DOCTORAL STUDIES

PhD Thesis
– June 2014 –

# A METHOD FOR THE UNIFIED DEFINITION AND TREATMENT OF CONCEPTUAL SCHEMA QUALITY ISSUES

**David Aguilera**

Advisors
*Dr. Antoni Olivé*
*Dr. Cristina Gómez*

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

A thesis presented by *David Aguilera*
in partial fulfillment of the requirements for the degree of
*Doctor per la Universitat Politècnica de Catalunya – BarcelonaTech*

| | |
|---|---|
| *Author:* | David Aguilera |
| *Address:* | Department of Service and Information System Engineering<br>Edifici Omega, Despatx S206<br>C/ Jordi Girona, 1–3<br>08034 Barcelona, Spain |
| *Email:* | daguilera@essi.upc.edu — david.aguilera.moncusi@gmail.com |
| *Telephone:* | (+34) 93 413 71 74 |
| *Fax:* | (+34) 93 413 78 33 |

This thesis was written in LaTeX,
using free software tools.

*To my family.*
*To María.*

# Acknowledgments

This thesis has been possible thanks to the help, assistance, and friendship of many people, to whom I am indebted. I would like to express my sincere gratitude to all of them.

First and foremost, I would like to thank my advisors Dr. Cristina Gómez and Dr. Antoni Olivé. They gave me the opportunity to meet the research world, and they showed me how to enjoy researching. Their rigor, guidance, patience, and support has been crucial during all these years.

I am also very grateful to my colleagues in the *Grup de Modelització Conceptual* and the *Departament d'Enginyeria de Serveis i Sistemes d'Informació*. Their comments and valuable insights on my work helped me improving the overall quality of my research. I am also indebted to the reviewers of this thesis, whose contributions have been essential in pursuing excellence.

My special thanks to all those friends that have been by my side during the past four—and in some case more than ten—years, Antonio Villegas, David Sancho, Marc Rodríguez, Mikel Fernández, Nikolaos Galanis, and many, many others. You were there when I felt down. You made my life funnier. For this, I truly thank you.

Finally, I want to thank my family. A lot of things happened during these past years, but we overcame all problems. Thank you mom, dad, and Marc, and thank you María. You always believed in me. I took your strength and determination whenever mine was missing.

This work has been partly supported by
*Ministerio de Ciencia y Tecnología*, TIN2008-00444 project (*Grupo Consolidado*).
*BarcelonTech* (UPC), *FPI-UPC* program.

# Abstract

The modern world is software-intensive. National infrastructures, smartphones and computers, health-care systems, e-commerce—everything is run by software. Therefore, developing *high-quality software* solutions is essential for our society.

Conceptual modeling is an early activity of the software development process whose aim is to define the conceptual schema of a domain. As the role played by conceptual schemas in software development becomes more relevant—because of, for example, the emergence of model-driven approaches—, their quality becomes crucial too. The quality of a conceptual schema can be analyzed in terms of "quality properties". All conceptual schemas should have the fundamental properties of syntactic and semantic correctness, relevance and completeness, as well as any other quality property that has been proposed in the literature and that may be required or recommended in particular projects.

It is a fact that only a few quality properties have been integrated into the development environments used by professionals and students, and thus enforced in the conceptual schemas developed by them. A possible explanation of this unfortunate fact may be that the proposals have been defined in the literature in disparate ways, which makes it difficult to integrate them into those environments. The goal of this thesis is to ease the integration of those quality properties that can be evaluated using the conceptual schema itself.

We propose a method that permits the unified definition and treatment of *conceptual schema quality issues*, which we understand as "important quality topics or problems for debate or discussion". Our work includes, on the one hand, a *characterization* and *formalization* of conceptual schema quality issues, and, on the other hand, the creation of a *catalog of quality issues* obtained from the literature and defined using the aforementioned formalization.

We also provide a *prototype implementation of our method*, which integrates the catalog of quality issues on top of a real modeling tool. This implementation provides assistance to conceptual modelers during the development of a conceptual schema in a non-disruptive manner. Moreover, our thesis discusses *incremental methods for the efficient evaluation of OCL* expressions in the context of quality issues and integrates one of them into our prototype tool.

# Contents

# List of Figures

# Part I

# Preface

All human wisdom is summed up in these two words: wait and hope.

A. Dumas, *The Count of Monte-Cristo*

# 1 Introduction

Modeling is hard. The definition of good models is especially hard. The main goal of this thesis is to improve the quality of conceptual schemas. In order to do so, we provide a method to uniformly define and treat *conceptual schema quality issues*, which we understand as "important [*quality*] topics or problems for debate or discussion"[1] in the field of conceptual modeling. Therefore, our proposal aims to point out the quality issues a conceptual schema contains so that the conceptual modeler becomes aware of them and, ultimately, she can solve them. These quality issues may be, for example, violating integrity constraints, having non-satisfiable associations, or not following best practices, among others.

Throughout the chapters of the work at hand we introduce the notion of quality in conceptual modeling, its relevance, and how it can be achieved; we present and describe the different elements of our method's formalization; we address and discuss the importance of an incremental evaluation of some quality

[1]Oxford dictionaries (`http://oxforddictionaries.com/`).

issues to obtain instant feedback; and we outline the different artifacts we have built, including an extensive quality issue catalog that contains over 60 state-of-the-art issues (formalized as our method requires), and a prototype tool that implements this catalog into a modeling tool so that the tool can assist conceptual modelers.

This chapter introduces our work and presents the research approach we used, as well as the structure of the dissertation. Section 1.1 starts with a description of the motivation and the antecedents of this thesis, which deals with the *quality* and the *understandability* of conceptual schemas. Next, Section 1.2 introduces the basic notions and guidelines of the *design-science research methodology* and relates them with the different stages of our work. Section 1.3 enumerates and describes the different contributions of this thesis and the goals it tries to fulfill. Finally, Section 1.4 shows an overview of the structure of this document, including a brief description of each remaining chapter and a reading guide.

## 1.1 Motivation and Antecedents

We cannot run the modern world without software [114, p. 4]. National infrastructures, health-care systems, home-entertainment systems, communications, the film industry, our smartphones and computers—everything is software-intensive. Hence, software engineering has become essential for our society.

*Engineering* is about getting results of the *required quality* within the schedule and budget [114, p. 8]. The most effective way to do so is to adopt a systematic and organized approach. Fortunately for software engineers, there are several methods and techniques in the literature that aim to develop high-quality software, according to user expectations, quality criteria, estimated costs, and within the schedule.

*Conceptual modeling* is an early activity of the software development process. Its aim is to gather, organize, and classify the relevant, general information of a domain, and represent it as a *conceptual schema* [97, 133]. Conceptual schemas are a key component in information system development, since they provide an abstraction of the real-world.

Traditionally, conceptual schemas have only been used for documentation purposes, and were rarely maintained up to date with the implementation. Nonetheless, the emergence of model-driven approaches is changing this trend by increasing the role conceptual schemas play [12, 68, 110]. A Model-Driven Development (MDD) paradigm [78, 99] (or Model-Driven Architecture (MDA) according to the Object Management Group (OMG)) promotes the automatic generation of the system implementation based on its model. As a consequence, several techniques and methods that were usually applied to code—for example, *testing*—may now be applied to conceptual schemas [123, 124, 125].

As the role played by conceptual schemas in software development becomes more relevant, assessing their quality becomes crucial. Experience demonstrates that low quality leads to failures. On January 28, 1986, the NASA Shuttle Challenger exploded shortly after launch, destroying the vehicle and all crew members. The destruction of the Shuttle was caused by the hardware failure of a solid rocket booster (SRB) "O" ring. On August 2, 2012, the Knight Capital Group (an American global financial services firm) announced that it lost $440 million when it sold all the stocks it accidentally bought the previous day because "a computer glitch".

These two examples demonstrate that human errors are unavoidable. However, one can diminish the errors that may occur by analyzing the quality of the product before it is released. In the field of conceptual modeling, there are empirical studies showing that more than half the errors that occur during system development are requirements errors [47, 73]. As a consequence, it is clear that addressing quality issues during the development of a conceptual schema is our best choice. Unfortunately, software quality has traditionally focused on evaluating the final product [87], not the schema. As a result, evaluating the quality of a conceptual schema looks more like an "art" than an engineering discipline [85].

There are several quality frameworks in the literature whose aim is to evaluate the quality in the conceptual modeling stage. Some of these frameworks focus on the *quality of data models*, whilst others focus on the *quality of the modeling process*. Moreover, there are many analytical methods that can be used to validate and verify software artifacts. Nowadays, conceptual modelers need to master these techniques so as to guarantee the quality of their products.

It is clear that assessing the quality of conceptual schemas is very important [9, 30]. In order to conduct this task effectively, conceptual modelers require the assistance of modeling tools that implement facilitate the validation and testing of conceptual schemas [98]. In [52], the authors present a research roadmap for model-driven development of complex software. In particular, they identify the challenge of "having quality assurance programs" for the effective modeling of domains using Domain-Specific Languages. In [119], the authors summarize some Model-Driven Engineering challenges that were identified during a workshop. Quality in conceptual modeling is one of these challenges and, specifically, the importance of verifying and validating (partially incomplete) models automatically and incrementally. In [18], the authors point out the kinds of modeling errors that novice analysts are most likely to produce and discuss the importance of developing checklists and guidelines for quality assurance teams. Once this step is done, they also propose developing validation procedures, creating instruments for measuring quality from different perspectives, and integrating all this knowledge into current modeling tools.

The purpose of this thesis is to define a method that permits the definition and treatment of conceptual schema quality issues in a uniform manner. A quality issue might be any topic or problem that has an impact in the resulting quality of a conceptual schema and can be evaluated using the schema solely. The formalization included in our method can be used to describe *when* a conceptual schema contains an issue that requires the modeler's attention and *how* the issue can be tackled and fixed. Moreover, if the method is integrated within a modeling tool, then conceptual modelers can benefit from an automated real time support and assistance during the development of their schemas.

The general problems addressed in this thesis are:

- the *characterization* and *formalization* of conceptual schema quality issues,
- the *creation of a catalog* of uniformly-defined quality issues, available to practitioners and tool developers, and
- the *improvement on addressing quality issues* during the development of a conceptual schema by automating its *detection* and suggesting *which actions fix them*.

## 1.2 Research Approach

The work presented in this thesis is structured following the main ideas of the Design Science Research methodology. This methodology is a problem-solving paradigm based on the *creation* and the *evaluation* of *artifacts* whose purpose is to solve identified problems. In other words, it involves the analysis of the use and performance of designed artifacts to understand, explain, and very frequently improve on those artifacts [128]. As stated by Hevner et al. in [62], "the fundamental principle of design-science research is that knowledge and understanding of a design problem and its solution are acquired in the building and application of an artifact". That is, design-science research requires the *creation* of an innovative, purposeful artifact for a specified problem domain which yields utility for the specified problem.

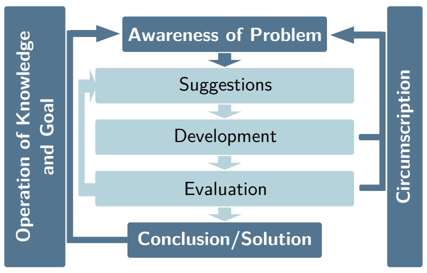


**Figure 1.1.** Reasoning on Design Cycle.

Figure 1.1 illustrates the course of a general design cycle, as Takeda et al. analyzed in [121]. The cycle begins by being *aware of the problem* to be addressed. Next, *suggestions* to solve this problem are abductively drawn using the existing knowledge available. Finally, an artifact that partially or fully implements the proposed solution is built during the *development* stage so that it can be *evaluated*. The *conclusion* indicates the termination of a project. *Development*, *evaluation*, and, sometimes, further *suggestion* stages are frequently performed iteratively. At each iteration, expertise and knowledge about the problem at hand is gained, and the quality and the accuracy of the proposed solutions can thus be improved.

In [62], the authors proposed the following guidelines for Design Research in Information Systems Research:

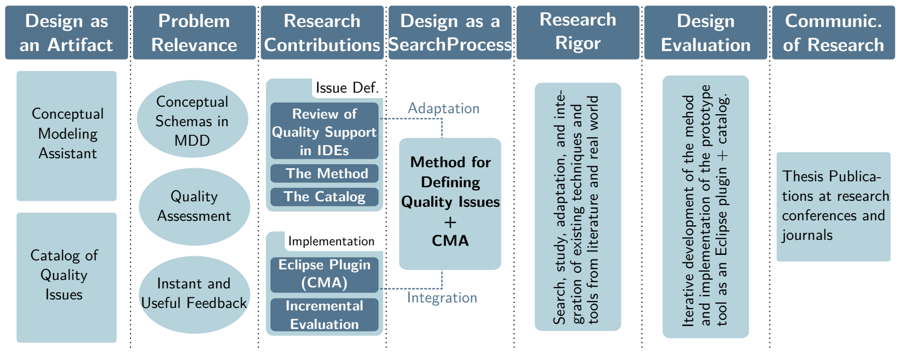

Design as an Artifact
Problem Relevance
Research Contributions
Design as a SearchProcess
Research Rigor
Design Evaluation
Communic. of Research
Conceptual Modeling Assistant
Catalog of Quality Issues
Conceptual Schemas in MDD
Quality Assessment
Instant and Useful Feedback
Issue Def.
Review of Quality Support in IDEs
The Method
The Catalog
Implementation
Eclipse Plugin (CMA)
Incremental Evaluation
Adaptation
Method for Defining Quality Issues + CMA
Integration
Search, study, adaptation, and integration of existing techniques and tools from literature and real world
Iterative development of the mehod and implementation of the prototype tool as an Eclipse plugin + catalog.
Thesis Publications at research conferences and journals

**Figure 1.2.** Design Research in this Thesis.

i. **Design as an Artifact**. Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.

ii. **Problem Relevance**. The objective of design-science research is to develop technology-based solutions to important and relevant business problems.

iii. **Design Evaluation**. The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.

iv. **Research Contributions**. Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.

v. **Research Rigor**. Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.

vi. **Design as a Search Process**. The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.

vii. **Communication of Research**. Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Figure 1.2 shows the application of design science research in this thesis. As we can see, the thesis follows the guidelines presented above because (i) in Chap. 7 we present a prototype tool that solves the problem of improving the quality of conceptual schemas, and in Chap. 6 we present a public catalog of conceptual schema quality issues; (ii) the problem of quality in conceptual modeling is relevant, as we outline throughout Chapters 1, 2, 3; (iii) we evaluate our results from different perspectives (Chapter 6 and Section 8.3.4 and 9.2); (iv) and (v) our work is clearly based on the existing knowledge in the field of quality in conceptual modeling (Chapters 3, 6, and 9) and in the field of the efficient evaluation of integrity constraints and OCL expressions (Chap. 8); (vi) our results were obtained and improved iteratively; and (vii) we communicate our results to the community for their evaluation and reusal (Sect. 10.3).

## 1.3 Research Contributions

As we have mentioned in Sect. 1.1, conceptual schemas are becoming one of the most relevant artifacts in software development. Therefore, their quality is a very important facet that has to be considered after and probably during its creation. As we shall see in Chap. 3 and Chap. 6, there are several frameworks, methods, and proposals that deal with the quality of conceptual schemas, making it difficult for conceptual modelers to know and apply them all. In order to simplify their work, we propose a method that permits the unified definition and treatment of the relevant criteria a conceptual schema has to fulfill. In this sense, our method is two-folded: on the one hand, it provides a formalization for defining conceptual schema quality issues to whoever wants to define a new type of issue. On the other hand, it describes how to detect the issues a conceptual schema has and how they change as the schema evolves.

The main goal of this thesis is to **improve the quality of conceptual schemas**, which is defined as the degree up to which a set of properties are met. In particular, we focus on the properties that can be evaluated looking at the schema itself—i.e. **syntactic correctness** of the conceptual schema, as well as on the proper application of any **guidelines** and/or **best practices** that are required by the company or project in which the conceptual schema is defined.

Our method for defining and treating conceptual schema quality issues can be outlined in the following sub-goals:

- **Study and characterization of several quality criteria.** The criteria we address include correctness, completeness, best practices, and guidelines. We are interested in defining and treating them uniformly in terms of quality issues—i.e. *when* there are issues regarding any of these criteria and *how* these issues can be fixed.

- **Creation of a catalog of quality issues.** There are several quality issues published in the literature. We want to formalize as many of them as possible[2] using our formalization and present the results in a public catalog. Such a catalog would be useful to practitioners who want to assure and improve the quality of conceptual schemas, and to any engineer implementing a modeling tool that is interested in integrating into the tool an automatic assistance to conceptual modelers. In this thesis, we focus on conceptual schemas written in UML/OCL and, thus, the quality issues we deal with are targeted at this language. However, the method could be applied to both Domain Specific Languages (DSL) and other conceptual modeling languages.

- **Implementation of a prototype modeling tool that demonstrates the usefulness of our method.** In order to demonstrate the usefulness of our method, we integrate the catalog of quality issues within a modeling tool. We believe that quality issues are really useful when they are addressed during the development of a conceptual schema. The easiest way to do so is to have them implemented within a modeling tool that automatically monitors the issues the conceptual schema has. Implementing a prototype tool serves this purpose perfectly, and permits us to discuss different aspects of the catalog integration, such as how to provide a real-time assistance during the development of a conceptual schema.

[2](rev3.16) We do not aim to create a complete catalog that includes *all* the quality issues available in the literature. The catalog presented in the thesis is a first step in this direction and its aim is to demonstrate the descriptive validity of our method.

### 1.3.1 Characterization and Formalization of Conceptual Schema Quality Issues

As we have already mentioned, one of the most effective ways to get a conceptual schema of the *required quality* within the schedule and budget is to adopt a systematic and organized approach [114, p. 8]. However, evaluating the quality of a conceptual schema looks more like an "art" than an engineering discipline [85]. This thesis aims to solve this problem. Our first contribution is a *characterization* and *formalization* of conceptual schema quality issues that permits us to uniformly define and treat them. We understand a quality issue as an "important [*quality*] topic or problem for debate or discussion" in the field of conceptual modeling. Quality issues may be, for example, violating integrity constraints, having non-satisfiable associations, or not following best practices.

Our proposal addresses the following aspects:

1. **How and who *defines* a quality issue.** We characterize and formalize the notion of quality issue. As we introduce in Chap. 4 and describe in depth in Chap. 5, a quality issue has to specify which type of elements—like entity types, relationship types, or attributes, among others—may present a certain type of issue, how we can detect in a schema the elements of that type that actually have an issue, which kinds of issues exist, how issues may be classified, and so on.

2. **How and when we may *detect* the issues a conceptual schema contains.** Our proposal includes an algorithm to compute the issues a conceptual schema has (see Sect. 5.4). Our method aims to detect quality issues as they arise during the development of a conceptual schema, but it also permits the on-demand detection of quality issues.

3. **How issues change as the schema evolves.** Issues appear and disappear as the conceptual modeler changes the schema. The method describes the different states in which a particular issue may be and how the changes over the schema affect issues' states.

4. **How and when the conceptual modeler may tackle issues.** Not only detecting issues is important, but also tackling them. For each issue, our

formalization includes one or more actions the modeler can take in order to fix it.

5. **How to *efficiently* detect the issues a conceptual schema contains and thus provide *instant feedback*.** Chapter 8 presents an improved version of the algorithm that computes issues efficiently, using an incremental approach.

### 1.3.2 A Catalog of Conceptual Schema Quality Issues

The second most important contribution of this thesis is a review of the quality criteria we may found in the literature. These criteria include, as we have already mentioned, the fundamental properties of syntactic correctness, as well as several best practices and guidelines that have been proposed during the past years in modeling-related conferences, journals, and books. These criteria are then defined in terms of quality issues and compiled in the first version of a free-to-access, public catalog. Obviously, this catalog can and should be extended so that it includes most, if not all, available quality issues. We focus on conceptual schemas written in UML/OCL and, thus, the quality issues we deal with are targeted at this language. However, the method could be applied to both DSLs and other conceptual modeling languages.

We believe the modeling community can benefit a lot from such a catalog. On the one hand, it can be used as a quality reference guide by practitioners who want to assure and improve the quality of conceptual schemas. On the other hand, it can be used by any engineer that is interested in integrating into her modeling tool the automatic assistance our method puts forward.

### 1.3.3 A Conceptual Modeling Assistant

The guidelines of the design-science research, which we discussed in Sect. 1.2, require an artifact to evaluate the benefits of the approach under test. In this thesis we have built two artifacts: on the one hand, there is the catalog of quality issues, which we have already commented. On the other hand, we implemented a prototype tool that integrates this catalog and our method within a real Integrated Development Environment (IDE).

We built our prototype tool as an Eclipse[3] plugin. The plugin is responsible of downloading a catalog of OCL-defined quality issue types from a remote server and, then, evaluating the conceptual schema in order to determine the issues it has. Moreover, it offers the list of actions that may solve each particular issue.

This plugin transforms Eclipse in what we call a Conceptual Modeling Assistant (CMA). A CMA is responsible of detecting issues automatically and of notifying the modeler. Chapter 7 describes this prototype implementation in detail. Moreover, in Chap. 8 we discuss the importance of using incremental methods for providing instant feedback and we outline the integration of an incremental method into our CMA prototype. As a result, the detection of issues and the feedback generation becomes instant—i.e. the modeler has information on the issues she is introducing in or fixing from the schema in real time.

## 1.4 Overview of the Thesis

Figure 1.3 depicts the structure of this thesis. The thesis contains ten chapters which are organized in three parts:

**Part 1: Preface** It only includes Chap. 1. There, we introduce the motivation of this thesis, the research approach we used, the goals we pursue, and the main contributions. It also includes a brief overview of the rest of the thesis.

**Part 2: Background on Conceptual Modeling and Quality** This part includes Chapters 2 and 3. This part introduces some relevant concepts about conceptual modeling and metamodeling, as well as some related work on quality in conceptual modeling, including quality frameworks and quality properties.

**Part 3: Contributions** It includes all the remaining chapters of this thesis. First, Chapter 4 outlines the method we propose. Next, Chapters 5, 6, and 7 present the main contributions of the thesis in depth: (1) the formalization of quality issues, (2) the catalog of quality issues, and (3) a prototype

[3]Eclipse is an open-source IDE whose core can be extended easily using a plugin system. Using the appropriate plugins, it can be used as a UML modeling environment.

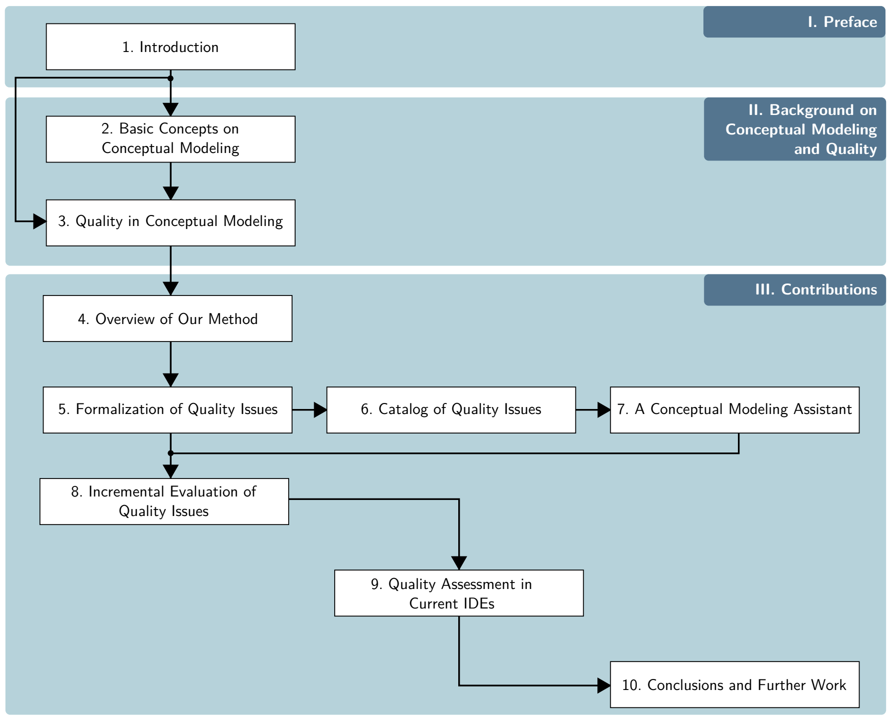

I. Preface
1. Introduction
II. Background on Conceptual Modeling and Quality
2. Basic Concepts on Conceptual Modeling
3. Quality in Conceptual Modeling
III. Contributions
4. Overview of Our Method
5. Formalization of Quality Issues
6. Catalog of Quality Issues
7. A Conceptual Modeling Assistant
8. Incremental Evaluation of Quality Issues
9. Quality Assessment in Current IDEs
10. Conclusions and Further Work

**Figure 1.3.** Structure of the thesis.

tool that implements our method. Chapter 8 describes how issues can be detected efficiently during the development of the conceptual schema, providing instant feedback to the conceptual modeler. In Chap. 9 we analyze several IDEs that are being used by practitioners in order to determine up to which extent they support modelers in assessing quality. Finally, in Chap. 10 we expose our conclusions and we outline future work.

In the following, we describe the contents of each chapter in more detail.

## Chapter 2. Basic Concepts on Conceptual Modeling

In this introductory chapter we present conceptual modeling as one of the most important activities in software development, especially since the arise of model-driven development approaches. We briefly introduce the main components of conceptual schemas, as well as some basic notions on metamodeling. It is important to understand these concepts before reading the rest of the thesis.

## Chapter 3. Quality in Conceptual Modeling

This chapter reviews the state of the art in quality assessment. On the one hand, we focus on some of the main quality frameworks proposed in the literature for evaluating the quality of software artifacts. On the other hand, we review some related work on actually improving the quality of conceptual schemas. The review explores a wide range of proposals, including *inconsistency management*, *best practices*, *naming guidelines*, *refactorings*, and *tool support*. The chapter also points out the need of creating a method to uniformly define and treat many of the proposals explored here.

## Chapter 4. Overview of Our Method

This chapter presents an overview of our method. In order to do so, it includes a motivating example that serves as an illustration of the quality issues we deal with in this thesis. Next, we informally present the definition and formalization of conceptual schema quality issues. Finally, we describe the three different phases of our method, from which we derive the main contributions of our work: (i) defining quality issues, (ii) compiling a catalog of quality issues, and (iii) using quality issues during the development of a conceptual schema.

## Chapter 5. Formalization of Quality Issues

We present the main contribution of our thesis in this chapter—i.e. the concepts of *quality issue type* and *quality issue instance*. We formally describe the different components of an *issue type*, which basically include *when* a conceptual schema has issues of a certain type, *how* these issues are affected by the changes performed over the conceptual schema, and *which actions* have to be taken in order to solve these issues. We also present an algorithm that, given a conceptual schema and a set of issue types, computes the issues present in the schema. This algorithm can be used in “real-time” during the development of the schema, providing instant[4] feedback to the modeler on the issues she has to consider, or as a batch process at the end of the development of the schema. The evaluation of our method is presented throughout the subsequent chapters of the thesis in terms of its *expressiveness* and *usefulness*.

## Chapter 6. Catalog of Quality Issues

Another important contribution of the thesis is the creation of a comprehensive catalog of quality issues. This chapter presents a review of the literature, looking for issues in modeling journals, conferences, and books. The issues are defined for UML-defined conceptual schemas, although some of them may apply to other modeling languages. The catalog demonstrates the descriptive validity of our method. Moreover, it also explores the usefulness of the quality issues it includes. We argue that such a catalog, which may be extended in future iteration, may help conceptual modelers and practitioners, as well as IDE engineers who want to integrate issues into their tools.

## Chapter 7. A Conceptual Modeling Assistant

This chapter presents the prototype tool we have built to demonstrate the feasibility and usability of our method. We conceived our prototype tool as a plugin for Eclipse (an open-source IDE that can be used as a modeling tool). The prototype implements the detection of conceptual schema quality issues, providing feedback that points out the issues a conceptual schema has.

[4]We can only provide *instant* feedback if we are able to evaluate quality issues quickly. When there are many issue types and the schema is big enough, the time required to perform the computation would be to high to be considered “instant”. This issue is addressed and discussed in Chap. 8.

## Chapter 8. Incremental Evaluation of Quality Issues

As we have already stated, we want our method to be used during the development of a conceptual schema. If it is, conceptual modelers can get real-time feedback on the issues their schemas have. Therefore, it is necessary that our method is able to detect issues quickly so that it can provide instant feedback.

In this chapter we first review the state of the art related to the efficient and incremental evaluation of OCL expressions and metamodel constraints. Incremental methods aim to re-evaluate an expression if, and only if, the result of that expression may have changed. Next, we adapt one incremental proposal to our method and we explore how some quality issue types (in particular, those that are defined in our catalog using OCL) can be evaluated incrementally. Finally, the chapter also includes an experimental evaluation that compares the efficiency of our method in detecting quality issues with and without the incremental evaluation.

## Chapter 9. Quality Assessment in Current IDEs

Integrated Development Environments (IDEs) can help conceptual modelers assessing quality. Usually, modeling tools only focus on creating syntactically correct schemas. However, they can also be used to automatize the process of creating a conceptual schema, especially when it comes to analyzing its quality and detecting its flaws. In this chapter we review some tools that are being used by practitioners nowadays and we analyze the support their offer with regard to quality. This review provides a better understanding on how modelers can be assisted and which areas can and should be improved. Finally, in this chapter we demonstrate that our catalog leads to the detection of more quality issues compared to the issues current modeling tools are able to detect.

## Chapter 10. Conclusions and Further Work

This chapter presents the conclusions of the work at hand. It summarizes the main contributions of this thesis, and points out some insights and future work on how to continue our research line. It also enumerates the main publications related to the thesis we published during these past years.

# Part II

# Background on Conceptual Modeling and Quality

The best books... are those that tell you what you know already.

G. Orwell, *1984*

# 2 Basic Concepts on Conceptual Modeling

This thesis aims to contribute to the challenge of developing high-quality conceptual schemas. In order to do so, we propose a method that allows the treatment of the quality issues that appear in a conceptual schema. Before we can discuss quality in the conceptual modeling domain and how we characterize and formalize quality issue, an introduction to conceptual modeling seems appropriate.

Therefore, in this chapter we introduce conceptual modeling and we clarify our vision on conceptual schemas. Moreover, we also introduce some basic concepts on metamodeling that will be necessary to understand how quality issues are defined. Section 2.1 outlines the importance of conceptual modeling in the software development process and introduces the main artifact it delivers—i.e. the conceptual schema. In Sect. 2.2 we describe the key parts of a complete conceptual schema, taking into account both the structural and the behavioural (sub)schemas. Section 2.3 introduces the modeling languages and presents two examples: the Entity-Relationship (ER) model and the Unified Modeling Language (UML). The UML is the language we will be using in this thesis for defining conceptual schemas. Next, Section 2.4 introduces some basic concepts on metamodeling in general, and the UML metamodel in particular. Understanding metamodeling and the UML metamodel is important because, as we shall see in Chap. 5, quality issues are defined at the metamodel level. Finally, Section 2.5 summarizes the chapter.

## 2.1 Conceptual Modeling

*Conceptual modeling* is an early activity of the software development process, closely related to requirements engineering. Its aim is to gather, organize, and classify the relevant, *general* information of a domain into a *conceptual schema*, so that a system implementing that schema can maintain *concrete* information about the domain [97, 133]. The conceptual schema is thus a simplified representation of a complex reality that makes the understanding of that reality easier.

Conceptual schemas are a key component in information system development, since they provide an abstraction of the real-world. According to the *principle of conceptualization*, this abstraction "should only include conceptually relevant aspect (both static and dynamic) of the domain, and exclude all aspects of data representation, physical data organization, and access" [64]. In other words, conceptual schemas are much less bound to any underlying implementation technology and much closer to the problem domain [110], making the understanding of a domain easier for developers and stakeholders.

Generally, a conceptual schema comprises a *structural schema* and a *behavioural schema*. The former consists of a set of entity and relationship types. It is usually known as the *static component* of the general knowledge, and it allows maintaining a consistent representation of the state of the domain. This representation is accessible at any moment of the system's lifetime, and it is called the *Information Base*. The latter represents the valid changes in the domain state as well as the actions the system can perform [97].

Classically, conceptual schemas have been used for documentation purposes only. However, as model-driven approaches emerge and gain importance, they are becoming the primary artifact in all phases of the development life-cycle [12, 68, 110]. A Model-Driven Development (MDD) paradigm [78, 99] (or Model-Driven Architecture (MDA) according to the Object Management Group (OMG)) promotes the automatic generation of the system implementation based on its model.

In the MDA paradigm there are two kinds of models. On the one hand, Platform-Independent Models (PIM) provide a formal specification of the structure and behaviour of a system by avoiding any technical detail. On the other hand, Platform-Specific Models (PSM) specify the system in terms of the implementation constructs that are available in one specific implementation technology. The goal of an MDA approach is to automatically transform a PIM into a PSM, making the conceptual schema (i.e. a PIM) the final description of the domain that is required for the development of an information system [94].

## 2.2 Conceptual Schemas

Conceptual schemas are a key component in information system development, since they provide an abstraction of the real-world. Generally, a conceptual schema comprises a *structural schema*, which deals with the static facet of the information system, and a *behavioural schema*, which defines how the knowledge stored in the information system changes during its lifetime.

### 2.2.1 Structural Subschema

Human beings use concepts to structure their perception of a domain. As a consequence, they *classify* objects into concepts. In the field of information systems, we assume that a domain consists of a number of objects and the relationships between them. As expected, these objects are classified into concepts [97, p. 10].

The *structural schema* of an information system contains the relevant concepts of that domain so that it can maintain a representation of the state of the domain at a given time. This state consists of a set of objects, a set of relationships, and the set of concepts into which these objects and relationships are classified.

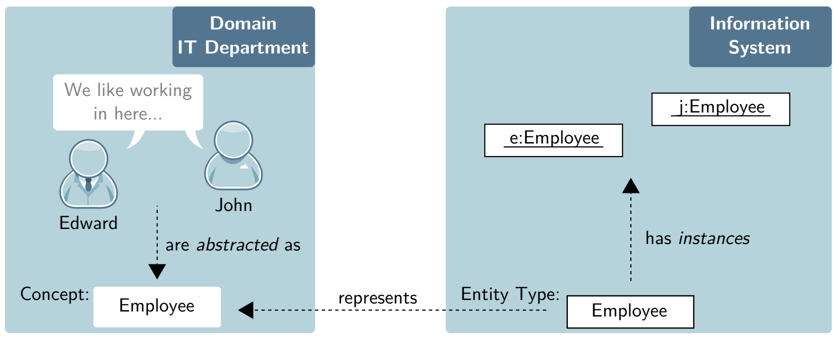


**Figure 2.1.** Representation of the structural part of an IT Department.

In conceptual modeling, a concept whose instances are unique and identifiable are called *entity types*, and its instances are simply called *entities*. An entity is always an instance of at least one entity type, but it may be the case that it is an instance of more than one entity type. Figure 2.1[1] depicts a portion of an IT Department domain. In this domain, *John* and *Edward* are entities (objects) of the entity type (concept) *Employee*.

Some concepts are associative, in the sense that their instances relate two or more entities [97, p. 13]. *Relationship types* are concepts whose instances are *relationships*. For example, Figure 2.2 shows two different relationships between an *Employee* and a *Project*. On the one hand, we can see that *John* is working on a *Project* named “Eclipse”, and, on the other hand, that *Eclipse* is managed by *Edward*—i.e. *Edward* is *Eclipse*’s manager.

Relationship types are similar to entity types, because they “are also concepts whose population are individual relationships that are considered to exist in the domain at that time” [97, p. 59]. In our example, we have two relationship types between an *Employee* and a *Project*. We may call one relationship type “IsWorkingOn”, so that it abstracts the notion of “an *Employee* who is working on a *Project*”, and we may call the other one “IsManagedBy”, so that it abstracts the notion of “a *Project* is managed by one *Employee*”.

As we can see, relationship types consists of a set of $n$ participants, where

[1]There may be more entity types in this context, such as *Projects*, *Customers*, and so on. However, for the sake of simplicity we only depicted *Employees*.

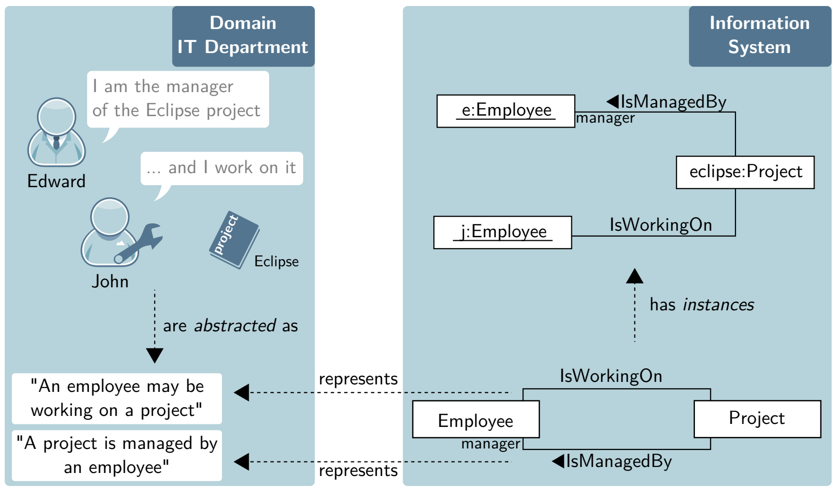


**Figure 2.2.** Examples of relationship types in the IT Department domain.

$n \geq 2$. A *participant* is no more than an entity type of the domain that plays a certain *role* in the relationship. In the previous example, we have seen that an *Employee* plays the role *manager* of a certain *Project* when she is responsible of the project's management.

Besides relationship types, structural schemas contain *attributes*. At a conceptual level, an attribute is very similar to a binary relationship type. In general, a binary relationship type relates two concepts that are equally important. In the previous example, an *Employee* cannot "be working" without an assigned *Project* nor a *Project* can "be developed" if no *Employees* are working on it. However, there are some scenarios where we may consider one participant as a *characteristic* of the other. Figure 2.3 shows an example of these characteristics: *Edward*'s name is "Edward" and that his salary is "960€/month". In this case, we may agree that both *name* and *salary* are attributes of the entity type *Employee*.

Finally, another important element for defining structural schemas are *Constraints*. In an ideal world, the information base would be an exact representation of the domain. Unfortunately, some of the facts an information base has may be invalid or incomplete. An information base has integrity when it contains all relevant facts and those facts are valid [97, p. 181]. To ensure this integrity,

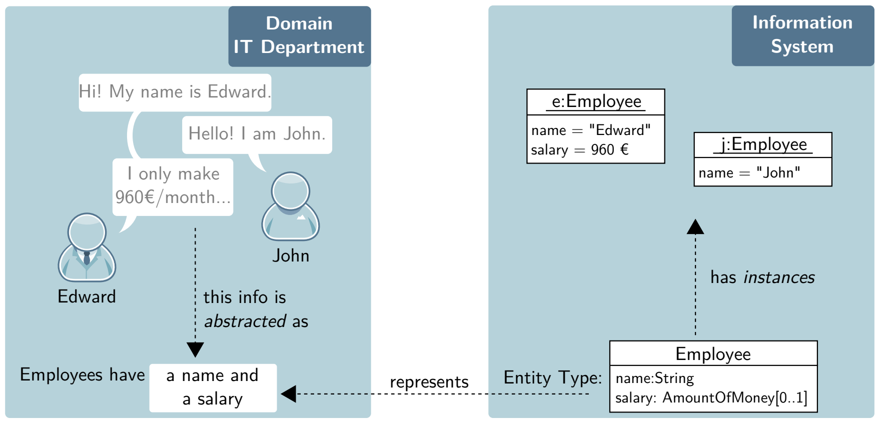


**Figure 2.3.** Examples of attributes in the IT Department domain.

we must check periodically the information base against the domain. However, it is possible to build some mechanisms—i.e. to define integrity constraints—that automatically guarantee some level of integrity. Moreover, they also help us to understand the meaning of the domain. There are many different types of constraints. Two of the most important are *Cardinality Constraints* and *Key Constraints*.

