for a Class), are excluded from our preliminary review.

We firstly searched for related works have been published in the literature using proper search strings, i.e., *("software" OR "source code") AND ("metrics") AND ("threshold")*. From a number of found result, the studies which were not satisfied the above requirements are excluded. Finally, seven primary studies have been selected. Table 4.1 represents these studies together with

Table 4.1: A catalog of model level metric thresholds

Source	Metric Thresholds				
	DIT	CBO	NOC	NOM	NMO
[Sha10]	NA	0-9	NA	NA	NA
[Sha+10]	NA	0-13	NA	NA	NA
[HGW11]	NA	0-5	NA	0-20	0-3
[Fer+12]	0-2	NA	NA	NA	NA
[JTS14]	0-2.5	0-13.6	0-6.4	NA	NA
[JTS14]	0-3.2	0-15.2	0-3.8	NA	NA
[FS15]	2/2-4/4	NA	1/1-3/3	6/6-14/14	2/2-4/4

the metric thresholds they introduced. The first column is the reference source of the studies. NA (not available) indicates that the metric is not included in the corresponding study. Table 4.2 shows the dataset and method are used in each study. In the following, we will discuss the computation methods, the used dataset and the derived metrics of these researches.

1. **Threshold computation methods.** Statistical analysis, metric analysis and machine learning techniques have been utilized for metric derivation. In particular, [FS15] and [Fer+12] use a data distribution analysis

Table 4.2: Methods and Dataset using for threshold derivation

Source	Dataset	Method
[Sha10]	Two versions of Eclipse	Logistic regression model
[Sha+10]	Three versions of Eclipse	Receiver Operating Characteristic (ROC) curves
[HGW11]	Eclipse Java projects	Machine learning and data mining techniques
[Fer+12]	Java open source systems	Data distribution analysis
[JTS14]	Java/C++ projects	Metric distribution analysis
[FS15]	Eclipse Java projects	Data distribution analysis

method to classify the studied metrics into three ranges: Good, Regular, Bad. For example, [FS15] recommends the DIT metric can be 2/2-4/4 as Good/Regular/Bad categories as shown in Table 4.1. In [Sha10] metric thresholds are calculated by a statistical method using a logistic regression model. Receiver Operating Characteristic (ROC) curves is employed for threshold identifying in [Sha+10]. [HGW11] use machine learning techniques, i.e., rectangle learning algorithm, to calculate metric thresholds from a dataset of open-source software written in C++ and Java. [FS15], on the other hand, gathers a dataset of metric distribution presented in the literature and then statistically analyze them to extract the low and high values of the selected metrics.

2. **Dataset used for threshold derivation.** As can be seen from the Table 4.2, all the studies presented in this section calculate and extract metric thresholds for source code metrics. Java or C/C++ software projects were used as the dataset for analyzing and calculating thresholds. Some source code level metrics are applicable to the design level as presented in Table 4.1, however studies of metric thresholds analysis and extraction using a dataset as a set of design models are still needed since some metrics, such as CBO, can only be approximately calculated at the model level. Therefore, statistical information about model metrics using in practice is useful for modelers or designers. In Section 5.3, we present a metric measurement evaluation on a large dataset of UML models in practice. A descriptive statistic of a number of model level metrics is also presented in this evaluation.

3. **Derived metrics.** CK object-oriented metrics [CK94], i.e., DIT (Depth

of Inheritance Tree), WMC (Weighted Methods per Class), NOC (Number of children), CBO (Coupling between objects), RFC (Response For a Class), LCOM (Lack of Cohesion of Methods), are the most commonly studied for the metric thresholds. However, among them, only DIT, CBO and NOC metrics are applicable to the model level (can be calculated without the information of the software implementation). Therefore, we only show the thresholds of DIT, CBO and NOC metrics in Table 4.1. Some other metrics, such as NOM (Number of local Methods), NMO (Number of Methods Overridden), LOC (Lines of code), Ca (Afferent coupling), Ce (Efferent coupling) are also included in some studies. However, only NOM and NMO metrics are presented in the catalog because the others do not apply to the model level. Interestingly, only class scope metrics are considered. To our knowledge, no empirical study about model scope metrics has been introduced in the literature. A study about the distribution and thresholds of some model scope metrics, such as Attribute inheritance factor (AIF), Method inheritance factor (MIF) or Abstractness (A) could provide a useful reference about the good or common level of object-oriented mechanism in system designs. Such a study will be represented in Section 5.3. In which, a number of selected model scope metrics, as well as class scope metrics will be computed on a large dataset on UML models. The statistic data of the metrics is then analyzed for good or common values of the practical models.

4.3 Design Smell Detection

As mentioned in Section 2.5.3, design smells are flaws or issues which may exist in a model. They have negative impacts on the development of software as well as the quality attributes of the software [BV10; Bet+19]. Detecting smells and resolve them at the design phase can improve the quality of the design and therefore, improve the quality of software in general. For this reason, approaches to detect design smells and to resolve them if possible is necessary. On the other hand, in the previous chapter, we have shown that our approach for metamodeling now supports full accessibilities to the metamodel. Consequently, by extracting internal information, one can detect bad smells on the user model. Some approaches have been introduced in order to deal with the smell detection and refactoring. The work in [AGO12] has presented a unified method for the definition and checking of UML class di-

agram quality properties. For example, syntactic issues, best practices and naming issues are treated as a plugin for EMF. Another approach [LGL14] has been using the *mmSpec* language, integrated into the tool *metaBest* to specify and check quality properties on UML models. [Bas+16] introduced an approach for quality assessment of modeling artifacts (metamodels, models, model transformations) by supporting hierarchical quality model definition using OCL and evaluated modeling artifacts based on metric measurements. Another tool, SDMetrics [Wüs], works with models stored in the XML Metadata Interchange (XMI) format, and metrics and design guidelines are defined in the form of XML-based specifications.

In this section, we will show how our three-level modeling approach presented in Chapter 3 can be applied for smell detection on UML class models. Firstly, a general formalization of design smells is presented. Our aim is to detect the existence of smells in a design and to find the "smelly elements" (design elements, which cause the smells).

Secondly, we present a work on utilizing our metamodeling architecture presented in the previous chapter to detect design smells defined by our proposed formalization. An evaluation of our work on design smell detection is presented in Section 5.4 in terms of its expressiveness, feasibility and usefulness.

4.3.1 Motivating Example

This section presents a motivating example, which will be used to illustrate the definition of the design smell presented in the next section.

This simple model describes a relationship between employees and a company. It contains six classes, one associations and a few generalizations. Even if the model is small, we can manually detect several issues on this model.

1. The name of the class *manage* does not start with a capital letter. That violates the PascalCase convention and several naming guidelines recommendations [Amb05].

2. All subclasses of the class *Employee* has the *salary: Real* attribute. The *salary* attribute should be declared in the superclass *Empoyee* and all its subclasses inherit this attribute from the superclass.

3. The generalization $FulltimeEmp \prec Person$ is redundant.

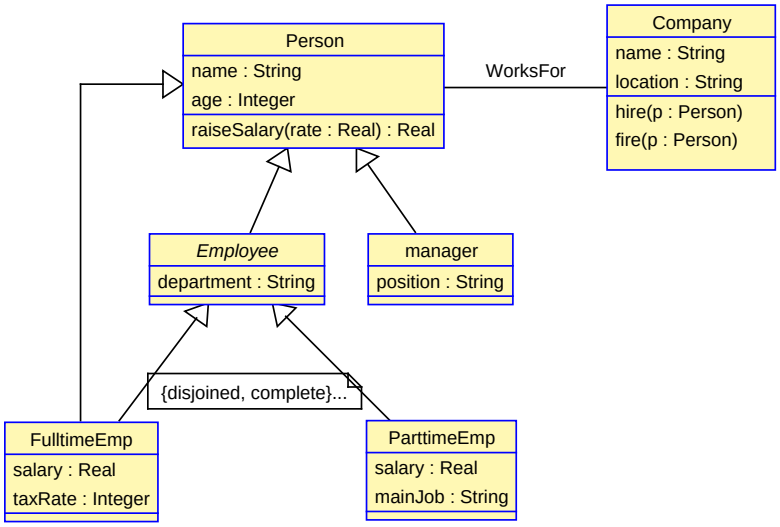


Figure 4.7: A motivating example of smell detection.

4. Abstract class *Employee* is a subclass of the concrete class *Person*. Abstract class should be a subclass of only abstract classes.

In the next sections, we represent how the design smells such as those above are automatically detected.

4.3.2 Design Smell Formalization

This section presents a formal definition of smells of UML models. A smell S_i is a flaw or issue which may exist in UML modelsand it can be defined as follows.

$$\mathcal{S}_i = (\mathcal{E}_i, \pi_i, \mathcal{K}_i, \mathcal{L}_i)$$

where:

- $\mathcal{E}_i$ is the context of the smell.
- π_i is the smell condition.
- $\mathcal{K}_i$ is the smell kind.
- $\mathcal{L}_i$ is the severity of smell.

Context of Smell

The context of smell $\mathcal{E}_i$ is the type of metamodel element causing the smell. It can be any type of metamodel element: Class, Property, Operation, Association or Generalization. For example, the context of the smell (1) from the

example in Section 4.3.1 is *Class* and the context of the smell (3) is *Generalization*.

Smell Condition

Smell condition π_i is a logic expression (the evaluation returns True/False). This describes the condition applying on an element $e : \mathcal{E}_i$. If there exists an element $e : \mathcal{E}_i$ in a class model $\mathcal{CM}$ which $\pi_i(e) = true$, then the class model $\mathcal{CM}$ is determined having smell $\mathcal{S}_i$ and $e : \mathcal{E}_i$ is a "smelly element" of the smell $\mathcal{S}_i$. For example, the condition of the smell (4) from the example in Section 4.3.1 can be expressed by natural language: "Class $e \in \mathcal{C}$ is abstract and one of its superclass is not abstract". And the *EmployeeCompany* example model is determined as a faulty model regarding the smell (4) since there exists a class, i.e., *Employee* class, violating the smell condition.

Kind of Smell

Kind of smell $\mathcal{K}_i$ can be considered as the type of issue that the smell dealing with. Naturally, smells can be classified into categories depending on their nature and relevance as well as the experts' experiences. In this formalization, we classify smells into five groups of kinds.

- **Trivial**: A trivial smell is considered as a syntax error and it can be checked by most of modeling tools. For example, smells such as *"A class has no name"* or *"An attribute has no type"* are trivial smells.

- **Syntactic**: A syntactic smell is a kind of smell which is already specified in the UML metamodel as invariants, e.g., the constraint *"Generalization hierarchies must be directed and acyclical"* [Obj11b, p. 93].

- **Essential property**: An essential property is considered as a design error. A model is considered incorrect or incomplete if it has a essential property smell. "An abstract general classifier has one specific class only" is an example of essential property smell.

- **Best practice**: A smell of this type indicates a violation of design quality guidelines. Design quality guidelines are the suggestions recommended by MDE experts to improve the quality of class models. "A generalization is redundant" is an example of best practice smell.

- **Metric**: This kind of smell is a type of the best practice smell, which directly related to metric. For example, "A class has too many direct children (10 by default)" is a metric smell.

- **Naming conventions**: This kind of smell is a type of the best practice smell regarding naming the features in the model. For example, every class should be named in pascal-case, attributes and operations should be named in camel-case.

Severity of Smell

The severity of smell $\mathcal{L}_i$ is the acceptability level of smell. If the severity of a design smell is *critical*, it means the smell must be fixed, otherwise, it is impossible to create a run-time system of the design under consideration. *Medium* severity level means the smell should be fixed, and the *low* level means that considerations of the designer must be taken into account to decide whether the smell is really an issue or not. For example, the severity of smell "Cycle reflexive association" is *critical*, smell "Redundant generalization paths" has severity level *medium*, and smell "A class without method" is considered as *low* severity.

Smell Instances

In a user model, there may exist several different smells. And regarding a single smell $\mathcal{S}_i$, there may exist distinct model elements e_i which violating the definition of the issue. Each smell element e_i is called an instance of $\mathcal{S}_i$. The set of smelly model elements corresponding to a particular smell $\mathcal{S}_i$ in a class model $\mathcal{CM}$ is the instances of $\mathcal{S}_i$ in $\mathcal{CM}$, denoted by $\sigma_{\mathcal{S}_i}(\mathcal{CM})$.

Let $\mathcal{S}_i = (\mathcal{E}_i, \pi_i, \mathcal{K}_i, \mathcal{L}_i)$ is a design smell. The instances of $\mathcal{S}_i$ in a class model $\mathcal{CM}$ can be formally defined as follows.

$$\sigma_{\mathcal{S}_i}(\mathcal{CM}) = \{e_j\}$$

where $e_j : \mathcal{E}_i$ is a user model element, and $\pi_i(e_j) = true$.

4.3.3 Design Smell Detection Utilizing Metamodeling

In the previous section, a formal definition of design smells as well as smell instances have been introduced. In this section, we show how the three-level

modelling architecture will be utilized for design smells detection and the smell instances computation.

Define a Smell Library

Firstly, a set of design smells that we want to detect are defined following the formalization presented in the previous section. This smell library is stored in a separated file. In this work, design smells are stored in an external XML file.

```
<DesignSmell id ="">
   <Name>. . .</Name>
   <Definition>. . .</Definition>
   <Context>. . .</Context>
   <Desc>. . .</Desc>
   <Type>. . .</Type>
   <Severity>. . .</Severity>
</DesignSmell>
```

As presented in the previous chapter, our proposed three-level modeling architecture supports full accessibilities to the metamodel and the data of the user model is now also available at the metalevel for reflective querying. Therefore, in this work, we define the condition of smell as a meta-level OCL boolean expression. This meta-level OCL expression now can be evaluated thanks to the three-level modeling architecture. And by evaluating this meta-level OCL expression, the existence of the corresponding smell on a user model will be discovered. To show examples, we consider the smells from the example in Section 4.3.1. The condition for smell (3), i.e., "*Redundant generalization path*", can be specified as the following OCL expression:

```
Generalization.allInstances()→exists(g|g <> e
   and g.specific = e.specific
   and (g.general.allParents()→includes(e.general)
       or g.general = e.general)
)
```

And the below OCL expression is the condition of smell (4), i.e., "An abstract class is a subclass of a concrete class".

```
e.isAbstract and e.superClass→exists(c|not c.isAbstract)
```

Smell Detection

This step checks the existence of smells from the predefined smells library. Specifically, each smell in the library will be checked on the model under consideration by evaluating a boolean OCL expression based on the following template:

```
<Context of the smell>.allInstances()→exists(e|
  <smell definition>)
```

If the evaluation of this OCL expression on the meta instances corresponding to the user model returns "*true*", that means there exists the consideration smell in the user model. Otherwise, the user model is determined to have no smell. For example, the smell *"An abstract class is a subclass of a concrete class"* has the context `Class`; therefore the OCL expression used to evaluate the smell is

```
Class.allInstances()→exists(e|
  e.isAbstract and e.superClass→exists(c|not c.isAbstract))
```

Smell Instances Computation

In the case there exists a smell on the model (the evaluation of the OCL expression on the metamodel level returns true), the list of user model elements that cause the smell can be specified by executing an OCL query based on the following template:

```
<Context of the smell>.allInstances()→select(e|
  <smell definition>)
```

It is clear that regarding the performance, smell instances calculation (which selects all smelly elements) is costly than smell detection (which only needs to check for the existence of a smelly element). Therefore, by detecting the existence of smells first and then compute the smell instances when needed, we can improve the performance of the process, especially when the predefined library contains a large number of smells.

The smell detection approach has been successfully integrated in to the tool USE. Fig. 4.8 shows the result when we utilize our tool to detect smells on the simple model shown on the left. Particularly, from a predefined library of smells, our tool detects four smells in the model and the instance of a smell, for example, the *"Redundant generalization paths"* smell, is computed and

displayed a separate window. They are also highlighted in the class diagram as can be seen in Fig. 4.8.

