MASTER'S THESIS

Empirical Investigations in
Software Architecture Completeness

by

Christian F. J. Lange

Supervisor: Michel Chaudron
Advisors: Eric Dortmans
          Lou Somers

Venlo, September 2003
The writer was enabled by Océ-Technologies B.V. to perform research that partly forms the basis for this report. Océ-Technologies B.V. does not accept responsibility for the accuracy of the data, opinions and conclusions mentioned in this report, which are fully for the account of the writer.
A. Abstract

Especially in large-scale industrial software projects there is the need for applying quality assuring techniques already in early stages of the development process, i.e. the architecture and design phases. In these stages models are used to analyze and describe the software system. As the models in large-scale projects are created and used by many team members and external stakeholders it is a necessity to maximize comprehensibility.

The UML is the most commonly used notation used in software engineering projects. It is a common hypothesis that the freedom allowed by the UML involves miscommunication and integration overhead and complicates the two afore mentioned needs in software projects. In large projects it is difficult to oversee the completeness of a model under development. For the MSc project described in this thesis we performed a survey through questionnaires. The survey confirmed this hypothesis. The UML is used loosely by practitioners; deadline stress and lack of conventions cause that the architecture models are incomplete and inconsistent, i.e. they lack elements and contain conflicting information between different types of diagram. This leads to uncertainty whether quality inspections are trustworthy at a given point of time, and it leads to miscommunication, integration problems and testing overhead.

In this project completeness and consistency of UML models is defined. A set of rules and metrics is proposed to manage large models (with respect to completeness and consistency).

To specify violations of completeness and consistency we have created a meta model of the four most commonly used diagram types. We extended an existing tool to calculate the rules and metrics automatically given an UML model.

In six large-scale case studies these rules and metrics are validated. Besides validation of the rules and metrics the case studies serve to investigate violations of completeness and consistency in UML models.

We make the following observations:

- The absolute number of completeness and consistency violations is quite large. In industrial practice, the designs are moved to the next stage with still a significant number of violations.
- The types of violations occurring in a model are related to the architect’s habits, education and other factors.
- A heuristic is presented that indicates the internal dynamic behavior of a class. The presented dynamicity is useful for the identification of highly dynamic classes. Interestingly the dynamicity histogram shows similar characteristics for all cases.
- The two case studies performed as a post-graduate student project have the best score. This probably due to the fact that the time pressure for such a small project is less than in industrial settings and that the students learned UML modeling at the university and not on the job, like most industrial architects.
- The presented rules and metrics cannot ultimately identify violations, but point the architect to the suspects. The architect has to judge whether it is a violation. The techniques help finding violations efficiently and encourage model improvements. In practice most suspects turned out to be guilty.
B. Preface

In 1998 I started my studies of computer science at the Technische Universiteit Eindhoven (TUE). From the various courses and projects that were encountered throughout the terms I was always fond of the ones related to modeling. When I returned from my stay abroad at the University of Calgary to attend the last term of my curriculum at the TUE I attended the lectures of Michel Chaudron. We started talking about a graduation project; we were both interested in enhancing quantitative analysis methods for architecture models. The contacts of Lou Somers to Océ Technologies lead to the first discussions with software architects Marten Jansen and Rolf Eisenberg, who proposed the topic of software architecture completeness. Hence the subject for the graduation project was set.

The literature often depicts the analogy between software architecture and building architecture. When someone wants to build a doghouse it is usually sufficient to start with a pile of lumber, some nails and basic tools and even without prior planning the “building” will be satisfactory. For a building with the complexity of a skyscraper it is not that simple, as many stakeholders participate in the construction. Therefore models and blueprints are essential and since the discipline of building architecture has matured over centuries, the techniques for modeling the architecture are nowadays dependable and under control. The discipline of software engineering is still in the fledgling stages. The need for architecture models has been caught and is drawing attention.

The focus of this research is software architecture completeness. To illustrate architecture (in)completeness we use the church “la Sagrada Familia” in Barcelona, Spain, as an example. The cover of this thesis shows a picture of the actual state of the church. The project started in 1882 and is not completed until today; the estimated date of completion is in 2010. Antonio Gaudí, the architect of the Sagrada Familia, wanted to create a monumental cathedral with eighteen towers symbolizing the Apostles, the four Evangelists, and the Virgin Mary and Christ. The latter would stand 170 meters high. The church would consist of five naves and three other parts of the building. The genius architect Gaudí, who dedicated his life to the construction of the church, did not produce detailed blueprints to document the architecture. He produced partial draft models and an explanation of the religious symbolism. In 1926 he died in a car accident, far before the construction was finished and without leaving a “complete architecture”. Now succeeding generations could and had to take part in architecting the unfinished building. This democratic approach might have been in Gaudí’s sense, but lead to a lengthy and unpredictable period of construction.

Such an approach makes a church a myth – but for software projects it is unacceptable, as amongst others lead-time, predictability, dependability, cost and maintainability are key issues. Building architecture has matured but what about software architecture? The goal of this project is to develop techniques to assess software architecture completeness and to get insight to the answer of this question.

I would like to acknowledge with particular gratitude the assistance of my supervisor Michel Chaudron, my Advisor Eric Dortmans and Lou Somers, who played an advisory role in this project. I am grateful to Johan Muskens, the author of the original SAAT tool for taking part in several brainstorming sessions and for his technical assistance. Furthermore I would like to thank Bruce Watson for taking place in my examining board. For useful discussions I would like to thank Chris Delnooz, Rolf
Eisenberg, Gert Florijn, Frits Jacobs, Marten Jansen, Kai Koskimies, Gerard Schouten, Kees Jan Sonnenberg, Laurens Vrijnsen and Jos Warmer. I am very thankful to Waldo Ruiterman, the Photoshop guru who helped designing the cover page and was very helpful with many computer tricks, and Teun Willems, who taught me the XSL transformations for the XMI parser. I am also grateful to all my colleagues who were always helpful and motivating in all situations of this project. Finally I would like to thank my family for supporting me through my entire study and provided me a basis for completing this project.

Christian Lange

Venlo, September 2003
C. Table of Contents

1 Introduction .................................................................................................................. 11
  1.1 Research Question ................................................................................................. 11
  1.2 Related Work ......................................................................................................... 12
  1.3 Outline of Thesis .................................................................................................... 13
2 Preliminaries .................................................................................................................. 15
  2.1 Software Metrics .................................................................................................... 15
    2.1.1 Goals of Metrics ............................................................................................... 15
    2.1.2 Types of Metrics ............................................................................................. 15
    2.1.3 Evaluation of Metrics ...................................................................................... 17
    2.1.4 Mapping and Scales ......................................................................................... 17
  2.2 Survey Results ........................................................................................................ 18
  2.3 Software Architecture ............................................................................................ 19
    2.3.1 Overview of Definitions .................................................................................. 20
    2.3.2 Kruchten’s 4+1 View Model ............................................................................ 20
  2.4 Unified Modeling Language .................................................................................... 22
    2.4.1 Use Case Diagram ........................................................................................... 23
    2.4.2 Class Diagrams ............................................................................................... 23
    2.4.3 Message Sequence Charts .............................................................................. 24
    2.4.4 State Diagrams ............................................................................................... 25
3 Completeness .................................................................................................................. 26
  3.1 Introduction ............................................................................................................ 26
  3.2 Metaphors and Examples ....................................................................................... 28
  3.3 Definition ................................................................................................................ 28
    3.3.1 Dependency on Predominant Diagram ........................................................... 30
  3.4 Meta model .............................................................................................................. 30
    3.4.1 The Relational Meta Model ............................................................................. 31
    3.4.2 Relations ........................................................................................................ 32
  3.5 Rules and Metrics ................................................................................................... 34
    3.5.1 Notational remarks ......................................................................................... 34
  3.6 Basic Size Metrics .................................................................................................. 36
    3.6.1 Number of Elements ...................................................................................... 36
  3.7 Use cases ................................................................................................................ 37
    3.7.1 Size of Use Case ............................................................................................. 37
    3.7.2 Actor needs Use Case ..................................................................................... 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7.3</td>
<td>Scenario needs Use Case</td>
<td>39</td>
</tr>
<tr>
<td>3.8</td>
<td>Class Diagram Metrics</td>
<td>40</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Private Attributes</td>
<td>40</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Data Records</td>
<td>40</td>
</tr>
<tr>
<td>3.8.3</td>
<td>Number of Methods per Class</td>
<td>41</td>
</tr>
<tr>
<td>3.8.4</td>
<td>Methods must be called</td>
<td>42</td>
</tr>
<tr>
<td>3.8.5</td>
<td>Classes must belong to a Use Case</td>
<td>42</td>
</tr>
<tr>
<td>3.9</td>
<td>Class – Scenario Metrics</td>
<td>44</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Objects must have names</td>
<td>44</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Messages must have Names</td>
<td>44</td>
</tr>
<tr>
<td>3.9.3</td>
<td>Objects must have a type</td>
<td>45</td>
</tr>
<tr>
<td>3.9.4</td>
<td>Messages must correspond to Methods</td>
<td>45</td>
</tr>
<tr>
<td>3.9.5</td>
<td>Messages only between related Classes</td>
<td>46</td>
</tr>
<tr>
<td>3.9.6</td>
<td>No abstract Classes in Scenario</td>
<td>46</td>
</tr>
<tr>
<td>3.10</td>
<td>Inheritance Related Metrics</td>
<td>48</td>
</tr>
<tr>
<td>3.10.1</td>
<td>No abstract Leaf Classes</td>
<td>48</td>
</tr>
<tr>
<td>3.10.2</td>
<td>Abstract Classes should have abstract Super class</td>
<td>48</td>
</tr>
<tr>
<td>3.11</td>
<td>State Diagram Metrics</td>
<td>49</td>
</tr>
<tr>
<td>3.11.1</td>
<td>Classes have no more than one State Diagram</td>
<td>49</td>
</tr>
<tr>
<td>3.11.2</td>
<td>Dynamic Classes need a State Diagram</td>
<td>49</td>
</tr>
<tr>
<td>3.11.3</td>
<td>Incoming messages correspond to State Transitions</td>
<td>50</td>
</tr>
<tr>
<td>3.12</td>
<td>Classification of Rules and Metrics</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>Case Studies</td>
<td>55</td>
</tr>
<tr>
<td>4.1</td>
<td>Use Case related Metrics</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Class Diagram related Metrics</td>
<td>57</td>
</tr>
<tr>
<td>4.3</td>
<td>MSC related Metrics</td>
<td>60</td>
</tr>
<tr>
<td>4.4</td>
<td>State Diagrams</td>
<td>61</td>
</tr>
<tr>
<td>4.5</td>
<td>Dynamicity Heuristic</td>
<td>61</td>
</tr>
<tr>
<td>4.6</td>
<td>Evaluation</td>
<td>64</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Observations concerning Case Studies</td>
<td>64</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Observations concerning Rules and Metrics</td>
<td>65</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>Observations</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Conclusions</td>
<td>67</td>
</tr>
<tr>
<td>5.3</td>
<td>Recommendations</td>
<td>68</td>
</tr>
</tbody>
</table>
8 Table of Contents

5.3.1 Theoretical Recommendations ................................................. 68
5.3.2 Technical Recommendations ............................................... 69
6 Bibliography ............................................................................. 70
7 Appendices ............................................................................... 72

7.1 Implementation of the Tool ....................................................... 72
  7.1.1 Environment .................................................................. 72
  7.1.2 Components .................................................................. 73
  7.1.3 Implementation .............................................................. 74

7.2 Weyuker's Properties ................................................................ 75

7.3 Investigation of discovering entity-objects by fan-in/fan-out analysis ...... 77

7.4 Outcomes of the Survey ............................................................ 78
  7.4.1 Introduction .................................................................. 78
  7.4.2 Job Context .................................................................... 79
  7.4.3 Definition of Completeness ............................................. 80
  7.4.4 UML Usage ................................................................... 81
  7.4.5 Current and desired use of metrics ................................... 83
  7.4.6 Completeness and related questions ................................. 85
  7.4.7 Conclusions .................................................................. 87
  7.4.8 Quotes from the Survey .................................................... 87
### D. Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAM</td>
<td>Architecture Tradeoff Analysis Method</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer Aided Software Engineering</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines of Code (classic Software Metric)</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility. Meta language that is used for e.g. the UML meta model.</td>
</tr>
<tr>
<td>MSC</td>
<td>Message Sequence Charts. Synonyms: Sequence Diagrams, Scenarios</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OOP</td>
<td>Object Oriented Programming</td>
</tr>
<tr>
<td>Rose</td>
<td>Rational Rose, CASE tool to develop UML models</td>
</tr>
<tr>
<td>RUP</td>
<td>Rational Unified Process</td>
</tr>
<tr>
<td>SAAM</td>
<td>Software Architecture Analysis Method</td>
</tr>
<tr>
<td>SAAT</td>
<td>Software Architecture Analysis Tool</td>
</tr>
<tr>
<td>SACA</td>
<td>Software Architecture Completeness Analysis</td>
</tr>
<tr>
<td>SQL</td>
<td>Standard Query Language</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>UML-RT</td>
<td>UML Real Time</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSL</td>
<td>Extensible Stylesheet Language</td>
</tr>
<tr>
<td>XSLT</td>
<td>XSL Transformations</td>
</tr>
</tbody>
</table>
1 Introduction

This thesis documents the results obtained during the graduation project with the topic Software Architecture Completeness Analysis. In this chapter we will discuss the central research question of the project, we will present related work to put this project into perspective and we give an outline of the further chapters of this document.

1.1 Research Question

Software systems have gained more and more importance for all fields of society and individuals. This development is expected to grow at the same velocity. But not only the impact of software has increased enormously, also the complexity of software products has grown tremendously; software projects of teams in the magnitude of several hundred man-years are not an exception.

This increasing demand in software products and the growing complexity force software developers two achieve two business goals:

- Delivering high-quality software products
- Reducing process cost and resources.

A first step in the software lifecycle in fulfilling these is goals to make an excellent architecture. The architecture phase is a very early stage in the software development process and its deliverable artifact – the architecture – influences all latter phases and, hence all artifacts of latter phases including the final product. We will define software architecture precisely in the next chapter. Architectural design decisions influence whether the product fulfills the functional requirements, but also whether it gets a good score for the non-functional quality attributes. Wrong design decisions propagate through the entire development process and contribute to process overhead and high cost.

High-quality architecture is a non-trivial topic of high interest to the software engineering community. The most popular architecture evaluation methods are SAAM and ATAM [KAZM99]. These methods and the derived ones facilitate an evaluation with respect to quality attributes such as modifiability, extensibility, exploring costs. Architecture specialists perform these qualitative evaluations in reviews and inspections. A more recent approach is quantitative analysis using metrics. Metrics based evaluations can usually be performed by tools, but still need specialists for the interpretation of the outcomes.

This project’s central idea came up when we discussed metrics based quality analysis with architects. In large projects there is often uncertainty about whether the architecture models have already reached a maturity level such that quality analysis makes sense. The question was broadened to:

"Has the completeness of the model reached a level such that we can proceed to the next phase?"

An architecture model is used mainly for two reasons:

- Analyzing and structuring the problem and finding a solution,
- Communication about the problem and solution amongst various stakeholders.
12 Introduction

The assumption is, that an incomplete model (i.e. a model not containing all requirements and/or design decisions) increases the risk of not delivering the desired product or causing process overhead due to miscommunication or integration problems.

This project aims at developing (quantitative) techniques to detect incompleteness and inconsistencies in UML (architecture) models. The techniques will be implemented in a tool that aims to support architects by identifying suspicious parts of an architectural design. One reason for suspicion is incompleteness and inconsistency.

In the validation phase of this project we will address the following questions in case studies:

- Do the proposed techniques contribute in the assessment of model completeness and consistency?
- What kind of violations of completeness and consistency occur in large-scale industrial models and to what extent do they occur?

1.2 Related Work

The main areas of related work are: software product metrics and automated analysis UML models.

This project considers tool-supported collection of object-oriented metrics. Software metrics are of high interest to software engineering for more than 30 years now. Due to the large acceptance of the object-oriented paradigm more and more dedicated object-oriented metrics appeared. Fetcce [FETC95a, b] gives a theoretical model to analyze and validate object-oriented metrics, grounded on measurement theory.

The work on software product metrics attempts to quantify quality attributes of software design. The work on software product metrics has been oriented at the level of source code. In recent years, some work on metrics has been applied at the level of software design [CK94] [BM96] and software architecture [N+2000]. However, this has mainly been a migration of existing code-level metrics to the level of UML. This plain migration fails to acknowledge that software architecture and design differ from code in that they exhibit incompleteness and inconsistency.

Metrics for incompleteness and inconsistency might be good predictors for the quality of the resulting systems. The research described in this paper is the first attempt known to the author that proposes the use of incompleteness metrics.

Of course, the problem of completeness is not limited to UML-based approaches and. As the UML is the de facto industry standard our focus is on UML-based approaches.

Florijn’s [FLOR03] RevJava is a tool that assists in collecting metrics. It applies so called critcits to Java programs and helps to find design and style improvements. The tool operates on Java bytecode, therefore it is applicable only in the implementation phase of the software lifecycle. For reasons of intuitive understanding the tool offers some elementary graph visualizations.

Emam and Melo [EMME99] studied the predictability of faulty classes based on object-oriented design metrics. The discussed metrics are applicable in an early stage of the development. Emam and Melo constructed a prediction model that showed high accuracy in the validation with a case study.
A predecessor of the MSc project presented in this thesis was the graduation project by Muskens [MUSK02]. That project dealt with the construction of a tool to calculate metrics to help evaluating software architecture. The tool's inputs are XMI representations of UML models. Muskens presents a collection of case studies.