*Cardinality Constraints* constrain the population of relationship types. Imagine, for example, that a *Project* has one, and only one, *manager*, and that an *Employee* can only manage up to two different *Projects*. In this case, the binary relationship type *IsManagedBy*(*Project manager*:*Employee*) would define the following two cardinality constraints: $Card(Manages; manager, project) = (0,2)$, which states that an *Employee* may be the *manager* of up to two *Projects*, and $Card(Manages; project, manager) = (1)$, which states that a *Project* has exactly one *Employee* as its *manager*.

On the other hand, *Key Constraints* are one of the best-known constraints. A *key* of entity type *E* is a set of one or more attributes of *E* such that the mapping from the population *E* to the corresponding group of attribute values is one-to-one [97, p. 196]. A key is *simple* is it consists of a single attribute, and *composite* otherwise. Two different instances of *E* cannot have identical values for all attributes in the key. An entity type may have any number of keys.

### 2.2.2 Behavioural Subschema

As aforementioned, the structural schema defines the relevant entity and relationship types of a domain so that the information system can maintain a representation of the state of that domain at a given point in time—i.e. the information base. The facts contained in the information base are not static, but change over time according to a set of changes called *domain events*. The set of *event types* that are relevant to an information system and the effects of those event types are described in the *behavioural schema* [97, p. 18]. We may say that the definition of these event types is the most important part of the behavioural (sub)schema [97, p. 246].

According to [97, Chap. 11], domain events can be seen as entities. An *event entity type* encapsulates a set of one or more structural events—i.e. atomic changes to the information base such as creating a new instance of an entity type, “creating a new instance of a relationship type”, or “modifying an attribute”, among others. Like any other entity, event entities may participate in relationships with other entity types, they may have attributes, and they may also be included in a taxonomy of events, where one event type inherits and specializes another one.

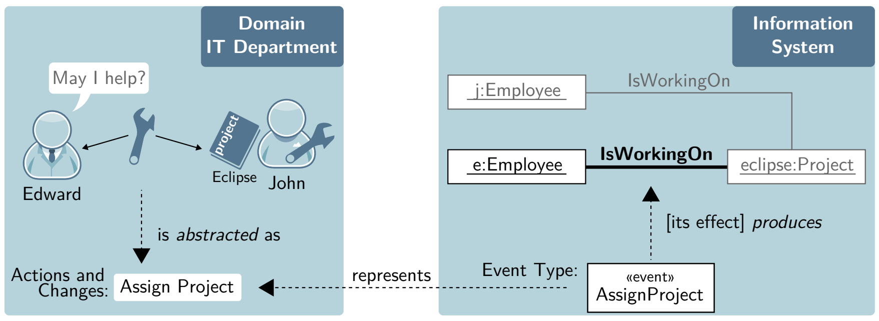


**Figure 2.4.** Representation of some event types in an IT Department.

Consider, for example, the domain of the IT Department introduced in the previous section. In that domain, where *Employees* work on *Projects*. In Fig. 2.4, we can see that *John* is already working on the *Project* “Eclipse”, and how *Edward* wants to start working on it too. In this context, we may find an event such as “assigning a *Project* $p$ to an *Employee* $e$ so that $e$ works on $p$”. This event would create a new instance of the relationship type *IsWorkingOn*(*Employee*, *Project*) is created using the instances *e* and *p*—.

More examples of event types in this domain may be "hiring a new *Employee*" (where a new instance of the entity type *Employee* is inserted in the information base), "set a new *manager* for a certain *Project* (where the relationship between the former manager and the project is deleted and a new one is created), or "firing an *Employee*" (where an instance of the entity type *Employee* is removed from the information base), among others.

## 2.3 Modeling Languages

So far we have seen that conceptual modeling is an activity whose aim is to provide a simplified representation of a complex reality (i.e. a conceptual schema) that makes the understanding of that reality easier. In order to formally represent these conceptual schemas we use modeling languages.

*Modeling languages* are artificial languages whose purpose is to represent information or knowledge about a domain. These languages, which can be graphical or textual, express the concepts we find in a domain, the relationships between these concepts, some constraints that must be satisfied by the information base, and so on.

In this section, we describe the Entity-Relationship (ER) model briefly. ER is the precursor of modern object-oriented approaches to model data and it is still widely used. Next, we introduce the Unified Modeling Language (UML) and the Object Constraint Language (OCL), which have become the de-facto standard modeling languages to graphically describe conceptual schemas. This thesis assumes UML/OCL as the modeling language to be used for defining conceptual schemas.

### 2.3.1 The Entity-Relationship (ER) Model

The Entity-Relationship (ER) model was firstly introduced by Chen in [26]. As stated in [26], "ER adopts the natural view that the real world consists of entities and relationships". Therefore, it defines a conceptual representation of data, which was formerly used for modeling databases graphically.

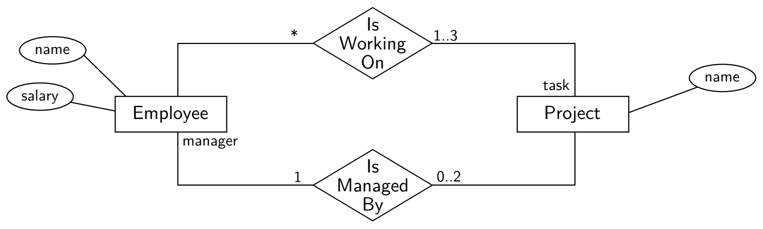


**Figure 2.5.** Example of a conceptual schema modeled using the ER.

In ER, entity types are represented as rectangles and relationship types as diamonds with lines connecting each of the relationship's participants. Entities may be thought as nouns whilst relationships may be thought as verbs that relate and connect two or more nouns [27]. ER also allows the definition of attributes, represented as ellipses connected to the entity or relationship type that owns them. It also permits the definition of some cardinality constraints. Figure 2.5 depicts the domain we presented throughout the previous sections, using the ER notation.

## 2.3.2 The Unified Modeling Language (UML)

In the last years, the Unified Modeling Language (UML) has become the de-facto standard modeling language to graphically describe conceptual schemas. UML is a standardized, graphical, general-purpose modeling language maintained by the Object Management Group (OMG). The specification of its latest version (2.4.1) can be found in [93].

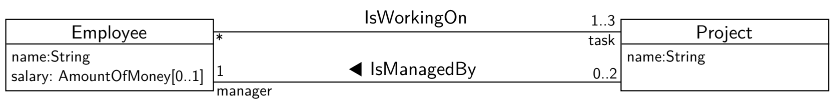


**Figure 2.6.** Example of a conceptual schema modeled using the UML.

In UML, entity types are defined as classes (which are graphically drawn as boxes that include the class name and its attributes) and relationship types are defined as associations (lines connecting classes). For example, in Fig. 2.6 we see a UML conceptual schema with two classes and two associations. The class *Employee* has two attributes (*name* and *salary*), and the class *Project* has only one (*name*). The figure also shows the associations *IsWorkingOn* and *IsManagedBy* between the entity types *Employee* and *Project*.

We have already stated that constraints are also important elements of conceptual schemas. UML permits the graphical definition of many constraints like, for example, cardinality constraints. Figure 2.6 includes several examples of cardinality constraints:

- An *Employee* has one salary (or it may be unknown).
- An *Employee* works on at least one *Project*, and on no more than three.
- A *Project* has as many *Employees* working on it as required (there is no maximum) and, in fact, it may be the case that none is working on it.
- An *Employee* may manage up to two *Projects*.
- A *Project* has exactly one *manager*.

However, there are some constraints that cannot be formally described using UML alone (like, for example, key constraints) and, therefore, a formal language to represent them is required. The Object Constraint Language (OCL) is a declarative language that complements UML and permits the description of rules in object-oriented models. Firstly introduced by IBM, the OCL is now a widely accepted standard maintained by the OMG. OCL constraints are boolean expressions targeted at a certain class, which means that they are evaluated for every single instance of that class. Since they are constraints, it is expected that the expression returns *True* for each instance.

Consider, for example, the entity type *Employee*. Assume that an employee is identified by its name (i.e. there are no two different employees with the same name). This [key] constraint can be defined in OCL as follows:

```
context Employee inv is identified by his name:
   Employee.allInstances()->isUnique(
      Tuple{name:String=name}
   )
```

## 2.4 Metamodeling

As we have already seen in Sect. 2.2, domain objects are instances of entity types. Entity types can also be seen as objects that are instances of types known as *meta entity types* [97, p. 383]. A *metaschema* is a schema that represents general knowledge about a domain that consists of a schema [97, p. 400].

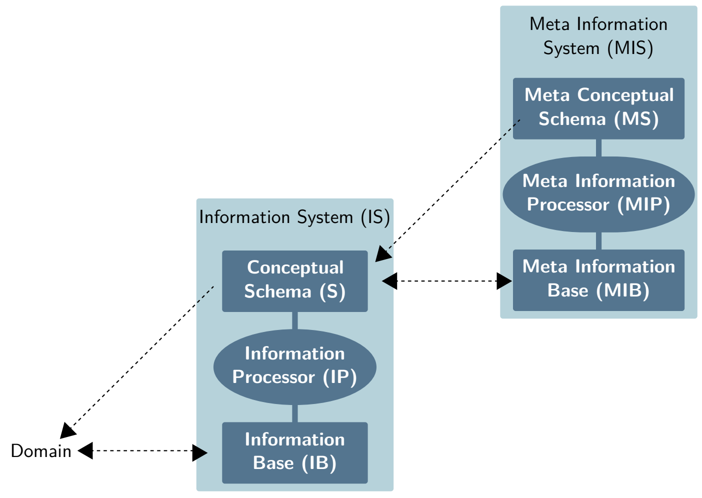

Meta Information System (MIS)
Meta Conceptual Schema (MS)
Meta Information Processor (MIP)
Meta Information Base (MIB)
Information System (IS)
Conceptual Schema (S)
Information Processor (IP)
Information Base (IB)
Domain

**Figure 2.7.** Relationships between a meta information system and an information system [97, p. 400].

Figure 2.7 shows the differences between a schema and a metaschema. The information system $IS$ contains a conceptual schema $S$ that represents the general, relevant knowledge of a domain. This general knowledge is then used to maintain concrete information of that domain in the information base $IB$. The processor $IP$ receives external messages and changes the $IB$ and/or produces an output according to these messages.

If we now look at the meta information system $MIS$, we shall see the same pattern: it contains a (meta) conceptual schema $MS$ that represents the general knowledge of a domain—i.e. the domain of conceptual schemas. This general knowledge is then used to define a concrete conceptual schema $S$, which is stored as the (meta) information base $MIB$. The processor $MIP$ receives external messages and changes the $MIB$ and/or produces an output.

A *metamodel* is a precise definition of the *constructs* and *rules* needed for creating models. Metamodels can be used as a schema for semantic data that needs to be exchanged and stored, as a language that supports a particular methodology or process, or as a language that expresses additional semantics of existing information. In order for a conceptual schema to be syntactically correct, it has to conform to this metamodel—i.e. it has to satisfy all metamodel constraints.

In this thesis, metamodeling plays a key role. As we have already outlined in Chap. 1, quality issues aim to detect any situation within a conceptual schema that may be improved. As we shall see in Chap. 5, quality issues define these situations at the metamodel level so that any conceptual schema, regardless of the specific domain, can in principle benefit form them. As a result, understanding metamodeling and, in particular, the UML metamodel is very important.

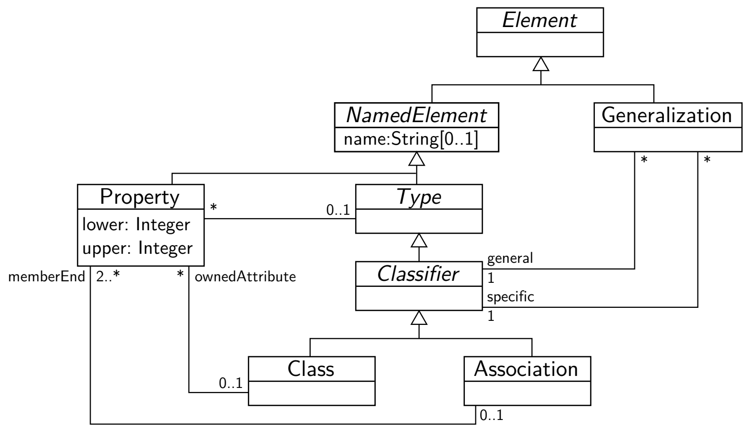


**Figure 2.8.** A simplified version of the UML metamodel.

Figure 2.8 shows a simplified version of UML's metamodel. We can see that any element in a UML schema is an *Element*. Some of these elements have an associated name, like *Class* or *Association*; this is why they are also *Named Elements*. A *Generalization*, for example, is a special kind of *Element* that relates two instances of *Classifier*. The metamodel also defines some constraints. For example, an *Association* has to be related to, at least, two *Properties*.

Figure 2.9 shows the conceptual schema[2] depicted in Fig. 2.6 as an instantiation (known as "object diagram") of the UML metamodel. As we can see, *Employee* and *Project* are instances of the UML metaclass *Class*, and the association *IsManagedBy* is an instance of the UML metaclass *Association*. The *memberEnds* of this association, as well as the *name* attribute of the class *Project*, are instances of the UML metaclass *Property*. According to [our simplified version of] the UML metamodel, *Properties* have a type, and the lower and upper cardinality constraints.

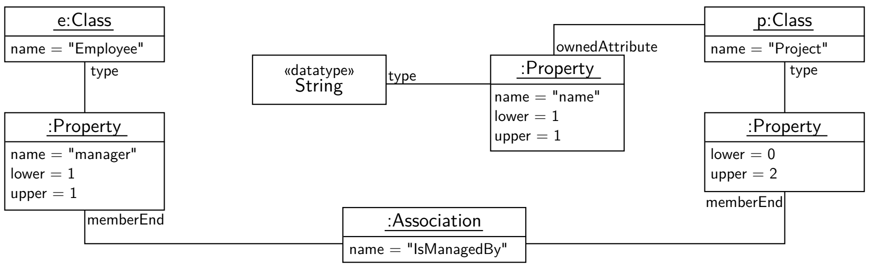


**Figure 2.9.** UML metamodel instantiation of the conceptual schema depicted in Fig. 2.6.

## 2.5 Summary

In this chapter we have presented a brief introduction to the basic concepts about conceptual modeling, conceptual schemas, and metamodeling. In particular, we have described the structural and the behavioural components of a conceptual schema, and how they can be modeled using the UML/OCL. We have also introduced a few notions on metamodeling and we have presented the UML's metamodel.

[2]For the sake of simplicity, we have omitted the association *IsWorkingOn* and all *Employee*'s attributes.

She generally gave herself very good advice, (though she very seldom followed it).

L. Carroll, *Alice in Wonderland*

# 3 Quality in Conceptual Modeling

The meaning of "quality" has been widely discussed during the past years. In general, the quality of a product is "the degree up to which a set of properties are met" [54]. Experience demonstrates that low quality leads to failures. In order to prevent an error to become a disaster, it is crucial to assure that a product reaches a certain level of quality before it is released. Unfortunately, measuring quality is very complicated.

In this chapter we review the state of the art in quality assessment, paying special attention to quality in conceptual modeling. First, Section 3.1 presents the main quality frameworks proposed in the literature for evaluating the quality of software artifacts. Section 3.2 reviews some works that describe how to address quality in conceptual modeling, which is currently a "hot topic" [55]. First, we focus on a few proposals that encourage conceptual modelers to create conceptual schemas that have inconsistencies (i.e. they are syntactically incorrect) during their development. Next, we review some quality properties that have been published in the literature, including naming guidelines and best practices. Finally, we discuss the important role tools may play in assisting conceptual modelers address quality.

## 3.1 Frameworks for Evaluating Quality

The quality of a product is the degree up to which a set of properties are met. In [54], Garvin defines the concept of *product quality* and identifies "five major approaches to its definition":

**The transcendent approach** has its basis on philosophy and sees quality as a synonymous of "innate excellence", an unanalyzable property.

**The product-based approach** views quality as a precise and measurable variable. According to this view, differences in quality reflect in differences of some attributes possessed by a product.

**The user-based approach** starts from the opposite premise: the quality lies in the eyes of the beholder. Every person has her own needs, and those goods that best satisfy her needs are those they regard as having more quality.

**The manufacturing-based approach** is concerned with manufacturing and engineering practice. It identifies quality as "conformance to requirements": once a design or specification has been established, any deviation implies a reduction in quality.

**The value-based approach** defines quality in terms of costs and prices. Thus, a quality product is one that provides performance at an acceptable price or conformance at an acceptable cost.

Software solutions and, more specifically, conceptual schemas, can also be seen as "products" and, therefore, it is possible to evaluate their quality. Traditionally, software quality has focused on *evaluating* the final product [87]. There are some international standards aimed at this end like, for example, *ISO 9000* [65] and ISO/IEC 9126 [66]. Moreover, there are some proposals in the literature devoted to evaluate the quality of specific software categories, including data quality [100, 132, 136], code quality [116], or data and process quality [89].

Although quality evaluation of final products is positive and necessary, it is not as efficient as it could be. It is widely accepted that the cost of an error increases very rapidly over the development cycle, and empirical studies show that most errors are introduced in the requirements stage [16, 85]. Fagan stated that "[in the early stages of development], the cost to remedy defects is 10–100 times less than it would be during testing or maintenance" [48]. Since errors are unavoidable, it is highly important to remove them as soon as possible in order to reduce the associated costs. Therefore, it seems necessary to be able to evaluate and improve the quality before the final product is available—i.e. we need some means to evaluate and improve the quality of a conceptual schema.

According to [85], there are no general accepted guidelines to evaluate the quality of a conceptual, which results in *ad hoc* and subjective procedures to do so. A possible explanation for this unfortunate fact may be that "it is easier to evaluate the quality of a finished product than a logical specification that needs to be aligned with the expectations of people involved in the system under development" [129].

In [91], the authors reported on an empirical investigation on the impact of proper documentation in UML on the quality of software system, focusing on defect density as a measure of software quality. They found that Java classes that were modeled using UML had lower defect density than those that were not modeled, thus proving the potential benefits of UML modeling for improving the quality of software. In the context of MDD and MDA paradigms, where models are the key artifact in the development process and not only documentation assets, their quality has an even higher impact on the quality of the delivered system. [85] suggests that any conceptual quality analysis should comply with both *ISO 9000* [65] and *ISO/IEC 9126* [66]. On the one hand, *ISO 9000* defines a framework of quality concepts, terminology, principles, and processes that apply to all software products and services and, on the other hand, *ISO/IEC 9126* defines a framework for evaluating the quality of software products and covers the software development cycle.

There are several quality frameworks in the literature whose aim is to evaluate the quality in the conceptual modeling stage. Some of these frameworks

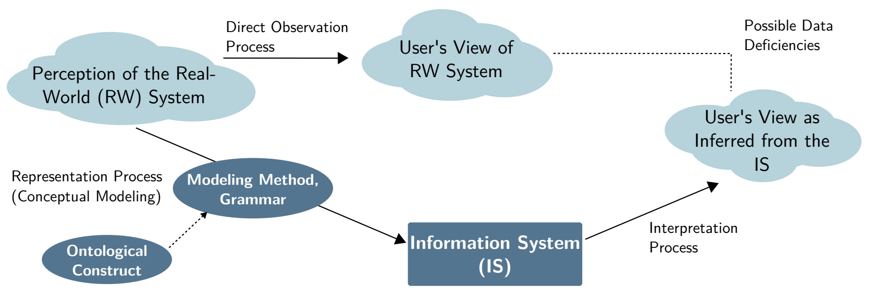

Direct Observation Process
Possible Data Deficiencies
User's View of RW System
Perception of the Real-World (RW) System
User's View as Inferred from the IS
Representation Process (Conceptual Modeling)
Modeling Method, Grammar
Ontological Construct
Information System (IS)
Interpretation Process

**Figure 3.1.** Wand and Wang's framework.

focus on the *quality of data models*, whilst others focus on the *quality of the modeling process*. An example of the former group is the framework proposed by Moody et al. in [86], which was later refined and extended in [87]. This framework proposes a set of quality factors (completeness, correctness, simplicity, flexibility, integration, understandability, and implementability), their relationships with stakeholders, their contribution to the improvement strategy, their importance, and quality measures to evaluate them. An example of the second group is the framework proposed by Wand and Wang in [132], where the quality of a product is said to "depend on the process by which the product is designed and produced". It is important to remark that there are new frameworks that combine both views [89].

The model proposed in [132] makes a distinction between the internal and the external view of an information system and the quality dimensions related to each one. The external view is concerned with the use and effect of the system, whilst the internal view addresses the construction and operation necessary to attain the required functionality. Table 3.1 categorizes the set of data quality dimensions that were identified by the authors, based on the definitions of internal and external views. Figure 3.1 shows their framework and the possible deficiencies that may be detected by a user when she compares the view she has of the domain and the view inferred from the build information system. According to Wand and Wang, the process by which the information system is built is the key factor to success.

Lindland et al. identify three main conceptual schema quality goals in [75]: *syntactic quality*, *semantic quality*, and *pragmatic quality*. The authors consider a

**Table 3.1.** Data quality dimensions as related to the internal or external views.

| | **Dimensions** |
|---|---|
| Internal View (design, operation) | **Data-related**<br>accuracy, reliability, timeliness, completeness, currency, consistency, precision<br>**System-related**<br>reliability |
| External View (use, value) | **Data-related**<br>timeliness, relevance, content, importance, sufficiency, usableness, usefulness, clarity, conciseness, freedom from bias, informativeness, level of detail, quantitativeness, scope, interpretability, understandability<br>**System-related**<br>timeliness, flexibility, format, efficiency |

conceptual schema as a set of statements in a modeling language, whose foundations are in the theory of semiotics [101]. Semiotics is related to the field of linguistics and includes the evaluation of codes and signs based on tree main points of view, that correspond to the goals identified by the authors: syntactic view (relations among signs in formal structures), semantic view (relations between signs and the concepts they refer), and pragmatic (relation between signs and the "effects" they have on the people who use them). The SEQUAL framework (presented in [70]) extends "Lindland's" and separates quality goals from quality means (or quality types). Table 3.2 presents an overview of the framework.

## Quality Dimensions Addressed in this Thesis

According to the framework presented in [89], the main quality properties addressed in this thesis correspond to the *physical layer*, which contains the observable elements of the quality framework. In particular, our method aims to improve the *empirical quality* and the *syntactic quality* of the conceptual modeling process.

On the one hand, *syntactic quality* is defined as "the correspondence between

**Table 3.2.** SEQUAL Quality Framework.

| Quality Type | Goals | Description |
|---|---|---|
| **Physical** | Externalization | The conceptual model is available as a physical artifact, representing the knowledge of some social actor using statements of the modeling language. |
| | Internalizability | The conceptual schema is available and persistently enabling the audience to interpret it. |
| **Empirical** | Minimal Error Frequency | The conceptual schema can be evaluated looking the schema itself, comprising comprehensibility matters such as layout for graphs and readability indexes for text. |
| **Syntactic** | Syntactic Correctness | All statements in the conceptual model schema according to the syntax and vocabulary of the modeling language. |
| **Semantic** | Feasible Validity | All the statements in the conceptual schema are (sufficiently) correct and relevant to the problem. |
| | Feasible Completeness | The schema contains all valuable statements that would be correct. |
| **Perceived Semantic** | Perceived Validity and Completeness | The model is valid and complete, according to the actors' interpretation of the schema and their knowledge about the domain. |
| **Pragmatic** | Feasible Comprehension | It is the correspondence between the model and the audience's interpretation of the model. In other words, the model has to be understood. |
| **Social** | Feasible Agreement | It implies resolving inconsistencies by choosing alternatives where benefits of choosing exceed the cost of working out consensus. |
| **Organizational** | Modeling Goals Satisfaction | The model has to fulfill the goals of modeling, which define why the conceptual modeling process is undertaken. |
| **Knowledge** | Feasible Knowledge Validity and Completeness | Validity and completeness taking into account the audience's knowledge about the domain. |

the conceptual schema representation and the language in which it is written". As we describe in Chap. 5, our proposal encourages working with syntactically incorrect schemas whilst they are being developed, as long as the errors are fixed at some point. Therefore, evaluating the syntactic quality becomes crucial.

On the other hand, *empirical quality* includes any quality property that can be evaluated by looking at the schema itself. These properties comprise comprehensibility matters such as readability indexes for text. The next sections include several quality properties, such as best practices and naming guidelines, that directly contribute this SEQUAL's quality dimension.

## 3.2 Related Work on Improving Quality

Software systems are always validated by their users once the system is delivered and they start using it. In general, any software artifact—and, in particular, conceptual schemas—may be *verified* and *validated* according to a set of quality criteria defined in a specific quality framework. There is a wide range of properties that can be validated and contribute to different quality dimensions, as well as several activities and methods aimed to perform this validation.

In this section, we first review some works that describe how to address syntactic quality. These works encourage conceptual modelers to create conceptual schemas that have inconsistencies (i.e. they are syntactically incorrect) during their development. Next, we review some quality properties that have been published in the literature. These properties include naming guidelines and best practices. Finally, we discuss the important role tools may play in assisting conceptual modelers address quality. In this sense, we briefly outline the support that tools offer in other domains (such as, for example, programming environments), and the support conceptual modelers get nowadays.

### 3.2.1 Working with Inconsistencies

In Sect. 2.4, we have introduced metamodeling, and we have seen that, for a conceptual schema to be syntactically correct, all metamodel constraints have to be satisfied. The violation of a metamodel constraint is known as an *inconsistency*. Classically, inconsistencies are forbidden—i.e. the modeler cannot perform any change such that her conceptual schema becomes inconsistent. However, some authors consider inconsistencies inevitable and, therefore, they encourage working with them, as long as they get fixed sometime in the (near) future [44, 49, 115]. The rationale is that developing a conceptual schema is a creative process and, as such, inconsistencies inevitable arise whilst exploring different alternatives, or simply because of different stakeholders having different (inconsistent) views of the system under development. When inconsistencies are allowed, inconsistency handling becomes a key piece in the development of a schema.

In [15, 112], Blanc, Silva, et al. propose an incremental consistency checker based on the idea of representing models as sequences of primitive construction operations. The four elementary operations they define are: *create*, *delete*, *setProperty*, and *setReference*. In order to detect an inconsistency, they define *Inconsistency Detection Rules*. Any inconsistency rule is a logic formula over the sequence of model construction operations: if a set of operations is triggered in a specific order, it can be assured that an inconsistency has been introduced into the model.

In [43, 44], Egyed proposes an instant consistency checking for the UML. Its immediacy makes it similar to the one proposed by Blanc et al., but it takes a completely different approach to deal with inconsistency detection. His approach defines a few consistency rules for UML 1.3 and checks whether they hold or not each time a change is performed. His main contribution involves the *detection of scope*. The rules have to be checked against the elements that have changed or can be indirectly affected by the change, not against the whole schema. In previous methods, rules were defined in terms of types, but Egyed suggests a new solution where rules are checked against concrete instances, not types.

### 3.2.2 Quality Properties in the Literature

Metamodel constraints are one of the most important and obvious quality properties a conceptual schema has to satisfy. If they are not, the schema is syntactically incorrect. Nonetheless, there is plenty of proposals in the literature that propose other quality properties, including layout guidelines, best practices, and naming guidelines, among others. Here, we review some of them.

### Best Practices

Many proposals in the literature that aim to improve the resulting quality of a conceptual schema can be categorized as *best practices*. A best practice is a method or technique that has consistently shown results that are superior to those achieved by other means. In the field of conceptual modeling, best practices include refactoring a schema, distributing the elements in a certain layout, or making an implicit OCL constraint explicit.

Some proposals deal with the definition of constraints. In [34], the authors propose a set of stereotyped constraints that aim to simplify the definition of general constraints. These include "identifier constraints" or "path inclusion and path exclusion constraints", among others. The usage of stereotyped constraints is useful to both conceptual modelers and conceptual schema's audience. On the one hand, the definition of this constraints becomes easier. On the other hand, since the meaning of an stereotyped constraint is well established, it is easier for the audience to comprehend it.

In [33], the authors deal with association redefinitions, which is a new concept in UML 2.0. Their work includes an analysis of the interactions between taxonomic constraints and redefined associations. As a result, they propose a set of well-formedness rules, which can be summarized as "making an implicit constraint explicit". Thus, for example, a set of association redefinitions is not well-formed if it entails a *disjoint* (complete) constraint for a generalization set that the designer has specified as *overlapping* (incomplete).

More examples of best practices and relevant quality properties include:

- Classes must be identifiable [97]. The rationale behind this requirement is that when an entity type is identifiable, the users and the information system have a shared means to refer to its instances. If, on the other hand, an entity type is not identifiable, then the users and the information system will be unable to share information about instances of it.
- Recursive binary associations may define the properties of *reflexivity*, *symmetry*, and *transitivity*. When they do, the modeler has to define explicit OCL constraints [97].
- Remove redundant generalizations [97].

- The elements of a conceptual schema must be relevant. An element of a conceptual schema is relevant if it is used in a constraint, a derivation rule, or an event type [97].
- Detect when a binary relationship makes the schema not strongly satisfiable [60].

### Naming Guidelines

The names conceptual modelers give to the elements of a conceptual schema have a strong influence on the understandability of that schema. It is widely recognized that good names make it easier for requirement engineers, conceptual modelers, system developers and users to understand conceptual schemas [1, 38, 79, 82]. However, choosing good names is one of the most difficult aspects of conceptual modeling [107, p. 46]. In the literature, there have been several proposals of naming guidelines for conceptual schema elements [10, 13, 27, 46, 79].

Appendix A provides an in-depth review of the literature. Moreover, it includes a naming guideline proposal for the UML.

### Refactorings

As introduced in [17, 51], "refactorings are changes made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior". They describe what can be changed, how the modification can be done without altering the semantics, and what problems to look out for when doing so. Until now, refactorings have usually been discussed in the context of program code. As stated by Boger et al. in [17] refactorings may be defined on the level of models, so refactoring browsers could be implemented in the context of UML CASE tools rather than IDEs.

Refactoring by hand is time consuming. This simple fact prevents programmers from making refactorings they know they should, simply because refactoring costs too much. Automated tools that support refactoring would improve model's quality [51]. Since refactorings aim to change the schema in a way such that the two alternatives are equivalent, we may identify which is best-suited for our needs and, thus, has “more quality”.

### 3.2.3 Tool Support to Assess Quality

Users from many different domains rely on computer tools to carry on their tasks. Complex general tasks such as undoing and redoing changes, sharing files, or finding information, became easier only because they were performed within a computer. For example, writers do no longer write manuscripts manually—they use text processors—, and architects do not draw blueprints manually either—they use Computer Aided Design (CAD) programs.

Nowadays, the support offered by computer tools is much greater: they do not only simplify the work itself, but they provide further assistance to their users. One clear example are word processors, which include advanced functionalities such as detecting misspellings in real-time, finding word synonyms, and automatically fixing case typos, among others.

If we look at the domain of programming languages (which are closer to conceptual modeling), we can see that this kind of support does also exist. Computer programmers use Integrated Development Environments (IDE)—software applications that integrate any tool that is required to write, build, and test a computer program. IDEs include many functionalities aimed to simplify programmer’s work and improve code’s quality like, for example, *code refactoring*, *code completion*, *real-time compilation errors and warnings*, and so on.

As we shall see in Chap. 9, the conceptual modeling community has also tools to support the development of conceptual schemas. In general, professional modeling applications simplify the creation of conceptual schemas by ensuring (some, if not all) metamodel constraints and, in some cases, checking a few quality properties.

In the literature, there are several methods and frameworks to ensure the quality of a conceptual schema (in Chap. 6 we review many of them). These proposals are usually presented along with a prototype tool that provides some automatic assistance to the conceptual modelers. Unfortunately, these proposals have not been integrated into a single application (as IDEs do in the field of programming), making it difficult to use them all in practice.

In [135], Wohed presents a case study where a wizard tool that suggests the most suitable pattern from a pattern library was tested. She tries to determine whether such a tool could be successfully used as a pedagogical aid, as well as to gather opinion about such a tool. The study shows that the subjects who used the wizard tool were more willing to change their schemas following its suggestions.

In [17], Boger et al. discuss the important role refactorings play within the extreme programming and agile process community. According to the authors, refactorings—which are widely supported by IDEs on code—could (and should) be applied on UML models. In order to support them, the authors present a refactoring browser for UML—a prototype tool that assists conceptual modelers on detecting and applying refactorings.

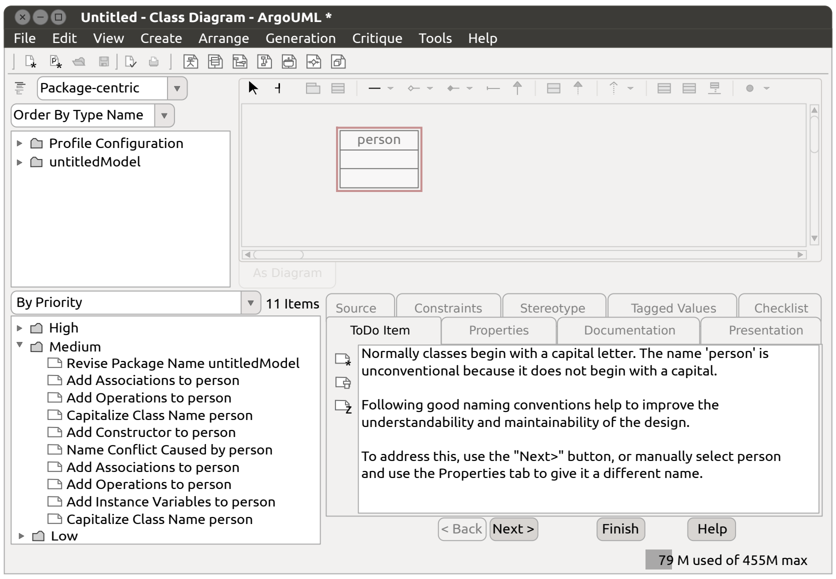
Another example where a tool may be useful is conceptual schema testing. In [123, 124], the authors present a method to test conceptual schemas. This method was implemented in a prototype tool [125]. In [40], the authors implement their own approach on conceptual schema testing.

In [37], Davies et al. tried to determine whether practitioners used conceptual modeling within the organizations they worked for and, if they did, which tools, techniques, and purposes were used. The most used tool according to their survey is Visio, far ahead from the second one Rational Rose. In fact, the authors reported that at least 40% of respondents did either not know or use any of the 24 tools named in the survey.

### ArgoUML—A Modeling Tool that Focuses on Quality

As stated in [105], ArgoUML is a *domain-oriented* design environment that provides *cognitive support* of object-oriented design. It provides some of the same automation features of a commercial CASE tool, but it focuses on features that support the cognitive needs of designers. Figure 3.2 shows the ArgoUML's User Interface criticizing modeler's work.

ArgoUML is particularly inspired by three theories within cognitive psychology [105]: (i) reflection-in-action, (ii) opportunistic design and (iii) comprehension and problem solving.



- High
- Medium
  - Revise Package Name untitledModel
  - Add Associations to person
  - Add Operations to person
  - Capitalize Class Name person
  - Add Constructor to person
  - Name Conflict Caused by person
  - Add Associations to person
  - Add Operations to person
  - Add Instance Variables to person
  - Capitalize Class Name person
- Low

Normally classes begin with a capital letter. The name 'person' is unconventional because it does not begin with a capital.

Following good naming conventions help to improve the understandability and maintainability of the design.

To address this, use the "Next>" button, or manually select person and use the Properties tab to give it a different name.

**Figure 3.2.** ArgoUML's User Interface showing some improvements available.

**Reflection-in-action** This theory observes that modelers do not conceive a fully-formed design. Instead, they construct a partial design and evaluate it so that, ultimately, they can revise, improve and extend it.

**Opportunistic design** A theory which states that, despite the fact that users plan and describe their work in an ordered fashion, in the end they choose successive tasks based on the criteria of cognitive cost.

**Comprehension and Problem Solving** The theory notes that designers have to bridge a gap between their mental model of the problem or situation and the formal model of a solution or system.

ArgoUML implements these theories using a number of techniques:

- A user interface which allows the user to view the design from a number of different perspectives.
- Processes running in parallel with the design tool that evaluate the current

design against models of what “best practice” design might be like (*design critics*).

- The use of *to-do lists*, so the user can record areas for future work.

- The use of checklists, to guide the user through a complex process.

## 3.3 Summary

In this chapter we have reviewed the state of the art in quality assessment in the field of conceptual modeling. We have seen that quality is a complicated concept that has been widely discussed during the past years.

First, we have discussed the concept of quality. The quality of a product is generally described as “the degree up to which a set of properties are met” [54]. In the software domain, software quality has traditionally focused on *evaluating* the final product [87] and, in fact, there are some international standards for evaluating it [65, 66]. We have also seen that there are several frameworks for evaluating the quality of a conceptual schema [70, 75, 87].

Second, we have reviewed some related work on improving the quality of conceptual schemas. We have seen that there is plenty of proposals in the literature that aim to improve some specific properties within a conceptual schema, including naming guidelines [10, 13, 27, 46, 79] and best practices [34, 33, 60, 97]. Some works, on the other hand, do not propose new quality properties, but discuss how modelers should work with them. An example of the latter are those works related to working with inconsistencies [15, 43, 44, 112], which focus on how to efficiently deal with inconsistencies and where the properties addressed are the well-known metamodel constraints.

We have also discussed the importance of tool support to assess quality. In this sense, we have briefly outlined some examples of the features that tools in a code programming domain offer—which include refactorings, code completion, error detection, and so on—. Moreover, we have seen that many authors implement their quality proposals within a prototype tool to make their proposals useful to practitioners.

We have presented ArgoUML as an example of a modeling tool that is focused on improving the quality of conceptual schemas. This tool criticizes the flaws of a conceptual schema, providing useful feedback to the modeler. The critiques it offers include some of the quality properties that we may find in the literature, but the catalog it includes is far from complete.

Clearly, there is a lot of work available on quality in conceptual modeling. Nonetheless, we believe there are still some open issues to be solved in this field. All these proposals are not arriving to practitioners, probably due to the difficulties of understanding and integrating them all in a unique tool. One possible explanation might be that each proposals focuses on a particular problem, and does not present the solution using a more global, unified framework. Thus, we may find, for example, proposals that describe how to use stereotyped constraints to define general constraints [34] and, therefore, simplify its understanding, as well as other proposals that propose working with inconsistencies [15, 43, 44, 112].

In this thesis we try to solve this problem by providing a formalization that unifies the definition of all these proposals in terms of quality issues. Therefore, our formalization permits to define *which* quality issues we address and *how* we treat them.

# Part III

# Contributions

Blah, blah, blah, let's have another scotch.

D. Koontz, *False Memory*

# 4 Overview of Our Method

A conceptual schema defines the general knowledge that an information system needs to know in order to perform its functions [97]. The increasingly important role conceptual schemas play in information system development requires that they must be of high-quality [75, 76]. This quality can be analyzed in terms of properties (or dimensions). All conceptual schemas should have the fundamental properties of syntactic and semantic correctness, relevance and completeness, but other quality properties have been proposed in the literature [10, 14, 27, 29, 33, 34, 38, 51, 31], and may be required or recommended in particular projects.

As we have already said in the previous chapters, the method proposed in this thesis is based on the notion of conceptual schema quality issue. We understand an issue as "an important [*quality*] topic or problem for debate or discussion". In essence, an issue is a condition. The condition may be an integrity constraint a schema must satisfy to be syntactically correct, a necessary condition for a schema to be satisfiable, a condition for a schema element to be relevant, a best practice defined as a condition that must be satisfied, and so on.