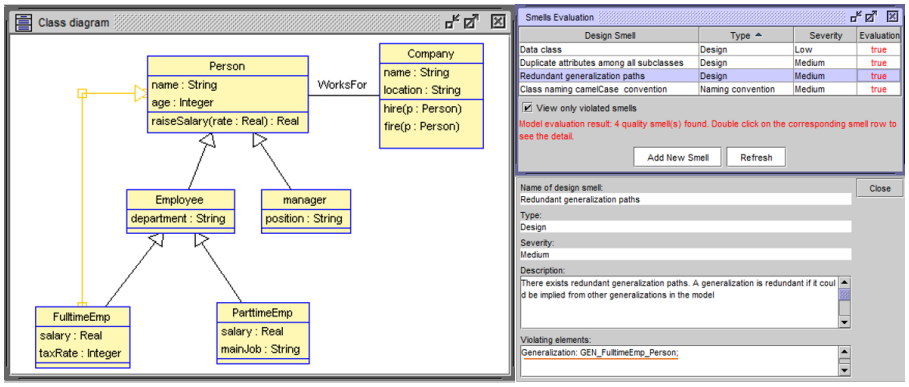


Figure 4.8: An example of a smell detection.

4.4 Related Work

Class model metrics definition with OCL: In [Bar+02] an approach for object-oriented design metrics definition on the UML metamodel with OCL was introduced for the first time. The authors defined metrics as post-conditions of additional operations in the UML 1.3 metamodel. Continuing this idea, the work in [MP06] has made an extension by decoupling the metric definitions from the metamodel, which had been upgraded to the UML 2.0 metamodel, into a separate metrics package. Another approach [Chi11] presented the SQUAM framework as a tool-supported method for metrics definition. Within this framework, 26 metrics for UML 2.2 class diagrams have been developed as additional operations of the metaclass *Class*. Our approach for metrics definition with OCL introduced here has utilized these works with extensions and improvements indicated below.

- We have upgraded to the full UML 2.4 metamodel (with 69 classes and 99 associations).
- We offer an interactive process for developing new metrics thanks to the availability of the *metamodel*, the *metamodel instantiation*, and OCL expression evaluation on the *metamodel*. Moreover, designers can use the newly defined metrics immediately for measuring or assessing the model, for example, by applying our proposed assessment method.

- With our approach, one can achieve metrics similar to a property of the class itself. This is much more natural and straightforward for metrics treatment, since a metric can be considered as a property of a class.

Metric definition using other languages: Besides OCL, several different languages have been used as the formal languages for model metrics definition. XQuery [Elw+05] and SQL [HW02] are examples from the academic context. Several tools have been developed for specification and calculation model metrics. EMF Metrics [AS10] is a prototype Eclipse plug-in working with models-based ideas in the Eclipse Modeling Framework. Within MagicDraw[1], UML metrics are measured by a Java hard-coded component, and the SDMetrics[2] tool works with models stored in the XML Metadata Interchange (XMI) format, and metrics are defined in the form of XML-based specifications. In contrast, our approach supports OCL, and we offer an interactive process for metrics definition (for developers with OCL expertise) or a threshold, template-based process (for developers with minimal or no OCL expertise).

Model quality assessment and smell detection: The work in [AGO12] has presented a unified method for the definition of UML class diagram quality properties, such as syntactic issues, best practices and naming issues. This work was also realized as a plug in of the Eclipse framework. Another approach [LGL14] was using the *mmSpec* to specify and check quality properties on UML models. In this work, the authors introduced some metric-related properties. A tool, i.e., *metaBest*, was also introduced in this work to illustrate and report the problematic elements. In [Bas+16], Francesco *et al.* introduced an approach for quality assessment of modeling artifacts (metamodels, models, model transformations) by supporting hierarchical quality model definition using OCL and evaluate modeling artifacts based on metric measurements. [AT13] presents the EMF Refactor tool that supports modelers to perform quality analysis of models and to apply resolutions to detected design smells if needed. A collection of ten different bad smells within the context of Palladio Component Model [Str+16] is defined and presented in [Bec+16]. Moreover, the authors also discussed the negative effects of these smells to the model quality. Bettini *et al* [Bet+19] exploit a domain-specific language named Edelta [Bet+17] for smell definition and detection. In this work, the

[1]MagicDraw — http://www.magicdraw.com/
[2]SDMetrics — UML,http://www.sdmetrics.com/

authors deal with five different bad smells, and each smell is also linked to one or a few corresponding quality attributes.

Chapter 5

Evaluation of Metric Measurement and Smell Detection

In Chapter 4, we have presented how design metrics can be measured and design smells can be detected utilizing metamodeling concept. To show the feasibility and the usefulness of these mechanisms, we have conducted an evaluation on a large data set of UML models. This chapter presents the evaluation result. In order to perform the evaluation, firstly we extend the tool USE by adding the metric measurement and smell detection functionalities for a collection of models. The metrics of all models in the data set then are computed using the tool USE, and the collected metric data is analyzed for common characteristics of practical models. The metric measurement evaluation is represented in Section 5.3. A set of predefined design smells is also checked by the tool USE on all analyzed models. We then analyze the result to uncover the quality of models in practice. Section 5.4 illustrates the evaluation result of smell detection.

5.1 Tool Extension

In this section, we present an extension of the tool USE for metric measurement and model quality assurance for a collection of models utilizing the approaches presented in the previous chapters. More details about this tool extension can be found in [DG19] These new functionalities will be used for the evaluation presented in the next sections.

Figure 5.1 illustrates an overview of the extension that we have made for the tool USE (see Section 2.4) regarding performing the evaluation. USE originally was a two-level modeling tool supporting the specification, visualization, validation and verification of information systems based on UML and OCL. The input is a textual specification containing the description of model elements found in UML class diagrams. The model is enriched by additional integrity constraints written in the Object Constraint Language (OCL). Developers can create a system state of the specified model and validate the defined constraints on this system state. USE offers various kinds of UML diagrams for model exploration, e.g., object diagrams and sequence diagrams. It also provides other features for model validation, verification and exploration.

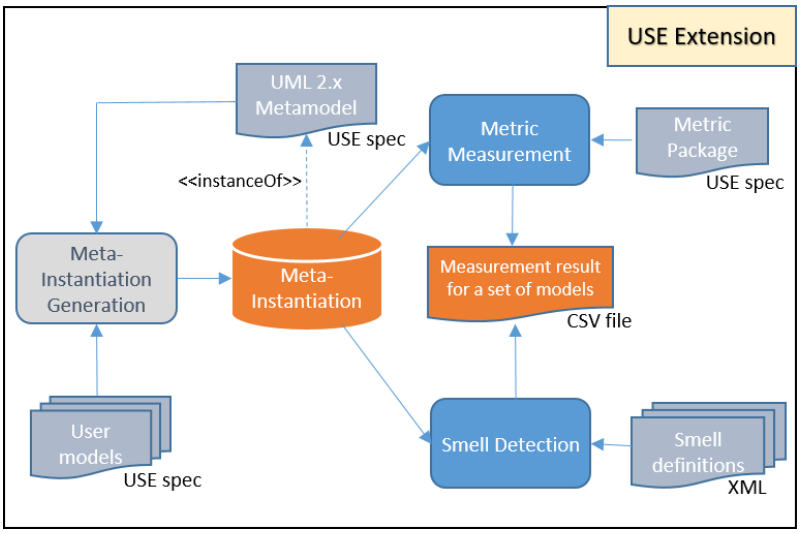


Figure 5.1: Extension of tool USE.

In the extended version o USE[1], new features are now available, in particular for model quality assessment, i.e., metric measurement, smell detection. These new features utilize the UML 2.4 metamodel (the OMG superstructure), which is internally added as a type model for all UML user models. The UML metamodel is applied for the definition of metrics and design smells. In principle, the metrics and smells are defined as OCL expressions in the context of the UML metamodel, however, the evaluation of these OCL expressions is done on the level of a metamodel instantiation, i.e., a UML metamodel instantiation that corresponds to the UML user model. Metamodel instantiation generation is a functionality that runs in the background and takes responsibility for

[1]Download the tool at: `https://github.com/doankh/my_use`

generating the metamodel instantiation corresponding to the input UML user model.

In order to conduct the evaluation, a set of predefined metrics and smells must be evaluated on a large number of class models. Therefore, we extend the tool USE to allow users working with a set of input models and collecting many measurement result from the input dataset. Moreover, several new commands have been added to the extended version of USE to compute and record the result.

- `metric m|c <list of metrics>`

  This command computes a list of metrics of the current class model. The first parameter, i.e., `m` or `c`, determines the scope of the metrics to be measured. The last parameter indicates a list of metrics that the user wants to measure. If this parameter is missing, it will compute all predefined model/class scope metrics. For example, command `metric m AIF,MIF` records the value of AIF and MIF model scope metrics of the current class model.

- `metricAll m|c <list of the metrics> <directory path>`

  This command measures a list of metrics on all class models in a directory. The meaning of the first and second parameters are similar to the previous command. The last parameter indicates the directory which contains the class models that need to be analyzed. For example, command `metricAll c F:\EvaluationModels` records the value of all predefined class scope metrics of all class models in the indicated directory.

- `smell <directory path>`

  This command checks smells of the current class model or a set of models present in a directory. If the parameter `<directory path>` is missing, it only detects issues of the current class model. For example, command `smell F:\EvaluationModels` detects the occurrence of all predefined design smells of all class models in the specified directory.

The result of executing each command is recorded in a CSV (Comma-separated values) file, and it can be utilized for further analysis.

5.2 Evaluation Dataset

Our aim is not only evaluating our approach on metric computation and smell recognition but also analyzing the common level of object-oriented mechanisms and design quality used in practice. Therefore, the dataset should contain a significantly high number of models. The more models, the more reliable the analysis result. Moreover, the size of the analyzed models should fluctuate from small ones to very large ones.

To achieve a dataset satisfying the above requirements, the models using in this experiment are collected from two sources. The first source is a series of models gathered from systematically selected GitHub repositories. How this dataset is collected and organized is presented in [NMS17]. In this evaluation, we only consider the structure of the models, therefore, only `.ecore` files are taken into account. From 16502 `.ecore` files in this data set, 1118 models are selected based on the following reasons.

- There are many duplicate files (files with the same names) in the original data set. To prevent bias when analyzing the data, the duplicate files are removed. We only include unique files.

- We conduct the evaluation on the tool USE, therefore, the model definition stored in the Ecore models must be transformed into USE specifications. Due to e.g., syntax errors or unsupported features of Ecore models in USE, some original models are unable to transform to USE specifications.

- Our aim is not just to evaluate our work on metric measurement, we also want to analyse the data for a common level of the object-oriented mechanism used in practice through metrics. Thus, the models which are too small, i.e, models which have fewer than 3 classes, are excluded. We do not include these models in the experiment because applying the object-oriented mechanisms onto a model which has only one or two class is meaningless.

The second source is the Atlan Ecore zoo [2] dataset of 305 Ecore models. This dataset has been chosen because it contains a set of modes that MDE practitioners built in practice. Some models in this dataset are also excluded due to the same criteria as in the first data source.

[2] `https://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Zoos`

As a result, we have accumulated a dataset of 1338 publicly available models with 28884 classes in total used for the evaluation [3]. Fig 5.2 illustrates the statistical information of the dataset used in the experiment. The size of models in this repository varies from small ones with four class to large ones with more than 100 classes. The largest model has 332 classes, and 21.6 is the average number of classes of models in the data set. We believe that by analysing metrics and the design issues on a larger number of models in this dataset, one can learn about the typical level of object-oriented mechanisms as well as the popularity of the issues occurring in the models used in practice.

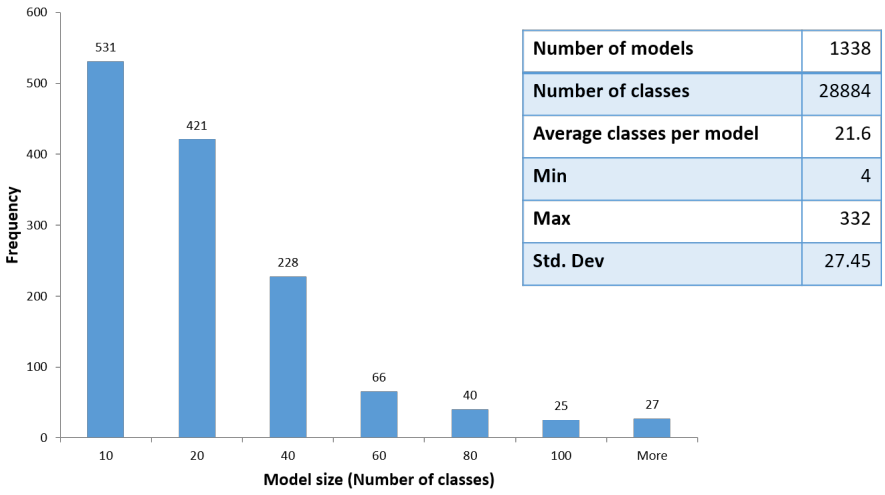


Figure 5.2: The size of models using in the evaluation.

As can be seen in Fig 5.2, more than 70% (952 of 1338) are small models with less than 20 classes. The number of models decreases when their size increases. However, the variability of the model size is acceptable with nearly 150 models which contain more than 60 classes and 52 models are considered to be big models (including more than 100 classes).

5.3 Metric Measurement Evaluation

To show the applicability of our approach for defining metrics presented in the previous section, we have selected 25 metrics from the literature and define them using the proposed mechanism. They have been integrated into the tool USE and are ready to be applied for model measurement, model querying and quality assessment. These metrics are selected based on the following criteria.

[3]The dataset is available at: `https://github.com/doankh/USE_Dataset`

- They are applicable at the model level. Therefore, a metric such as LCOM [CK94], which needs the implementation of methods to measure, is excluded.

- We select only one metric from several metrics which measure the same data. For example, NOH [BD02] and NGenH [Gen02] both measure the total number of generalization hierarchies within a class diagram; thus, we only define one of those (NOH).

Table 5.1 represents a list of metrics used in the evaluation. The reference next to the name indicates the reference source, in which the metric was first defined. We classify the metrics by (a) the quality aspects they measure, i.e., complexity, coupling, object-oriented design principles, size, and (b) the context where they are applicable, i.e., class scope or model scope. Detailed definitions of these metrics in the form of OCL expressions can be found in Appendix A

To investigate the common characteristics of models in practice, we have defined two more metrics (NOIC and NFLC), which relate to the inheritance aspect. In the following, these two metrics are briefly explained.

- **NOIC** (Number of Inherited Classes). This computes the total number of classes which inherit from at least one super class. The following OCL expression (at the metamodel level) defines of the NOIC metric.

```
ModelMetrics :: NOIC(): Integer =
  Class.allInstances()→select(c|
        not c.parents()→isEmpty())→size()
```

- **NFLC** (Number of immediately Featureless Classes). This metric calculates the total number of concrete classes which have no attributes and do not involve in any association. These class are identified as "immediately featureless" because they may inherit features from its superclasses. The definition of the NFLC metric can be formulated as the following OCL expression (at the metamodel level).

```
ModelMetrics :: NFLC(): Integer =
  Class.allInstances()→select(c|c.isAbstract = false and
    c.ownedAttribute→isEmpty() and
   c.association→isEmpty)→size()
```

Table 5.1: Selected UML class model metrics.

Metric [source]	**Type**
Model scope Metrics	
Number of inherited classes (NOIC)[*]	Size
Number of immediately featureless classes (NFLC)[*]	Size
Total number of classes in the design (DSC) [BD02]	Size
Number of total associations (NAssoc) [Gen02]	Size
Maximum depth of inheritance (MaxDIT) [CK94]	Complexity
Number of hierarchies (NOH) [BD02]	Complexity
Attribute inheritance factor (AIF) [AM96]	OO design
Method inheritance factor (MIF) [AM96]	OO design
Attribute Hiding Factor (AHF) [AM96]	OO design
Method Hiding Factor (MHF) [AM96]	OO design
Polymorphism factor (PF) [AM96]	OO design
Abstractness (A) [Mar03]	OO design
Class scope Metrics	
Number of attributes per class (NOA) [BHe96]	Size
Number of local methods (NOM) [LH93]	Size
SIZE2 = NOA + NOM [LH93]	Size
Average Parameters per Method (APPM) [LK94]	Size
Number of children (NOC) [CK94]	Complexity
Depth of inheritance (DIT) [CK94]	Complexity
Number of Ancestor Classes (NAC) [Li98]	Complexity
Number of Descendant Classes (NDC) [Li98]	Complexity
Specialization index (SIX) [LK94]	Complexity
Number of methods inherited (NMI) [LK94]	OO design
Number of methods overridden (NMO) [LK94]	OO design
Coupling between object (CBO) [CK94]	Coupling
Number of attributes that have another class as their type (DAC) [LH93]	Coupling
Number of different classes that are used as types of class attributes (DAC') [LH93]	Coupling
Number of associations (NAS) [HCN98]	Coupling
Direct class coupling (DCC) [BD02]	Coupling
Cohesion among Methods of a Class (CAMC) [Ban+99]	Cohesion
Normalized Hamming Distance (NHD) [CSC06]	Cohesion
Method-Method through Attributes Cohesion (MMAC) [AB10]	Cohesion

To see the detailed definition (in natural language) and the description of the metrics from the literature, interested readers can follow the reference source indicated in the first column of Table 5.1.

