# Towards an Integrated Quality Assessment and Improvement Approach for UML Models

Akhtar Ali Jalbani[1], Jens Grabowski[1],
Helmut Neukirchen[2], and Benjamin Zeiss[1]

[1] Institute for Computer Science, University of Göttingen
Goldschmidt Str. 7, 37077 Göttingen, Germany
{ajalbani,grabowski,zeiss}@informatik.uni-goettingen.de

[2] Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
University of Iceland, Dunhagi 5, 107 Reykjavík, Iceland
helmut@hi.is

**Abstract.** Models defined using the *Unified Modeling Language* (UML) are nowadays common parts of software documentations, specifications and sometimes even implementations. However, there is a broad variety of how UML is used. Reasons can be found, for example, in the lack of generally accepted modeling norms and guidelines, the semi-formal semantics of UML, or the complexity of the language. In practice, these factors inevitably lead to quality problems in UML models that need to be addressed. We investigate and discuss existing work in the field of quality assessment and improvement of UML models and present how we envision an integrated approach to quality assessment and improvement of UML models. We assess a model with a *Factor-Criteria-Metrics* (FCM) based quality model, detect issues by finding smells and violated metric thresholds in UML models, and improve UML models by applying refactorings using model-to-model transformations.

## 1 Introduction

Quality control for a software development process requires ongoing quality assurance measures for all artifacts produced during the development process. Assessing the quality of artifacts produced in early phases of a process, such as requirement or design specifications, is critical since a change in these specifications often implies change in documents, specifications, or code (during later development phases) as well. The *Unified Modeling Language* (UML) from the *Object Management Group* (OMG) has been widely adopted as a common modeling language for the creation of requirements and design specifications. In later stages of development, UML models may also serve as a base for code and test generation. Unfortunately, there is a wide variety of how UML is used in practice. This is often due to the lack of generally accepted modeling norms and guidelines, the semi-formal semantics of UML and the complexity of UML as a whole.

In practice, these factors inevitably lead to quality problems in UML models that need to be addressed. Therefore, continuous tool-supported quality assurance and quality improvement measures, throughout the whole development process, are required. Based on our experience [1, 2] with the quality engineering of large test specifications written in the standardized *Testing and Test Control Notation* (TTCN-3), we started to investigate the possibility of using similar techniques for the quality engineering of UML models. Our quality engineering approach for TTCN-3 specifications is based on:

– a quality model for test specifications that defines the main quality characteristics of a test specification,
– test metrics to assess the quality characteristics,
– smell detection for identifying problematic locations in the test code using pattern-based analysis and metric thresholds, and
– refactoring for the improvement of those problematic locations.

While TTCN-3 from its appearance is comparable to a typical general purpose programming language like C or Java, the challenges to adopt this approach for UML are numerous. As already mentioned, the usage and actual knowledge about UML is very diverse. It is not unusual that people mistake UML as a standardized way to draw diagrams, rather than understanding it as a modelling language, that uses diagrams for partial presentations of the model. This variety of how UML is perceived differently by the people using it, eventually has an effect on what UML specifications look like, how they can be used later on, and also how quality engineering must be implemented for UML specifications. The contribution of this paper is a survey on the topics quality models, metrics, bad smells, and refactoring for UML models. The papers investigated in the survey are selected to cover those topics that are related to our proposed approach for UML quality assurance.

This paper is structured as follows: in Section 2, we introduce the foundations of this paper, i.e., the basics of UML, quality models, metrics, bad smells, and refactoring. Sections 3–6 present existing work on the respective topics. Section 7 outlines current tool support for metrics, smell detection, and refactoring for UML. In Section 8, we present our ideas for an integrated quality engineering approach for UML. We conclude with Section 9, where we summarize our progress towards the realization of our ideas and we discuss what topics still need to be addressed.

## 2 Foundations

In this section, we briefly provide the foundations of the topics that are relevant for this paper — the *Unified Modeling Language* (UML), quality models, metrics, bad smells, and refactoring. Section 2.1 is of particular importance as it also attempts to address common misunderstandings and misconceptions about UML, e.g., what the UML architecture is about and what kinds of notations exist for UML models.

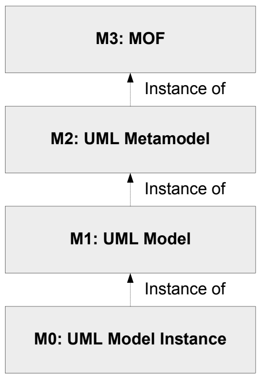


**Fig. 1.** The UML Architecture

## 2.1 The *Unified Modeling Language* (UML)

We assume basic knowledge of UML and therefore we concentrate on facts that may not be immediately apparent to everyone who has not dealt with UML as a modelling language. The UML architecture is composed of four layers (Figure 1). The M3 layer, the foundation of UML, is called the *Meta-Object Facility* (MOF) [3]. OMG itself describes MOF as a metadata management framework and metadata services. In essence, MOF is a language that is used to model itself as well as other models or metamodels. In the context of UML, the most prominent use of MOF is the definition of the UML metamodel (the M2 layer). MOF can be considered a meta-metamodel in this case. A distinction is made between *Essential MOF* (EMOF) and *Complete MOF* (CMOF). As the names suggest, EMOF is a slimmed down version, a subset of CMOF. MOF is used to specify the UML metamodel that consists of the *Infrastructure* [4] and *Superstructure* [5] standards. These standards define the abstract syntax of the language, i.e., basic UML modeling concepts, attributes, relationships, as well as the semantics of each modeling concept. In the language definition, the cohesion between the Infrastructure definition and MOF is more complex, as MOF is again defined using UML. The M1 layer is again an instance of the M2 layer. On the M1 layer, we find those models that we typically create for requirements or design specifications. The instance of a UML model is then finally found on the M0 layer, which describes instantiated objects.