In literature the topic model completeness is not addressed yet. But the closely related topic consistency is addressed in several publications. The area of consistency in UML can broadly be divided into the complete approaches and the partial approaches. The complete approaches aim to define a fully formal semantics for the complete UML notation. Notable initiatives in this direction are the 'precise UML' project [pUML] and the 'Omega' project [OM].

Partial approaches select a subset of the UML notation, typically related to one view such as the process view, and proceed to develop a well-defined meaning for that subset with the aim of automatically identifying inconsistencies in this partial design. Starting point for literature on this line of research is the workshop on consistency problems in UML [UML02].

The partial approaches can be divided into two principle types: a formal approach that is concerned with mapping (parts of) UML designs onto formal methods in order to give a meaning to the UML constructs. The second is a design-oriented approach. Here emphasis is on modelling the UML in order to analyse properties of the design. In this approach UML is modelled using a meta-model or using OCL [WK99]). Subsequently, consistency rules for actual UML designs are defined in terms of the meta-model.

The formal approaches attempt to map partial designs that describe system dynamics such as Sequence Charts and State charts onto operational formalisms such as process algebra's and Petri-Nets. For example, Engels et. al. [ENGE02] have presented consistency checking for UML State chart diagrams by providing a mapping onto CSP. McUmber and Cheng provide a mapping of UML onto Promela [UC01]. An example of a formal approach that formalizes the functional/structural view of UML designs using Z is described by [SF97].

In the design-oriented approach, Hnatkowska et. al. [HZM01] present a number of consistency rules that are defined in terms of the OCL. Also our approach falls in this category. We use a subset of the UML meta-model and formulate consistency rules in terms of that meta-model. As we will show later in the paper, this approach supports the detection of inconsistencies as well as the computation of design metrics and heuristics, while it does not require the more complex machinery of fully formal methods.

### 1.3 Outline of Thesis

Chapter 2 introduces the preliminary knowledge for this project. We introduce software metrics, software architecture and the Unified Modeling Language (UML). Furthermore the main conclusions of the conducted survey amongst software architecture practitioners are presented. These observations motivate further decisions of the project.

Chapter 3 introduces the main concept of this project: completeness. A definition of completeness and the related concept consistency is given. Furthermore the meta model that serves as a foundation of the formal definition of the rules and metrics is
presented. We propose in this chapter a set of rules and metrics to assess model completeness.

Chapter 4 describes the six case studies that are performed in this project and presents the observations concerning validation of the rules and metrics and the observations about rule violations detected in the real-world models.

Chapter 5 contains the conclusions of the project and gives recommendations for future research.

The bibliography is listed in chapter 6 and the appendices contain the description of the tool implementation, Weyuker's properties for the theoretical validation of metrics, an additional investigation of one of the case studies concerning the enhancement of a proposed heuristic and the results of the survey conducted amongst practitioners.
2 Preliminaries

This thesis documents a research project that deals with quantitative assessment of software architecture completeness. In this chapter we give a short introduction in the central topics of the underlying project. This introduction includes software metrics, software architecture, the Unified Modeling Language [UML], our notion of completeness and consistency and an overview of the results obtained by our survey.

2.1 Software Metrics

Software metrics are a subject of interest to the software engineering community since more than 30 years. The interest in metrics by both practitioners and managers is growing more and more.

One of the first software metrics was the LOC [CONT86], the lines of code. This metric is based on source code and is widely used to measure effort and complexity. Its high acceptance is based on its simplicity: it is intuitive to understand and easy to collect automatically. On the other hand, it is often criticized, because there is no agreement on what a line of code is and because it is dependent of the implementation language. The LOC values of one system implemented in assembly and the other implemented in Java or Haskell cannot be compared at all.

2.1.1 Goals of Metrics

During the development of software engineering in the last decades the interest in software metrics has grown rapidly. Both practitioners and managers are mainly interested in using software metrics as support in reaching the two goals in the software lifecycle:

- Process improvement and
- Quality improvement.

In the first case, metrics are used to predict, measure and control resources in the process of developing the software and therefore identify decisions to (hopefully) reduce the development cost.

The latter case focuses on deploying software metrics to analyze quality attributes of software designs and implementations.

Especially when development teams are adopting new technologies (e.g. the paradigm of object-orientation, aspect-oriented programming, new development environments, ...) and there is no prior experience with the new topic, software metrics are widely used to control the process and the product [CHID94].

Corresponding to the main goals of metrics usage, literature makes a distinction between product and process metrics, according to the attribute a metric measure or predicts. Fenton gives even three categorizations: product, process and resource metrics [FENT91].

2.1.2 Types of Metrics

Measurement theory in general categorizes metrics also in another dimension: internal and external metrics. Internal metrics do not depend on the measurement of any other attribute, whereas external metrics measure or predict an attribute with help of other metrics.
Based on the two dimensions described here Fenton constructed a framework to
categorize all software metrics. Table 1 shows this framework.

<table>
<thead>
<tr>
<th>Entities</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Products</strong></td>
<td></td>
</tr>
<tr>
<td>Specifications</td>
<td>Size, reuse, modularity,</td>
</tr>
<tr>
<td></td>
<td>redundancy, functionality,</td>
</tr>
<tr>
<td></td>
<td>syntactic correctness, ...</td>
</tr>
<tr>
<td>Designs</td>
<td>Size, reuse, modularity,</td>
</tr>
<tr>
<td></td>
<td>coupling, cohesiveness,</td>
</tr>
<tr>
<td></td>
<td>inheritance, functionality,</td>
</tr>
<tr>
<td></td>
<td>reliability, ...</td>
</tr>
<tr>
<td>Code</td>
<td>Functionality, algorithmic</td>
</tr>
<tr>
<td></td>
<td>complexity, control-flow</td>
</tr>
<tr>
<td></td>
<td>structuredness, ...</td>
</tr>
<tr>
<td>Test data</td>
<td>Size, coverage level, ...</td>
</tr>
<tr>
<td>Processes</td>
<td>Time, effort, number of</td>
</tr>
<tr>
<td></td>
<td>requirements changes, ...</td>
</tr>
<tr>
<td>Constructing specification</td>
<td>Quality, cost, stability, ...</td>
</tr>
<tr>
<td>Detailed design</td>
<td>Time, effort, number of</td>
</tr>
<tr>
<td></td>
<td>specification faults found, ...</td>
</tr>
<tr>
<td>Testing</td>
<td>Time, effort, number of coding</td>
</tr>
<tr>
<td></td>
<td>faults found, ...</td>
</tr>
<tr>
<td>Resources</td>
<td>Age, price, ...</td>
</tr>
<tr>
<td>Personnel</td>
<td>Productivity, experience,</td>
</tr>
<tr>
<td></td>
<td>intelligence, ...</td>
</tr>
<tr>
<td>Teams</td>
<td>Size, communication level,</td>
</tr>
<tr>
<td></td>
<td>structuredness, ...</td>
</tr>
<tr>
<td>Organizations</td>
<td>Size, ISO Certification, CMM</td>
</tr>
<tr>
<td></td>
<td>level</td>
</tr>
<tr>
<td>Software</td>
<td>Price, size, ...</td>
</tr>
<tr>
<td>Hardware</td>
<td>Price, sped, memory size, ...</td>
</tr>
<tr>
<td>Offices</td>
<td>Size, temperature, light, ...</td>
</tr>
</tbody>
</table>

| Table 1 – Classification of Software measurement activities (measurement can be either assessment or prediction) [FENT91] |

This project mainly deals with product software metrics. The two main groups of product software metrics are source code metrics and design metrics. Source code metrics have a longer history and therefore more research has been performed on this topic. With the propagation of the object-oriented paradigm in the eighties and nineties of the last century the need for object-oriented (design) metrics has rapidly grown. The advantage of design metrics is that they can be applied in a much earlier stage of the software lifecycle. Therefore the software quality can be improved and faults can be avoided by a fraction of the cost that would be caused in a later (implementation) stage.

In the recent years the number of object-oriented design metrics has grown rapidly and the number of papers and articles in that field is even larger.

It is not trivial to choose the appropriate metric for a particular task from the large variety of metrics. There are two possible ways to validate a metric's applicability to measure a specific attribute: by theoretical reasoning and by empirical experiments.
Chidamber and Kemerer [CHID94] summarize the types of criticism on software metrics:

- the lacking of theoretical basis,
- the lacking (or the uncertainty) of desirable measurement properties,
- insufficient generalized or too implementation technology dependent, and
- highly labor-intensive to collect.

Earlier we mentioned the two main goals of software metrics: process and quality improvement. To reach these goals it is indeed a necessity that we know the relationship between internal and external attributes (Table 1). Consider the variety of software quality attributes, it must be determined which metric to choose to measure or predict a particular attribute or property.

It is observed that the large number of software metrics exceeds the set of measurable dimensions by far. Briand et. al. [BRIA2000] draw the conclusion that many metrics are based on the similar hypothesis and have similar measuring goals. Therefore they are in some way redundant.

This project is an attempt to aid in the difficulty of collecting labor-intensive metrics using a tool and to make a classification, which metrics can be deployed for completeness analysis of a software design. Automated and tool-supported metrics collection can speed up the process of design analysis and contribute to a larger acceptance of software metrics.

The Function Points metric is not suitable for tool-supported collection. The metric was proposed by Albrecht [ALBR79] to make managerial resource estimation independent from the implementation environment. There are efforts in automated counting of function points, but user report 400% error count [FP].

2.1.3 Evaluation of Metrics

The properties of software metrics can be evaluated in two ways: in an empirical approach or in a formal approach. This project evaluates the presented metrics empirically in industrial case studies. The most widely accepted way of formal evaluation is by using the nine criteria of Weyuker. This framework is explained in the Appendix.

2.1.4 Mapping and Scales

In fact software metrics are a mapping from the empirical domain, the domain of software objects (of any level of abstraction), to the numerical domain, usually the integer or the real numbers. Fenton and Pfleeger [FENT96] give the following classifications for the target domain of the mapping:

- **Nominal Scale.** The scale distinguishes different classes without allowing an ordering of or associating a magnitude to the classes.

- **Ordinal Scale.** The classes are ordered with respect to the related empirical object’s attribute. The mapping must preserve the ordering. The numbers represent ranking only.

- **Interval Scale.** As in the ordinal scale the order is preserved in the interval scale. The distances between the empirical classes are reflected in the mapped scale, but ratios do not make sense. Therefore addition and subtraction is possible in the interval scale, but multiplication and division is not.
2 Preliminaries

- **Ratio Scale.** This scale has the same properties as the interval scale but in addition it has a zero element, which represents the entity where the measured attribute has the zero value and the measurement starts on the zero element to increase at equal intervals. All arithmetic operations can be (meaningfully) applied on this scale.

- **Absolute Scale.** The mapped value of an entity is simply the count of the attribute in the entity. Therefore there is only one mapping possible. All arithmetic operations can be (meaningfully) applied on this scale.

Assumed that a software metric is using an ordinal scale, the mapping can be seen as a preference relation on the objects of the empirical domain.

### 2.2 Survey Results

The methods and tools delivered during the underlying graduation project are meant as a theoretical basis to support practitioners in the architecture and design phase of (large) software projects.

In the scope of the graduation project a survey amongst practitioners was conducted for the following two reasons:

- For development of methods it was valuable to gain more insight in the way practitioners deal with the UML, how they proceed in the development of their models and which problems are encountered due to completeness and consistency problems of UML models.

- When the first project phase – building the theoretical basis – was finished and the implementation phase was about to start it was desirable to get practitioners' feedback on the hitherto gathered results (definition, sorts of metrics...).

The survey was conducted in the form of a web-based questionnaire. In addition to collecting data from within Océ and therefore reaching a homogenous audience (with respect to company culture, project and application background, tool usage, management, ...) this approach enabled to reach a large and heterogeneous audience within a reasonable amount of time. The URL of the questionnaire was posted on various related Usenet newsgroups (e.g. **comp.object**, **comp.object.measurement** and **comp.software-eng**), distributed within the software department of Océ and forwarded to people with experience and interest in UML modeling.

Experience has shown that too extensive questionnaires do not get a large amount of responses; therefore we decided that the questionnaire should not exceed 20 questions, and should be completed in no more than 10 to 15 minutes. We grouped the 20 questions in four sections:

- **Job context.** To gain information about the job background of the respondent.

- **Definition of completeness.** To gain feedback and opinions on the definition of completeness.

- **UML usage.** To gain information on the way practitioners are using the UML, which tools they use and in which way they develop their models using the UML.
Completeness and related questions. To get feedback on some of the concepts, rules and metrics established so far within the project.

No question asked for personal information, e.g. the email address. The questionnaire was therefore anonymous. By this means, the results are not biased by someone feeling he has to give a certain answer, e.g. saying that he is using the UML strictly conforming to the specification.

Over a period of two month we received a total of 80 responses of the questionnaire, 63 of those responses came from outside the Océ company.

The usage of the Unified Modeling Language (UML) has mainly two essential purposes:

- understanding the problem and representing a solution,
- usage as communication medium.

UML can be regarded as a communication medium in the situations when members of (large) project teams are sharing models or if stakeholders with different views, i.e. interests, are using common models. Clearly this behavior requires a common understanding of the UML. Therefore the UML is defined in the UML specification [UML]. The results of the underlying survey, though, show that in practice the UML is followed only loosely.

Many practitioners stated that they have encountered problems due to incompleteness. A very prevalently reported problem was miscommunication, which might be caused by the fact that the UML is mainly used loosely and that completeness is not yet an understood attribute of architecture evaluation.

Completeness analysis is a new topic in the evaluation of software architectures and practitioners show great interest in this field. The responses demonstrate that consistency is regarded as an integral part of completeness.

In the hitherto existing quantitative support facilities of evaluation during the software lifecycle most analysis activity occurs on implementation, i.e. source code level. Amongst specialists the agreement is large that especially in the early phases (architecture and design) tool supported quantitative evaluation techniques would be very useful.

The outcomes of the survey show that the most widely used views in software architecting are the logical view and the scenario view. Therefore we will narrow down the focus of our research on these views.

Furthermore the relationship of consistency as part of completeness will be elaborated. When developing our rules and metrics set we will keep in mind that the UML is used more or less loosely, therefore we will keep in mind that the metrics set should be easily adjustable to specific needs (context dependent, abstraction dependent, implicit UML “coding standard” dependent) and that practitioners show high acceptance especially for intuitive metrics.

For the entire survey results please refer to the Appendix.

2.3 Software Architecture

Software architecture can be compared to building architecture. Architecture is the first artifact of a product. In building architecture the product is the building to be
Built and in software architecture the software system to be developed. In architecture the various decisions that influence the entire (software) system (and it's development process) are documented. Architecture is a high-level abstraction of a software system that contains information about functional requirements but also about non-functional quality attributes such as performance, reliability or maintainability. It serves as a common communication platform for the various stakeholders, i.e. end-users, programmers, project leaders etc. Therefore the architecture is the groundwork of building a successful software system.

2.3.1 Overview of Definitions
But what exactly is software architecture? In recent years the number of definitions has grown rapidly. We will now present the essentials of some popular and widely accepted definitions found in literature.

Perry and Wolf [PERR92] give a definition saying architecture is a set of design elements of a particular form. They distinguish between processing, data and connecting elements. This distinction is seen in many other approaches and definitions.

Bass, Clements and Kazman [BASS98] define software architecture as the set consisting of components, connectors and constraints. In this definition connectors between components manage the data or control flow of a system and the constraints define the systems' behavior.

According to the Three Amigos – Grady Booch, James Rumbauch and Ivar Jacobson – software architecture is the set of significant decisions with respect to the system's organization, structural and behavioral elements and the composition of the latter two. Additionally in their definition is incorporated the decomposition in subsystems and the architectural style [AMIG99].

Architecture is defined by the IEEE [IEEE] as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

2.3.2 Kruchten's 4+1 View Model
We have mentioned above the fact that various different stakeholders take part in the software development process, e.g. developers, end-users, process managers, etc. The analogy of building architecture also holds for stakeholders. The stakeholders have different needs in the (information offered by an) architecture, e.g. the end-user of the building is mainly interested in a visual model that helps to see that the future building satisfies his functional and esthetical needs. The electrician and the plumber need architecture models, that technically specify the work they have to perform. There are many other examples of stakeholders. In software, the stakeholders are for example the end-user who does not need all information needed by a programmer and vice-versa. In literature and in practice there are many examples where the whole architecture is alleged putatively presented in one large diagram. A situation like this usually leads to an ambiguous accumulation of boxes and lines and therefore mainly contributes to confusion of all stakeholders.

To remedy, Philippe Kruchten of Rational Software proposes his “4+1 View Model” [KRUC95]. Kruchten's definition of architecture is that it is a high level structure addressing the systems major functionality and performance requirements and non-functional requirements or quality attributes. The widely accepted 4+1 view model
proposes five views addressing specific sets of concerns that are of special interest for the different stakeholders.

Figure 1 – “4+1 View Model of Software Architecture” from [KRUC95]

Figure 1 shows Kruchten’s decomposition in the 4+1 views, and the stakeholders and the concerns dealt with in the different views.

The logical view reflects the functional requirements of the end-user, resulting in a domain model. Kruchten suggests an object-oriented decomposition for this view as the system is here described in its key abstractions. The principles of abstraction, encapsulation and inheritance are exploited in the logical view. Additional to the functional analysis this decomposition serves to identify common mechanisms, i.e. patterns.

The process view describes the control of execution in several layers of abstraction. It enables analysis of non-functional properties such as performance, availability of resource and issues like fault tolerance, distribution of processes and system integrity.

In the development view the focus lies on the organization of the actual software development process, e.g. the mapping of source code to directories. The requirements are related to support the system development and the process management. The software system to be developed is decomposed in small chunks. The subsystems are layered hierarchically with strictly defined interfaces to the surrounding layers.

