Embedding Information about Defects, Decisions, Context, Quality, and Traceability in CIM- and PIM-level Software Models

Jörg Rech[1] and Mario Schmitt[2]

a: Fraunhofer Institute for Experimental Software Engineering, Fraunhofer Platz 1, Kaiserslautern, Germany

ABSTRACT
Additional information about software models comes in different forms such as detected defects, used design patterns, traceability information to other abstraction levels, etc. This information, documented by users or annotated by automated mechanisms, can support the maintenance, visualization, or quality assurance and has to be persisted to be available over a long period of time. In this paper, we present how additional information about defects, context, traceability, etc. can be embedded into a UML- and BPMN-based software model. Furthermore, we present a tool that uses the information about quality defects within a PIM to visualize defects directly in the software model diagrams.

Keywords: Quality Defects, Traceability, Context, Quality, Decisions, PIM, CIM, Embedded Information, Defect Annotation, Traceability Annotation, Software Models, MDSD.

Contents

1 Introduction

Business users together with business analysts and architects generate the basic characteristics of a software system that results in computational independent models (CIM) including, e.g., role, product, or process models. These models are further refined and, finally, transformed into platform-independent modes (PIM), which include architectural, computational, or deployment-oriented aspects of the modeled software system. During the modeling process

[1] Corresponding author: Tel: +49 (0) 631/6800-2210, Fax: +49 (0) 631/6800-1599, Joerg.Rech@gmail.com
[2] Tel: +49 (0) 631/6800-2215, Fax: +49 (0) 631/6800-1599, Mario.Schmitt@iese.fraunhofer.de

for the CIM, the business user describes the envisioned support the software system should provide him, while the business analyst formalizes this vision via business process modeling techniques and notations. The PIM is mainly modeled by the business analyst and software architect within the software organization – either manually or with support of automated techniques (e.g., a CIM to PIM model transformer). Here, they refine and transform the CIM into a PIM and document design decisions or used design patterns, establish traces from CIM to PIM, analyze the model for design flaws, state quality requirements, etc.

1.1 Usage Scenarios

In the context of developing CIMs, PIMs, and transforming CIMs to PIMs, we identified the following usage scenarios that can occur, which requires to store additional information on the models and that we want to support:

- *Traceability of elements*: During the development of a model as well as during transformation activities, many elements are derived, refined, extended, etc. from elements of the higher models. Business analysts and software architects need to document and check these associations using traceability links in case of changes to an element (or (sub-)system) at needs to be synchronized with the associated elements (potentially in another model) to keep the models and diagrams consistent with each other. In order to identify the occurrence and impact of (potential) additional changes on these associated models, traceability links are needed from the CIM-level to the PIM-level (downwards) and from the PIM-level to the CIM-level (upwards).
- *Documentation of decisions*: During the manual or automated transformation from CIM to PIM (downwards) as well as the refinement of CIMs or PIMs (sidewards) many decisions have to be made, which might be helpful in later evolution, maintenance, or re-engineering processes. Business users, business analysts, and software architects need to document and check these decisions if they want to change the system in the future. In order to keep the architecture understandable, changeable, and stable, these decisions (e.g., that we "use polling to transfer data between listeners and observers") need to be documented and linked to the model element of the CIM and/or PIM.
- *Documentation of (quality) defects* [29]: During the development of software models, as with other artifacts, often errors are made that lead to failures of the final software system or higher effort in later evolution, maintenance, or re-engineering of the developed models and system. Software architects, tester and quality engineers have to document and check for these quality defects, functional defects, or the compliance to standards as well as previously stated quality requirements (internal as well as external (e.g., from the U.S. Food and Drug Administration (FDA))). In order to improve the model quality and to be aware of potential problems to the model's quality, information on real or potential defects need to be documented. Further the persistent storage of information from manual or automated defect detection [30] and testing activities must support defects that concern a single location as well as multiple locations.
- *Documentation of context factors*: During the transformation from CIM to PIM (downwards) as well as the refinement of PIMs (sidewards) many elements are modeled using best practices, such as design patterns [16], architectural styles [22], or domain-specific naming, which might be helpful in later evolution, maintenance, or re-engineering processes. Business analysts and software architects need to document and check this context information for future reference, to support cross-phase and inter-team communication, or to enable context-sensitive defect diagnosis activities. In order to keep the architecture understandable, changeable, and stable, these context factors (e.g., class X is a "façade class") need to be documented and linked to the model element of the CIM and/or PIM.
- *Documentation of quality requirements*: During the analysis of the problem, the to-be-supported (business) process, and the system design, several quality requirements and quality (sub-)models are formulated that the whole system, a specific subsystem, or even an individual element (e.g., a class responsible for the authentication), needs to comply to. Business analysts, software architects, and quality engineers have to document these quality requirements and quality models for the later use in project management or quality assurance activities as well as to prioritize quality defects. In order to conform to the model's quality and to be aware of specific requirements of (parts of) the software model, information on the expected needs of the (business) environment, such as performance, reliability, or security, as well as the needs of the software organization, such as the maintainability or portability of the software model, needs to be documented.

1.2 Storage Requirements

As standard metamodel do not provide mechanisms to formally store this additional information within the software model itself, we have to store them in a new proprietary way. This information is created and documented by users or automated mechanisms and has to be formal enough to be machine readable, in order to be presented in standard or special views of a modeling tool, made available to other systems for further analysis (e.g. impact analyses), or persisted over a longer period of time. The handling of additional information, even in the absence of supporting tools, requires the storage mechanisms that support as much of the following requirements as possible. We further describe the assignment criteria that were used in Table 1 to evaluate several possible mechanisms (cf. section 1.3) to store additional information:

1. *Conform*: Information is associated, such that the meta-model of the software model still conforms to the standard. In Table 1, we assigned a full dot "●" when the storage mechanism would not require changes to, at least, one metamodel for CIMs and one for PIMs (in our case UML and BPMN), a half dot "⭘" when only one metamodel is supported, an empty dot "○" when the mechanism would require lightweight changes, and a slash "–" when the mechanism would require heavyweight changes.
2. *Flexible*: The mechanism can be used with multiple meta-models, such as UML, BPMN, etc., are supported. In Table 1, we assigned a full dot "●" when the mechanism can be used with any meta-model (e.g., it is part of a meta-metamodel), a half dot "⭘" when the mechanism is slightly associated with a meta-model (and needs nearly no rework), an empty dot "○" when the mechanism is strongly associated with a meta-model (and needs much rework), and a slash "–" when it is inherently connected to the metamodel and cannot be used with other metamodels.
3. *Exchangeable*: Information can be easily exchanged between software architects, designers, testers, inspectors, etc. In Table 1, we assigned a full dot "●" when the information would be transferred/copied without consideration by the engineer, a half dot "⭘" when the information can but has to be transferred/copied by the engineer manually, an empty dot "○" when the information has to be transferred/copied and modified by the engineer (e.g., by changing paths), and a slash "–" when it cannot (easily) transferred/copied by the engineer.
4. *Versionable*: Information is correctly used with multiple versions of a software model and the mechanism supports conflict resolution. In Table 1, we assigned a full dot "●" when the information can be versioned together with the model without consideration by the engineer, a half dot "⭘"when the information can be versioned but conflicts have to be checked by the engineer manually (e.g., information files have to be combined by the engineer), an empty dot "○" when the information has to be versioned and modified by the engineer (e.g., changing paths), and a slash "–" when it cannot (easily) be versioned by the engineer.
5. *Synchronizable*: Changes to the software model, such as adding, removing, or modifying elements, are easily synchronized with the information (even when supporting tools are inactive). In Table 1, we assigned a full dot "●" when the information is removed if an element is removed and points to the correct element if this is modified, a half dot "⭘" when the information stays (and gets invalid) when a model element is removed and points to the correct element if this is modified, an empty dot "○" when the information stays (and is still valid) when a model element is removed or points to an incorrect element if this is modified, and a slash "–" when the information gets completely invalid.
6. *Fusionable*: The fusion of two software models or the copying of (parts of) one model into another (including the additional information), should be supported. In Table 1, we assigned a full dot "●" when the information is fused without consideration by the engineer, a half dot "⭘" when it can but has to be (partly) fused by the engineer manually (e.g., by copying or fusing UML profiles), an empty dot "○" when the information has to be fused and updated by the engineer manually (esp. in a proprietary format), and a slash "–" when it cannot (easily) be fused by the engineer.
7. *User friendly*: The information should not or only be minimally visible to the user, i.e., directory trees or element trees used to identify elements (e.g., to use them on a diagram) should not include too much additional information (i.e., minimal pollution of the model). In Table 1, we assigned a full dot "●" when the engineer will not see the information or at most one additional element (e.g., in the software model), a half dot "⭘" when the engineer will see multiple elements for the information at places where it does not (always) disturb the engineer (e.g., multiple annotations), an empty dot "○" when the engineer will see and is (probably) disturbed by the information (e.g., within comments), and a slash "–" when the information would change the normal view on the model as expected from the metamodel.

8. *Tool independent*: The information should mostly be correct and consistent after another engineer with a different tool, which does not support the additional information (e.g., trace information), has changed the software model (esp. when removing an element). This only includes single-element relations and not necessarily multi-location relations within the same model (e.g., if one element in a trace relationship is removed the information in the remaining element can be inconsistent). In Table 1, we assigned a full dot "●" when the information is removed if an element is removed and points to the correct element if this is modified, a half dot "⭘" when the information stays (and gets invalid) when a model element is removed and points to the correct element if this is modified, an empty dot "○" when the information stays (and is still valid) when a model element is removed or points to an incorrect element if this is modified, and a slash "–" when the information gets completely invalid.