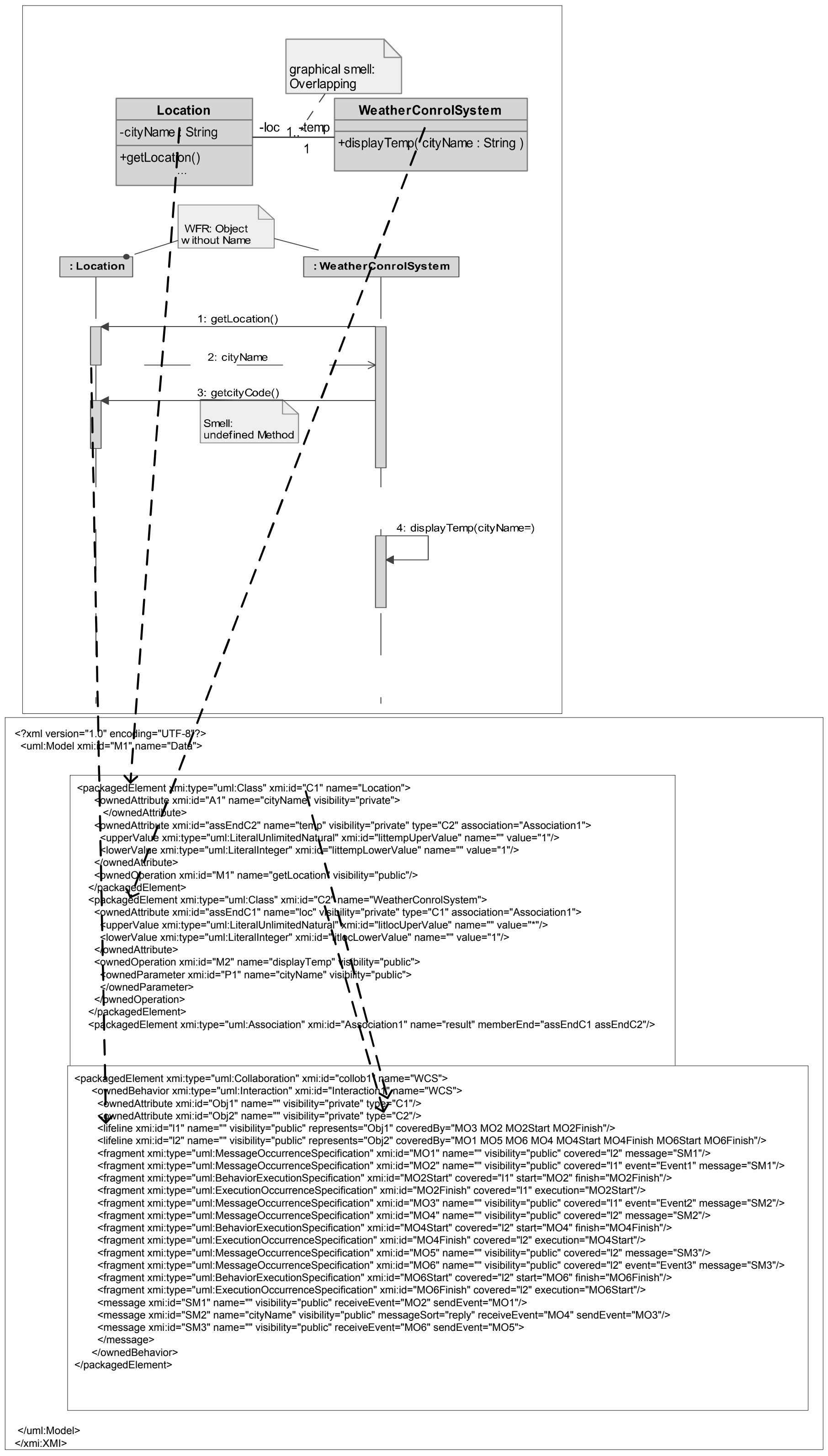
The UML models we deal with everyday are typically the ones found on the M1 layer — we create instances of the UML metamodel. One common way to create such a model is to use the graphical notation provided in the UML Superstructure standard. However, it is crucial to understand that a UML model and a UML diagram are different things. It is easily possible to draw a set of diagrams in the UML notation on paper. However, on paper these cannot be validated, transformed, or even used for code generation. Even if we transfer our diagrams as they are into digital form, they are missing important pieces of information that are not part of the diagrams such as how the diagrams relate to each other and where the definitions to model references can be found. If the graphical notation is used to create a UML model (i.e., by using a UML tool), each diagram represents only a partial view of the complete model. Thus, a UML model may be described by multiple diagrams or no diagram at all — a UML model may still contain all elements we know from the commonly used graphical notation without including a single diagram. However, there is no common and unified notation that can represent a UML model completely, but attempts to solve this problem exist, e.g, TextUML [6]. One way, although not entirely human-readable, to represent a complete UML model is the *XML Metadata Interchange* (XMI) format [7] which is, however, an exchange format rather than a useful notation for modeling.

To illustrate the difference between a model and diagrams, we present a simple specification of a weather information system in Figure 2. At the top of the figure, we have the graphical notation of a UML model consisting of a class and a sequence diagram. At the bottom part of the figure, we present the XMI notation of the same model. The figure illustrates two things. First, a complete model can represent multiple diagrams and vice versa — multiple diagrams may be part of a single UML model. In this case, the model contains the definitions from the class diagram and the sequence diagram. Second, the XMI representation explicitly references the previously defined UML classes. Such an explicit reference is not possible when we deal with diagrams in UML notation (that are created using pencil and paper or a diagramming tool) rather than UML models. Finally, it is necessary to mention the *Object Constraint Language* (OCL) [8]. OCL is a declarative language used to express constraints (preconditions, postconditions, or invariants) on UML models and is based on first-order predicate logic. Although not its intended use, it is also possible to use OCL as a query language by evaluating result sets of OCL expressions.

## 2.2 Software Quality and Quality Models

*Software quality* refers to all attributes of a software product that show the appropriateness of the product to fulfill its requirements. For a software product, Fenton et al. [9] distinguish between attributes of *processes*, *resources*, and *products*. For each class, *internal* and *external attributes* can be distinguished. External attributes refer to how a process, a resource, or a product relates to its environment. Internal attributes, on the other hand, are properties of a process, a resource, or a product on its own, i.e. separate from any interactions with its environment. Hence, the assessment of external attributes of a product, the so-called *external quality*, requires the execution of the product, whereas usually static analysis is used for the assessment of its internal attributes, the so-called *internal quality*. Since this article treats quality characteristics for UML models, which are products that do not need to be executable, only internal quality is considered in the following.