In this chapter, we briefly introduce quality issues and we outline our method. First, Section 4.1 provides an informal introduction to quality issues. The section starts with a motivating example that serves as an illustration of the quality issues we deal with in this thesis. Next, Section 4.1.2 defines quality issues informally, focusing on the difference between *quality issue types* and *quality issue instances*. In Sect. 4.2, we describe the general structure of our method. The method's structure can be summarized in three phases: (i) definition of quality issue types using our formalization, (ii) compilation of quality issue types in a catalog, and (iii) usage of quality issue types during the development of a conceptual schema. Finally, Section 4.3 summarizes the chapter.

## 4.1 A Brief Introduction to Quality Issues

In this section, we first introduce a simple motivating example that serves as an illustration of the quality issues we deal with in this thesis. Next, we use the motivating example to give an informal definition of quality issues.

### 4.1.1 Motivating Example

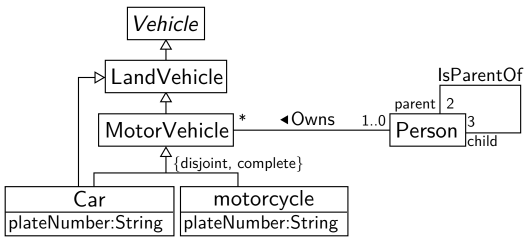


**Figure 4.1.** A conceptual schema with several quality issues.

Consider the structural conceptual schema of Fig. 4.1. There is a four-level hierarchy of *IsA* (*Car* and *motorcycle IsA MotorVehicle IsA LandVehicle IsA Vehicle*). A person may own any number of motor vehicles. Moreover, there is the typical parent-child relationship type.

Even if the example is very small, we can detect several quality issues. On the one hand, there are some issues that are clearly “problems”:

a) the cardinality constraint [1..0] of the participant *person* in association *Owns* is syntactically incorrect,

b) the entity type *motorcycle* does not start with a capital letter, despite several naming guidelines recommend that entity types start with a capital letter (e.g. [10]),

c) the abstract entity type *Vehicle* has only one subtype (*LandVehicle*). Either *Vehicle* is not abstract, or both type and subtype have the same population,

d) the cardinality constraints of association *IsParentOf* are not satisfiable,

e) the specialization *Car IsA LandVehicle* is redundant, and

f) the attribute *plateNumber* is repeated in all subtypes of *MotorVehicle*.

On the other hand, there are a few issues that should be checked. For example, the modeler has to check whether:

g) the name of the association *IsParentOf* (as well as the name of the association *Owns*) makes sense[1], and

h) the recursive binary association *IsParentOf* needs a constraint enforcing the *asymmetry* property.

### 4.1.2 Informal Definition of Quality Issues

In Chap. 1, we have defined a quality issue as “an important *topic* or *problem* for debate or discussion”. According to this definition, an issue can therefore point out a “problem” that needs to be solved, or a “topic” the modeler has to pay attention to. Our method makes this distinction too and classifies issues in two different kinds: *problem issues* and *checking issues*. Regardless the issue kind,

[1]See Sect. A.3 for further details on this issue.

the conceptual modeler is responsible of addressing and solving all issues her conceptual schema has before it is “finished”.

Informally, a *problem issue* is an issue that should not happen in the schema. These issues can be automatically detected and, when they are, we know “for sure” the schema has a defect. In order to solve issues of this kind, the conceptual modeler *has to* change the schema in a way such that the issues cease to exist. As long as one of them exists, the schema is not correct.

In the example from Fig. 4.1, (e) is a *problem issue* because it is not considered good practice to have redundant specializations in a schema. Therefore, the schema “has a problem that has to be fixed”. Note that (1) we can automatically detect this situation, and (2) as long as there is a redundant generalization in the schema, the schema is not correct. A possible solution to this problem issue could be to remove the redundant specialization from the schema.

On the other hand, a *checking issue* is an issue that requires the conceptual modeler to check something that cannot be automatically checked. For example, (h) is a *checking issue* because the modeler has to manually check if a recursive binary association is asymmetric or not. This checking issue forces the modeler to pay attention to a particular part of the schema and make sure that the model represents the domain properly. Because of this issue, the modeler may ask herself the following questions: “is the association asymmetric?”, “if it is, is an integrity constraint enforcing the asymmetry property missing?”, “if it is not, is there an unnecessary constraint that enforces it?”, and so on.

The only way to solve a checking issue is by manually setting it as “checked”. If the checking issue pointed out a problem in the schema that was unnoticed (e.g. the association was asymmetric, but the corresponding integrity constraint was missing), the modeler has to perform an action to solve it and, then, mark the issue as “checked”.

There are some issues that can be defined as problem or checking, depending on the information we have/use. Consider, for example, the identifiability issue, which states that “all entity types defined in a conceptual schema must be identifiable” [97, p. 109]. The rationale behind this requirement is that when an entity type is identifiable, the users and the information system have a shared means to refer to its instances.

In order to make an entity type identifiable, the conceptual modeler has to specify the set of properties (i.e. attributes and/or association ends) that uniquely identifies each entity. In UML/OCL, this is usually achieved by creating an OCL integrity constraint.

Consider, for example, the entity type *Car* from Fig. 4.1. Assuming that, in general, a car's plate number uniquely identifies the car, we can define the following constraint to specify that "a *Car* is identifiable using its property *plateNumber*":

```
context Car inv is identified by its plate number:
   Car.allInstances()->isUnique(
      Tuple{plateNumber:String=plateNumber}
   )
```

When using OCL integrity constraints, the identifiability issue has to be a checking issue. The rationale is we cannot automatically know for sure if there is an OCL integrity constraint specifying the set of properties that uniquely identify an entity type. As a consequence, the modeler has to manually check whether there is such constraint for each entity type, define it when it does not exist, and set the issue as checked.

However, this issue can also be defined as a problem issue, if we were able to automatically determine whether the required constraint exists or not. In [34], the authors propose a UML profile to simplify the definition of several general constraints, including the identifiability constraint. Clearly, once the constraint is stereotyped, we can automatically check its existence and, therefore, we can define the issue as a problem issue. Figure 4.2 shows the stereotyped *Identifier* constraint applied to the entity type *Car*.

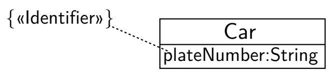


**Figure 4.2.** Example of the use of the *Identifier* stereotype.

## Issue Types and Issue Instances

Consider, for example, issues (b) and (f). These issues point out specific problems of the schema depicted in Fig. 4.1—(b) points out that "the class *motorcycle* does not start with a capital letter”, and (f) that “the attribute *plateNumber* is repeated in all subtypes of *MotorVehicle*”. These issues are, in fact, instances of the following issue types:

$I_b =$ “the name of a class is not properly capitalized”, and

$I_f =$ “an attribute is repeated among all specific classes of a complete generalization set”.

> Informally, an **issue type** describes a *problem* we want to avoid in a conceptual schema, or a *situation* the conceptual modeler has to be aware of. An **issue instance**, on the other hand, points out that specific problem or situation in a concrete conceptual schema.[2]

A conceptual schema may have several issue instances of different issue types. Moreover, there may be several issue instances of the same issue type in a particular schema. These concepts are further detailed in Chap. 5.

## 4.2 General Structure of our Method

The goal of this thesis is to improve the quality of conceptual schemas. In order to do so, we propose the usage of quality issues during the development of conceptual schemas. Quality issues provide useful feedback to the modeler on those aspects of a conceptual schema that can (and thus should) be improved, as well as on how to do so.

In this section, we outline the main components of our method, which consists in three phases: (i) the definition of quality issue types using our formalization, (ii) the compilation of quality issue types in a catalog, and (iii) the usage of quality issue types during the development of a conceptual schema. Figure 4.3 depicts our method and outlines each phase.

[2]We may use the term “issue” indistinctly to refer both to issue types and issue instances.

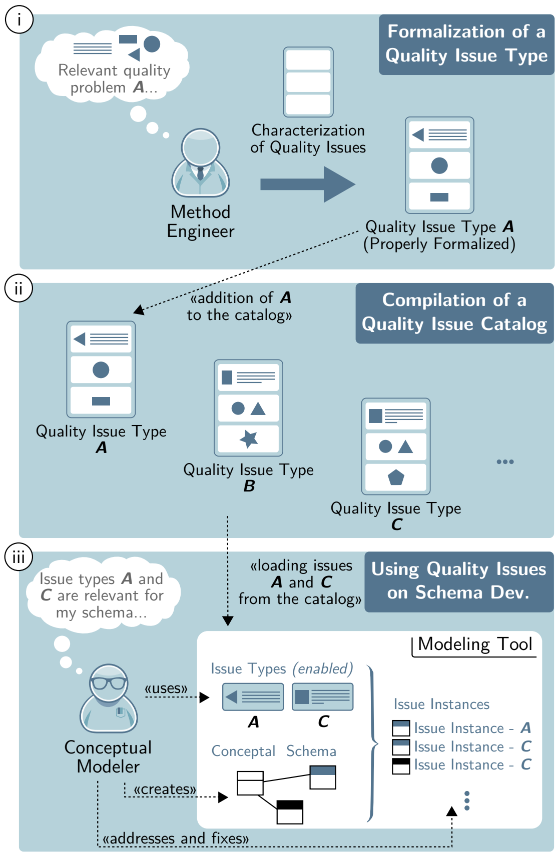

i
Formalization of a Quality Issue Type
Relevant quality problem A...
Characterization of Quality Issues
Method Engineer
Quality Issue Type A (Properly Formalized)
ii
«addition of A to the catalog»
Compilation of a Quality Issue Catalog
Quality Issue Type A
Quality Issue Type B
Quality Issue Type C
...
iii
Issue types A and C are relevant for my schema...
«loading issues A and C from the catalog»
Using Quality Issues on Schema Dev.
Modeling Tool
Issue Types (enabled)
«uses»
A
C
Conceptal Schema
Issue Instances
Issue Instance - A
Issue Instance - C
Issue Instance - C
Conceptual Modeler
«creates»
«addresses and fixes»

**Figure 4.3.** Overview of our method.

### 4.2.1 Defining Quality Issue Types

Conceptual schemas should not have defects and they should satisfy the quality criteria required by the methods used in their development [18, 28, 69, 75, 85, 111]. In Sect. 4.1.1 we have presented a motivating example which consisted of a conceptual schema with several quality issues. As we have introduced in Chap. 3 and as we shall see in Chap. 6, there is plenty of quality criterion proposals in the literature.

The first phase of our method aims to define these quality criteria uniformly in terms of conceptual schema quality issue types. The characterization and formalization of quality issues—which is explained in Chap. 5,—is the most important contribution of this thesis, as well as a major requirement for this phase. Quality issue types are defined by *method engineers* and include:

- a *Scope*, which includes the metaelements that may have issues of the type that is being defined,
- a *Condition*, which is defined at the metamodel level and it is used to determine whether an instance of the *scope* actually has an issue of that type,
- a *Kind*, which (as we have seen) can either define the issue type as a *Problem* or *Checking* issue,
- a set of *Actions* that, when executed, fix a particular instance of that issue type, and
- a few more elements that are discussed in Chap. 5.

### 4.2.2 Compilation of Quality Issue Types in a Catalog

The second phase of our method consists in the compilation of a quality issue catalog. Chapter 6 presents, as another contribution of the thesis, the catalog we have built so far using our method. The benefits provided by a catalog of uniformly-defined quality issues are two-folded. On the one hand, it can be used as a reference point for any conceptual modeler who wants to improve her conceptual schema. Hopefully, the catalog will contain all published quality issues using a unified representation and will be updated when necessary. On the other hand, we believe that using IDEs that automatize the detection and tracking of quality issues is one of the best ways to ensure the quality of a conceptual schema. IDE engineers are responsible of integrating quality issues within their tools. Obviously, the catalog can be used to this end and become a useful resource for these engineers.

### 4.2.3 Using Quality Issues on Schema Development

Finally, the last phase of our method consists in using quality issues to actually improve the quality of conceptual schemas. The formalization we propose for quality issues determines *when* quality issues appear in a conceptual schema, *how* they (may) change, and *which actions* need to be taken in order to fix them—i.e. the method we propose guides the conceptual modeler in the process of improving the quality of the schema. On the one hand, it can be used to detect which issues are present in the schema. On the other hand, the method provides useful insights on which actions may fix a particular issue.

We believe our method is especially useful when it is integrated in an IDE. In Chap. 7, we introduce the third major contribution of the thesis—a prototype Conceptual Modeling Assistant (CMA) that implements the method in a real IDE. As a result, the responsibilities of assessing the quality of a conceptual schema are split between the CMA and the modeler. On the one hand, the CMA is responsible of:

- identifying which quality issues are contained in the schema,
- keeping track of the state in which they are,
- monitor the changes the modeler makes to the schema and how they affect issues (i.e. whether new issues appear, whether old issues get fixed, and so on), and
- providing useful feedback on both

On the other hand, the *conceptual modeler* is responsible of:

- selecting which quality issue types (among those available in the catalog) are relevant in her conceptual schema, and
- making sure no issues of those types are present in the schema and, if they are, fixing them.

A fourth contribution of the thesis is the incremental evaluation of quality issues, which provides *instant feedback*. In Chap. 8, we introduce the importance of this incremental evaluation. In particular, we review the available literature on incremental evaluation—paying special attention to those works related to incremental evaluation of OCL expressions—, and discusses and evaluates how one specific method can be adapted to our formalization of issue types and implemented in our CMA.

## 4.3 Summary

First, Section 4.1 has introduced a motivating example that serves as an illustration of the quality issues we deal with. Using this example, we have provided an informal definition of quality issues. We have seen that quality issues may be classified as *problem issues* or *checking issues*. Roughly, the former ones are issues that point out a defect on the schema that should not happen. The latter, on the other hand, point out situations that the modeler needs to be aware of. It may be the case that these situations are a defect actually and, therefore, the modeler is responsible of performing some action to fix them. We have also introduced the differences between *issue types* and *issue instances*.

Section 4.2 has presented the general structure of our method. The method we propose in this thesis consists in three main steps: (i) the definition of quality issue types in terms of the formalization presented in Chap. 5, (ii) the compilation of a catalog of quality issues, as presented in Chap. 6, and (iii) the usage of quality issues during the development of a schema. In the remaining chapters of this thesis, we describe in detail the contributions of this thesis that conform the method.

Most men would rather deny a hard truth than face it.

G. R.R. Martin, *A Game of Thrones*

# 5 Characterization and Formalization of Conceptual Schema Quality Issues

*Engineering* is about getting results of the *required quality* within the schedule and budget [114, p. 8]. The most effective way to do so is to adopt a systematic and organized approach. Conceptual modeling is no exception to this rule. The increasingly important role conceptual schemas play in information system development requires that they must be of high-quality—i.e. they all should have the fundamental properties of syntactic and semantic correctness, relevance and completeness, as well as any other quality properties have been proposed in the literature that are relevant in the context in/for which they are being developed.

It is a fact that only a few quality properties (mainly those related to syntax) have been integrated into the development environments used by professionals and students, and thus enforced in the conceptual schemas developed by them. A possible explanation of this unfortunate fact may be that the proposals have been defined in the literature in disparate ways, which makes it difficult to use them in practice.

In the previous chapter we introduced our method for assessing the quality of conceptual schemas. The method we propose is based on the notion of “conceptual schema quality issues”, which we understand as “important quality topics or problems for debate or discussion”. Our method is structured in three main stages: (i) formalizing quality issues, (ii) compiling them into a catalog, and (iii) using them while developing a conceptual schema.

In this chapter we focus on the first stage of our method—i.e. the formalization of quality issue types and quality issue instances. We aim to formalize how quality issues can be *detected* within a conceptual schema, how they *evolve* as the schema changes, and how the conceptual modeler can *fix* them [4]. Its evaluation is presented in the subsequent chapters of the thesis in terms of its *expressiveness* and *usefulness*. On the one hand, the *expressiveness* of our method is demonstrated by analyzing its capability to define many existing quality properties (i.e. issue types) we may find in the literature. The result of this evaluation is the second contribution of the thesis—a catalog of conceptual schema quality issues (Chapter 6). On the other hand, we evaluate in two steps the *usefulness* of our method by analyzing the presence of quality issues in a set of conceptual schemas developed by students. First, in Chap. 6 we demonstrate the feasibility of a quality assurance approach based on the catalog—i.e. conceptual schemas could be improved by using the catalog. Second, in Chap. 9 we demonstrate that our catalog leads to the detection of more quality issues compared to the issues current modeling tools are able to detect.

Section 5.1 starts with the formal definition of the main components that shape all these relevant aspects of an issue type. These components include the types for which an issue may exist, the conditions that determine whether an instance of these types produce or not an issue instance, how a particular issue type is classified, or the actions that can be executed to fix a particular issue instance, among others. Next, Section 5.2 describes the different states in which an issue instance can be depending on its classification. Section 5.3 describes the notion of “issue precedence”, which is used to prioritize which issue instances should be presented to the conceptual modeler and, thus, optimize the amount of feedback she gets. In Sect. 5.4, we present an algorithm that can be used to compute the issues raised in a schema, either under request or continuously. Next, Section 5.5 presents in detail the concept of “issue actions”, which are operations the modeler can execute in order to fix a particular issue instance. Finally, Section 5.6 summarizes this chapter.

## 5.1 Definition of Issue Type and Issue Instance

Let $S$ be a schema that consists of $n$ schema elements $e_1$, …, $e_n$, which are an instance of the corresponding schema metatypes. The most important schema elements are entity, relationship and event types, *IsA* relations, constraints, derivation rules and pre/post conditions. Among the auxiliary schema elements there are strings (for example, names of relationship types) and integers (for example, minimum values of cardinality constraints).

We define a conceptual schema quality issue instance (for short, issue) of type $I_x$ as a fact $I_x(e_1, \ldots, e_m)$ where $e_1, \ldots, e_m$ are schema elements, $m \geq 1$. In a schema there may be several distinct issues of the same issue type, and there may be several issues for the same tuple $\langle e_1, \ldots, e_m \rangle$.

For example, consider the issue (e) presented in Sect. 4.1.1 “the specialization *Car IsA LandVehicle* is redundant”, and assume that $g$ is the schema element corresponding to *Car IsA LandVehicle*. Then, issue (e) can be formalized as an issue $I_e(g)$ of issue type $I_e$ = “The specialization is redundant”.

Each issue is an instance of an issue type $I_x$. Formally, $I_x$ is a tuple:

$$I_x = \langle \mathcal{S}_x, \phi_x, \rho_x, \mathcal{K}_x, \mathcal{A}_x, \mathcal{O}_x, \mathcal{P}_x \rangle \tag{5.1}$$

where

– $\mathcal{S}_x$ is the *scope* of the issue type
– $\phi_x$ is the *applicability condition*
– $\rho_x$ is the *issue condition*
– $\mathcal{K}_x$ is the *kind* of the issue type
– $\mathcal{A}_x$ is the *acceptability* of the issue type
– $\mathcal{O}_x$ is a set of *issue actions*
– $\mathcal{P}_x$ is a set of *precedents*.

### 5.1.1 Potential Issues: the Scope and Applicability Condition

The scope $\mathcal{S}_x$ of an issue type $I_x$ is a tuple $\mathcal{S}_x = \langle T_1, \ldots, T_m \rangle$ of $m$ ($m \geq 1$) schema metatypes. At a given time, there could be an instance of $I_x$ for each element of the Cartesian product of $T_1 \times \ldots \times T_m$. In the previous example, assuming that $T_{\text{IsA}}$ is the schema metatype corresponding to the *IsA* relations of the schema $S$, then $\mathcal{S}_e = \langle T_{\text{IsA}} \rangle$. In principle, there could be an instance of issue type $I_e$ for each instance of $T_{\text{IsA}}$ in schema $S$.

In practice, often not all elements of $T_1 \times \ldots \times T_m$ may raise an issue of type $I_x$, but only a subset of them. Therefore, we find it convenient to define for each issue type $I_x$ an applicability condition $\phi_x(e_1, \ldots, e_m)$ such that the potential set $Pot(I_x)$ of elements of $T_1 \times \ldots \times T_m$ that may raise an issue of type $I_x$ is:

$$Pot(I_x) = \{\langle e_1, \ldots, e_m \rangle \mid \langle e_1, \ldots, e_m \rangle \in T_1 \times \ldots \times T_m \wedge \phi_x(e_1, \ldots, e_m)\} \tag{5.2}$$

When $Pot(I_x) = T_1 \times \ldots \times T_m$ then we define $\phi_x(e_1, \ldots, e_m) = True$.

If we consider the issue (e) from Sect. 4.1.1, we may assume that an *IsA* specialization can be redundant only if its subtype is a subtype of another specialization and its supertype is a supertype of another specialization. Therefore, we could define $\phi_e(g)$ = "the subtype of $g$ is a subtype of another specialization and the supertype of $g$ is a supertype of another specialization". Figure 5.2 illustrates the fact that the generalization *Car IsA LandVehicle* is in the $Pot(I_e)$ set, because its general classifier *LandVehicle* is a supertype of another generalization (*Car IsA MotorVehicle*) and its specific classifier is a subtype of another specialization (*MotorVehicle IsA LandVehicle*).

Other examples where the applicability condition becomes very useful include, for instance, issues:

- (g), which corresponds to a naming guideline for binary associations de-

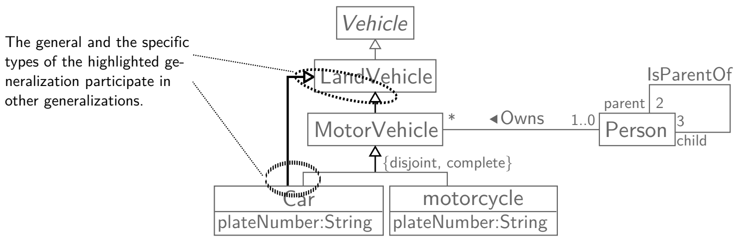


**Figure 5.1.** Computation of the *Potential* set.

scribed in Sect. A.3 and, therefore, has an applicability condition that selects binary associations only, and

- (h), which requires the modeler to check whether a recursive binary association (such as *IsParentOf*) needs a constraint enforcing the *asymmetry* property. Since this constraint makes sense within the context of recursive binary associations only, the potential set of elements that may raise an issue of this type only includes, as expected, any binary association whose member ends have the same type.

### 5.1.2 Raised Issues: the Issue Condition

An instance of issue type $I_x$ at a given time is an element of $Pot(I_x)$ that satisfies the issue condition $\rho_x(e_1, \ldots, e_m)$ at that time. The set $Raised(I_x)$ of issues of type $I_x$ raised at a given time is:

$$Raised(I_x) = \{\langle e_1, \ldots, e_m\rangle \mid \langle e_1, \ldots, e_m\rangle \in Pot(I_x) \land \rho_x(e_1, \ldots, e_m)\} \tag{5.3}$$

As before, when $Raised(I_x) = Pot(I_x)$ then we define $\rho_x = True$.

The issue condition corresponding to issue (e) in Sect. 4.1.1 would be (written in the appropriate language) $\rho_e(g) =$ "there is an indirect specialization between the subtype and supertype of $g$". Figure 5.2 illustrates the application of this condition, showing that there is an indirect specialization between *Car* and *LandVehicle* (in particular, the generalizations *Car IsA MotorVehicle* and *MotorVehicle IsA LandVehicle*).

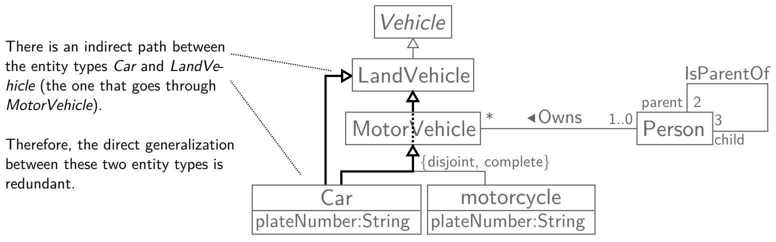

There is an indirect path between the entity types *Car* and *LandVehicle* (the one that goes through *MotorVehicle*).
Therefore, the direct generalization between these two entity types is redundant.
Vehicle
LandVehicle
MotorVehicle
◂Owns
*
1..0
Person
IsParentOf
parent
2
3
child
{disjoint, complete}
Car
plateNumber:String
motorcycle
plateNumber:String

**Figure 5.2.** Computation of the *Raised* set.

Another example is issue (f), which was first introduced in Sect. 4.1.1 and uses, as we shall see, a scope with more than one metatype. This particular issue points out the fact that there is an attribute (*plateNumber*) that is repeated in all subtypes of *MotorVehicle*.

The corresponding issue type $I_f$ could be defined as “an attribute is repeated among all specific classes of a complete generalization set”. Figure 5.3 shows the aforementioned *GeneralizationSet* with the attribute *plateNumber* repeated in all its specific classes, as well as a few additional attributes.

The issue type $I_f$ might be formalized as follows:

$$\begin{aligned}
\mathcal{S}_f &= \langle gs : \text{GeneralizationSet}, s : \text{String}\rangle \\
\phi_f(gs,s) &= \text{“There is an attribute named } s \text{ in a subclass} \\
&\quad\ \text{of the } \textit{GeneralizationSet gs}\text{”} \\
\rho_f(gs,s) &= \text{“All subclasses of the } \textit{GeneralizationSet gs} \text{ have an} \\
&\quad\ \text{attribute named } s\text{”}
\end{aligned}$$

that is, there is a raised issue for each tuple $\langle gs,s\rangle$ such that $gs \in$ *Generalization-Set*, $s \in$ *String*, and there is an attribute named $s$ in each subclass of $gs$.

In this example, the potential set *Pot*$(I_f)$ of elements of *GeneralizationSet* × *String* is:

$$Pot(I_f) = \{\langle gs,\ plateNumber\rangle, \langle gs,\ maxSpeed\rangle, \langle gs,\ numOfSeats\rangle\}$$

which, as expected, does not include the pairs $\langle gs,\ name\rangle$ nor $\langle gs,\ birthday\rangle$,

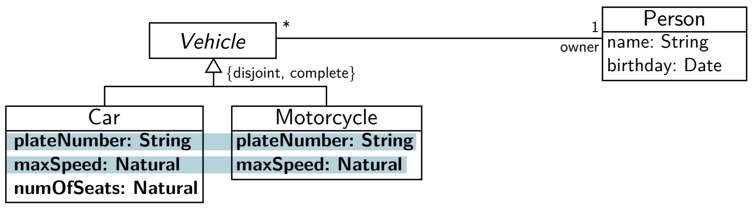


**Figure 5.3.** Computation of the *Potential* and *Raised* sets for an issue type that has a scope with more than one metatype.

because both *Strings* correspond to the names of attributes that do not belong to any subclass of $gs$.

Finally, the set *Raised*($I_f$) = {⟨*gs*, *plateNumber*⟩, ⟨*gs*, *maxSpeed*⟩}.

This issue type illustrates the utility of using scopes with more than one metatype: the issue instances of type $I_f$ clearly point out the *GeneralizationSet* where one or more attributes are repeated and provide an accurate and concise feedback by pointing out the specific names of these attributes.

### 5.1.3 Kind and Acceptance of an Issue Type

In our method, issue types have an issue kind $\mathcal{K}_x$, which may be *problem* or *checking* issue. All issues of an issue type are of the same kind. A *problem issue* $I_x(e_1, \ldots, e_m)$ is an issue that in principle (we will see later on that there may be exceptions) should not happen in a schema. Once raised, the issue should be solved (and thus, it ceases to exist), which can only be done by changing the schema in a way such that:

- $\langle e_1, \ldots, e_m \rangle$ is not an element of $T_1 \times \ldots \times T_m$, or
- $\langle e_1, \ldots, e_m \rangle$ does not satisfy $\phi_x(e_1, \ldots, e_m)$, or
- $\langle e_1, \ldots, e_m \rangle$ does not satisfy $\rho_x(e_1, \ldots, e_m)$

In the running example, $\mathcal{K}_e$ = *problem issue*, because it is not considered good practice to have redundant specializations in a schema. A possible solution to this problem issue could be to remove the redundant specialization from the schema.

A *checking issue* $I_x(e_1,\ldots,e_m)$ is an issue that requires the conceptual modeler to check something that cannot be automatically checked, or—in general—to perform some action that cannot be automatically performed. Once the checking has been done, or the action has been performed, the issue usually remains raised, but in a different state (as we will describe shortly).

In the schema of Fig. 4.1, an example of checking issue could be $I_h$ = "The symmetric property of the association is well defined". In this case, we have:

$$\begin{aligned}
\mathcal{S}_h &= \langle \text{Association} \rangle \\
\phi_h(a) &= \text{"Association } a \text{ is binary and recursive"} \\
\rho_h(a) &= \mathit{True}
\end{aligned}$$

The issue requires the conceptual modeler to check whether or not *IsParentOf* is *symmetric*, *asymmetric* or *antisymmetric* and, if so, to check that there is a constraint (invariant) that enforces it [97, p. 203]. Once checked, the issue becomes checked, although it is still a raised issue.

On the other hand, the issues of an issue type $I_x$ may or may not be acceptable, $\mathcal{A}_x = \{\mathit{True}, \mathit{False}\}$. An issue type may be defined as acceptable if the method engineer believes that some of its instances are acceptable in some circumstances. The exact meaning of the acceptability depends on the issue kind.

If $\mathcal{K}_x = \mathit{problem\ issue}$, then:

- $\mathcal{A}_x = \mathit{True}$ means that a conceptual modeler may find it reasonable that there are some instances of $I_x$ in a particular schema.
- $\mathcal{A}_x = \mathit{False}$ means that all issues of type $I_x$ must be solved.

If $\mathcal{K}_x = \mathit{checking\ issue}$, then:

- $\mathcal{A}_x = \mathit{True}$ means that a conceptual modeler may find it reasonable not to check some instances of $I_x$ in a particular schema.

**Table 5.1.** Classification of some issues of the schema shown in Fig. 4.1

| | Acceptable | Non-acceptable |
|---|---|---|
| **Problem Issue** | $I_b$: Entity type *motorcycle* does not start with a capital letter. | $I_a$: Cardinality constraint of participant *person* in association *Owns* is syntactically incorrect. |
| | $I_c$: Abstract entity type *Vehicle* has only one subtype. | $I_d$: Cardinality constraints of association *IsParentOf* are not satisfiable. |
| | $I_f$: Attribute *plateNumber* is repeated in all subtypes of *MotorVehicle*. | $I_e$: The specialization *Car IsA LandVehicle* is redundant. |
| **Checking Issue** | $I_g$: The name of the association *IsParentOf* must make sense. | $I_h$: The *symmetric* property of *IsParentOf* is well defined. |

- $\mathcal{A}_x = False$ means that all issues of type $I_x$ must be checked.

In the example of Fig. 4.1, the issue $I_c$ = "Abstract entity type has only one subtype" could be an instance of an *acceptable* issue type, because there may be situations in which the conceptual modeler can accept issues of this type.

Table 5.1 shows the classification according to *kind* and *acceptability* of the issues raised in the motivating example of Fig. 4.1.

## 5.2 The Lifecycle of an Issue Instance

Figure 5.4 depicts the states in which an issue can be,depending on its kind an acceptability. As we have already seen, an issue $I_x(e_1, \ldots, e_m)$ is automatically created at the time when $\langle e_1, \ldots, e_m \rangle \in Raised(I_x)$. Its initial state is always *Pending*. Similarly, an existing issue $I_x(e_1, \ldots, e_m)$ is automatically deleted at the time when $\langle e_1, \ldots, e_m \rangle \notin Raised(I_x)$. This transition to its deletion is represented by the action *Deletion*.

The simplest case is when $I_x$ is a non-acceptable problem issue (Fig. 5.4 (a)). Issues of this type are created in the initial state of *Pending*, and they remain in this state until they cease to exist. An example is $I_a$ = "The cardinality constraints of an association participant are syntactically incorrect". When an issue of this type is raised, it becomes *Pending* and it remains in this state until the conceptual modeler changes the schema in a way that the issue is not raised.

Problem issues that are acceptable may be in the states of {*Pending*, *Accepted*}

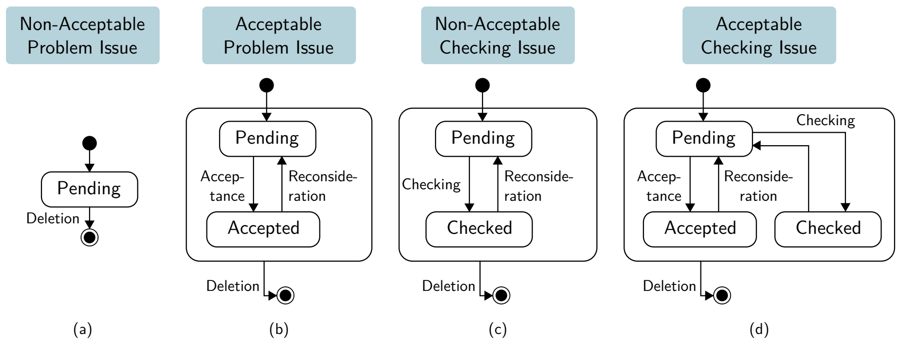


**Figure 5.4.** The lifecycle of an issue of type $I_x$, depending on its $\mathcal{K}_x$ and $\mathcal{A}_x$

as shown in Fig. 5.4 (b). If the conceptual modeler accepts (event *Acceptance*) one of these issues, then the issue changes to the state of *Accepted*. The transition can be reversed if the conceptual model reconsiders the acceptance (event *Reconsideration*). An example is $I_b$ = "The name of entity type does not start with a capital letter". When an issue of this type is raised, it becomes *Pending*. In most cases, the conceptual modeler will change the name and then the issue will be automatically deleted. However, given that $I_b$ has been defined as acceptable, the conceptual modeler may choose to accept issues of this type.

Non-acceptable checking issues can be in the states of {*Pending*, *Checked*} as shown in Fig. 5.4 (c). Once the conceptual modeler checks (event *Checking*) the issue, it changes to the state of *Checked*. The transition can be reversed if the conceptual model reconsiders the checking (event *Reconsideration*). An example is $I_h$ = "The symmetric property of the association is well defined" When an issue of this type is raised, it becomes *Pending* and it remains in this state until the conceptual modeler performs (event *Checking*) one of the issue actions, which checks the issue.

Finally, acceptable checking issues can be in the states of {*Pending*, *Accepted*, *Checked*} as shown in Fig. 5.4 (d). The semantics of *Checked* is the same as the previous case, but now an issue can be accepted (event *Acceptance*), meaning that the conceptual modeler, for whatever reason, decides not to check the issue. The transition to *Accepted* can be reversed. An example is $I_g$ = "The name of the association must make sense". If the conceptual modeler performs (event *Checking*) one of the issue actions, then the issue becomes *Checked*. However, the conceptual modeler may decide not to check the issue and accept the name of the association as it is.

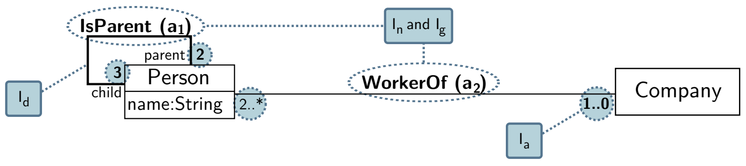
## 5.3 Issue Precedence

In some cases, the issues of a given issue type $I_x$ should only be considered if there are no other unsolved issues of some specific types. For example, the issues of type $I_d$ = “The cardinality constraints of an association are not satisfiable” should only be considered if there are no unsolved issues of type $I_a$ = “The cardinality constraints of an association participant are not syntactically correct”. Clearly, it makes no sense to check satisfiability if some cardinality constraints are incorrect. We formalize these issue relationships by means of precedence relationships, which are obviously acyclical. For each issue type $I_x$ we can define two sets of issue type precedents: *global* and *instance*.

Figure 5.5 depicts a conceptual schema with a few issues and their precedents. On the one hand, if an issue type $I_y$ is a *global precedent* of another issue type $I_x$, then there cannot be issues of type $I_x$ as long as there are any unsolved issues of type $I_y$. For instance, in the above example we would define $I_a$ as a global precedent of $I_d$. On the other hand, if $I_y$ is an *instance precedent* of $I_x$, then $I_x$ should only be considered *for those instances* that are not involved in an issue of type $I_y$. For example, consider the issue type $I_g$ = “The name of the association must make sense” and a new problem issue type based on a certain naming guideline $I_n$ = “The name of the association is not a verb phrase in third person singular”. If there is a schema with two associations $a_1$ and $a_2$ and the name of $a_1$ is not in the correct form, then the issue $I_n(a_1)$ would be raised. Clearly, the modeler has to fix the name of $a_1$ prior to checking whether it makes sense or not. However, if $I_n(a_2) \notin \mathit{Raised}(I_n)$, then she should be able to check whether the name of $a_2$ makes sense regardless the issue $I_n(a_1)$ is raised or not.

Formally, the *global precedents* of an issue type $I_x$, denoted $\mathcal{GP}_x = \{I_1, \ldots, I_p\}$, is a set of issue types $I_1, \ldots, I_p$ such that any issue of type $I_x$ should be considered only if there is no any unsolved issue of types $I_1, \ldots, I_p$. Then, the first example would be formalized as $\mathcal{GP}_d = \{I_a\}$.

The *instance precedents* of an issue type $I_x$, denoted $\mathcal{IP}_x = \{\langle I_1, \mathcal{M}_{x:1}\rangle, \ldots,$



**Global Precedent Examples**

$I_a$: "Syntactically correct cardinality constraints"

$I_d$: "Non-satisfiable cardinality constraints" (precedent of $I_n$)

**Issue Instances**

**$I_a$(** *1..0* **)**, because the lower value (1) is greater than the upper value (0).

**$I_d$(** *IsParentOf(parent:Person[2], child:Person[3])* **)**
└ Issue type *d* is never evaluated because its global precedents are not satisfied and, thus, this issue instance is not reported to the modeler.

**Instance Precedent Examples**

$I_g$: "The name makes sense"

$I_a$: "The name is a verb phrase" (precedent of $I_g$)

**Issue Instances**

**$I_n$(** $a_1$ **)**, because the name *is not* a verb phrase.

**$I_g$(** $a_1$, *"WorkerOf"* **)**
└ This issue instance is not created until its instance precedent is fixed.

~~$I_n$( $a_2$ )~~ does not exist, because the name *is* a verb phrase

**$I_g$(** $a_2$, *"IsParent"* **)**, because the instance precedent is satifsied.

**Figure 5.5.** Example of Global Issue Precedence and Instance Issue Precedence.

$\langle I_q, \mathcal{M}_{x:q}\rangle\}$, is a set of tuples of the form $\langle I_y, \mathcal{M}_{x:y}\rangle$ where $I_y$ is an issue type whose scope $\mathcal{S}_y \subseteq \mathcal{S}_x$ and $\mathcal{M}_{x:y}$ is a set of pairs $\langle i, j\rangle$ $(1 \leq i \leq |\mathcal{S}_y|$ and $1 \leq j \leq |\mathcal{S}_x|)$ that map each metatype in $\mathcal{S}_y$ to one, and only one, metatype in $\mathcal{S}_x$. Therefore, an issue $I_x(e_{x_1}, \ldots, e_{x_m})$ should not be considered if there is an unsolved issue $I_y(e_{y_1}, \ldots, e_{y_l})$ such that $\forall i, j \mid \langle i, j\rangle \in \mathcal{M}_{x:y} \Rightarrow e_{y_i} = e_{x_j}$. Then, the second example would be formalized as $\mathcal{IP}_g = \{\langle I_n, \{\langle 1, 1\rangle\}\rangle\}$ (assuming that the scopes of $I_g$ and $I_n$ are $\mathcal{S}_g = \langle Association, String\rangle$ and $\mathcal{S}_n = \langle Association\rangle$).

## 5.4 On Computing Issues Instances

In this section we propose an algorithm for computing the issues that are present in a conceptual schema $S$, given a concrete set of issue types $\mathcal{T}$. The algorithm can be used to compute raised issues either under request or continuously.