1.3 Storage Mechanism and Comparison

In our context, the de-facto metamodel on PIM-level is the UML which is built using the OMG's Meta Object Facility (MOF) meta-metamodel [21]. MOF is a common modeling language kernel providing a unified basis for all OMG metamodels. On CIM-level modeling focuses on business process modeling using mostly the Business Process Modeling Notation BPMN [4] and the Business Process Definition Metamodel BPDM [3]. While BPMN provides just a graphical notation for process orchestration, BPDM, however, is a CIM-level metamodel for business process modeling, using BPMN as the graphical notation. Similar to UML, BPDM is based on the MOF meta-metamodel.

Several solutions to the abovementioned problem of embedding information in software models are possible. In order to store the information in an Eclipse-based IME for PIMs, such as Topcased [33], and CIMs such as the BPMN-Editor of the SOA Tools Platform (STP) [9], we can persist our information as:

- *Markers/Properties* to the software model that are stored by the modeling tool, but cannot easily be shared between users or across a versioning system (e.g., CVS). Furthermore, the fusion with other models is extremely hard to impossible.
- *MOF Annotations*, a construct in MOF (a.k.a. MOF::Tag) that enable the "tagging" of MOF elements with attribute-value pairs and can be shared across a versioning system.
- *MOF Comments*, an element in MOF and many other meta-metamodels (e.g., "Text Annotations" in BPMN) that can store at least one comment (i.e., text field) per element, but, typically, is used for (humanly) readable comments by and for the modelers.
- *UML Stereotypes/Profiles*, an extension mechanism in UML that can be used to integrate additional elements into the UML. However, the information might be confused with, e.g., domain-specific stereotypes and could floods the user with too much information, not necessary in the day-to-day work.
- *External files,* similar to diagram interchange [7] files in Topcased, which use a semantic bridge to refer to elements of the software model(s). However, these files need to be used by the modeling tools at work in order to synchronize changes to model elements.
- *External database,* similar to external files with the change that the information is stored within a database, which stores the information with an unique (primary) key (e.g., the IDs of the model elements).
- *New Element,* extensions to the meta-model (or meta-metamodel) that introduces one new model element (e.g., "AdditionalInformation") or multiple new model elements (e.g., "TraceInformation", "DefectInformation", etc.)

Table 1. Information Storage vs. Requirements

	Conform	Flexible	Exchange-able	Versiona-ble	Synchro-nizable	Fusionable	User friendly	Tool inde-pendent
Markers/Properties	●	●	–	–	●	–	●	○
MOF Annotations	●	●	●	●	●	●	●	⭘
MOF Comments	●	●	●	●	●	●	○	⭘
UML Stereotypes/Profiles	●	⭘	⭘	●	●	⭘	○	⭘
External file(s)	●	○	⭘	⭘	⭘	○	●	–
External DB	●	○	⭘	–	⭘	–	●	–
New Element(s)	–	●	●	●	●	●	●	–

In order to enable the annotation of elements in a MOF-based software model, with respect to provide easily synchronizable and versionable information, we selected MOF Annotations to persist information about detected defects, context factors, and traceability information. Furthermore, the annotation mechanism allows embedding complex information within a software model using a XML schema [39] to describe and structure the specific content for every specific annotation. The XML schema represents a metamodel that allowed us to define the substructures for the information on defects, traceability, etc.

1.4 Paper Outline

In this paper, we present how additional information about defects, context, decisions, quality, or traceability can be embedded into CIM- or PIM-level software models. After describing requirements and possible mechanisms to store this information (cf. section 1), we describe the design of a metamodel to store this additional information in section 2. Furthermore, in section 3, we presented a tool that demonstrates how this information can be used in a PIM-level software modeling tool such as Topcased. This tool uses the information about quality defects within a PIM to visualize defects directly in the software model diagrams. Related work from the traceability and design decision documentation areas is presented in section 4. Finally, in section 5 we discuss our contribution and state future research opportunities.

2 A Metamodel for Defect and Traceability Annotations

While traceability, decision, quality, and context information has to be annotated manually (for now), defects are identified by diagnostic mechanisms that analyze the system and find typical recurring problems, which have a negative effect on a quality aspect (e.g., maintainability, portability, or usability) [26].

The three types of defect-related information embedded are: *Defect Annotations* (with information about the diagnosed quality defects), *Symptom Annotations* (with information on the identified symptoms), and *Treatment Annotations* (which are used to store the treatments applicable for removing the diagnosed quality defects). Other information types include:

- *Context Annotations* (with information on used design pattern roles or special “stereotypes”) are used to differentiate the diagnosis but can also describe defect-unrelated characteristics of the software system (e.g., which packages are part of the view layer).
- *Decision Annotations* (with decisions about the design or quality defects, such as “ignore”) are used to persist decisions made about the design or the defects diagnosed and inspected.
- *Quality Annotations* (with information on quality assessments or the impact of defects on the quality) are used to describe the quality of a (part of a) software model or the impact of defects or context factors on the quality aspects.

Traceability information uses just one type of annotation (*Trace Annotations*) that realizes traces from one element to one or more other elements (e.g., from one CIM element to multiple PIM elements (downward), from one PIM to multiple CIM elements (upward), or from one PIM to multiple other PIM elements (sideward)). Multiple types of references can be used between abstractions (i.e., up- and downwards) and within abstractions (e.g., sidewards between diagrams.

Annotations do not only concern one element in one model or abstraction layer. While single-location annotations are enclosed within one abstraction at one element, *multi-location annotations* can refer to other elements within the same model. Furthermore, both types of annotations might refer to elements in another abstraction level, e.g., to document rationales for not removing a defect or to pinpoint a cause or (design) decision (e.g., in a CIM).

Fig. 1 shows the three aspects of our metamodel and represents a extended version of a smaller on presented in [27]. The first aspect focuses on the specification (right center); it outlines the generic approach as proposed by MOF [21] for annotating model elements with additional information (metainformation) using the *Tag* entity. It introduces the base model elements of BPMN-based CIMs (*BPDM::Element*) and UML-based PIMs (*UML::Element*) both deriving from *MOF::Element* as common model element abstraction. Regarding the second aspect (center), the realization in eclipse either implements or maps (center of Fig. 1) these concepts. For MOF, the element *MOF::Element* is mapped to the Ecore element *Ecore::EModelElement* of the Eclipse Modeling Framework (EMF) and *MOF::Tag* is mapped to *Ecore::EAnnotation*. Similarily, for CIM-modeling, the BPMN element *BPDM::Element* is mapped to the SOA Tools Platform (STP) project object *STP::BPMN::NamedBpmnObjects*. Finally, the element *UML::Element* of OMG’s UML are implemented (a one-to-one representation) by the Model Development Tools (MDT) project’s UML2 *MDT::UML2::Element*.

Furthermore, the third aspect is represented in Fig. 1 which presents a XML-based [38] metamodel (left) for defect-, decision-, context-, quality-, and traceability-oriented model metainformation and shows how actual metainformation is embedded using the tagging mechanism within annotations. A metamodel similar to the traceability information metamodel is used by Feng et al. [13] for external traceability models.

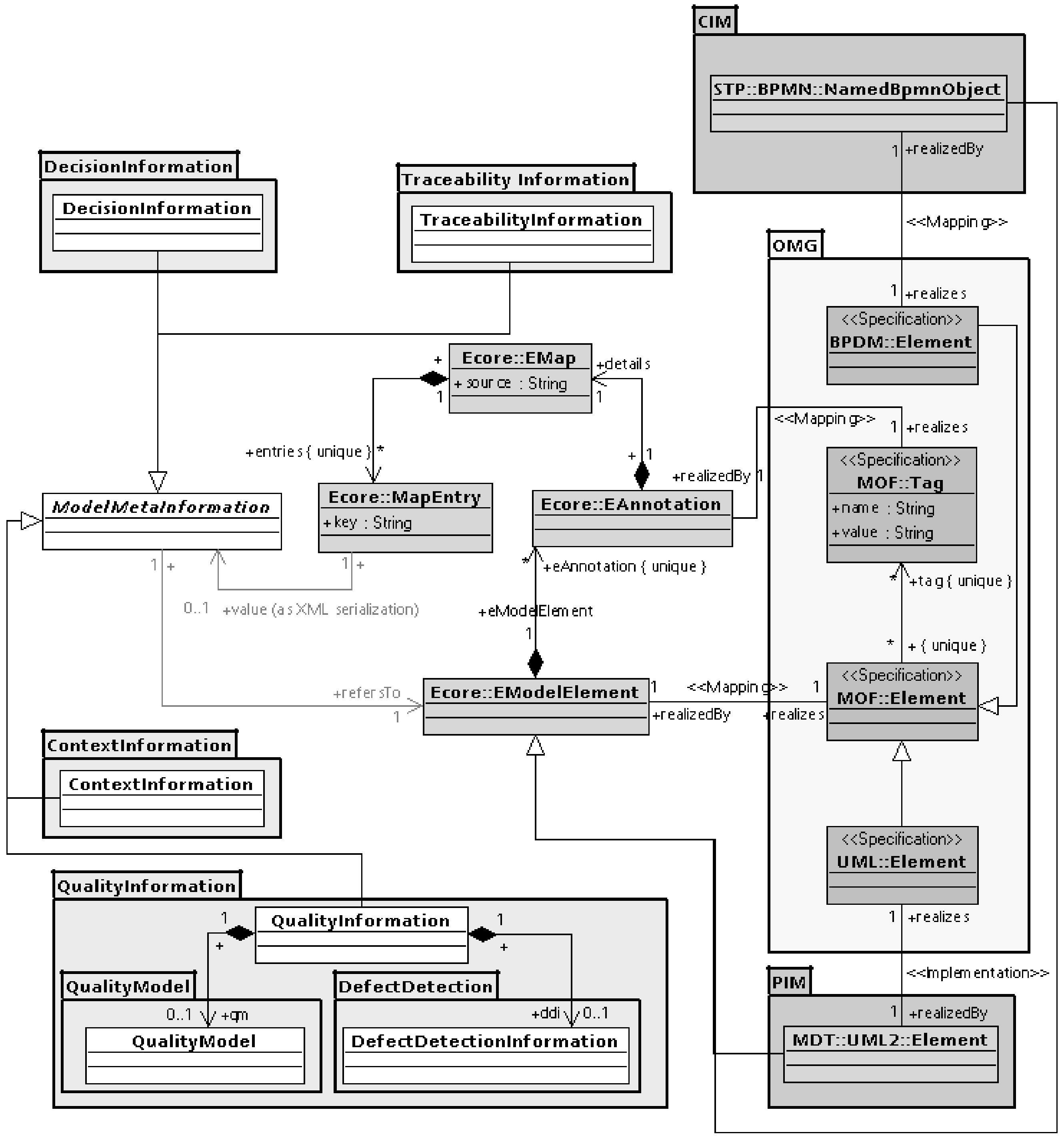


Fig. 1. Metamodel for Quality and Traceability Information in CIM and PIM Models

A model element may have multiple annotations (*EAnnotations*) associated with it, each consisting of a *source* URI denoting an annotation's type and an arbitrary number of *key/value* pairs. Following the structure of annotations, we bundle all model metainformation in a single annotation element, but distributing information internally to multiple *key/value* pairs according to the information's scope/type (e.g. «Traceablity» or «Quality»). That is, *key* determines the type/scope of the XML-based metainformation assigned to *value*.