Quality models are used to assess software quality. Our work concentrates on hierarchical *Factor-Criteria-Metrics quality model*s (FCM-models). Prominent



```
<?xml version="1.0" encoding="UTF-8"?>
 <uml:Model xmi:id="M1" name="Data">

    <packagedElement xmi:type="uml:Class" xmi:id="C1" name="Location">
       <ownedAttribute xmi:id="A1" name="cityName" visibility="private">
         </ownedAttribute>
       <ownedAttribute xmi:id="assEndC2" name="temp" visibility="private" type="C2" association="Association1">
         <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="littempUperValue" name="" value="1"/>
         <lowerValue xmi:type="uml:LiteralInteger" xmi:id="littempLowerValue" name="" value="1"/>
       </ownedAttribute>
       <ownedOperation xmi:id="M1" name="getLocation" visibility="public"/>
     </packagedElement>
     <packagedElement xmi:type="uml:Class" xmi:id="C2" name="WeatherConrolSystem">
       <ownedAttribute xmi:id="assEndC1" name="loc" visibility="private" type="C1" association="Association1">
         <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="litlocUperValue" name="" value="*"/>
         <lowerValue xmi:type="uml:LiteralInteger" xmi:id="litlocLowerValue" name="" value="1"/>
       </ownedAttribute>
       <ownedOperation xmi:id="M2" name="displayTemp" visibility="public">
         <ownedParameter xmi:id="P1" name="cityName" visibility="public">
         </ownedParameter>
       </ownedOperation>
     </packagedElement>
     <packagedElement xmi:type="uml:Association" xmi:id="Association1" name="result" memberEnd="assEndC1 assEndC2"/>

   <packagedElement xmi:type="uml:Collaboration" xmi:id="collob1" name="WCS">
      <ownedBehavior xmi:type="uml:Interaction" xmi:id="Interaction1" name="WCS">
       <ownedAttribute xmi:id="Obj1" name="" visibility="private" type="C1"/>
       <ownedAttribute xmi:id="Obj2" name="" visibility="private" type="C2"/>
       <lifeline xmi:id="l1" name="" visibility="public" represents="Obj1" coveredBy="MO3 MO2 MO2Start MO2Finish"/>
       <lifeline xmi:id="l2" name="" visibility="public" represents="Obj2" coveredBy="MO1 MO5 MO6 MO4 MO4Start MO4Finish MO6Start MO6Finish"/>
       <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MO1" name="" visibility="public" covered="l2" message="SM1"/>
       <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MO2" name="" visibility="public" covered="l1" event="Event1" message="SM1"/>
       <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="MO2Start" covered="l1" start="MO2" finish="MO2Finish"/>
       <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="MO2Finish" covered="l1" execution="MO2Start"/>
       <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MO3" name="" visibility="public" covered="l1" event="Event2" message="SM2"/>
       <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MO4" name="" visibility="public" covered="l2" message="SM2"/>
       <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="MO4Start" covered="l2" start="MO4" finish="MO4Finish"/>
       <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="MO4Finish" covered="l2" execution="MO4Start"/>
       <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MO5" name="" visibility="public" covered="l2" message="SM3"/>
       <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MO6" name="" visibility="public" covered="l2" event="Event3" message="SM3"/>
       <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="MO6Start" covered="l2" start="MO6" finish="MO6Finish"/>
       <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="MO6Finish" covered="l2" execution="MO6Start"/>
       <message xmi:id="SM1" name="" visibility="public" receiveEvent="MO2" sendEvent="MO1"/>
       <message xmi:id="SM2" name="cityName" visibility="public" messageSort="reply" receiveEvent="MO4" sendEvent="MO3"/>
       <message xmi:id="SM3" name="" visibility="public" receiveEvent="MO6" sendEvent="MO5">
       </message>
      </ownedBehavior>
   </packagedElement>

 </uml:Model>
</xmi:XMI>
```

**Fig. 2.** Graphical and XMI Representation of a UML Model

examples for FCM-models are the quality model developed by McCall et al. (McCall-model) [10] and the ISO/IEC standard 9126 (ISO/IEC 9126-model) [11].

The highest level of the McCall-model are the three uses: *operation*, *transition* and *maintenance*. The operation use refers to quality characteristics that concern the product when it is being executed, i.e., its external quality. The transition use combines quality characteristics that concern the product when it is moved to another environment, and the maintenance use focuses on quality characteristics that concern the product when it is changed. As indicated by the abbreviation FCM, on the second, third and fourth level, the McCall model defines *factors*, *criteria* and *metrics*. A factor defines a high-level quality criterion such as efficiency. On the next lower level, criteria for judging factors are defined. For example, criteria for the factor efficiency are storage and execution efficiency. Metrics are then used to assess criteria, e.g., storage efficiency may be assessed by calculating the ratio between allocated and used storage.

The ISO/IEC 9126-model defines no uses, but distinguishes between internal quality, external quality and quality-in-use. The quality ISO/IEC 9126-model is a generic quality model that covers internal and external quality in one abstract model (Figure 3). The model for *quality-in-use* is similar to the operation use of the McCall model. However, quality-in-use and external quality are out of the scope of this paper and therefore not discussed any further. In the ISO/IEC 9126-model, factors are called *characteristics* and criteria are called *subcharacteristics*.

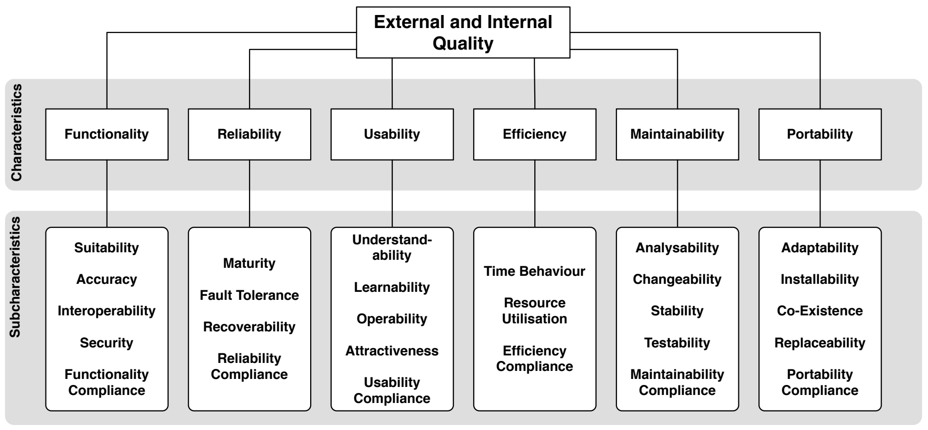


**Fig. 3.** ISO/IEC 9126 Quality Model for Internal and External Quality

## 2.3 Software Metrics

Fenton et al. structured internal product metrics, i.e., metrics that measure internal quality, into *size* and *structural* metrics [9]. Size metrics measure properties of the number of usage of programming or specification language constructs, e.g., the *number of source statements*. Structural metrics analyze the structure of a program or specification. Popular examples of structural metrics are complexity metrics based on control flow or coupling metrics.