The tool USE supports batch metric measurement on a set of models and report the result in a CSV (Comma-separated values) file. Therefore, the first step of the evaluation is running the tool USE to compute the selected metrics on all models in the corpus. The result will show the feasibility and applicability of our approach. Next, we analyze the recorded data for the level of object-oriented mechanisms used in the practical models through metrics. Particularly, the descriptive statistics of the values of selected metrics are extracted from the result. From that, we calculate the common value of the selected metrics following the classification technique presented in [JTS14]. Particularly, for each metric, the common value is defined as an interval between a low value and high value. The low value and high value are estimated as follows.

- Lower bound value = IQM(Mean) - IQM(SD)
- Upper bound value = IQM(Mean) + IQM(SD)

IQM is the "Interquartile Mean" [Sch12]. IQM(Mean) is the mean value of the interquartile data and IQM(SD) is the standard derivation of the interquartile data.

5.3.1 Model Scope Metrics

Data Analysis

Firstly, the value of nine pre-defined model scope metrics of 1338 analyzed models were computed and recorded. Table 5.2 represents the statistical characteristics of the recorded data.

The interquartile range (IQR) column indicates the difference between the 1st quartile and 3rd quartile of the data. It gives an overview of the distribution of the metric. The column (N=0) represents the number of models which have the metric value 0. For example, among 1338 models, there are 208 models which have the MaxDIT metrics 0. That means more than 15.5% of the analyzed models are formulated without the inheritance mechanism. It is a significant figure since inheritance is one of the most essential OO mechanisms. The column (N/A) shows the number of models which fail to

Table 5.2: Descriptive statistics of the model scope metrics.

Metric	N	N=0	N/A	Min	Median	Max	Mean	Std Dev	IQR
DSC	1338	0	0	4	13	332	21.6	27.4	15
NOIC	1338	208	0	0	7	299	13.9	23.5	12
NFLC	1338	561	0	0	1	132	3.6	8.5	4
NAssoc	1338	30	0	0	13	778	23.5	38.2	17
MaxDIT	1338	208	0	0	2	10	1.86	1.55	2
AIF	1338	361	100	0	0.36	0.97	0.36	0.3	0.63
MIF	1338	77	1153	0	0.25	0.96	0.35	0.35	0.67
AHF	1338	1238	100	0	0	0	0	0	0
MHF	1338	185	1153	0	0	0	0	0	0
PF	1338	96	1230	0	0	0.7	0.02	0.11	0
A	1338	562	0	0	0.08	1	0.12	0.14	0.2

calculate the corresponding metric. The reason for the failure is a *divide by zero* error. For example, the MIF metric is calculated as a ratio between the sum of the total number of inherited methods of all classes and the sum of the total number of available methods of all classes. Therefore, this metric is unable to be measured on models without any method definition. As can be seen from Table 5.2, the MIF metric is unable to be computed on 1153 of total 1338 models (86%), that means only 14% models in the dataset contain method definition. This low number indicates that in practice, models mostly utilized to define the structural aspect of the system under consideration. The behaviour of the systems has not been significantly concerned. A large number of failures also can be seen in the MHF and PF metrics due to the same reason. The MHF metric measures the method hiding factor by dividing the sum of the invisibilities of all methods defined in all classes and the total number of methods in the model. The PF metric measures the level of polymorphism mechanism used in a model. The statistics of the AHF metric, which measures the attribute hiding factor of a model, shows that the calculation of this metric is either fail (100 models) or returning 0 (1238 of 1338 models). This statistic occurrence happens because of the lack of visibility declarations in the Ecore models.

Regarding the usage of abstract classes within the practical user models, the measurement of A (Abstractness) metric indicates the low popularity. Five hundred sixty-two user models having the Abstractness metric as 0 meaning 42% of the analyzed models do not contain any abstract class, even with huge models (having more than 100 classes). A similar result is also mentioned

in [Wil+13]. It is a bad practice since using abstract classes will increase the abstractness of the design. As a result, it raises the understandability and maintainability.

In the following, we want to investigate the correlations of different metrics to have a better understanding of the characteristics of user models in practice. Particularly, we compute the correlation index between seven metrics, i.e., DSC, Assocs, NOIC, NFLC, MaxDIT, AIF, A, using the Pearson's correlation coefficient [RN88]. The remaining metrics are excluded due to the lack of proper data. Pearson's correlation coefficient is widely used in science to measures the degree of linear relationship between two variables. Its values vary from -1 to 1. Values close to 1 imply a strong positive linear correlation, while values close to -1 indicate a strong negative linear correlation. The value of 0 means that there is absolutely no correlation between the two analyzed variables. Figure 5.3 presents the Pearson correlation matrix between the analyzed metrics. We highlight the correlation values grater than 0.7 or less than -0.7 to emphasize pairs of metrics which are strongly related.

	DSC	*Assoc*	*NOIC*	*NFLC*	*MaxDIT*	*AIF*	*A*
DSC	1.00						
Assoc	**0.83**	1.00					
NOIC	**0.94**	**0.72**	1.00				
NFLC	0.67	0.35	**0.71**	1.00			
MaxDIT	0.60	0.42	**0.70**	0.47	1.00		
AIF	0.25	0.17	0.33	0.17	0.44	1.00	
A	0.08	-0.02	0.15	0.00	0.36	0.27	1.00

Figure 5.3: Correlation indexes between selected model scope metrics.

As can be seen, the total number of classes (DSC) strongly correlates to the number of total associations (Assoc) and the number of inherited classes (NOIC) (with the correlation indexes are 0.83 and 0.94, respectively). The correlation value of 0.71 shows a significant positive correlation of the relationship between the number of inherited classes (NOIC) and number of immediately featureless classes (NFLC). More specifically, the growth of the size of models leads to the increase of the adoption of inheritance and structural feature reusability. On the other side, the value of the attribute inheritance factor (AIF) and Abstractness(A) metrics do not show any significant correlation with the value of other metrics since those correlation indexes are quite close to 0. In other words, the level abstractness and attribute inheritance are not

proportional to the size of the model measured by the number of classes. Only the value of the maximum depth of inheritance (MaxDIT) shows a slightly positive correlation to attribute inheritance factor and the abstractness level of the user models (with the correlation indexes are 0.44 and 0.36, respectively).

Metric Thresholds Extraction

From the above observation, it is clear that utilizing the recorded data for calculating the common value for MIF, AHF, MHF, PF metrics is not practical. As a result, we only select three metrics, i.e., **MaxDIT** (Maximum Depth of Inheritance), **AIF**(Attribute Inheritance Factor), and **A** (Abstractness), for the further analysis of the common value (threshold). Fig 5.4 depicts the distribution of these three metrics over the analyzed models.

The common value of MaxDIT, AIF and A(Abstractness) metrics are extracted based on the classification technique presented at the beginning of this section. The result is presented in Table 5.3. Note that the value of the MaxDIT metric is an integer, thus the common value of it has been rounded during the calculation.

Table 5.3: The common value of the selected model scope metrics

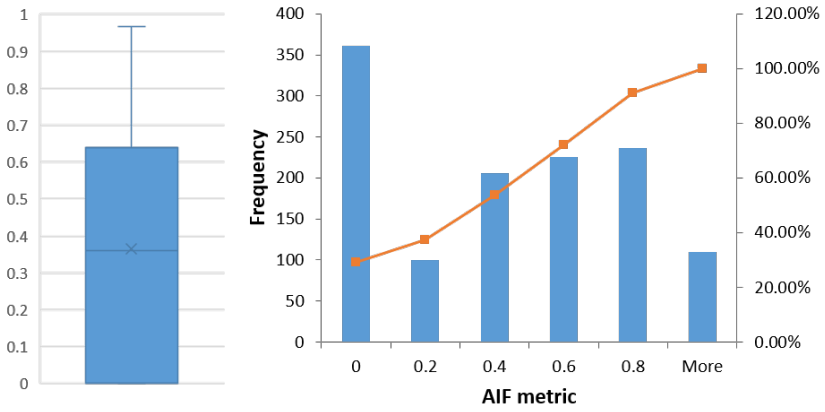
	MaxDIT	**AIF**	**A**
Low value	1	0.2	0
High value	2	0.6	0.16

5.3.2 Class Scope Metrics

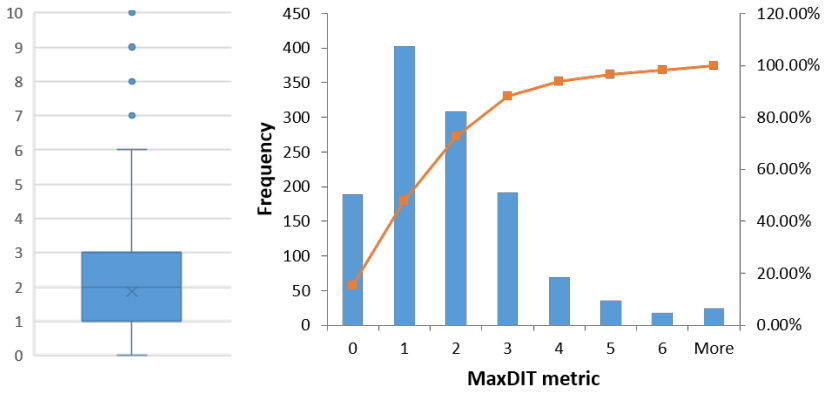
Data Analysis

Firstly, the value of 19 predefined class scope metrics of 28884 classes in 1338 models are computed by executing the batch metric measurement on the tool USE. The data is stored in a CSV file. The statistical characteristics of the recorded data is represented in Table 5.4

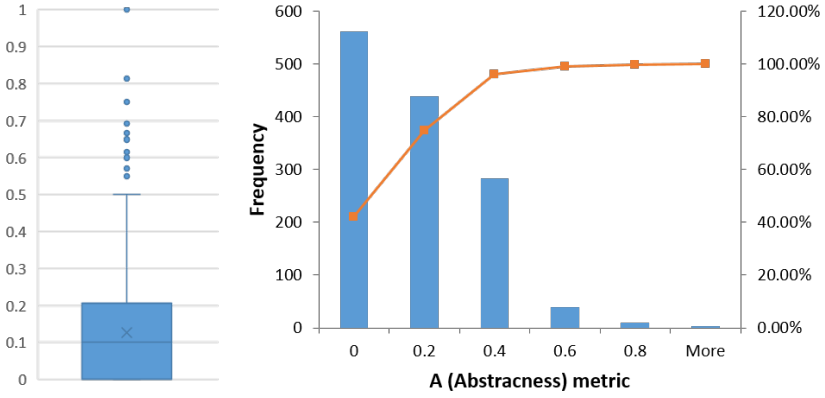
The descriptive statistics shown in Table 5.4 once again indicates the lack of method definition of the models in practice. As can be seen, 28238 of 28884 classes (nearly 98%) have the NOM (Number of Methods) metric as 0. Although only 2% of classes contain method definitions, interestingly, the max value of NOM metric is 58. This is a bad practice since the class which has 58 methods is too complicated and should be separated into smaller classes. This



(a) Distribution of the AIF metric.



(b) Distribution of the MaxDIT metric.



(c) Distribution of the A(Abstracness) metric.

Figure 5.4: The distribution of the selected model scope metrics.

Table 5.4: Descriptive statistics of the class scope metrics

Metric	N	N=0	N/A	Min	Median	Max	Mean	Std Dev	IQR
NOA	28884	17013	0	0	0	145	1	2.56	1
NOM	28884	28238	0	0	0	58	0.05	0.67	0
SIZE2	28884	16696	0	0	0	145	1	2.66	1
APPM	28884	327	28238	0	0	5	0.5	0.77	1
NOC	28884	22538	0	0	0	86	0.7	2.1	0
DIT	28884	10280	0	0	1	10	1.34	1.5	2
NAC	28884	10280	0	0	1	18	1.44	1.72	3
NDC	28884	22538	0	0	0	299	1.44	6.71	0
SIX	28884	1772	27052	0	0	4	0.06	0.26	0
NMI	28884	27561	0	0	0	57	0.19	1.35	0
NMO	28884	28528	0	0	0	15	0.02	0.28	0
CBO	28884	8704	0	0	1	103	1.56	2.2	2
DAC	28884	28884	0	0	0	0	0	0	0
DAC1	28884	28884	0	0	0	0	0	0	0
DCC	28884	28696	0	0	0	5	0.01	0.13	0
NAS	28884	8738	0	0	1	103	2.14	3.22	3
CACM	28884	0	28732	0.06	0.5	1	0.61	0.31	0.67
NHD	28884	50	28706	0	0.4	1	0.42	0.33	0.62
MMAC	28884	28655	0	0	0	1	0.0	0.07	0

fact also leads to a large number of classes which have the NMO (Number of Methods Overridden) and the NMI (Number of Methods Inherited) metrics value as 0 (99% and 94%, respectively). A high percentage of classes failed when calculating the APPM (Average Parameters per Method) and the SIX (Specialization Index) metrics is also the consequence of the lack of method definition of the models in the corpus. The calculation of these metrics will return the *divide by zero* error on classes without method definition. A similar result also can be seen from the calculation of the CACM (Cohesion among Methods of a Class), the NHD (Normalized Hamming Distance) metrics, since the method definition is needed to calculate these metrics.

On the other hand, the measurement of the SIZE2 metric, which counts the total number of attributes and operations of a class, returns 0 on 16696 classes. That means, more than 55% of classes do not have any attribute declaration or method definition. This is a bad practice since the definition of these classes do not contain any details.

The variability of the NOC (Number of Children) and the NDC (Number of Descendant Classes) metrics are very low. The measurement of the NOC and the NDC metrics return 0 on more than 78% of classes (mean is 0, and

IQR is 0). The low variability also can be seen on the statistics of the NOA and MMAC metrics.

A remarkable statistic can be seen from Table 5.4, that the measurement of the two data abstraction coupling metrics, i.e., DAC and DAC', on all classes result in 0. This statistic indicates that MDE practitioners avoid declaring attributes that have another class as their type. According to EMF design guidelines, this kind of coupling should be presented as an association. This may lead to the high value of the NAS metric, which measures the total number of associations a class involves. Table 5.4 illustrates that, on average, a class participates in 2.14 associations. And specifically, there is a class which couples to 103 classes through associations.

Metric Thresholds Extraction

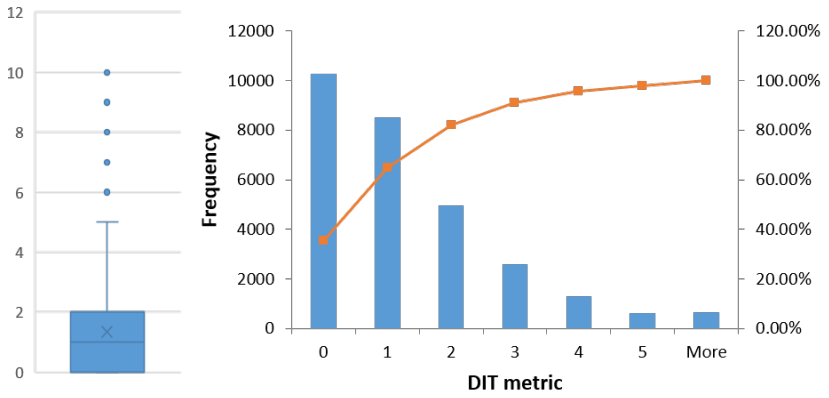
For the further analysis of the common value (threshold), three class scope metrics, i.e., **DIT** (Depth of Inheritance), **NAC**(Number of Ancestor Classes), and **NAS** (Number of Associations), are selected. The other metrics are excluded due to the lack of data, or the low variability. Table 5.5 illustrates the thresholds of these metrics, which are estimated using the classification technique presented at the beginning of this section.