The physical view is primarily applicable in the context of distributed systems. It addresses (again) the non-functional requirements. The system’s elements defined in the logical view, the process view and the development view are mapped on different (hardware) nodes. This mapping is preferably highly flexible and changes should have minimal impact on source code.
The scenario view is in fact redundant with the other four views (therefore “+1”). Its purpose is to demonstrate that the other view’s elements work together seamlessly. The scenarios are instances of more general use cases and therefore, thus in a certain sense they are abstractions of requirements. Scenarios serve as driver to help the architect to identify architectural elements and aid in validating and illustrating the system’s entire architecture. This view is of high interest for the analysis of model completeness.

2.4 Unified Modeling Language

In the previous section we presented the notion of software architecture and its different views. But how does it work in practice? Which modeling techniques exist? Over the past few years various different modeling languages evolved. The most widely accepted language is the Unified Modeling Language [UML]. The UML mainly evolved from Booch’s and Jacobson’s OOSE (Object-oriented Software Engineering) and Rumbaugh’s OMT (Object Modeling Technique), it was initially mainly used for design, but is more and more also used for architectural modeling. (It can even be applied to model non-software systems).

The exact purpose of the UML is visualizing, specifying, constructing and documenting, therefore the UML offers for the different concerns of the various stakeholders’ nine kinds of diagrams:

- Structural diagrams include class diagrams, object diagrams, component diagrams and deployment diagrams
- Behavioral diagrams include use case diagrams, message sequence diagrams, activity diagrams, collaboration diagrams and state diagrams.

Beyond this the UML offers packages, subsystems and models as facilities to manage the entire model.

<table>
<thead>
<tr>
<th>Diagram Type</th>
<th>Logical</th>
<th>Process</th>
<th>Development</th>
<th>Deployment</th>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use-case diagram</td>
<td>✦</td>
<td></td>
<td>✦</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class diagram</td>
<td></td>
<td>✦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object diagram</td>
<td></td>
<td></td>
<td></td>
<td>✦</td>
<td></td>
</tr>
<tr>
<td>State-chart diagram</td>
<td></td>
<td></td>
<td>✦</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence diagram</td>
<td></td>
<td></td>
<td></td>
<td>✦</td>
<td></td>
</tr>
<tr>
<td>Collaboration diagram</td>
<td></td>
<td></td>
<td>✦</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activity diagram</td>
<td></td>
<td></td>
<td></td>
<td>✦</td>
<td></td>
</tr>
<tr>
<td>Component diagram</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✦</td>
</tr>
<tr>
<td>Deployment diagram</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✦</td>
</tr>
</tbody>
</table>

Table 2 – UML Diagrams and the 4+1 Views

All these diagrams are applicable to an architectural view of Kruchten’s “4+1 view model”. Table 2 presents which diagram types are part of which views. The use case diagram, class diagram, message sequence chart and state chart diagram are the most commonly used diagrams and are therefore the only types of diagrams considered in this project. Here we will present these diagrams.

XMI. The Object Management Groups (OMG) specified a standardized interchange format for UML models: the XML Metadata Interchange (XMI) format. XMI is a
vendor independent representation of UML models based on the XML technology. Therefore XMI is applicable for streaming across networks and XMI has the advantage that it is machine processable and human-readable. Most contemporary CASE-tools support XMI import and export. In this project our tool makes use of UML models represented in XMI format.

### 2.4.1 Use Case Diagram

![Use Case Diagram](image)

**Figure 2 – Use Case Diagram**

Figure 2 shows a use case diagram. Use case diagrams describe the system’s behavior from the viewpoint of an external observer; they describe what the system does. The stick-mans are so called actors and represent roles of people or components outside the actual system. The ovals are the use cases; they are closely related to scenarios (in the sense of verbally described usage scenarios). Use case diagrams are helpful in the following areas:

- ! Determining requirements,
- ! Communicating with clients, and
- ! Generating test cases.

In Kruchten’s “4+1 View Model” use cases are part of the scenario view.

### 2.4.2 Class Diagrams

Class diagrams are static and show the structure of a system, they show what interacts, but not what happens during interaction. As Figure 3 shows, class diagrams depict classes, their attributes, operations and relations between classes, such as inheritance, association and dependency. Class diagrams deal with the central concepts of object-orientation: inheritance, encapsulation and abstraction. In the “4+1 View Model” class diagrams are part of the Logical View, they describe the functionality of the system.
2.4.3 Message Sequence Charts

In addition to class diagrams, the dynamic behavior of the system is described using message sequence charts (Figure 4). Throughout literature (and this document), message sequence charts (MSCs) are synonymously named scenarios or sequence diagrams. Message sequence charts describe how objects, i.e. instances of classes, interact. The MSC specifies what messages are sent and when. MSCs are organized across time, the time progresses downwards the diagram. Together with collaboration diagrams, MSCs are in the group of interaction diagrams. Collaboration diagrams and MSCs are semantically equal. We will focus on MSCs, as they are most widely used. In the “4+1 View Model” MSCs is a notation form of the Scenario View, as MSCs are instances of more abstract use cases.
2.4.4 State Diagrams

Figure 5 gives an example of a state diagram. Object's states are dependent on their attributes values. State transitions are triggered by events. In a state diagram the behavior of an object is specified. A state diagram shows all possible states of an object and the state transitions that cause a state change. In Kruchten's "4+1 View Model" state diagrams are part of the Process View, as they describe the behavior of objects, i.e. instances of classes.
3 Completeness

3.1 Introduction

Software systems are growing not only in size, but also in complexity. The same growth is valid for the project teams developing the software. This becomes a burden in managing the software development process. In the interviews and surveys we conducted during this project we have detected that especially in large-scale industrial projects the need arises to get better insight in the model's completeness. The problems reported by practitioners due to uncertainty about model completeness include uncertainty about when to proceed to the next project phase, uncertainty about whether the results of quality analysis methods are (already) trustworthy, miscommunication and integration overhead.

What are the attributes of model completeness and how to get a grip on them? Starting from the two main stakeholders of a software engineering project – the client and the software maker – we are going to decompose the notion of model completeness.

The client's perspective. The client is starting with an idea that is – probably in cooperation with the software maker – transformed into user requirements. Apart from certain restrictions (like cost, and delivery time) the client is interested in a product that exactly meets the user requirements; therefore an acceptance test is usually performed\(^1\).

The software maker's perspective. For the software maker, apart from delivering the product according to the requirements, it is important to develop the software in an (predictable) economical way, i.e. low resource cost, short time-to-market. This enables the software maker to maximize profits.

The success of pursuing the objectives related to both above mentioned perspectives depend on the controllability of the software development process. Being in control of the process means knowing ones progress. The progress of the design process can be measured in terms of model completeness.

---

\(^1\) We are aware of the fact, that during the development process there will probably arise new requirements or requirements will be changed.
Figure 6 depicts an abstract view on software development artifacts, the level of abstraction decreases from top to bottom (project phases are proceeding from top to bottom).

The top layer represents the requirements of the system. The requirements are detailed by the use cases (indicated by the dashed lines). The line between the requirements and the use cases is the upper boundary of our project, as the use cases are part of the UML model. The scenarios (or MSCs) are more detailed instantiations of the use cases (a MSC instantiating a use cases is indicated by a dashed line). The relation between the classes and the MSCs have the meaning that an instantiation of the class occurs as object in the MSC. The internal behavior of classes can be described by state diagrams (this relation is also indicated by dashed lines). The bottom layer is the implementation, the actual source code. This is divided from the middle layer by a horizontal line, as it is not part of the UML model. The dashed lines mean that the implementation classes are detailed model classes. The scope of this project lies on the middle layer, the actual UML model. Within the model the rules and metrics can be calculated by tool support. At the boundaries between model and requirements resp. implementation is a validation only possible by additional human effort.

We relate the two mentioned stakeholder perspectives to our initial question using this Figure 6. According to the client’s perspective model completeness is such that the model corresponds to all requirements and ensures that the product to be delivered complies with them. The tracing of the functional requirements is to a certain extent possible by automated tool-support. For non-functional requirements this is very difficult (if it is possible at all). Validating this conception of completeness is in fact done in the acceptance test, where the (functionality of the) artifact below the lower
horizontal line, i.e. the implementation, matches the requirements (above the upper horizontal line).

The scope of modeling is given in between the two horizontal lines. This is the scope where the conception of completeness is dealt with that relates to the software maker’s perspective. As the UML is the de-facto industry standard for software architecture modeling we focus our research on UML models. In the software maker’s perspective the main concern is the maturity of the model description, i.e. the collection of UML diagrams. The scope of our research is on the conception of completeness dealing with the actual UML model and therefore the software maker’s perspective. As the UML is used as a communication medium (immediately within a project team, but also to “next generations” of developers maintaining or extending a system) the main concern in this view is to ensure that the UML model is composed in such a way that ambiguity, miscommunication and integration effort are minimized. The methods developed in this project are designed to contribute to maximization of certainty about the model’s state of maturity (at any given time) according to this described conception.

3.2 Metaphors and Examples

Besides giving an overview between different process stages and artefact abstraction levels Figure 6 depicts situations of incompleteness in a model. Here we introduce a vocabulary of metaphors to describe various examples of incompleteness.

Leaf. With the term leaf we depict the situation that an architectural element exists in a specific level, but there is no element of a later phase (or more detailed abstraction), which is related to it. In Figure 6 the use case UC3 is an example for this situation. A leaf clearly indicates a dead end and therefore a “forgotten branch” that results in not implemented requirements.

Orphan. An orphan is the opposite of a leaf: an architecture element that occurs at some level of the design and is not related to any higher abstraction level. Of course, on the requirements level, by definition we only have orphans. If we have an orphan (like Class6 in Figure 6) we have a lack of documentation, as it is not clear why the architecture element is created.

3.3 Definition

The underlying model of architecture for our project is the 4+1 view of Philippe Kruchten. The views are not mutually disjoint, but they are overlapping. This is logical, as they eventually describe one and the same model and the first four views are anyway connected by the scenario view. The same applies to the UML: the different diagrams are overlapping and contain common information. This overlap is a possible cause of ambiguity and therefore a critical point in our further analysis.
For a better understanding we break the discussed conception of completeness down to three subconcepts: well-formedness, consistency and inter-diagram completeness (Figure 7).

**Model Completeness**

![Model Completeness Diagram]

Figure 7 - Decomposition of Model Completeness

**Well-formedness.** The basic level of our analysis of design flaws possibly causing the described problems is the separate diagram (of any type). Each diagram must follow certain restrictions, well-formedness rules and conventions to maximize its understandability and usefulness to the different users of the model. Possible violations of this category are classes without names or state diagrams without start and end states.

**Consistency.** Overlapping diagrams contain common information. If two diagrams in the system contain contradictory common information, i.e. they describe contradictory design decisions, and there is an inconsistency between these diagrams. An instance of this condition occurs if in a message sequence chart an object's method is called and the corresponding class in the class diagram does not contain that method. The goal of consistency rules therefore is to help identifying inconsistencies and improving consistency of the model. If an inconsistency between two models remains unresolved, miscommunication and high integration effort are likely to happen. Figure 8 depicts schematically an inconsistency.

**Diagram Completeness.** The elements in the common part of overlapping diagrams are contained in both diagrams. Hence, in a complete UML model each element in the one diagram has a matching counterpart in the other diagram (i.e. there are not orphans and leafs). An (inter-diagram) incompleteness therefore is when there is an element in the overlapping part of two diagrams (or views) in the one diagram without matching counterpart in the other diagram. For example when there is a use case in an use case diagram, but no interaction diagram or classes in a class diagram are related to the use case, the
condition for incompleteness is met, as clearly the model does not contain elements implementing the functionality required by the use case. (The use case therefore is a leaf). Violations of inter-diagram completeness can cause miscommunication and integration overhead. Another possible problem is unimplemented functionality. Figure 9 depicts schematically a diagram incompleteness.

We have decomposed model completeness in the two main concepts tracing requirements and UML modeling. The focus of our research lies on the latter. Figure 7 depicts the decomposition of model completeness and highlights the concepts, where our research is focused.

3.3.1 Dependency on Predominant Diagram

In some cases it depends on the perspective, whether a violation is an inconsistency or an incompleteness. In figure 10 we have a class diagram and a MSC (whose objects are instantiations of the classes in the class diagram). When you consider the class diagram to be the “good” diagram, we have an inconsistency in the MSC, as Class B does not have a Method_z (which is called in the MSC). But when you consider the MSC to be the “correct” diagram, the class diagram is not complete, as Class B does not contain Method_z. Hence, we see, that there are cases where it depends on which diagram is the predominant one whether the violation is an inconsistency or an incompleteness.

![Diagram](image)

Figure 10 - Incompleteness of Inconsistency

3.4 Meta model

For the further analysis and the definition of rules and metrics we need a meta model that describes the model elements and their interrelationships. A proximate candidate is the UML meta model. This model is a well-defined meta model for the UML specified by the Object Management Group [OMG] using the Meta Object Facility (MOF).

The interchange format for the UML is XMI, whose specification is based on the UML meta model specification. The specifications of both, the XMI and the UML meta model are very extensive and cover much more information than is needed for the metrics and rules defined in the section 3.5.

Our analysis focuses on the logical and the scenario view, therefore only a subset of the information defined in the UML meta model used in this project. These considerations convinced us to create a meta model that fits our needs and is not unnecessarily large. In this chapter the meta model will be explained. The meta model
is a relational data model, as the tool to be implemented in this project is based on a relational database and therefore the transition is easiest when the meta model is defined as a relational data model.

3.4.1 The Relational Meta Model

![Diagram of the Relational Meta Model]

Figure 11 – Relational Meta Model, View I

The meta model incorporates the central concepts of the addressed diagram types. Figure 11 depicts the view on the data in the different diagram types. Figure 12 depicts how the relations between the various model elements are incorporated in the meta model. In the use case diagram the use cases are central and have relations to actors and scenarios (message sequence charts). The MSCs consist of messages and
32 Completeness

objects. This diagram type has a relation to the class diagrams, where the classes are obviously the central concept. Classes may be described by state machines.

### 3.4.2 Relations

In the following we give a list of all relations, i.e. database tables, and their fields. The tables reflect the entities in the presented relational meta model.

Unique identifiers are underlined. Square brackets [] describe fields that are possibly containing no value (no value required).

<table>
<thead>
<tr>
<th>Entity</th>
<th>Table definition and description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>( \text{Ac(idac, name, [idp])} )</td>
</tr>
<tr>
<td>Objects</td>
<td>( \text{O(ido, name, [idc])} )</td>
</tr>
<tr>
<td>Classes</td>
<td>( \text{C(idc, name, isAbstract, type, [idp])} )</td>
</tr>
<tr>
<td>Methods</td>
<td>( \text{M(idm, name, isAbstract, visibility)} )</td>
</tr>
<tr>
<td>Attributes</td>
<td>( \text{A(ida, name, isAbstract, visibility)} )</td>
</tr>
<tr>
<td>Messages</td>
<td>( \text{E(idc, ids, name, [method], caller, callee, [predecessor])} )</td>
</tr>
<tr>
<td>Scenarios</td>
<td>( \text{S(ids, name, [idu], [idp])} )</td>
</tr>
<tr>
<td>State machines</td>
<td>( \text{D(idd, name, idc)} )</td>
</tr>
<tr>
<td>Entity</td>
<td>Table definition and description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>idd is a unique identifier for each state machine;</td>
</tr>
<tr>
<td></td>
<td>name is the state machine's name;</td>
</tr>
<tr>
<td></td>
<td>idc is the identifier of the class a state machine belongs to.</td>
</tr>
<tr>
<td>States</td>
<td>T(idd, name, idd)</td>
</tr>
<tr>
<td></td>
<td>Table T contains the states in state machines.</td>
</tr>
<tr>
<td></td>
<td>idd is a unique identifier for each state;</td>
</tr>
<tr>
<td></td>
<td>name is the state's name;</td>
</tr>
<tr>
<td></td>
<td>idc is the identifier of the state machine a state belongs to.</td>
</tr>
<tr>
<td>Use Cases</td>
<td>U(idu, name, [idp])</td>
</tr>
<tr>
<td></td>
<td>Table U contains the use cases.</td>
</tr>
<tr>
<td></td>
<td>idu is a unique identifier for use case;</td>
</tr>
<tr>
<td></td>
<td>name is the use case's name;</td>
</tr>
<tr>
<td></td>
<td>idp is the identifier of the package a use case belongs to.</td>
</tr>
<tr>
<td>State transitions</td>
<td>R(idr, idt1, idt2, name)</td>
</tr>
<tr>
<td></td>
<td>Table R contains the state transitions.</td>
</tr>
<tr>
<td></td>
<td>idr is a unique identifier for each state transition;</td>
</tr>
<tr>
<td></td>
<td>idt1 is the identifier of the source state;</td>
</tr>
<tr>
<td></td>
<td>idt2 is the identifier of the destination state;</td>
</tr>
<tr>
<td></td>
<td>name is the state transition's name.</td>
</tr>
<tr>
<td>Dependencies</td>
<td>Dep(iddep, name, idclient, idsupplier)</td>
</tr>
<tr>
<td></td>
<td>Table Dep contains the dependencies between elements.</td>
</tr>
<tr>
<td></td>
<td>iddep is a unique identifier for each dependency;</td>
</tr>
<tr>
<td></td>
<td>name is the state transition's name.</td>
</tr>
<tr>
<td></td>
<td>idclient is the element consuming the dependency;</td>
</tr>
<tr>
<td></td>
<td>idsupplier is the element supplier.</td>
</tr>
<tr>
<td>Packages</td>
<td>P(idp, name)</td>
</tr>
<tr>
<td></td>
<td>Table P contains the packages.</td>
</tr>
<tr>
<td></td>
<td>idp is a unique identifier for each package;</td>
</tr>
<tr>
<td></td>
<td>name is the package's.</td>
</tr>
<tr>
<td>Class – Attribute</td>
<td>CA(idc, ida, [idci])</td>
</tr>
<tr>
<td></td>
<td>Table CA contains the relation between classes and attributes.</td>
</tr>
<tr>
<td></td>
<td>idc is the identifier of the class containing the attribute identified by ida;</td>
</tr>
<tr>
<td></td>
<td>if the attribute is inherited, idci identifies the super class it is inherited from.</td>
</tr>
<tr>
<td>Class – Method</td>
<td>CM(idc, idm, [idci])</td>
</tr>
<tr>
<td></td>
<td>Table CM contains the relation between classes and methods.</td>
</tr>
<tr>
<td></td>
<td>idc is the identifier of the class containing the method identified by idm; if the method is</td>
</tr>
<tr>
<td></td>
<td>inherited, idci identifies the super class it is inherited from.</td>
</tr>
<tr>
<td>Inheritance</td>
<td>I(idc1, idc2)</td>
</tr>
<tr>
<td></td>
<td>Table I contains the inheritance relations.</td>
</tr>
<tr>
<td></td>
<td>idc1 identifies the superclass;</td>
</tr>
<tr>
<td></td>
<td>idc2 identifies the subclass.</td>
</tr>
<tr>
<td>Association</td>
<td>As(idc2, idc3)</td>
</tr>
<tr>
<td></td>
<td>(Class – class association)</td>
</tr>
<tr>
<td></td>
<td>Table As contains the association relations between classes.</td>
</tr>
<tr>
<td></td>
<td>idc2 and idc3 identify the associated classes.</td>
</tr>
<tr>
<td>Association</td>
<td>AcU(idac, idu)</td>
</tr>
<tr>
<td></td>
<td>(Actor – Use Case)</td>
</tr>
<tr>
<td></td>
<td>Table AcU contains the association relations between actors and use cases.</td>
</tr>
<tr>
<td></td>
<td>idac identifies the actor associated with the use case identified by idu.</td>
</tr>
</tbody>
</table>

Table 3 – Table Definitions of the Relational Meta Model
3.5 Rules and Metrics

This section presents the collection of rules and metrics for the software architecture completeness analysis. The purpose of this collection is to assess the completeness of a given model by identifying possible spots of incompleteness and inconsistencies.