2.1 Defect-related Information

In the context of VIDE-DD, detecting *quality defects* is the main technique of determining the quality of a software model. Fig. 2 represents the metamodel for the quality-related information which consists of information about the quality model used (cf. section 2.2) and the detected defects. A quality defect represents a system-independent defect at one or more model elements with a negative *impact* on certain *quality aspects*. Defects are diagnosed on the basis of one or more quantifiable characteristics of a model or its model elements, so-called *symptoms*. The intensity to which symptoms promote related defects differs and depends, amongst others, largely on the characteristic's deviation from previously defined *threshold*(s). For removing a defect or mitigating a defect's (negative) impact on certain quality aspects, *treatments* refer to available techniques (e.g. refactorings [14]).

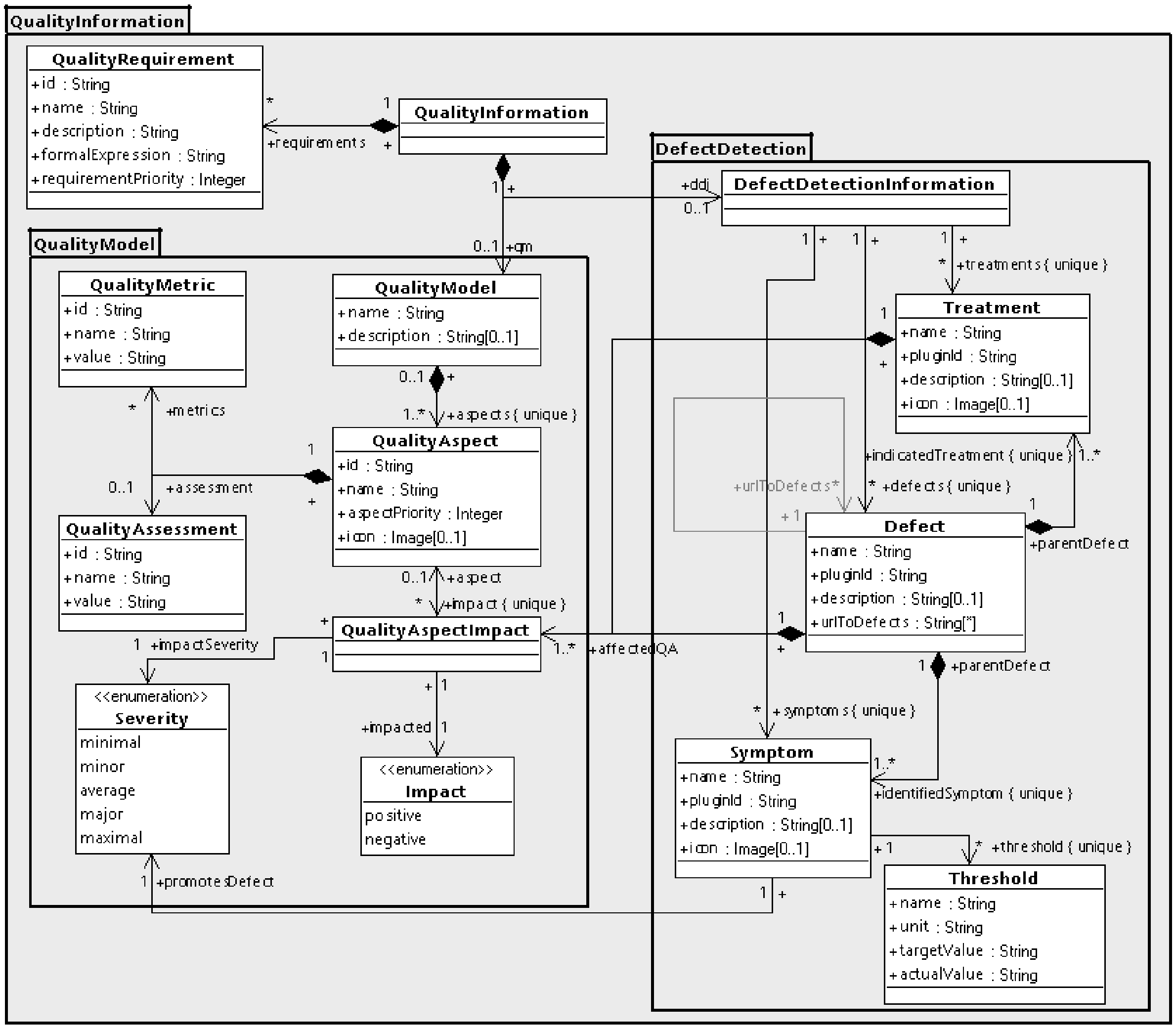


Fig. 2. Quality Information Metamodel: Defect Detection and Quality Model

The exemplary annotation in Fig. 3 illustrates the XMI-serialization of *defect detection information*: A *Lazy Class* [14] defect has been diagnosed for the PIM-level class *Opportunity* based on the *Number of Operations*. Hence, a negative impact on the declared quality aspect *Maintainability* is expected, treatable by applying an *Inline Class* [14] refactoring.

```
<uml:Model>
 ...
 <packagedElement xmi:type="uml:Class" xmi:id="_CyIsaF-fEdySHqlLXw_Tew" name="Opportunity">
  <eAnnotations source="http://www.iese.fraunhofer.de/ModelMetaInformation">
   <details key="QualityInformation" value="
    <!-- BEGIN: Embedded QualityInformation XML-string -->
    <QualityInformation>

     <DefectDetectionInformation>
      <Defects>
       <Defect name="Lazy Class" description="Class Opportunity provides
                    not enough functionality to justify its existence."
               pluginId="de.fhg.iese.modeldefectdetection.diagnosis.lazyclass"
               defectiveElement="_CyIsaF-fEdySHqlLXw_Tew">
        <IdentifiedSymptoms>
         <Symptom name="Low Number of Operations"
                  description="Number of operations is below threshold" pluginId="analysis.noo"
                  sourceElement="_CyIsaF-fEdySHqlLXw_Tew"
                  parentDefect="diagnosis.lazyclass" promotesDefect="major">
          <Thresholds>
           <Threshold name="Lower Threshold" unit="Integer" targetValue="6" actualValue="2"/>
          </Thresholds>
         </Symptom>
        </IdentifiedSymptoms>
        <AffectedQualityAspects>
         <QualityAspectImpact id="ISO9126_Maintainability" impact="negative" severity="major"/>
        </AffectedQualityAspects>
        <IndicatedTreatments>
         <Treatment name="Inline Class" pluginId="refactoring.inlineclass"
                    description="Move all features of Opportunity into another class and delete it."
                    destinationElement="_CyIsaF-fEdySHqlLXw_Tew" parentDefect="diagnosis.lazyclass"/>
        </IndicatedTreatments>
       </Defect>
      </Defects>
     </DefectDetectionInformation>

     <QualityModel name="QM1" description="">
      <QualityAspect id="ISO9126_Maintainability" name="Maintainability" aspectPriority="2"
                     description="The ease with which a software system or component can be modified…" />
     </QualityModel>

    </QualityInformation>
    <!-- END: Embedded QualityInformation XML-string -->
   "/>
  </eAnnotations>
 </packagedElement>
 ...
</uml:Model>
```

Fig. 3. Serialization of Defect-related Information

Furthermore, we distinguish single- from multi-location defects [28]. A single-location defect (e.g. Lazy Class [14]) affects one model element (e.g., a class), whereas multi-location defects apply to more than one element within the same model. For example a *Shotgun Surgery* [14] defect is present, when, due to strong coupling of classes, a change in one class requires many subsequent changes in other classes. As each concerned class is annotated with defect information, it is necessary to interrelate this information, e.g. for eliciting and applying adequate treatments. Thus, *urlToDefects* (cf. Fig. 1) allows for referencing related defects at other model elements.