Table 5.5: The common value of some model scope metrics

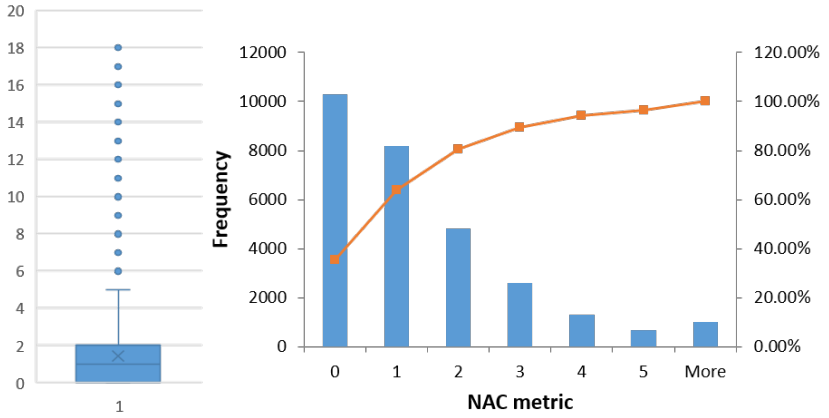
	DIT	**NAC**	**NAS**
Low value	0	0	0
High value	2	2	2

5.4 Smell Detection Evaluation

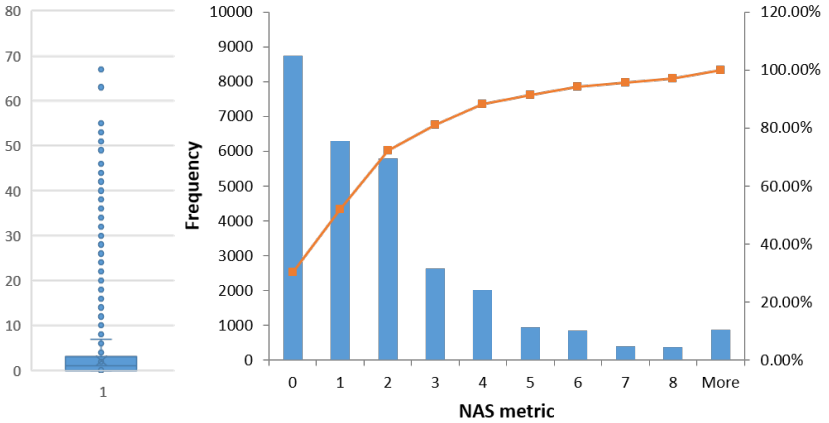
In order to evaluate the quality of the class models in practice regarding design smells, we have defined a library of design smells using the approach presented in Section 4.3. This library covers typical flaws or issues that modelers tend to commit. In this evaluation we only assess smells of four categories, i.e., essential property, best practice, metrics, naming convention. Some of them are collected from [AGO12; Wüs; LGL14]. We exclude the trivial smells and syntactic smells because they can mostly be checked by modeling tools (tool USE as well) while compiling the user models. The list of design smells used



(a) Distribution of the DIT metric.



(b) Distribution of the NAC metric.



(c) Distribution of the NAS metric.

Figure 5.5: The distribution of the selected model scope metrics.

in the evaluation is presented in Table 5.6. Detailed definitions of these smells in the form of OCL expressions can be found in Appendix B

Firstly, we run the tool USE to perform a batch evaluation of all predefined smells on every model in the dataset. The result is recorded in a CSV file. By doing that, we can check the feasibility and applicability of our smell detection approach on a large number of user models. The evaluation result can also be utilized to measure the quality of models used in practice. Table 5.7 represents a summary of the result.

As can be seen, 3069 issues are found in 1338 analyzed models. The minimum number of issues detected in a single model is 0, while the maximum is 12. Issues are discovered in 1092 models (79%) with an average number of detected issues per user model as 2.29. Within the smell library used in the evaluation, the best practice smells are most likely to occur, while naming convention smells have the lowest probability. Fig. 5.6 illustrates the number of flawed models regarding the number of detected smells. Models with less than three issues are most popular. The most common number of the detected smells is 2 with 348 models. From this peak, the number of faulty models significantly decreases corresponding to the increase in the number of the found issues. Only one model has 12 issues, the maximum number of issues founded in a model in the dataset.

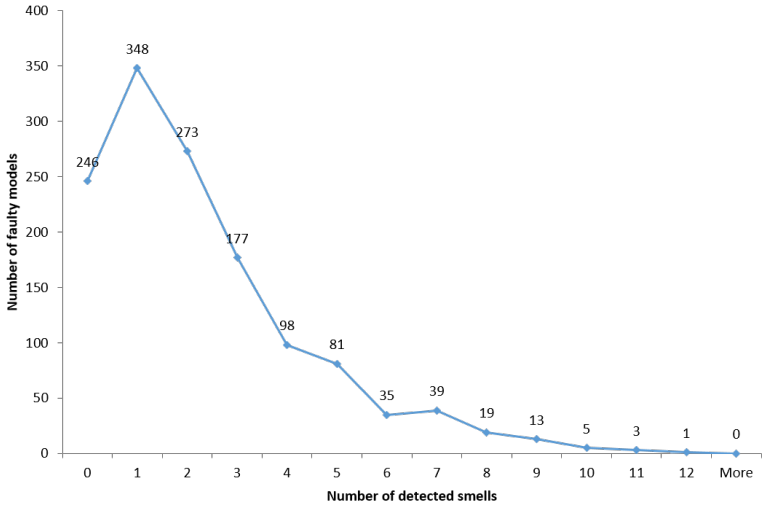


Figure 5.6: Flawed models regarding number of detected smells.

With regard to the number of faulty models corresponding to each smell, Fig. 5.7 presents a bar chart which depicts how many analyzed models violate each design smell in Table 5.6.

Table 5.6: Predefined design smells using in the evaluation.

ID	Description	Severity
Best Practice		
BP01	A class is contained in two classes.	Low
BP02	A root class is not an abstract class. A root class has subclasses and does not participate in any association.	Low
BP03	An attribute is repeated (with the same definition) among all specific classes of a hierarchy.	Medium
BP04	There exists an abstract class which is a general class of only one specified class. It negates the usefulness of the abstract classes.	Medium
BP05	There exist redundant generalization. A generalization is redundant if it could be implied from other generalizations in the model.	Medium
BP06	A class that associates to a descendant class.	Low
Essential Property		
EP01	An abstract class is a subclass of a concrete class.	Critical
EP02	There exist composition cycles in the design.	Critical
EP03	A class contains one of its superclasses and the multiplicity value in the composition end is 1 (this is finitely unsatisfiable).	Critical
EP04	A reflexive association that has two member ends x1..x2 and y1..y2, where y1¿x2.	Critical
EP05	There exist isolated classes in the design. An isolated class is a class which is not involved in any association or inheritance hierarchy).	Medium
EP06	A class is contained in two classes, and the cardinality in the composition end of one of them is 1.	Medium
Metrics		
ME01	A class is overloaded with attributes (10-max by default). This smell regards to the NOA (Number of Attributes) metric.	Low
ME02	A class participates in more than ten association. This smell regards to the NAS (Number of Associations) metric.	Medium
ME03	There exists a hierarchy which is too deep (5-level max by default) – This smell regards to the DIT (Depth of Inheritance) metric.	Medium
ME04	A class has too many direct children (10-max by default) - This smell regards to the NOC (Number of Children) metric.	Medium
ME05	A class has more than 60 attributes and operations (God class).	Medium
Naming Convention		
NA01	The name of a class does not start with a capital letter (PascalCase convention)	Medium
NA02	Attributes are named after their feature class (e.g., an attribute personID in class Person).	Low
NA03	A name of a class is a Java keyword.	Low
NA04	A name of a class is a C++ keyword.	Low
NA05	A name of an attribute or operation does not start with a lowercase letter (camelCase convention).	Medium

Table 5.7: Summary of the evaluation result.

	BP	ES	ME	NA	Total
Detected smells	1577	513	572	407	3069
Faulty models	910	378	390	340	1092
Min	0	0	0	0	0
Max	6	4	4	4	12
Average	1.18	0.38	0.43	0.3	2.29

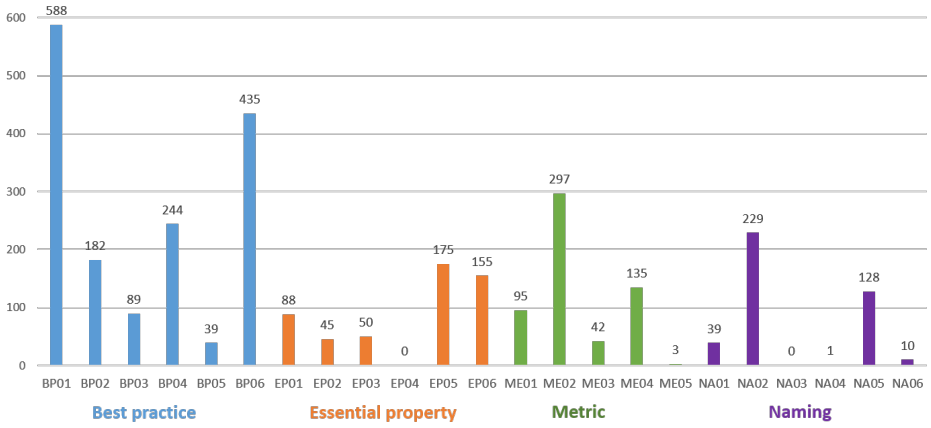
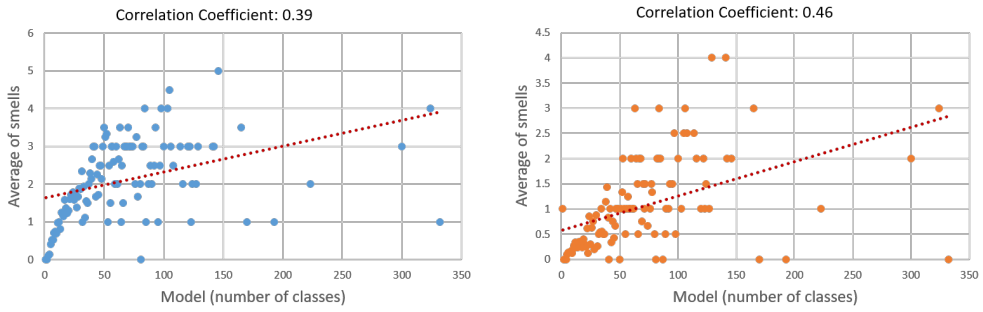


Figure 5.7: Number of faulty models corresponds to each smell.

The chart shows that the smell BP1 is discovered in the largest number of models with 558 models (40% of the analyzed models). This smell checks the existence of a class A, which is contained in two other classes, e.g., B and C. The stronger version of the BP1 smell is the EP6 smell, which also fails in 155 models. This smell specifies a class contained in not only two other classes but also the cardinality in the composition end of one of them being greater than 1. It is an error since an instance of A cannot be contained in an instance of one class, and it is mandatorily to be contained in an instance of another class. The high percentage of the faulty models also can be seen in the evaluation of the BP06 and ME02 smells, with 435 (31%) and 297 (21%) non-fulfilled models, respectively. The BP06 smell checks the classes that associate to their descendant class and the ME2 smell inspects the existence of classes which participate in too many associations, i.e., ten associations. Interestingly, there are two smells, i.e., EP04 and NA03, which are not violated in any analyzed model. EP04 checks the presence of a reflective association which has a lower bound at one end being greater than the upper bound of the other end. It is impossible to generate a valid system state of a model containing

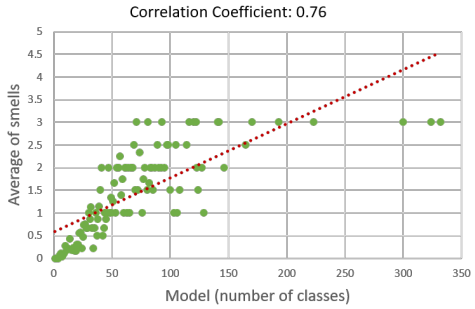
this kind of reflective association. Zero faulty models regarding the NA03 smell indicates that there is no class of any analyzed model named as a Java keyword. This is understandable since the analyzed models are collected from the EMF community. A very few numbers of flawed models are also found in the evaluation of the ME5 and NA04 smells, with only 3 and 1 models, respectively.

Concerning the correlation between the number of non-fulfilled models and the model size (with respect to the number of classes), we have conducted some analysis, and the result is presented in Fig 5.8. In four charts in Fig 5.8, the vertical axis is the average number of faulty models, and the horizontal axis is the model size measured by the number of classes.

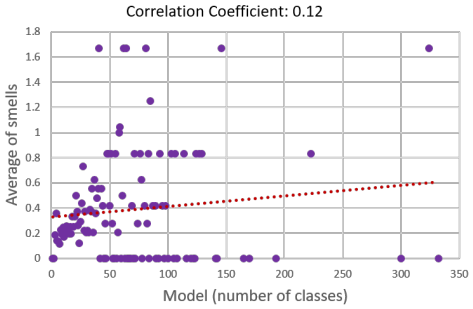


(a) Best practice

(b) Essential Property



(c) Metric



(d) Naming convention

Figure 5.8: Correlation between number of faulty models and model size.

The chart 5.8c indicates a quite strong relationship between model size and the number of faulty models regarding metric smells, with a correlation coefficient of 0.76. This means the number of non-fulfilled models regarding the metric issues seems to increase when the size of the model grows. This trend is understandable because the growth of the number of classes usually increases the complexity of the model. The correlation coefficient of 0.39 and 0.46 corresponding to the best practice and essential property categories show

that there is a positive correlation between the number of faulty models and the model size, but it is not strong and likely insignificant. The increase of the model size seems to be irrelevant to the number of flawed models regarding the naming convention smells since the correlation coefficient value is 0.12.

Chapter 6

Summary of Additional Contributions

In previous chapters, the main contributions of the author are represented. As our working group, i.e., the database systems group, is doing the researches in several directions in the context of model-driven development, the author of this thesis have been contributed in some other published works during the years of PhD study. This section briefly presents the additional contribution of the author to these publications.

6.1 Model Validation and Verification

Model-driven engineering is a software development paradigm that focuses on the use of models through the software development process. Models are the central artifacts in MDE. They specify the abstract representation of the structural and the behavioural aspects of the system under consideration. Therefore, model validation ("Are we building the right product?") and model verification ("Are we building the product right?") are important quality improvement techniques within MDE. The author of this thesis have contributed to two published works regarding model validation and verification. The first work is presented in [GHD18]. In this work we introduce eight use cases for model validation and verification in the context of a UML and OCL model. Brief descriptions of these use cases are represented as follows.

1. **Model consistency**. This use case tests whether a valid system state of the model can be generated under the stated finite search space from a configuration.

2. **Property satisfiability**. This use case verifies whether an additional OCL invariant can be satisfied with a system state as well.

3. **Constraint implication**. Whether an additional OCL invariant can be deducted from the original invariants of the model is determined with this use case.

4. **Constraint independence**. This use case checks the independence between OCL invariants in a model.

5. **Solution interval exploration**. This use case allows modelers to explore all satisfied system states in the form of object diagrams can be found.

6. **Partial solution completion**. From an incomplete object diagram, this task is then finding a completion in terms of objects, links and attribute values such that satisfying all constraints is presented.

7. **Equivalence implication**. The equivalence of two OCL formulas A and B can be checked using this use case.

8. **Partitioning with classifying terms**. This task allows constructing test cases that are partitioned by so-called classifying terms.

The main contribution of the author to this work is an evaluation of these proposed validation and verification mechanisms within a larger and classical example from the literature. It demonstrate the usefulness and applicability of the use cases.