To make sure that reasonable metrics for quality assessment are chosen, Basili et al. suggest the *Goal, Question and Metrics* (GQM) approach [12]: First, the goals which shall be achieved (e.g., improve maintainability) must be defined. Then, for each goal, a set of meaningful questions that characterize a goal is derived. The answers to these questions determine whether a goal has been met or not. Finally, one or more metrics are defined to gather quantitative data which can provide answers to each question.[3]

## 2.4 Smells

The metaphor of "*bad smells in code*" has been coined by Beck and Fowler [13]. They define smells as "*certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring*". According to this definition, defects with respect to program logic, syntax, or static semantics are not smells since these defects cannot be removed by a refactoring. A refactoring only improves internal structure, but does not change observable behaviour.

Beck and Fowler present smells for Java source code. They describe their smells using unstructured English text. A well-known smell is *Duplicated Code*. Code duplication affects in particular the *changeability* quality subcharacteristic in the ISO/IEC 9126-model: if code that is duplicated needs to be modified, it usually needs to be changed in all duplicated locations. Smells provide only hints: whether the occurrence of an instance of a certain smell in a source code is considered as a sign of low quality may depend on preferences and the context of a project. For the same reason, a list containing code structures that are considered smells is never complete and may also vary from project to project and from domain to domain [14].

The notions of metrics and smells are not disjoint: each smell can be turned into a metric by counting the occurrences of a smell, and often, a metric can be used to locate a smell. The latter is the case, for example, when a long function is expressed by a metric that counts the lines of code of this function and a threshold is violated. However, the above smell of duplicated code and other pathological structures in code require a pattern-based detection approach and cannot be identified by using metrics alone.

[3] The GQM approach can also be used to define individual FCM quality models as goals are similar to factors and questions similar to criteria.

### 2.5 Refactoring

*Refactoring* is defined as "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*" [13]. This means that refactoring is a remedy against software aging [15]. While refactoring can be regarded as cleaning up source code, it is more systematical and thus less error prone than arbitrary code clean-up, because each refactoring provides a checklist of small and simple transformation steps, which are often automated by tools.

The essence of most refactorings is independent from a specific programming language. However, a number of refactorings make use of particular constructs of a programming language, or of a programming paradigm in general, and are thus only applicable to source code written in that language.

## 3 Quality Models for UML

A surprisingly small number of researchers have addressed the problem of quality assessment for UML models. The comprehensive work in this area has been done by Lange and Chaudron [16, 17]. In [17], they discuss the difference between source code and UML models and highlight the particularities of UML models. As a consequence, a special quality model for UML has been developed (in the following called Lange-Chaudron-model). An overall view of the model is given in Figure 4.

Like the model developed by McCall, the Lange-Chaudron-model is a hierarchical model with four levels. On the highest level, the Lange-Chaudron-model defines the two uses *maintenance* and *development*. The maintenance use is taken from the McCall model. The other two uses from McCall, i.e., *operation* and *transition*, are not relevant for the quality of UML models. The operation use is related to external quality attributes and the transition use is not related to the development phases in which UML is used, i.e., modeling and design phases. The development use combines quality characteristics of a product and its artifacts in phases before the product is finished. The second level of the Lange-Chaudron-model defines the purposes of modeling. For example, the purpose *Testing* indicates that the model is used for test generation and the purpose *Code Generation* denotes a usage for automatic code generation. The third level of the Lange-Chaudron-model identifies the characteristics of the purposes. The meaning of most characteristics in Figure 4 is straightforward. For example, the characteristic *complexity* measures the effort required to understand a model or a system.

Two special characteristics of the Lange-Chaudron-model are *aesthetics* and *balance*. The quality of the graphical diagrams is addressed by the aesthetics characteristic only. Aesthetics is defined by the extent that the graphical layout of a model or system enables ease of understanding of the described system. Lange and Chaudron define balance as the extent that all parts of a system are described at an equal degree. All characteristics are included in the balance

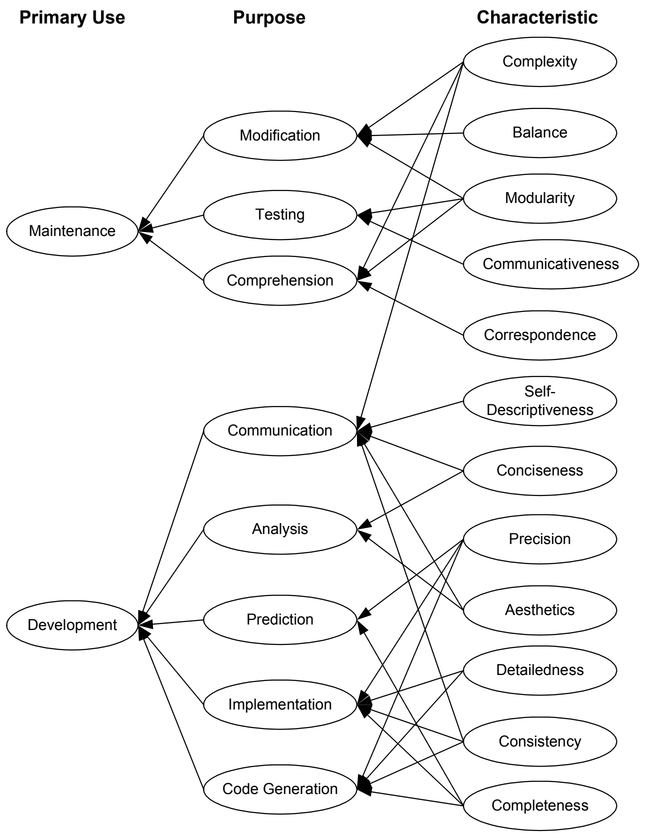


**Fig. 4.** Lange-Chaudron Quality Model

characteristic with the same weight. This has been criticized by Mohagheghi and Aagedal [18], because the assessment of the balance characteristic requires the evaluation of all metrics and rules defined in the fourth level, i.e., it is not a good abstraction. In [18], it is proposed to shift balance to the purpose level and to assess balance by using the characteristics completeness, conciseness, modularity, and self-descriptiveness.