2.2 Quality Model-related Information

Information about the quality model used is represented using a metamodel which is also shown in Fig. 2. A *quality model* consists of multiple *quality aspects* which are measured using *quality metrics* or are *assessed* by an engineer. The quality aspects are impacted by quality defects with a negative *impact* on certain *quality aspects* with different *severity*. We differentiate between the following four different quality-related information blocks:

- Quality (Sub-)Models [17]: A quality model can be annotation at a part of software model in order to specify the which quality aspects are more important than others. For example, the quality model for the (database) model layer can state that performance is more important than maintainability and this is more important than portability (i.e., Performance > Maintainability > Portability) while the quality model for the (GUI) view layer may state that usability is more important than portability (i.e., Usability > Portability).
- Quality Metrics [13]/ Measures [18]: Symptoms used in the diagnosis process are often identified characteristics or measured metrics about a part of software model. These symptoms are potentially useful in future diagnosis runs, manual inspections, visualization techniques, or reporting steps.
- Quality assessment results: Formal and, especially, informal results from quality assurance processes such as inspections, reviews, tests, model checking, etc. can be annotated at a part of a software model. This persists unspecific (e.g., treatment-less or general problems resp. statements), such as "Maintainability assessed as 'good'", "Readability assessed as 'bad'", or "Naming inconsistent with package X", which can be used for general improvement approaches such as systematic improvement (e.g., by observing the usage behavior to improve performance) or employee training.
- Quality requirements: Functional and non-functional requirements state information from the customers, stakeholders, or domain experts that needs to be fulfilled by a (part of a) system. These requirements can either be documented in a requirement document and linked using traceability information or, especially when they are very technical, specific, and design-focused, documented within the software model itself. These quality requirements document constraints such as "Braking system controlled by Controller X must react within 3ms", "GUI for process X, realized in classes A to D, changes often and requires high changeability", or "Data transferred from method Y over Bluetooth should not excess 512 byte".

Fig. 4 gives a simplified exemplary XMI-serialization [37] of a UML-based PIM model having a model element annotated with *quality information*. According to the (non-functional) requirements a software system has to meet, a quality model defines and prioritizes mandatory *quality aspects* and thus, is the basis for interpreting or verifying the quality of a software model.

```
<uml:Model>
 ...
 <packagedElement xmi:type="uml:Class" xmi:id="_CyIsaF-fEdySHqlLXw_Tew" name="Opportunity">
  <eAnnotations source="http://www.iese.fraunhofer.de/ModelMetaInformation">
   <details key="QualityInformation" value="
    <!-- BEGIN: Embedded QualityInformation XML-string -->
    <QualityInformation>

     <QualityModel name="Efficiency and Maintainability Profile" description="">
      <QualityAspects>
       <QualityAspect id="ISO9126_Maintainability" name="Maintainability" aspectPriority="2"
          description="The ease with which a software system or component can be modified …" />
       <QualityAspect id="ISO9126_Efficiency" name="Efficiency" aspectPriority="3"
          description="The performance of the software and the amount of resources used …" />
      </QualityAspects>
      <QualityMetrics>
       <QualityMetric id="qm-loc" name="Lines Of Code"        value="333">
       <QualityMetric id="qm-noa" name="Number Of attributes" value="120">
      </QualityMetrics>
      <QualityAssessments>
       <QualityAssessment id="qm-ea-x27" name="Subjective Efficiency evaluation" value="good">
      </QualityAssessments>
      <QualityRequirements>
       <QualityRequirement id="x4712" name="Status message feedback time" requirementPriority="2"
                           description="Status messages should be presented in less than 5 msec"
                           detailedDescriptionURL="http://...." formalExpression="" />
      </QualityRequirements>
     </QualityModel>

    </QualityInformation>
    <!-- END: Embedded QualityInformation XML-string -->
   "/>
  </eAnnotations>
 </packagedElement>
 ...
</uml:Model>
```

Fig. 4. Serialization of Quality-related Information

2.3 Decision-related Information

Many decisions are made during the design and quality assurance of a software model. While we document design decisions for later guidance and remembrance, other decisions and rationales why a quality defect was (not) removed, how it was handled, or why a trace needs to be documented are also valuable for the further evolution and maintenance of the software model and, therefore, should be stored.

Currently we specify *design decisions* as simple attribute value pairs where the decisions are informally encoded via free text and a small classification is used (see Fig. 1). This classification uses taxons, such as Design Pattern, Reference Architecture, or Layered Model.

Furthermore, for *defect decisions* we currently use the following taxons, which also support the quality improvement process by stating information for other people (e.g., to delegate work to specialized quality engineers) or later times when more time is available (e.g., by emphasizing a defect):

- *Ignore*: The quality defect should be ignored now and for later diagnosis activities. This will not remove the defect but will hide it from the diagram.
- *Emphasize*: Emphasizing a defect results in the increase of the defects priority. Therefore, priority-based methods to remove defects in a system will handle this defect earlier.
- *De-Emphasize*: De-Emphasizing a defect results in the decrease of the defects priority. Therefore, priority-based methods to remove defects in a system will handle this defect later (than normally).
- *Inspect later*: The quality defect should be inspected later by an expert for this problem (e.g., a specially trained quality engineer).
- *Re-engineer later*: larger changes are needed that cannot be conducted by the user (i.e., software designer) and might be discussed with another architect, department, etc.

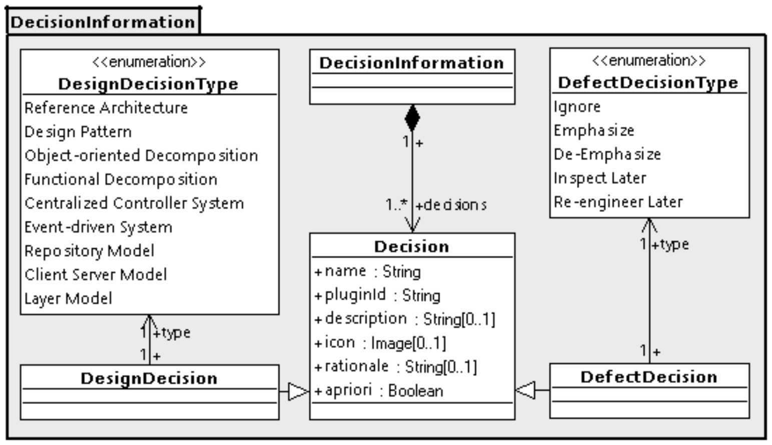


Fig. 5. Decision Information Metamodel

Fig. 6 gives a simplified exemplary XMI-serialization [37] of a model element with an decision annotation. This information consists of information about a design and defect decision.

```
<uml:Model>
 ...
 <packagedElement xmi:type="uml:Class" xmi:id="_CyIsaF-fEdySHqlLXw_Tew" name="Opportunity">
  <eAnnotations source="http://www.iese.fraunhofer.de/ModelMetaInformation">
   <details key="TraceabilityInformation" value="
    <!-- BEGIN: Embedded decision Information XML-string -->
    <DecisionInformation>

     <Decisions>
      <Decision type="DefectDecision::Ignore" defect="diagnosis.lazyclass" name="Ignore Defect"
       description="Ignores the defect this decision is assigned to when assessing the quality of
         the model" rationale="The underlying class is an abstract base class where the deriving
         child classes are carrying the main functionality" apriori="false"/>
      <Decision type="DesignDecision::DesignPattern" name="Design Pattern"
       description="Denotes that the underlying element is part of/implements a certain design pattern"
       rationale="The class implements the Strategy design pattern" apriori="true"/>
       ...
     </Decisions>

    <DecisionInformation>
    </ DecisionInformation >
    <!-- END: Embedded DecisionInformation XML-string -->
   "/>
  </eAnnotations>
 </packagedElement>
 ...
</uml:Model>
```

Fig. 6. Serialization of Decision Information

2.4 Context-related Information

Quality defects are not necessarily problematic at every location they are found in. Sometime the design of the object-oriented software system demands a structure that seems to be a quality defect but is intentional. For example, a “large class” [14] is a problem as the readability and understandability of the class is reduced. However, a “façade class” [15] gives access to a group of classes (e.g., a package) by aggregating many delegator methods and, therefore, is often unavoidably large.

In order to differentiate and, in the end, improve the diagnosis process, context factors, such as used design patterns, needs to be available. Context factors describe the role a software entity plays in the software system, which might be used during the diagnosis a) to block the diagnosis of a quality defect, b) to change the diagnosis mechanism, or c) to emphasize the quality defects severity.

In the this paper the concept “context” includes *neutral context factors* with no (known) positive or negative effect on a software quality, *positive context factors* (i.e. quality promoters such as design patterns) with a positive effect on a software quality, and *negative context factors* (i.e., quality defects) with a negative effect on the software quality. While the use of quality defects or quality promoters would result in defect-sensitive and promoter-sensitive diagnosis techniques we subsume them under the label “*context-sensitive diagnosis*”. Furthermore, characteristics of model element, such as metrics or visibility information, can be used as context factors. For example, the size of a method can be used to guide the diagnosis mechanism in the case of mistakable names. If parameter names are identical to the names of class variables (e.g., in constructor methods) this is a lesser problem in small and manageable classes than in larger classes (based on the characteristic (resp. metric) “lines of code”).

Some of these context factors are already explicitly documented within the model of a software system. For example, getter and setter methods can be identified by their “get” or “set” prefix and some systems enforce the use of pattern roles within class names - in the source code for the eclipse IDE adapters have the postfix “adapter” and interface classes start with “I” (e.g., “IModelAdapter”). However, not all of these context factors are documented directly in the available information sources. Some context factors can be extracted or inferred from the source code. Annotations such as `@deprecated` can be extracted and used to block the diagnosis as these elements will vanish in one of the next versions. Language specific keywords, such as `abstract` can be extracted from the source code and used to infer a superclass, which can be used to control the diagnosis of the “lazy class” [14] quality defect.

Furthermore, context factors such as `constructor method` can be inferred from the source code itself and used to change diagnosis rules for quality defects such as “long parameter list” as the class’s construction typically requires more parameters than standard methods.

In order to be used, the context model needs to fulfill several requirements. For the human reader it needs to be expressive and understandable. For the use in the diagnosis process it needs to be associated with an element of the information model (as it describes it’s context).