During this project an extensive literature study was performed and many interviews with practitioners and architecture- and UML-specialists were held. The obtained information combined with our own experiences led us to the set of rules and metrics presented in this chapter. In case studies and experiments these rules and metrics were validated.

In this chapter the rules and metrics are grouped in the following categories: basic metrics, use case related, class diagram related, class diagram – scenario related, inheritance related and state diagram related.

For each rule and metric the describing information is presented in a structured way, which we will explain here.

The heading of each rule and metric paragraph contains the name and the identifier (which is the section number).

At the top of each paragraph the type, context and architecture elements are given. For the type there are three possibilities: ‘rule’, ‘metric’ and ‘rule and metric’. A metric is a function from the model to the numeric domain. A rule can also be seen as a metric, specifically as a Boolean metric. An element either adheres to the rule or it violates the rule. We define metrics based on rules such that the metric expresses the count of number of occurrences of rule violations for each rule.

The ‘Context’ field describes the view and diagrams to which the rule applies and the ‘Architecture Elements’ field describes the specific architectural elements that must be available to apply the rule.

The ‘Description’ field contains textual depiction about what the rule or metric measures and what implications the outcomes have for the completeness analysis. In some cases the metric description is illustrated by a figure.

The ‘Category’ field describes, whether the presented rule or metric is of one of the three categories explained in the definition of completeness in section 3.3 (well-formedness, consistency or completeness).

In case the paragraph describes a rule, it is defined under ‘rule definition’ using a combination of relational algebra and predicate logic.

The query, i.e. the expression which evaluates the outcome, is defined in the same notion of relation algebra and predicate logic under the caption ‘Query or Metric Definition’.

Finally the query is given as implemented in SQL syntax under the caption ‘SQL Query’.

3.5.1 Notational remarks

The function class(x) with object $x \in O$ returns the type, i.e. the class of object $x$. In the notation of our underlying meta model we denote this with

$$\text{class } (x) = \{ c \mid c \in C \land \Pi_{ide}(x) = \Pi_{ide}(c) \}.$$
This expression returns the empty set if the object has no class type (we enforce each object to have a class type by rule R3) and a singleton set otherwise. It returns the singleton set, because idc is a unique identifier in C.

In the remainder of this document we will use a simplified notation of the projection operator \( \Pi \) from the relational algebra. Instead of the relational algebra expression \( \Pi_x(r) \), which denotes field \( f \) of relation \( r \) we will use the object-orientation like notation \( r.f \) to denote field \( f \) of relation \( r \).


3.6 Basic Size Metrics

The metrics in this section are metrics that measure the size of a model and its elements.

3.6.1 Number of Elements

<table>
<thead>
<tr>
<th>Type: Metric</th>
<th>Context: Entire model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Elements: N/A</td>
<td>Category: completeness</td>
</tr>
</tbody>
</table>

3.6.1.1 Description

Here we define the simple count queries for all entities that we have in the meta model. These metrics give an indication of the size of the entire model. Furthermore it assigns weights (numbers) to all sorts of architecture elements and therefore allows an overview over which elements are extensively used and which ones are not or hardly used.

Figure 13 - Number of Elements

3.6.1.2 Rule Definition

N/A

3.6.1.3 Query or Metric Definition

Use Cases:

\[
\# u \in U : true
\]

Scenarios:

\[
\# s \in S : true
\]

The definition for the other architecture elements is straightforward and therefore omitted.

3.6.1.4 SQL Query

Use Cases:  

```sql
SELECT COUNT(*) FROM U
```
3.7 Use cases

3.7.1 Size of Use Case

| Type: Metric | Context: Entire model |
| Architecture Elements: Use cases, scenarios, messages, objects | Category: completeness |

3.7.1.1 Description

The use cases are defined in a very early stage (requirements phase) of the architectural design process, thus they form the basis of the entire model. This (set of) metrics determines the size of each use case in terms of amount of scenarios, objects and method calls. This metrics helps the designer to identify underdeveloped use cases, and use cases that contain too much functionality and should better be split into more than one use case.

![Diagram of Use Case Size](Figure 14 - Use Case Size)

3.7.1.2 Rule Definition

N/A

3.7.1.3 Query or Metric Definition

i. Size in terms of scenarios:

\[
\{ (u, n) \mid u \in U \land n \in \text{Integer} \land \\
n = (\# s \in S : s.iud = u.iud) \}
\]

ii. Size in terms of messages:

\[
\{ (u, n) \mid u \in U \land n \in \text{Integer} \land \\
n = (\# e \in E : (\exists s \in S : s.iud = u.iud \land e.ids = s.ids)) \}
\]

iii. Size in terms of objects:
3.7.1.4 SQL Query

i. Size in terms of scenarios:

   SELECT U.NAME, COUNT(S.IDS)
   FROM U,S
   WHERE (S.IDU = U.IDU) GROUP BY U.IDU

ii. Size in terms of messages:

   SELECT U.NAME, COUNT(E.IDE)
   FROM U,S,E
   WHERE (S.IDU = U.IDU) AND (S.IDS = E.IDS)
   GROUP BY U.IDU

iii. Size in terms of objects:

   SELECT U.NAME, COUNT(DISTINCT O.IDO)
   FROM U,S,E,O
   WHERE (S.IDU = U.IDU) AND (S.IDS = E.IDS) AND
     ((E.CALLER = O.IDO) OR (E.CALLEE = O.IDO))
   GROUP BY U.IDU

3.7.2 Actor needs Use Case

<table>
<thead>
<tr>
<th>Type: Rule</th>
<th>Context: Use case diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Elements: Use case, actors</td>
<td>Category: well-formedness</td>
</tr>
</tbody>
</table>

3.7.2.1 Description

Each actor must be related to at least one use case.

![Diagram showing client and salesperson](image)

Figure 15 - Actor needs Use Case

3.7.2.2 Rule Definition

\((\forall \text{ac} \in \text{Ac} : (\exists \text{u} \in \text{U} : (\text{ac.idac, u.idu}) \in \text{AcU}))\)

3.7.2.3 Query or Metric Definition

Here we specify the metric that measures the number of use cases, an actor is related to.
{ (ac, n) | ac ∈ Ac ∧ n ∈ Integer ∧
   n = { # u ∈ Uc : (ac.idac, u.idu) ∈ AcU } }

3.7.2.4 SQL Query

SELECT AC.NAME, COUNT( DISTINCT ASS.IDASS)
FROM AC, ASS
WHERE ((AC.IDAC = ASS.IDCA) OR (AC.IDAC = ASS.IDC2))
GROUP BY AC.IDAC

3.7.3 Scenario needs Use Case

| Type: Rule | Context: Scenario, Use Case |
| Architecture Elements: Scenarios, use cases | Category: completeness |

3.7.3.1 Description

Each scenario should belong to at least one use case.

3.7.3.2 Rule Definition

{ ∀ s ∈ S : (∃ u ∈ U : u.idu = s.idu )}

3.7.3.3 Query or Metric Definition

Here we specify a query that selects all scenarios that do not belong to a use case.

{ s | s ∈ S ∧ ¬ (∃ u ∈ U : u.idu = s.idu )}

3.7.3.4 SQL Query

SELECT S.NAME
FROM U.RIGHT JOIN S ON U.IDU=S.IDU
WHERE U.IDU IS NULL
3.8 Class Diagram Metrics

The metrics and rules in this section address the static structure of class diagrams.

3.8.1 Private Attributes

<table>
<thead>
<tr>
<th>Type: Rule, metric</th>
<th>Context: Class diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Elements: Classes, associations</td>
<td>Category: well-formedness</td>
</tr>
</tbody>
</table>

3.8.1.1 Description

A class’ attributes should be declared private. Otherwise the concept of encapsulation does not hold.

3.8.1.2 Rule Definition

\[
\forall c \in C : (\forall a \in A \land (c.idc, a.ida) \in CA : a.visibility = 'private')
\]

3.8.1.3 Query or Metric Definition

Here we specify the metric that returns for each class the ratio of non-private attributes divided by total number of attributes. This metric helps to identify classes that infringe the concept of encapsulation. In a perfect situation (i.e. the above defined rule holds) the metric is 0 for all classes.

\[
\{ (c, r) \mid c \in C \land r \in \text{Real} \land r = \\
(\# a \in A : (c.idc, a.ida) \in CA \land a.visibility = 'public') \\
/ (\# a \in A : (c.idc, a.ida) \in CA) \}
\]

3.8.1.4 SQL Query

```
SELECT C.NAME, COUNT(A.IDA)
FROM C, A, CA
WHERE C.IDC=CA.IDC AND CA.IDA=A.IDA AND A.VISIBILITY='public'
GROUP BY C.NAME
```

3.8.2 Data Records

<table>
<thead>
<tr>
<th>Type: Rule</th>
<th>Context: Class diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Elements: Classes</td>
<td>Category: well-formedness</td>
</tr>
</tbody>
</table>

3.8.2.1 Description

Classes with only attributes and no methods are data records in disguise and therefore they do not comply with the paradigm of object-orientation.

3.8.2.2 Rule Definition

\[
\forall c \in C : (\exists m \in M : (c.idc, m.idm) \in CM)
\]
3.8.2.3 Query or Metric Definition
Here we specify the query that selects all classes without methods.

\[ \{ c \mid c \in C \land \neg (\exists m \in M : (c.idc, m.idm) \in CM) \}\]

3.8.2.4 SQL Query

SELECT C.NAME, C.TYPE, C.ISABSTRACT
FROM C
LEFT JOIN CM ON C.IDC=CM.IDC
WHERE CM.IDM IS NULL

3.8.3 Number of Methods per Class

<table>
<thead>
<tr>
<th>Type</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Elements: Classes</td>
<td></td>
</tr>
<tr>
<td>Context: Class diagram</td>
<td></td>
</tr>
<tr>
<td>Category: completeness</td>
<td></td>
</tr>
</tbody>
</table>

3.8.3.1 Description

The rule described in 5.2 (Data Records) criticizes classes without methods as data records. That is a very strong discrimination, as incompleteness might also occur in situations where classes are lacking methods and have only a few methods.

Therefore we introduce this metric, which counts the number of methods (in total, inherited and not inherited). The comparison of this metric amongst all classes helps to identify unbalanced components and therefore suspects (with respect to completeness).

3.8.3.2 Rule Definition

N/A

3.8.3.3 Query or Metric Definition

Number of all messages per class

\[ \{ (c, n) \mid c \in C \land n \in \text{Integer} \land n =
    \#m \in M, x : (c.idc, m.idm, x) \in CM \} \]

Number of all non-inherited messages per class

\[ \{ (c, n) \mid c \in C \land n \in \text{Integer} \land n =
    \#m \in M, x : (c.idc, m.idm, x) \in CM \land x = \text{NULL} \} \]

Number of all inherited messages per class

\[ \{ (c, n) \mid c \in C \land n \in \text{Integer} \land n =
    \#m \in M, x : (c.idc, m.idm, x) \in CM \land
    (\exists c_2 \in C : c_2.idc=x) \} \]

3.8.3.4 SQL Query

Number of all messages per class

SELECT C.NAME, COUNT(CM.IDM)
FROM C,CM
WHERE C.IDC=CM.IDC
GROUP BY C.IDC
42 Completeness

Number of all non inherited messages per class

```
SELECT C.NAME, COUNT(CM.IDC)
FROM C, CM
WHERE C.IDC=CM.IDC AND CM.IDCI IS NULL
GROUP BY CM.IDC
```

Number of all inherited messages per class

```
SELECT C.NAME, COUNT(CM.IDC)
FROM C, CM
WHERE C.IDC=CM.IDC AND CM.IDCI IS NOT NULL
GROUP BY CM.IDC
```

3.8.4 Methods must be called

<table>
<thead>
<tr>
<th>Type: Rule</th>
<th>Context: Class diagram, scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Elements: Methods, messages, classes, objects</td>
<td>Category: completeness</td>
</tr>
</tbody>
</table>

3.8.4.1 Description

Each public method of a class must be called in at least one scenario. Otherwise the method is unessential (assuming a complete description in terms of scenarios). A model with a large value for this metric that is not called in scenarios is likely to be underdeveloped in terms of scenarios.

3.8.4.2 Rule Definition

```latex
( \forall m \in M : \\
( \exists e \in E : e.method = m.idm ))
```

3.8.4.3 Query or Metric Definition

Here we specify the query that selects all methods that are not called in any scenario.

```
{ m | m \in M \land \\
\neg( \exists e \in E : e.method = m.idm) }
```

3.8.4.4 SQL Query

```
SELECT M.IDM, M.NAME
FROM M LEFT JOIN E ON M.IDM=E.IDM
WHERE E.NAME IS NULL
```

3.8.5 Classes must belong to a Use Case

<table>
<thead>
<tr>
<th>Type: Rule</th>
<th>Context: Class diagram, scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Elements: Messages, classes, objects</td>
<td>Category: completeness</td>
</tr>
</tbody>
</table>

3.8.5.1 Description

Classes that are not related to any use case come from “nowhere”, i.e. they do not implement functionality that is relevant for any use case or the diagrams do not contain enough information to trace the class to a use case (incompleteness of a diagram). A model with a large number of classes not belonging to any use case, i.e. scenario, is likely to be underdeveloped in terms of use cases, respectively scenarios.

In [3.7.1.4] we formulate the rule saying each scenario must belong to a use case. It is sufficient to state in this rule that each class must be related to a scenario. In Figure 16 the relations are displayed. All classes in the class diagram are instantiated (at least
once) in the scenario (the instantiation relations are depicted by a line). The scenario is an instantiation of a use case (depicted by the line).

![Image of class diagram]

**Figure 16 - Class-Scenario-UC-Relation**

3.8.5.2 Rule Definition

\[
\forall c \in C: (\exists s \in S:
\exists e \in E:
\text{e.ids}=s.ids \land (\text{class(e.caller)}=c \lor \text{class(e.callee)}=c)) > 0)
\]

3.8.5.3 Query or Metric Definition

Here we specify a query that measures the occurrence of each class in messages (of scenarios). If a class does not occur in any message, it is not related to a scenario, i.e. the defined rule does not hold.

\[
\{ c | c \in C \land \neg (\exists e \in E: \text{e.ids}=s.ids \land (\text{class(e.caller)}=c \lor \text{class(e.callee)}=c)) \}
\]

3.8.5.4 SQL Query

```
SELECT C.NAME, C.ISABSTRACT, C.TYPE
FROM O RIGHT JOIN C ON O.IDC=C.IDC
WHERE O.IDO IS NULL
```
3.9 Class – Scenario Metrics

The metrics and rules in this section address the interaction in scenarios of elements in class diagrams

3.9.1 Objects must have names

| Type: Rule | Context: Class diagram, scenario |
| Architecture Elements: Objects | Category: completeness, well-formedness |

3.9.1.1 Description

Each object in a scenario must have a name. A named object is much more expressive and understandable than an unnamed one. Especially in the case that several objects of the same type occur in one scenario it is essential to name objects.

3.9.1.2 Rule Definition

\( \forall o \in O : o.name \neq \bot \)

3.9.1.3 Query or Metric Definition

Here we specify the query that returns a set of all objects that do not conform with the defined rule, i.e. the objects without a class type specified.

\( \{ o \mid o \in O \land o.name = \bot \} \)

3.9.1.4 SQL Query

```
SELECT S.NAME, O1.IDO, O1.NAME
FROM E, S, O
WHERE S.IDS=E.IDS AND (E.CALLER=O1.IDO OR E.CALLEE=O1.IDO)
AND LENGTH(O1.NAME)=0
```

3.9.2 Messages must have Names

| Type: Rule | Context: Scenario |
| Architecture Elements: Messages, methods | Category: completeness |

3.9.2.1 Description

Each message in a scenario must have a name.