The fourth level of the Lange-Chaudron-model (not shown in Figure 4) defines metrics and rules for the assessment of the characteristics. We discuss this part of Lange-Chaudron-model in the sections 4 and 5. Lange and Chaudron underpinned their work with industrial case studies. They showed the applicability of their approach by interviewing representatives of project teams, analyzing UML models, and giving feedback to project teams. A quality model for design documentation in model-centric domains has been developed by Pareto and Boquist [19]. The background of this work is experience with the *Rational Unified Process* (RUP) as model-centric software development process. Even though

UML is an essential part of RUP, all kinds of artifacts on the abstraction levels between requirements specification and code are considered relevant. For the development of the quality model, Pareto and Boquist interviewed and discussed with designers, process engineers, line managers and architects. From these interviews and discussions, 22 quality attributes were identified and structured into six groups. Each group identified one quality characteristic. As the quality model is related to RUP also quality aspects for management are covered. However, they stop with the identification of quality attributes and quality characteristics. No means for the assessment of quality attributes and characteristics are provided.

## 4 Metrics for UML

In current research, a large number new software metrics are defined. Many metrics are based and calculated on grammatical structures such as *Abstract Syntax Tree*s (ASTs). A UML model is also based on a specific structure: the UML metamodel. However, numerous proposals are based on informal metrics definitions that often respect only diagrams, i.e., the graphical representation of UML with its partial views. In the following sections, we will present noteworthy literature on UML metrics. We differentiate between metrics that are based on the actual UML model and metrics that are solely based on the graphical notation, i.e., graphical metrics.

### 4.1 Model Metrics

Lange [20] uses metrics and rules (metrics with a binary result) and relates them to quality characteristics of his quality model (see Section 3) to assess the quality of a UML model. He reuses the most widely known metrics such as the metric suite from Chidamber and Kemerer [21] and describes them informally. He stresses that his list is by no means complete. Kim and Boldyreff [22] propose 27 metrics for UML that are supposed to predict characteristics at earlier stages in the software lifecycle. The metrics are defined informally and no relationship between the UML model quality and the metrics is established. Baroni et al. [23] propose to use OCL to describe UML metrics in a formal way in order to avoid ambiguities due to descriptions in natural language. By using several samples of different complexity, they demonstrate that OCL is a well suited formalism for defining UML metrics and that it is easier to understand than formulas using custom built mathematical frameworks. McQuillan and Power [24] extended this approach and use OCL to calculate coupling and cohesion metrics, as well as the metrics from the Chidamber and Kemerer metric suite [21]. They argue, however, that a metrics specific metamodel is a more generic solution than defining metrics directly over the UML metamodel. Furthermore, they demonstrate how to automatically generate test data and metamodel instances. Another interesting way to formalize metrics is proposed by El-Wakil et al. [25]. They propose to define metrics using *XQuery* over the XMI representation of the UML model under analysis. They argue that using XQuery to express metrics eases tool building. Also, they claim that metric libraries specified in XQuery are easy to extend and provide a proof-of-concept implementation.

### 4.2 Graphical Metrics

Graphical metrics for UML are not covered very well in the literature despite the fact that the quantification of visual elements can be an important part to assess the quality of a graphical layout. However, it seems that layouting itself draws more attention in research than the assessment of a layout by numbers. Kiewkanya and Muenchaisri [26] performed an experiment in which they evaluated whether metrics quantifying aesthetic aspects of class and sequence diagrams influence the maintainability of UML models. For the measurements, they selected aesthetic indicators that have been proposed by Purchase [27], Eichelberger [28], and others. Such aesthetic indicators are, for example, the maximum number of bends on the edges, the standard deviation of edge lengths, or the total numbers of edges fixed to an orthogonal grid divided by the total number of edges. Their conclusion is that aesthetic metrics can indeed be indicators for the maintainability of class and sequence diagrams. Gronback [29] provides a general catalog of UML metrics to detect deviations from best practices. Some of them are derived from style guidelines provided by Ambler [30]. He suggests generic diagram metrics such as “number of colors on diagram” or diagram-specific metrics such as “depth of inheritance hierarchy” (for class diagrams) and even provides minimum and maximum thresholds for his metrics. The metrics presented by Gronback, however, mix graphical properties with properties that are part of the UML model.

## 5 Smells for UML

As discussed earlier, UML models do not have a standardized textual notation like typical general purpose programming languages. However, bad smell analysis in source code is rarely executed directly on the textual notation. An abstract grammatical representation of the notation, the AST, can in fact be regarded as a model for the textual notation of the programming language that is subject of the analysis. Analyzing UML models is therefore not that much different than analyzing an AST. However, the underlying abstract syntax is more complex. In the following section, we present related work that deals with bad smells in UML models. We differentiate between model smells and graphical smells. With model smells, we regard design flaws or possible defects that we find by analyzing the UML model (independently from any diagrams) such as possible inconsistencies, ambiguities, or constructs that complicate maintenance. Graphical smells, on the other hand, are related to the graphical notation of UML. They primarily concern the understandability aspect of the diagram. For example, diagrams with overlapping or crossing elements are harder to understand than diagrams with elements that are properly laid out with aesthetic aspects in mind.

### 5.1 Model Smells

Lange [16] — with his goal to improve the overall quality of UML models — discusses that undetected defects can cause large problems in later development stages and identifies generic UML defects such as the number of messages in a sequence diagram that do not correspond to a method in a defined class diagram. The presented smells were identified by discussions with industrial partners and by performing case studies. He assumes that a set of UML diagrams defines a system as a whole and that those diagrams have consistency relationships between each other. The defects partially overlap with the well-formedness rules and are related in their scope, but are described informally, without a relationship to the abstract syntax of UML. Astels [31] presents UML smell detection in the context of UML refactoring. With smell detection, he locates where to refactor and which refactoring is suggested. He argues that the visual presentation of UML makes smell structures more evident and presents exemplarily what classical bad smells from Fowler [32] (e.g., lazy class or middle man) look like in the graphical notation. His own statement is that his list is by no means complete. His work is described informally in the visual notation of UML.

### 5.2 Graphical Smells

Graphical smells concern the graphical notation of UML models excluding problems that are of logical nature or that may introduce issues in efficiency or maintenance. Therefore, the main aspect of graphical smells is how model elements are laid out and what elements are represented by the diagrams. Ambler [30] provides more than 300 guidelines for all UML diagram types that primarily concern the graphical notation. The violations of these guidelines can be considered as graphical smells. As an example, a guideline with the aim to improve the understandability of a diagram is to split large diagrams with a high number of elements into multiple smaller diagrams, where no diagram must have more than nine elements. Purchase et. al [27] have studied graphical layout aesthetics in class and collaboration diagrams. By performing a case study where they questioned persons to investigate their subjective preferences, they conclude that there are certain common aesthetic properties that seem to be unfavorable. Among these properties are, for example, arc crossings, or orthogonality (for class diagrams). From their results, they derive that the aesthetics of graph layouts is dependent on the domain, i.e., properties that are important for one diagram type may not be important for another one.