The second work is presented in [DGH16]. In this paper, we introduce an approach for automatic behavioral property verification. An initial UML and OCL model will be enriched by frame conditions and then transformed into a (so-called) filmstrip model in which behavioral characteristics can be checked. The final step is to verify a property, which can be added to the filmstrip model in form of an OCL invariant. In order to make the process developer-friendly, UML diagrams can be employed for various purposes, in particular for formulating the verification task and the verification result. Fig 6.1 illustrates the overview of the proposed verification process.

The author also plays a part in [Bal+16], which compares three textual modeling languages, i.e., OCL, Alloy, FOML. The comparison of modeling languages regards to (1) mode of usage and problems being solved and (2) the

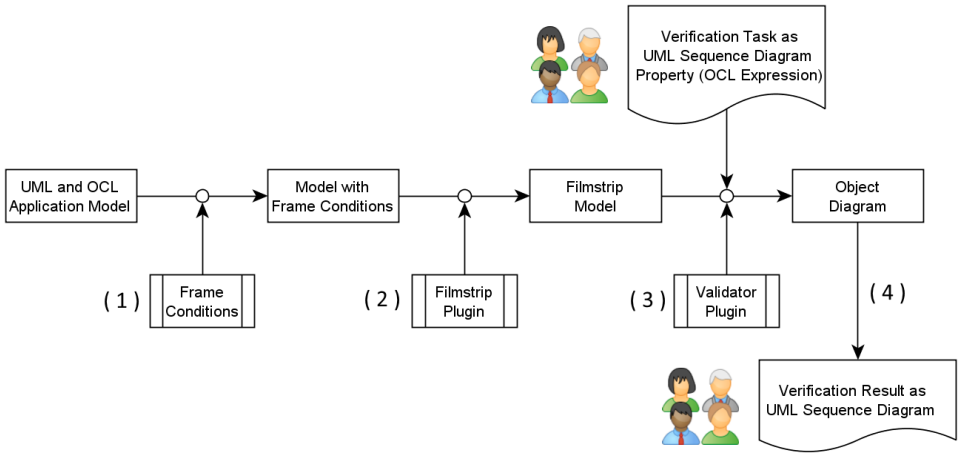


Figure 6.1: Overview of the proposed verification process

representation aspects. Specifically, the author is responsible for comparing the similarities and differences, as well as the advantages and drawbacks between OCL and the two other languages.

6.2 Logical Reasoning with Object Diagrams

UML diagrams, such as class and object diagrams, are utilized to diagrammatically represent real-world system at an abstract level with some constraints formulated in OCL. Taking this as a basis, UML and OCL can be a promising solution for representing and visualizing logical reasoning problem.

The author of this thesis have contributed to a work in [DG18c] by introducing an approach which utilized UML class diagram and object diagram to represent and solve the logical problems. Particularly, a logical reasoning problem is represented as a UML class model enhanced by OCL restrictions. Rules and questions are formulated as either partial object diagrams or OCL formulas within the model. The solutions can be found using a deduction system integrated in a tool and represented as object diagrams. This contribution focuses on representation and analysis of logical reasoning problems, in the context of the tool USE.

Chapter 7

Conclusion and Future Work

Assuring the quality of software artifacts in the early phases of the development process has been widely accepted in software engineering as a good practice. Detecting and fixing issues occurring in the design phase, for example, can significantly prevent faults arising in later stages, (e.g., coding phase). Thus, within Model-Driven Engineering (MDE), fixing issues of models reduces the cost and effort of the development process. In this thesis, we summarize the results of a research on achieving UML and OCL models quality by utilizing metamodeling.

7.1 Conclusion

This thesis firstly presents an approach for extending a traditional two-level modeling to a three-level modeling. The accessibility of the metamodel is fully supported within the proposed metamodeling architecture. Therefore, formulating reflective queries and constraints is now possible. We also provide a simultaneous representation of three levels of modeling thanks to the auto-generated meta instantiation at the M1 level. It helps to explore and understand the metamodel and the connection between modeling levels. We additionally offer a possibility of formulating level-crossing OCL expressions. Modelers can formulate OCL queries or constraints at the M2 level, and their semantics are applied at the M0 level.

We subsequently exploit the proposed three-level modeling architecture for calculating metrics and detecting design issues. With regard to metric measurement in particular, a library of desired metrics is defined as the operations of a newly added class at the M2 metalevel. These operations access the auto-

generated meta instantiation at the M1 level to compute the value of metrics on the model under consideration. Hence, one can both collect the metrics of models and utilize them for model querying and further model quality assurance. To detect a specific smell on models, the element that causes the issue is initially specified as a meta-level OCL expression. The existence of this smell is then inspected by evaluating a meta-invariant on the auto-generated meta instantiation. The violating elements are uncovered through a meta-query. These meta-invariants and meta-queries are generated from the smell definition. A further contribution of our work is the proposal of a complete process for model quality assessment with pre-defined metrics. In an assessment process, developers can achieve information about the quality of their design based on a metrics evaluation configuration, and they can detect problems in their model. Detailed OCL expertise is not required when a model is checked on the basis of such a configuration. Moreover, these works of model quality assurance are successfully integrated into the tool USE. They are closely related and can be applied interactively to help designers in achieving better model quality.

A part of this thesis represents an evaluation of our works on a large data set of UML models collected from practice. This evaluation illustrates the feasibility and usefulness of our mechanisms for model quality assurance. A further contribution of the evaluation is to answer the following research questions: (1) *How do the models in practice look?* (2) *What is the quality of the models in practice?*. To answer these research questions, we first run the tool USE to record the data regarding the metrics and smells of all models in the dataset. The data are subsequently analyzed, and the results are visually illustrated as tables and charts.

Finally, additional publications of the author are summarized. The contribution of the author in these published works mostly pertains to model validation and verification and logic reasoning with UML diagrams.

7.2 Future Work

Future work can be conducted in various directions. One promising direction would be the automation of a prediction system for external software properties starting from internal quality indicators (i.e., metric measurement). Software quality assessment models (QAM) are popular approaches for bridging the gap between software metrics and software quality attributes. Generally, a QAM

is a hierarchical model with the software metrics at the lowest level and the general quality attributes and sub-attributes at higher levels. An aggregation method is usually applied to aggregate software metrics to lowest-level sub-attributes and the sub-attributes to upper-level attributes. A decision- making technique, for example the fuzzy analytic hierarchy process [Saa08; Cha96], can be a promising aggregation method based on the measured metrics values and expert experience. Moreover, within the MDE context, select suitable metrics for filling in the lowest level of the hierarchical model could be a challenge for future work.

A further point for improving the applicability of our approach would be to define smells and metrics in terms of graphical patterns over the UML metamodel (UML metamodel class diagram fractions with particular requirements indicated) and to internally translate these patterns into OCL expressions that our approach currently supports. For example, a particular bad smell could be represented with a generalization arrow between two class templates SUB and SUPER and two attributes SUB::'name'+SUB and SUPER::'name'+SUPER indicating two similar attribute names.

Developing a means of automatically refactoring would be another direction for future research. Our current work allows modelers to detect the smells and the corresponding violating elements on models. Some changes might need to be applied to the faulty models to remove a detected smell, for example by adding new elements, removing certain elements, or updating existing elements. A set of these types of operations should be formally specified together with the definition of smells. Moreover, relations might have ensued between different bad smells, and the resolution of one bad smell might influence the occurrence of the others. Therefore, these relations must be considered when defining smells refactoring.

Bibliography

[AB10] Jehad Al-Dallal and Lionel C. Briand. "An object-oriented high-level design-based class cohesion metric". In: *Information & Software Technology* 52.12 (2010), pp. 1346–1361. DOI: 10.1016/j.infsof.2010.08.006. URL: https://doi.org/10.1016/j.infsof.2010.08.006.

[AG12] Colin Atkinson and Ralph Gerbig. "Melanie: Multi-level Modeling and Ontology Engineering Environment". In: *Proc. 2nd Int. Master Class MDE: Modeling Wizards, co-located with MODELS 2012*. MW'12. 2012, 7:1–7:2.

[AG16] Colin Atkinson and Ralph Gerbig. "Flexible Deep Modeling with Melanee". In: *Modellierung 2016, 2.-4. März 2016, Karlsruhe - Workshopband*. 2016, pp. 117–122. URL: https://dl.gi.de/20.500.12116/843.

[AGO12] David Aguilera, Cristina Gómez, and Antoni Olivé. "A Method for the Definition and Treatment of Conceptual Schema Quality Issues". In: *Proc. 31st Int. Conf. ER 2012*. 2012, pp. 501–514.

[AK02] Colin Atkinson and Thomas Kühne. "Profiles in a strict metamodeling framework". In: *Sci. Comput. Program.* 44.1 (2002), pp. 5–22. DOI: 10.1016/S0167-6423(02)00029-1. URL: https://doi.org/10.1016/S0167-6423(02)00029-1.

[AK03] C. Atkinson and T. Kuhne. "Model-Driven Development: A Metamodeling Foundation". In: *IEEE Software* 20.5 (2003), pp. 36–41. ISSN: 0740-7459.

[AK08] Colin Atkinson and Thomas Kühne. "Reducing accidental complexity in domain models". In: *Software and Systems Modeling* 7.3 (2008), pp. 345–359. DOI: 10.1007/s10270-007-0061-0. URL: https://doi.org/10.1007/s10270-007-0061-0.

[AKG10] Colin Atkinson, Bastian Kennel, and Björn Goß. "The Level-Agnostic Modeling Language". In: *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*. 2010, pp. 266–275. DOI: 10.1007/978-3-642-19440-5_16. URL: https://doi.org/10.1007/978-3-642-19440-5%5C_16.

[AM96] Fernando Brito e. Abreu and Walcelio Melo. "Evaluating the Impact of Object-Oriented Design on Software Quality". In: *Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results*. METRICS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 90–. ISBN: 0-8186-7364-8.

[Amb05] Scott W. Ambler. *The Elements of UML(TM) 2.0 Style*. USA: Cambridge University Press, 2005. ISBN: 0521616786.

[AS10] Thorsten Arendt, Pawel Stepien, and. "EMF Metrics: Specification and Calculation of Model Metrics within the Eclipse Modeling Framework". In: *BENEVOL*. 2010.

[AT13] Thorsten Arendt and Gabriele Taentzer. "A tool environment for quality assurance based on the Eclipse Modeling Framework". In: *Autom. Softw. Eng.* 20.2 (2013), pp. 141–184. DOI: 10.1007/s10515-012-0114-7. URL: https://doi.org/10.1007/s10515-012-0114-7.

[Atk97] C. Atkinson. "Meta-modelling for distributed object environments". In: *Proceedings First International Enterprise Distributed Object Computing Workshop*. 1997, pp. 90–101. DOI: 10.1109/EDOC.1997.628350.

[Bal+16] Mira Balaban et al. "A Comparison of Textual Modeling Languages: OCL, Alloy, FOML". In: *Proceedings of the 16th International Workshop on OCL and Textual Modelling co-located with 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 2, 2016*. 2016, pp. 57–72. URL: http://ceur-ws.org/Vol-1756/paper05.pdf.

[Ban+99] Jagdish Bansiya et al. "A Class Cohesion Metric For Object-Oriented Designs". In: *JOOP* 11.8 (1999), pp. 47–52.

[Bar+02] Aline Lúcia Baroni et al. "Using OCL to Formalize Object-Oriented Design Metrics Definitions". In: *In Proc. of QAOOSE'2002, Malaga*. Springer-Verlag, 2002.

[Bas+16] Francesco Basciani et al. "A Customizable Approach for the Automated Quality Assessment of Modelling Artifacts". In: *10th International Conference on the Quality of Information and Communications Technology, QUATIC 2016, Lisbon, Portugal, September 6-9, 2016*. 2016, pp. 88–93. DOI: 10.1109/QUATIC.2016.025. URL: http://doi.ieeecomputersociety.org/10.1109/QUATIC.2016.025.

[BD02] Jagdish Bansiya and Carl G. Davis. "A Hierarchical Model for Object-Oriented Design Quality Assessment". In: *IEEE Trans. Softw. Eng.* 28.1 (Jan. 2002), pp. 4–17. ISSN: 0098-5589.

[BDM97] Lionel Briand, Prem Devanbu, and Walcelio Melo. "An Investigation into Coupling Measures for C++". In: *Proceedings of the 19th International Conference on Software Engineering*. ICSE '97. Boston, Massachusetts, USA: ACM, 1997, pp. 412–421. ISBN: 0-89791-914-9.

[Bec+16] Steffen Becker et al. *Modeling and Simulating Software Architectures - The Palladio Approach*. Oct. 2016. ISBN: 9780262034760.

[Bet+17] Lorenzo Bettini et al. "Edelta: An Approach for Defining and Applying Reusable Metamodel Refactorings". In: *Proceedings of MODELS 2017 Satellite Event*. 2017, pp. 71–80. URL: http://ceur-ws.org/Vol-2019/me%5C_4.pdf.

[Bet+19] Lorenzo Bettini et al. "Quality-Driven Detection and Resolution of Metamodel Smells". In: *IEEE Access* 7 (2019), pp. 16364–16376. DOI: 10.1109/ACCESS.2019.2891357. URL: https://doi.org/10.1109/ACCESS.2019.2891357.

[Béz05] Jean Bézivin. "On the unification power of models". In: *Software and System Modeling* 4.2 (2005), pp. 171–188. DOI: 10.1007/s10270-005-0079-0. URL: https://doi.org/10.1007/s10270-005-0079-0.

[BG11] Fabian Büttner and Martin Gogolla. "Modular Embedding of the Object Constraint Language into a Programming Language". In: *Formal Methods, Foundations and Applications: 14th Brazilian Symposium*. Springer Berlin Heidelberg, 2011, pp. 124–139. ISBN: 978-3-642-25032-3. DOI: 10.1007/978-3-642-25032-3_9. URL: https://doi.org/10.1007/978-3-642-25032-3_9.

[BHe96] B.Henderson-sellers. *Object-Oriented Metrics, Measures of Complexity*. Prentice Hall, 1996.

[BK95] James M. Bieman and Byung-Kyoo Kang. "Cohesion and Reuse in an Object-Oriented System". In: *Proceedings of the ACM SIGSOFT Symposium on Software Reusability, SSR@ICSE 1995, April 23-30, 1995, Seattle, WA, USA*. 1995, pp. 259–262. DOI: 10.1145/211782.211856. URL: https://doi.org/10.1145/211782.211856.

[BKK16] Mira Balaban, Igal Khitron, and Michael Kifer. "Multilevel Modeling and Reasoning with FOML". In: *IEEE Int. Conf. SWSTE*. 2016, pp. 61–70.

[BMB96] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. "Property-Based Software Engineering Measurement". In: *IEEE Trans. Software Eng.* 22.1 (1996), pp. 68–86. DOI: 10.1109/32.481535. URL: https://doi.org/10.1109/32.481535.

[Boo95] Grady Booch. *Object-oriented analysis and design with applications (2. ed.)* Benjamin/Cummings series in object-oriented software engineering. Addison-Wesley, 1995. ISBN: 978-0-8053-5340-2.

[BV10] Manuel F. Bertoa and Antonio Vallecillo. *Quality Attributes for Software Metamodels*. Malaga, Spain, 2010.

[CG12] Jordi Cabot and Martin Gogolla. "Object Constraint Language (OCL): A Definitive Guide". English. In: *Formal Methods for Model-Driven Engineering*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. LNCS 7320. Springer, 2012, pp. 58–90. ISBN: 978-3-642-30981-6.

[Cha96] Da-Yong Chang. "Applications of the extent analysis method on fuzzy AHP". In: *European Journal of Operational Research* 95.3 (1996), pp. 649–655. ISSN: 0377-2217. DOI: https://doi.org/10.

1016/0377-2217(95)00300-2. URL: http://www.sciencedirect.com/science/article/pii/0377221795003002.

[Chi11] Joanna Chimiak-Opoka. "Measuring UML Models Using Metrics Defined in OCL within the SQUAM Framework". In: *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*. 2011, pp. 47–61.

[CK94] S. R. Chidamber and C. F. Kemerer. "A Metrics Suite for Object Oriented Design". In: *IEEE Trans. Softw. Eng.* 20.6 (June 1994), pp. 476–493. ISSN: 0098-5589.