3.9.2.2 Rule Definition

\( \forall e \in E : (\exists m \in M : e.name \neq \bot) \)

3.9.2.3 Query or Metric Definition

Here we specify the queries that return a set of all messages that do not conform with the defined rules, i.e. the without a name..

\( \{ e \mid e \in E \land \neg (\exists m \in M : e.name \neq \bot) \} \)

3.9.2.4 SQL Query

```
SELECT S.NAME, O1.NAME, O2.NAME, E.IDE
FROM S, E, O O1, O O2
WHERE E.IDS=S.IDS AND LENGTH(E.NAME)=0
AND E.CALLER=O1.IDO AND E.CALLEE=O2.IDO
```
3.9.3 Objects must have a type

| Type: Rule | Context: Class diagram, scenario |
| Architecture Elements: Objects, classes | Category: completeness, consistency |

3.9.3.1 Description
Each object in a scenario must have a type, i.e. a class that occurs in a class diagram of the described model.

3.9.3.2 Rule Definition
\[( \forall o \in O : ( \exists c \in C : \text{class}(o) = \{c\} ))\]

3.9.3.3 Query or Metric Definition
Here we specify the query that returns a set of all objects that do not conform with the defined rule, i.e. the objects without a class type specified.

\[\{ o \mid o \in O \land \neg ( \exists c \in C : \text{class}(o) = \{c\} ) \}\]

3.9.4 SQL Query
SELECT O.NAME, O.IDO
FROM O, C
WHERE [O.IDC=C.IDC AND C.NAME='<DummyClass>']
OR (LENGTH(O.IDC)=0)

3.9.4 Messages must correspond to Methods

| Type: Rule | Context: Class diagram, scenario |
| Architecture Elements: Objects, messages, classes, methods | Category: consistency |

3.9.4.1 Description
Each message received by an object in a scenario must correspond to a method in the object’s class interface. Note that 3.9.2 is implied by 3.9.4. (Illustrated in Figure 17)

![Diagram](image)

Figure 17 - Messages must correspond to Methods

3.9.4.2 Rule Definition
\[( \forall e \in E : ( \exists m \in M, : e.name = m.name \land (\text{class}(e.callee).idc, m.idm) \in CM))\]

3.9.4.3 Query or Metric Definition
Here we specify the query that returns a set of all messages that do not conform with the defined rule, i.e. their name is not a method of the callee.

\[\{ e \mid e \in E \land \neg (\exists m \in M, : e.name = m.name \land (\text{class}(e.callee).idc, m.idm) \in CM) \}\]
3.9.4.4 SQL Query

SELECT DISTINCT(E.IDE), E.NAME, E.CALLER, E.CALLEE, S.NAME
FROM E, C, CM, M, S
WHERE E.CALLEE=E.ID AND O.IDC=CM.IDC AND CM.IDM=M.IDM AND
(INSTR(M.NAME,E.NAME)>0) AND S.IDS=E.IDS

3.9.5 Messages only between related Classes

| Type: Rule | Context: Class diagram, scenario |
| Architecture Elements: Messages, classes, objects | Category: consistency |

3.9.5.1 Description

(Objects of) classes exchanging messages in a scenario must be related in a class diagram by means of an association.

This rule corresponds to the Law of Demeter [LIEB89, LIEB96], which comprehends “only talk to your immediate friends”. Many architects feel this rule is too strict. Note that if this rule does not hold, classes “talk to their friend’s friends”. In this situation the two communicating classes must be connected via (transitive) relations. We assume that this holds in every model and do not suggest any rules to check this (as a not connected graph would mean that unrelated systems would be described by one and the same model).

3.9.5.2 Rule Definition

\[ \forall e \in E : (\text{class}(e.caller), \text{class}(e.callee)) \in As \land e.caller = e.callee \]

3.9.5.3 Query or Metric Definition

Here we specify the query that returns a set of all messages that do not conform the defined rule, i.e. the messages between objects of classes that are not related in the class diagram by means of an association. It is sufficient to return a message as this contains all necessary data (i.e. involved objects and scenario).

\[ \{ e | e \in E \land (\text{class}(e.caller), \text{class}(e.callee)) \notin As \land e.caller \neq e.callee \} \]

3.9.5.4 SQL Query

SELECT S.NAME, E.NAME, E.IDE, OCALLER.NAME, OCALLER.IDNAMET, OCALLER.IDC
FROM S, E, O OCALLER LEFT JOIN ASS ON OCALLER.IDC=ASS.IDC
WHERE OCALLER.IDC=ASS.IDC2 AND OCALLER.IDC=ASS.IDC3
WHERE E.CALLER=OCLASS.IDS AND IDASS IS NULL AND
E.CALLEE=OCLASS.IDS AND NOT (E.CALLEE=E.CALLER)
AND S.IDS = E.IDS

3.9.6 No abstract Classes in Scenario

| Type: Rule | Context: Class diagram, scenario |
| Architecture Elements: Objects, classes | Category: well-formedness, consistency |

3.9.6.1 Description

In a scenario no instances of an abstract class are allowed to occur.

3.9.6.2 Rule Definition

\[ \forall o \in O : \text{class}(o).isAbstract = "false" \]
3.9.6.3 Query or Metric Definition
Here we specify the query that returns a set of all objects that do not conform the
defined rule, i.e. the objects that instantiate abstract classes.

\{ \circ \mid \circ \in \mathcal{O} \land \text{class}(\circ).\text{isAbstract} \neq \text{"false"} \}\n
3.9.6.4 SQL Query
SELECT S.NAME, C.NAME
FROM S, E, O, C
WHERE C.ISABSTRACT='true' AND S.IDS=E.IDS AND ((E.CALLER=O.IDO
AND C.IDO=O.IDC) OR (E.CALLEE=O.IDO AND C.IDC=O.IDC))
3.10 Inheritance Related Metrics

The metrics and rules in this section address the concerns related to inheritance.

3.10.1 No abstract Leaf Classes

<table>
<thead>
<tr>
<th>Type: Rule</th>
<th>Context: Class diagram (inheritance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Elements: Classes</td>
<td>Category: completeness, well-formedness</td>
</tr>
</tbody>
</table>

3.10.1.1 Description

Classes that are declared abstract should have subclasses (that inherit from the abstract class).

Note that this rule does not hold when the model describes a framework or a library where the dedicated purpose of some classes is to act as a super class for extensions.

3.10.1.2 Rule Definition

\[
\{ \forall c \in C \land c.isAbstract : \{ \exists c_2 \in C : (c.idc, c_2) \in I \} \}
\]

3.10.1.3 Query or Metric Definition

Here we specify the query that selects all abstract classes that do not have subclasses.

\[
\{ c | c \in C \land c.isAbstract \land \neg (\exists c_2 \in C : (c.idc, c_2) \in I) \}
\]

3.10.1.4 SQL Query

```sql
SELECT C.NAME
FROM C
WHERE ISABSTRACT='TRUE' AND ISLEAF='TRUE' AND TYPE = 'CLASS'
```

3.10.2 Abstract Classes should have abstract Super class

<table>
<thead>
<tr>
<th>Type: Rule</th>
<th>Context: Class diagram (inheritance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Elements: Classes</td>
<td>Category: well-formedness</td>
</tr>
</tbody>
</table>

3.10.2.1 Description

An abstract class that is a subclass of a non-abstract class reflects poor design and a conflict in the model’s inheritance hierarchy.

3.10.2.2 Rule Definition

\[
\{ \forall c \in C \land c.isAbstract :
\neg(\exists d \in C : (c.idc, d.idc) \in I) \land d.isAbstract \}
\]

3.10.2.3 Query or Metric Definition

Here we specify the query that selects all abstract classes that are subclasses of non-abstract classes.

\[
\{ c | c \in C \land c.isAbstract \land 
\exists d \in C : (c.idc, d.idc) \in I \land d.isAbstract \}
\]

3.10.2.4 SQL Query

```sql
SELECT SUB.NAME, SUPER.NAME
FROM C SUB, C SUPER, I
WHERE SUB.ISABSTRACT='TRUE' AND SUPER.ISABSTRACT='FALSE' AND
SUB.IDC=I.IDC1 AND SUPER.IDC=I.IDC2
```
3.11 State Diagram Metrics

The metrics and rules in this section address the state diagrams.

3.11.1 Classes have no more than one State Diagram

<table>
<thead>
<tr>
<th>Type: Rule, metric</th>
<th>Architecture Elements: Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context: Class diagram, state diagrams</td>
<td>Category: well-formedness</td>
</tr>
</tbody>
</table>

3.11.1.1 Description

Each class should be described by at most one state machine.

3.11.1.2 Rule Definition

\[ ( \forall c \in C : \exists d \in D : d.idc = c.idc ) \]

3.11.1.3 Query or Metric Definition

Here we specify the query that selects all classes that have more than one state machine in the model.

\[ \{ c | c \in C \land \exists l < ( \# d \in D : d.idc = c.idc ) \} \]

3.11.1.4 SQL Query

```
SELECT C.NAME, COUNT(D.IDD)
FROM C,D WHERE C.IDC=D.IDC
GROUP BY C.NAME HAVING COUNT(D.IDD)>1
```

3.11.2 Dynamic Classes need a State Diagram

<table>
<thead>
<tr>
<th>Type: Rule, metric</th>
<th>Architecture Elements: Classes, state diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context: Scenario, class diagram, state machines</td>
<td>Category: completeness</td>
</tr>
</tbody>
</table>

3.11.2.1 Description

Classes with a high dynamicity usually have a complex state structure; therefore a state machine should describe highly dynamic classes. Assuming, that events (receiving or sending messages) correspond to state changes in the object, we define dynamicity as the number of messages (in all scenarios) in which a class is involved either as caller or as callee. The expressiveness of the dynamicity metric is dependent from the degree of the models dynamic description by message sequence charts.
3.11.2.2 Rule Definition

\[
( \forall c \in \text{Dyn}(X) : ( \exists d \in D : d.idc = c.idc )
\]

3.11.2.3 Query or Metric Definition

Here we specify the set \( \text{Dyn}(X) \) that denotes the set of all classes with dynamicity greater or equal than \( X \). In an implementation the number \( X \) can be chosen for example as a particular integer value, or context dependent as the average dynamicity over all classes. Note that \( \text{Dyn}(X) \subseteq \text{Dyn}(0) = C \)

\[
\text{Dyn}(X) = \{ c \mid c \in C \land X \leq \\
( \# e \in E : ( \exists o \in O \land c.idc = o.idc : o.ido = \\
e.caller)) + \\
( \# e \in E : ( \exists o \in O \land c.idc = o.idc : o.ido = \\
e.callee) )
\}
\]

3.11.2.4 SQL Query

```sql
SELECT C.NAME, COUNT(E.IDS) 
FROM C LEFT JOIN D ON D.IDC=C.IDC, E, O 
WHERE (E.CALLER=O.IDO OR E.CALLEE=O.IDO) AND O.IDC=C.IDC 
AND D.IDD IS NULL 
GROUP BY C.NAME HAVING COUNT(E.IDS)>X
```

3.11.3 Incoming messages correspond to State Transitions

<table>
<thead>
<tr>
<th>Type: Rule</th>
<th>Context: Scenario, state machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Elements: Classes, state machines</td>
<td>Category: consistency</td>
</tr>
</tbody>
</table>

3.11.3.1 Description

In a scenario an ingoing arrow, i.e. a called method, corresponds to a state transition in the object’s state diagram (if applicable).
3.11.3.2 Rule Definition

\( \forall r \in R : \)
\( ( \exists t \in T, d \in D, m_2 \in M \land r.idt_2=t.idt \land t.idd=d.idd : \)
\( (d.idc, m_2.idm) \in CM \land \)
\( (\exists m \in M: m.idm= m_2.idm \land m.name=r.name) ) ) \)

3.11.3.3 Query or Metric Definition
Here we specify the query that selects all state transitions that are not conform with the rule. The state transition contains all necessary information to trace the source and destination state, the state diagram and the class.

\( \{ r \mid r \in R \land \neg (\exists t \in T, d \in D, m_2 \in M \land r.idt_2=t.idt \land t.idd=d.idd : \)
\( (d.idc, m_2.idm) \in CM \land (\exists m \in M: m.idm= m_2.idm \land m.name=r.name) ) \) \}

3.11.3.4 SQL Query

SELECT *
FROM R LEFT JOIN T ON R.IDT2=T.IDT LEFT JOIN D ON T.IDD=D.IDD
LEFT JOIN CM ON D.IDC=CM.IDC LEFT JOIN M ON CM.IDM=M.IDM
WHERE R.NAME=M.NAME
### 3.12 Classification of Rules and Metrics

In the previous sections we have presented a collection of rules and metrics. In our studies we encountered the fact that not all of these rules and metrics are applicable at the same time in all situations. We tracked two reasons for this: dependencies between rules and different requirements (concerning the rules).

<table>
<thead>
<tr>
<th>Dependency consumers</th>
<th>Dependency suppliers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.1.1 # of use cases</td>
</tr>
<tr>
<td>i. in Scenarios</td>
<td></td>
</tr>
<tr>
<td>ii. in Messages</td>
<td></td>
</tr>
<tr>
<td>iii. in Objects</td>
<td></td>
</tr>
<tr>
<td>4.2 Actor needs use case</td>
<td></td>
</tr>
<tr>
<td>4.3 Scenario needs use case</td>
<td></td>
</tr>
<tr>
<td>5.1 Public attributes</td>
<td></td>
</tr>
<tr>
<td>5.2 Data records</td>
<td></td>
</tr>
<tr>
<td>5.3 Methods per class</td>
<td></td>
</tr>
<tr>
<td>i. total</td>
<td></td>
</tr>
<tr>
<td>ii. inherited</td>
<td></td>
</tr>
<tr>
<td>iii. not inherited</td>
<td></td>
</tr>
<tr>
<td>5.4 Methods must be called</td>
<td></td>
</tr>
<tr>
<td>5.5 Class not in scenario</td>
<td></td>
</tr>
<tr>
<td>6.1 Object needs name</td>
<td></td>
</tr>
<tr>
<td>6.2 Message needs</td>
<td></td>
</tr>
<tr>
<td>i. name</td>
<td></td>
</tr>
<tr>
<td>ii. method</td>
<td></td>
</tr>
<tr>
<td>6.3 Object must have type</td>
<td></td>
</tr>
<tr>
<td>6.4 Message must correspond to method</td>
<td></td>
</tr>
<tr>
<td>6.5 Messages between related classes</td>
<td></td>
</tr>
<tr>
<td>6.6 Scenario contains abstract class</td>
<td></td>
</tr>
<tr>
<td>7.1 No abstract leaves</td>
<td></td>
</tr>
<tr>
<td>7.2 Abstract cl. – non-abstr. sup.cl.</td>
<td></td>
</tr>
<tr>
<td>8.1 at most 1 state machine</td>
<td></td>
</tr>
<tr>
<td>8.2 Dynamicity</td>
<td></td>
</tr>
<tr>
<td>i. general</td>
<td></td>
</tr>
<tr>
<td>ii. high dyn. needs state mach.</td>
<td></td>
</tr>
<tr>
<td>8.3 Incoming messages must correspond to state trans.</td>
<td></td>
</tr>
</tbody>
</table>

*Table 4 - Dependencies*
In table 4 the dependencies between the rules and metrics are presented. The diamonds indicate that the metric in the row depends on the metric in the column, e.g. 4.2 depends on 3.1.1. We have observed, that a specific outcome of one rule or metric sometimes implies a specific outcome of other rules or metrics or makes them not applicable. We give a simple dependency example: Rule 7.2 (Abstract classes must have abstract superclasses, if any) is only applicable if the underlying model data contains classes and inheritance relations. Therefore the metrics 3.1.5 (number of classes) and 3.1.13 (number of inheritance relations) must be greater than 0. This example shows how specific results of a rule or metric (3.1.5, 3.1.13) imply that the consideration of another rule (7.2) is senseless. We can conclude that 7.2 is dependent from 3.1.4 (and 7.2 is dependent from 3.1.13, of course). The dependency relation is transitive. In table 4 the rules and metrics in the rows are dependent of the rules and metrics in the columns if the cell in the specific row and column is marked by a diamonds.

In addition to dependencies, which influence the applicability of rules and metrics, we investigated the applicability of rules and metrics for different abstraction levels of models. In the software lifecycle there are different phases with different requirements concerning the rules and metrics to apply. We have defined four abstraction levels as follows:

1. very high abstraction in the (requirements) analysis phase
2. high abstraction in the architecture phase
3. more detailed in the design phase
4. high detail in the late design phase, usage of the models for seamless transition to the implementation phase.

We broke down the entire set of rules and metrics into four subsets that are applicable in the four described abstraction levels; these subsets are presented in table 5. A diamond indicates that the a metric in a row is applicable in the phase in the column. We concede that the breakdown in four distinct subsets is in practice not as ‘black-white’ as described here. For specific project phases (i.e. abstraction levels of models) and other influencing factors (such as application area, culture of UML usage in organization...) the set of metrics can and should be adjusted to the analyst’s needs.
<table>
<thead>
<tr>
<th>3.1.1 Number of use cases</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.2 Number of actors</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.3 Number of packages</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.4 Number of objects</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.5 Number of classes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.6 Number of methods</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.7 Number of attributes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.8 Number of messages</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.9 Number of scenarios</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.10 Number of state machines</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.11 Number of states</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.12 Number of state transitions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.13 Number of inheritance relations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.14 Number of class associations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1 Use Case Size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i. in Scenarios</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ii. in Messages</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iii. in Objects</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2 Actor needs use case</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3 Scenario needs use case</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.1 Private attributes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2 Data records</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3 Methods per class</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i. total number</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ii. only inherited</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iii. only not inherited</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.4 Methods must be called</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.5 Classes not in scenario</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.1 Objects need name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.2 Message needs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i. name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ii. method</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.3 Objects must have type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.4 Messages correspond methods</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.5 Messages only between related classes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.6 Scenarios don’t have abstract classes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.1 No abstract leaves</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.2 Abstract classes have abstract superclasses (if any)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.1 Classes have at most one state machine</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.2 Dynamicity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i. dynamicity of each class</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ii. highly dynamic classes need a state machine</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.3 Incoming messages must correspond to state transitions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5 - Classification of Rules and Metrics
4 Case Studies

During the project we have defined the notion of completeness and developed techniques to assess a model’s completeness. The developed techniques are rules and metrics such that tool supported collection of the results is possible. The implementation of the tool used in our project is outlined in the appendix.