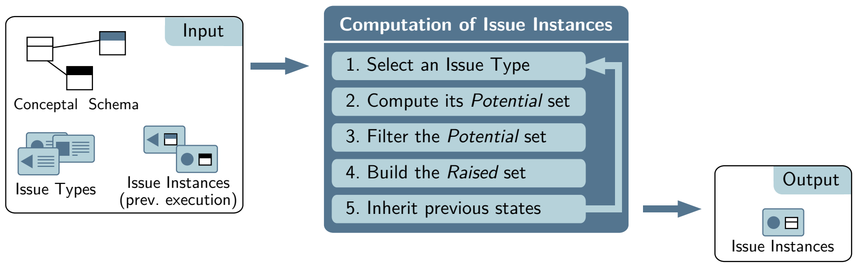

Input
Conceptal Schema
Issue Types
Issue Instances (prev. execution)
Computation of Issue Instances
1. Select an Issue Type
2. Compute its Potential set
3. Filter the Potential set
4. Build the Raised set
5. Inherit previous states
Output
Issue Instances

**Figure 5.6.** The main stages to compute issue instances.

Figure 5.6 depicts the main stages of the issue computation process. The process is divided in five stages that are executed iteratively for each issue type in $\mathcal{T}$. In the following, we briefly introduce the inputs and outputs of the process, and we outline each stage.

**The Input** In order to compute the issue instances of a conceptual schema $S$, our algorithm requires:

- the conceptual schema $S$,
- the set of Issue Types $\mathcal{T}$ the conceptual modeler selected as relevant, and
- the set of issue instances $\Psi_{\text{prev}}$ we computed in a previous execution of the process. These issue instances are a set of pairs $\langle I_x(e_1, \ldots, e_m), \varepsilon \rangle$ computed in the previous execution, where $I_x(e_1, \ldots, e_m)$ was a raised issue and $\varepsilon$ was its state (as described in Sect. 5.2).

**1. Select an Issue Type** In general, given any $I_x$ and $I_y$ issue types, the computation of the respective issue instances are independent processes that do not interfere each other. As a result, it does not matter whose instances are computed first. However, if $I_y$ is a precedent of $I_x$, it is a necessary condition that the issues of type $I_y$ are determined *before* the issues of type $I_x$, since the existence of issues of the former type may modify the set of raised issues of the latter type. Therefore, the first stage of the algorithm selects an Issue Type $I_x$ such that all its precedents (if any) have already been computed in a previous iteration.

**2. Compute the *Potential* set** In order to compute the issues of any type $I_x \in \mathcal{T}$ (whose precedents have already been computed in a previous iteration), we first have to determine the set *Pot*$(I_x)$. As we have already seen, the elements of a tuple $\langle e_1, \ldots, e_m \rangle \in Pot(I_x)$ must (1) be instances of the corresponding metatypes in the scope $\mathcal{S}_x$ (that is, $T_1(e_1) \wedge \ldots \wedge T_m(e_m)$) and (2) satisfy the applicability condition $\phi_x(e_1, \ldots, e_m)$.

**3. Filter the *Potential* set** Issue types may have *global* and *instance* precedents. If there is an issue whose type is a *global* precedent of $I_x$, then we must skip the evaluation of $I_x$ (that is, the *Potential* set is the empty set). If, on the other hand, there is one or more issues whose type is an *instance* precedent of $I_x$, we have to remove any tuple from the *Potential* set that does not have its *instance* precedents satisfied (as described in Sect. 5.3).

**4. Build the *Raised* set** Next, we have to compute the set *Raised*$(I_x)$. This can be easily achieved by simply selecting those tuples $\langle e_1, \ldots, e_m \rangle \in Pot(I_x)$ that satisfy the issue condition $\rho_x(e_1, \ldots, e_m)$. As a result, we have an issue $I_x(e_1, \ldots, e_m)$ for each tuple $\langle e_1, \ldots, e_m \rangle$ in *Raised*$(I_x)$.

**5. Inherit previous states** Finally, we have to properly set the state $\varepsilon$ of each issue instance we have just created. As we have seen in Sect. 5.2, when an issue instance is created, its state $\varepsilon$ is automatically set to *Pending*. During a continuous evaluation, however, this process is periodically executed—hopefully each time a change occurs—. Hence, it may be the case that some of the issues of any type $I_x$ we have just created (*Raised*$(I_x)$) already existed from a previous iteration (*Raised*$_{\text{prev}}(I_x)$).

Therefore, for each issue instance $I_x(e_1, \ldots, e_m)$ in *Raised*$(I_x)$, we have to check whether the issue is also in *Raised*$_{\text{prev}}(I_x)$. If that is the case, then the state $\varepsilon$ of the issue instance $I_x(e_1, \ldots, e_m)$ must be $\varepsilon_{\text{prev}}$ instead of *Pending*.

**The Output** The process finalizes once all issue types in $\mathcal{T}$ have been processed. The result is the set of pairs $\langle I_x(e_1, \ldots, e_m), \varepsilon \rangle$ where $I_x(e_1, \ldots, e_m)$ is a raised issue and $\varepsilon$ is its state (either *Pending*, if the issue did not exist in the previous iteration, or $\varepsilon_{\text{prev}}$, if the issue did already exist).

Algorithm 1 computes the raised issues of a conceptual schema. The algorithm can be executed under request or continuously. Each time the algorithm is executed:

**Algorithm 1** Computing Issue Instances

**Input** – $S$: a set of $n$ schema elements $e_1, \ldots, e_n$,
– $\mathcal{T}$: the set of issue types $I_1, \ldots, I_p$, and
– $\Psi_{\text{prev}}$: the set of pairs $\langle I_x(e_1, \ldots, e_m), \varepsilon \rangle$ computed in the previous execution, where $I_x(e_1, \ldots, e_m)$ was a raised issue and $\varepsilon$ was its state.

**Output** – $\Psi_{\text{new}}$: the set of pairs $\langle I_x(e_1, \ldots, e_m), \varepsilon \rangle$ computed in this execution, where $I_x(e_1, \ldots, e_m)$ is a raised issue and $\varepsilon$ is its state.

```
1:  procedure $updateIssues(S, \mathcal{T}, \Psi_{\text{prev}}) : \Psi_{\text{new}}$
2:      $\Psi_{\text{new}} \leftarrow \emptyset$
3:      $\mathcal{T}_{\text{pending}} \leftarrow \mathcal{T}$
4:      while $\mathcal{T}_{\text{pending}} \neq \emptyset$ do
5:          $I_c \leftarrow I_x \mid (I_x \in \mathcal{T}_{\text{pending}} \wedge (\nexists I_y \mid I_y \in \mathcal{T}_{\text{pending}} \wedge I_y \in \mathcal{P}_x))$
6:          $CandidateIssues \leftarrow \emptyset$
7:          $Pot(I_c) \leftarrow \{\langle e_1, \ldots, e_m \rangle \mid \langle e_1, \ldots, e_m \rangle \in T_1 \times \ldots \times T_m \wedge \phi_c(e_1, \ldots, e_m)\}$
8:          for all $\langle e_1, \ldots, e_m \rangle \in Pot(I_c)$ do
9:              if $arePrecedentsSatisfied(\langle e_1, \ldots, e_m \rangle, \mathcal{P}_c, \Psi_{\text{new}})$ then
10:                 if $\rho_c(e_1, \ldots, e_m)$ then
11:                     $CandidateIssues \leftarrow CandidateIssues \cup \langle e_1, \ldots, e_m \rangle$
12:                 end if
13:             end if
14:         end for
15:         $IssuesToKeep \leftarrow \{\langle e_1, \ldots, e_m \rangle \mid \langle I_c(e_1, \ldots, e_m), \varepsilon \rangle \in \Psi_{\text{prev}} \wedge$
                                        $\langle e_1, \ldots, e_m \rangle \in CandidateIssues\}$
16:         $IssuesToCreate \leftarrow CandidateIssues - IssuesToKeep$
17:         $\Psi_{\text{new}} \leftarrow \Psi_{\text{new}} \cup$
                    $\{i = \langle I_c(e_1, \ldots, e_m), \varepsilon \rangle \mid i \in \Psi_{\text{prev}} \wedge \langle e_1, \ldots, e_m \rangle \in IssuesToKeep\} \cup$
                    $\{\langle I_c(e_1, \ldots, e_m), Pending \rangle \mid \langle e_1, \ldots, e_m \rangle \in IssuesToCreate\}$
18:         $\mathcal{T}_{\text{pending}} \leftarrow \mathcal{T}_{\text{pending}} - \{I_c\}$
19:     end while
20:     return $\Psi_{\text{new}}$
21: end procedure
```

1. an issue type $I_c \in \mathcal{T}$ (whose issue precedents have already been computed in previous iterations) is randomly selected (line 5).

2. Then, for each tuple $\langle e_1, \ldots, e_m \rangle$ of the schema to which $I_c$ may apply—that is, $\langle e_1, \ldots, e_m \rangle \in Pot(I_x)$ (line 7)—, the algorithm checks that

   - its issue precedents are satisfied (line 8), and
   - it satisfies the issue condition $\rho_c$ (lines 7 to 13).

3. Finally, for those tuples $\langle e_1, \ldots, e_m \rangle \in \mathit{CandidateIssues}$, the algorithm generates the associated issues in the proper state (stage 5). If there was a pair $i = \langle I_c(e_1, \ldots, e_m), \varepsilon \rangle$ in $\Psi_{\text{prev}}$, $i$ is added to the result $\Psi_{\text{new}}$; otherwise, a new pair $i$ such that $i = \langle I_c(e_1, \ldots, e_m), \mathit{Pending} \rangle$ is created and added to $\Psi_{\text{new}}$ (lines 14 to 18).

## 5.5 Issue Actions: Tackling Issues

The goal of this thesis is—as we already know—to improve the quality of a conceptual schema. In order to do so, we propose a method to uniformly define and treat conceptual schema quality issues. As we outlined in Chap. 4, the general idea is to (1) automatically detect the issues a schema contains and (2) require the conceptual modeler to fix them so that, in the end, the schema contains no issues.

Throughout the previous section of this chapter we have discussed all the relevant aspects for *detecting* the issue instances that a conceptual schema contains. In particular, we have described (a) the main components of an issue type, (b) the differences between *Problem* and *Checking* issue types, (c) the lifecycle of issue instances, and (d) an algorithm to compute the issue instances that are present in a schema.

The conceptual modeler is the one responsible of fixing issue instances. In Sect. 5.2 we have described the different states in which an issue instance can be and how an issue moves from one state to another. In general, an issue instance is fixed whenever the schema is changed in a way such that the issue is no longer raised. However, *Checking* issues types or *Acceptable* issue types can also be fixed by setting them as *Checked* or *Accepted*, respectively.

In order to provide better assistance to conceptual modelers, our formalization requires an issue type $I_x$ to have a set $\mathcal{O}_x$ of one or more *issue actions*. Assuming that $\mathcal{S}_x = \langle T_1, \ldots, T_m \rangle$, each issue action in $\mathcal{O}_x$ is an operation $op(p_1{:}T_1, \ldots, p_m{:}T_m)$ whose effect depends on $\mathcal{K}_x$. If it is a problem issue, then the execution of the operation solves the issue $I_x(e_1, \ldots, e_m)$ (that is, $\langle e_1, \ldots, e_m \rangle \notin \mathit{Raised}(I_x)$), and the issue ceases to exist. If $I_x$ is, on the other hand, a checking issue, then the execution of the operation may either solve the issue by removing it from the *Raised* set, or by setting the state of the issue to *Checked*.

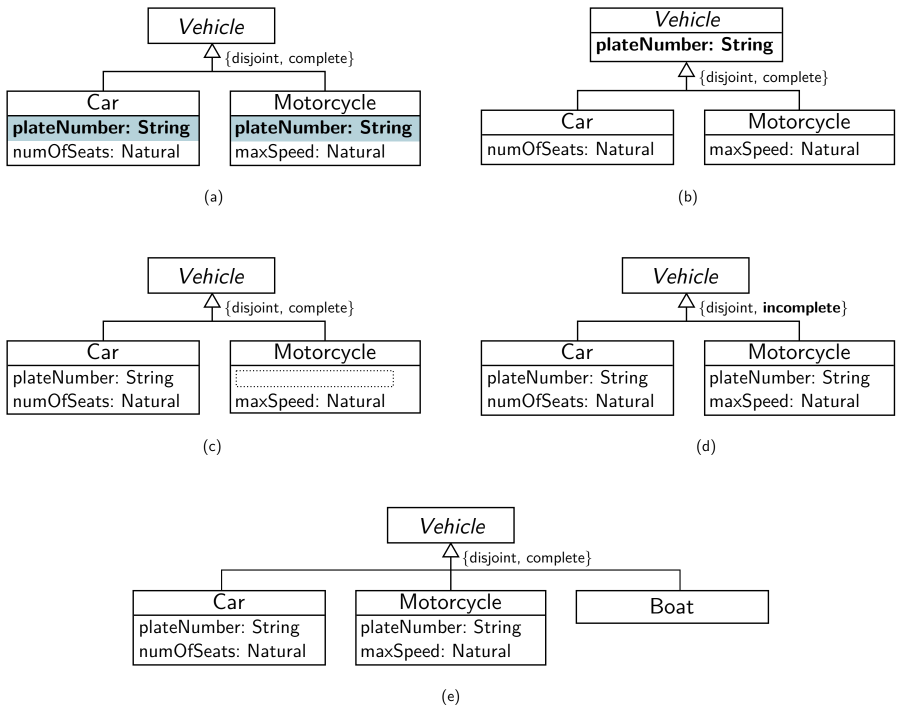


**Figure 5.7.** The result of applying different issue actions to solve the issue a conceptual schema contains.

For example, consider $I_f$ = "An attribute is repeated in all subtypes of a complete generalization set". Figure 5.7 (a) shows a conceptual schema with an instance $I_f(gs, plateNumber)$, where the attribute *plateNumber* appears in *Car* and *Motorcycle*. There are many actions the modeler can perform in order to fix an instance of the issue type $I_f$, including, for instance:

(b) the refactoring operation *pullUpAttribute* described in [51], which removes the repeated attributed from the specific classes and creates it in the general class (in the example, the attribute *plateNumber* is moved from *Car* and *Motorcycle* to *Vehicle*),

(c) removing one (or more) of the repeated attributes from the specific classes (in the example, the attribute *plateNumber* was removed from *Motorcycle* only),

(d) changing the *completeness* constraint of the *GeneralizationSet* from *complete* to *incomplete*, so that there may be instances of *Vehicle* that are not *Cars* nor *Motorcycles* and, therefore, do not need the attribute *plateNumber*, or

(e) creating a new specific class in the *GeneralizationSet* which does not include the repeated attribute (in the example, a new class named *Boat*).

and many more like, for example, renaming one of the repeated attributes, removing the *GeneralizationSet*, adding an already existing class in the *GeneralizationSet*, and so on.

Issue actions can be *automated* or *manual*. An *automated* action is an operation *op* whose parameters correspond to the scope of the issue type—i.e. assuming that $\mathcal{S}_x = \langle T_1, \ldots, T_m \rangle$, then the operation is $op(p_1{:}T_1, \ldots, p_m{:}T_m)$. If we have an issue $I_x(e_1, \ldots, e_m)$, calling the operation *op* using $e_1, \ldots, e_m$ as its input (that is, $op(e_1, \ldots, e_m)$) solves the issue. For example, solutions (b) and (d) may be achieved by executing the following operations respectively:

```
context System::op_b(gs:GeneralizationSet, n:String)
   post: gs.generalization.specific.ownedAttribute->select(
         a | a.name = n)->isEmpty()
   post: a.oclIsNew() and a.oclIsTypeOf(Property) and
         a.name = n and gs.generalization.general->
            any(true).ownedAttribute->includes(a)

context System::op_d(gs:GeneralizationSet, n:String)
   post: gs.isComplete = false
```

A *manual* action is an operation *op* that has more parameters than the types included in the scope. The signature of such an operation is therefore $op(p_1{:}T_1, \ldots, p_m{:}T_m, p_{m+1}{:}T_{m+1}, \ldots, p_{m+l})$, where the first *m* parameters match the scope of the issue type, and the other *l* are new, additional parameters that have to be initialized by the conceptual modeler. These additional parameters make the operation *manual*, for they can not be automatically determined. For example, solutions (c) and (e) may be obtained by executing the following operations respectively:

```
context System::op_c(gs:GeneralizationSet, n:String, a:Property)
```

```
   pre: gs.generalization.specific.ownedAttribute->includes(a)
   pre: a.name = n
   post: (Property.allInstances@pre() - Property.allInstances())->
            includes(a)

context System::op_e(gs:GeneralizationSet, n:String, s:String)
   pre: not gs.generalization.specific.name->includes( s )
   post: c.oclIsNew() and c.oclIsTypeOf(Class) and c.name = s and
         g.oclIsNew() and g.oclIsTyepOf(Generalization) and
         g.general = gs.generalization.general.any(true) and
         g.specific = c and
         gs.generalization->includes(g)
```

Note that $op_c$ includes an additional parameter *a*:*Property*, which is the attribute to remove, and $op_e$ includes an additional parameter *s*:*String*, which is the name of the class that has to be created and added to the *GeneralizationSet*.

These actions are *manual* because they require the conceptual modeler to manually specify the additional parameters before the operation $op_m$ can be executed. However, when the operation is called with the proper parameters, the modifications to the conceptual schema are performed automatically.

## 5.6 Summary

In this chapter we have formalized the concepts of *quality issue type* and *quality issue instance*—both introduced in Chap. 4—and we have presented their relationship. Hence, we have focused on the different aspects that define a quality issue type, how the instances of an issue type can be computed, and how they can be fixed. This formalization was published in [4].

First, we have seen that an issue of type $I_x$ is basically a fact of the form $I_x(e_1, \ldots, e_m)$, where $e_1, \ldots, e_m$ are schema elements. In order to determine the tuples $\langle e_1, \ldots, e_m \rangle$ that raise an issue of type $I_x$, our formalization includes the *scope* of an issue—that is, the metatypes $T_i$ such that $e_i \in T_i$—, and the *applicability* and *issue conditions* ($\phi_x$ and $\rho_x$, respectively) that filter the tuples in $T_1 \times \ldots \times T_m$. We have also seen that issue types may be classified according to their *kind*—which defines an issue as a *problem* (i.e. something that the method engineer defines as incorrect and, therefore, has to be fixed) or a *checking* issue type (i.e. something the conceptual modeler has to be aware of, either because it may lead to an error or because it is troublesome)—and their *acceptability*. Section 5.2 has described the lifecycle of the instances of an issue type depending on its kind and acceptability. Section 5.3 has introduced the concept of *issue precedence*, which is basically used to filter the amount of information the user is exposed to by not considering certain issue types while instances of other issue types (the former's precedents) exist.

Next, we have presented an algorithm to compute the issues that a conceptual schema has. The algorithm can be executed under demand—i.e. when the conceptual modeler is interested in knowing which issues her schema contains—or periodically. The process of computing issue instances is divided into five main stages, which are repeated for each issue type: (1) select the issue type whose instances are to be computed, (2) compute its potential set, (3) filter the potential set taking into account the precedents, (4) build the raised set of tuples, and (4) inherit the states of those issue instances that existed in previous iterations and hold still.

Finally, Section 5.5 has introduced *issue actions* as a mechanism that provides some insights to conceptual modelers on how issues can be fixed. Thus, we have seen that issue actions are basically operations whose execution fix issue instances.

Our formalization should be powerful enough to define in a uniform way most (if not all) of the quality properties proposed in the literature, which is a necessary prerequisite for fostering their integration and use in the diverse development environments used by professionals and students. In the next chapter we address this issue by building a catalog of quality issues that includes many quality properties.

Property is theft!
P.J. Proudhon, *What is property?*

# 6 Catalog of Quality Issues

As we have already seen in Sect. 1.2, the work presented in this thesis is following the main ideas of the Design Science Research methodology. The fundamental principle of design-science research is that knowledge and understanding of a design problem and its solution are acquired in the building and application of an artifact [62].

The method we have introduced in Chap. 4 aims to improve the quality of conceptual schemas by detecting and fixing quality issues. In the previous chapter, we have presented a formalization for quality issues. In principle, all quality properties proposed in the literature could be defined in terms of this formalization, and they could therefore be included in a unified catalog.

This chapter presents the catalog of quality issues we have built so far for UML class diagrams [6]. It contains all UML metamodel constraints plus 65 additional issue types that are available in the literature, including conceptual modeling conferences, journals, and books, as well as current modeling environments. Although the issues we present here are targeted at UML conceptual schemas, our method can be applied to other conceptual modeling languages and DSLs and, therefore, the issues may be defined and/or used for them too.

The compilation of this catalog is important for two main reasons. On the one hand, it validates the *expressiveness* of our formalization, since almost all the quality issues we found in the literature could be expressed using our method. On the other hand, the catalog makes quality issues openly and easily accessible to conceptual modelers, students, and practitioners, who could use it as a reference catalog to improve the quality of their conceptual schemas, especially if IDE developers integrate them into their tools.

The chapter is structured as follows. Section 6.1 presents a classification of issue types that ease the presentation and analysis of the catalog. In Sect. 6.2, we present a list of all the quality issue types we have defined so far using our formalization. For each issue type, we include a *name* and a brief *description*, as well as its *scope*, *kind*, and *acceptability*. Section 6.3 presents some quality issues that could not be defined using our formalization, discusses the rationale behind this problem, and proposes a solution. Section 6.4 evaluates the *usefulness* of our catalog. We conducted an experiment where we selected 13 conceptual schemas developed by students as part of their final projects—i.e. during the last year of their Computer Science degree—, and we evaluated how many issues they contain. Finally, Section 6.5 summarizes the chapter.

## 6.1 Classification of Issue Types

In Chap. 5, we formalized the concepts of issue type and issue instance. This formalization does not include any particular classification of issue types and, therefore, we can classify them as we please. For analysis and presentation purposes, in this thesis we classify issues according to their source. When integrating the catalog in a real modeling tool, the categories in which issue types are classified may be different than the ones we use in here. Table 6.1 shows a few more quality issues, classified using the following categories:

**Syntactic** An integrity constraint defined in the UML metamodel. An example is the above mentioned problem issue of $I_1$ = “There is a cycle in a general-

**Table 6.1.** Examples of quality issues.

| Source/Kind | Problem Issue | Checking Issue |
|---|---|---|
| Syntactic | $I_1$ = There is a cycle in a generalization hierarchy | $I_7$ = A constraint expressed in natural language evaluates to a *Boolean* value [93, p. 57] |
| Syntactic+ | $I_2$ = An attribute has no type<br>$I_3$ = A property has no stereotype [19] | $I_8$ = A derivation rule gives consistent number of values |
| Basic property | $I_4$ = A property derived by union does not have specific properties | $I_9$ = An $n$-ary association defines all non-graphical cardinality constraints that are relevant [77] |
| Naming guideline | $I_5$ = An attribute does not start with a lowercase letter | $I_{10}$ = The name of the entity type is semantically meaningful |
| Best practice | $I_6$ = The type of an attribute is an entity type | $I_{11}$ = The aggregation kind of a property is correct [19] |

ization hierarchy".

**Syntactic+** A syntactic integrity constraint applicable when UML is used as a conceptual modeling language (i.e. redefining and limiting an already existing UML constraint) or a constraint that is defined in a UML profile. One example is the problem issue $I_2$ = "An attribute has no type". In UML it is not mandatory that attributes have a type, but it is so in conceptual modeling.

**Basic property** A fundamental property that conceptual schemas should have to be semantically correct, relevant, and complete [75]. An example is the above mentioned checking issue of $I_9$ = "An $n$-ary association defines all non-graphical cardinality constraints that are relevant" [77], which is required for completeness.

**Best practice** A practice (not including naming guidelines) recommended by some authors in some contexts that aims to improve the quality of conceptual schemas. For example, some authors recommend that the type of an attribute should not be an entity type [108, p. 189]. This becomes the problem issue $I_6$ = "The type of an attribute is an entity type". Another example is the checking issue $I_{11}$ = "The aggregation kind of a property is correct" which may be enforced by a method that analyzes conceptual schemas in terms of collaboration patterns and determines that an aggregation could be better expressed by a composition [19].

**Naming guideline** Any naming guideline presented in [5]. For example, the guideline that recommends attributes to start with a lowercase letter [93, p. 54] corresponds to the problem issue $I_5$ = "An attribute does not start with a lowercase letter".

## 6.2 Issue Types

In this section, we present the catalog of conceptual schema quality issues we have compiled so far. The goal of the catalog is to demonstrate the *expressiveness* of our formalization. The catalog includes all UML metamodel constraints and 65 additional issue types that are integrated in current modeling environments[1] or published in conceptual modeling conferences[2].

Research articles published in literature related to the field of quality of conceptual schemas were extracted from the ER and the CAiSE conferences. We first selected the papers whose title included any of the following terms:

- quality
- refactor*
- design AND (critique* or flaw*)
- antipattern* or anti-pattern* or pattern*
- conceptual and model*
- measure*
- constraint

A total of 97 papers were found as a result of the simple search process. Next, we read the abstracts of the papers and selected those that provide measures or

[1]When compiling this catalog we realized that many modeling environments include quality issues that are not relevant for conceptual modeling activities, but for the design stage. We have not included these issues in our catalog, but as we shall see in Chap. 9, it is possible to do so.

[2]Chapter 9 analyzes the support provided by current IDEs, with a special emphasis on *ArgoUML* and *SDMetrics* .

guidelines for evaluating and improving the quality of a structural conceptual schema. As a result, 21 papers were selected. For each of them, we read the full paper and selected those that present specific quality properties in terms of guidelines that conceptual models should follow, excluding the papers that focus on the process of defining the conceptual schema.

## 6.2.1 Syntactic

As we have already stated, syntactic issue types correspond to metamodel constraints. Since we focus on conceptual schemas written in UML, the 92 syntactic quality issues included in our catalog correspond to UML metamodel constraints [93]. In order to create a quality issue from a metamodel constraint we simply need to:

- use the context of the constraint as the scope of the issue type,
- define the applicability condition as *True*, and
- define the issue condition as the negation of the metamodel constraint's expression.

Consider, for example, the metamodel constraint "Generalization hierarchies must be directed and acyclical" [93, p. 53], which is formalized as follows:

```
context Classifier inv: not self.allParents()->includes(self)
```

When we want to transform a metamodel constraint into a quality issue type we have to decide, on the one hand, whether it is *acceptable* or *non-acceptable* and, on the other hand, whether it is a *problem* or a *checking* issue type. Clearly, all metamodel constraints correspond to *non-acceptable* issue types because, by definition, all metamodel constraints have to be satisfied (at least) when the schema is finished. On the other hand, all metamodel constraints that are formally defined in OCL can be defined as *problem* issue types. For instance, the previous example would be defined as:

**There is a cycle of generalizations**
Non-acceptable Problem Issue Type
    $\mathcal{S}$: ⟨ Classifier ⟩

```
    ϕ(Classifier): true
    ρ(Classifier): self.allParents()->includes(self)
```

Nonetheless, the UML metamodel has a few constraints that cannot be defined as *problem* issue types—i.e. they are not formally defined in OCL. These constraints require the conceptual modeler to *ensure* (i.e. to *check*) that something is correct. Consider, for example, the constraint "Evaluating the value specification for a constraint must not have side effects" [93, p. 58]. When transforming this constraint into a *non-acceptable checking issue type*, we have:

**Evaluating the value specification for a constraint does not have side effects**
Non-acceptable Checking Issue Type

```
    S: ⟨ Constraint ⟩
    ϕ(Constraint): true
    ρ(Constraint): true
```

which is an issue type that, by default, exists for every single *Constraint* in the schema. The conceptual modeler is now responsible of checking that no *Constraint* has side effects.

### 6.2.2 Syntactic+

This section presents the list of syntactic+ quality issue types included in our catalog. They correspond to syntactic integrity constraints applicable when UML is used as a conceptual modeling language, and thus they are *non-acceptable*. We define these issues as *problem* quality issue types. Note that the name of a *problem* issue indicates the specific defect that the schema contains and that needs to be fixed by the conceptual modeler.

#### A binary association has both member ends as aggregate

**Scope** ⟨Association⟩

**Information** Problem Issue Type

**Description** According to the *CrMultipleAgg* critique in [105], aggregation and composition are used to indicate whole-part relationships, and by definition, the "part" end cannot be aggregate.

## An n-ary association has a navigable association end

**Scope** ⟨Association⟩

**Information** Problem Issue Type

**Description** According to the *associationNaryNavEnds* metric in [109], *n*-ary associations must not indicate navigability at any of the association ends.

## A class has no name

**Scope** ⟨Class⟩

**Information** Problem Issue Type

**Description** According to the *classUnnamed* metric in [109] and the *CrMissingClassName* critique in [105], as well as other authors in the literature [38, 97], classes should have a descriptive name that reflects the concept they represent.

## A class has specializations and it is marked as leaf

**Scope** ⟨Class⟩

**Information** Problem Issue Type

**Description** When a conceptual modeler sets the *isLeaf* property of a class to *True*, then no further specialization of this class are allowed. This issue controls this situation.

## An OCL expression does not compile

**Scope** ⟨Constraint⟩

**Information** Problem Issue Type

**Description** OCL expressions must be syntactically and semantically correct [92].

## A data type has no name

**Scope** ⟨DataType⟩

**Information** Problem Issue Type

**Description** According to the *classUnnamed* metric in [109], data types should have a descriptive name that reflects the concept they represent.

## A named element has an illegal name

**Scope** ⟨NamedElement⟩

**Information** Problem Issue Type

**Description** According to the *CrIllegalName* critique in [105], the names used in a model should only use letters, digits and underscore characters.

Please note that this requirement is of special importance in MDD environments, where code is usually very sensitive to variable names.

## A derived attribute has no derivation rule

**Scope** ⟨Property⟩

**Information** Problem Issue Type

**Description** According to [84, 97], derived entity or relationship types, as well as attributes, require a *derivation rule*. This derivation rule is an expression that defines the necessary and sufficient conditions for the derived entity or relationship type to be an instance of the given type, or for the attribute to determine its value(s).

Therefore, a derived attribute without a derivation rule is incomplete.

## An attribute has no name

**Scope** ⟨Property⟩

**Information** Problem Issue Type

**Description** According to the *CrMissingAttrName* critique in [105], attributes should be named.

## An attribute has no type

**Scope** ⟨Property⟩

**Information** Problem Issue Type

**Description** According to [84] and the *propertyNoType* metric in [109], a property has to define its type. Otherwise, the attribute has no meaning at all.

## An attribute overrides an inherited attribute without explicitly redefining it

**Scope** ⟨p:Property, inherited:Property⟩

**Information** Problem Issue Type

**Description** When UML diagrams are graphically drawn, attribute overriding is “implicit”—an attribute $a_s$ whose name is the same as the name of an attribute $a_g$ in a superclass automatically redefines $a_g$.

However, when considering the UML metamodel and its instantiation, $a_s$ has to explicitly redefine the inherited attribute $a_g$. Otherwise, the schema contains two indistinguishable attributes, which is forbidden by the UML metamodel [93].

## 6.2.3 Basic Quality

Fundamental properties that conceptual schemas should have to be semantically correct, relevant, and complete [75] are classified into the Basic Quality category as *non-acceptable* issue types.

### A recursive association with a mandatory member end does not imply an infinite population

**Scope** ⟨Association⟩

**Information** Checking Issue Type

**Description** According to [97, p. 203], many binary recursive relationship types have particular properties that may be defined as constraints. In conceptual modeling, the most important properties are *symmetry*, *transitivity*, and *reflexivity*.

When defining a recursive association, the modeler has to take special care in the multiplicities of the association.

This issue controls these special associations and warns the modeler to be aware of this situation.

**Example**

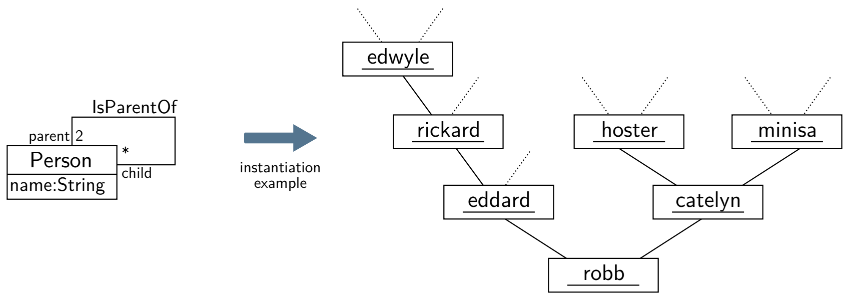


**Figure 6.1.** Example of a recursive binary association that results in an infinite population.

Consider, for instance, the typical recursive association *IsParentOf*(*parent*:*Person*, *child*:*Person*[*]) depicted in Fig. 6.1. This association represents the concept that each person in a domain has two parents (her father and her mother). Therefore, a modeler may be tempted to define the association with the multiplicities shown in Fig. 6.1—each *Person* has two parents exactly, and each *Person* may have any number of children.

Moreover, the *IsParentOf* association has to fulfill the following constraints:

- *asymmetry*: if *Eddard* is *Robb*'s parent, then *Robb* cannot be *Eddard*'s parent,
- *intransitivity*: if *Rickard* is *Eddard*'s parent and *Eddard* is *Robb*'s parent, then *Rickard* cannot be *Robb*'s parent,
- *irreflexivity*, a *Person* cannot be her own parent, and
- a more general constraint stating that "the ancestors of a *Person*'s parent cannot be parents of that *Person*" (for example, *Edwyle* cannot be *Robb*'s parent, because he is an ancestor of *Eddard*).

As a result, given all these constraints and taking into account that one of the multiplicities in the relationship *IsParentOf* is mandatory—each *Person* has two parents—, the information base has an infinite population: for each new *Person* we introduce in the information base, we potentially need to introduce her two parents, and then the parents of her parents, and so on.

Therefore, there are some situations in which the domain cannot be precisely represented in a conceptual schema. In this particular example, we have to define the multiplicity of the *parent* member end as optional—i.e. [0..2].

## A recursive binary association needs a constraint enforcing the reflexivity property

**Scope** ⟨Association⟩

**Information** Checking Issue Type

**Description** According to [97, p. 203], many binary recursive relationship types have particular properties that may be defined as constraints. In conceptual modeling, the most important properties are *symmetry*, *transitivity*, and *reflexivity*.

A recursive association can be *reflexive*, *irreflexive*, or none of these. If it is *reflexive* or *irreflexive*, an OCL constraint specifying this behaviour has to be defined by the modeler.

Note that $R(p_1 : E, p_2 : E)$ is

- *reflexive* if $E(x) \rightarrow R(x,x)$
- *irreflexive* if $E(x) \rightarrow \neg R(x,x)$

**Example**

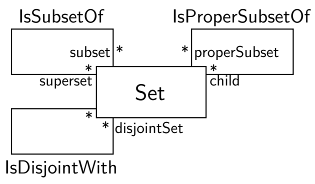


**Figure 6.2.** Examples of recursive associations.

Figure 6.2 depicts three recursive associations types in the domain of sets and their relationships, as first presented in [97, p. 203]: *IsSubsetOf*, *IsProperSubsetOf*, and *IsDisjointWith*.

When considering the reflexivity property for these associations, we have that the association *IsProperSubsetOf* is reflexive (i.e. a set $x$ is a proper subset of itself), and the association *IsDisjointWith* is irreflexive (i.e. a set $x$ cannot be disjoint with itself).

## A recursive binary association needs a constraint enforcing the symmetry property

**Scope** ⟨Association⟩

**Information** Checking Issue Type

**Description** According to [97], many binary recursive relationship types have particular properties that may be defined as constraints. In conceptual modeling, the most important properties are *symmetry*, *transitivity*, and *reflexivity*.

A recursive association can be *symmetric*, *asymmetric*, *antisymmetric*, or none of these. If it is *symmetric*, *asymmetric*, or *antisymmetric*, then an OCL constraint specifying this behaviour has to be defined by the modeler.

Note that $R(p_1 : E, p_2 : E)$ is

- *symmetric* if $R(x, y) \rightarrow R(y, x)$
- *asymmetric* if $R(x, y) \rightarrow \neg R(y, x)$
- *antisymmetric* if $R(x, y) \wedge R(y, x) \rightarrow x = y$

**Example**

Consider, for example, the recursive binary associations depicted in Fig. 6.2. When considering the symmetry property for these associations, we have that the association *IsDisjointWith* is symmetric (i.e. if a set $x$ is disjoint with a set $y$, then $y$ is also disjoint with $x$), the association *IsProperSubsetOf* is asymmetric (i.e. if $x$ is a proper subset of $y$, then $y$ cannot be a proper subset of $x$), and the association *IsSubsetOf* is antisymmetric (i.e. if a set $x$ is subset of a set $y$ and $y$ is subset of $x$, then $x$ and $y$ are the same set).

## A recursive binary association needs a constraint enforcing the transitivity property

**Scope** ⟨Association⟩

**Information** Checking Issue Type

**Description** According to [97], many binary recursive relationship types have particular properties that may be defined as constraints. In conceptual modeling, the most important properties are *symmetry*, *transitivity*, and *reflexivity*.

A recursive association can be *transitive*, *intransitive*, or none of these. If it is *transitive* or *intransitive*, then an OCL constraint specifying this behaviour has to be defined by the modeler.

Note that $R(p_1 : E, p_2 : E)$ is

- *transitive* if $R(x, y) \wedge R(y, z) \rightarrow R(x, z)$
- *intransitive* if $R(x, y) \wedge R(y, z) \rightarrow \neg R(x, z)$

**Example**

Consider again the recursive binary associations depicted in Fig. 6.2. When considering the transitivity property for these associations, we have that the associations *IsProperSubsetOf* and *IsSubsetOf* are transitive (i.e. if $x$ is a [proper] subset of $y$ and $y$ is a [proper] subset of $z$, then $x$ is a [proper] subset of $y$).

## The schema is not strongly satisfiable because of a recursive association

**Scope** ⟨Association⟩

**Information** Problem Issue Type

**Description** According to [59, 60] and [97, pp. 88+], a schema $S$ is satisfiable if it admits at least one legal instance of an information base. For some constraints, it may happen that only empty or nonfinite information bases satisfy them. In conceptual modeling, the information bases of interest are finite and may be populated.

We then say that a schema $S$ is strongly satisfiable if it admits at least one nonempty and finite legal instance of the information base. Schemas that are not strongly satisfiable are considered to be incorrect. This issue implements a necessary condition for a conceptual schema to be satisfiable. The method is able to deal with the cardinality constraints of a recursive binary association.

**Example**

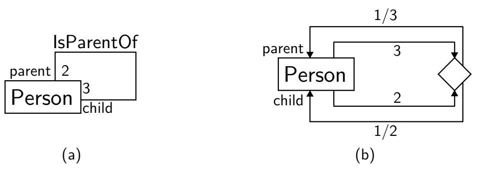


**Figure 6.3.** A recursive relationship type with nonsatisfiable cardinality constraints (as presented in [97, p. 90]).

The method we described in the Issue Type "The schema is not strongly satisfiable because of two binary associations" also applies to recursive types. An example is shown in Fig. 6.3. The schema (a) includes the constraints that each person must have two parents and three children. The corresponding graph (b) has a critical cycle, which proves that the schema is not strongly satisfiable.

## The schema is not strongly satisfiable because of two binary associations

**Scope** ⟨x:Association, y:Association⟩

**Information** Problem Issue Type

**Description** According to [59, 60] and [97, pp. 88+], a schema $S$ is satisfiable if it admits at least one legal instance of an information base. For some constraints, it may happen that only empty or nonfinite information bases satisfy them. In conceptual modeling, the information bases of interest are finite and may be populated.

We then say that a schema $S$ is strongly satisfiable if it admits at least one nonempty and finite legal instance of the information base. Schemas that are not strongly satisfiable are considered to be incorrect. This issue implements a necessary condition for a conceptual schema to be satisfiable. The method is able to deal with the two cardinality constraints that we can define for binary relationship types.

In the following, we provide an example and a brief explanation of the method, as described in [97, pp. 88+].