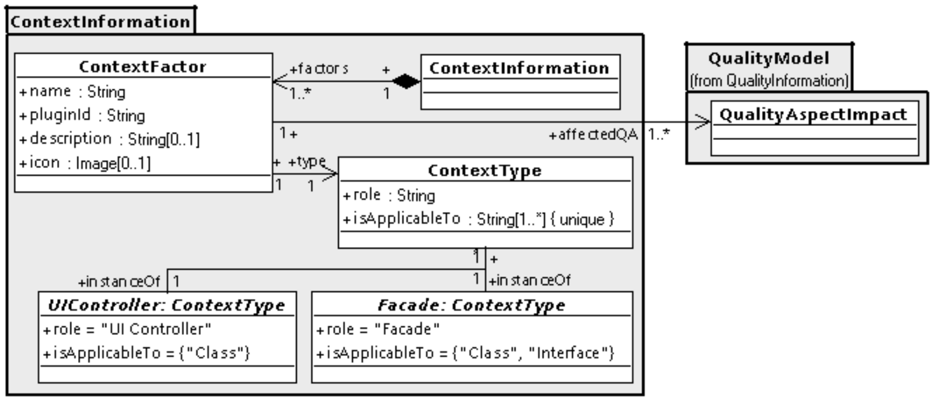


Fig. 7. Context Information Metamodel

Fig. 8 gives a simplified exemplary XMI-serialization [37] of a model element with an context annotation. This information consists of information about their type, specific roles, and impact on quality aspects.

```
<uml:Model>
 ...
 <packagedElement xmi:type="uml:Class" xmi:id="_CyIsaF-fEdySHqlLXw_Tew" name="Opportunity">
  <eAnnotations source="http://www.iese.fraunhofer.de/ModelMetaInformation">
   <details key="TraceabilityInformation" value="
    <!-- BEGIN: Embedded Traceability Information XML-string -->
    <ContextInformation>

    <Contexts>
      <Context name="Facade Class" pluginId="de.fhg.iese.modeldefectdetection.menus"
               Role="Facade" isApplicableTo="Class, Interface"
               description="The facade provides an interface to a larger part of the system" />
    </Contexts>

    </ContextInformation>
    <!-- END: Embedded Traceability Information XML-string -->
   "/>
  </eAnnotations>
 </packagedElement>
</uml:Model>
```

Fig. 8. Serialization of Context Information

2.5 Traceability-related Information

A model element’s *traceability information* comprises one to many *traces* to elements at both, different as well as same abstraction level. As presented in Fig. 10, the key component of a trace is *urlToElement* for identifying related elements using a URL reference. The URL syntax is a path to the containing model repository, followed by a model identifier (the model’s name) and the XMI-ID of the model element. To qualify the relation of two elements linked by a trace, different types of references can be assigned to a) *traces between abstractions*, such as “realizes / is realizes by”, “refines / is refined by”, “specifies / is specified by“, “requires / is required by”, etc. and b) *traces within abstractions*, such as “includes / is part of”, “verifies / is verified by”, “defines / is defined by”, “constrains / is constrained by”, etc. (see [35] or [10]).

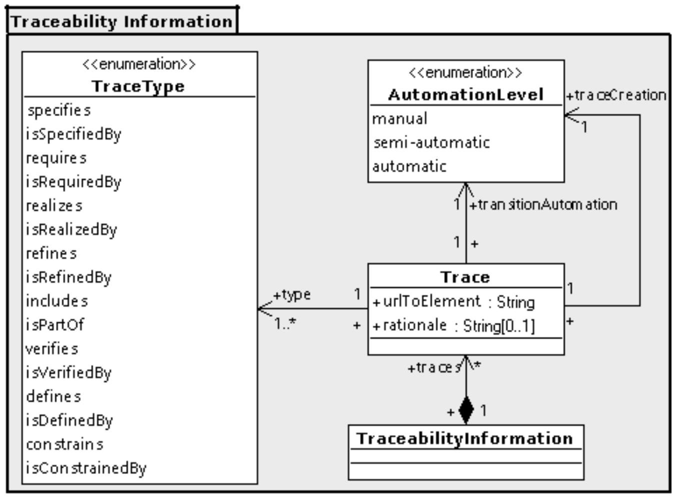


Fig. 9. Traceability InformationMetamodel

Fig. 10 gives a simplified exemplary XMI-serialization [37] of a model element with an traceability annotation. This information consists of information about traces, their type, and origin.

```
<uml:Model>
 ...
 <packagedElement xmi:type="uml:Class" xmi:id="_CyIsaF-fEdySHqlLXw_Tew" name="Opportunity">
  <eAnnotations source="http://www.iese.fraunhofer.de/ModelMetaInformation">
   <details key="TraceabilityInformation" value="
    <!-- BEGIN: Embedded Traceability Information XML-string -->

    <TraceabilityInformation>
     <Trace urlToElement="http://iese.fhg.de/SalesOpportunity_CIM.bpmn#_TG7coT3iEd2hQ-HeytPXvA"
            type="realizes" rationale="Implementation of Opportunity data object"
            traceCreation="automatic" transitionAutomation="automatic" />
    </TraceabilityInformation>

    <!-- END: Embedded Traceability Information XML-string -->
   "/>
  </eAnnotations>
 </packagedElement>
</uml:Model>
```

Fig. 10. Serialization of Traceability Information Annotation in PIM

As generative model-driven development relies on model transformations between abstraction levels, the information if a *trace creation* or *transition* between two related elements has been carried out manually, semi-automatically, or automatically is of interest for e.g. evaluating the quality of model transformations/model generators or determining the overall automation-level. The XMI-serialization of traceability information between an Opportunity data object at CIM-level and its implementation class at PIM-level is exemplified in Fig. 10 (PIM-to-CIM) and Fig. 11 (CIM-to-PIM).

```
<bpmn:BpmnDiagram>
 ...
 <artifacts xmi:type="bpmn:DataObject" xmi:id="_TG7coT3iEd2hQ-HeytPXvA" name="Opportunity">
  <eAnnotations source="http://www.iese.fraunhofer.de/ModelMetaInformation">
   <details key="TraceabilityInformation" value="
    <!-- BEGIN: Embedded Traceability Information XML-string -->

     <TraceabilityInformation>
      <Trace urlToElement="http://iese.fhg.de/SalesOpportunity_PIM.uml#_CyIsaF-fEdySHqlLXw_Tew"
             type="isRealizedBy" rationale="Implementation of Opportunity data object"
             traceCreation="automatic" transitionAutomation="automatic"/>
     </TraceabilityInformation>

     <!-- END: Embedded Traceability Information XML-string -->
   "/>
  </eAnnotations>
 </artifacts>
 ...
</bpmn:BpmnDiagram>
```

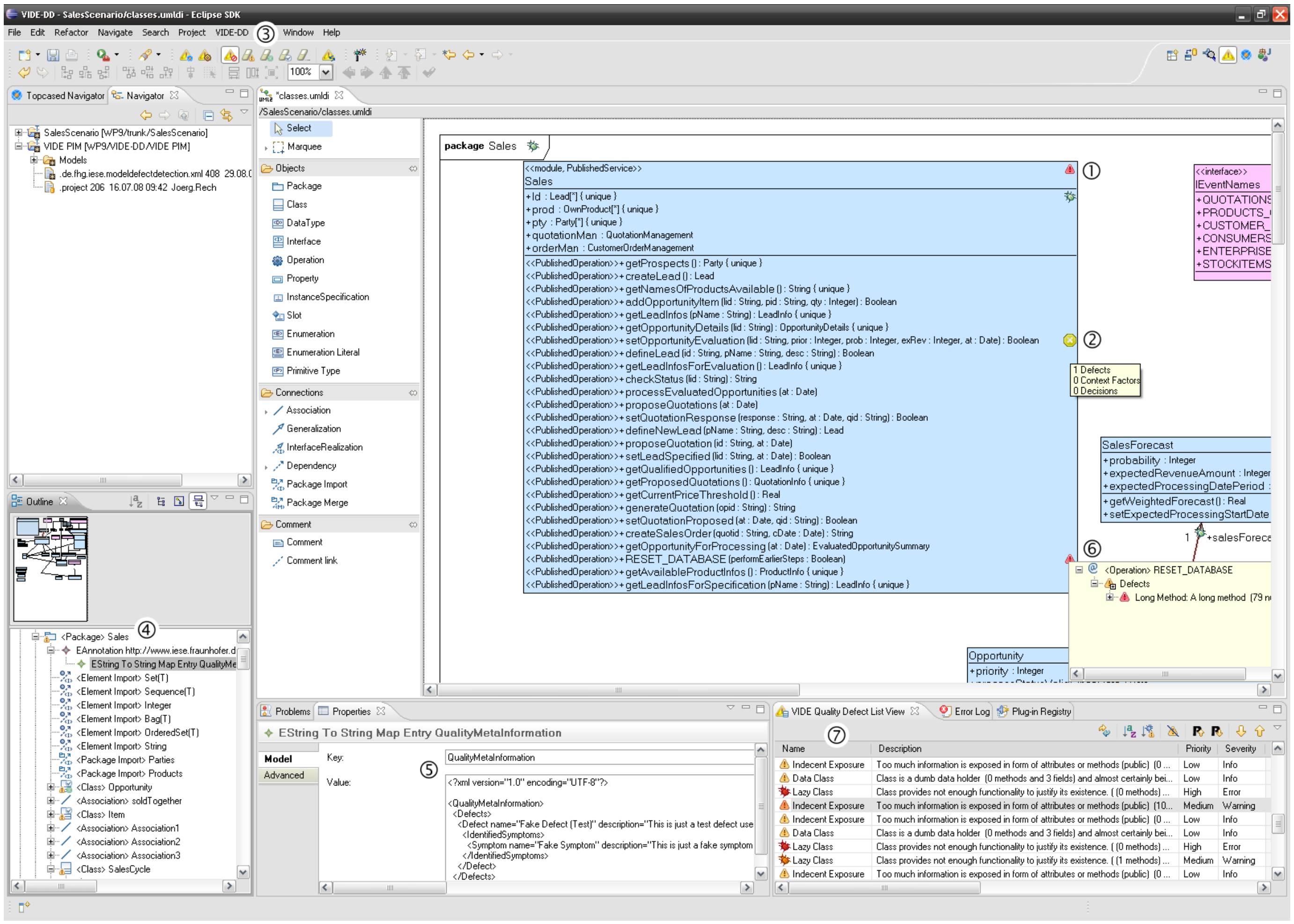
Fig. 11. Serialization of Traceability Information Annotation in CIM

3 Using (Defect) Annotations in Modeling Environments

The VIDE Quality Defect Detector (VIDE-DD) is responsible for the automatic diagnosis and presentation of quality defects within platform independent models, in order to inform designers and maintainers about potential threats to model quality. This encompasses structural as well as behavioral diagrams used to visually model a VIDE-based system. VIDE-DD uses information about the software model, stored within the VIDE PIM repository, to analyze the model, diagnose quality defects, and annotate the model using Annotations [31]. The information stored within these Annotations is then used by the quality defect presenting mechanism (an extension to the diagrams presentation mechanisms) to visually enrich the diagrams (within the VIDE visual editor) with information on these defects.