In order to investigate the applicability of our developed techniques in real-world models we have performed a series of case studies. The objective of our case studies is to get insight into the following questions:

1. How strong is the agreement of architecture specialists with the results?
2. Are there common characteristics in the results of different (related or unrelated) models?
3. Are the characteristics bound to a specific domain or attribute of the development process?

We have investigated designs from three different origins and domains. Here we will briefly introduce the designs provided by the suppliers, then we will interpret and compare the outcomes of our completeness analysis.

Model A1 and A2. These are two models from large-scale industrial systems. A1 and A2 describe two different packages of the same system (same version). The models are design models of an image processing system. The entire project size is in the magnitude of several hundred man-years.

Model B1 and B2. The models are provided from another industrial project from a different application domain. These models are high-level analysis models of a large scale, industrial embedded controller. Both models describe the same system in two subsequent versions, B1 is the initial version and B2 is its successor. Actually B2 is not an evolution or enhancement of B1 but is a new UML model from scratch (of the same system). The entire project size is in the magnitude of several hundred man-years.

Model C1 and C2. As we are interested in investigating generic characteristics, we incorporate these models into our case studies. They are supplied by a group of postgraduate M.Sc. students in advanced software engineering. The models are the design models of the group’s final project, whose stakeholder is an industrial organization. C1 is an intermediate version and C2 is the final version. Interesting is the fact, that the changes in C2 were made after the suppliers were provided with the analysis results of C1. The entire project size is 1.5 man-years.

Note that the models we studied contain fundamental design decisions and are part of the corporate intellectual property. Therefore we cannot disclose any further details on the models here.

Table 6 gives the basic size metrics (refer to section 3.6) of the analyzed models to give an impression of their dimension.
The basic size metrics show that we are analyzing relatively large models, especially in the cases A1 and A2 with 142 and 168 classes. The cases B1 and B2 do not have methods or attributes. This is indeed an incompleteness, but as the cases are analysis models, the focus is on defining the entities. The techniques applying to methods and attributes can therefore not be applied. All cases contain the use case view, class diagram and message sequence charts. C2 is the only model that contains state chart diagrams.

In the following we will summarize the results of the performed case studies. The results of the techniques presented in chapter 3 are presented and compared. Additionally we give our comments and the comments of the consulted experts. Our focus lies on the remarkable observations (that are shaded in the tables). The
identifiers in front of the metric names denote the section number, where the metric is explained (e.g. “ID3.7.1 Size of Use Cases”).

### 4.1 Use Case related Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>A1</th>
<th>A2</th>
<th>B1</th>
<th>B2</th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Use Cases</td>
<td>0</td>
<td>21</td>
<td>32</td>
<td>49</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>ID3.7.1 Size of Use Cases ²</td>
<td>N/A</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ID3.7.2 Actors without Use Cases</td>
<td>1</td>
<td>7</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ID3.7.3 Scenarios without Use Cases</td>
<td>30</td>
<td>65</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7 – Use Case related metrics

**ID3.7.1 Size of Use Cases (in terms of objects).** This metric is not applicable in three cases due to the fact that no relation between scenarios (MSCs) and use cases exists in the model. This fact obviously is an indication for a gap between the use case view and the scenario view at least for traceability. In C1 one use case was identified to be extremely large compared with all other use cases. The architects admitted that there was an unnecessary unbalance and shifted functionality between the use cases in C2.

**ID3.7.2 Actors without Use Cases.** In the most cases this metric is low. The identified actors are usually supertypes of other vectors. In case B1 was identified, that the use case view carries a burden of (unrelated) elements from outside the actual system. This shortcoming was corrected in B2.

**ID3.7.3 Scenarios without Use Cases.** Scenarios (or MSCs) should relate to a use case as they are instantiations of use cases. Three groups of results can be identified: A1 and A2 (from the same supplier) have no scenario at all related with a use case, this remedy is, again, an indication for a ‘missing link’ in the model. B1, C1 and C2 have the perfect result, as all scenarios belong to use cases. An interesting observation was made in B2, where the architect said he would not be deploying scenarios, but 7 scenarios were found, all of them not related to the 49 use cases. This unbalance (7 vs. 49) identified this malformation. It was discovered that the model contained a niche for notes, which actually should not be part of the model at all.

### 4.2 Class Diagram related Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>A1</th>
<th>A2</th>
<th>B1</th>
<th>B2</th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID3.8.1 Non-Private Attributes</td>
<td>67.23%</td>
<td>5.08%</td>
<td>N/A</td>
<td>N/A</td>
<td>0.00%</td>
<td>2.44%</td>
</tr>
<tr>
<td>ID3.8.2 Classes without Methods</td>
<td>45.77%</td>
<td>51.19%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>20.00%</td>
<td>23.91%</td>
</tr>
<tr>
<td>ID3.8.2 Interfaces without Methods</td>
<td>8.82%</td>
<td>9.38%</td>
<td>N/A</td>
<td>100.00%</td>
<td>60.00%</td>
<td>61.29%</td>
</tr>
<tr>
<td>ID3.8.3 Size of Class in # of Methods ³</td>
<td>3.52%</td>
<td>1.19%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>2.22%</td>
<td>2.17%</td>
</tr>
<tr>
<td>ID3.8.4 Methods not called in MSC</td>
<td>67.65%</td>
<td>77.59%</td>
<td>N/A</td>
<td>N/A</td>
<td>40.14%</td>
<td>55.00%</td>
</tr>
<tr>
<td>ID3.8.5 Classes not called in MSC</td>
<td>46.48%</td>
<td>59.52%</td>
<td>35.29%</td>
<td>84.06%</td>
<td>42.22%</td>
<td>43.48%</td>
</tr>
<tr>
<td>ID3.8.5 Interfaces not called in MSC</td>
<td>100.00%</td>
<td>87.50%</td>
<td>N/A</td>
<td>100.00%</td>
<td>70.00%</td>
<td>70.97%</td>
</tr>
<tr>
<td>ID3.10.1 Abstract Leaf Classes</td>
<td>3.52%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>ID3.10.2 Abstract Classes having abstract Super Classes</td>
<td>2.82%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

² The number of Use Cases identified to be outliers. Outliers are defined as the values outside the range of the sample’s mean +/- 2 times the standard deviation.

³ This is the percentage of the classes that were identified to be outliers. Outliers are defined as the values outside the range of the sample’s mean +/- 2 times the standard deviation.
In contrast to the previous table, this table gives the percentages of elements violating a specific rule (i.e. the percentage of the number of violations relative to the total number of elements of a specific type.)

**ID3.8.1 Non-Private Attributes.** Attributes should be declared private (data hiding), so the number of non-private attributes should be low. In the most cases, this rule is adhered to well and the data of classes is hidden. Except for A1. Obviously the particular designer did not follow this rule.

**ID3.8.2 Classes/Interfaces without Methods.** Class and interface interaction is primarily based on methods; therefore it is desirable that all classes and interfaces have methods (aiming at a low percentage). The cases C1 and C2 have a relatively good score for the classes; the interfaces are worse, but they are probably mainly declared to define the interface entities. It is interesting that the pairs in each case show analogue results, especially in the cases A1 and A2 that describe different systems but come from the same organization. B1 and B2 are analysis models; they do not define any methods (nor attributes). For models of high abstraction level it is acceptable not to define methods.

**ID3.8.3 Size of Class in Methods.** No significant amounts of classes that exhibit very unbalanced sizes (relative to the entire system) were identified. The few identified classes were central and important. The designer therefore intended the size.

**ID3.8.4 Methods not called in MSC.** The underlying idea is that classes that are not called in an MSC are either redundant, or the model lacks their description in MSCs. The aim is a low value such that many methods are described in message sequence charts. The student project (C1, C2) has the better – but still not very good – score. Interesting is the increase, i.e. worsening, from C1 to C2. This is explainable by the fact that methods are added but due to an approaching deadline there was not enough time to describe the interactions explicitly.

**ID3.8.5 Classes/Interfaces not called in MSC.** As for 3.8.4 the aim is a low value for classes. For interfaces the argumentations can tend in two directions: the object-oriented argument is, that interfaces cannot be instantiated (aiming on a high value) and the pure modeling argument is, that interfaces' interaction must be described (aiming on a low value). The designers in our case study were obviously advocates of
the object-oriented argument. Interesting is to see that B1 has the best result, but B2
the worst. In the case of B2 it was actually not yet intended to have message sequence
charts at all, as B2 – a new UML model from scratch – is still in a premature phase,
but the message sequence charts are going to be added to the model.

**ID3.10.1 Abstract Leaf Classes.** Abstract classes should be superclasses of concrete
classes. This rule is followed strictly in all cases.

**ID3.10.2 Abstract Classes having abstract Super Class.** This rule is followed
strictly in all cases.
4.3 MSC related Metrics

<table>
<thead>
<tr>
<th>ID3.9.1 Objects without Name</th>
<th>A1</th>
<th>A2</th>
<th>B1</th>
<th>B2</th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID3.9.2 Messages without Name</td>
<td>52.00%</td>
<td>61.58%</td>
<td>91.92%</td>
<td>86.67%</td>
<td>76.70%</td>
<td>75.00%</td>
</tr>
<tr>
<td>ID3.9.3 Objects without Type</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>ID3.9.4 Messages without Method</td>
<td>0.00%</td>
<td>0.00%</td>
<td>6.67%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>ID3.9.5 Messages between unrelated Classes</td>
<td>58.73%</td>
<td>7.62%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>27.14%</td>
<td>27.40%</td>
</tr>
<tr>
<td>ID3.9.6 Abstract Classes in Scenario</td>
<td>71.94%</td>
<td>79.37%</td>
<td>77.73%</td>
<td>43.14%</td>
<td>81.90%</td>
<td>81.74%</td>
</tr>
</tbody>
</table>

Table 9 – MSC related Metrics

![Bar chart showing MSC related Metrics](image)

**Figure 20 - MSC related Metrics (Diagram)**

**ID3.9.1 Objects without name.** Named objects improve the understandability of MSCs, but they are especially essential in cases of multiple instantiations of the same class in an MSC to distinguish the roles. The values between 52% and 92% are poor values.

**ID3.9.2 Messages without name.** The rule of naming messages to increase understandability of a model is followed in all cases very strictly.

**ID3.9.3 Objects without type.** The rule of typing objects in a model is followed in all cases very strictly. The poorest value (B2) is caused in the fact that the few MSCs in B2 are auxiliary notes instead of real parts of the model.

**ID3.9.4 Messages without Method.** The aim is a low value, meaning that many messages correspond to a method in the class diagram. The best value has A2 and also C1 and C2 are acceptable. Interesting is the large difference between A1 and A2, which are from the same organization. B1 and B2 do not have methods; therefore this result is no surprise.

**ID3.9.5 Messages between unrelated Classes.** As this is a very strict rule, the scores around 71% to 81% are not surprising. Interesting is the observation that the values of all case studies are similar (except for C2, where the MSCs are not supposed to be part of the model).
ID3.9.6 Abstract Classes in Scenario. This rule can, again, be argumented in two manners: the object-oriented argumentation says that abstract classes cannot be instantiated (aiming on a low value), the modeling approach arguments that all classes must be described in MSCs (aiming on a large value). The results show that the practitioners adhere to the object-oriented argumentation.

4.4 State Diagrams
From the six visited models only one model (C2) is making use of state diagrams. C2 does not assign more than one state diagram to any class; therefore it follows rule 3.11.1 strictly. Rule 3.11.3 (Incoming messages correspond to state transitions) could not be validated, as the tool’s database does not yet support queries of the necessary complexity.

Rule 3.11.2 postulates that a state diagram must describe dynamic classes. This rule builds on the heuristic dynamicity. We will discuss the dynamicity in more detail.

4.5 Dynamicity Heuristic
A first assumption was, that a complete model would necessarily have a state diagram for each class. This assumption was abandoned very early for two reasons: it is unpractical to create a state diagram for each class and the internal behavior is not at the same high dynamic level for all classes. The criterion to decide whether a class needs a state diagram should therefore be dependent on its internal dynamic level – its dynamicity. In 3.11.2 we define the dynamicity as the amount of all incoming and outgoing messages of the class’ instantiations in all MSCs. But what is a high dynamicity? How to define a threshold? In a first attempt we tried to use the third quartile as a threshold. The pro argument is, that it identifies a given percentage (in the case of the third quartile 25%) of the classes, but the counter-argument is that the distribution of dynamicity is not necessarily similar to various models, thus a class belonging to the upper quarter of dynamic classes does not mean it has an extraordinarily high dynamicity.

To get insight in the distribution of dynamicity over classes we have created the histograms of our cases (Table 10).

\[\text{Footnote} 4\text{ In a sample that is ordered according to an ordinal scale the third quartile contains the 25\% of the elements with the highest values.}\]
Table 10 – Dynamicity Histograms

The table displays the distribution of all six cases. The dynamicity is given on the x-axis and the y-axis gives the percentage of classes with the particular dynamicity.

We observe that there are strong similarities between the distributions in the six underlying cases. The curves clearly distinguish two groups of classes. On the left of the leftmost minimum (indicated by the dashed line) is the largest group of classes. This group contains the classes with a relatively low dynamicity. The remaining classes are in the group on the right of the dashed line. These few classes exhibit remarkably high dynamicity values.

Our first conclusion here is, that the distribution of dynamicity indeed shows similar characteristics for models from various domains, organizations and size. Note that B2 only contains a few (7) state diagrams, which are actually not part of the real description of the system. In spite of that the curve of B2 shows the same characteristic as the curves of the much larger systems A1 and A2.

To validate the expressiveness of the dynamicity heuristic, we have now to investigate whether the first minimum really distinguishes the highly dynamic classes from the classes that do not necessarily need to be described by a state diagram. Another question to answer is, how the expressiveness of the heuristic relates to the degree of a
model's completeness with respect to message sequence charts. (Remember that the
dynamicity is the count of events in all MSCs, but what if there are only a few
MSCs?)

<table>
<thead>
<tr>
<th></th>
<th>Total Classes</th>
<th>Classes in MSC</th>
<th>Identified</th>
<th>% Identified</th>
<th>% Identified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(of total)</td>
<td>(of occurring in MSC)</td>
</tr>
<tr>
<td>A1</td>
<td>142</td>
<td>76</td>
<td>3</td>
<td>2.11%</td>
<td>3.95%</td>
</tr>
<tr>
<td>A2</td>
<td>168</td>
<td>68</td>
<td>7</td>
<td>4.17%</td>
<td>10.29%</td>
</tr>
<tr>
<td>B1</td>
<td>34</td>
<td>22</td>
<td>4</td>
<td>11.76%</td>
<td>18.18%</td>
</tr>
<tr>
<td>B2</td>
<td>69</td>
<td>11</td>
<td>3</td>
<td>4.35%</td>
<td>27.27%</td>
</tr>
<tr>
<td>C1</td>
<td>45</td>
<td>26</td>
<td>4</td>
<td>8.89%</td>
<td>15.38%</td>
</tr>
<tr>
<td>C2</td>
<td>46</td>
<td>26</td>
<td>3</td>
<td>6.52%</td>
<td>11.54%</td>
</tr>
</tbody>
</table>

Table 11 – Classes Identified by Dynamicity Heuristic

Table 11 gives the numbers of classes that were identified to be extraordinarily
dynamic. The percentage of identified classes (per total number of classes) varies
dependent from the model between 2% and almost 12%. For the further analysis of
the expressiveness of the dynamicity heuristic we consider cases B1, C1 and C2 (as
the MSCs in B2 do not really belong to the model and the supplier of A1 and A2 was
no longer available for further analysis).

For the analysis of the expressiveness the architects were asked to pinpoint the classes
with a highly dynamic behavior and that set was compared with the set identified by
the dynamicity heuristic.

<table>
<thead>
<tr>
<th></th>
<th>Identified by Heuristic</th>
<th>Identified by Heuristic after Enhancement</th>
<th>Identified by Architect</th>
<th>False Accepted</th>
<th>False Accepted after Enhancement</th>
<th>False Rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C1</td>
<td>4</td>
<td>N/A</td>
<td>5</td>
<td>1</td>
<td>N/A</td>
<td>2</td>
</tr>
<tr>
<td>C2</td>
<td>3</td>
<td>N/A</td>
<td>5</td>
<td>0</td>
<td>N/A</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 12 – Comparison of Indications by Heuristic vs. by Architects

**B1.** The architect pinpointed two classes as highly dynamic. Both were also identified
by the heuristic. The score is therefore 50%. The model in B1 defines the classes as
*analysis objects*, i.e. the stereotypes *boundary*, *control* and *entity*-object are applied
to the classes.