**Example**

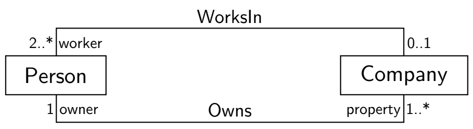


**Figure 6.4.** Example of unsatisfiable cardinality constraints (as presented in [97, pp. 88+]).

In [97, pp. 88+], the author describes a method that determines whether a schema with a set of cardinality constraints is strongly satisfiable or not. The method is able to deal with the two cardinality constraints may be defined for binary relationship types. It is based on building a directed graph $G$ and checking that it does not contain cycles of a particular type.

Consider, for example, the conceptual schema depicted in Fig. 6.4. There is no nonempty finite population of the four types that satisfies the four cardinality constraints. Figure 6.5 shows the graph that corresponds to the example in Fig. 6.4. The graph $G$ contains a vertex for each entity or relationship type in the schema. There are two arcs for each participant in a relationship type: one from the relationship type to the participant entity type and the other in the opposite direction.

Each arc is weighted as described in [97, p. 89]. A schema is strongly satisfiable if the graph $G$ does not contain a critical cycle. A *critical cycle* of $G$ is a non-empty sequence of arcs$(v_0, v_1)$, $(v_1, v_2)$, $\ldots$, $(v_{k-1}, v_k)$ such that

- $v_0 = v_k$,

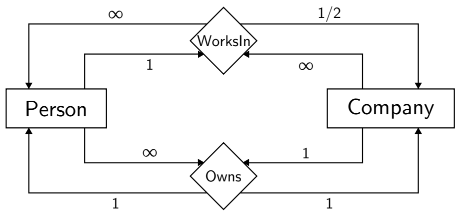


**Figure 6.5.** The graph $G$ corresponding to the schema of Fig. 6.4.

- $v_1, \ldots, v_k$ are mutually distinct, and
- the product of the weights of each arc in the cycle is less than 1.

In Fig. 6.5 there are some critical cycles, such as:

$$(\mathit{Works}, \mathit{Company}), (\mathit{Company}, \mathit{Owns}), (\mathit{Owns}, \mathit{Person}), (\mathit{Person}, \mathit{Works})$$

## A class cannot have instances

**Scope** ⟨Class⟩

**Information** Problem Issue Type

**Description** According to [93, p. 49], “a class describes *a set of objects* that share the same specifications of features, constraints, and semantics”. Clearly, we are only interested in those classes that are instantiable—i.e. they must have a nonempty population [97, p. 52].

A *Class A* is instantiable if it is not *abstract* or, if it is, there is a *complete GeneralizationSet* in the schema whose general classifier is *A*, and the specific classes in *A* are instantiable. This issue type points out that the referenced class is not instantiable.

## A class has really so many attributes

**Scope** ⟨Class⟩

**Information** Checking Issue Type

**Description** According to the *CrTooManyAttr* critique in [105], classes with too many attributes may be a maintenance bottleneck, and may reduce the understanding of the class. Furthermore, such situation may also evidence that a class hierarchy or a set of classes is being collapsed to a single class, which generally is a bad modeling practice. This issue notifies the modeler of this situation. By default, the issue is raised if the number of attributes is higher than ten. However, we assume the conceptual modeler is able to configure this value.

## A class is really involved in so many associations

**Scope** ⟨Class⟩

**Information** Checking Issue Type

**Description** According to the *CrTooManyAssoc* critique in [105], classes with too many associations may be a maintenance bottleneck and may reduce the understanding of the model. The class referenced by this issue type may have too many associations. By default, the issue is raised if the number of associations in which the class participates is higher than ten. However, we assume the conceptual modeler is able to configure this value.

## A class really has more than 60 attributes and operations

**Scope** ⟨Class⟩

**Information** Checking Issue Type

**Description** According to the *GodClass* metric in [109], classes with too many attributes may be a maintenance bottleneck, sources of unreliability, indicate a lack of (object-oriented) architecture and architecture enforcement, and may reduce the understanding of the class. This issue notifies the modeler of this situation. By default, the issue is raised if the number of attributes and operations is higher than sixty. However, we assume the conceptual modeler is able to configure this value.

### A class without generalizations or associations is relevant

**Scope** ⟨Class⟩

**Information** Checking Issue Type

**Description** In general, isolated classes are very rare in conceptual modeling [113, p. 106]; they are more frequent during the design stage where utility classes are used. This issue type notifies the modeler of this situation (see *UnusedClass* rule in [109] and *CrNoAssociations* critique in [105]).

### A cycle of composition relationships needs a constraint that enforces there are no composition cycles at the instance level

**Scope** ⟨Class⟩

**Information** Checking Issue Type

**Description** According to the *CrCircularComposition* critique in [105], a series of composition relationships (associations with black diamonds) that form a cycle is not permitted.

### An abstract class that has a concrete parent class is relevant

**Scope** ⟨Class⟩

**Information** Checking Issue Type

**Description** According to [71], a child class should be substitutable for the parent class. Since the parent class can be instantiated, but the child class can not, substitution is not possible anymore. This issue type notifies of this situation.

## An abstract general classifier has one specific class only

**Scope** ⟨Class⟩

**Information** Problem Issue Type

**Description** According to [97], a schema should not include redundant entity types. Two entity types are redundant if they must have always the same population.

A generalization $A_1$ *IsA* $A$ such that $A$ is *abstract* and $A_1$ is concrete is redundant if there is not, at least, another class $A_2$ such that $A_2$ *IsA* $A$.

## There are no attributes missing in a class without attributes

**Scope** ⟨Class⟩

**Information** Checking Issue Type

**Description** According to [84] and the *CrNoInstanceVariables* critique in [105], classes usually define instance variables. When they only define static attributes and/or methods, they should then be given the stereotype *utility*. However, this stereotype usually applies at a design level. As a result, this issue type points out that the class may either be unnecessary at a conceptual modeling stage or it may lack some relevant attributes.

## Two classes with synonym names do not represent the same concept

**Scope** ⟨a:Class, b:Class⟩

**Information** Checking Issue Type

**Description** This issue type identifies classes that may represent the same concept. In [113, p. 167], the author recommends reducing the possibility of synonyms for attributes within a class. According to [93], the named elements within a certain namespace must be distinguishable. Assuming the same namespace, two names are distinguishable if they are different.

It may be the case that two classes with different names (and, thus, distinguishable) represent the same concept, because the names are synonyms. Clearly, this may be a conceptualization error and lead to further errors, so the modeler has to make sure the schema is correct.

**Example**

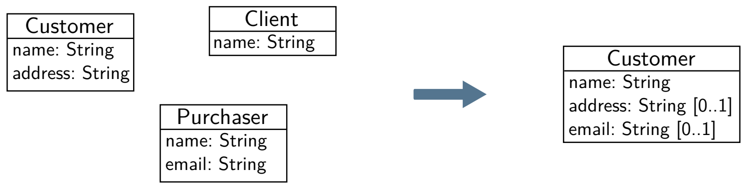


**Figure 6.6.** Example of three different entity types representing the same concept.

Figure 6.6 depicts a conceptual schema with three different entity types: *Customer*, *Client*, and *Purchaser*. These three names are synonyms. Therefore, it may be the case that they are representing the same concept but, for some reason, the modeler created three different entity types.

This issue type detects these entity types may be representing the same concept and requires the modeler to check this situation. In the example, the conceptual modeler realises that there should be one entity type only, and thus she merges them into only one entity type.

## An abstract class is the general classifier of a single generalization set that is incomplete

**Scope** ⟨GeneralizationSet⟩

**Information** Problem Issue Type

**Description** According to [97], a generalization $E$ *Gens* $E_1, \ldots, E_n$ satisfies the covering constraint if the instances of $E$ must be an instance of at least one $Ei$. Formally,

$$E(e) \rightarrow E_1(e) \vee ... \vee E_n(e)$$

Therefore, if a generalization set is incomplete (i.e. it is not covering), then an instance of $E$ is *not* necessary an instance of $E_i$. As a consequence, if $E$ is set as *abstract*, it is not possible (in general) to create instances of $E$ that are not instances of $E_i$, which means that either (a) $E$ is not abstract or (b) the generalization set is not complete.

## The default completeness and disjointness constraints are correct for a generalization set

**Scope** ⟨GeneralizationSet⟩

**Information** Checking Issue Type

**Description** The default values of a *GeneralizationSet*'s constraints are *overlapping* and *incomplete*. In practice, generalization set generally define different constraints than defaults (like, for example, the *disjointness* constraint). This issue type serves as a warning to the modeler to check that the default values apply.

## The specific classes of a disjoint generalization set are (in)-directly related to each other via generalizations

**Scope** ⟨GeneralizationSet⟩

**Information** Problem Issue Type

**Description** Let $G$ be a disjoint *GeneralizationSet*, and $A_1, \ldots, A_n$ its specific classes. The following necessary condition must hold for the satisfiability of $G$ [97, p. 219]:

- There are no (direct or indirect) *Generalizations* in the schema between $A_j$ and $A_k$ $(j, k : 1..n)$.

This issue type points out that there are two classes $A_j$ and $A_k$ such that, even though they both participate in a disjoint *GeneralizationSet*, they are (in)directly related to each other via generalizations.

### An element is not relevant

**Scope** ⟨NamedElement⟩

**Information** Problem Issue Type

**Description** One basic property all conceptual schemas have to satisfy is *completeness*, which states that "all relevant general static and dynamic aspects—i.e. all rules, laws, etc.—of the universe of discourse should be described in the conceptual schema" [57]. An elements is relevant if, and only if, the conceptual schema is not complete when the element is taken away. In practice, this means that the element is used in an event type or in a constraint.

### 6.2.4 Best Practice

Best practices are recommended by some authors in some contexts to improve the quality of conceptual schemas. The section presents the 18 best practices included in our catalog. One of the main differences between best practices (as well as naming guidelines) and issue types in the previous categories is their acceptability. The *acceptability* of a best practice depends on the conceptual modeler (or the project or context in which the conceptual schema is being developed). Thus, for example, the conceptual modeler may decide that redundant generalizations are *non-acceptable*—and therefore each time a redundant generalization is detected, the modeler has to change the schema so that the redundancy disappears—, but he may consider that it is sometimes *acceptable* that the type of an attribute is a class.

## An association class has no attributes nor relations

**Scope** ⟨AssociationClass⟩

**Information** Problem Issue Type

**Description** According to [97], UML provides the association class construct for defining reified relationship types. Reifying a relationship consists in viewing it as an entity. Usually, an association is reified in order to represent additional information (i.e. attributes or associations) that could not be represented if it was not.

This issue type points out the fact that the referenced association class does not include any additional information like attributes or relations. Therefore, the modeler has determine if there is some information missing and, if there is not, she has to remove the unnecessary association class.

## A class has a specialization that is not included in a generalization set

**Scope** ⟨Class⟩

**Information** Checking Issue Type

**Description** According to [97, pp. 220+], a generalization may be any set of *IsA* relationships with the same supertype. In practice, however, it makes sense to group into a generalization the *IsA* relationships that belong to the same dimension.

Whenever a class has several specializations, they are usually grouped into a *GeneralizationSet*. For example, a class *Person* may be specialized with the classes *Man* and *Woman*, which clearly belong to a *Sex* generalization set.

**Example**

Figure 6.7 depicts a small conceptual schema with the entity types *Person*, *Man*, and *Woman*. *Man* and *Woman* are defined as subtypes of the entity

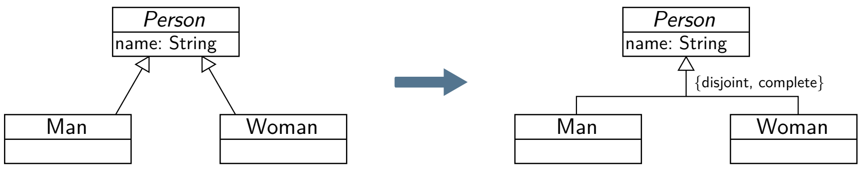


**Figure 6.7.** Example of two specializations that are not included in a *GeneralizationSet* and should be defined as a partition.

type *Person*.

In general, *Man* and *Woman* are a *partition* of the entity type *Person*. In other words, each instance of *Person* is either an instance of *Man* or *Woman*. In order to make it clear, the entity types *Man* and *Woman* have to be included in a complete and disjoint generalization set, as depicted in the second version of the conceptual schema.

## A class is identifiable

**Scope** ⟨Class⟩

**Information** Checking Issue Type

**Description** Entity types have to be identifiable [97, 113]. An entity type is identifiable if all its instances are identifiable. The rationale behind this requirement is that when an entity type is identifiable, the users and the information system have a shared means to refer to its instances. If, on the other hand, an entity type is not identifiable, then the users and the information system will be unable to share information about instances of it.

Informally, an entity is identifiable if there is an expression formed by lexical entities that denotes it [97, p. 107]. For example, the expression 'The person called Mark' is a reference to a person, formed from the lexical entity 'Mark'.

In order to make an entity type externally identifiable, an identifier constraint is required. Identifier constraints specify the set of properties that uniquely identify each instance of an entity type.

**Example**

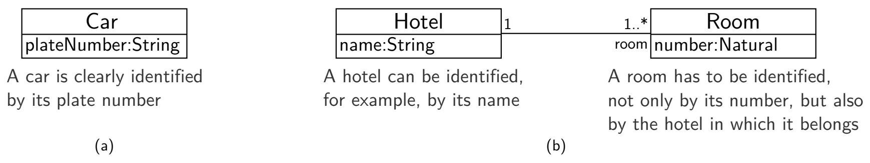

Car
plateNumber:String
A car is clearly identified by its plate number
(a)
Hotel
name:String
1
1..*
room
Room
number:Natural
A hotel can be identified, for example, by its name
A room has to be identified, not only by its number, but also by the hotel in which it belongs
(b)

**Figure 6.8.** Two examples of identifiable entity types.

In order for the users of an information system and the information system itself to share information about the instances the system has, these instances have to be identifiable.

In Fig. 6.8, we present two small examples that illustrate how entity types may be identified. In [97, pp. 107–108], the author describes the six different ways we have to identify an entity. Figure 6.8 (a) presents a single entity type *Car* with a *plateNumber* attribute. Presumably, *Cars* are identified by their *plateNumbers*, which means that two different cars cannot have the same plate number. Therefore, the modeler has to define an identifiability constraint [3] stating that two different cars cannot have the same plate number [34]. On the other hand, Figure 6.8 (b) is modeling a domain where we have *Hotels* and their *Rooms*. We may consider, for example, that *Hotels* are identified by its *name* (two different hotels have two different names necessarily), and *Rooms* are identified by the combination of the *Hotel* to which they belong and their *number*.

## An attribute named id is needed at the conceptual level

**Scope** ⟨Class⟩

**Information** Checking Issue Type

**Description** In conceptual modeling, classes do not require an *id* attribute to be uniquely identified [113, p. 184+]. These artificial attributes

[3]Please note that version 2.5 of the UML metamodel, introduced an attribute *isID* for *Properties*. According to its definition, if this boolean attribute is set to *True*, it specifies that the *Property* can be used for to uniquely identifying an instance of the containing Class.

(called “surrogate keys”) are usually added on a design stage, where instances have to be identifiable in a, for instance, relational database. These symbols are generated internally by the system and are not visible outside it.

It is a bad practice in conceptual modeling to create these attributes for identifying entites externally. As stated in [97, p. 109], “the identifiability requirement is independent of the symbols used in the information base to denote the domain objects; these symbols are generated internally by the system and are not visible outside it; therefore, they cannot be used to identify entities externally.” Consequently, the conceptual modeler has to select which attributes *in the domain* identify the entities.

### A data type is related to at least one class by a binary association

**Scope** ⟨DataType⟩

**Information** Problem Issue Type

**Description** Good object oriented analysis and design depends on careful choices about which entities to represent as full objects and which to represent as attributes of objects. According to the *CrNonAggDataType* critique in [105], data types should not be associated with classes. In principle, they should be used as the type of an attribute.

### A generalization is redundant

**Scope** ⟨Generalization⟩

**Information** Problem Issue Type

**Description** In UML, multiple inheritance is allowed—i.e. a class $A$ may have two superclasses $B$ and $C$ ($A$ *IsA* $B$ and $A$ *IsA* $C$).

According to [74] and [97, p. 51], “the representation of objects and entity types in an information system, and the classification of objects into entity types must all satisfy certain properties (...) including the *nonredundancy property*. This issue type points out *redundant generalization*, which should be avoided because they make the conceptual schema more complicated. We say that a generalization *A IsA B* is redundant if there is an indirect inheritance hierarchy between *A* and *B* (for instance, *A IsA C IsA B*).

**Example**

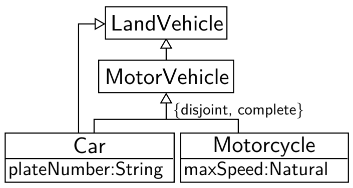


**Figure 6.9.** Example of a redundant generalization.

Figure 6.9 presents a conceptual schema with a redundant *Generalization*. As we can see, there is a direct *Generalization* between *Car* and *LandVehicle* (*Car IsA LandVehicle*). However, the schema also contains the generalizations *Car IsA MotorVehicle* and *MotorVehicle IsA LandVehicle*. Therefore, the former *Generalization Car IsA LandVehicle* is redundant.

## A generalization set is not complete, but it behaves as such because of type or type and multiplicity redefinitions

**Scope** ⟨GeneralizationSet⟩

**Information** Problem Issue Type

**Description** As D. Costal and C. Gómez describe in [33]:

*Theorem 4.6*: Let $A$ and $B$ be two classes and $R$ a binary association between them. Let $b$ be the association end that connects $R$ to class $B$ and $a$ the opposite end that connects $R$ to class $A$ with multiplicity $n$. Let $gs_1$ be a generalization set that specializes superclass $A$ into subclasses $A_1, \ldots, A_n$. Let $gs_2$ be a generalization set that specializes

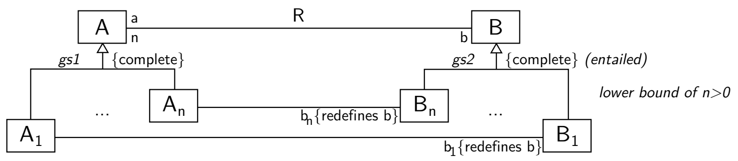


**Figure 6.10.** Complete constraint entailed by type or type and multiplicity redefinitions.

superclass $B$ into subclasses $B_1, \ldots, B_n$. Assume that $b_1, \ldots, b_n$ are type or type and multiplicity redefinitions of the end $b$ with their corresponding multiplicities, that each $b_i$ is connected to $B_i$ and that the opposite end of $b_i$ is connected to class $A_i$. Then, if the lower bound of $n$ is greater than zero and $gs_1$ is complete, the complete constraint of $gs_2$ is entailed by the type or type and multiplicity redefinitions (see Figure 6.10).

## A generalization set is not disjoint, but it behaves as such because of disjoint multiplicity redefinitions

**Scope** ⟨GeneralizationSet⟩

**Information** Problem Issue Type

**Description** As D. Costal and C. Gómez describe in [33]:

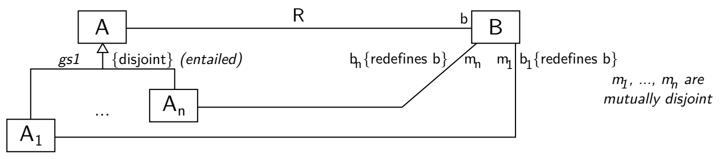


**Figure 6.11.** Disjointness constraint entailed by disjoint multiplicity redefinitions.

*Theorem 4.3*: Let $A$ and $B$ be two classes and $R$ a binary association between them. Let $b$ be the association end that connects $R$ to class $B$. Let $gs_1$ be a generalization set that specializes superclass $A$ into subclasses $A_1, \ldots, A_n$. Assume that $b_1, \ldots, b_n$ are multiplicity redefinitions of the end $b$ with their corresponding multiplicities $m_1, \ldots,$

$m_n$, that each $b_i$ is connected to $B$ and that the opposite end of $b_i$ is connected to class $A_i$. Then, if each pair of multiplicities $m_i$ and $m_j$ , where $1 \leq i \leq n$ and $1 \leq j \leq n$ and $i \neq j$, are mutually disjoint (i.e. each intersection is empty), the disjointness constraint of $gs_1$ is entailed by the multiplicity redefinitions (see Figure 6.11).

## A generalization set is not disjoint, but it behaves as such because of type and disjoint multiplicity redefinitions

**Scope** ⟨GeneralizationSet⟩

**Information** Problem Issue Type

**Description** As D. Costal and C. Gómez describe in [33]:

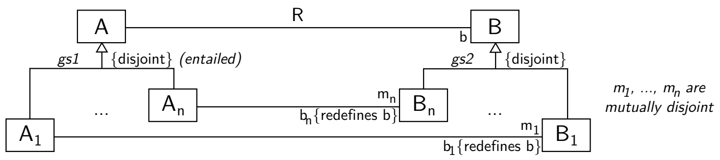


**Figure 6.12.** Disjointness constraint entailed by type and multiplicity redefinitions.

*Theorem 4.5*: Let $A$ and $B$ be two classes and $R$ a binary association between them. Let $b$ be the association end that connects $R$ to class $B$. Let $gs_1$ be a generalization set that specializes superclass $A$ into subclasses $A_1, \ldots, A_n$. Let $gs_2$ be a generalization set that specializes superclass $B$ into subclasses $B_1, \ldots, B_n$. Assume that $b_1, \ldots, b_n$ are multiplicity redefinitions of the end $b$ with their corresponding multiplicities $m_1, \ldots, m_n$, that each $b_i$ is connected to $B_i$ and that the opposite end of $b_i$ is connected to class $A_i$. Then, if each pair of multiplicities $m_i$ and $m_j$, where $1 \leq i \leq n$, $1 \leq j \leq n$ and $i \neq j$, are mutually disjoint (i.e. each intersection is empty), the disjointness constraint of $gs_1$ is entailed by the type and multiplicity redefinitions (see Figure 6.12).

## A generalization set is not disjoint, but it behaves as such because of type and multiplicity redefinitions

**Scope** ⟨GeneralizationSet⟩

**Information** Problem Issue Type

**Description** As D. Costal and C. Gómez describe in [33]:

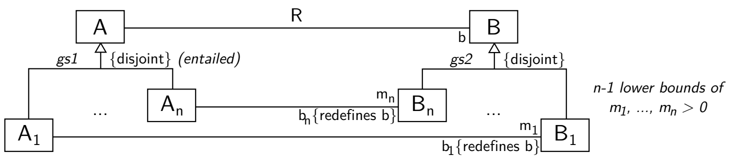


**Figure 6.13.** Disjointness constraint entailed by type and multiplicity redefinitions.

> *Theorem 4.4*: Let $A$ and $B$ be two classes and $R$ a binary association between them. Let $b$ be the association end that connects $R$ to class $B$. Let $gs_1$ be a generalization set that specializes superclass $A$ into subclasses $A_1, \ldots, A_n$. Let $gs_2$ be a generalization set that specializes superclass $B$ into subclasses $B_1, \ldots, B_n$. Assume that $b_1, \ldots, b_n$ are multiplicity redefinitions of the end $b$ with their corresponding multiplicities $m_1, \ldots, m_n$, that each $b_i$ is connected to $B_i$ and that the opposite end of $b_i$ is connected to class $A_i$. Then, if $n-1$ lower bounds of $m_1, \ldots, m_n$ are greater than zero and $gs_2$ is disjoint, the disjointness constraint of $gs_1$ is entailed by the type and multiplicity redefinitions (see Figure 6.13).

## A generalization set is not disjoint, but it behaves as such because of type redefinitions

**Scope** ⟨GeneralizationSet⟩

**Information** Problem Issue Type

**Description** As D. Costal and C. Gómez describe in [33]:

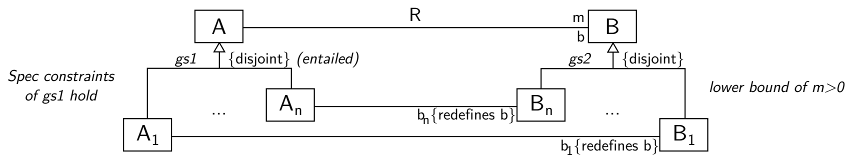


**Figure 6.14.** Disjointness constraint entailed by type redefinitions.

*Theorem 4.2*: Let $A$ and $B$ be two classes and $R$ a binary association between them. Let $b$ be the association end that connects $R$ to class $B$ and let $m$ be its multiplicity. Let $gs_1$ be a generalization set that specializes superclass $A$ into subclasses $A_1, \ldots, A_n$. Let $gs_2$ be a generalization set that specializes superclass $B$ into subclasses $B_1, \ldots, B_n$. Assume that all specialization constraints of $gs_1$ hold. Assume that $b_1, \ldots, b_n$ are type redefinitions of the end $b$, that each $b_i$ is connected to $B_i$ and that the opposite end of $b_i$ is connected to class $A_i$. Then, if the lower bound of $m$ is greater than zero and $gs_2$ is disjoint, the disjointness constraint of $gs_1$ is entailed by the type redefinitions (see Figure 6.14).

## An attribute is repeated among all specific classes of a complete generalization set

**Scope** ⟨gs:GeneralizationSet, n:String⟩

**Information** Problem Issue Type

**Description** As Fowler states in [51], if subclasses are developed independently, or combined through refactoring, you often find that they duplicate features. In particular, certain fields can be duplicates. If they are being used in a similar way, you can generalize them. Doing this reduces duplication in two ways. It removes the duplicate data declaration and allows you to move from the subclasses to the superclass behavior that uses the field.

When we have a *complete GeneralizationSet*, any attribute that is repeated among all its specific classes should be removed from them and should be placed in the superclass instead. As a result, we obtain a simplified model that is easier to understand.

**Example**

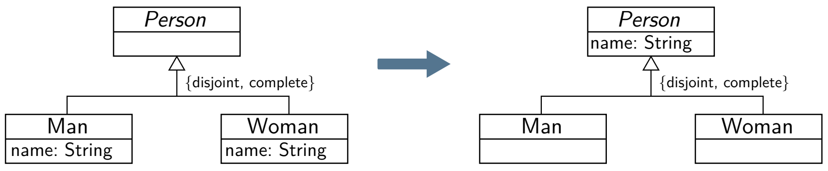


**Figure 6.15.** Example of a complete generalization set with an attribute repeated among all its specific classes.

Figure 6.15 depicts a small conceptual schema with the entity types *Person*, *Man*, and *Woman*. *Man* and *Woman* are defined as a partition of the entity type *Person*, which means that any instance of *Person* is either an instance of *Man* or *Woman*.

Both *Man* and *Woman* have an attribute named *name*. Since all *Persons* have to be either a *Man* or a *Woman*, then all *Persons* have also a name. However, the first version of the conceptual schema does not make it clear, because the attribute is placed in the specific classes instead of the general.

In this situation, a better solution is applying Fowler's pull-up property refactoring and, thus, placing the *name* attribute within the entity type *Person*.

## An attribute repeated among all specific classes of an incomplete generalization set is correct

**Scope** ⟨gs:GeneralizationSet, n:String⟩

**Information** Checking Issue Type

**Description** As Fowler states in [51], if subclasses are developed independently, or combined through refactoring, you often find that they duplicate features. In particular, certain fields can be duplicates. If they are being used in a similar way, you can generalize them. Doing this reduces duplication in two ways. It removes the duplicate data declaration and allows you to move from the subclasses to the superclass behavior that uses the field.

When we have an *incomplete GeneralizationSet*, any attribute that is repeated among *all* its specific is very likely (but not necessary) to be a feature of the general class too. This issue type serves as a warning to the modeler—it is possible the modeler made a mistake and did not include the attribute in the general class, as the domain may suggest.

### Example

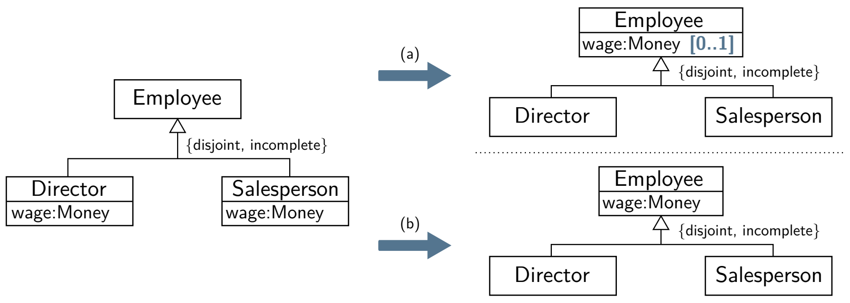


**Figure 6.16.** Example of an incomplete generalization set with an attribute repeated among all its specific classes.

Figure 6.15 depicts a small conceptual schema with the entity types *Employee*, *Director*, and *Salesperson*. *Director* and *Salesperson* represent some (but not all the) specific types of *Employee*. In other words, an instance of *Employee* may be a *Director*, a *Salesperson*, or none of these two. Both *Director* and *Salesperson* have an attribute named *wage*.

Assuming that the original solution is correct—i.e. not all *Employees* have a *wage*, but only *Directors* and *Salespersons*, the first solution proposes to apply a pull-up property refactoring (Solution (a)). In order for the original and alternative (a) schemas to be equivalent, the property's cardinality constraint has to be optional. The resulting schema (a) is simpler, because there is only one attribute in the upper class instead of multiple attributes scattered among the subclasses[4].

Another possibility is Solution (b). Even though not all instances of *Employee* may be a *Director* or a *Salesperson*, it looks very suspicious that both entity types have an attribute named *wage* but an *Employee* does not. This issue type detects this situation and requires the modeler to make sure that the attribute *wage* is not a property that all *Employees* have. If that was the case, she would have to apply Fowler's pull-up property refactoring (whose result is depicted in the upper-right alternative).

### There are two different elements with very similar names

**Scope** ⟨Namespace⟩

**Information** Checking Issue Type

**Description** According to the *CrNameConfusion* critique in [105], when two names in the same namespace have very similar names—i.e. differing only by one character—, it could potentially lead to confusion.

### All attributes in the schema are mandatory

**Scope** ⟨Package⟩

**Information** Checking Issue Type

**Description** The default multiplicity constraint for an attribute in UML is "mandatory". In general, a conceptual schema may contain optional attributes. If all attributes of a conceptual schema are mandatory—i.e. they use the default multiplicity value—, it may be the case that the modeler is not modeling the domain properly.

[4] In order to make this solution semantically equivalent to the original one, the conceptual modeler has to define some additional integrity constraints.

## An attribute is being used instead of an association between its owner and a class when the attribute and the class have the same name

**Scope** ⟨Property⟩

**Information** Checking Issue Type

**Description** The names in a conceptual schema play a key role. They are important, because they help describing the knowledge behind them.

If a conceptual schema has a class $C_1$ with an attribute $A$ named $n$, and another class $C_2$ named $n$, it may be the case that the knowledge behind the attribute $A$ and the class $C_2$ is the same. If this is the case, a better solution is to replace the attribute $A$ in $C_1$ by an association.

**Example**

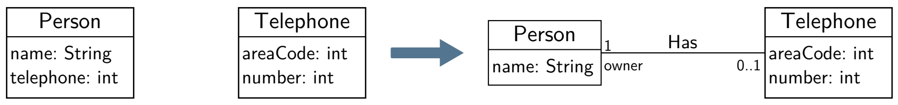


**Figure 6.17.** Example of a conceptual schema where an attribute is being used instead of a relationship type.

Figure 6.17 depicts a conceptual schema with the entity types *Person* and *Telephone*. Both *Person* and *Telephone* have two attributes: *name* and *telephone*, and *areaCode* and *number*, respectively. The attribute *telephone* looks very suspicious, because it may represent the same knowledge as the entity type *Telephone*. If that is the case, the relationship between the two entity types has to explicit using an association, not the attribute. The second version of the conceptual schema fixes this situation by removing the attribute *owner* from the entity type *Company* and creating an explicit association named *Owns* between both entity types.

### The name of an attribute needs to include the name of the class that owns the attribute

**Scope** ⟨Property⟩

**Information** Checking Issue Type

**Description** As stated in [13], the name of an attribute should not contain the name of the entity.

### The type of an attribute is a class

**Scope** ⟨Property⟩

**Information** Problem Issue Type

**Description** In general, the type of a property has to be a datatype. Using a full class as a property's type is discouraged.

## 6.2.5 Naming

Naming guidelines are a special kind of best practices, whose focus is on the names given by conceptual modelers to the elements of a conceptual schema. These issue types are important because the names used in a conceptual schema have a strong influence on the understandability of that schema.

### A binary association does not define any of the three names it may define

**Scope** ⟨Association⟩

**Information** Problem Issue Type

**Description** According to the guidelines we present in [5], a binary association has to define, at least, one of the three possible names it may define: the association's name or a member end's name. If there is no explicit name, we do not know the information abstracted by this association.

### Example

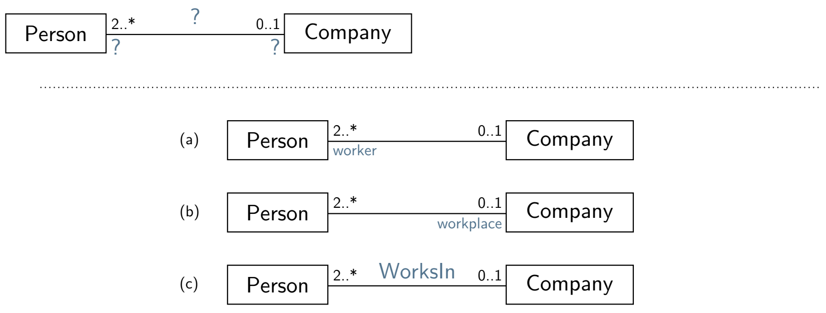


**Figure 6.18.** Example of a binary association with no explicit names and three possible solutions.

According to the guidelines we present in Appendix A, a binary association in UML may define up to three names: the association's name itself and the names of its member ends. Figure 6.18 depicts a binary association between the entity types *Person* and *Company*, without any names on it. As a result, it is unclear which concept from the domain is being modeled:

1. Does a *Person* own a *Company*?
2. Is a *Person* a cusomter of a *Company*?
3. Does a *Person* work in a *Company*?
4. ...

If we, for example, assume that the binary association represents the relationship between a *Person* and the *Company* where she works (that is, the third option), we may define (at least) one of the following names:

(a) a *Company* has two or more *workers*.

(b) a *Person* has zero or one *workplaces*.

(c) a *Person* works in zero or one *Companies*.

## The name of a binary association is not a singular third-person verb phrase

**Scope** ⟨Association⟩

**Information** Problem Issue Type

**Description** Let $R(p_1{:}E_1\ [min_1{,}max_1],\ p_2{:}E_2\ [min_2{,}max_2])$ be a binary association between entity types $E_1$ and $E_2$, playing roles $p_1$ and $p_2$. In UML there may be up to three explicit names related with this association (all are optional): the name of the association $R$ and the names of the two roles $p_1$ and $p_2$.

According to the guidelines we present in [5], the name $R$ of a binary association should be a verb phrase in third-person singular form.

## The name of a binary association is not a verb phrase

**Scope** ⟨Association⟩

**Information** Problem Issue Type

**Description** Let $R(p_1{:}E_1\ [min_1{,}max_1],\ p_2{:}E_2\ [min_2{,}max_2])$ be a binary association between entity types $E_1$ and $E_2$, playing roles $p_1$ and $p_2$. In UML there may be up to three explicit names related with this association (all are optional): the name of the association $R$ and the names of the two roles $p_1$ and $p_2$.

According to the guidelines we present in [5], the name $R$ of a binary association should be a verb phrase in third-person singular form.

## The name of a binary association makes sense

**Scope** ⟨a:Association, n:String⟩

**Information** Checking Issue Type

**Description** Let $R(p_1{:}E_1\ [min_1{,}max_1],\ p_2{:}E_2\ [min_2{,}max_2])$ be a binary association between entity types $E_1$ and $E_2$, playing roles $p_1$ and $p_2$. In UML there may be up to three explicit names related with this association (all are optional): the name of the association $R$ and the names of the two roles $p_1$ and $p_2$.

According to the guidelines we present in [5], (a) the name $R$ of a binary association should be a verb phrase in third-person singular form, written in the Pascal case and (b) the following sentence has to be grammatically well-formed and semantically meaningful:

- If $min_2 = 0$ and $max_2 = 1$:
  [A|An] *lower*($E_1$) *lower*($R$) at most one *lower*($E_2$).
- If $min_2 = 1$ and $max_2 = 1$:
  [A|An] *lower*($E_1$) *lower*($R$) [a|an] *lower*($E_2$).
- If $min_2 = 0$ and $max_2 = *$:
  [A|An] *lower*($E_1$) *lower*($R$) zero or more *lower*(*plural*($E_2$)).
- If $min_2 = 1$ and $max_2 = *$:
  [A|An] *lower*($E_1$) *lower*($R$) one or more *lower*(*plural*($E_2$)).

This issue type requires the modeler to manually check whether the pattern sentence makes sense or not.

## The name of a class is a noun prhase with a plural head

**Scope** ⟨Class⟩

**Information** Problem Issue Type

**Description** According to the guidelines we present in [5], the name of an entity type should be a noun phrase whose head is a countable noun in singular form. The name should be written in the Pascal case.

This issue type points out the fact that the name of this class is not a noun phrase with a singular head.

### The name of a class is not a noun phrase

**Scope** ⟨Class⟩

**Information** Problem Issue Type

**Description** According to the guidelines we present in [5], the name of an entity type should be a noun phrase whose head is a countable noun in singular form. The name should be written in the Pascal case.

### The name of a class is not properly capitalized

**Scope** ⟨Class⟩

**Information** Problem Issue Type

**Description** According to the guidelines we present in [5], the name of an entity type should be a noun phrase whose head is a countable noun in singular form. Moreover, the name should be written in the Pascal case.

Therefore, the name must start with a capital letter.

### The name of a class is uncountable

**Scope** ⟨Class⟩

**Information** Problem Issue Type

**Description** According to the guidelines we present in [5], the name of an entity type should be a noun phrase whose head is a countable noun in singular form. The name should be written in the Pascal case.

This issue type points out the fact that the name of this class is not a noun phrase with a countable head.

## A named element has a name that is too complex to be automatically processed

**Scope** ⟨NamedElement⟩

**Information** Problem Issue Type

**Description** The naming guidelines we present in [5] require the names to follow certain rules. Thus, for example, entity types have to be “noun phrases with countable heads in singular form” and binary associations have to be “verb phrases written in third-singular form”. This rules constraint the variability of names we may assign to entity types and binary associations.

Our name processor has to automatically determine whether the name of an element follows the associated rule or not. However, if the name is too complex—i.e. it is too long, or it has too many complements—, the processor is unable to perform the analysis. When this occurs, it is assumed that the modeler is using a name that is no appropiate for conceptual modeling and, hence, she is required to simplify it.

## The name of a boolean attribute is not a third-person singular verb-phrase

**Scope** ⟨Property⟩

**Information** Problem Issue Type

**Description** According to the guidelines we present in [5], the name of a boolean attribute should be a verb phrase in third-person singular form, written in the Camel case.

## The name of a boolean attribute is not a verb-phrase

**Scope** ⟨Property⟩

**Information** Problem Issue Type

**Description** According to the guidelines we present in [5], the name of a boolean attribute should be a verb phrase in third-person singular form, written in the Camel case.

## The name of a boolean attribute makes sense

**Scope** ⟨p:Property, s:String⟩

**Information** Checking Issue Type

**Description** According to the guidelines we present in [5], the name of a boolean attribute should be a verb phrase in third-person singular form, written in the Camel case. Moreover, the following sentence must be grammatically well-formed and semantically meaningful:

– [A|An] *lower*($E$) *lower*(*withOrNeg*($A$)) [, or it may be unknown].

where the last optional fragment is included only if *min* is equal to zero.