3.1 Defect Presentation

As shown in Fig. 12 the VIDE-DD extends the Topcased modeling environment [33] and decodes the information within the annotation to decorate an element (e.g., a class) with a defect icon or to list all annotations to the user (see ⑥). Defective relations are decorated using a red "aura" (see ①). In VIDE-DD the defects, decisions and context factors are presented in four different ways to the user: a) as icon annotations within the diagram, b) as decorators in the model outline (only the existence of at least one defect), c) in the special defect view, and d) as markers in the standard eclipse problem view. Furthermore, for debugging purposes, the XML-code of the element can be viewed via the properties view of Topcased. In order to differentiate the different kinds of defects they have different severities (as calculated by different rules and thresholds), which are used to emphasize the defect using different icons and colors.



EString To String Map Entry QualityMetaInformation

Key: QualityMetaInformation

Value:

```
<?xml version="1.0" encoding="UTF-8"?>

<QualityMetaInformation>
 <Defects>
  <Defect name="Fake Defect (Test)" description="This is just a test defect use
   <IdentifiedSymptoms>
    <Symptom name="Fake Symptom" description="This is just a fake symptom
   </IdentifiedSymptoms>
  </Defect>
 </Defects>
```

Name	Description	Priority	Severity
Indecent Exposure	Too much information is exposed in form of attributes or methods (public) (0 ...	Low	Info
Data Class	Class is a dumb data holder (0 methods and 3 fields) and almost certainly bei...	Low	Info
Lazy Class	Class provides not enough functionality to justify its existence. ((0 methods) ...	High	Error
Indecent Exposure	Too much information is exposed in form of attributes or methods (public) (10...	Medium	Warning
Indecent Exposure	Too much information is exposed in form of attributes or methods (public) (0 ...	Low	Info
Data Class	Class is a dumb data holder (0 methods and 3 fields) and almost certainly bei...	Low	Info
Lazy Class	Class provides not enough functionality to justify its existence. ((0 methods) ...	High	Error
Lazy Class	Class provides not enough functionality to justify its existence. ((1 methods) ...	Medium	Warning
Indecent Exposure	Too much information is exposed in form of attributes or methods (public) (0 ...	Low	Info

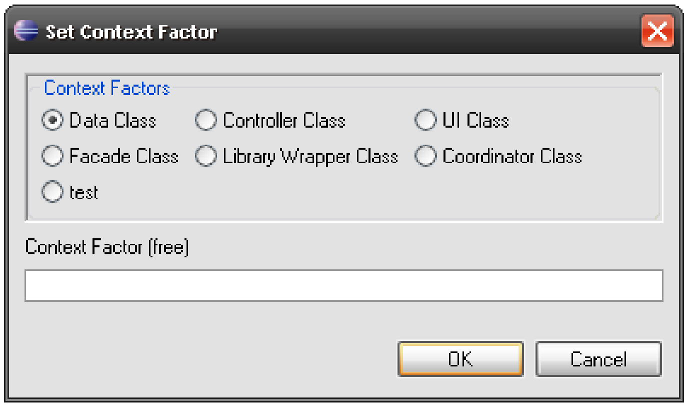
Fig. 12. The VIDE Quality Defect Detector

The presentation layer of the VIDE-DD focussed on the visualization of quality defects in UML diagrams. Fig. 12 shows the different components of the VIDE-DD presentation layer, which we introduce briefly:

- The graphical editor from Topcased is adapted to present Quality *Defects for diagrams, packages, references and classes* ① as well as for *methods and attributes* ②. Defective model elements are presented using small icons that can be varied based on the defect type (not shown in the screenshot).
- Buttons & Menu ③ enable the user to easily initiate a scan of the complete workspace or the selected Java file.
- The defects are annotated directly in the model using UML Annotations (resp. EAnnotations in EMF), which are shown in the model tree ④.
- The *defect description* ⑤, as well as the symptoms used in the diagnosis process is stored using a XML-based structure in the value-tag of the UML annotation (see 3.2).
- This *information in the annotations* can also be displayed in the graphical editor within a context menu ⑥. The menu shows the tree structure of the XML-based information including descriptions and symptoms.
- The *quality defect view* ⑦ lists all defects diagnosed within the model and enables browing through the defects and diagrams where they occur.

After the model is analyzed the defects are presented in the diagram ①, the VIDE-DD defect list view②, the standard eclipse problem view ③, and in the outline as decorators for the elements within the model ④.

The defects are embedded by the automated diagnosis system while the decisions have special actions (e.g., “Ignore Defect”) that embeds the information. Context factors are set using a radio button selection menu (see Fig. 13), which can be extended by freely definable context factors.



Set Context Factor

Context Factors

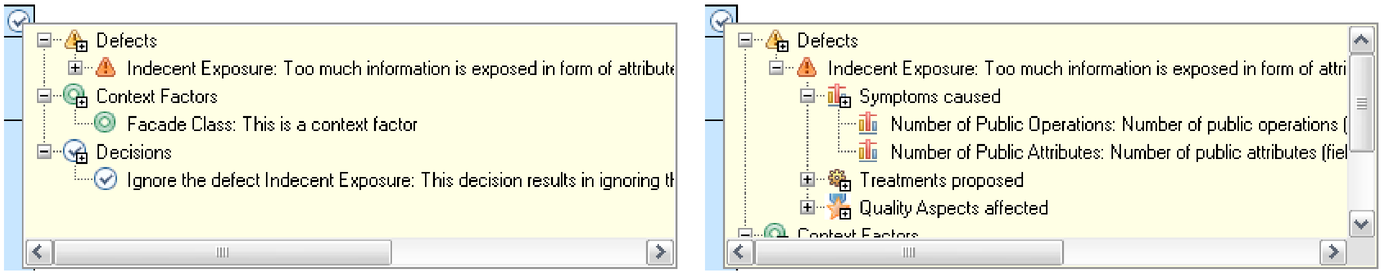
(● Data Class) ○ Controller Class ○ UI Class
○ Facade Class ○ Library Wrapper Class ○ Coordinator Class
○ test

Context Factor (free)

Fig. 13. Radio Button Selection Menu for Context factors

3.2 *Defect Explanation*

For a better description of the defect, decisions, or context factors the icons in the diagram can be selected. They show a tree list of the diagnosed defects, symptoms, treatments, etc. In Fig. 14 (left) annotated defects, context factors, and decisions are shown and the software designer can get more information on the defect, used symptoms, proposed treatments and affected quality aspects (see Fig. 14 right). Please note that the context factor “façade class” also blocked a “Data Class” quality defect and the “Ignore the defect Indecent Exposure” decision hides this defect – the reason why the icon for the class (see top left “checkmark” icon in Fig. 14) is a “checkmark”.

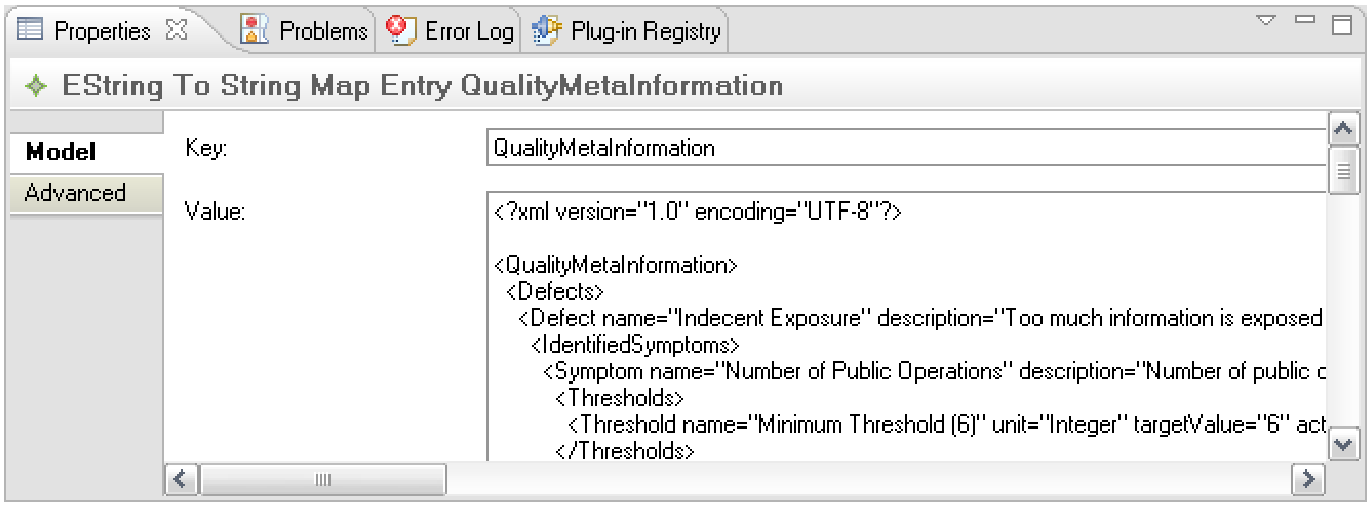


Defects
Indecent Exposure: Too much information is exposed in form of attribute
Context Factors
Facade Class: This is a context factor
Decisions
Ignore the defect Indecent Exposure: This decision results in ignoring th

Defects
Indecent Exposure: Too much information is exposed in form of attri
Symptoms caused
Number of Public Operations: Number of public operations (
Number of Public Attributes: Number of public attributes (fiel
Treatments proposed
Quality Aspects affected
Context Factors

Fig. 14. Defect explanation in the UML Diagrams of Topcased

Furthermore, the raw data about the annotated information can be viewed using the Properties view of Topcased/eclipse. Here the user has access to all the data stored about the defects in the model.