[CSC06] Steve Counsell, Stephen Swift, and Jason Crampton. "The interpretation and utility of three cohesion metrics for object-oriented design". In: *ACM Trans on Software Engineering and Methodology*. 15.2 (2006), pp. 123–149. DOI: 10.1145/1131421.1131422. URL: https://doi.org/10.1145/1131421.1131422.

[Dan09] Duc-Hanh Dang. "On integrating triple graph grammars and OCL for model-driven development". PhD thesis. University of Bremen, 2009. URL: http://elib.suub.uni-bremen.de/diss/docs/00011595.pdf.

[DG18a] Khanh-Hoang Doan and Martin Gogolla. "Assessing UML Model Quality by Utilizing Metrics". In: *11th International Conference on the Quality of Information and Communications Technology, QUATIC 2018, Coimbra, Portugal, September 4-7, 2018*. 2018, pp. 92–100. DOI: 10.1109/QUATIC.2018.00022. URL: https://doi.org/10.1109/QUATIC.2018.00022.

[DG18b] Khanh-Hoang Doan and Martin Gogolla. "Extending a UML and OCL Tool for Meta-Modeling: Applications towards Model Quality Assessment". In: *Modellierung 2018*. Ed. by Ina Schaefer et al. GI, LNI 280, 2018, pp. 135–150.

[DG18c] Khanh-Hoang Doan and Martin Gogolla. "Logical Reasoning with Object Diagrams in a UML and OCL Tool". In: *Diagrammatic Representation and Inference - 10th International Conference, Diagrams 2018, Edinburgh, UK, June 18-22, 2018, Proceedings*. 2018, pp. 774–778. DOI: 10.1007/978-3-319-91376-6_79. URL: https://doi.org/10.1007/978-3-319-91376-6%5C_79.

[DG19] Khanh-Hoang Doan and Martin Gogolla. "Quality Improvement for UML and OCL Models Through Bad Smell and Metrics Definition". In: *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*. 2019, pp. 774–778. DOI: 10.1109/MODELS-C.2019.00121. URL: https://doi.org/10.1109/MODELS-C.2019.00121.

[DGH16] Khanh-Hoang Doan, Martin Gogolla, and Frank Hilken. "Towards a Developer-Oriented Process for Verifying Behavioral Properties in UML and OCL Models". In: *Software Technologies: Applications and Foundations - STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO, SEMS, VeryComp, Vienna, Austria, July 4-8, 2016, Revised Selected Papers*. 2016, pp. 207–220. DOI: 10.1007/978-3-319-50230-4_15. URL: https://doi.org/10.1007/978-3-319-50230-4%5C_15.

[Dra16] Dirk Draheim. "Reflective Constraint Writing". In: *Special Issue on Database- and Expert-Systems Applications on Transactions on Large-Scale Data- and Knowledge-Centered Systems XXIV - Volume 9510*. Springer-Verlag New York, Inc., 2016, pp. 1–60. ISBN: 978-3-662-49213-0. DOI: 10.1007/978-3-662-49214-7_1. URL: http://dx.doi.org/10.1007/978-3-662-49214-7_1.

[Elw+05] Mohamed Aly M. El-wakil et al. "A novel approach to formalize Object - Oriented Design Metrics". In: *Conf. Evaluation and Assessment in Software Eng*. 2005.

[Fer+12] Kecia Aline M. Ferreira et al. "Identifying thresholds for object-oriented software metrics". In: *Journal of Systems and Software* 85.2 (2012), pp. 244–257. DOI: 10.1016/j.jss.2011.05.044. URL: https://doi.org/10.1016/j.jss.2011.05.044.

[Fer+16] Eduardo Fernandes et al. "A review-based comparative study of bad smell detection tools". In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, EASE 2016, Limerick, Ireland, June 01 - 03, 2016*. 2016, 18:1–18:12. DOI: 10.1145/2915970.2915984. URL: https://doi.org/10.1145/2915970.2915984.

[FS15] Tarcísio G. S. Filó and Mariza Andrade da Silva Bigonha. "A Catalogue of Thresholds for Object-Oriented Software Metrics". In: *Proceedings of the Advances and Trends in Software Engineering, SOFTENG 2015, April 19-24, 2015, Barcelona, Spain*. 2015.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. "USE: A UML-based Specification Environment for Validating UML and OCL". In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 27–34. ISSN: 0167-6423.

[Gen02] Marcela Genero. *Defining and validating metrics for conceptual models (Ph.D. thesis)*. University of Castilla-La Mancha, Jan. 2002.

[GHD18] Martin Gogolla, Frank Hilken, and Khanh-Hoang Doan. "Achieving model quality through model validation, verification and exploration". In: *Computer Languages, Systems & Structures* 54 (2018), pp. 474–511. DOI: 10.1016/j.cl.2017.10.001. URL: https://doi.org/10.1016/j.cl.2017.10.001.

[Gog15] Martin Gogolla. "Experimenting with Multi-Level Models in a Two-Level Modeling Tool". In: *Proc. 2nd Int. Workshop Multi-Level Modelling co-located with MoDELS 2015*. 2015, pp. 3–12.

[GPC05] Marcela Genero, Mario Piattini, and Coral Caleron. "A survey of metrics for UML class diagrams". In: *Journal of Object Technology* 4 (9 2005), pp. 59–92.

[HCN98] R. Harrison, S. Counsell, and R. Nithi. "Coupling metrics for object-oriented design". In: *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No.98TB100262)*. 1998, pp. 150–157. DOI: 10.1109/METRIC.1998.731240.

[HGW11] Steffen Herbold, Jens Grabowski, and Stephan Waack. "Calculation and optimization of thresholds for sets of software metrics". In: *Empirical Software Engineering* 16.6 (2011), pp. 812–841. DOI: 10.1007/s10664-011-9162-z. URL: https://doi.org/10.1007/s10664-011-9162-z.

[HW02] Terence J. Harmer and F. George Wilkie. "An Extensible Metrics Extraction Environment for Object-Oriented Programming Languages". In: *2nd IEEE International Workshop on Source Code*

Analysis and Manipulation (SCAM 2002), 1 October 2002, Montreal, Canada. 2002, pp. 26–35. DOI: 10.1109/SCAM.2002.1134102. URL: https://doi.org/10.1109/SCAM.2002.1134102.

[IGS14] Muzaffar Igamberdiev, Georg Grossmann, and Markus Stumptner. "An Implementation of Multi-Level Modelling in F-Logic". In: *Proc. Workshop Multi-Level Modelling co-located with MoDELS 2014*. 2014, pp. 33–42.

[ISO01] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[Jac93] Ivar Jacobson. "Object-Oriented Software Engineering - a Use Case Driven Approach". In: *TOOLS 1993: 10th International Conference on Technology of Object-Oriented Languages and Systems, Versailles, France, Europe*. 1993, p. 333.

[JTS14] Renato Correa Juliano, Bruno A. N. Travençolo, and Michel S. Soares. "Detection of Software Anomalies Using Object-oriented Metrics". In: *ICEIS 2014 - Proceedings of the 16th International Conference on Enterprise Information Systems, Volume 2, Lisbon, Portugal, 27-30 April, 2014*. 2014, pp. 241–248. DOI: 10.5220/0004889102410248. URL: https://doi.org/10.5220/0004889102410248.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained - the Model Driven Architecture: practice and promise*. Addison Wesley object technology series. Addison-Wesley, 2003. ISBN: 978-0-321-19442-8.

[LG10] Juan de Lara and Esther Guerra. "Deep Meta-modelling with MetaDepth". In: *Proc. 48th Int. Conf. TOOLS 2010*. 2010, pp. 1–20.

[LGL14] Jesús J. López-Fernández, Esther Guerra, and Juan de Lara. "Assessing the Quality of Meta-Models". In: *Proc. 11th Workshop MoDeVVa@MODELS 2014*. 2014, pp. 3–12.

[LH93] Wei Li and Sallie Henry. "Object-oriented Metrics That Predict Maintainability". In: *J. Syst. Softw.* 23.2 (Nov. 1993), pp. 111–122. ISSN: 0164-1212. DOI: 10.1016/0164-1212(93)90077-B. URL: http://dx.doi.org/10.1016/0164-1212(93)90077-B.

[Li98] Wei Li. "Another metric suite for object-oriented programming". In: *Journal of Systems and Software* 44.2 (1998), pp. 155–162. DOI: 10.1016/S0164-1212(98)10052-3. URL: https://doi.org/10.1016/S0164-1212(98)10052-3.

[LK94] Mark Lorenz and Jeff Kidd. *Object-oriented Software Metrics: A Practical Guide*. Prentice-Hall, Inc., 1994. ISBN: 0-13-179292-X.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. ISBN: 0135974445.

[Mie+14] Simon Van Mierlo et al. "Multi-Level Modelling in the Modelverse". In: *Proc. Workshop Multi-Level Modelling co-located with MoDELS 2014*. 2014, pp. 83–92.

[MP06] Jacqueline A. Mcquillan and James F. Power. *A definition of the Chidamber and Kemerer metrics suite for the Unified Modeling Language*. Tech. rep. National University of Ireland, 2006.

[NMS17] Jeroen Noten, Josh Mengerink, and Alexander Serebrenik. "A data set of OCL expressions on GitHub". In: *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. 2017, pp. 531–534. DOI: 10.1109/MSR.2017.52. URL: https://doi.org/10.1109/MSR.2017.52.

[Obj06] Object Management Group – OMG. *OMG: Object Constraint Language, version 2.0*. 2006. URL: http://www.omg.org/spec/OCL/2.0/.

[Obj11a] Object Management Group – OMG. *OMG Unified Modeling Language(OMG UML), Infrastructure, version 2.4.1*. 2011. URL: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF.

[Obj11b] Object Management Group – OMG. *OMG Unified Modeling Language(OMG UML), Superstructure, version 2.4.1*. 2011. URL: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.

[Obj15a] Object Management Group – OMG. *OMG Meta Object Facility (MOF) Core Specification, version 2.5*. 2015. URL: http://www.omg.org/spec/MOF/.

[Obj15b] Object Management Group – OMG. *Unified Modeling Language Specification, version 2.5*. 2015. URL: http://www.omg.org/spec/UML/.

[Ric02] Mark Richters. "A precise approach to validating UML models and OCL constraints". PhD thesis. University of Bremen, Germany, 2002. ISBN: 978-3-89722-842-9. URL: http://d-nb.info/963496360.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. 1st. Addison Wesley Longman Publishing Co., Inc., 1996. ISBN: 020163385X.

[RJB05] James E. Rumbaugh, Ivar Jacobson, and Grady Booch. *The unified modeling language reference manuel - covers UML 2.0, Second Edition*. Addison Wesley object technology series. Addison-Wesley, 2005. ISBN: 978-0-321-24562-5.

[RN88] Joe Rodgers and Alan Nicewander. "Thirteen Ways to Look at the Correlation Coefficient". In: *American Statistician - AMER STATIST* 42 (Feb. 1988), pp. 59–66. DOI: 10.1080/00031305.1988.10475524.

[Rum+91] James E. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991. ISBN: 0-13-630054-5.

[Saa08] Thomas Saaty. "Decision making with the Analytic Hierarchy Process". In: *International Journal of Services Sciences* 1 (Jan. 2008), pp. 83–98.

[Sch12] Huck Schuyler. *Reading statistics and research*. Pearson, Boston, MA, USA, 2012.

[Sei03] Ed Seidewitz. "What Models Mean". In: *IEEE Software* 20.5 (2003), pp. 26–32. DOI: 10.1109/MS.2003.1231147. URL: https://doi.org/10.1109/MS.2003.1231147.

[Sha+10] Raed Shatnawi et al. "Finding software metrics threshold values using ROC curves". In: *Journal of Software Maintenance* 22.1 (2010), pp. 1–16. DOI: 10.1002/smr.404. URL: https://doi.org/10.1002/smr.404.

[Sha10] Raed Shatnawi. "A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems". In: *IEEE Trans. Software Eng.* 36.2 (2010), pp. 216–225. DOI: 10.1109/TSE.2010.9. URL: https://doi.org/10.1109/TSE.2010.9.

[SSS14] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014.

[Str+16] Misha Strittmatter et al. "Challenges in the Evolution of Metamodels: Smells and Anti-Patterns of a Historically-Grown Metamodel". In: *Proceedings of the 10th Workshop on Models and Evolution co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 2, 2016*. 2016, pp. 30–39. URL: http://ceur-ws.org/Vol-1706/paper5.pdf.

[Tor+13] Marco Torchiano et al. "Relevance, benefits, and problems of software modelling and model driven techniques - A survey in the Italian industry". In: *J. Syst. Softw.* 86.8 (2013), pp. 2110–2126. DOI: 10.1016/j.jss.2013.03.084. URL: https://doi.org/10.1016/j.jss.2013.03.084.

[Uni07] Database Systems Group Bremen University. *USE: A UML based Specification Environment*. Tech. rep. University of Bremen, 2007. URL: http://www.db.informatik.uni-bremen.de/projects/USE/use-documentation.pdf.

[Wil+13] James R. Williams et al. "What do Metamodels Really Look Like?" In: *Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, USA, October 1, 2013*. 2013, pp. 55–60. URL: http://ceur-ws.org/Vol-1078/paper7.pdf.

[Wüs] Jürgen Wüst. *SD Metrics*. https://www.sdmetrics.com/index.html. Accessed: 2019-06-05.

Appendix A

Catalog of Metrics Definition

This appendix represents detailed definition of the metrics used in the evaluation in Chapter A.1. For each metric, both formal definition and OCL definition are presented.

A.1 Auxiliary Functions

This section depicts a list of auxiliary functions, which are utilized for metrics definition presenting in next section. These functions are defined as operations of `ClassMetrics` metaclass at metalevel.

```
--------------------0. Auxiliary operations--------------------
--get local defined operations
localOperations(): Set(Operation) =
  self.class.ownedOperation

--get local defined attributes
localAttributes(): OrderedSet(Property) =
  self.class.ownedAttribute

--get a set of associations that the class is a part of
allAssociations(): Set(Association) =
  self.class.association

--get all inherited attributes
inheritedAttributes(): Set(NamedElement) =
  self.class.inheritedMember→select(m|m.oclIsTypeOf(Property))
```

```
--get all inherited operations
inheritedOperations(): Set(NamedElement) =
  self.class.inheritedMember→select(m|m.oclIsTypeOf(Operation))

--get attributes that have another class as their type
userdefinedTypeAttributes(): OrderedSet(Property) =
  localAttributes()→select(att|att.type.oclIsTypeOf(Class))

--get parameters that have another class as their type
userdefinedTypeParameters(): Bag(Parameter) =
  localOperations()→collect(op|op.ownedParameterRedefined)
    →select(att|att.type.oclIsTypeOf(Class))

--check if a class c is coupled with the class under consideration
--by attribute declaration
isCoupledbyAttribute(c: Class): Boolean =
  userdefinedTypeAttributes()→collect(att|att.type)→includes(c)

--check if a class c is coupled with the class under consideration
--by methods' parameters declaration
isCoupledbyParameter(c: Class): Boolean =
  userdefinedTypeParameters()→collect(pa|pa.type)→includes(c)

--check if a class c is coupled with the class under consideration
--by an operation's return type
isCoupledbyReturnType(c: Class): Boolean =
  localOperations()→collect(op|op.type)→includes(c)

--check if a class c is coupled with the class under consideration
--by an association
isCoupledbyAssociation(c: Class): Boolean =
  allAssociations()→collect(as|as.endType)→includes(c)

--get all ancestors of the corresponding class
allAncestors(): Set(Class) =
  self.class.superClass→closure(superClass)

--get all descendants of the corresponding class
allDescendants(): Set(Class) =
```

```
  self.class.subClass→closure(subClass)

--get immediate descendants of the corresponding class
children(): Set(Class) =
  self.class.subClass

--get immediate ancestors of the corresponding class
parents(): Set(Class) =
  self.class.superClass

--get all ancestors of the corresponding class
allAncestors() : Set(Class) =

 self.oclAsType(ClassMetrics).class.superClass→closure(superClass)

--get all descendants of the corresponding class
allDescendants() : Set(Class) =
  self.oclAsType(ClassMetrics).class.subClass→closure(subClass)

--get the maximum length of all inheritance trees from class c
--to the root classes
getMaxInheritancetree(c: Class): Integer =
  if c.metrics.parents()→size() = 0 then
    0
  else
    c.metrics.parents()
      →collect(pc|pc.metrics.getMaxInheritancetree(pc))→max() + 1
  endif

--get the number of hierarchies (inheritance trees) from class c
--to the root classes
getNumberofHierarchies(c: Class): Integer =
  if c.metrics.parents()→size() = 0 then
    if c.metrics.children()→size() = 0 then
      0
    else
      1
    endif
  else
```

```
    c.metrics.parents()
      →collect(pc|pc.metrics.getNumberofHierarchies(pc))→sum()
  endif
```

A.2 Metrics Catalog

To provide a more precise definition, in this catalog we present the definition of metrics in a more formal expression using our formal definition of the class model presented in Chapter 2. First of all, let $\mathcal{CM}$ is a class model defined as follows

$$\mathcal{CM} = (\mathcal{C}, \mathcal{T}, \mathcal{P}, \mathcal{OP}, \mathcal{A}, \mathcal{G}, Mappings, Constraints)$$

1. **NOIC**: Number of Inherited Classes
   - Definition: This metric computes the total number of classes which inherit from at least one super class.

   $$\text{NOIC}(\mathcal{CM}) = |c_i| \text{ where } c_i \in \mathcal{C} \text{ and } \exists c_j \in \mathcal{C}, c_i \prec c_j$$

   - OCL definition:

   ```
   ModelMetrics :: NOIC(): Integer =
   Class.allInstances()→select(c|
       not c.parents()→isEmpty())→size()
   ```

   - Scope: Model
   - Type: Size
   - Viewpoint: This metric indicates the adoption of inheritance mechanism. A high value of NOID metric points out the inheritance mechanism is significantly used in the model under consideration.