The function *withOrNeg*($A$) extends $A$ with the insertion of "or *negative*($A$)" after the verb of $A$, where *negative*($A$) is the negative form of the verb of $A$. For example:

*withOrNeg*(isDerived) = isOrIsNotDerived
*withOrNeg*(hasChildren) = hasOrHasNotChildren

## The name of a non-boolean attribute is not a noun-phrase in singular form

**Scope** ⟨Property⟩

**Information** Problem Issue Type

**Description** According to the guidelines we present in [5], the name of a non-boolean attribute should be a noun phrase in singular form, written in the Camel case.

## The name of a non-boolean attribute is not a noun-phrase

**Scope** ⟨Property⟩

**Information** Problem Issue Type

**Description** According to the guidelines we present in [5], the name of a non-boolean attribute should be a noun phrase in singular form, written in the Camel case.

## The name of a non-boolean attribute makes sense

**Scope** ⟨p:Property, s:String⟩

**Information** Checking Issue Type

**Description** According to the guidelines we present in [5], the name of a non-boolean attribute should be a noun phrase in singular form, written in the Camel case. Moreover, one of the following sentences must be grammatically well-formed and semantically meaningful:

- If $min = 0$ and $max = 1$:
  [A|An] *lower*($E$) may have [a|an] *lower*($A$).
- If $min = 0$ and $max > 1$:
  [A|An] *lower*($E$) may have zero or more *lower*(*plural*($A$)).
- If $min = max = 1$:
  [A|An] *lower*($E$) has [a|an] *lower*($A$).
- If $min > 0$ and $max > 1$:
  [A|An] *lower*($E$) has one or more *lower*(*plural*($A$)).

Note that *plural*($A$) is a function that gives the plural form of $A$.

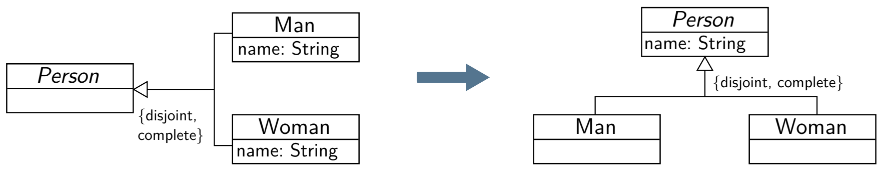


**Figure 6.19.** Example of an issue instance that is related to the graphical facet of a UML conceptual schema.

### The name of a property is not properly capitalized

**Scope** ⟨Property⟩

**Information** Problem Issue Type

**Description** According to several naming guidelines [1, 5, 93, 120, 134], property names must begin with a non-capital letter.

## 6.3 Limitations

The UML is a graphical modeling language. As such, the layout in which elements are distributed, font faces, or element sizes might be relevant. In fact, these "graphical properties" do also have an impact on the quality of the resulting conceptual schema. Thus, for example, in [10] the author proposes several guidelines for distributing and presenting the elements of a UML conceptual schema. These guidelines should be represented as quality issues but, as we shall see, they cannot.

Consider, for example, a layout quality issues stating that "in a generalization, the specific classes have to be drawn below the general class" [10]. Figure 6.19 depicts two versions of the same conceptual schema—the former contains the issue, the latter does not.

Our method is not able to represent this issue. As we have seen in Chap. 5, quality issue types are defined using the metamodel of the language for which they are defined. However, the UML metamodel does not include any components for describing the specific layout of its schemas. As a result, it is impossible to define the previous issue type. Nonetheless, it is important to note that this limitation is not a limitation of our method, but a limitation of the UML metamodel—if the metamodel included graphical-related information of a conceptual schema, we could use it to define the quality issue.

On the other hand, our method has also limitations for defining quality issues that cannot be evaluated looking at the schema directly. Thus, for example, we cannot define issue types that check the *completeness* or *correctness* of a conceptual schema.

## 6.4 Evaluation

This section demonstrates the feasibility of a quality assurance approach based on the catalog—i.e. the *usefulness* of our method. For this, we analyze the presence of quality issues in a set of conceptual schemas. In Sect. 9.2, we compare the number of quality issues detected by this catalog and the number of quality issues detected by two current modeling tools, and we demonstrate that our catalog leads to the detection of more quality issues.

The starting point of the experiment was the random selection of 13 conceptual schemas that were developed by students as part of their final projects during the last year of their Computer Science degree at the *Universitat Politècnica de Catalunya*. The conceptual schemas were defined using different modeling environments. Table 6.2 summarizes, for each conceptual schema, the number of classes, association classes, associations, attributes, and invariants present in the conceptual schema.

According to our method, conceptual schemas can and should be improved as long as they contain issues. Table 6.3 summarizes the number of issues we found for each category and conceptual schema. Clearly, all conceptual schemas have a lot of issues the conceptual modeler should have addressed before releasing it. The high number of quality issues—both problem and checking—found in each schema makes it clear that the catalog we propose in here is relevant and would help conceptual modelers deliver better schemas.

Consider, for example, the three conceptual schemas fragments depicted in

**Table 6.2.** Characteristics of the conceptual schemas developed by students.

| Project | Classes | Assoc-Classes | Assocs | Attributes | Invariants |
|---|---|---|---|---|---|
| P1 | 10 | 1 | 8 | 16 | 3 |
| P2 | 16 | 3 | 9 | 45 | - |
| P3 | 31 | 3 | 23 | 109 | 5 |
| P4 | 18 | - | 10 | 75 | - |
| P5 | 15 | 4 | 6 | 33 | 6 |
| P6 | 11 | 1 | 5 | 91 | 2 |
| P7 | 14 | - | 19 | 11 | 15 |
| P8 | 44 | 6 | 18 | 117 | 9 |
| P9 | 15 | 1 | 15 | 50 | 16 |
| P10 | 18 | - | 27 | 73 | 19 |
| P11 | 20 | 6 | 12 | 42 | 23 |
| P12 | 28 | 6 | 7 | 56 | 23 |
| P13 | 366 | 55 | 264 | 1144 | 386 |

**Table 6.3.** Issues detected by our catalog [7].

| | Syn+ | | BasicProp. | | BestPract. | | Naming | |
|---|---|---|---|---|---|---|---|---|
| Project | Prob | Chk | Prob | Chk | Prob | Chk | Prob | Chk |
| P1 | - | - | 53 | 11 | - | 10 | 13 | - |
| P2 | - | - | 74 | 9 | - | 20 | 15 | 6 |
| P3 | 4 | - | 197 | 21 | - | 50 | 49 | 8 |
| P4 | - | - | 87 | 9 | 15 | 24 | 24 | - |
| P5 | - | - | 78 | 8 | - | 20 | 21 | 2 |
| P6 | - | - | 68 | 8 | - | 11 | 16 | - |
| P7 | 1 | - | 91 | 9 | 3 | 14 | 24 | 3 |
| P8 | - | - | 170 | 41 | 3 | 52 | 45 | 13 |
| P9 | - | - | 106 | 6 | - | 22 | 21 | - |
| P10 | 2 | - | 171 | 16 | - | 33 | 38 | 21 |
| P11 | 13 | - | 104 | 14 | 14 | 29 | 25 | 11 |
| P12 | 6 | - | 87 | 24 | 14 | 34 | 10 | 1 |
| P13 | 221 | - | 2031 | 395 | 271 | 407 | 284 | 21 |

Fig. 6.20. As we shall see, these small fragments contain several quality issues that were detected by our method.

On the one hand, fragments (a) and (b) conceptualize the concept of a *Message* within a system where users can post their comments/messages. According to both conceptualizations, a *Message* has an identifier (attributes *messageId* and *id*), it has some content (attributes *text* and *content*), and it may reply to another *Message* (recursive association). Fragment (a) contains the following checking issues:

- The recursive association *RepliesTo* does not include any constraints regard-

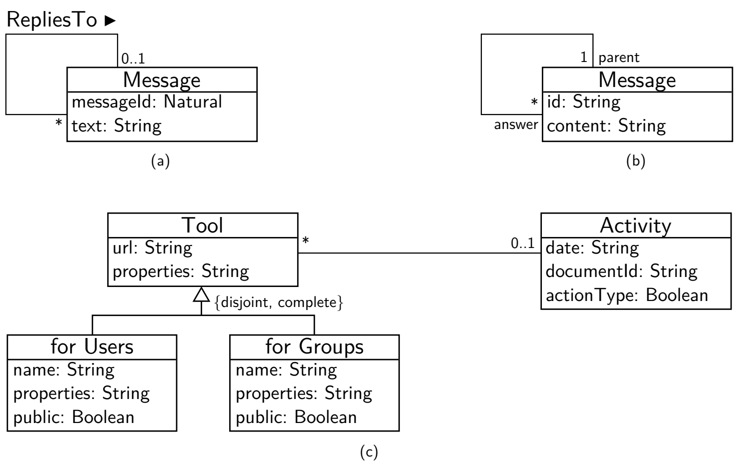


**Figure 6.20.** Fragments of three conceptual schemas, each fragment with several quality issues.

ing the properties of symmetry, transitivity, or reflexivity. In this particular case, they were not defined by the modeler, even though they seem necessary—e.g. a *Message* cannot reply to itself.

- The attribute *messageId* includes the name of the entity type (*Message*). In general, this situation should be avoided.
- If the name of the previous attribute is actually *id* instead of *messageId*, then another issue would have raised: the one asking the modeler to make sure whether a *Message* needs an attribute named *id*.

whereas fragment (b) contains the following checking issues:

- The recursive association does not include any constraints regarding the properties of symmetry, transitivity, or reflexivity.
- The recursive association has one member end that is mandatory, which looks very suspicious (do all messages have to reply to another message?
- Does the entity type *Message* need an attribute named *id*?

On the other hand, fragment (c) corresponds to a conceptual schema of a collaborative website. The fragment depicts the entity type *Activity* which is somehow related to *Tool*. *Tools* can either be *for Users* or *for Groups*. The schema also contains several attributes for each entity type. In our opinion, this tiny fragment is very complicated to understand because of the problem issues it contains:

- All attributes in the entity types *for Users* and *for Groups* are repeated, despite they participate in a disjoint generalization set. Clearly, these attributes should be removed from these entity types and defined in *Tool* instead.

- If all attributes are removed from the specific classes (as the previous issue points out) and taking into account that they are not related to any other classes, another issue would be raised pointing out that, probably, these specific classes are unnecessary. A better solution would probably be an enumerated attribute in the superclass that classifies tools into the appropriate type—i.e. “for users” and “for Groups”.

- The attribute *properties* is specified both in the superclass and the specific classes.

- The entity types *for Users* and *for Groups* do not seem to follow any naming guidelines in general. Their names are not properly capitalized and they are not noun phrases. Probably, the name the conceptual modeler should have used was *ToolForUsers* and *ToolForGroups*.

- The association between *Tool* and *Activity* does not define any of the three possible names it may define. As a result, it is completely unclear the relationship from the domain that is being conceptualized (does a tool “generate” an activity? Does an activity “use” any tools? Does an activity “require” tools? . . . )

- The entity type *Activity* contains a *Boolean* attribute named *actionType*, which does not follow our naming guidelines for boolean attributes—i.e. it has to be a verb phrase written in third-person singular form.

## 6.5 Summary

In Chap. 4, we have introduced the notion of quality issues and the three phases of our method: (i) the definition of quality issue types using a specific formalization, (ii) the compilation of quality issue types in a catalog, and (iii) the usage of quality issue types during the development of a conceptual schema. In Chap. 5, we have focused on the first phase of our method and we have presented a formalization of quality issues.

This chapter has focused on the second phase of our method. Its main contribution has been to present the catalog of quality issues we had built so far. We have used the catalog to analyze the *expressiveness* and the *usefulness* of our method. For analysis and presentation purposes, we have also introduced a classification of issue types based on its source. The results of this chapter were published in [6].

Throughout Sect. 6.2 we have presented the list of issue types our catalog contains. These issue types have been obtained from the literature and current modeling environments. The compilation of this catalog demonstrates that our formalization is powerful enough to express almost all existing quality issues with it. We have also seen in Sect. 6.3 that there are some issues for the UML—like, for example, those that are related to the layout of a conceptual schema—that cannot be expressed using our formalization. We have discussed the rationale behind this problem and realized that it is not a limitation of our formalization, but a limitation of the UML itself: our formalization can only define issues based on the information available in the metamodel.

In Sect. 6.4 we have evaluated the usefulness of our method. In order to do so, we analyzed the presence of quality issues in a set of conceptual schemas. The conceptual schemas had been randomly selected and were developed by students as part of their final projects during the last year of their Computer Science degree at the *Universitat Politècnica de Catalunya*. We have seen that all conceptual schemas contain several quality issues. All these issues could have been avoided if the conceptual modelers had been aware of them.

It is much more difficult to judge oneself than to judge others.
A. de Saint-Exupéry, *The Little Prince*

# 7 A Conceptual Modeling Assistant based on Quality Issues

Throughout the previous chapters of this thesis, we have discussed the importance of developing high-quality conceptual schemas. We have claimed that, in order to do so, conceptual modelers may benefit a lot from the support a modeling tool can provide. In particular, we envision a Conceptual Modeling Assistant (CMA) as an Integrated Development Environment (IDE) that analyzes and criticizes the conceptual schema under development so that its modeler can improve it. Thus, our prototype tool [2, 3] implements the detection of conceptual schema quality issues. As a result, it provides useful feedback by pointing out the issues the conceptual schema has[1].

The aim of this chapter is to describe the implementation of our method

[1]Please note our prototype tool implement issue detection only. A complete implementation of our method would also include the actions a modeler can perform to fix issues.

in a real IDE. Section 7.1 presents an overview of Eclipse. Eclipse is an IDE that can be easily extended by means of plugins. The section also describes the architecture of our plugin and its main components—i.e. an *Issue Type Manager*, which is responsible of downloading and managing the available issue types; an *Issue Processor*, which is responsible of computing the issues a conceptual schema has; and an *Issue View*, which is the graphical component responsible of showing feedback to the modeler. Section 7.2 outlines the publication of the catalog presented in Chap. 6. In Sect. 7.3 we discuss how our method can be integrated in the development of a conceptual schema by providing a continuous non-disruptive feedback view. Finally, Section 7.4 summarizes this chapter.

## 7.1 Overview and Architecture

We conceived our prototype tool [2, 3] as a plugin for Eclipse—an open-source IDE that consists in a core project that includes a generic framework for tool integration and a Java development environment built using it, and hundreds of plugins that extent its basic functionalities [32]. Eclipse can be used as a UML modeling environment by installing the *UML2 Tools* [41].

In this section, we introduce the Eclipse Platform briefly, taking special care to its architecture and the plugin mechanism it provides. We also present the architecture of our plugin and the set of plugins we implemented in order to integrate our method in Eclipse. Finally, we introduce an OCL interpreter available in Eclipse and how it can be extended to evaluate OCL-defined quality issue types.

### 7.1.1 Introduction to the Eclipse Platform

Eclipse is an open-source IDE launched by Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft and Webgaint [42], whose purpose is to provide a highly integrated tool platform. It is basically a core project that includes a generic framework for tool integration and a Java development environment built using it. Other projects extend the core framework integrating additional features and thus supporting specific kinds of tools, methods, and development environments. Since Eclipse is implemented in Java, it can run on many operating systems, including Linux, Mac OSX, and Windows [118].

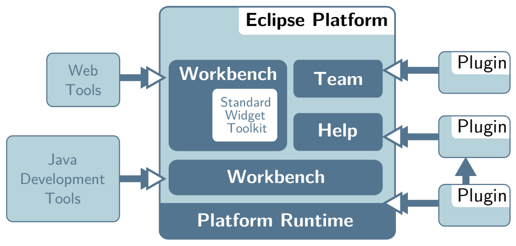


**Figure 7.1.** The Eclipse's Architecture—the kernel and its plugin system.

Figure 7.1 depicts the architecture of the Eclipse Platform. The basic component in Eclipse is a *plugin*: whenever a developer wants to add a new functionality in Eclipse, she has to create one or more plugins that contribute that functionality. The Platform Runtime Engine is responsible for discovering and running plugins. As expected, a plugin includes everything needed to be run, both regular assets—such as the Java code, images, internationalization texts, and so on—and a "manifest file" that identifies and describes the plugin—i.e. it specifies, among other things, its dependencies and how it can be extended by other plugins. What is interesting about this architecture is that plugins can be extended and refined by other plugins, which makes it easy for developers to create tools by reusing other people's work.

## Eclipse as a UML Modeling Environment

The Eclipse Modeling Framework (EMF) aims to create and promote a model-based development technology under the Eclipse Platform [118, p. 22]. EMF provides the basic framework for modeling. Other modeling sub-projects built on top of the EMF framework provide additional capabilities such as model transformation, database integration, or graphical editors, among others.

The Model Development Tools (MDT) project converts Eclipse into a modeling environment. In particular, Eclipse can be used as a UML modeling environment by installing an MDT's subproject called *UML2 Tools*, a set of editors that enable Eclipse to view and edit UML models [41]. Moreover, MDT does also provide a parser and interpreter for OCL constraints and expressions on any EMF-based metamodel.

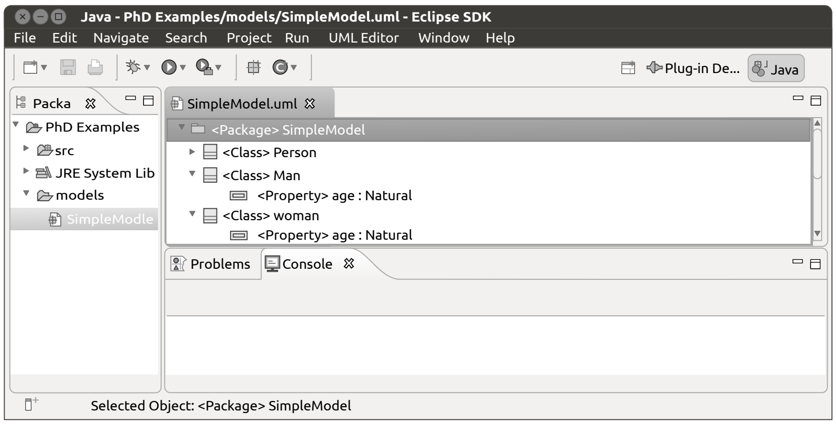


**Figure 7.2.** Screenshot of Eclipse with a UML model editor opened.

We chose to develop our prototype tool on top of Eclipse because of the UML support it offers, as well as because of the popularity and the maturity of this project. Furthermore, the open-source MDT's OCL interpreter permits a straightforward evaluation of issue types defined in OCL and can be extended to support (as we shall see in the next section) more complex operations easily.

### 7.1.2 CMA's Architecture

We have already said that we envisioned our prototype implementation of a CMA as an Eclipse extension. As we have just seen, Eclipse uses a plugin system to include new functionalities. Figure 7.3 depicts the architecture of our prototype. We can see that the CMA's architecture consists in two parts basically. On the one hand, there is a remote server that contains a catalog of all available quality issue types[2] which, in principle, are described using OCL. On the other hand, there is the Eclipse plugin that loads this remote catalog into the environment and, by implementing the algorithm presented in Sect. 5.1, detects the issues that are present in the conceptual schema that is being developed.

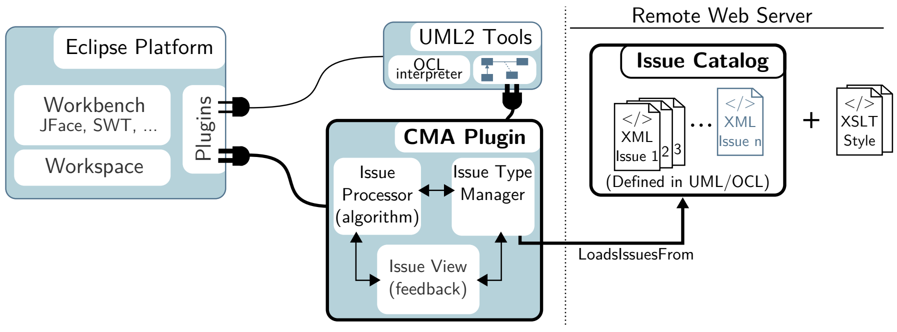


**Figure 7.3.** CMA's Architecture—an Eclipse plugin that extends the UML2Tools.

The *CMA plugin* is divided in the following three components:

**The *Issue Type Manager*** is responsible of downloading the issue type definitions available in the catalog. As described in Sect. 5.1, an issue definition includes the *scope*, the *applicability* and *issue conditions*, the *precedents*, and so on. Moreover, issue definitions included in the catalog also contain useful information for conceptual modelers, such as, for instance, a *description* of the issue type or the *label* to be shown when an issue instance is detected.

**The *Issue Processor*** is responsible of computing issue instances. This component implements the algorithm presented in Sect. 5.1[3] and keeps track of all issue instances present in the conceptual schema. As expected, the issue processor computes the instances of an issue type if, and only if, it has already computed the instances of its precedents.

**The *Issue View*** simply displays the list of issue instances that are present in the conceptual schema. This view provides relevant feedback about each issue, such as a definition of the issue type, the scope of the issue instance, or the actions that may solve it.

[2]See Sect. 7.2 for further details on how this catalog was implemented.

[3]Our prototype tool implements an incremental algorithm too, as described in Chap. 8. This alternative implementation provides better response times when dealing with larger conceptual schemas and many issue types.

In the following, we provide a more detailed description of these components.

### 7.1.3 The Issue Type Manager

In Chap. 6 we have introduced the catalog of quality issues we have compiled given the current state of the art. As we shall see in Section 7.2, this catalog has been published in a public web server for anyone to access it. The *Issue Type Manager* is responsible of downloading the issue type definitions available in this catalog, parse their information—i.e. names, descriptions, applicability and issue conditions, and so on—, and load them into the CMA.

It also permits the conceptual modeler to enable or disable issue types, so that the CMA will only evaluate those that are relevant for her project. In Sect. 7.3 we describe how the conceptual modeler can select the relevant issue types.

### 7.1.4 The Issue Processor

As we have already stated, the *Issue Processor* is responsible of computing issue instances. This component implements the algorithm presented in Sect. 5.1 and is responsible of keeping track of any issue instance present in the conceptual schema.

The algorithm for computing issue instances can be used as a batch process or periodically. Our CMA implements the algorithm as a background process that is executed periodically, after a certain amount of time has passed by[4]. The time interval between executions may be configured by the conceptual modeler. When the timeout triggers, the background process computes the issue instances that appeared or disappeared due to the changes the conceptual modeler performed after the last execution. Since the issue computation is only performed once in a while as a background process, this computation does not affect, in general, the responsiveness of the conceptual modeling tool.

[4]When testing the prototype implementation, we detected that some low-spec machines had problems in executing the background process in a non-disruptive manner. Therefore, we permitted the conceptual modeler to disable the periodic evaluation of issue types and perform an on-demand evaluation instead.

Given a certain issue type, the algorithm has to evaluate its applicability and issue conditions in order to determine the issue instances of this type. Our CMA is able to deal with two kinds of issue types: (a) those whose applicability and issue conditions are defined using OCL, and (b) those that are computed by an external, "black-box-like" tool. In the following, we describe OCL issue types and Black-Box issue types in more detail.

### OCL Issue Types

In order to determine the issue instances of a particular issue type, it is necessary to evaluate both the applicability and the issue conditions. Assuming that these conditions are defined using OCL, a tool that implements the issue types included in our catalog can either:

- translate the OCL expressions into a platform-specific programming language (such as, for example, Java), or
- use an OCL interpreter to evaluate the OCL expressions over an instantiation of the UML.

The Issue Processor we implemented in our tool uses the MDT's OCL interpreter to evaluate OCL expressions. As a consequence, our CMA prototype can directly benefit of any new issue type we define in OCL: it is only necessary to include it in the catalog, and our CMA is capable of downloading its definition using the Issue Type Manager and evaluate the OCL expressions when required. Moreover, as we shall see in Chap. 8, we also implemented an incremental method for the efficient evaluation of OCL expressions, providing instant feedback to conceptual modelers.

The experience we acquired building our catalog demonstrates that almost all issue types can be defined using regular OCL. Nonetheless, there are certain situations in which defining (a part of) an issue type in OCL can be extremely complicated. In the following, we describe how to extend MDT's OCL interpreter—i.e. we extend the OCL language by adding new, additional functions that simplify the resulting OCL expressions.

**How to Extend MDT's OCL Interpreter**

Consider, for example, a naming guideline stating that "the name of class has to be a noun phrase whose head is a countable noun". We could divide this guideline into the following two issue types:

1. The classname is a noun phrase.
2. The head of the classname is a countable noun (assuming the previous issue type is a precedent of this one).

The definition of both issue types using natural language is straight-forward. Probably, all conceptual modelers can understand what each issue type means and, therefore, check whether the name of a certain class has issues or not. Unfortunately, defining these issue types in OCL is not easy, because we need to (a) determine whether a String is a noun phrase or not, and (b), assuming it is a noun phrase, find its head and determine whether it is countable or uncountable. The easiest way to overcome this problem is to assume we have some "helper operations" that can deal with these complex situations. Thus, for example, we may agree that we have the following String operations available:

```
context String::isNounPhrase() : Boolean
  post: -- it returns true if the string is a noun phrase;
        -- false otherwise.

context String::getHead() : String
   pre: -- self is a noun phrase
  post: -- it returns the head of the Noun Phrase

context String::isCountable() : Boolean
   pre: -- self is a noun
  post: -- it returns true if self is countable;
        -- false otherwise
```

which can be used in an OCL expression. By using them properly, we can easily write the previous two issue types as follows:

$I_1$ : The classname is a noun phrase.

```
φ1(Class) = true
ρ1(Class) = not self.name.isNounPhrase()
```

$I_2$ : The head of the classname is a countable noun[5].

```
$\phi_1$(Class) = true
$\rho_1$(Class) = not self.name.getHead().isCountable()
```

The MDT's OCL plugin allows the customization of its evaluation environment. That is, we can define additional variables and operations that can be used within the context of an OCL expression. In our case, we can define one or more *helper operations* whose body is defined in Java. Using a full programming language to define a helper operation is very powerful and permits us to overcome almost any trouble we may find such as, for example, finding the head of a noun phrase or accessing an online dictionary to determine the grammatical form of a word.

In principle, there are no limitations on what can be done within a helper operation: MDT's OCL plugin provides whole access to the UML model through a Java interface. In our thesis, however, we only define helper operations that (a) deal with basic data types (i.e. *String*, *Boolean*, *Integer*, and so on) and (b) all the information required for the operation's execution is available through its parameters (i.e. it does not require access to the UML schema). Note that all the examples we have seen so far follow this rule. If we need to define a "complex" issue type that requires more powerful operations, then we have to use *Black-Box Issue Types*, which are discussed below.

### Black-Box Issue Types

There are some situations in which defining an issue in OCL is not feasible or practical. On example is, for instance, the following quality issue type: "The OCL expression of a *Constraint* does not compile". Clearly, we want our OCL constraints to be correct, and their compilation is the first step in this direction.

Any issue type for which a tool that detects its instances already exists (or can be easily implemented) can be integrated into our CMA using *Black-Box Issue Types*. Black-Box issues are regular issue types whose applicability and issue conditions are described using (probably) natural language, and their implementation and computation are performed in an external tool. In our example, we

[5]Being $I_1$ is an instance precedent.

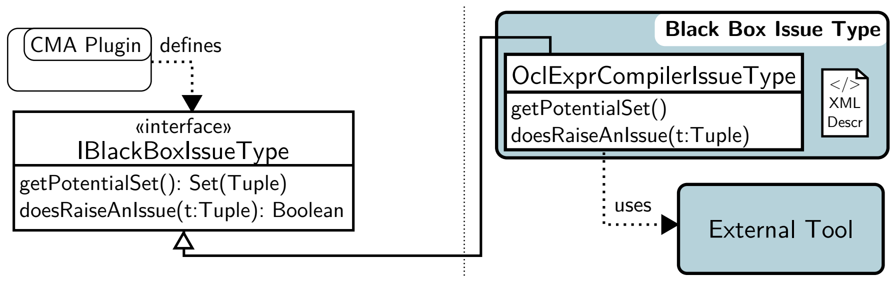


**Figure 7.4.** Integration of Black Box Issue Types in our CMA using an interface.

know that there are many tools that, given a conceptual schema and an OCL expression, compile the expression and report the encountered errors (if any). Eclipse itself packages such a tool.

Figure 7.4 depicts the architecture of our solution. Basically, the CMA defines a simple interface *IBlackBoxIssueType* with two methods[6]: *getPotentialSet* and *doesRaiseAnIssue*. When a method engineer wants to add a Black-Box Issue Type inside our CMA, she simply implements this interface and connects her tool to the concrete class. The former method returns the potential set (as described in Sect. 5.1) and the latter evaluates, for each tuple in the potential set, whether it raises an issue of the associated issue type.

### 7.1.5 The Issue View

The *Issue View* displays the list of issue instances that are present in the conceptual schema. It also offers some relevant and useful feedback for the conceptual modeler. These issues may be classified into *categories* (such as, for example, "priority") for the ease of use. By using this list, the conceptual modeler can *check* or *accept* a particular issue instance, view the *description* of the issue type to gather more knowledge about the issue at hand, or review the *actions* that may solve the issue. In Sect. 7.3 we can see this view in action.

[6]Our interface provides a few additional methods and hooks to provide full access to the Eclipse environment. As a result, the method engineer can obtain all the required information the external tool needs to perform its computations.

## 7.2 Online Catalog (for OCL Issue Types)

Chapter 6 presented a catalog of quality issue types. This catalog included more than 60 issue types from the literature, which we defined using our method. We believe that such a catalog has a lot of value to the conceptual modeling community and hence we granted anyone access to this catalog, uploading it to a public web server.

Each issue type included in our catalog is defined in XML. The Extensible Markup Language (XML) is a simple, very flexible text format. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere [130].

Each XML entry in our catalog includes the different elements of the formalization presented in Sect. 5.1, as well as some additional meta-data like the *name*, the *description*, the *references* where it is originally described, or the *category*, among others, of an issue type.

Consider, for example, the issue "a generalization is redundant". The XML file for this issue in our catalog is the following:

```
<issue
   nid="038"
   id="BEST-PRACTICE-gen-redundant"
   kind="problem"
   acceptance="non-acceptable">

   <context>Generalization</context>
   <title>A generalization is redundant</title>

   <label>
      Specialization ''__self.specific.printName()__ extends
      __self.general.printName()__'' is redundant
   </label>

   <categories>
      <category>Priority::Mandatory</category>
      <category>Source::Best Practice</category>
   </categories>

   <description>
```

```
        In UML, multiple inheritance is allowed. However,
        generalizations have to be carefully used, in order
        to avoid conflicts. In this particular case, it may
        be the case that a generalization _A_ IsA _B_ is
        redundant, because there is an indirect inheritance
        hierarchy between _A_ and _B_ (for instance, _A_ IsA
        _C_ IsA _B_).
    </description>

    <applicability-condition>
        true
    </applicability-condition>

    <issue-condition>
        Generalization.allInstances()->select( g |
            g <> self and
            g.specific = self.specific and
            ( g.general.allParents()->includes(self.general)
              or
              g.general = self.general )
        )->notEmpty()
    </issue-condition>


    <precedents>
        <precedent type="global">
            Non directed and acyclical hierarchies
        </precedent>
    </precedents>

    <issue-actions>
        <issue-action
                name="removeGeneralization"
                params="g:Generalization">
            Remove the redundant generalization.
        </issue-action>
        <issue-action
                name="removeAnotherGeneralization"
                params="g:Generalization, h:Generalization">
            If there is an indirect path between g.general
            and g.specific, remove the generalization h that
            participates in the indirect path.
        </issue-action>
    </issue-actions>
</issue>
```

The applicability and issue conditions are defined in OCL. The usage of an XML representation provides two key benefits: on the one hand, the catalog and the specification of an issue type can be downloaded and parsed by an IDE automatically. On the other hand, they can be presented in a user-friendly manner by means of Extensible Stylesheet Language Transformations (XSLT) sheets, which allow these XML files to be browsed by conceptual modelers and practitioners using a web browser.

Figure 7.5 shows our catalog as it is seen by a conceptual modeler that is browsing our catalog using a web browser. As we can see, there is a list of all issue types available in our catalog. When the conceptual modeler clicks on one issue type, the information is presented in a human-friendly fashion, with full formatted texts, links to external references, and so on.

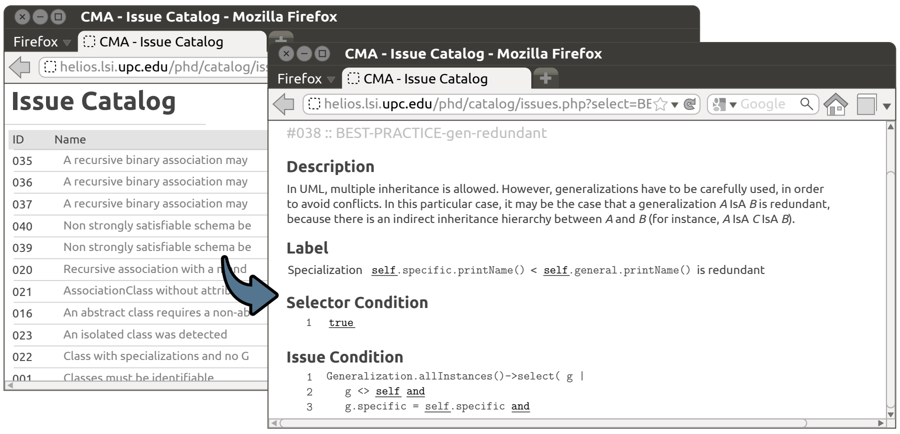
## 7.3 User Interaction

One of the premises of our work is to provide assistance in a non-disruptive manner. Therefore, we envision our CMA as passive element within an IDE that analyzes the schema without disturbing the conceptual modeler.

In Sect. 4.2, we have presented an overview of our method. The method is divided in three main phases: (i) formalization of issue types, (ii) compilation of a quality issue catalog, and (iii) using quality issues in practice. The CMA implements the third phase of our method which require:

- selecting the issue types that are relevant for the conceptual schema under development,
- analyzing the conceptual schema in order to provide feedback on the issue it contains, and
- fixing these issues.

In this section, we describe how a conceptual modeler interacts with the CMA.


CMA - Issue Catalog - Mozilla Firefox
Firefox
CMA - Issue Catalog
helios.lsi.upc.edu/phd/catalog/is
CMA - Issue Catalog - Mozilla Firefox
Firefox
CMA - Issue Catalog
helios.lsi.upc.edu/phd/catalog/issues.php?select=BE
Google

Issue Catalog

| ID | Name |
|---|---|
| 035 | A recursive binary association may |
| 036 | A recursive binary association may |
| 037 | A recursive binary association may |
| 040 | Non strongly satisfiable schema be |
| 039 | Non strongly satisfiable schema be |
| 020 | Recursive association with a m nd |
| 021 | AssociationClass without attrib |
| 016 | An abstract class requires a non-ab |
| 023 | An isolated class was detected |
| 022 | Class with specializations and no G |
| 001 | Classes must be identifiable |

#038 :: BEST-PRACTICE-gen-redundant

**Description**

In UML, multiple inheritance is allowed. However, generalizations have to be carefully used, in order to avoid conflicts. In this particular case, it may be the case that a generalization *A* IsA *B* is redundant, because there is an indirect inheritance hierarchy between *A* and *B* (for instance, *A* IsA *C* IsA *B*).

**Label**

Specialization `self.specific.printName() < self.general.printName()` is redundant

**Selector Condition**

```
true
```

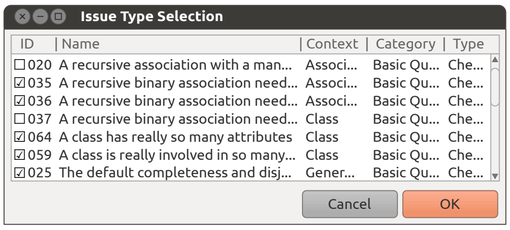
**Issue Condition**

```
Generalization.allInstances()->select( g |
  g <> self and
  g.specific = self.specific and
```

**Figure 7.5.** Screenshot of a web browser displaying an issue type from our catalog.

## 7.3.1 Selecting the Relevant Issue Types

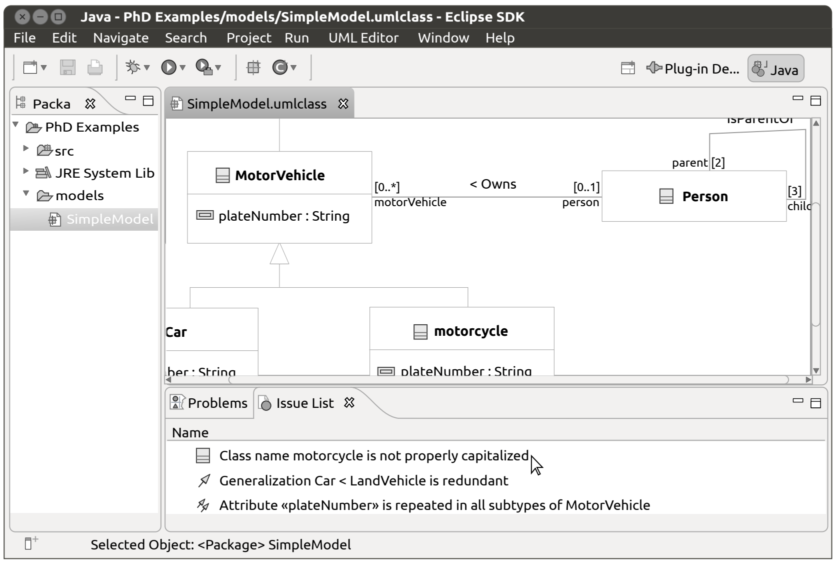
According to our CMA's architecture, the set of quality issue types the CMA is able to deal with is defined in an external server. This solution offers an endlessly growing, up-to-date catalog of issue types. However, when defining a conceptual schema, not all issue types are relevant—the conceptual schema has to satisfy a subset of them only. When the conceptual modeler starts the CMA, she is able to select which issue types her conceptual schema has to satisfy.



| ID | Name | Context | Category | Type |
|---|---|---|---|---|
| ☐ 020 | A recursive association with a man... | Associ... | Basic Qu... | Che... |
| ☑ 035 | A recursive binary association need... | Associ... | Basic Qu... | Che... |
| ☑ 036 | A recursive binary association need... | Associ... | Basic Qu... | Che... |
| ☐ 037 | A recursive binary association need... | Class | Basic Qu... | Che... |
| ☑ 064 | A class has really so many attributes | Class | Basic Qu... | Che... |
| ☑ 059 | A class is really involved in so many... | Class | Basic Qu... | Che... |
| ☑ 025 | The default completeness and disj... | Gener... | Basic Qu... | Che... |

**Figure 7.6.** Mockup of our CMA showing the "Issue Type Selection" dialog.

Figure 7.6 shows a mockup of the CMA's "Issue Type Selection" dialog. This dialog lists the issue types defined in the catalog and permits the conceptual modeler to select those that are relevant. For each issue type, the modeler can



Class name motorcycle is not properly capitalized
Generalization Car < LandVehicle is redundant
Attribute «plateNumber» is repeated in all subtypes of MotorVehicle

Selected Object: <Package> SimpleModel

**Figure 7.7.** Screenshot of our CMA (built on top of Eclipse) with the conceptual schema depicted in Fig. 4.1.

see at a glance whether it is enabled or not, its name, its scope, its category, and its kind.

### 7.3.2 Defining the Model and Obtaining Feedback

Once the conceptual modeler has selected the relevant issue types, she can start defining her schema. Our implementation does not modify Eclipse's default workflow for defining conceptual schemas. The detection of issues is performed as a background process that runs constantly. Therefore, the conceptual modeler can work on her model and modify it as she pleases without the CMA getting into her way.

Whenever an issue is introduced in the schema, it is shown in a specific view within the Eclipse Platform. This view, which can be shown and hidden according to the conceptual modeler's will, provides all necessary feedback to identify and fix any present issue.