EString To String Map Entry QualityMetaInformation

Key: QualityMetaInformation

Value:

```
<?xml version="1.0" encoding="UTF-8"?>

<QualityMetaInformation>
 <Defects>
  <Defect name="Indecent Exposure" description="Too much information is exposed
   <IdentifiedSymptoms>
    <Symptom name="Number of Public Operations" description="Number of public c
     <Thresholds>
      <Threshold name="Minimum Threshold (6)" unit="Integer" targetValue="6" act
     </Thresholds>
```

Fig. 15. The Topcased Properties View with Annotated Information

3.3 Defect Decorators

Beside the presentation of defects in the diagram this information is also presented in the model outline. As shown in Fig. 15 the three kinds of information are presented to the user via so-called "decorators" to the icons of the model elements (as used, for example, in contemporary programming environments for source code such as eclipse). The three kinds of information are all visualized in different corner – decisions in the top right corner ①, defects in the bottom left corner ②, and context factors in bottom right corner ③.

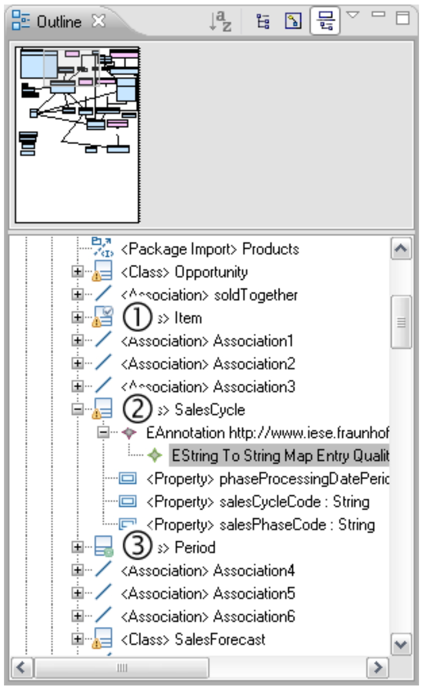
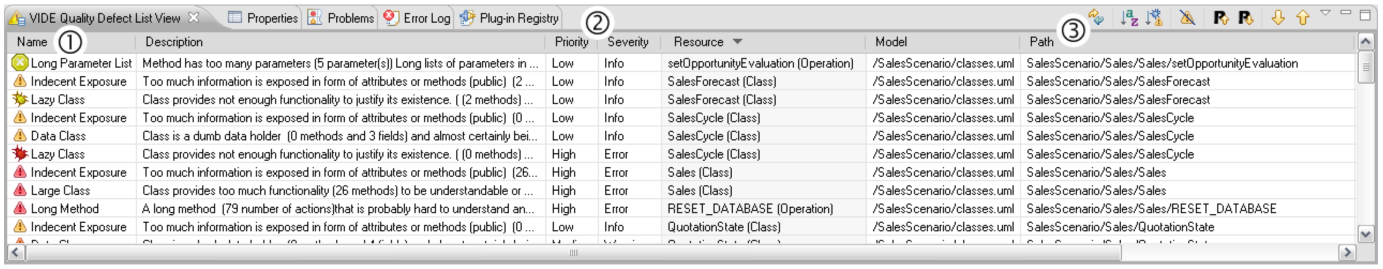

Outline
<Package Import> Products
<Class> Opportunity
<Association> soldTogether
①
» Item
<Association> Association1
<Association> Association2
<Association> Association3
②
» SalesCycle
EAnnotation http://www.iese.fraunhof
EString To String Map Entry Qualit
<Property> phaseProcessingDatePeric
<Property> salesCycleCode : String
<Property> salesPhaseCode : String
③
» Period
<Association> Association4
<Association> Association5
<Association> Association6
<Class> SalesForecast

Fig. 16. The Topcased Outline with Defect, Decision, and Context Annotations from VIDE-DD

3.4 The VIDE-DD View

The main presentation area for defects is the VIDE defect view. Here the VIDE programmer can see all defects (①) in the model and sort them by name, priority (②), or element. As functionality he can choose to ignore a defect, remove a defect, or increase or decrease the priority or severity of a defect (either via the buttons at ③ or via a context menu).



Name	Description	Priority	Severity	Resource	Model	Path
Long Parameter List	Method has too many parameters (5 parameter(s)) Long lists of parameters in ...	Low	Info	setOpportunityEvaluation (Operation)	/SalesScenario/classes.uml	SalesScenario/Sales/Sales/setOpportunityEvaluation
Indecent Exposure	Too much information is exposed in form of attributes or methods (public) (2 ...	Low	Info	SalesForecast (Class)	/SalesScenario/classes.uml	SalesScenario/Sales/SalesForecast
Lazy Class	Class provides not enough functionality to justify its existence. ((2 methods) ...	Low	Info	SalesForecast (Class)	/SalesScenario/classes.uml	SalesScenario/Sales/SalesForecast
Indecent Exposure	Too much information is exposed in form of attributes or methods (public) (0 ...	Low	Info	SalesCycle (Class)	/SalesScenario/classes.uml	SalesScenario/Sales/SalesCycle
Data Class	Class is a dumb data holder (0 methods and 3 fields) and almost certainly bei...	Low	Info	SalesCycle (Class)	/SalesScenario/classes.uml	SalesScenario/Sales/SalesCycle
Lazy Class	Class provides not enough functionality to justify its existence. ((0 methods) ...	High	Error	SalesCycle (Class)	/SalesScenario/classes.uml	SalesScenario/Sales/SalesCycle
Indecent Exposure	Too much information is exposed in form of attributes or methods (public) (26...	High	Error	Sales (Class)	/SalesScenario/classes.uml	SalesScenario/Sales/Sales
Large Class	Class provides too much functionality (26 methods) to be understandable or ...	High	Error	Sales (Class)	/SalesScenario/classes.uml	SalesScenario/Sales/Sales
Long Method	A long method (79 number of actions)that is probably hard to understand an...	High	Error	RESET_DATABASE (Operation)	/SalesScenario/classes.uml	SalesScenario/Sales/Sales/RESET_DATABASE
Indecent Exposure	Too much information is exposed in form of attributes or methods (public) (0 ...	Low	Info	QuotationState (Class)	/SalesScenario/classes.uml	SalesScenario/Sales/QuotationState

Fig. 17. VIDE-DD Defect View

4 Related Work

Today, different researches have started to approach the problem of enriching software models with additional information, such as traceability links or the documentation of design decisions. However, storing additional complex information in a metamodel such as the UML [34] at PIM level is not straight-forward – an extension of the UML metamodel would result in non-standard models that are not exchangeable between tools. Besides, in order to apply similar mechanisms to models at different abstraction levels based on different (or previously unknown) metamodels, we need a generic approach that can be easily adapted to and complies with a broad range of metamodels. In this section, we present the work that is most related to our contribution.

In the traceability area [1], we find several approaches that store the traceability information in an external source or internally within (resp. together with) the model. Kolovos et al. [19] [8] differentiate between external and embedded traceability information and made a decision towards the external approach. They developed a merging mechanism that embeds the information into the model on demand but argue against the general embedded approach (based on stereotypes) as it does not support inter-model relations, pollutes the models, and degrading uniformity. However, as presented, several other mechanisms for embedding annotations – beside stereotypes – are possible and, as previously discussed, has its benefits. Oldevik and Neple [23] defined a trace metamodel for the use in model to text transformations that is not covered by our metamodel. It describes the applied transformation operations as well as the resulting effects on the text (e.g., Java file(s)). Other approaches follow these basic directions and uses external databases to store the concrete traces of requirements (e.g., Ramesh [24]) or use standard extension mechanisms for UML such as stereotypes. Letelier [20] even specified a UML-based metamodel that is transformed in a specific UML Profile for each modeling tool. Nevertheless, our approach to use annotations to store information regarding traces (and other types of information) is still new and more adequate to fulfill our previously stated requirements.

While many types of traceability links exists (see [11] for a summary), they can all be realized by a simple multi-source multi-destination relation, such as we use in out metamodel. Traceability links are typically used for impact analyses and might be manually annotated or mined using Data Mining techniques [6] [5]. For example, Walderhaug et al. [36] defined a comprehensive trace model that is aimed at supporting the whole lifecycle, supports the tracing across different tools, and allows customization.

In the area of documenting design decisions, additional approaches were developed to store these decisions with models. Similar to traceability links, the persistent storage of design decisions is not investigated – the approaches often use external files or databases. Jansen and Bosch [2] use a metamodel in their "Archium" approach, which is used in a special view on the models, in order to structure design decisions.

While these approaches demonstrate the feasibility of annotating additional information to software models, they fall short on the comprehensive annotation of different kind of information. In this respect, we combine several metamodels from different research areas, extend the information typically associated with software models with context, quality, and defect information, as well as use them among each other (e.g., by tracing defects to other models or model elements).

5 Conclusion

We presented how additional information about defects, context, decisions, quality, and traceability can be embedded in MOF-based metamodels such as UML or BPMN. We embedded this additional information within the PIM or CIM software models using annotations (i.e., MOF::Tags resp. EAnnotations). To structure the information within these annotations, we used a XML-based metamodel that supports single- and multi-location annotations from CIM-to-PIM, within PIM or CIM, and from PIM-to-CIM. Furthermore, we presented the VIDE defect detector (VIDE-DD) – a tool that integrates quality defect diagnosis into the contemporary modeling environment Topcased and uses the annotations to present them in standard diagrams. While our approach can also be seen as a step towards an annotation and documentation language for UML (similar to JavaDoc, EpyDoc, Doxygen, etc.[25]), we are still far from such an extensible and comprehensive documentation language as well as the associated technical infrastructure that could generate the API documentation, quality defect reports, or traceability representations.