## 6 Refactorings for UML

UML refactoring is an emerging research topic that can already be considered as important as classical source-code refactoring. We again differentiate between model refactorings, i.e., semantically preserving model changes and graphical refactorings that improve the aesthetics of UML diagrams.

## 6.1 Model Refactorings

Astels [31] presents UML smells in class and sequence diagrams and describes a number of Fowler refactorings that are applicable to UML. His refactoring descriptions are based on UML diagrams and are informal. His examples are intended to motivate that UML refactoring is applicable in the context of agile development processes. France and Bieman [33] want to avoid uncontrolled change and increased evolution costs of a system due to deteriorating structure and system quality by introducing a goal-directed, cyclic process for object-oriented software when object-oriented models, such as UML models, are transformed and evaluated in each cycle. For the model transformation, they explicitly mention model refactoring to enhance quality attributes of the model that should be realized using patterns involving roles, i.e., each participant in the pattern plays a certain role with specific properties within the pattern description. A formal method for pattern-based transformation with role models does not exist yet. Sunyé et al. [34] propose refactorings for class diagrams and state charts to make software easier to extend and maintain. Using pre and post conditions expressed in OCL, they ensure that transformation preserve behavioral properties. However, they describe the refactoring mechanics informally. Porres [35] presents how to describe and execute UML refactorings using a rule-based transformation formalism and he argues that an update-based mapping mechanism that modifies a model in place is more efficient for describing refactorings than *mapping transformations* that transform into a different target model. For the realization and description of refactoring transformations, he uses his own language called *SMW* that operates on the UML metamodel — when the paper was written, there were no widely adopted transformation languages available. Dobrzański [36] provides a comprehensive survey on UML model refactorings in his master's thesis that deals with the refactoring of executable UML models [37]. He introduces an initial refactoring catalog for executable UML models. The refactorings are formalized with pre and post conditions in OCL. According to him, the main difference in refactoring executable models is that the update of the behavioral aspects of the models has to be taken into account.

More recent work on UML model refactoring and transformation is often based on the *Eclipse Modeling Framework* (EMF) representation of UML models. Biermann et al. [38] present work on an EMF model transformation framework that is based on graph transformations. They show how the rich theory of algebraic graph transformation can be applied to EMF model transformations. Using their method, the validation of the model transformations with respect to functional behavior and correctness is possible. They demonstrate their approach by using selected state chart refactorings. Similarly, Folli and Mens [39] suggest the use of graph transformations for model refactoring as well and present, as a proof-of-concept, how they have implemented a number of more complex UML model refactorings using the *AGG* [40] graph transformation tool.

### 6.2 Graphical refactorings

Graphical refactorings are applied when the graphical notation of a UML model, i.e., corresponding diagrams containing partial views of the UML model are hard to read and understand. There are a huge variety of generic graph layout algorithms, and graph drawing itself is a very active research topic. Summaries can be found in a variety of textbooks, for example, Graph Drawing by Battista et al. [41]. Work on layouts of UML diagrams is rare. Ambler [30] provides informal guidelines that lack a systematic transformation mechanic to improve diagrams. However, it is arguable whether graphical refactorings should only change parts of a model using the refactoring mechanic or whether UML diagram specific transformations for complete optimal layouts are more desirable. Eichelberger and Gudenberg [28] discuss existing automatic layout methods for class diagrams and present their approach to laying out class diagrams that respect aesthetic rules, such as those described by [27]. Castello et al. [42] propose an automatic layout algorithm that improves the readability of state chart diagrams. It reduces the number of edge crossings and edge bends.

## 7 Tool Support

It is encouraged to use tools for measuring metrics, detecting smells, and applying refactorings to UML models. Manual application of refactorings, for example, is very error-prone and there is a risk that the changes are not semantically preserving due to human mistakes. Popular tools that support the automatic calculation of metrics and detection of bad smells in UML models are SDMetrics [43], Together [44], IBM Rational Systems Developer [45], and ArgoUML [46]. These tools partially use different terminologies for the term “bad smell”. SDMetrics, for example, calls them *design rules*, Together calls them *audits*, or ArgoUML names them *design critics*. The toolset from Chaudron et al. [47] calculates metrics on UML models, it detects rules in sequence diagrams, it checks model consistency, and visualizes metrics in a metric viewer. Except for the commercial tool Poseidon for UML, which provides a refactoring browser supporting the refactorings from Boger at al. [48], none of the major commercial UML tools support refactoring beyond renaming and moving model elements. Tools that support more sophisticated UML refactorings are academic prototypes. An overview over existing academic UML refactoring tools is given by Dobrzański [36]. Van Gorp et al. [49] have implemented refactorings as plug-in for the Fujaba UML tool. Recently, several academic UML refactoring tools are evolving that build on EMF, for example, GaliciaUML [50].

## 8 Our Approach

The variety in the understanding and application of UML is visible in research as well. There is work dealing with UML diagrams only while neglecting the fact that UML is a modeling language. On the other hand, others respect UML as a modeling language or mix descriptions based on the graphical notation with descriptions based on the UML metamodel. Most authors realize that there are relationships and dependencies between different diagrams that have to be respected when applying UML refactorings. To describe those relationships based on diagrams, however, is the wrong approach and we strongly believe that metrics, smell detection, and refactoring for UML should be described in a formalism that works on the UML metamodel. This ensures precision regarding the actual underlying UML model and regards complete models rather than just partial views. Authors such as [35] have realized this as well, but neither do they present a complete quality engineering approach for UML models including assessment and improvement that can be applied in every iteration of a development process, nor were there any widely spread model transformation languages available that could be applied for such uses. With our experience in the quality assurance of TTCN-3 [2, 51], we aim to provide an integrated quality engineering approach for UML (Figure 5) that consists of two main parts: quality assessment and quality improvement. For the assessment, a quality model is used and metrics quantify quality characteristics of this model. For the improvement, smell detection is used for locating possible issues and refactoring is used to improve the quality. Once the improvement step is completed, a quality reassessment quantifies whether the improvement was successful.