As an example, consider the conceptual schema presented in Sect. 4.1.1, which has been modeled using Eclipse (Fig. 7.7). The *Issue View* presents a list with all the issues the schema has like, for example, “class name motorcycle is not properly capitalized” or “attribute ‘plateNumber’ is repeated in all subtypes of *MotorVehicle*”.

### 7.3.3 Fixing Issue Instances

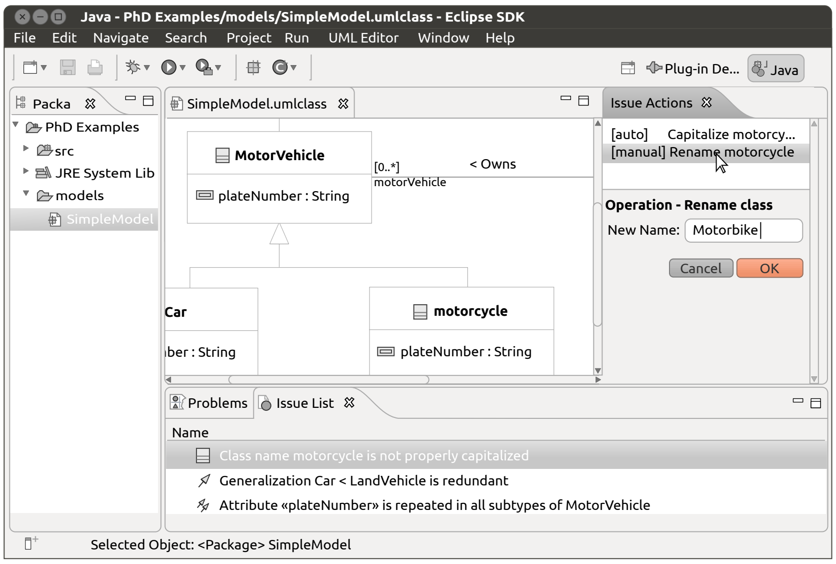
When the conceptual modeler wants to fix some of them, she only needs to click on an issue in the issue list in order to view the list of automatic or manual actions she can perform to fix it. Then, she selects the most appropriate action and executes it. Once the issue has been fixed, it disappears from the list.

The mockup depicted in Fig. 7.8, shows two issue actions for the issue type “The name of a class is not properly capitalized”. The conceptual modeler has selected a manual operation, which triggers a new view that requires the conceptual modeler to introduce any data required to perform the operation (such as, for example, the new name). This user interaction is like the one provided by *ArgoUML*, and requires the engineers of a specific IDE to provide some sort of user interface for manual operations that guide conceptual modelers.

## 7.4 Summary

In this chapter we have demonstrated the feasibility of our method by implementing a prototype tool of a Conceptual Modeling Assistant. We built this prototype as a plugin for the Eclipse platform—one of the most popular, open-source development environments available. The plugin is divided in three main components: an *Issue Processor* that implements the algorithm presented in Sect. 5.1. The algorithm is executed periodically as a background process to detect the issues that are present in a conceptual schema. This component uses the MDT’s OCL framework to evaluate OCL expressions. We have also described how MDT’s OCL framework can be extended to support more complex operations and, thus, ease the definition of complex issues.

Secondly, we have seen the *Issue Type Manager*, the component responsible



[auto] Capitalize motorcy...
[manual] Rename motorcycle

Operation - Rename class
New Name: Motorbike

Class name motorcycle is not properly capitalized
Generalization Car < LandVehicle is redundant
Attribute «plateNumber» is repeated in all subtypes of MotorVehicle

Selected Object: <Package> SimpleModel

**Figure 7.8.** Mockup of Eclipse showing the available actions that may fix an issue.

of downloading and managing the issue types Eclipse is able to deal with. This component downloads the definition of issue types from a public catalog. The issues included in this catalog are defined using XML files. We have also seen that, by using XSLT transformations, we are able to make this catalog user-friendly.

Finally, the third component of our plugin is the *Issue View*. This view is responsible of providing feedback to the modeler, as well as the set of actions that may fix each of them. Each time the *Issue Processor* executes the algorithm, the set of issues the schema has is updated and displayed in the *Issue View*.

"Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."

L. Carroll, *Alice in Wonderland*

# 8 Incremental Evaluation of OCL-defined Quality Issues

According to the method we present in this thesis, the quality of a conceptual schema can be measured in terms of the number of issues it contains: the less issues, the more quality. Conceptual modelers are responsible of ensuring their conceptual schemas match the expected degree of quality, which means that their conceptual schemas must not contain any issues at all. Nonetheless, having issues temporarily in a conceptual schema under development is not a problem, as long as they get fixed sometime in the future.

During the development of a conceptual schema, new quality issues may arise and old ones may be solved because of the changes the conceptual modeler performs to that schema. Since the conceptual modeler is the one responsible for correcting these issues, she has to be aware of their presence. Given a conceptual schema under development, the issues it contains can be computed (1) *when re-*

*quested* by the conceptual modeler (usually, when it is complete); (2) in a *periodic* fashion, after a pre-established time interval goes by; or (3) *continuously*, that is, every time the schema is modified. In evaluations (1) and (2), a process—which can be manual or automated—analyzes *the whole schema* and reports all quality issues that have been found. In (3), on the other hand, an automated process analyzes *the changes to a schema* and points out the quality issues that have arisen *due to those changes*.

In this chapter we discuss the importance of an incremental evaluation in order to provide *continuous feedback instantly*. First, Section 8.1 introduces the main ideas behind incremental evaluation. It presents a simple conceptual schema with several naming-related issues and discusses how these issues can be detected using (a) a batch process executed periodically and (b) an incremental method. Section 8.2 reviews the available literature on incremental evaluation, paying special attention to those works related to incremental evaluation of OCL expressions. Next, in Sect. 8.3 we formally describe incremental methods in general. We also discuss how one specific method (in particular, the one presented in [44]) can be adapted to our formalization of issue types. In this sense, we present a new version of the Alg. 1 in Sect. 5.4 that computes issue instances. Section 8.3.4 evaluates the improvements we obtain in terms of response time when using incremental methods instead of simple batch processes. Finally, Section 8.4 summarizes the key points of this chapter.

## 8.1 Understanding Incremental Evaluation

Conceptual modeling is a complex activity. In this thesis we present a method for defining an treating conceptual schema quality issues. Its ultimate goal is, as we already know, to improve the quality of a conceptual schema. Unfortunately, conceptual modelers can easily get overwhelmed by the difficulties entailed by fixing and keeping track of quality issues. Thus, we believe that a modeling tool that automates the management of quality issues can be very helpful, especially if it is able to provide feedback instantly as issues arise.

In Chap. 5, we presented our formalization to define quality issues in a uniform way, along with an algorithm that computes the issue instances that are present in a conceptual schema. We believe that our work is the first step towards this necessary integration of quality issues into modeling environments. Chapter 7 describes a prototype integration of our method into Eclipse. In order to obtain continuous feedback, our prototype integrates the algorithm presented in Sect. 5.4 as a batch process that may be executed periodically. As a result, the conceptual modeler can get relevant feedback in "real-time"—i.e. whilst the schema is being developed. Unfortunately, the checking process may take too long to complete when there is a large number of elements in the conceptual schema as well as a large number of quality issue types to consider[1], making the tool unresponsive and, thus, worsening the user experience. In order to overcome this problem and to achieve an immediacy in the process, an incremental approach is, as we shall see, very important.

This section introduces incremental methods and outlines the main advantages they offer with respect to non-incremental methods. In order to do so, we use the conceptual schema first introduced in Sect. 4.1.1 as a running example. In Sect. 8.1.2 we describe a list of changes aimed to evolve the conceptual schema and how these changes affect the set of issues the schema contains. Using this example, we show the problems that arise when using non-incremental methods and how we may overcome them using an incremental method instead.

### 8.1.1 A Motivating Example

In Sect. 4.1.1 we presented a simple conceptual schema with several quality issues. For ease of reading, we reproduce the schema in Fig. 8.1. The schema contains several problem and checking issues, including:

a) the cardinality constraint [1..0] of the participant *person* in association *Owns* is syntactically incorrect,

b) the entity type *motorcycle* does not start with a capital letter, despite several naming guidelines recommend that entity types start with a capital letter (e.g. [10]),

c) the specialization *Car IsA LandVehicle* is redundant.

These issues correspond to the following issue types:

[1]Note this process evaluates each issue type for each instance in the scope *every time* it is executed.

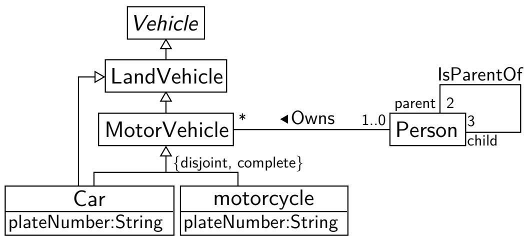


**Figure 8.1.** A conceptual schema with several quality issues.

- $I_a$: Cardinality constraints are syntactically incorrect.
- $I_b$: The name of a class is not properly capitalized.
- $I_c$: A generalization is redundant.

which can be formalized as the following OCL expressions[2]:

```
context Property inv Ia:
   self.lower > self.upper

context Class inv Ib:
   self.name.at(1) <> self.name.at(1).toUpperCase()

context Generalization inv Ic:
   Generalization.allInstances()->select( g |
      g<>self and
      g.specific = self.specific and
      ( g.general.allParents()->includes(self.general)
        or
        g.general = self.general )
   )->notEmpty()
```

that return *True* when there is an issue of the associated type and *False* otherwise.

[2]For the sake of simplicity and to ease the understanding of incremental methods, we define the issue using the OCL expression that corresponds to the issue condition, and we ignore any other aspects of the issue formalization we presented in Chap. 5 such as, for example, the applicability condition, any precedents it may have, or its acceptability.

### 8.1.2 Computing Issue Instances as the Schema Evolves

During the development of a conceptual schema, the set of issues it contains change as the schema evolves. Consider, for example, the changes described in Tab. 8.1. Our goal is to detect the issues the schema contains after each set of changes.

**Table 8.1.** Timeline of changes and issues in Fig. 8.1.

| Time | Change |
|---|---|
| $t_1$ | Renaming *motorcycle*. New name: *Motorcycle* |
| $t_4$ | Changing multiplicities of *person* to [0..1]<br>Creating new class *c*<br>Naming the new class *c*. New name: *Man* |
| $t_5$ | Removing the generalization *Car IsA LandVehicle* |

| At $t_0$ | After $t_1$ | After $t_4$ | After $t_5$ |
|---|---|---|---|
| $I_b$(*motorcycle*) | ~~$I_b$(*motorcycle*)~~ | | |
| $I_a$(*person*[1..0]) | $I_a$(*person*[1..0]) | ~~$I_a$(*person*[0..1])~~ | |
| $I_c$(*Car IsA LandVehicle*) | $I_c$(*Car IsA LandVehicle*) | $I_c$(*Car IsA LandVehicle*) | ~~$I_c$(*Car IsA LandVehicle*)~~ |

In Sect. 5.4, we proposed an algorithm to compute the issues that are present in a conceptual schema. The algorithm is, in principle, intended to be executed as a batch process—given a conceptual schema and a set of issue types, the algorithm evaluates all issue types for each instance in the scope and, thus, it computes all the issue instances the schema contains. In our running example, the algorithm would have to compute the value of the following expressions in order to determine the issues contained in Fig. 8.1:

- $I_a$(*motorVehicle*[0..*]) = *False*
- **$I_a$(*person*[1..0]) = *True***
- $I_a$(*parent*[2]) = *False*
- $I_a$(*child*[3]) = *False*
- $I_a$(*Car::plateNumber*[1]) = *False*
- $I_a$(*motorcycle::plateNumber*[1]) = *False*

- $I_b$(*Vehicle*) = *False*
- $I_b$(*LandVehicle*) = *False*
- $I_b$(*MotorVehicle*) = *False*
- $I_b$(*Car*) = *False*
- **$I_b$(*motorcycle*) = *True***
- $I_b$(*Person*) = *False*

- $I_c$(*Car IsA MotorVehicle*) = *False*
- $I_c$(*motorcycle IsA MotorVehicle*) = *False*
- $I_c$(*MotorVehicle IsA LandVehicle*) = *False*
- $I_c$(*LandVehicle IsA Vehicle*) = *False*
- **$I_c$(*Car IsA LandVehicle*) = *True***

A naive approach to get "continuous" and up-to-date feedback consists in executing the algorithm presented in Sect. 5.4 periodically (like, for example, after each $t_i$). Each time the algorithm is run, it evaluates each issue type $I_x$ for each instance of its scope. Clearly, this periodic approach results in completely unnecessary evaluations—on the one hand, if there were no changes at all (i.e. at $t_2$ and $t_3$), the issues also remain unchanged; on the other hand, if, for example, *LandVehicle*'s name is never changed, it will never raise an issue of type $I_b$.

At a specification level, the number of unnecessary executions of the algorithm is not a problem. However, when this solution is implemented into a modeling environment, it might be a problem—the computation of issue instances takes some time. For small conceptual schemas and not many issue types, this time may be short enough to provide "instant" feedback, but as the schema and the number of issue types get bigger, a full continuous evaluation becomes extremely inefficient quickly.

In order to overcome this efficiency problem, the best solution is to perform an *Incremental Evaluation*. Incremental methods aim to perform the re-evaluation of an expression if, and only if, its result *may have changed*. The only way a result may have changed is because the schema itself has changed. Thus, if no changes occur at, for example, $t_2$ or $t_3$, there is no need to re-compute the issue instances at $t_2$ and $t_3$, because they are exactly the same ones that existed at $t_1$. Moreover, in general not all changes may affect the truth value of an expression, but only a few. Therefore, renaming, for instance, the class *motorcycle* at $t_1$ has nothing to do with having the redundant generalization *Car IsA LandVehicle*.

The goal of an *Incremental Method* is to re-evaluate an expression using a certain input if, and only if, its result *may have changed* with respect to the previous one. Incremental methods need to:

- *determine which changes* over the conceptual schema *may* modify the truth value of an expression, and
- *monitor the changes* that occur in the schema and see if any of them were included in the previous set, since it would require the affected expressions to be re-evaluated.

### Examples of Incremental Methods

An incremental method has to determine which changes over the conceptual schema may modify the truth value of an expression. In general, the set of changes that actually modify an expression is much, much smaller than the set of all possible changes over the schema. Therefore, the more accurate the method is when it comes to determine this set of changes, the better. In the following, we briefly outline three examples of incremental methods.

Consider the conceptual schema of our running example and the timeline of changes presented in Tab. 8.1. If we focus on the issue type $I_b$ and its evaluation over the class *motorcycle*—i.e. $I_b(motorcycle)$—, it is clear that re-evaluating $I_b(motorcycle)$ periodically as aforementioned is unnecessary as long as the schema remains unchanged ($t_2$ and $t_3$).

The incremental method $m_1$ re-evaluates $I_b(motorcycle)$ if, and only if, the schema has changed. Thus, for example, the expression is not re-evaluated at $t_2$ or $t_3$, but it is at $t_1$ or $t_4$.

We have said that the set of changes that may actually modify the result of an expression is much smaller than the set of all possible changes. However, $m_1$ follows a naive approach and re-evaluates an expression whenever a change occurs. If we focus on the issue type $I_a$, we will see that it checks that the *name* of a *Property* is properly capitalized. Therefore, it is clear that only those changes that modify the *name* of a *Property* may require $I_a$ to be re-evaluated:

The incremental method $m_2$ re-evaluates $I_b(motorcycle)$ if, and only if, a *Class* has been renamed. Thus, for example, the expression is not re-evaluated at $t_2$ or $t_5$, but it is at $t_1$ or $t_4$.

If we take a closer look to $m_2$, we will realize it still performs unnecessary re-evaluations. The only “class-naming event” that may actually change the result of $I_b(motorcycle)$ is, precisely, renaming the class *motorcycle* itself:

The incremental method $m_3$ re-evaluates $I_b(motorcycle)$ if, and only if, the *Class motorcycle* has been renamed. Thus, for example, the expression is not re-evaluated at $t_2$, $t_4$ or $t_5$, but it is at $t_1$.

In the next section, we review some incremental methods for the efficient evaluation of OCL expressions. The reviewed works precisely describe how methods like $m_2$ or $m_3$ can actually determine the set of changes that may modify the truth value of an expression.

## 8.2 Related Work on Incremental Evaluation

Quality issues may be seen as some sort of *integrity constraints* at a metamodel level. There are several proposals in the literature devoted to the problem of efficient integrity checking. Many of them deal with integrity checking in the database field [24, 25, 122, 126, 127]. However, in what follows we focus on the most representative proposals on incremental integrity checking in the UML/OCL language.

### 8.2.1 Efficient Evaluation of OCL Integrity Constraints

An integrity constraint defines a condition that must be satisfied in each state of the information base. In principle, an information system has to guarantee that the state of the information base is always consistent with respect to the integrity constraints of its conceptual schema. Therefore, the efficient evaluation of any integrity constraint becomes crucial for ensuring this consistency. Incremental methods exploit available information about the structural events applied during modifications of the information base in order to avoid re-computing the truth values of unaffected constraints. A structural event is defined as “a change in the population of an entity or a relationship type”, including inserting a new entity type, changing the value of an attribute, or deleting a relationship type, among others.

In [22, 23], Cabot and Teniente propose an incremental checking of OCL integrity constraints for UML conceptual schemas[3]. Their method is fully automatic and works at the conceptual level (which means it is technology-independent). The main goal of Cabot and Teniente's method approach is to consider as few entities of the information base as possible during the evaluation of an integrity constraint.

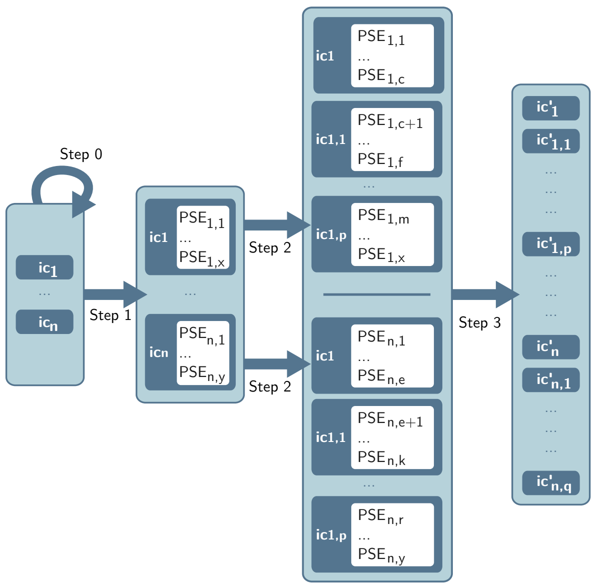

Step 0
$ic_1$
...
$ic_n$
Step 1
ic1
$PSE_{1,1}$ ... $PSE_{1,x}$
icn
$PSE_{n,1}$ ... $PSE_{n,y}$
Step 2
Step 2
ic1
$PSE_{1,1}$ ... $PSE_{1,c}$
ic1,1
$PSE_{1,c+1}$ ... $PSE_{1,f}$
ic1,p
$PSE_{1,m}$ ... $PSE_{1,x}$
ic1
$PSE_{n,1}$ ... $PSE_{n,e}$
ic1,1
$PSE_{n,e+1}$ ... $PSE_{n,k}$
ic1,p
$PSE_{n,r}$ ... $PSE_{n,y}$
Step 3
$ic'_1$
$ic'_{1,1}$
$ic'_{1,p}$
$ic'_n$
$ic'_{n,1}$
$ic'_{n,q}$

**Figure 8.2.** General schema of Cabot and Teniente's method [22, 23].

Figure 8.2 depicts the general schema of the method presented in [22, 23], which consists of three main steps (steps 1-3) plus a preliminary step (step 0). These steps are applied to the original OCL expressions so that they result in a different, alternate representation that allows an efficient evaluation:

- **Step 0: simplification of OCL expressions** OCL expressions are simplified by reducing OCL expressiveness (only a subset of OCL operators can be

[3]Despite this work addresses the efficient evaluation of integrity constraints over an information system, the presented approach can be clearly applied to the metamodel level, where the "conceptual schema" is the "metaschema", the "integrity constraints" are "metamodel constraints", and the "information base" is "the conceptual schema under development".

thus used).

- **Step 1: determining potentially-violating structural events (PSE)** This step associates to each constraint of the conceptual schema a set of PSE, which are drawn from the syntactical definition of the constraint. These include updating the value of an attribute, inserting a new relationship in a specific relationship type, and so on.

- **Step 2: obtaining an appropriate syntactic representation for each constraint** For each integrity constraint $ic_i$ and each event $ev \in$ PSE, this step determines an appropriate alternative syntactic representation $ic_{i,j}$ with respect to $ev$.

- **Step 3: Redefining the constraints to evaluate over the relevant instances** Finally, each constraint resulting from step 2 is redefined to be evaluated only over the instances of its context type affected by events instance of the event types included in its particular subset of PSEs.

### 8.2.2 Inconsistency Management

For a conceptual schema to be syntactically correct, all metamodel constraints have to be satisfied. The violation of a metamodel constraint is known as an *inconsistency*. Classically, inconsistencies are forbidden—i.e. the modeler cannot perform any change such that her conceptual schema becomes inconsistent. However, some authors consider inconsistencies inevitable and, therefore, they encourage working with them, as long as they get fixed sometime in the (near) future [44, 49, 115]. The rationale is that developing a conceptual schema is a creative process and, as such, inconsistencies inevitable arise whilst exploring different alternatives, or simply because of different stakeholders having different (inconsistent) views of the system under development. This view is completely compatible with our definition of quality issues: during the development of a conceptual schema, we allow the existence of issues and we expect the conceptual modeler to fix them before the schema is finished.

When inconsistencies are allowed, inconsistency handling becomes a key piece in the development of a schema. Inconsistency handling involves, on the one hand, *detecting* the insertion of an inconsistency into the model and, on the other hand, deciding a *response* on how to deal with this newly found inconsistency: should it be rejected or should it be accepted?

Inconsistencies, as a concrete instance of quality issues, can be detected (1) *when requested* by the conceptual modeler, (2) in a *periodically* fashion, where a complete check of the schema is performed after a certain amount of time went by; or (3) *continuously*, that is, every time that the conceptual modeler makes a change to the schema. Since conceptual modelers are better off knowing about the existence of inconsistencies to avoid follow-on errors and unnecessary rework [44], the sooner the inconsistencies are detected and reported (2 and 3), the better. There are several works in the literature that deal with the efficient and incremental detection of inconsistencies as they arise.

In [15, 112], the authors propose an incremental consistency checker based on the idea of representing models as sequences of primitive construction operations. The four elementary operations they define are: *create*, *delete*, *setProperty*, and *setReference*. In order to detect an inconsistency, they define *Inconsistency Detection Rules*. Any inconsistency rule is a logic formula over the sequence of model construction operations: if a set of operations is triggered in a specific order, it can be assured that an inconsistency has been introduced into the model.

In [43, 44], Egyed proposes an instant consistency checking for the UML. He evaluates up to 24 consistency rules for UML class, sequence, and statechart diagrams. His work assumes that consistency rules are stateless and deterministic—i.e. if any rule is evaluated on the same portion of the model twice, then it will perform the same actions and return the same truth value.

Consistency rules are typically defined (and evaluated) from a *context element*, and are defined as follows:

$$\textbf{\textit{ConsistencyRule}} = \langle ContextElement, Condition \rangle \rightarrow Boolean$$

where *ContextElement* ∈ *MetaModelElements*.

Clearly, a consistency rule is affected by any change over the instances of its context element. However, it may also be affected by many other model elements that are not explicitly identified (i.e. the elements referenced in the *Condition* expression). Therefore, the problem is to determine efficiently the model elements that may affect the truth value of a consistency rule and (re)evaluate it if, and only if, one of these elements is modified.

To this end, Egyed defines two main concepts: (1) a *ConsistencyRule – Instance* ($CRI$) that represents the evaluation of a consistency rule over an instance of its context, formally defined as the pair $\langle ConsistencyRule, Model\text{-}Element\rangle$, and (2) the *Scope* of a $CRI$, which includes the set of elements accessed during the evaluation of the $CRI$. Specifically, this *Scope* is the set of $\langle ModelElement, Field\rangle$ pairs that have been accessed during the evaluation of a $CRI$.

Every $CRI$ starts its evaluation at a different instance $e$ of the context element and references/accesses different model elements. As a result, the truth value of the *Condition* depends on the specific instance $e$ that it has been evaluated over. Obviously, each $CRI$ is affected differently by model changes. The same type of model change (for example, "changing the name of a *Class c*"), may affect one $CRI_{x1} = \langle cr_x, e_1\rangle$ and not another $CRI_{x2} = \langle cr_x, e_2\rangle$, just because $CRI_{x1}$ depends on the name of $c$, whilst $CRI_{x2}$ does not. A change in the schema is defined as $\langle ModelElement, Field\rangle : oldValue \rightarrow newValue$. A $CRI$ has to be re-evaluated if, and only if, the scope of $CRI$ contains the same pair included in the change.

At this point, the problem of efficiently evaluating inconsistencies becomes creating and destroying $CRIs$ and determining its scope as it changes over the time. On the one hand, Egyed's approach simply creates CRIs when context elements are created and destroys them once their context elements are destroyed. On the other hand, the scope can be easily computed by using a model profiler: when a $CRI$ is (re)evaluated, the profiler monitors all the elements that are accessed in order to evaluate the truth value of the consistency rule's condition (more details on this approach and its validity are described in [44]).

According to Egyed, the approach he presents is outstanding, as the checking of the consistency rules can be considered "instant" regardless of the size of the conceptual schema. Apparently, the only drawback of their method is the additional memory it requires. In order to be time-efficient during the evaluation of consistency rules, their approach requires more memory to store information about $CRIs$ and their scopes. However, their empirical studies demonstrate that, given a concrete number of consistency rules, the required memory increases linearly with respect to the size of the conceptual schema, and that the size of the scopes is bounded.

## 8.3 Incremental Evaluation of Quality Issues

In this section, we first formalize the main concepts involved in an incremental evaluation of expressions. Next, we describe how to integrate one incremental approach to our formalization of issue types. Finally, we compare the results of evaluating quality issues using (a) the algorithm we described in Sect. 5.4 and (b) an incremental approach.

### 8.3.1 Formalization of Incremental Evaluations

Let $S$ be a schema that consists of $n$ schema elements $e_1, \ldots, e_n$, which are instances of the corresponding schema metatypes. Let $\psi_x(T)$ be an expression that has to be evaluated for each instance $e_i \in T$. Let $r_{\langle x,i\rangle}$ be the result of evaluating $\psi_x(T)$ using $e_i$ as its input. We define the "evaluation of an expression using a certain input" $\mathscr{E}_{\langle x,i\rangle}$ as follows:

$$\mathscr{E}_{\langle x,i\rangle} = \langle \psi_x(T), e_i \rangle \rightarrow r_{\langle x,i\rangle}$$

We have seen that $r_{\langle x,i\rangle}$ remains unchanged as long as the schema remains unchanged too. Moreover, we have also seen that not all changes that may be applied to a schema affect it, but only a subset of them. Let $\mathcal{C}$ be the set of changes that may be applied to the conceptual schema, such as creating classes, deleting associations, or setting the name of properties, to mention a few. We define $\mathcal{C}(\mathscr{E}_{\langle x,i\rangle})$ as the set of changes over the schema that may change $r_{\langle x,i\rangle}$. It is expected that the number of changes included in $\mathcal{C}(\mathscr{E}_{\langle x,i\rangle})$ will be smaller than the set of all possible changes—formally, $\mathcal{C}(\mathscr{E}_{\langle x,i\rangle}) \subseteq \mathcal{C}$.

Consider the running example we presented in the previous section as defined at $t_0$. The set of changes we may apply to the schema are[4]:

$$\begin{aligned}\mathcal{C} = \ &\{\mathit{SetAttr(LandVehicle, name), SetAttr(motorcycle, name)}, \ldots \\ &\quad \mathit{CreateProperty(), CreateClass()}, \ldots \\ &\quad \mathit{Link(LandVehicle, attr{:}Property)}, \ldots\}\end{aligned}$$

[4]These changes are only used for illustrative purposes. Thus, for the sake of simplicity we only include a subset of all possible changes.

which correspond to changing the names of *LandVehicle* or *motorcycle*, creating new properties or classes in the schema, or adding a new attribute to the class *LandVehicle*. Next, $\psi_c(Class)$ for the input *motorcycle* is defined as:

$$\mathscr{E}_{\langle c,motorcycle\rangle} = \langle \psi_c(Class), motorcycle\rangle \rightarrow True$$

And, finally, the set of changes[5] that may actually change the result of $\mathscr{E}_{\langle c,motorcycle\rangle}$ is:

$$\mathcal{C}(\mathscr{E}_{\langle c,motorcycle\rangle}) = \{\mathit{SetAttr(motorcycle,\ name)}\}$$

As we have said, the goal of an incremental method $m$ is to determine a subset of changes of $\mathcal{C}$ that may affect the result of evaluating a certain expression using a certain input. Formally, we define $\mathcal{C}_m(\mathscr{E}_{\langle x,i\rangle})$ as the set of changes that, according to the incremental method $m$, may change the result of $\mathscr{E}_{\langle x,i\rangle}$. Clearly, this set has to include all the changes included in $\mathcal{C}(\mathscr{E}_{\langle x,i\rangle})$—formally, $\mathcal{C}(\mathscr{E}_{\langle x,i\rangle}) \subseteq \mathcal{C}_m(\mathscr{E}_{\langle x,i\rangle})$.

In Sect. 8.1.2, we outlined three examples of incremental methods. In the following, we summarize each method using the formalization of incremental methods we have just described:

- $m_1$ re-evaluates any expression whenever a change occurs, no matter which one:

$$\mathcal{C}_{m_1}(\mathscr{E}_{\langle c,motorcycle\rangle}) = \mathcal{C}$$

- $m_2$ re-evaluates an expression if the triggered change may affect, at a "type level", the result. Thus, for example, if the expression to re-evaluate is $\mathscr{E}_{\langle c,motorcycle\rangle}$, any change that modifies the name of a *Class* (a type) would force its re-evaluation. Thus:

$$\begin{aligned}\mathcal{C}_{m_2}(\mathscr{E}_{\langle c,motorcycle\rangle}) = \ & \{\mathit{SetAttr(LandVehicle,\ name)},\\ & \mathit{SetAttr(motorcycle,\ name)},\\ & \ldots\}\end{aligned}$$

[5] Obviously, deleting the instance *motorcycle* affects $\mathcal{C}(\mathscr{E}_{\langle c,motorcycle\rangle})$.

- $m_3$ re-evaluates an expression if the triggered change may affect, at an "instance level", the result. Thus, for example, if the expression to re-evaluate is $\mathscr{E}_{\langle c,motorcycle\rangle}$, only a change modifying the name of *motorcycle* (an instance) would force its re-evaluation. Hence:

$$\mathcal{C}_{m_3}(\mathscr{E}_{\langle c,motorcycle\rangle}) = \{\mathit{SetAttr(motorcycle,\ name)}\}$$

### 8.3.2 Integrating an Incremental Approach to our Method

In Chap. 5 we have presented a formalization for defining conceptual schema quality issue types. Section 5.4 presented an algorithm for computing the issue instances of a certain issue type $I_x$. Issue instances are basically computed using the following information:

- the applicability condition $\phi_x$, which computes the *Potential* set of elements that may raise an issue of type $I_x$ and
- the issue condition $\rho_x$, which determines whether a tuple in the *Potential* set actually raises an issue of type $I_x$.

Clearly, these two expressions can be computed using an incremental method. In this thesis, we have decided to use Egyed's method $m_{\text{Egyed}}$ [44], even though any method reviewed in Sect. 8.2 might serve. We chose this method instead of the others because, on the one hand, it is one of the quickest methods available, and, on the other hand, it is very easy to integrate to our prototype tool—i.e. it does not require to change the definition of issue conditions to another language, and it works for issues defined in OCL and Java, once we integrated the profiler in the CMA.

When evaluating an expression $\psi_x(T)$ using an instance $e_i$ (formally, $\mathscr{E}_{\langle x,i\rangle} = \langle \psi_x(T), e_i\rangle$) with Egyed's method, not only the regular result ($r_{\langle x,i\rangle}$) is returned, but also the list $\mathcal{L}$ of elements in the conceptual schema that were accessed during the evaluation $\mathscr{E}_{\langle x,i\rangle}$:

$$\mathscr{E}_{\langle x,i\rangle} \rightarrow \langle r_{\langle x,i\rangle}, \mathcal{L}\rangle$$

If we focus on the applicability and issue conditions of an issue type $I_x$, we have:

- the *Potential* set and the elements that were accessed for building it. Formally:

  $\mathcal{EP}_x \rightarrow \langle Pot(I_x), \mathcal{L}_{\phi_x} \rangle$

- For each tuple $\langle e_1, \ldots, e_m \rangle$ in $Pot(I_x)$, whether it raises an issue or not, and the element that were accessed for determining it. Formally:

  $\mathcal{ER}_{x,\langle e_1,\ldots,e_m \rangle} = \langle \rho_x(T_1, \ldots, T_m), \langle e_1, \ldots, e_m \rangle \rangle \rightarrow \langle \langle True, False \rangle, \mathcal{L}_{I_x(e_1,\ldots,e_m)} \rangle$

Consider, for example, the issue type $I_a$ = "Cardinality constraints [of a member end] are syntactically incorrect". The scope of $I_a$ is a single *Property*, and the applicability and issue conditions may be formally defined as:

```
ϕa(Property) : self.association->notEmpty()
ρa(Property) : self.lower > self.upper
```

In order to compute the potential set $Pot(I_a)$ of tuples that may actually raise an issue of type $I_a$, $\phi_a$ has to be evaluated over all *Properties* of the schema. Clearly, we can evaluate $\phi_a$ incrementally using Egyed's method:

$$\begin{aligned} Pot(I_a) &= \{\textit{motorVehicle, person, parent, child}\} \\ \mathcal{L}_{\phi_a} &= \{\textit{motorVehicle, motorVehicle.assoc=Owns,} \\ &\quad \textit{person, person.assoc=Owns, } \ldots \\ &\quad \textit{Car::plateNumber, Car::plateNumber.assoc=}\emptyset\textit{,} \\ &\quad \ldots\} \end{aligned}$$

where $Pot(I_a)$ is the regular result we expect from the evaluation of $\phi_a$ and $\mathcal{L}_{\phi_a}$ includes all the elements of the conceptual schema that were accessed during the computation of $Pot(I_a)$. If we take a closer look to $\mathcal{L}_{\phi_a}$, we see that, on the one hand, it contains all properties of the schema (note that $\phi_a$ is evaluated for each *Property* in the schema and that $\phi_a$ starts with an access to *self*, i.e. to a *Property*) and, on the other hand, any association (if any) that is related to a *Property* (because of the *self.association* navigation).

Similarly, when evaluating $\rho_a$ using an element in $Pot(I_a)$ as its input (e.g. *person*), the result consists in:

$$\begin{aligned} \rho_a(person) &= True \\ \mathcal{L}_{I_a(person)} &= \{\textit{person, person.upper, person.lower}\} \end{aligned}$$

where $\rho_a(person)$ indicates whether *person* actually raises an issue of type $I_a$ and $\mathcal{L}_{I_a(person)}$ includes all the elements of the conceptual schema that were accessed during the evaluation of $\rho_a$ over the *Property person*. In this case, the only elements that were accessed for evaluating the expression are *person* (because of the *self* clause) and the *upper* and *lower* attributes of the *Property*.

In the following we present an algorithm for incremental evaluation of quality issues, and we describe how the new information provided by Egyed's method is used for preventing unnecessary re-computations.

## Adapting the Algorithm for Computing Issue Instances

In Sect. 5.4 we presented an algorithm for computing the issues that are present in a conceptual schema. This algorithm can be run as a background process in order to provide "continuous" feedback, at the expense of performing several unnecessary re-evaluations. In this section we describe a new algorithm for computing issue instances using an incremental approach[6]. The algorithm is presented in Alg. 2.

The three main differences with respect to Alg. 1 are:

1. The algorithm requires new parameters:
   - $\mathcal{C}$, the set of changes that occurred in the schema since the last execution,
   - $\Pi_{\text{prev}}$, the set of *Potential* sets that were computed in the previous execution (along with the elements that were accessed for each computation), and
   - $\xi_{\text{prev}}$, the set of $\mathcal{ER}_{x,\langle e_1,\ldots,e_m\rangle}$, i.e. the results of evaluating $\rho_x$ for each tuple in $Pot(I_x)$ (along with the elements that were accessed for this evaluation).
2. Instead of computing the potential set of an issue $I_c$ each time (see line 7 in Alg. 1), Algorithm 2 checks if the previous evaluation of $Pot(I_c)$ is still valid (lines 7 to 9).

[6]For the ease of understanding, we do not consider precedents.

Consider, for instance, the example presented in Sect. 8.3.2. When $Pot(I_a)$ was computed, the list of accessed elements included, among many others, the navigation *motorVehicle.assoc* = *Owns*. If there has been a change such that *motorVehicle.assoc* is no longer *Owns*, but a different association or none association at all, then we clearly have to re-compute $Pot(I_a)$, because it may be the case that *motorVehicle* is no longer a member end of an association and, therefore, it cannot be in $Pot(I_a)$.

3. Similarly, instead of checking whether a tuple $\langle e_1, \ldots, e_m\rangle$ in $Pot(I_c)$ raises an issue (see line 10 in Alg. 1), Algorithm 2 checks if the previous evaluation of $\rho_c(e_1, \ldots, e_m)$ is still valid (lines 11 to 13).

   Consider, again, the example presented in 8.3.2. If the set of changes that occurred since the last execution of the algorithm include, for instance, changing the value of the *person*'s *upper* attribute (say, from 0 to 3), then we have to re-compute whether *person* still raises an issue of type $I_a$ or not (and, in this case, it does not, because, finally, the upper value is greater or equal than the lower).

Please note that, in order to simplify the algorithm, we defined the following auxiliary functions:

- *computePotential*$(S,\ \phi_x) = \langle Pot(I_x, \mathcal{L}_{\phi_a})\rangle$, which computes the potential set of an issue type $I_x$ using the elements of the schema $S$ and the applicability condition $\phi_x$, and returns this potential set, as well as the elements that were accessed for this computation.

- *computeInstance*$(S,\ \rho_x,\ e_1, \ldots, e_m) = \langle\langle True, False\rangle, \mathcal{L}_{I_x(e_1,\ldots,e_m)}\rangle$, which determines whether the tuple $e_1, \ldots, e_m$ from $Pot(I_x)$ raises an issue of type $I_x$ or not, returning *True* or *False*, along with the elements that were accessed for this computation.

- *affects*$(\mathcal{C},\ \mathcal{E}_x) = \langle True, False\rangle$, which returns whether a change in $\mathcal{C}$ may have modified the result of $\mathcal{E}_x$. For example, if $\mathcal{C}$ includes a change such that "class *c* has been renamed", and the evaluation of a certain expression using a certain input $\mathcal{E}_x$ accessed *c.name*, then this function returns *True*.