![boundary](image)

![control](image)

![entity](image)

**Analysis Symbol Stereotypes as defined by Jacobson [JACO94]**

The entity-object stereotype classifies the classes of a database like character, i.e. they
are accessed frequently and do not perform actions themselves. Boundary objects are
located at the system’s boundary and are responsible for the interaction of the system
and the periphery. Control objects are the internal objects that “do the work”. In the
case of this model, taking into account these used stereotypes enhanced the heuristic. The classes identified by the dynamicity heuristic are either really of a highly dynamic character, or they are data-model elements and are therefore accessed frequently. Thus we enhanced the heuristic by taking into account the stereotypes. After the enhancement the heuristic identified exactly the same classes as the architect pinpointed. We studied this case in more detail to investigate whether entity-objects can even be detected when no stereotypes are applied by considering the fan-in and fan-out. Please refer to the Appendix for this investigation.

C1 and C2. In C1 the heuristic (without enhancement, because no stereotypes were used) identified four classes. The architects identified five classes; three of those classes were also identified by the heuristic. The two classes not identified by the heuristic but pinpointed by the architects were closely related to one of the identified classes (say class X). The two classes had two work synchronized with class X and therefore their interactions caused their highly dynamic internal behavior. Hence, a state diagram should also describe them. The architects took the indications of the heuristic into account when they created C2, the successor of C1. In C2 they created state diagrams for three of the identified classes and the two classes with the strong interaction to class X. Furthermore they changed the MSCs slightly to describe the system’s behavior more precisely. As a result of the evolution of the interaction diagrams the heuristic did not have any false accepts in the evaluation of C2.

### 4.6 Evaluation

We have visited six case studies from three different suppliers in the validation phase of this project. This – limited but inhomogeneous – collection of models enables us to make two kinds of observations:

- About the specific characteristics and defects detected in the case studies and
- About the applicability and properties of the rules and metrics.

#### 4.6.1 Observations concerning Case Studies

The models A1 and A2 were obtained from the same supplier. The models describe two different packages of one and the same system; therefore members of the same organization, even the same team, create them. It is noteworthy that there are large differences in some results of A1 and A2, mainly in the well-formedness rules. This indicates that there are no “modeling” standards within the team. The architect of A2 adheres very well to most well-formedness rules whereas A1 exhibits serious flaws. For the completeness and consistency metrics, both models had similar results. Unfortunately the supplier could not be contacted to answer our questions (mainly about the large difference between some A1 and A2 results).

The high level models B1 and B2 are subsequent versions of the same system. B2 is the from-scratch designed successor of B1. The analysis models B1 and B2 use the analysis symbol stereotypes for classes (entity, boundary and control object). The reflections of the architect on the dynamicity results helped to enhance the heuristic by taking the stereotypes into account. This enhanced heuristic yielded a very accurate conformance with the architect’s indications. The rules and metrics indicated a mismatch between the scenario view and the logical view, which was indeed attested by the supplier. It seemed that the use cases were taken from outside the scope of the system and only the requirements for interaction with other systems instead of the actual requirements. This problem was corrected in B2. Whereas in B2 the rules and
metrics indicated that there was “something strange” with the message sequence charts. The very few MSCs in the architecture were only very loosely related to the use cases and class diagrams. The supplier confirmed that the model was not supposed to contain MSCs at all yet, but the found MSCs were relicts of some “notes”.

The models C1 and C2 (subsequent version of the same system) had overall the best scores out of our six cases. This was in some way expected from post-MSc graduation projects (amongst other things because of the smaller project size). The architects appreciated the results from C1 and enhanced the successor, C2, according to the indications obtained from C1. Especially the dynamicity heuristic was taken into account for the enhancements of the model. An interesting observation was that the identification of one specific class by the heuristic triggered the architects in identifying two more classes with the need for a state diagram. The two additional classes were functionally closely related to the first class.

4.6.2 Observations concerning Rules and Metrics

Besides assessing the models the case studies served to evaluate the applicability and distinguishing power of the proposed rules and metrics. We will review the basic size metrics (3.6) apart from the other rules and metrics, as they are more to get an overview over the entire system, than specifically to identify rule violations. They are easy to collect and intuitive to understand. They give a quick overview over the dimensions of the model and allow an easy assessment of which diagram types and views are modeled (this might become even more valuable when the meta model is extended). The ratios between different element types give a quick indication of the model’s abstraction level and diagram types that are possible underdeveloped.

According to the results received in the six case studies, we can divide the remaining rules and metrics in three categories. Note, that this categorization only reflects six underlying case studies and we suggest an assessment of the techniques in a broader field of cases studies to be able to generalize the observations. We found the following three categories of rules and metrics:

Category A. These rules and metrics proved in the case studies to be powerful in distinguishing “good” models from “bad” ones (w.r.t. completeness, consistency and well-formedness). The results for these techniques are for the most of our six visited cases rather worse than the preferred and expected outcomes.

Techniques in Category A:

3.7.3 Scenarios without Use Cases
3.8.1 Non-Private Attributes
3.8.2 Classes/Interfaces without Methods
3.8.4 Methods not called in MSC

3.8.5 Classes/Interfaces not called in MSC
3.9.1 Objects without Name
3.9.4 Messages without Method
3.11.2 Dynamicity

Category B. The results of techniques in this category could not be calculated correctly due to a simple implementation of the underlying database. The SQL statements (that implement the rules and metrics) for this category are too complex for the SQL interpreter.

Techniques in Category B:

3.7.2 Actor needs Use Case
3.9.5 Messages only between related Classes
3.11.3 Incoming Messages correspond to State Transitions
Category C. All of our case studies score a good value, i.e. (almost) no violation, for these techniques. Note that this might depend on the sample of case studies available in this project. Thus these techniques should be validated in future case studies, to consolidate or discard this observation.

Techniques in Category C:
3.7.1 Size of Use Cases
3.8.3 Size of Class in Methods
3.9.2 Messages without Name
3.9.3 Objects without Type
3.9.6 Abstract Classes in Scenario
3.10.1 Abstract Leaf Classes
3.10.2 Abstract Classes having Non-Abstract Super Classes
3.11.1 Classes have at most one State Machine
5 Evaluation

This chapter starts by briefly summarizing the problem and what we established during this project. Then our observations during of the case studies are reflected and the conclusions we draw are presented. The chapter ends with recommendations for future work.

In large-scale industrial projects practitioners encounter the difficulty that the early artifacts, i.e. the architecture and design models, have such large complexity and size that it is hard to oversee the completeness of the system model. This results in uncertainty about when to proceed to the next project stage and uncertainty about the value of design metrics.

Based on interviews, a survey and a literature study we have decomposed and defined the concept of completeness and defined the scope of our project. We have created a meta model that serves as basis for a set of techniques that we propose to assess the model completeness. The techniques are rules and metrics that apply in three different levels: well-formedness, consistency and diagram completeness. These techniques were implemented in an extension of an existing metrics-based evaluation tool.

Six case studies were conducted to validate the proposed techniques and to empirically investigate the completeness and consistency of industrial models. The obtained results were discussed with the architects. For a large subset of the proposed techniques the architects confirmed the obtained indications.

A first step in model completeness analysis is to obtain the complete set of models. For small projects, e.g. C1 and C2, it is easy to obtain the model file(s). A large system is often described by a model that is distributed over several files. Especially when different model versions are available it is sometimes tricky to find the different files from the same version. Possible problems are introduced, when the storage-format of the model-files changed over time (digital durability). In large projects it is usually necessary to consult project documentation and ask project members for assistance.

5.1 Observations

The quantitative assessment of completeness gives insight into how the UML is used for architecture and design. The total number of rule violations detected in the models is quite large and industrial practice projects move to the next stage with still a significant number of violations in the model. The case studies show that the types and quantity of violations is strongly related to the architects habits and conventions. Even within the same project strong differences were detected, probably due to the fact that no "coding standards" exist for modeling. The two student project cases exhibit the fewest number of rule violations. The weaker time pressure than in industrial projects and the fact that the students learned the UML during their education and not on-the-job are possible factors for this. An interesting observation is that all cases show remarkably similarities in distribution of dynamicity of classes.

5.2 Conclusions

The two basic goals in software engineering are to reduce process cost and resources and to deliver high-quality software. The presented rules and metrics contribute to these two goals as follows:
Goal: Reducing Process Cost and Resources
- Shortening time of reviews
- Correcting faults early is cheaper
- Tool helps to establish and control modeling conventions ⇒ common semantics about UML models in organization; less miscommunication

Goal: Delivering High-Quality Software
- Assessment of model quality in early stage
- Complete model ⇒ seamless transition to implementation ⇒ model exactly reflects code ⇒ model is better usable for maintenance and reuse

In contrast to reviews or inspections of the model that are performed by the architect, the tool calculates the results fast and impartially. It must be clear that the tool indicates suspects, but cannot judge. The task of judging whether an incompleteness or inconsistency is present is up to the architect. We suppose to use the tool as an aid to indicate potential suspects in the model. It can be applied by architects and as a managerial tool to quickly identify suspects in an efficient way. In our case studies this approach has indeed pinpointed the architects to flaws in the model and the review time was reduced.

In the case studies the techniques successfully discovered incompletenees and inconsistencies in the models. Thus, the techniques contributed to detect modeling design faults in an early stadium. The early applicability of quality assessment techniques after assessment of the completeness has to be investigated.

The case studies showed that even in one organization different developers adhere to different modeling styles. This clearly emphasizes the lack of “modeling standards”. The presented techniques are applicable to describe and check such conventions. This contributes to a common semantics within the organization and reduces miscommunication and possible integration problems.

In this project we encountered the problem that in practice models often strongly differ from their actual implementations. The goal of the presented techniques is to allow the designer more control over the model with respect to completeness. This should minimize the gap between model and implementation by developing more complete models. The assumption that a seamless transition between model and implementation decreases maintenance effort and encourages reuse (as the code is better documented) has to be investigated in more detail.

5.3 Recommendations

5.3.1 Theoretical Recommendations
As suggest to future projects we recommend to investigate more case studies to create a base of reference numbers. Additionally it is very interesting to address the following questions:
- Are specific characteristics of occurring rule violations related to parameters of the development process (possible parameters: project size, project domain, degree of education of architects...)?
- How do modeling conventions – explicit or implicit – within an organization improve model completeness and consistency?
• Does a more controlled modeling process (with respect to completeness and consistency) decrease problem that occur in later phases? (integration problems, miscommunication, testing overhead, defects...)?

• What is the relevance and impact of the different rule violations on the delivered system (with respect to quality, i.e. maintainability, reusability, comprehensibility...)?

• How do UML models evolve over various versions and in different life-cycles (incremental, waterfall...)?

5.3.2 Technical Recommendations
The presented collection of techniques does not claim to be the entire collection of completeness and consistency rules. We suggest to extend the collection and the meta model, and validate (new) rules and metrics in further cases studies. The implementation of the rules and metrics as SQL queries allows easy extensibility. A drawback of the actual implementation is that the MySQL database only allows queries of a simple complexity. Therefore the maintainer should look for new releases of the database or port the implementation to a more advanced database system.
6 Bibliography


Recommendations


7 Appendices

7.1 Implementation of the Tool
Especially in large-scale projects the complexity of models exceeds human imagination. We have proposed a set of metrics and rules in this document that aids architects in assessing model completeness. For this aid to be helpful and efficient tool support is a necessity.

In our project we have extended (and slightly modified) the SAAT tool of Johan Muskens. Here we will give an outline of the tool’s design and implementation. For a more detailed description of the tool refer to [MUSK02].

7.1.1 Environment
Figure 21 depicts the environment of the SAAT tool.

The SAAT tool analyzes architectures described by UML models. The SAAT tool’s input is the UML model in the XMI representation. In our project we have used the case tool Rational Rose [ROSE] to create (or read) the models and to export the models into the XMI representation. Any case tool supporting the XMI standard can be used to generate the input files for the SAAT tool.

The SAAT tool performs the actual analysis of the model. The tool generates the report of the analysis in the HTML format, such that any arbitrary HTML browser can view the results.

The dashed line in Figure 21 separates the scope of the SAAT tool from its environment.

![Figure 21- Environment of the SAAT Tool (from [MUSK02])](image)
7.1.2 Components

The SAAT tool’s primary task is the analysis of architectures. The functionality is spread across a collection of components. The specific tasks of the component will be explained here; in Figure 22 we have illustrated the component’s interaction.

**Preprocessor.** The XMI representation of a model may contain control characters, which are not relevant for the analysis. These control characters might disturb the Parser and Analyzer components and are therefore removed from the XMI file by the Preprocessor.

**Parser.** The Parser component extracts the relevant information from the XMI representation of the model. The relevant information is the data needed by the relational data model, described in chapter 3. The Parser transforms the extracted data in a format that can be read by the database filler component.

**Analyzer.** This component actually performs the analysis of the model. It executes the predefined rules and metrics as SQL queries on the model data stored in the database. The results are written to an ASCII file to be further processed by the statistic filter and statistic calculator.

**Inheritance Relator.** After the database has been filled with the model data, the inheritance relator extends the data such that for subclassed model elements, the inherited methods and attributes can be retrieved immediately. This is necessary, as the actual MySQL implementation does not allow traversing the database in queries.

**Statistic filter.** This component filters the outlying values. This functionality was not used for this project.

**Statistic Calculator.** This component calculates the outlying values. This functionality was not used for this project.

**Database Creator.** The database creator uses the MySQL database management tool to create a new database for each analysis. The structure of the database is according to the relational data model presented in chapter 3.

**Database Checker.** This component was used in the SAAT project for performing simple checks on the data in the database. It is redundant in our project, as the profound completeness and consistency checks are performed in the main analysis.

**Database Filler.** This component fills the database with the architecture information extracted by the parser component.
7.1.3 Implementation

The various SAAT tool’s components are implemented using several different technologies and programming languages. Most of the components were not or only slightly changed with respect to their primary implementation in the SAAT tool. We will present their implementation only briefly and refer the reader to [MUSK02] for more details.

The parser component has been implemented completely new. As the XMI standard is an extension of the XML standard, we have used XML technology for the implementation of a robust and efficient parser component. The Transformations of the XMI file are defined in XSLT stylesheets and the SAXON parser [SAXON] is used to perform the transformations defined in the XSLT stylesheets on the XMI input file.

The preprocessor component is an extension to the SAAT tool. This component is a PERL script. Additionally the following components are implemented using the scripting language PERL: statistic filter, statistic calculator, analyzer (database access by a C++ module), database checker (database access by a C++ module), and database filler (database access by a C++ module).

Mainly the database modifications and access components are making use of C++ modules: database creator, database checker, analyzer, and database filler.

The main control of the SAAT tool lies in the SAAT batch file, as the SAAT tool is a command line tool.

The database management system used in the implementation is the open source MySQL server, version 4.1.0α.
7.2 Weyuker’s Properties

The nine criteria proposed by Weyuker [WEYU88] give a framework to evaluate software metrics’ properties using a formal theoretical basis. The widely used criteria will be presented in the following paragraphs.

The Weyuker properties are intended to evaluate complexity measures on source code metrics. Weyuker denotes programs with the letters P, Q and R and the value of a metric with the notation |P|, |Q| and |R|.

**Property 1:** \( \exists \ P, \ Q : \ |P| \neq |Q| \)

The metric does not map all inputs on the same output. There has to be variation in output when the metric is applied to different inputs.

**Property 2:** Let c be a nonnegative number. Then there are only finitely many programs of complexity c.

This property states the requirement that changing a program must ultimately change the measured value.

**Property 3:** \( \exists \ P, \ Q \land P \neq Q : \ |P| = |Q| \)

There exist programs that differ, but are effectively the same, which must have the same measured value.

**Property 4:** \( \exists \ P, \ Q \land P = Q : \ |P| \neq |Q| \)

This property states that the metric must apply to the internal implementation of the program’s effect and not only the external effect. \( P = Q \) denotes that programs P and Q halt on the same input.

**Property 5:** \( \forall \ P, \ Q : \ |P| \leq |P;Q| \land |Q| \leq |P;Q| \)

The “monotonicity” property: components of a program cannot be more complex than the program itself.

**Property 6:**

a) \( \exists \ P, \ Q, \ R \land |P| = |Q| : \ |P;R| \neq |Q;R| \)

b) \( \exists \ P, \ Q, \ R \land |P| = |Q| : \ |R;P| \neq |R;Q| \)

The composition of programs may introduce complexity due to interaction of the program parts. This property states that two programs, which have the same measured complexity, might result in programs of different complexity when combined with one and the same other program. This phenomenon occurs because of the differences in complexity due to interaction between program parts.

**Property 7:** There are programs P and Q with Q a permutation of the program statements in P such that |P| \( \neq |Q| \).

This property denotes that the order of program statements affects the outcome of the complexity measure.

**Property 8:** \( P \) is a renaming of \( Q \) \( \Rightarrow \ |P| = |Q| \)

This is an intuitive property, as the complexity should not differ due to renaming. Weyuker gives a counterexample with a metric that measures the understandability of a program by the choice of names.

**Property 9:** \( \exists \ P, \ Q : \ |P| + |Q| < |P;Q| \)
This property is stronger than property 5. There are cases, that the additional complexity introduced by interaction between combined programs is greater than zero.

Note that the Weyuker properties were not introduced for object-oriented designs. The concatenation operation cannot simply be translated to an operation in object-orientation. Chidamber and Kemerer, who evaluated their six metrics of the metrics suite using Weyuker properties, invented a concatenation operation on class level comparable to union in sets [CHID94]. Fetcke formulates concatenation operations on class level based on the intersection in sets and concatenation operations for the other abstraction levels of object-orientation [FETC95a, FETC95b].
7.3 Investigation of discovering entity-objects by fan-in/fan-out analysis

In section 4.5 the dynamicity heuristic was validated for our six case studies. In case B1 we encountered the fact, that the dynamicity heuristic is successful for the stereotypes control- and boundary-objects, but it accepts the entity-objects wrongly. Our assumption was, that the database-like entity object is accessed much more, than that it calls messages from other objects. Therefore we counted the number of incoming messages (fan-in) and the number of outgoing messages (fan-out) of all instantiated classes in case B1.