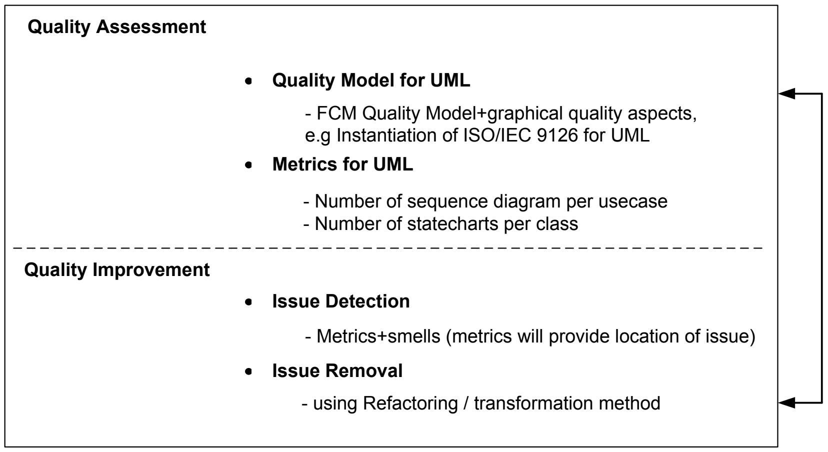

Quality Assessment
Quality Model for UML
- FCM Quality Model+graphical quality aspects, e.g Instantiation of ISO/IEC 9126 for UML
Metrics for UML
- Number of sequence diagram per usecase
- Number of statecharts per class
Quality Improvement
Issue Detection
- Metrics+smells (metrics will provide location of issue)
Issue Removal
- using Refactoring / transformation method

**Fig. 5.** Quality Assessment and Quality Improvement for UML

The first part in Figure 5, i.e., the quality assessment requires a quality model for UML. We aim to use a generic FCM-based quality model, such as the ISO/IEC 9126 model, that can be instantiated with metrics to quantify its quality characteristics. The quality model has to be described in detail for its target environment, i.e., UML models and, if necessary, it must be adapted for this domain. For example, the understandability characteristic of the ISO/IEC

9126 quality model does not only relate to the actual complexity of the UML model, but also to the graphical aspects of UML, such as the graphical smells described in Section 5.2. For the calculation of metrics, we use a widely used formalism such as OCL that works on the UML metamodel. While OCL is a language for expressing constraints in the first place, it can also be used to query models and to evaluate set sizes. For the quality improvement, we further plan to use languages that are now adopted for querying and transforming UML models. For the smell detection, we plan to provide a guideline catalog of bad smells in UML models using both an informal description in natural language and formal descriptions, e.g., given in OCL, where metrics with additional thresholds are defined to locate the smells. We are also evaluating the use of model transformation languages to identify smell locations by transforming the UML model into an instantiation of a metamodel that describes the results of the smell detection. For the actual refactorings, we plan to use existing model-to-model transformation languages such as *Query/View/Transformation* (QVT) [52], Xtend [53], or the *ATLAS Transformation Language* (ATL) [54] to describe the refactoring transformations, and OCL to define pre and post conditions for each refactoring. The techniques and languages described above are all based on the analysis at the model level instead of the graphical notation of UML. We plan to emphasize on the actual model analysis instead of on the graphical problems, as the involved layout techniques are part of a different field of research. However, graphical issues do play an important role in the overall assessment of the quality of a UML model — especially for the understandability quality characteristic. As we plan to use existing languages as formalisms to describe our metrics, smells, and refactorings, a proof-of-concept implementation will only involve the application of our metrics, smells, and refactoring descriptions to actual tool implementations that exist.

## 9 Status and Future Work

In our current work, we have successfully applied OCL for the calculation of metrics on a UML model and we have made first experiments to describe refactorings using Xtend. In both cases, we directly executed our experiments against existing UML models with tools that implement these languages. For the evaluation of OCL, we used the Eclipse OCL implementation of the *Eclipse Model Development Tools* project [55], which also allows the evaluation of the OCL result sets. For the refactoring, we have implemented so far simple refactorings using Xtend [53]. Our first experiments to detect smells and to apply refactoring using model-to-model transformation languages were successful, however, describing model-to-model transformations for UML models is not always an easy task due to the complexity of the UML metamodel. We plan to evaluate QVT and ATL to find out whether they make these descriptions more compact or more complicated and we also intend to evaluate the possibility of building a refactoring toolkit that eases the definition of refactorings. For the validation of our approach, we currently start to work on a case study that involves a UML model similar in size to industrial models. We then plan to apply our combined approach in an automated manner and use the quality reassessments after improvements to check whether the quality has improved. Expert reviews should then validate whether the automatic reassessment is correct.

## References

1. Neukirchen, H., Zeiss, B., Grabowski, J.: An Approach to Quality Engineering of TTCN-3 Test Specifications. International Journal on Software Tools for Technology Transfer (STTT) **105**(4) (2008) 309–326
2. Neukirchen, H., Zeiss, B., Grabowski, J., Baker, P., Evans, D.: Quality Assurance for TTCN-3 Test Specifications. Software Testing, Verification and Reliability (STVR) **18**(2) (2008)
3. Object Management Group (OMG): Meta Object Facility (MOF) Core Specification, Version 2.0, formal/2006-01-01 (2009)
4. Object Management Group (OMG): UML Infrastructure Specification, Version 2.2, formal/2009-02-04 (2009)
5. Object Management Group (OMG): UML Superstructure Specification, Version 2.2, formal/2009-02-02 (2009)
6. Abstratt Technologies: TextUML Toolkit. `http://www.abstratt.com`, Last Visited March 2009
7. Object Management Group (OMG): MOF 2.0/XMI Mapping, Version 2.1.1, formal/2007-12-01 (2007)
8. Object Management Group (OMG): OCL Core Specification version 2.0, formal/2006-05-01 (2009)
9. Fenton, N., Pfleeger, S.: Software Metrics: A Rigorous and Practical Approach. PWS Publishing, Boston (1997)
10. McCall, J., Richards, P., Walters, G.: Factors in Software Quality. Technical Report RADC TR-77-369, US Rome Air Development Center (1977)
11. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC): ISO/IEC Standard No. 9126. Software Engineering-Product Quality; Part 1-4 (2001-2004)
12. Basili, V.R., Weiss, D.M.: A Methodology for Collecting Valid Software Engineering Data. IEEE Transactions on Software Engineering **10**(6) (1984) 728–738
13. Fowler, M.: Refactoring – Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
14. van Emden, E., Moonen, L.: Java Quality Assurance by Detecting Code Smells. In: Proceedings 9th Working Conference on Reverse Engineering (WCRE 2002), IEEE (2002) 97–106
15. Parnas, D.: Software Aging. In: Proceedings of the 16th International Conference on Software Engineering (ICSE), Sorrento, Italy, IEEE/ACM (1994) 279–287
16. Lange, C.: Assessing and Improving the Quality of Modeling. PhD thesis, Technische Universiteit Eindhoven, Netherland (2007)
17. Lange, C., Chaudron, R.: Managing Model Quality in UML-Based Software Development. In: Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering in Practice (STEP 2005), IEEE (2005)
18. Mohagheghi, P., Aagedal, J.: Evaluating Quality in Model-Driven Engineering. In: Proceedings of the International Workshop on Modeling in Software Engineering (MISE 2007), IEEE (2007)