2. **NFLC**: Number of immediately Featureless Classes
   - Definition: NFLC metric is the total number of concrete classes which have no attributes and do not involve in any association. These class are identified as "immediately featureless" because they may inherit features from its superclasses.

   $$\text{NFLC}(\mathcal{CM}) = |c_i|$$

   where $c_i \in \mathcal{C}$, $\nexists p_i \in \mathcal{P}, c_i \Rightarrow p_i$ and $\nexists a_i \in \mathcal{A}, a \rightarrow c_i$
   - OCL definition:

   ```
   ModelMetrics :: NFLC(): Integer =
   Class.allInstances()→select(c|c.isAbstract = false
   ```

```
and c.ownedAttribute→isEmpty()
and c.association→isEmpty)→size()
```

- Scope: Model
- Type: Size
- Viewpoint: This metric measures the structural feature reusability. A high value of NFLC metric indicates the structural features, i.e., attributes and associations, are remarkably reused through inheritance hierarchy.

3. **DSC**: Total number of classes in the design [BD02]
   - Definition: This metric counts the total number of classes in the class model.

$$\text{DSC}(\mathcal{CM}) = |c_i| \text{ where } c_i \in \mathcal{C}$$

   - OCL definition:

```
ModelMetrics :: DSC(): Integer =
  Class.allInstances()→size()
```

   - Scope: Model
   - Type: Size
   - Viewpoint: This metric measures the size of a model. DSC metric indicates the complexity of the model based on the number of classes in the model.

4. **NAssoc**: Number of total association [Gen02]
   - Definition: This metric counts the total number of associations within the class model.

$$\text{NAssoc}(\mathcal{CM}) = |a_i| \text{ where } a_i \in \mathcal{A}$$

   - OCL definition:

```
ModelMetrics :: NAssoc(): Integer =
  Association.allInstances()→ size()
```

   - Scope: Model
   - Type: Size
   - Viewpoint: NAssoc metric indicates the complexity of the model based on the number of associations connecting classes in a model. The high value of NAssoc means classes in the model strongly connecting together.

5. **MaxDIT**: Maximum depth of inheritance [Gen02]
   - Definition: This metric indicates the maximum value among the DIT metric of all classes in the class model

$$\text{MaxDIT}(\mathcal{CM}) = \text{Max}(\text{DIT}(c_i)) \forall c_i \in \mathcal{C}$$

   - OCL definition:

```
ModelMetrics :: MaxDIT(): Integer =
  Class.allInstances()→collect(c|c.metrics.DIT())→max()
```

   - Scope: Model
   - Type: Complexity
   - Viewpoint: The low value of MaxDIT (0 or 1) indicates the poor usage of inheritance in the design. The high value of MaxDIT (more than 5, for example) may increase the complexity of the model.

6. **NOH**: Number of hierarchies [Gen02]
   - Definition: This metric counts the total number of class hierarchies within the class model. It can be calculated by the summation of the number of inheritance trees from all root classes (classes are on the top of inheritance trees)
   - OCL definition:

```
ModelMetrics :: NOH(): Integer =
  Class.allInstances()→select(c|c.metrics.children()
      →size() = 0)→collect(lc|lc.metrics.NOH())→sum()
```

   - Scope: Model
   - Type: Complexity
   - Viewpoint: NOH metric indicates the complexity of the model based on the number of hierarchy trees in a model.

7. **AIF**: Attribute Inheritance Factor [AM96]
   - Definition: This metric is defined as a quotient between the sum of inherited attributes of all classes and the sum of the total number of available attributes (inherited ones plus locally defined ones) of all classes.

$$AIF(\mathcal{CM}) = \frac{\sum_{i=1}^{TC} A_i(c_i)}{\sum_{i=1}^{TC} A_a(c_i)}, c_i \in \mathcal{C}$$

where TC is the total number of classes in the class model, $A_i(c_i)$ is the total number of attributes that class c_i inherited from its superclasses, and $A_a(c_i)$ is the total available attributes of class c_i.
- OCL definition:

```
ModelMetrics :: AIF(): Real =
  (Class.allInstances()→collect(c|c.metrics.NAI())→sum())/
  (Class.allInstances()→collect(c|c.metrics.NAI()
        + c.metrics.NOA())→sum())
```

- Scope: Model
- Type: OO design
- Viewpoint: AIF metric indicates how much the attribute inheritance mechanism is used throughout all classes in the design. A class which inherits many attributes from its ancestor classes contributes to a high AIF.

8. **MIF**: Method Inheritance Factor [AM96]
   - Definition: This metric is defined as a ratio between the sum of the total number of inherited methods of all classes and the sum of the total number of available methods (locally defined within the class and inherited from other classes) of all classes.

$$MIF(\mathcal{CM}) = \frac{\sum_{i=1}^{TC} M_i(c_i)}{\sum_{i=1}^{TC} M_a(c_i)}, c_i \in \mathcal{C}$$

   where TC is the total number of classes in the class model, $M_i(c_i)$ is the total number of methods that class c_i inherited from its superclasses, and $A_a(c_i)$ is the total available methods of class c_i.
   - OCL definition:

```
ModelMetrics :: MIF(): Real =
  (Class.allInstances()→collect(c|c.metrics.NMI())→sum())/
  (Class.allInstances()→collect(c|c.metrics.NMI()
        + c.metrics.NOM())→sum())
```

   - Scope: Model
   - Type: OO design
   - Viewpoint: MIF metric indicates how much the method inheritance

mechanism are using throughout all classes in the design. A class which inherits many attributes from its ancestor classes contributes to a high MIF.

9. **AHF**: Attribute Hiding Factor [AM96]
   - Definition: This metric is a quotient between the sum of the invisibilities of all attributes defined in all classes and the total number of attributes in the model. The invisibility of an attribute is calculated as the percentage of total classes from which the attribute is hidden.

$$AHF(\mathcal{CM}) = \frac{\sum_{i=1}^{TC}\sum_{m=1}^{A_d(c_i)} invis(A_{mi})}{\sum_{i=1}^{TC} A_d(c_i)}, c_i \in \mathcal{C}$$

   where TC is the total number of classes in the class model, $A_d(c_i)$ = number of attributes defined in class c_i and $invis(M_{mi})$ is the invisibility of method M_{mi}.

$$invis(A_{mi}) = \begin{cases} 0 & \text{if } A_{mi} \text{ is defined as public} \\ 1 & \text{if } A_{mi} \text{ is defined as private} \\ DC(c_i)/(TC-1) & \text{if } M_{mi} \text{ is defined as protected} \end{cases}$$

   where $DC(c_i)$ is the number of descendants of class c_i.
   - OCL definition:

```
ModelMetrics :: AHF():Real =
(Class.allInstances()→collect(c|c.ownedAttribute→collect(a|
      if(a.visibility = VisibilityKind::public) then 0
      else if(a.visibility = VisibilityKind::private) then 1
         else c.metrics.NDC()/(Class.allInstances()→size()-1)
      endif
   endif
   )→sum())→sum())
   /(Class.allInstances()→collect(c|c.metrics.NOA())→sum())
```

   - Scope: Model
   - Type: OO design
   - Viewpoint: AHF metric measures the level of encapsulation applied in a model. A high value of AHF means most of the attributes are private and a low value of AHF indicates most of the attributes are public.

10. **MHF**: Method Hiding Factor [AM96]
    - Definition: This metric is a quotient between the sum of the invisibilities of all methods defined in all classes and the total number of methods in the model. The invisibility of a method is calculated as the percentage of total classes from which the method is hidden.

$$MHF(\mathcal{CM}) = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(c_i)} invis(M_{mi})}{\sum_{i=1}^{TC} M_d(c_i)}, c_i \in \mathcal{C}$$

    where TC is the total number of classes in the class model, $M_d(c_i) =$ number of methods defined in class c_i and $invis(M_{mi})$ is the invisibility of method M_{mi}.

$$invis(M_{mi}) = \begin{cases} 0 & \text{if } M_{mi} \text{ is a public method} \\ 1 & \text{if } M_{mi} \text{ is a private method} \\ DC(c_i)/(TC-1) & \text{if } M_{mi} \text{ is a protected method} \end{cases}$$

    where $DC(c_i)$ is the number of descendants of class c_i.
    - OCL definition:

```
  ModelMetrics :: MHF():Real =
(Class.allInstances()→collect(c|c.ownedOperation→collect(op|
     if(op.visibility = VisibilityKind::public) then 0
     else if(op.visibility = VisibilityKind::private) then 1
         else
 c.metrics.NDC()/(Class.allInstances()→size()-1)
         endif
   endif
     )→sum())→sum()
 )/(Class.allInstances()→collect(c|c.metrics.NOM())→sum())
```

    - Scope: Model
    - Type: OO design
    - Viewpoint: MHF metric measures the level of encapsulation applied in a model. The high value of MHF means most of the methods are private, which indicates very little functionality of them. The low value of MHF indicates most of the methods are public, which means they are unprotected.

11. **PF**: Polymorphism Factor [AM96]

- Definition: This metric represents the degree of possibility of polymorphic situations over all classes in a model.

$$PF(\mathcal{CM}) = \frac{\sum_{i=1}^{TC} M_o(c_i)}{\sum_{i=1}^{TC}(M_n(c_i) * DC(c_i))}, c_i \in \mathcal{C}$$

where TC is the total number of classes in the class model, $M_o(c_i)$ is the total number of overriding methods in class c_i, $M_n(c_i)$ is the total number of new methods in class c_i and $DC(c_i)$ is the number of descendants of class c_i.

- OCL definition:

```
ModelMetrics :: PF(): Real =
  (Class.allInstances()→collect(c|c.metrics.NMO())→sum())/
  (Class.allInstances()→collect(c|(c.metrics.NOM()
        - c.metrics.NMO()) * c.metrics.NDC())→sum())
```

- Scope: Model
- Type: OO design
- Viewpoint: PF metric shows the level of polymorphism OO mechanism is used throughout all classes in the design.

12. **A**: Abstractness [Mar03]
    - Definition: This metric measures the degree of abstraction in a class model. It represents the ratio between the number of abstract classes and the total number of classes in a model.

$$A(\mathcal{CM}) = \frac{T_{abstract}}{DSC(\mathcal{CM})}, c_i \in \mathcal{C}$$

where $T_abstract$ is the total number of abstract classes, $DSC(\mathcal{CM})$ is the total number of classes in a model.

- OCL definition:

```
ModelMetrics :: A(): Real =
  (Class.allInstances()→select(c|c.isAbstract = true)
       →size())/DSC()
```

- Scope: Model
- Type: OO design
- Viewpoint: The high value of this metric indicates high number of

abstract classes in the model. This will increase of the abstractness of the design, and as the result, increase the understandability and the maintainability.

13. **NOA**: Number of Attributes per Class [BHe96]
    - Formal definition: This metric counts the total number of attributes in a class.

$$\text{NOA}(c) = |p_i| \text{ where } c \Rightarrow p_i,\ c \in \mathcal{C}, p_i \in \mathcal{P}$$

    - OCL definition:

```
ClassMetrics :: NOA(): Integer =
  localAttributes()→size()
```

    - Scope: Class
    - Type: Size
    - Viewpoint: This metric indicates the complexity of a class in terms of the information it stores. A class knows to much if it has too many attributes (greater than 10, for example). This kind of class should be separated into more than one class.

14. **NOM**: Number of Local Methods [LH93]
    - Definition: This metric counts the total number of methods locally defined in a class. The original definition does not state about the visibility of the methods, therefore, we assume that this metric counts all the methods, regardless of the visibility of methods.

$$\text{NOM}(c) = |op_i| \text{ where } c \Rightarrow op_i,\ c \in \mathcal{C}, op_i \in \mathcal{OP}$$

    - OCL definition:

```
ClassMetrics :: NOM(): Integer =
  localOperations()→size()
```

    - Scope: Class
    - Type: Size
    - Viewpoint: This metric indicates the complexity of a class in terms of responsibilities. A class does too much if it has too many methods (greater than 20, for example). This kind of class should be separated into more than one class.

15. **SIZE2**: Number of Attributes + Number of local methods [LH93]
    - Definition: This metric counts the total number of methods plus the

total number of methods defined in a class.

$$\text{SIZE2}(c) = \text{NOM}(c) + \text{NOA}(c),\ c \in \mathcal{C}$$

- OCL definition:

```
ClassMetrics :: SIZE2(): Integer =
  NOA() + NOM()
```

- Scope: Class
- Type: Size
- Viewpoint: This metric measures the size of a class. A class which has too many methods and attributes (more than 60, for example) is considered as a "God class" [Rie96], a class which knows too much or does too much. God class is an example of anti-patterns.

16. **APPM**: Average Parameters per Method [LK94]
- Definition: This metric counts the average number of parameters per method in a class.

$$APPM(c) = \frac{\sum_{i=1}^{m} paras(op_i)}{NOM(c)}, c \in \mathcal{C}$$

where $paras(op_i)$ is the number of parameters of method op_i - OCL definition:

```
ClassMetrics :: APPM(): Real =
  localOperations()→collect(op|op.ownedParameterRedefined
    →size())→sum()/NOM()
```

- Scope: Class
- Type: Size
- Viewpoint: This metric might indicates the complexity of the methods in a class. A high value of APPM metric points out the methods seem to do a lot of work and the class seems highly coupled with other classes.

17. **NOC**: Number of Children [CK94]
- Definition: This metric measures the number of immediate subclasses of a class.

$$\text{NOC}(c) = |c_i| \text{ where } c_i \prec c,\ c, c_i \in \mathcal{C}$$

- OCL definition:

```
ClassMetrics :: NOC(): Integer =
  children()→size()
```

- Scope: Class
- Type: Complexity
- Viewpoint: A high value of NOC increases the reusability of class, however, it also increases the responsibility of the class. Any change in the class can affect many other sub-classes. Therefore, more testing may require for the class with a high value of NOC. An empirical validation from [BMB96] shows that the higher value of NOC, the lower the probability of fault detection.

18. **DIT**: Depth of Inheritance Tree [CK94]
    - Definition: This metric measures the maximum length from a class to root classes through all hierarchy tree the class involves. - OCL definition:

```
ClassMetrics :: DIT(): Integer =
  getMaxInheritancetree(self.class)
```

    - Scope: Class
    - Type: Complexity
    - Viewpoint: A high value of DIT indicates a large number of ancestor classes this class potentially inherit from. Brian et al. [BMB96] has conducted an empirical validation, and conclude that the higher value of DIT increasing the probability of fault detection.