Figure 23 - fan-in / fan-out analysis for case B1

Figure 23 shows the (normalized) value of fan-in divided by fan-out with the red bars. The white bars show the normalized dynamicity for all instantiated classes. (Classes are ordered according to their dynamicity). The labels indicate the stereotype of each class. It is remarkably that the fan-in/fan-out value indeed corresponds to the stereotype: a positive value (fan-in > fan-out) corresponds to entity objects and a negative value corresponds to control and boundary objects. This observation leads us to the assumption, that a database-like class (entity object) can even be distinguished from other classes by analyzing the fan-in/fan-out-value. Note that in Figure 23 only three out of 22 classes do not follow this rule. And these three classes are in the in the second half, so they have a low dynamicity, i.e. they are not relevant for the dynamicity analysis.
7.4 Outcomes of the Survey

7.4.1 Introduction

The methods and tools to be delivered during the graduation project are meant as a theoretical basis to support practitioners in the architecture and design phase of (large) software projects. In the scope of the graduation project a survey among practitioners was conducted for the following reasons:

a) at the transition between the phase of building the theoretical phase of the project and the beginning of implementing the methods it was desirable to get practitioners' feedback on the hitherto gathered results (definition, sorts of metrics, ...).

b) for a further development of the methods it was valuable to gain more insight in the way practitioners deal with the UML, how they proceed in the development of their models and which problems are encountered due to completeness and consistency problems of UML models.

The survey was conducted in the form of a web-based questionnaire. In addition to collecting the data only within the Océ company and therefore reaching a homogenous audience (with respect to company culture, project and application background, tool usage, management, ...) this approach enables to reach a large and heterogeneous audience within a reasonable amount of time. The URL of the questionnaire was posted on various related usenet newsgroups (e.g. comp.object, comp.object.measurement and comp.software-eng), distributed within the software department of Océ and forwarded to people with experience and interest in UML modeling.

Experience has shown that too extensive questionnaires do not get a large amount of responses, therefore we decided that the questionnaire should not exceed 20 questions, and should be completed in no more than 10 to 15 minutes. We grouped the 20 questions in four sections:

Job context; to gain information about the job background of the practitioner.

Definition of completeness; to gain feedback and opinions of the definition of completeness.

UML usage; to gain information on the way practitioners are using the UML, which tools they use and in which way they develop their models using the UML.

Completeness and related questions; to get feedback on some of the concepts established so far within the project.

No question asked for the personal information, e.g. the email address. The questionnaire was therefore anonymous. By this means, the results are not biased for example because of someone feeling he has to answer that he is using the UML strictly conforming to the specification.

Over a period of two month we received a total of 80 responses of the questionnaire, 63 of those responses came from outside Océ.

---

5 "Software Architecture Completeness Analysis" is the topic of the MSc research project of Christian Lange at Eindhoven University of Technology (The Netherlands) in cooperation with Océ Technologies B.V, Venlo, The Netherlands. The project is supervised by Dr. Michel Chaudron (M.R.V.Caudron@TUE.nl)
7.4.2 Job Context

1. What is your employment status?

The intent of the survey was to investigate the way practitioners deal with the UML and indeed their experiences and feedback concerning completeness analysis. The vast majority of the received answers was sent by employees. As employees and contractors are considered to be practitioners, we can conclude that the respondents come from the intended audience. We can observe that the influence of other respondent groups (e.g., students) is a bit larger for small projects. In large projects these groups (almost) disappear.

2. What are the main tasks of your responsibility?

This question allowed more than one answer. The result reflects that most questionnaires were completed by people, who are familiar with the first four phases of the software lifecycle. Our research focuses especially on analysis of software artifacts in the early stages of the development.

3. What is the average size of the projects you are involved with (in man years)?

<table>
<thead>
<tr>
<th>Average</th>
<th>Stdev</th>
<th>Max</th>
<th>Median</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>31.05</td>
<td>66.4</td>
<td>300</td>
<td>6</td>
<td>0.03</td>
</tr>
</tbody>
</table>

The distribution of project sizes ranges from 0.03 man years to 300 man years. The average project size is greater than 30 man years. For the analysis of the survey it is wishful to have a population which does not mainly consist of opinions of developers who are only involved in small toy projects. When we consider projects of size less than 5 man years as toy projects, we have almost 60% large (non-toy) projects. This is the majority of all completed questionnaires.
4. What is the application area of your activity?

Information systems are by far the largest population amongst all received answers. But still more than a quarter of all responses comes from an embedded systems context. This is very interesting, as in embedded system environments the UML-RT is widely used. This UML variant allows code generation from the model (and therefore a different notion of ‘completeness’ is applicable). See the quotes in the appendix for some comments on UML-RT and completeness.

Please note, that this question allowed more than one answer. The option “distributed systems” was not checked at all and is therefore omitted in the diagram.

7.4.3 Definition of Completeness

5. Please indicate which aspects you feel should be part of a definition of completeness.

A UML model is complete if...

() it demonstrably meets the functional requirements (e.g. through some means for tracing)

() it demonstrably meets the non-functional requirements (e.g. through dedicated analysis methods)

() it is consistent with the syntactic rules for UML

() the diagrams are well-structured and laid-out

() the model reflects the significant design decisions
7.4.4 UML Usage

6. What tool(s) do you use for designing your UML model

No big surprise that the market leader, Rational Rose, is also in this population the most widely used tool. The two other mainly used tools are Visio and ArgoUML. These two tools do enable XMI export as well, and therefore allow analysis with the SAAT tool.

The results of this questionnaire show that professional projects (large project sizes) tend to use dedicated CASE tools (Rose, Argo, Visio).

The tools from Together Software and ArgoUML are mainly used in information system environments, whereas they hardly play a role in embedded systems.

Non-dedicated tools such as Word and Powerpoint are used supplementary to the dedicated tools in larger projects. In some toy projects these tools appear to be used as primary tools.

Most people who answered “none” stated in the comment field that they use pen and paper.

7. To what degree do you follow the UML standard (syntax and meaning) in your diagrams?

The UML is followed loosely with a slight tendency to strict. Indeed these results are based on subjective self-assessment. We expected to find a correlation between strictness of UML usage and project size and strictness of UML usage and CASE tool, but the average was in all subsets of our population around 3.3 (on a scale from 1 = “not at all” to 5 = “very strict”), therefore no correlation could be concluded.
8. I make use of the following architectural views (as they are termed in Rational Rose)...

The use case view and the logical view are the most frequently used views, this in no big surprise. However the low acceptance of the scenarios is unexpected. But when we compare the results of this question with the results of question 9, we observe that 16 of the received answers state scenario usage in question 9 (which meets the expectations). Therefore we might conclude that the usage of the scenario view should be a lot higher than this diagram supposes.

When we compare the usage of scenarios per application domain, we observe that (relatively) the logical view is very widely used in embedded systems, whereas the deployment view is hardly used. In information systems the use case view plays a greater role.

9. In which order does your model develop?
Please assign numbers, that allow a (partial) order. E.g. Use Cases=1, Scenarios=2, Actors=2 means first the Use Cases, and then at simultaneously Scenarios and Actors. Assign only numbers to the architecture elements you actually use.

This question allows us for a twofold interpretation. First we take a look at the occurrence count for the architecture elements:

Note that we distinguish between identifying classes, which means just declaring a class to exist, and defining classes, which means defining their attributes and methods.

The figure shows that there are four groupings:
Identification of classes is most common, followed by use cases and defining the classes' interfaces.

The second most common group of UML elements contains actors, interfaces, inheritance and associations.

The third, but still common group, is formed by scenarios, state diagrams and multiplicities.

Not very accepted is the use of analysis symbols, only 13 practitioners in this survey mention analysis symbols in their response.
We can make an order of how an architectural UML develops over time. According to the practitioner’s responses, most developers begin with

1. actors and use cases, then there follow
2. analysis symbols (if any),
3. scenarios, identifying classes
4. inheritance, associations, interfaces, define classes,
5. state diagrams, multiplicities.

The first step contains elements of the requirements phase and helps to narrow the problem domain down. Steps two and three are clearly an analysis of the problem whereas the last two steps contain design elements leading to an implementation.

As we can see in the figure above, not always all architecture elements are used. The usage depends on the context of the model, its abstraction level and the taste of the designer/developer.

**7.4.5 Current and desired use of metrics**

**10. Why are you collecting SW metrics?**

For the interpretation of this figure it is important to know that the question allowed more than one choices. This means that almost half of the practitioners participating in this survey are measuring neither their process nor their artifacts.

We can observe that small and medium size projects prevalently use metrics for reasons of improvement, whereas this aspect for large projects comes behind understanding and controlling (as large projects give rise to higher complexity).

This diagram displays the acceptance of metrics usage split per project size. We observe that the larger the project, the higher the acceptance of metrics usage. Indeed this is an assumption and should be proven by surveying a larger sample.
11. In my projects, I currently use SW product metrics in the following phases

12. I think using SW product metrics would be useful in the following phases

Questions 11 and 12 show that in the hitherto situation most widely used software metrics are implementation, i.e. source code metrics. In the practitioners opinion it is use especially useful to use metrics not only in the implementation phase but also in the early stages of the software lifecycle, architecting and design. It is in the early stages, where bug defect detection is most successful and contributes to safe wasting resources. In the architecting and design phase the methods of architecture completeness analysis apply.

13. What criteria do you use to stop modeling?

It is obvious that when schedule or a deadline triggers a project to switch from modeling to implementation phase, it is makeshift. This will most likely have negative influence on the product quality. Therefore the relatively large accumulation for schedule/deadline is alarming. However it is interesting that completeness has the largest value as a criterion. The concept of completeness has almost not been addressed in the literature, but many of the practitioners give it as their criterion to stop modeling. Currently there is no more than rules of the thumb concerning architecture completeness, this emphasizes the need for a scientific approach to this topic.

This shows how the two reasons deadline and completeness correlate with the project size. The larger the project size, the higher the interest in complete models. Large projects yield to reason as deadlines and schedule constraints get less important as a trigger to transfer from design to implementation phase.

Indeed this is an assumption and should be proven by surveying a larger sample.
14. Which of the following problems have you encountered due to start implementing without having a complete model?

As UML is besides a modeling language also a communication medium it is not surprising that miscommunication gets the highest score. Interesting to see is that large testing effort gets a relatively high score for large projects. This is probably due to the need of integrating many small chunks of software in large projects.

15. It must be intuitive how a SW metric relates to a system's quality attribute.

The majority of the responses state that software metrics must be easy to understand in their function and meaning. Practitioners have more trust in intuitive metrics. In our evaluation of the responses we observed that the agreement is highest amongst the large projects (average 4.1), whereas the average from small projects is 3.3.

7.4.6 Completeness and related questions

16. The number of inconsistencies in a model is a proper indication of the model's (in)completeness.

A clear majority of the practitioners agrees with the usefulness of inconsistency counting as a metrics concept. This is slightly contrasting to the result of the completeness definition related question 5. In question 5 the choice “consistency with the syntactic rules for UML” has got the lowest score of all responses. Indeed question 5 deals with UML syntax consistency only, whereas this questions addresses inconsistencies in general. This observation might also relate to question 7, the degree of how strict the UML is followed. A large number of practitioners follow the UML standard only loosely.
17. In a complete model, not every class needs a state diagram associated with it, only the classes with strong dynamic behavior need a state diagram.

18. Classes having methods that do not occur in any scenario/sequence diagram indicate incompleteness of the model.

There is a large consensus that state diagrams are only necessary for classes with strong dynamic behavior.

19. On source code level, the amount of comment lines is a well known metric. UML CASE tools support to document/comment elements in the model. The amount of documentation of the architecture elements in a model is a meaningful measure for completeness analysis.

In the evaluation of this answer we detected that the agreement correlates to the size of the project.

20. The ratio between simple size metrics (number of use cases, number of scenarios, number of classes, ...) can give an indication of a models maturity.
7.4.7 Conclusions

The usage of the Universal Modeling Language (UML) has mainly two essential purposes:

- understanding the problem and finding the solution,
- usage as communication medium.

UML can be regarded as a communication medium in the situations when members of (large) project teams are sharing models or if stakeholders with different views, i.e. interests, are using common models. Clearly this behavior requires a common understanding of the UML. Therefore the UML is defined in the UML specification [UML]. The results of the underlying survey, though, say that in practice the UML is followed loosely.

Many practitioners stated that they have encountered problems due to incompleteness. A very prevalently reported problem was miscommunication, which might be caused in the fact that the UML is mainly used loosely and that completeness is not yet an understood attribute.

Completeness analysis is a new topic in the evaluation of software architectures and practitioners show exalted interest in this field. The responses to definition related questions and examples demonstrate that consistency is regarded as an integral part of completeness. In the context of completeness consistency is meant in the notion of consistency between overlapping architectural views (and diagrams).

In the hitherto quantitative support of evaluation during the software lifecycle most effort is spent in the evaluation on implementation level. Amongst specialists the agreement is large that especially in the early phases (architecture and design) tool supported quantitative evaluation techniques are very useful.

For our further research the outcomes of the survey influence the choice of the views to analyze in depth (logical, use case, scenario). Furthermore the relationship of consistency as part of completeness will be elaborated. When developing our rules and metrics set we will keep in mind that the UML is used loosely, therefore that the metrics set can be adjusted to specific needs (context dependent, abstraction dependent, implicit UML “coding standard” dependent) and that the intuition of metrics is an important factor for practitioners.

7.4.8 Quotes from the Survey

The presented questionnaire consists of 20 questions (most of them multiple-choice). Some of the questions allowed various interpretations or were designed to provoke discussions, therefore the respondent was able to write a comment on the question. These comments were taken very serious and surely inspired our considerations during this project. The comments cannot be reflected in numbers or diagrams, thus we present a selection of the comments in this section.

Q 5: Please indicate which aspects you feel should be part of a definition of completeness...

“All – if one of them is missing, you have a potential risk when going to the implementation phase. What is miss in this list is: the model is complete when at least the actor, static view (class diagram) and the dynamic view (MSC/collaboration diagram) is complete.”
“Any model, even a UML model, will not be complete. A model is an higher level abstraction of some lower level system. We use models to understand such lower level systems. I would say, a model is complete if it is sufficient for understanding such system. . . .”

“It depends also on the phase you are in. Analysis, Design, Implementation. Sometimes not all functional requirements are totally clear yet, so you can not be “complete” by definition”

“As a translationist my OOA models are, themselves, executable. So one applies the same validation tests to the model as the generated code.”

“I have never actually seen such a [complete] UML model. One of the weakest points of UML is handling non-functional requirements.”

“Completeness does not seem a universal attribute of a model. A model is complete when it sufficiently serves its purpose. This can be very different from situation to situation. Granted, this is an industry formed opinion and not academic.”

“I've never seen a model that is in the above sense ‘complete’ though. This is largely due to lack of time in the project – once the deadline gets in sight the model suffers and reverse engineering is not enough. Besides that, UML is not altogether sufficient for modeling non-functional requirements apart from capturing them.”

Q 13: What criteria do you use to stop modelling?

“I know it is often not effective to stop modeling because we are running out of time, but it is better to tell our customer we are ready to begin implementing and correct some design flaws in the implementation phase.”

“In fact one always uses multiple criteria in a business situation. One criteria may overrule the others, . . . But I rather only want to stop when I'm absolutely sure that the model is complete and is also experienced as such by the inspectors.”

“When I have enough information and when it is clear to me.”

“It is my goal to have one model where you can view analysis, zoom in on design and eventually zoom in on implementation.”

“In practice, I do not use ‘completeness’ in a strict sense. As a rule of the thumb completeness means all relevant, major design decisions have been covered.”

“None of the above or a combination of all of them. Reason to stop is either if the model is sufficient for communication or if the model is sufficient for analysis and design in such a way that you can start implementing without much risk of changes because of analysis and design errors or lack of detail.”

“ROI: Return on Investment. When adding more detail or elements to a model does not help in increasing the understanding of the model, it is time to stop modeling.”

“Stop modeling if it does not contribute anymore to understanding or communicating the system.”

Q 14: Which of the following problems have you encountered due to start implementing without having a complete model?

“Currently we are refactoring a component. This component has been developed very quickly without thinking about design. It is a wonderful experience to see which effort a bad design can cause at last…”
“More communication than what would be necessary with a complete model.”

“A lot of ‘rethinking’ during the implementation phase, ‘what is meant with this or that?’ This kind of unclearness cannot be afforded in that phase.”

“You have to put a lot of effort when something has to be changed. Documentation, Model, Implementation.”

“ ‘Magic’ being performed in between high level requirements and detailed module design. This magic is performed by human-beings that are called specialists.”

“Lack of roadmap that slows down programming”

**Q 21: What type of inconsistencies have you found to be a problem in your system?**

“Parts of the system on which analysis are started late on, causes sometimes major impact on the total systems (architecture/behaviour). Analysis/modeling on interface dependencies must be done completely for the total system, before starting to engineering the product.”

“Missing dynamical view, missing instantiation/usage examples (collaboration view), especially for information models, missing actor view. Inconsistency between model and implementation! (this should be a check in my perception)”

“How about adhering to certain rules? Modules may not use methods of certain layers in a layered system.”

“Multiple implementation of the same requirements in different places.”

“Differences between early documents (specifications, architecture design) and later documents/implementation. Inconsistency of interfaces – (dis)appearing methods, attributes. Inconsistency between model and implementation – usually the implementation includes more items (classes, attributes, etc.) than the original design.”

“Class Interface interpretation mismatches, with the real-world object. Example: A class-interface is mostly not entirely implemented, due to project (cost) constraints. So defining a Class-interface and the 100% fulfillment of that interface is a big project problem.”

“… I think the ‘maturity’ or quality level of your model should be specified (adjusted) for each project.”

“Inconsistencies between the model(s) and the other documentation that make up the design of a system.”