19. Pareto, L., Boquist, U.: A Quality Model for Design Documentation in Model-Centric Projects. In: Proceedings of the 3rd International Workshop on Software Quality Assurance (SOQUA 2006), ACM (2006)
20. Lange, C.: Improving the Quality of UML Models in Practice. In: Proceedings of 28th International Conference on Software Engineering (ICSE 2006), ACM (2006)
21. Chidamber, S.R., Kemerer, C.: A Metric Suite for Object-Oriented Design. IEEE Transactions on Software Engineering **20**(6) (1994) 476–493
22. Kim, H., Boldyreff, C.: Developing Software Metrics Applicable to UML Models. In: Proceedings of the 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Engineering, Malaga, Spain. (2002)
23. Baroni, A., Braz, S., e Abreu, F.B.: Using OCL to Formalize Object-Oriented Design Metrics Definitions. In: Proceedings of ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Spain, Springer (2002)
24. McQuillan, J., Power, J.: A Metamodel for the Measurement of Object-Oriented Systems: An Analysis using Alloy. In: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST 2008), IEEE (2008)
25. El-Wakil, M., El-Bastawisi, A., Riad, M.B., Fahmy, A.A.: A Novel Approach to Formalize Object-Oriented Design Metrics. In: Proceedings of the 9th International Conference on Empirical Assessment in Software Engineering. (2005)
26. Kiewkanya, M., Muenchaisri, P.: Measuring Maintainability in Early Phase using Aesthetic Metrics. In: Proceedings of the 4th WSEAS International Conference on Software Engineering, Parallel & Distributed Systems. (2005)
27. Purchase, H., Allder, J., Carrington, D.: Graph Layout Aesthetics in UML Diagrams: User Preferences. Journal of Graph Algorithms and Applications **6**(3) (2002) 255–279
28. Eichelberger, H., von Gudenberg, J.W.: UML Class Diagrams - State of the Art in Layout Techniques. In: Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis, Amsterdam. (2003)
29. Gronback, R.: Model Validation: Applying Audits and Metrics to UML Models (2004) `http://conferences.codegear.com/jp/article/32089`, Last Visited March 2009.
30. Ambler, S.: The Elements of UML 2.0 Style. Cambridge University Press (2005)
31. Astels, D.: Refactoring with UML. In: Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP2002). (2002)
32. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
33. France, R., Bieman, J.: Multi-View Software Evolution — A UML-based Framework for Evolving Object-Oriented Software. In: Proceedings of 17th IEEE International Conference on Software Maintenance (ICSM 2001), IEEE (2001)
34. Sunyé, G., Pollet, D., Traon, Y., Jézéquel, J.: Refactoring UML Models. In: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. Volume 2185 of Lecture Notes in Computer Science., Springer (2001)
35. Porres, I.: Model Refactorings as Rule-Based Update Transformations. In: UML 2003 - The Unified Modeling Language. Volume 2863 of Lecture Notes in Computer Science., Springer (2003)
36. Dobrzański, Ł.: UML Model Refactoring- Support for Maintenance of Executable UML Models. Master's thesis, Blekinge Institute of Technology, School of Engineering, Ronneby, Sweden (2005)

37. Mellor, S., Balcer, M.: Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley (2002)
38. Biermann, E., Ermel, C., Taentzer, G.: Precise Semantics of EMF Model Transformations by Graph Transformation. In: Model Driven Engineering Languages and Systems. Volume 5301 of Lecture Notes in Computer Science., Springer (2008)
39. Folli, A., Mens, T.: Refactoring of UML models using AGG. In: Proceedings of the 3rd International ERCIM Symposium on Software Evolution. (2007)
40. Taentzer, G.: A Graph Transformation Environment for Modeling and Validation of Software. In: Applications of Graph Transformations with Industrial Relevance. Volume 3062 of Lecture Notes in Computer Science., Springer (2004) 446–453
41. Battista, G.D., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing - Algorithms for the Visualization of Graphs. Prentice-Hall (1998)
42. Castello, R., Mili, R., Tollis, I.: Automatic Layout of Statecharts. Software — Practice & Experience **32** (2002) 25–55
43. SDMetrics: The Software Design Metrics tool for the UML. `http://www.sdmetrics.com`, Last Visited March 2009
44. Borland: Borland Together. `http://www.borland.com/us/products/together`, Last Visited March 2009
45. IBM: IBM Rational Systems Developer. `http://www.ibm.com/software/awdtools/developer/systemsdeveloper`, Last Visited March 2009
46. ArgoUML Project: ArgoUML. `http://argouml.tigris.org`, Last Visited March 2009
47. Lange, C., Chaudron, R.: Empanada: Empirical analysis of architecture and design quality. `http://www.win.tue.nl/empanada/tools.htm`, Last Visited March 2009
48. Boger, M., Sturm, T., Fragemann, P.: Refactoring Browser for UML. In: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World. Volume 2591 of Lecture Notes in Computer Science., Springer (2003)
49. Gorp, P., Stenten, H., Mens, T., Demeyer, S.: Towards Automating Source-Consistent UML Refactorings. In: UML 2003 – Modeling Languages and Applications. Volume 2863 of Lecture Notes in Computer Science., Springer (2003)
50. Seuring, P.: Design and Implementation of a UML Model Refactoring Tool. Master's thesis, Hasso-Plattner-Institute for Software Systems Engineering at the Univesity of Potsdam (2005)
51. Zeiss, B., Vega, D., Schieferdecker, I., Neukirchen, H., Grabowski, J.: Applying the ISO 9126 Quality Model to Test Specifications – Exemplified for TTCN-3 Test Specifications. In: Proceedings of Software Engineering 2007 (SE 2007). Volume 105 of Lecture Notes in Informatics (LNI)., Köllen Verlag (2007)
52. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, formal/08-04-03 (2009)
53. openArchitectureWare.org: openArchitectureWare (oAW). `http://www.openarchitectureware.org`, Last Visited March 2009
54. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: A QVT-Like Transformation Language. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, ACM (2006)
55. Eclipse Foundation: Eclipse Model Development Tools (MDT) OCL. `http://www.eclipse.org/modeling/mdt/?project=ocl`, Last visited March 2009