19. **NAC**: Number of Ancestor Classes [Li98]
    - Definition: This metric counts the total number of ancestor classes from which the given class inherits.

$$\text{NAC}(c) = |c_i| \text{ where } c_i \text{ is an ancestor of } c,\ c, c_i \in \mathcal{C}$$

    - OCL definition:

```
ClassMetrics :: NAC(): Integer =
  allAncestors()→size()
```

    - Scope: Class
    - Type: Complexity

- Viewpoint: NAC can be an alternative to DIT metric. A high value of NAC indicates the large number of classes that can influence the given class through inheritance relations, therefore it reduces the understandability of the class.

20. **NDC**: Number of Descendant Classes [Li98]
    - Definition: This metric counts the total number of descendant classes (subclasses) of the given class inherits.

      $$\text{NDC}(c) = |c_i| \text{ where } c_i \text{ is a descendant of } c,\ c, c_i \in \mathcal{C}$$

    - OCL definition:

```
ClassMetrics :: NDC(): Integer =
  allDescendants()→size()
```

    - Scope: Class
    - Type: Complexity
    - Viewpoint: NDC is an extension of the NOC metric. A high value of NDC indicates the large number of classes that may potentially be influenced by the given class through inheritance relations, therefore it negatively affects the changeability and maintainability of the class.

21. **SIX**: Specialization Index [LK94]
    - Definition: This metric can be calculated as follows.

      $$SIX(c) = \frac{NMO(c) * DIT(c)}{NOM(c) + NMI(c)}, c \in \mathcal{C}$$

    - OCL definition:

```
ClassMetrics :: SIX(): Real =
  (NMO()*DIT())/(NOM()+NMI())
```

    - Scope: Class
    - Type: OO Design
    - Viewpoint: This metric evaluates the OO mechanism quality of a subclass. A high value of SIX may show a poor subclass concerning to OO design mechanism.

22. **NMI**: Number of Methods Inherited [LK94]
    - Definition: This metric counts the total number of methods a class

inherits from its ancestors. The overridden methods are excluded.

$$\text{NMI}(c) = \text{IM}(c) \text{ - } \text{NMO}(c)$$

where IM(c) is the total methods that class c can inherits from its ancestor.

- OCL definition:

```
ClassMetrics :: NMI(): Integer =
  inheritedOperations()
    →select(op|not self.class.ownedOperation.name
      →includes(op.name))→size()
```

- Scope: Class
- Type: OO Design
- Viewpoint: The high value of NMI indicates a good usage of inheritance.

23. **NMO**: Number of Methods Overridden [LK94]
    - Definition: This metric counts the total number of overridden methods in a class. An overridden method is a method which overrides a method from a superclass.

$$\text{NMO}(c) = |op_i|$$

where $c \Rightarrow op_i$, op_i redefines a method op_j of a superclass of (c), $c \in \mathcal{C}, op_i, op_j \in \mathcal{OP}$.

- OCL definition:

```
ClassMetrics :: NMO(): Integer =
  inheritedOperations()
    →select(op|self.class.ownedOperation.name
      →includes(op.name))→size()
```

- Scope: Class
- Type: OO Design
- Viewpoint: A high value of NMO may indicate poor subclassing, since the subclass does not inherit to many function from its superclasses.

24. **CBO**: Coupling Between Object [CK94]
    - Definition: CBO is computed as the number of classes that are coupled to a given class. Two classes are coupled if they interact with each other. This metric is originally defined at the code level, and the coupling can appear when one class use methods or attributes defined by another

class. This information is not available at the model level, therefore, this metric can only be approximately measured at the model level as follows.

$$\text{CBO}(c) = |c_i|, \text{isCoupled}(c, c_i)$$

where $\text{isCoupled}(c, c_i)$ = true when (1)c and c_i are coupled by attribute declaration, (2)c and c_i are coupled by parameter declaration, (3)c and c_i are coupled by method return type declaration, (4)c and c_i are coupled by an association.

- OCL definition:

```
ClassMetrics :: CBO(): Integer =
  Class.allInstances()→excluding(self.class)→select(c|
    isCoupledbyAttribute(c) or isCoupledbyParameter(c) or
    isCoupledbyReturnType(c) or isCoupledbyAssociation(c))
      →size()
```

- Scope: Class
- Type: Coupling
- Viewpoint: This metric shows the dependency of a given class to other classes. A value 0 indicates that a class has no relationship with any class in the system, and it can be considered as an isolated class, one type of design smell. A high value of CBO indicates that the class is too strongly coupled with other classes in the design. This reduces the possibility of class reuse.

25. **DAC**: Data Abstraction Coupling [LH93]

- Definition: This metric measures the number of attributes that have another class as their type. The attributes inherit from superclasses are excluded when computing this metric.

$$\text{DAC}(c) = |p_i| \text{ where } c \Rightarrow p_i, type(p_i) \in \mathcal{C}, c \in \mathcal{C}, p_i \in \mathcal{P}$$

- OCL definition:

```
ClassMetrics :: DAC(): Integer =
  userdefinedTypeAttributes()→size()
```

- Scope: Class
- Type: Coupling
- Viewpoint: This metric shows how this class is coupled with other classes in the model through data declaration. A higher value of DAC indicates more complexity in data structures and classes.

26. **DAC1**: Data Abstraction Coupling 1 [LH93]
    - Definition: This metric measures the number of different classes that are used as types of attributes in a given class. The attributes inherit from superclasses are excluded when computing this metric.

    $$\text{DAC1}(c) = |c_i| \text{ where } c \Rightarrow p_i, type(p_i) = c_i \in \mathcal{C}, c \in \mathcal{C}, p_i \in \mathcal{P}$$

    - OCL definition:

```
ClassMetrics :: DAC1(): Integer =
  userdefinedTypeAttributes()→collect(type)→asSet()→size()
```

    - Scope: Class
    - Type: Coupling
    - Viewpoint: This metric shows how this class is coupled with other classes in the model through data declaration.

27. **NAS**: Number of Association [HCN98]
    - Definition: This metric counts the number of associations that the given class involves.

    $$\text{NAS}(c) = |a_i| \text{ where } a_i \in \mathcal{A}, a \rightarrow c$$

    - OCL definition:

```
ClassMetrics :: NAS(): Integer =
  allAssociations()→size()
```

    - Scope: Class
    - Type: Coupling
    - Viewpoint: This metric shows how this class is coupled with other classes through associations (inter-class coupling). A class with high value of NAS may depend on many classes and it reduces the maintainability and reuseability of the class.

28. **DCC**: Direct Class Coupling [BD02]
    - Definition: This metric counts the number of classes that directly relates to the given class by attribute declarations or message passing (parameters) in class methods.

    $$\text{DCC}(c) = |c_i| \text{ where isDirectlyCoupled}(c, c_i)$$

    where isDirectlyCoupled(c, c_i) = true when (1)c and c_i are coupled by attribute declaration, (2)c and c_i are coupled by parameter declaration.
    - OCL definition:

```
ClassMetrics :: DCC(): Integer =
  Class.allInstances()→excluding(self.class)→select(c|
    isCoupledbyAttribute(c) or isCoupledbyParameter(c))
      →asSet()→size()
```

- Scope: Class
- Type: Coupling
- Viewpoint: A high value of DCC increases the class complexity, and as the result, it has a negative impact on the understandability, and maintainability of the class. The potential for class reuse also decreases when the value of the DCC metric increases.

29. **CACM**: Cohesion among Methods of a Class [Ban+99]
    - Definition: This metric measures the degree of cohesion between methods within a class based on the parameter types defined in methods.

$$CACM(c) = \frac{a}{kl}, c \in \mathcal{C}$$

    where k is number of methods, l is number of distinct data types using to define parameters in all methods, and a is the summation of the number of distinct parameter types of each method in the class.
    - OCL definition:

```
ClassMetrics :: CAMC(): Real =
  let M = localOperations() in
  let l = M→collect(m|m.ownedParameterRedefined.type)
    →asSet()→size() in
  let a = M→collect(m|m.ownedParameterRedefined.type→asSet()
    →size())→sum() in
  a /(l * M→size())
```

    - Scope: Class
    - Type: Cohesion
    - Viewpoint: A high value of CACM indicates that the methods in the class is reasonably cohesive. Higher cohesion increase the maintainability of the class.

30. **NHD**: Normalized Hamming Distance [CSC06]
    - Definition: This metric measures the degree of cohesion between meth-

ods within a class. It calculates the average parameter agreement between each pair of methods.

$$NHD(c) = 1 - \frac{2}{lk(k-1)} \sum_{j=1}^{l} x_j(k - x_j), c \in \mathcal{C}$$

where k is number of methods, l is number of distinct data types using to define parameters in all methods, and x_j is a is the number of methods that have a parameter of type j.
- OCL definition:

```
ClassMetrics :: NHD(): Real =
  let M = localOperations() in
  let L = M→collect(m|m.ownedParameterRedefined.type)
      →asSet() in
  let tmp = L→iterate(t : Type; S : Integer = 0|
      let x = M→select(m|m.ownedParameterRedefined.type
        →includes(t))→size() in
    S + x * (M→size() - x)
  ) in
  1 - 2/(L→size() * M→size() * (M→size() - 1)) * tmp
```

- Scope: Class
- Type: Cohesion
- Viewpoint: A high value of NHD indicates that the methods in the class is reasonably cohesive. Higher cohesion increase the maintainability and understandability of the class.

31. **MMAC**: Method-Method through Attributes Cohesion (MMAC) [AB10]
- Definition: This metric measures the degree of cohesion between methods within a class. It calculates the average cohesion between each pair of methods.

$$MMAC(c) = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ \frac{\sum_{i=1}^{l}(x_i(x_i-1))}{lk(k-1)} & \text{otherwise} \end{cases}$$

where k is number of methods, l is number of distinct data types using to define parameters in all methods, and x_i is a is the number of methods

that have a parameter or return type of type *j*.
- OCL definition:

```
ClassMetrics :: MMAC():Real =
  let M = localOperations() in
  let L = M→collect(m|m.ownedParameterRedefined.type)
      →asSet() in
  let tmp = L→iterate(t : Type; S : Real = 0|
      let x = M→select(m|m.ownedParameterRedefined.type
        →includes(t))→size() in
    S + x*(x-1)
  ) in
  if (M→size() = 0 or L→size() = 0) then 0
  else if(M→size() = 1) then 1
      else tmp/(L→size()* M→size() * (M→size() - 1))
      endif
  endif
```

- Scope: Class
- Type: Cohesion
- Viewpoint: A high value of MMAC indicates that the methods in the class is reasonably cohesive. Higher cohesion increase the maintainability and understandability of the class.

Appendix B

Catalog of Smells Definition

This appendix represents detailed OCL definitions of the smells evaluated in Chapter A.1.

1. **EP01**: Abstract class is subclass of a concrete class
   - Context: Class
   - OCL definition:

```
e.isAbstract and e.superClass→exists(c|not c.isAbstract)
```

2. **EP02**: Composition cycles
   - Context: Class
   - OCL definition:

```
(e.association.memberEnd
   →select(r|r.type <> e and r.isComposite).type
   →closure(c|c.association.memberEnd
       →select(r|r.type <> c and r.isComposite).type))
           →includes(e)
```

3. **EP03**: Composite end is a super class
   - Context: Association
   - OCL definition:

```
e.memberEnd→exists(
    r1,r2|r1.lower = 1 and r1.isComposite and
        r1.type.oclAsType(Class).allParents()
        →includes(r2.type.oclAsType(Class)))
```

4. **EP04**: Cycle reflexive association
   - Context: Association
   - OCL definition:

```
e.endType→asSet()→size() = 1 and e.memberEnd
    →exists(r1,r2|r1.upper <> -1 and
        r1.upper < r2.lower)
```

5. **EP05**: Isolated class
   - Context: Class
   - OCL definition:

```
e.association→isEmpty() and e.superClass→isEmpty()
    and e.subClass→isEmpty()
```

6. **EP06**: A class is contained in two classes, and the cardinality in the composition end of one of them is 1
   - Context: Class
   - OCL definition:

```
let E = e.association.memberEnd→select(type <> e and
    isComposite = true)
in E→asSet()→size() > 1 and E→exists(lower = 1)
```

7. **BP01**: A class is contained in two classes
   - Context: Class
   - OCL definition:

```
e.association.memberEnd→select(type <> e and
    isComposite = true)→asSet()→size() > 1
```

8. **BP02**: Root class is not an abstract class
   - Context: Class
   - OCL definition:

```
e.superClass→isEmpty() and not e.subClass→isEmpty()
    and e.isAbstract = false and e.association→isEmpty()
```

9. **BP03**: Duplicate attributes among all subclasses
   - Context: Attribute
   - OCL definition:

```
e.class.superClass→exists(c|c.subClass→size() >= 2 and
    c.subClass→forAll(c1|c1.ownedAttribute
    →exists(p|p.name = e.name and p.type = e.type)))
```

10. **BP04**: Abstract class has only one subclass
    - Context: Class
    - OCL definition:

```
e.isAbstract and e.subClass→size() = 1
```

11. **BP05**: Redundant generalization paths
    - Context: Generalization
    - OCL definition:

```
Generalization.allInstances()→exists(g|
    g <> e and g.specific = e.specific and
    (g.general.allParents()→includes(e.general) or
        g.general = e.general))
```

12. **BP06**: Descendant reference
    - Context: Association
    - OCL definition:

```
e.endType→ exists(c1,c2|c1.oclAsType(Class).subClass
    →closure(subClass)→includes(c2.oclAsType(Class)))
```

13. **ME01**: Overloaded class
    - Context: Class
    - OCL definition:

```
e.metrics.NOA() > 10
```

14. **ME02**: A class is really involved in so many associations
    - Context: Class
    - OCL definition:

```
e.metrics.NAS() >= 10
```

15. **ME03**: Deep inheritance hierarchy
    - Context: Class
    - OCL definition:

```
e.metrics.DIT() > 5
```

16. **ME04**: Too many direct children
    - Context: Class
    - OCL definition:

```
e.metrics.NOC() >= 10
```

17. **ME05**: God class in the design
    - Context: Class
    - OCL definition:

```
e.metrics.NOA() + c.metrics.NOM()) > 60
```

18. **NA01**: Class naming PascalCase convention
    - Context: Class
    - OCL definition:

```
e.name.substring(1,1).toUpper() <> e.name.substring(1,1)
```

19. **NA02**: Attributes are named after their feature class
    - Context: Attribute
    - OCL definition:

```
e.class→ notEmpty() and
    e.name.substring(1,e.class.name.size()).toLower() =
        e.class.name.toLower()
```

20. **NA03**: Class name is a Java keyword
    - Context: Class
    - OCL definition:

```
let javaKeywords =
    Set{'abstract','continue','for','new','switch',
     'assert','default','goto','package','synchronized',
     'boolean','do','if','private','this',
     'break','double','implements','protected','throw',
     'byte','else','import','public','throws',
     'case','enum','instanceof','return','transient',
     'catch','extends','int','short','try',
     'char','final','interface','static','void',
     'class','finally','long','strictfp','volatile',
     'const','float','native','super','while', 'true', 'false',
    'null'}
in javaKeywords→includes(e.name))
```

21. **NA04**: Class name is a C++ keyword
    - Context: Class
    - OCL definition:

```
let cppKeywords = Set{'asm','else','new','this',
    'auto','enum','operator','throw',
    'bool','explicit','private','true',
    'break','export','protected','try',
    'case','extern','public','typedef',
    'catch','false','register','typeid',
    'char','float','reinterpret_cast','typename',
    'class','for','return','union',
    'const','friend','short','unsigned',
    'const_cast','goto','signed','using',
    'continue','if','sizeof','virtual',
    'default','inline','static','void',
    'delete','int','static_cast','volatile',
    'do','long','struct','wchar_t',
    'double','mutable','switch','while',
    'dynamic_cast','namespace','template', 'true', 'false',
   'null'}
in cppKeywords→includes(e.name))
```

22. **NA05**: Attribute naming camelCase convention
    - Context: Attribute
    - OCL definition:

```
e.class→notEmpty() and
    e.name.substring(1,1).toLower() <> e.name.substring(1,1)
```