In the future, more and more information will be associated with elements of a software model as well as supporting models, such as transformation models. Tools for that support quality improvement, traceability, or design activities are developed and integrated into software development and modeling environments that have to overcome the exchange, synchronization, and versioning challenges required by large distributed software projects. Furthermore, the ad-hoc visualization of multi-location defects, i.e., defects based on multiple element not visualized together on a diagram, or multi-element and multi-model traces (esp. when combining different diagrams, such as class diagrams and business process diagrams) presents challenges for future research.

Acknowledgements

This work has been partially funded by the European Union under the sixth framework programme IST-2005-2.5.5 in project VIDE ("Visualize all moDel drivEn programming"; 033606). We thank Sheridan Jeary, Melanie Coles, Sebastian Weber, Andreas Jedlitschka, and Eric Ras, for their feedback, suggestions, and encouragement for this work.

References

[1] Aizenbud-Reshef, N., Nolan, B. T., Rubin, J., and Shaham-Gafni, Y., "Model traceability," *IBM System Journal*, vol. 45, no. 3, pp. 515-526, 2006.
[2] Anton, J. and Jan, B., "Software Architecture as a Set of Architectural Design Decisions," presented at the Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, 2005.
[3] BPDM-Beta1, "Business Process Definition MetaModel (BPDM), Beta 1," OMG, OMG Adopted Specification dtc/07-07-01, July 2007 2007.
[4] BPMN-1.1, "Business Process Modeling Notation, V1.1," OMG January 2008 2008.
[5] Briand, L. C., Labiche, Y., Sullivan, L. O., and Sówka, M. M., "Automated impact analysis of UML models," *J. Syst. Softw.*, vol. 79, no. 3, pp. 339-352, 2006.
[6] Briand, L. C., Labiche, Y., and Yue, T., "Automated traceability analysis for UML model refinements," *Information and Software Technology*, vol. In Press, Corrected Proof, no. doi:10.1016/j.infsof.2008.06.002, 2008.
[7] DI-1.0, "UML Diagram Interchange Specification, version 1.0," Object Management Group, Inc. (OMG), Needham, MA, USA, Specification April 2006 2006.
[8] Drivalos, N., Paige, R. F., Fernandes, K., and Kolovos, D. S., "Towards Rigorously Defined Model-to-Model Traceability," presented at the Proc. 4th Workshop on Traceability, ECMDA'08, Berlin, Germany, 2008.
[9] EMF, "Eclipse Modeling Framework (EMF)," http://www.eclipse.org/modeling/emf/, last accessed on 1. April 2008
[10] Espinoza, A., Alarcon, P. P., and Garbajosa, J., "Analyzing and Systematizing Current Traceability Schemas," presented at the Annual IEEE/NASA Software Engineering Workshop (SEW), 2006.
[11] Espinoza, A., Alarcon, P. P., and Garbajosa, J., "Analyzing and Systematizing Current Traceability Schemas," presented at the Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop, 2006.
[12] Feng, Y., Huang, G., Yang, J., and Mei, H. M., *Traceability between Software Architecture Models*: IEEE Computer Society, 0-7695-2655-1, 2006.
[13] Fenton, N. and Pfleeger, S. L., *Software metrics (2nd ed.): a rigorous and practical approach*: PWS Publishing Co., 0-534-95600-9, 1997.
[14] Fowler, M. and Beck, K., *Refactoring: improving the design of existing code*, 1st Edition ed. Reading, MA: Addison-Wesley, ISBN: 0201485672 LCCN: 99-20765, 1999.
[15] Gamma, E. and Beck, K., *Contributing to Eclipse : principles, patterns, and plug-ins*. Boston: Addison-Wesley, ISBN: 0321205758 (alk. paper) LCCN: 2003-20914, 2004.
[16] Gamma, E., Vlissides, J., Johnson, R., and Helm, R., *Design patterns: elements of reusable object-oriented software*. Reading, MA, USA: Addison-Wesley, ISBN: 0201633612, 1995.
[17] ISO/IEC-9126-1, *Software engineering: product quality. Part 1, Quality model*, Ed. 1. ed. Pretoria: International Organization for Standardization / International Electrotechnical Commission, ISBN: 0626146747 (pbk.), 2003.
[18] ISO/IEC-25000, "Software Engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Guide to SQuaRE," Standard 2005-07-27 2005.

[19] Kolovos, D. S., Paige, R. F., and Polack, F. A. C., "On-Demand Merging of Traceability Links with Models," presented at the 2nd EC-MDA Workshop on Traceability, Bilbao, Spain, 2006.

[20] Letelier, P., "A Framework for Requirements Traceability in UML-based Projects," presented at the Workshop on Traceability in Emerging Forms of Software Engineering, Edinburgh, U.K., 2002.

[21] MOF-2.0, "Meta Object Facility (MOF) Core Specification, version 2.0," Object Management Group, Inc. (OMG), Needham, MA, USA, Specification formal/06-01-01, January 2006 2006.

[22] Monroe, R. T., Kompanek, A., Melton, R., and Garlan, D., "Architectural Styles, Design Patterns, and Objects," *IEEE Software*, vol. 14, no. 1, pp. 43-52, 1997.

[23] Oldevik, J. and Neple, T., "Traceability in Model to Text Transformations," presented at the 2nd ECMDA Traceability Workshop (ECMDA-TW), Bilbao, Spain, 2006.

[24] Ramesh, B. and Jarke, M., "Toward reference models for requirements traceability," *Software Engineering, IEEE Transactions on*, vol. 27, no. 1, pp. 58-93, 2001.

[25] Rech, J., "Handling of Software Quality Defects in Agile Software Development," in *Agile Software Development Quality Assurance*, I. Stamelos and P. Sfetsos, Eds.: Idea Group Inc., 2007.

[26] Rech, J. and Bunse, C., *Model-Driven Software Development: Integrating Quality Assurance*, 1st Edition ed. Hershey, USA: IGI Global, 978-1-60566-006-6, 2009.

[27] Rech, J. and Schmitt, M., "Embedding Defect and Traceability Information in CIM- and PIM-level Software Models," presented at the International Workshop on Business Support for MDA (MDABIZ), ETH Zurich, Switzerland 2008.

[28] Rech, J. and Spriestersbach, A., "Quality Defects in Model-driven Software Development," Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, Deliverable D4.1, 9. August 2007 2007.

[29] Rech, J. and Spriestersbach, A., "Quality Defects in Model-driven Software Development," Fraunhofer IESE, Kaiserslautern, Project Deliverable VIDE Report No. D4.1, 9. August 2007 2007.

[30] Rech, J., Spriestersbach, A., and Schmitt, M., "QA support methods for VIDE and quality defect discovery tool for VIDE modellers," Fraunhofer IESE, Kaiserslautern Report No. 012.07/E, 13. December 2007 2007.

[31] Schmitt, M., *Analyse von Qualitätsdefekten in Modell-getriebenenen Architekturen (MDA)*, Diploma Thesis. Mannheim, Germany: Hochschule Mannheim, Department of Computer Science, 2007.

[32] Tekinerdogan, B., Hofmann, C., and Aksit, M., "Modeling traceability of concerns in architectural views," presented at the Proceedings of the 10th international workshop on Aspect-oriented modeling, Vancouver, Canada, 2007.

[33] TopCased, "Topcased IDE," http://www.topcased.org/, last accessed on 27 November 2007

[34] UML-2.1.1, "Unified Modeling Language (UML), version 2.1.1," Object Management Group, Inc. (OMG), Needham, MA, USA 2007.

[35] von Knethen, A., *Change-Oriented Requirements Traceability. Support for Evolution of Embedded Systems* PhD Thesis, PhD Thesis. Kaiserslautern: University of Kaiserslautern, Department of Computer Science, 2002.

[36] Walderhaug, S., Johansen, U., Stav, E., and Aagedal, J., "Towards a Generic Solution for Traceability in MDD," presented at the Second ECMDA Traceability Workshop, 2006.

[37] XMI-2.1.1, "MOF 2.0/XMI Mapping, Version 2.1.1," Object Management Group, Inc. (OMG), Needham, MA, USA formal/2007-12-01, December 2007 2007.

[38] XML-1.1, "Extensible Markup Language (XML) 1.1," http://www.w3.org/TR/2006/REC-xml11-20060816/, last accessed on 23. September 2008

[39] XML-Schema, "XML Schema Part 0: Primer," http://www.w3.org/TR/xmlschema-0/, last accessed on 23. September 2008

Jörg Rech is an entrepreneur in the area of Web 2.0 and Web 3.0 with a focus on social semantic networks. He was a senior scientist and project manager at the Fraunhofer Institute for Experimental Software Engineering (IESE), an applied research and transfer institute, in Kaiserslautern, Germany. His research mainly concerns software antipatterns, software patterns, defect discovery, software mining, software retrieval, automatic software reuse, software analysis, and knowledge management, esp., in the area model-driven software engineering. Jörg Rech authored over 50 international journal articles, book chapters, and refereed conference papers, mainly on software engineering and knowledge management. Additionally, he is a member of the German Computer Society (Gesellschaft für Informatik, GI) and served as a PC member for different workshops, work group leader, and conferences as well as an editor for several books in the domain of software engineering and knowledge management. Contact him at joerg.rech@gmail.com.

Mario Schmitt is a junior scientist at the Fraunhofer Institute for Experimental Software Engineering (IESE), an applied research and transfer institute, in Kaiserslautern, Germany. His research mainly concerns context modeling, diagnosis, software model analysis, and knowledge management. Contact him at Mario.Schmitt@iese.fraunhofer.de.