**Algorithm 2** Computing Issue Instances Incrementally (without Precedents)

```
Input   – $S$: a set of $n$ schema elements $e_1,\dots,e_n$,
        – $\mathcal{T}$: the set of issue types $I_1,\dots,I_p$,
        – $\Psi_{\text{prev}}$: the set of pairs $\langle I_x(e_1,\dots,e_m),\varepsilon\rangle$ computed in the previous execution,
          where $I_x(e_1,\dots,e_m)$ was a raised issue and $\varepsilon$ was its state.
        – $\Pi_{\text{prev}}$: the set of $\mathcal{EP}_x$,
        – $\xi_{\text{prev}}$: the set of $\mathcal{ER}_{x,\langle e_1,\dots,e_m\rangle}$, and
        – $\mathcal{C}$: the set of changes that occurred in the previous evaluation.
Output  – $\Psi_{\text{new}}$: the set of pairs $\langle I_x(e_1,\dots,e_m),\varepsilon\rangle$ computed in this execution, where
          $I_x(e_1,\dots,e_m)$ is a raised issue and $\varepsilon$ is its state,
        – $\Pi_{\text{new}}$: the set of $\mathcal{EP}_x$, and
        – $\xi_{\text{new}}$: the set of $\mathcal{ER}_{x,\langle e_1,\dots,e_m\rangle}$.
1:  procedure $updateIssues(S,\mathcal{T},\Psi_{\text{prev}},\Pi_{\text{prev}},\xi_{\text{prev}}) : \langle\Psi_{\text{new}},\Pi_{\text{new}},\xi_{\text{new}}\rangle$
2:      $\Psi_{\text{new}},\Pi_{\text{new}}\text{and}\xi_{\text{new}} \leftarrow \emptyset$
3:      $\mathcal{T}_{\text{pending}} \leftarrow \mathcal{T}$
4:      while $\mathcal{T}_{\text{pending}} \neq \emptyset$ do
5:          $I_c \leftarrow I_x \mid (I_x \in \mathcal{T}_{\text{pending}} \wedge (\nexists I_y \mid I_y \in \mathcal{T}_{\text{pending}} \wedge I_y \in \mathcal{P}_x))$
6:          $CandidateIssues \leftarrow \emptyset$
7:          $\mathcal{E}_c \leftarrow \mathcal{EP}_c \mid \mathcal{EP}_c \in \Pi_{\text{prev}}$
8:          if $\mathcal{E}_c = \emptyset \ \vee\ affects(\mathcal{C},\mathcal{E}_c)$ then $\mathcal{E}_c \leftarrow computePotential(S,\phi_c)$
9:          $\Pi_{\text{new}} \leftarrow \Pi_{\text{new}} \cup \mathcal{E}_c$
10:         for all $\langle e_1,\dots,e_m\rangle \in getResultFrom(\mathcal{E}_c)$ do
11:             $\mathcal{E}_i \leftarrow \mathcal{ER}_{c,\langle e_1,\dots,e_m\rangle} \mid \mathcal{ER}_{c,\langle e_1,\dots,e_m\rangle} \in \xi_{\text{prev}}$
12:             if $\mathcal{E}_i = \emptyset \ \vee\ affects(\mathcal{C},\mathcal{E}_i)$ then $\mathcal{E}_i \leftarrow computeInstance(S,\rho_c,\langle e_1,\dots,e_m\rangle)$
13:             $\xi_{\text{new}} \leftarrow \xi_{\text{new}} \cup \mathcal{E}_i$
14:             if $getResultFrom(\mathcal{E}_i)$ then
15:                 $CandidateIssues \leftarrow CandidateIssues \cup \langle e_1,\dots,e_m\rangle$
16:             end if
17:         end for
18:         $IssuesToKeep \leftarrow \{\langle e_1,\dots,e_m\rangle \mid \langle I_c(e_1,\dots,e_m),\varepsilon\rangle \in \Psi_{\text{prev}} \wedge$
                                    $\langle e_1,\dots,e_m\rangle \in CandidateIssues\}$
19:         $IssuesToCreate \leftarrow CandidateIssues - IssuesToKeep$
20:         $\Psi_{\text{new}} \leftarrow \Psi_{\text{new}} \cup$
                    $\{i = \langle I_c(e_1,\dots,e_m),\varepsilon\rangle \mid i \in \Psi_{\text{prev}} \wedge \langle e_1,\dots,e_m\rangle \in IssuesToKeep\} \cup$
                    $\{\langle I_c(e_1,\dots,e_m),Pending\rangle \mid \langle e_1,\dots,e_m\rangle \in IssuesToCreate\}$
21:         $\mathcal{T}_{\text{pending}} \leftarrow \mathcal{T}_{\text{pending}} - \{I_c\}$
22:     end while
23:     return $\langle\Psi_{\text{new}},\Pi_{\text{new}},\xi_{\text{new}}\rangle$
24: end procedure
```

## How Precedents Affect Incremental Evaluation of Issues?

In Sect. 5.3 we have seen that there are cases in which the issues of a given issue type $I_x$ should only be considered if there are no other unsolved issues of some specific types. For example, $I_n =$ "The name of a class is not properly capitalized" should only be considered for a class $c$ if there is not an issue $I_p =$ "A class has no name" for that same class $c$. This relationship between issues has been formalized by means of *issue precedents*[7].

Algorithm 1 (presented in Sect. 5.4) evaluates the instances of an issue type $I_x$ after it has evaluated all its precedents (line 5). Then, for each tuple in $Pot(I_x)$, it checks whether the precedents are satisfied or not. If they are, the algorithm evaluates $\rho(e_1, \ldots, e_m)$ in order to determine whether $\langle e_1, \ldots, e_m \rangle$ raises an issue of type $I_x$ or not. If, on the other hand, the precedents are not satisfied, $\rho(e_1, \ldots, e_m)$ is not evaluated.

When evaluating the issue instances of an issue type $I_x$ incrementally, we may already have the result of evaluating $\rho(e_1, \ldots, e_m)$. As we have seen throughout this chapter, the incremental method has to determine whether this previous result is still valid or has to be re-computed.

Algorithm 2 integrated an incremental approach to Alg. 1, without taking into account precedents. When using precedents and evaluating a certain issue type $I_x$, there are only three possible scenarios for the algorithm to deal with:

1. $I_x$ does not have any precedents, which means that Algorithm 2 will operate properly,
2. $I_x$ does have precedents, but they are all satisfied for all tuples in $Pot(I_x)$, which means that, again, Algorithm 2 will operate properly, and
3. $I_x$ has precedents and one tuple $\langle e_1, \ldots, e_m \rangle$ in $Pot(I_x)$ has a precedent that is not satisfied. When this third scenario occurs, it may be the case that:
    a. $\mathcal{ER}_{c,\langle e_1,\ldots,e_m \rangle} = \emptyset$, which means that $\rho(e_1, \ldots, e_m)$ had not been evaluated in a previous execution of the algorithm. Since the precedents are not satisfied, we do not need to evaluate it now either.
    b. we do have a previous evaluation $\mathcal{ER}_{c,\langle e_1,\ldots,e_m \rangle}$, but the set of changes that occurred in the schema since the previous execution of the algorithm would require $\mathcal{ER}_{c,\langle e_1,\ldots,e_m \rangle}$ to be re-evaluated. Again, since the precedents are not satisfied, we are not performing this evaluation now.

[7]In our example, $I_p$ is an issue precedent of $I_x$.

c. we do have a previous evaluation $\mathcal{ER}_{c,\langle e_1,\ldots,e_m\rangle}$ and the set of changes that occurred in the schema since the previous execution of the algorithm do not affect its result. In this case, we do not include a raised issue in $\Psi_{\text{new}}$ (because that issue should not be considered by the modeler), but we do save $\mathcal{ER}_{c,\langle e_1,\ldots,e_m\rangle}$ in $\Pi_{\text{new}}$ for future reference.

### 8.3.3 Integrating the Incremental Evaluation of Quality Issues in our CMA

In this section we describe the changes that we applied to our CMA—which is based on the Eclipse Platform—so that it can incrementally evaluate issue types defined in OCL. Given that we have full access to the source code of both our CMA and the Eclipse Platform, the integration of an incremental method to our prototype tool is easy[8].

When using an incremental approach, the algorithm for computing issue instances has to be executed each time the conceptual schema is modified—i.e. whenever a set of *structural events* is triggered. As we have seen throughout this chapter, when the incremental algorithm is executed, it only re-evaluates the expressions that might have been affected by the issued changes. In essence, this depends on (a) the *structural events* that were issued and (b) the elements of the conceptual schema that were accessed during a previous evaluation of the expression. As a consequence, our CMA has to simply modify two components of the Eclipse Platform:

**The UML Editor.** When the conceptual modeler modifies her conceptual schema, the UML editor has to generate the list of *structural events* that correspond to the changes applied to the model.

In principle, Eclipse provides two different editors for creating, viewing, and modifying UML conceptual schemas—(a) the *default UML Editor*, which uses a tree-based view (as depicted in Fig. 7.2), and (b) a *graphical UML editor*, which is based on the Graphical Modeling Framework (GMF). GMF editors use an Eclipse component called *Transactional Editing Domains*.

[8]Note that it is also possible to extend software components without changing them and assuming there is no access to the source code [45].

These components generate events whenever the model they work with is used by the modeler. Thus, for example, the UML GMF editor triggers one or more change events whenever the modeler moves a class around the diagram, adds a new association, or deletes a class.

In order to use an incremental approach, our CMA simply listens to all the events generated by the GMF editor and filters those that are relevant. The relevant events are those that modify the instantiation of the UML metamodel. When there is one or more events of these kind available, the incremental algorithm is executed using these events as an input parameter.

**The OCL Evaluation Environment.** We need to add a profiler in the Eclipse component that evaluates OCL expressions—the OCL evaluation environment. The profiler is responsible of logging the list of elements $\mathcal{L}$ that are accessed during the evaluation of any OCL expression.

In Sect. 7.1, we have seen that the MDT's OCL Interpreter can be easily extended. The plugin provides access to the evaluation environment. We use this access to integrate a profiler that tracks all the navigations that are performed through the UML metamodel during the evaluation of an expression.

### 8.3.4 Comparison of Regular and Incremental Evaluation of Quality Issues

In [43], the author compares the efficiency of evaluating a set of OCL expressions as (a) a batch process, (b) following a type-based approach, and (c) following the instance-based approach he proposes. Figure 8.3 depicts the results of "empirically evaluating model changes on 24 UML metamodel rules and 29 sample models". A type-based consistency checking is scalable for conceptual schemas with up to 10,000 elements, whereas the batch process can only deal with schemas with up to 1,000 elements. The instance-based approach, on the other hand, remains constant.

In this section, we present the results of a similar experiment we conducted. In particular, we measured the evaluation times per model change for computing the issue instances of the issue types presented in Chap. 6 (excluding syntactic is-

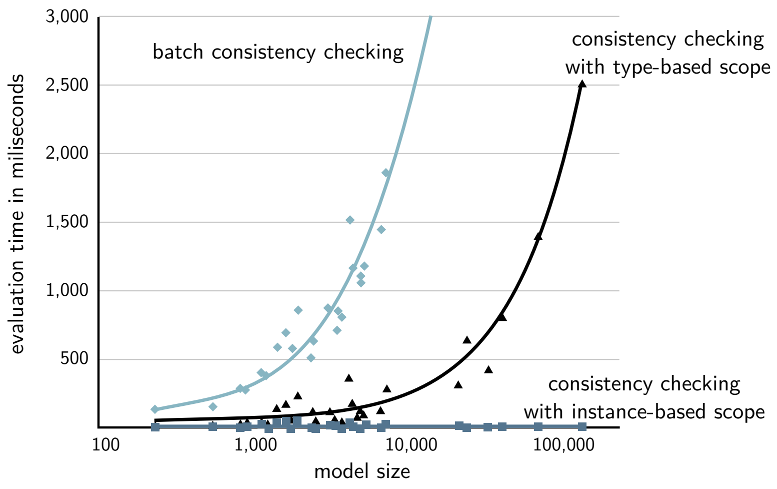


**Figure 8.3.** Comparison of evaluation times per model change using a batch process, a type-based approach, and an instance-based approach, as reported in [43].

sue types) using (a) the algorithm we presented in Chap. 5 and (b) the algorithm presented in this chapter. The issue types were tested over the 13 conceptual schemas first introduced in Sect. 6.4.

Table 8.2 shows the results of the experiment, which are quite conclusive. The incremental approach is able to evaluate the less than half a second, even for the biggest conceptual schema (250 milliseconds). The regular algorithm, on the other hand, takes up to 27.8 seconds for computing issue instances for conceptual schemas with less than 500 elements (even though the average time is 9 seconds). However, the required time this algorithm takes for the biggest conceptual schema (over 5 minutes) makes the approach very ineffective.

There are two main reasons for getting these results. Firstly, the time that evaluating an expression takes depends on the expression itself. Thus, it may be possible that certain expressions, as written by the method engineer, are not the most efficient alternative. Secondly, the use of OCL helper operations (as described in Sect. 7.1) can also be extremely inefficient. Consider, for example, our naming guidelines. These guidelines require some “complex” operations to determine, for instance, whether a *String* is a noun phrase (*isNounPhrase*) and, if it is, whether its head (*getHead*) is a noun (*isNoun*) that is countable (*isCount-*

**Table 8.2.** Characteristics of the conceptual schemas developed by students.

| Project | Number of Elements | Regular Eval. Time (s) | Incremental Eval. Time (s) |
|---|---|---|---|
| P1 | 136 | 2.8 | 0.056 |
| P2 | 181 | 8.1 | 0.063 |
| P3 | 511 | 11.1 | 0.071 |
| P4 | 204 | 12.0 | 0.067 |
| P5 | 200 | 9.3 | 0.058 |
| P6 | 169 | 4.6 | 0.061 |
| P7 | 221 | 7.3 | 0.070 |
| P8 | 402 | 27.8 | 0.083 |
| P9 | 258 | 4.4 | 0.059 |
| P10 | 449 | 5.3 | 0.063 |
| P11 | 346 | 14.3 | 0.062 |
| P12 | 361 | 4.8 | 0.059 |
| P13 | 7,500 | 350.0 | 0.263 |

*able*). These operations were implemented using an online dictionary whose response time is not very fast and may be a few seconds. Therefore, it is clear that avoiding unnecessary calls to the online service—as intended by the incremental method—is an extremely efficient way to improve the responsiveness of our method.

## 8.4 Summary

In this chapter we have seen the importance of an incremental evaluation of quality issue types. Incremental methods re-evaluate expressions if their result may have changed with respect to the last time they were evaluated. The result of an expression can only change if the schema itself has changed.

In Sect. 8.1 we have introduced the necessity of an incremental evaluation by means of a simple example. We have seen that issues can be incrementally evaluated. In Sect. 8.2 we have reviewed some of the available literature on incremental evaluation, focusing our attention on those works related to incremental evaluation of OCL expressions and integrity constraints. Finally, in Sect. 8.3 we have formalized incremental methods and we have focused on an instance-based method that was originally presented in [44]. Next, we have adapted this method to our formalization of issue types. We have seen that we can incrementally evaluate the applicability and issue conditions of an issue type. Moreover, we have also seen how issue precedents affect an incremental evaluation of issue types.

Age brought wisdom (. . . )
C. McCullough, *Too Many Murders*

# 9 Quality Assessment in Current IDEs

Throughout the previous chapters of this thesis we have argued the important role that conceptual schemas play in information system development. In Chap. 4 we presented a method to ensure that a conceptual schema satisfies the quality criteria required by the methods used in their development [18, 28, 69, 75, 85, 111]. The method is based on the notion of "quality issues", which are defined as "an important *topic* or *problem* for debate or discussion". According to our method, conceptual schema have to be "issue-free". This implies that during its development (or once it is finished) it should be checked that the schema contains no issues and, if there are some, that the appropriate actions are taken to fix them before the schema is released.

In this chapter we analyze the support provided by current IDEs in the enforcement of quality criteria, as published in [6]. The goal of this chapter is two-folded. On the one hand, we want to analyze the support that is being currently offered—i.e. which issue types they do enforce *now*. Hence, in Sect. 9.1 we review 29 UML IDEs in order to determine (a) *which* and (b) *how* issue types are being enforced by each tool. On the other hand, we want to know whether this tools provide extension mechanisms so that, given their current implementations, they can easily improve their support by integrating the issues we presented in Chap. 6. Next, Section 9.2 evaluates the benefit that would be gained if current IDEs included all quality issues defined in the catalog. In Sect. 9.3, we formalize their definition of issue type and compare it to ours. Finally, Section 9.4 summarizes the key points discussed in the chapter.

## 9.1 Current Support of Quality Issues

In Chap. 7 we argued that one of the most effective ways of increasing the quality of conceptual schemas is by using an IDE that assists conceptual modelers in detecting and tackling quality issues. As a result, we presented a prototype implementation of our method on top of Eclipse and evaluated the usefulness of such assistance. Here, we analyze the support provided by current IDEs on dealing with issues.

The list of tools presented in Tab. 9.1 has been obtained from the Open Directory Project (ODP) [88] mainly, and complemented with some additional tools that, according to [37], are being used by UML practitioners nowadays. After a quick analysis of the tools included in ODP, we decided to exclude from our analysis the tools that are not intended for conceptual modeling tasks[1].

The analysis we perform in this chapter also includes *ArgoUML* and *SDMetrics*. As we introduced in Sect. 3.2.3, both tools are good examples of how to assist conceptual modelers in creating high quality conceptual schemas. However, it is important to point out that only *ArgoUML* is a UML modeling tool. *SDMetrics* is does not provide any means for defining the conceptual schema; it is a "measurement tool for the UML" that includes several UML design rules. *SDMetrics* permits a conceptual modeler to check whether her UML conceptual schemas adhere to these design rules. We believe *SDMetrics* is a relevant tool for

[1]Some IDEs focus on other activities like code generation, reverse engineering, or transformations from natural language specifications to UML conceptual schemas, among others.

**Table 9.1.** Quality issue enforcement in current IDEs.

| Tool | Issue Type | | | | | Issue | Ext. | Corr. |
|---|---|---|---|---|---|---|---|---|
| | S | S+ | Ba | NG | Be | Tole. | | Act. |
| ArgoUML | ◐ | 3 | 3 | 3 | 5 | ALL | ○ | ● |
| Astah | ◐ | - | - | - | - | FOR | ○ | ○ |
| Blueprint Software Modeler | ◐ | - | - | - | - | MIXED | ○ | ○ |
| Cadifra UML Editor | ○ | - | - | - | - | NONE | ○ | ○ |
| Design Pattern Autom. TK | ◐ | - | - | - | - | NONE | ○ | ○ |
| Dia | ○ | - | - | - | - | NONE | ○ | ○ |
| Eclipse UML2Tools | ◐ | - | - | - | - | MIXED | ○ | ○ |
| Enterprise Architect | ◐ | - | - | - | - | MIXED | ○ | ○ |
| Fujaba | ◐ | - | - | - | - | MIXED | ○ | ○ |
| Gaphor | ○ | - | - | - | - | NONE | ○ | ○ |
| Generic Mod. Env. (GME) | ◐ | 1 | - | - | - | MIXED | ● | ○ |
| IBM Rational Rose | ◐ | - | - | - | - | MIXED | ● | ○ |
| MagicDraw UML | ◐ | - | - | - | - | MIXED | ● | ○ |
| MetaEdit+ | ○ | - | - | - | - | NONE | ○ | ○ |
| MosKITT | ◐ | - | - | - | - | MIXED | ○ | ○ |
| ObjectiF | ◐ | - | - | - | - | FOR | ○ | ○ |
| Oclarity | ● | - | - | - | - | ALL | ○ | ○ |
| Poseidon for UML CE | ○ | - | - | - | - | NONE | ○ | ○ |
| SDMetrics | ◐ | 6 | 3 | 2 | 2 | ALL | ● | ○ |
| Umbrello UML Modeler | ◐ | - | - | - | - | FOR | ○ | ○ |
| UML Sculptor | ○ | - | - | - | - | NONE | ○ | ○ |
| UML/INTERLIS-editor | ◐ | - | - | - | - | MIXED | ○ | ○ |
| UMLet | ○ | - | - | - | - | NONE | ○ | ○ |
| UModel | ● | - | - | - | - | FOR | ○ | ○ |
| USE | ● | - | - | - | - | FOR | ○ | ○ |
| Violet | ○ | - | - | - | - | NONE | ○ | ○ |
| Visio | ◐ | - | - | - | - | FOR | ○ | ○ |
| Visual Case | ○ | - | - | - | - | NONE | ○ | ○ |
| Visual Paradigm (VP) | ◐ | - | - | - | 1 | MIXED | ○ | ● |

our analysis because, as we shall see in this chapter, it is the only tool, along with *ArgoUML*, that includes many design rules.

### 9.1.1 The Analysis

In this section, we aim to analyze, on the one hand, *how many* and *which* issue types are enforced by current IDEs and, on the other hand, *how* IDEs deal with the issue types they enforce. We obtained this information by reviewing the feature list that is published in each tool's website and, moreover, manually testing each tool. Table 9.1 presents the results for the following criteria:

**Issue Types:** It shows the number of issue types enforced by the IDE in each category. The results are presented according to the categories introduced in Chap. 6. For the syntactic category, the symbols used are:

- *full* (●), if the IDE claims it controls all metamodel constraints,
- *partial* (◐), if the IDE controls only a subset of these constraints, and
- *none* or *unknown* (○), otherwise.

**Issue Tolerance:** IDEs that enforce one or more quality issues react differently when they detect an issue is about to be introduced in the conceptual schema. The introduction of an issue may be:

- *Forbidden*, which means that the IDE does not allow a schema change such that it raises the issue, and therefore it "rolls back" the modeler's change to avoid the issue,
- *Allowed*, which means that the change is accepted, and the IDE notifies the modeler somehow of the issue,
- *Mixed*, which means that the IDE allows having some issues raised and prohibits others.

**Extensibility:** IDEs enforcing issue types may offer one or more mechanisms to extend the issue types they deal with. New issue types may be added using a constraint language such as OCL, or creating a plugin that implements an issue type interface. The symbols used are:

- ●, if the tool suggests some sort of extension mechanism, and
- ○, otherwise.

**Corrective Actions:** Whenever an issue is raised in the conceptual schema, the conceptual modeler has to take some action to fix it. IDEs that enforce some quality issues and permit them to be raised should include the set of actions that may fix these issues. The symbols used are:

- ●, if the tool offers issue actions to fix a particular issue, and
- ○, otherwise.

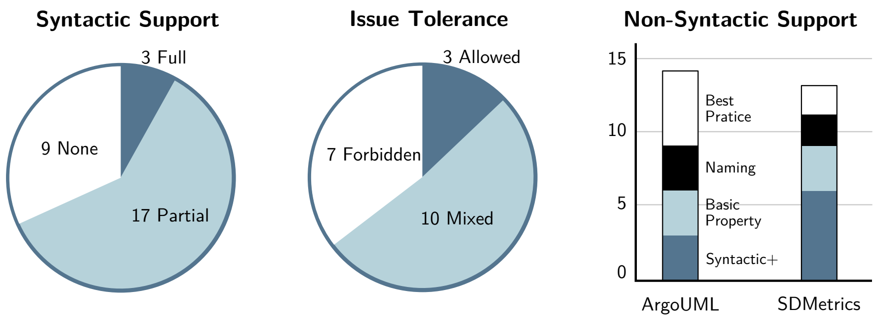


**Figure 9.1.** Summary of quality issue enforcement in current IDEs.

The analysis of the 29 IDEs showed very interesting—as well as unexpected—results, which we summarized graphically in Fig. 9.1. First of all, we can see that the vast majority of the analyzed IDEs assist modelers in dealing with syntactic issue types. However, only three IDEs *fully support* these category of issue types, whilst the others only deal with them partially. After testing each IDE individually, we discovered that many IDEs do not control syntactic issue types like having, for example, (1) a cycle in a generalization hierarchy, (2) a property whose lower multiplicity has a greater value than the upper, or (3) a namespace that contains two different elements that are indistinguishable.

Second, our analysis also shows that, in general, IDEs have little or no support on issue types that are not syntactic—only four tools include non-syntactic quality issue types in their own catalogs and, in fact, only two of them (*ArgoUML* and *SDMetrics*) include a significant number of issue types[2] (14 and 13 respectively). Table 9.2 shows the 21 conceptual schema quality issue types that are included in *ArgoUML*, *SDMetrics*, *Generic Modeling Environment*, and *Visual Paradigm*.

Next, we also detect a trend in issue tolerance. In the literature, we may find some authors who agree that inconsistencies should be tolerated—that is, model consistency has not to be preserved at all times—, and it is the IDE's responsibility to manage the "detection of inconsistencies" [15, 49, 115]. If we look at the

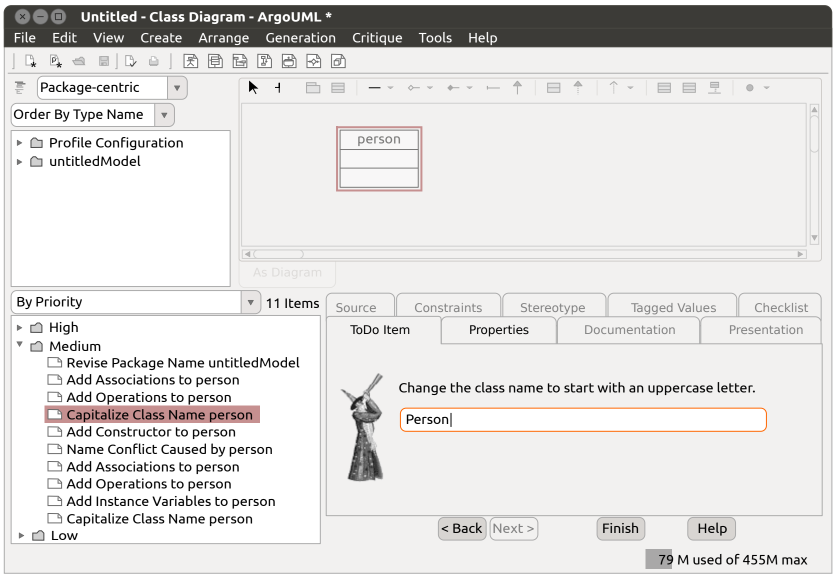
[2]It is important to mention that in this analysis we only considered those quality issues that are related to the conceptual modeling phase of software development. Both *ArgoUML* and *SDMetrics* include many more quality issue types (over 90 and over 140 respectively), mainly related to the design phase.

**Table 9.2.** Conceptual Schema Quality Issues included in some IDEs.

| | **Argo UML** | **SDM** | **GME** | **VP** |
|---|---|---|---|---|
| **Syntactic+** | | | | |
| 1. Overriding attribute does not redefine the overrides one | ○ | ● | ○ | ○ |
| 2. Named element has an illegal name (invalid characters) | ● | ○ | ● | ○ |
| 3. Unnamed class | ● | ● | ○ | ○ |
| 4. Unnamed attribute | ● | ○ | ○ | ○ |
| 5. Unnamed datatype | ○ | ● | ○ | ○ |
| 6. Property without a type | ○ | ● | ○ | ○ |
| 7. Class has specializations and it is marked as a *leaf* | ○ | ● | ○ | ○ |
| 8. *n*-ary association has a navigable member end | ○ | ● | ○ | ○ |
| **Basic Properties** | | | | |
| 9. Binary association with both member ends as aggregate | ● | ● | ○ | ○ |
| 10. Abstract class is not instantiable | ● | ● | ○ | ○ |
| 11. Cycle of composition relationships | ● | ○ | ○ | ○ |
| 12. Abstract class has a parent class that is concrete | ○ | ● | ○ | ○ |
| **Naming Guidelines** | | | | |
| 13. Class name is not properly capitalized | ● | ● | ○ | ○ |
| 14. Property name is not properly capitalized | ● | ● | ○ | ○ |
| 15. Namespace contains two elements with very similar names | ● | ○ | ○ | ○ |
| **Best Practices** | | | | |
| 16. Data type as a member end of a binary association | ● | ○ | ○ | ○ |
| 17. Class without attributes | ● | ○ | ○ | ○ |
| 18. Class with too many associations | ● | ○ | ○ | ○ |
| 19. Class with too many attributes | ● | ○ | ○ | ○ |
| 20. Class with too many attributes and operations | ○ | ● | ○ | ○ |
| 21. Isolated class | ● | ● | ○ | ● |

results, we will realize that two thirds of the analyzed IDEs that include quality issues permit the "introduction" of issues, and thus only one third of the analyzed IDEs forbid their introduction at any time. Another interesting thing that is worth mentioning is that the four IDEs that include non-syntactic issue types tolerate issues: they allow issues of (some or all) types to be raised.

Correcting quality issues is as important as detecting and tracking them. Therefore, issue correction is another important aspect where IDEs can (and should) contribute. IDEs that permit issues to be raised could and should include corrective actions. Unfortunately, solely two IDEs include these kind of actions, which means that the provided assistance is very poor.



High
Medium
Revise Package Name untitledModel
Add Associations to person
Add Operations to person
Capitalize Class Name person
Add Constructor to person
Name Conflict Caused by person
Add Associations to person
Add Operations to person
Add Instance Variables to person
Capitalize Class Name person
Low

Change the class name to start with an uppercase letter.

Person

**Figure 9.2.** Screenshot of *ArgoUML*, where a corrective action is being executed.

In Fig. 9.2, we can see an example of a corrective action being executed in *ArgoUML*. The screenshot depicts a conceptual schema with only one class named *person*. According to *ArgoUML*'s issue types, the class is improperly capitalized. Thus, it offers a corrective action to change its first letter to an upper-case letter. *ArgoUML* automatically proposes *Person* as the new name.

Finally, our analysis also shows that extension mechanisms are not widely present in current IDEs. Only four of them provide a powerful mechanism—that is, OCL or a similar language—to define new issue types. In the next section we discuss the extensibility of these IDEs in-depth, comparing the issue type formalization they implement to ours.

## 9.2 Comparing IDEs vs Our Catalog

In Sect. 6.4 we have described experiment aimed to evaluate the *usefulness* of our method. In order to do so, we analyzed the presence of quality issues in a

**Table 9.3.** Summary of conceptual schema characteristics.

| UML Element | Average | Minimum | Maximum |
|---|---|---|---|
| Classes | 47 | 10 | 366 |
| Associations | 33 | 5 | 264 |
| Association Classes | 7 | 0 | 55 |
| Specializations | 19 | 2 | 158 |
| Attributes | 144 | 11 | 1144 |
| Invariants | 39 | 0 | 386 |

set of thirteen conceptual schemas developed by students. Table 9.3 summarizes the characteristics of these schemas.

Here, we present a parallel experiment where we defined the previous conceptual schemas using the two current tools that provide better support to detect quality issues: *ArgoUML* and *SDMetrics*. As indicated in Tab. 9.1, *ArgoUML* detects 3 syntactic+ issue types, 3 basic properties, 3 naming guidelines and 5 best practices, whereas *SDMetrics* detects 6 syntactic+ issue types, 3 basic properties, 2 naming guidelines and 2 best practices. Both tools partially check syntactic issue types but we do not consider them in this evaluation. We found that all the conceptual schemas present quality issues. Tables 9.4 and 9.5 show the problem issues detected for each category and for all conceptual schemas by *ArgoUML* and *SDMetrics*, respectively.

**Table 9.4.** Issues detected by *ArgoUML* [11].

| | Syntax+ | Basic Prop. | Best Pract. | Naming | Total | Avg. |
|---|---|---|---|---|---|---|
| **Problem** | 0 | 15 | 500 | 23 | 538 | 41.38 |
| **Avg.** | 0 | 1.15 | 38.46 | 1.77 | 41.38 | |

**Table 9.5.** Issues detected by *SDMetrics* [109].

| | Syntax+ | Basic Prop. | Best Pract. | Naming | Total | Avg. |
|---|---|---|---|---|---|---|
| **Problem** | 91 | 15 | 178 | 23 | 307 | 23.62 |
| **Avg.** | 7 | 1.15 | 13.69 | 1.77 | 23.62 | |

*ArgoUML* detects, on average, 42 problem issues for each conceptual schema whereas *SDMetrics* detects 24. If we compare these results with the ones we obtained in Sect. 6.4 (which are summarized in Tab. 9.6), it is clear that the "best" support that practitioners and conceptual modelers can get *today* when defining conceptual schemas can be much greater. The use of a broader catalog with relevant quality issue types for conceptual modeling increases, as expected,

**Table 9.6.** Issues detected by our catalog [7].

| | Syntax+ | Basic Prop. | Best Pract. | Naming | Total | Avg. |
|---|---|---|---|---|---|---|
| **Problem** | 247 | 3317 | 320 | 585 | 4469 | 343.77 |
| **Checking** | 0 | 571 | 726 | 86 | 1383 | 106.38 |
| **Total** | 247 | 3888 | 1046 | 671 | 5852 | 450.15 |
| **Avg.** | 19 | 299.07 | 80.46 | 51.61 | 450.15 | |

the number of detected issues and, therefore, fosters the improvement of the quality of the developed conceptual schemas.

## 9.3 Extending Current IDEs with the Catalog

In Sect. 9.1 we have seen up to which extent current IDEs integrate issue types. Surprisingly, only four tools include non-syntactic issue types, making it clear that the assistance a conceptual modeler can get from these tools is very poor. The results of our analysis also show that only four tools provide extension mechanisms—i.e. they permit the definition and, hence, assessment of new issue types. In this section we analyze their extensibility capabilities in detail. Specifically, we try to determine whether it is possible to integrate into these tools the issue types (that can be) included in our catalog.

### 9.3.1 Comparing Issue Type Formalizations

All IDEs that include syntactic and/or non-syntactic issue types describe them using a certain formalism. In general, this formalism is basically the same used to describe UML metamodel constraints—i.e. they include a *context*, which corresponds to the metatype of the instances for which issues may exist, and a *condition*, which determines whether the issue exists or not for each instance of the context. Besides these two components, our tests determined that IDEs may also include additional elements to their formalization. Basically, IDEs use a formalization that may include (1) permitting the modeler to *ignore* issues and (2) one or more *operations* that automatically solve (or help the modeler solve manually) an issue instance. Formally, the four IDEs that can be extended implement the following definition of issue type:

$$Z_y = \langle \mathcal{C}_y, \theta_y, \mathcal{I}_y, \mathcal{O}_y \rangle \tag{9.1}$$

where

- $\mathcal{C}_y$ is the *context* in which the issue type has to be evaluated,
- a *condition* $\theta_y$ that determines whether the issue exists or not for each instance of the context,
- a *Boolean* value $\mathcal{I}_y$ that specifies whether the issue can be or cannot be *ignored* by the modeler, and
- a set $O_y$ of operations that solve (or help solving) the issue.

It is important to emphasize that the previous formalization represents the *best case scenario*. If we take a look at the reviewed tools, only *ArgoUML* integrates all these components into its issue type formalization. Moreover, it is also important to remember that, unfortunately, *ArgoUML* does not provide any extension mechanism (even though it is an open source project and, thus, anyone can modify its code an integrate new issue types).

Now, in order to determine whether this formalization can integrate the issue types we can define using our method, we shall compare it to our issue type formalization. For the ease of reading, we reproduce our formalization in the following:

$$I_x = \langle \mathcal{S}_x, \phi_x, \rho_x, \mathcal{K}_x, \mathcal{A}_x, \mathcal{O}_x, \mathcal{P}_x \rangle \tag{9.2}$$

where

| | |
|---|---|
| – $\mathcal{S}_x$ is the *scope* of the issue type | – $\mathcal{K}_x$ is the *kind* of the issue type |
| – $\phi_x$ is the *applicability condition* | – $\mathcal{A}_x$ is the *acceptability* of the issue type |
| – $\rho_x$ is the *issue condition* | – $\mathcal{O}_x$ is the set of *issue actions* |
| | – $\mathcal{P}_x$ is the set of *precedents*. |

Clearly, formalization (9.1) is less expressive than 9.2. The main problems we will encounter when trying to integrate an issue type defined using our formalization into an IDE that follows formalization (9.1) are:

- The context $\mathcal{C}_y$ permits one metatype only, whereas our scope $\mathcal{S}_x$ permits one or more metatypes.
- Applicability and issue conditions ($\phi_x$ and $\rho_x$ respectively) have to be condensed into one unique condition $\theta_y$.
- It is not possible to differentiate between *Problem* and *Checking* issues.
- It is not possible to define *precedents*.

All these problems are further discussed in the remaining of the section.

## 9.3.2 Adapting Scopes

The first problem we encounter when adapting an issue type from our formalization to theirs is the scope. Our formalization allows an issue type to have more than one metatype as a scope, whereas theirs only uses one. Using more than one metatype provides better feedback to the modeler under certain circumstances.

Consider, for example, the *pull up property* refactoring, which consists in removing an attribute named $n$ from one (or more) specific classes and defining it in the general class [51]. A situation in which this refactoring would be highly recommended is when the attribute is repeated among all specific classes of a complete generalization set (see Fig. 9.3). We may consider formalizing this situation as a *problem* issue type. As one may expect, one of the issue actions that fixes an issue of this type is applying the refactoring.

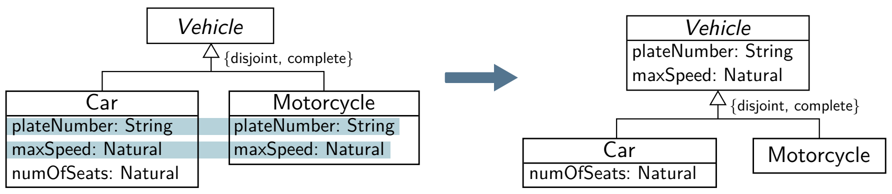


**Figure 9.3.** Example of a conceptual schema with a complete *GeneralizationSet* that has repeated attributes.

Using our formalization (9.2), we may define the scope of this issue type as follows:

$$\mathcal{S}_x = \langle gs : GeneralizationSet, n : String \rangle$$

where $n$ is the name of the attributes that are repeating among all the specific subclasses of the *GeneralizationSet* $gs$. When looking at the example depicted in Fig. 9.3, the only two issue instances of this issue type that would be raised:

- $\langle gs, \text{“}plateNumber\text{”} \rangle$
- $\langle gs, \text{“}maxSpeed\text{”} \rangle$

If, on the other hand, we use formalization (9.1), we have to determine the metatype we will be using to define the context of the issue type. In this case, it is clear that the best candidates are (a) *GeneralizationSet* or (b) *Property*. If *GeneralizationSet* was to be used as the context ($\mathcal{C}_y = GeneralizationSet$), only one issue instance would be raised:

- $\langle gs \rangle$

which would point out that “all subclasses in the *GeneralizationSet* $gs$ have one or more attributes with the same name”. Note we have no specific information on *which* attributes, but only *some* attributes; the modeler is thus responsible of figuring out the specific attributes.

Alternatively, if *Property* was to be used as the context ($\mathcal{C}_y = Property$), four issue instances would be raised:

- $\langle Car :: plateNumber \rangle$
- $\langle Car :: maxSpeed \rangle$
- $\langle Motorcycle :: plateNumber \rangle$
- $\langle Motorcycle :: maxSpeed \rangle$

where each issue instance would point out that "this *Property* is an attribute of a class that belongs to a complete *GeneralizationSet* and, moreover, the name of the attribute is repeated among all subclasses in the *GeneralizationSet*".

Clearly, a scope that permits the inclusion of more than one metatype provides, in general, a more accurate and concise feedback. If, on the other hand, we can only use one metatype, it is still possible to define the issue type, but there are some cases (a) where the feedback will come short, and there are some cases (b) where there will be too many issue instances pointing out the same problem.

### 9.3.3 Adapting Applicability and Issue Conditions

Throughout the whole thesis, we have described issues as "important [*quality*] topics or problems for debate or discussion" which, in essence, are nothing more than conditions. In order to improve the understandability of issue types, we find it convenient to define two different conditions in our formalization for each issue type $\mathcal{I}_x$: the *applicability condition* $\phi_x$ and the *issue condition* $\rho_x$.

Consider, for example, the following problem issue type:

- $\mathcal{I}_a$: "a binary association does not define any of the three names it may define".

Clearly, the scope $\mathcal{S}_a$ is *Association* and the associated conditions are:

- $\phi_a(a{:}Association)$: "$a$ is binary".
- $\rho_x(a{:}Association)$: "$a$ nor any of its member ends ($a$.`memberEnd`) define an explicit name".

In principle, translating these two conditions into a single one—as required by formalization (9.1)—entails no problems. The condition $\theta_a(a{:}Association)$ would thus be defined as follows:

```
context IDE::$\mathcal{I}_a$(a:Association):
   if $\phi_a(a)$ then $\rho_a(a)$
   else false endif
```

In Chap. 8, we have seen that incremental methods try to minimize the number of expression re-executions. In particular, we have seen that we can store the potential sets computed using *applicability conditions* and, thus, minimize the number of executions.

### 9.3.4 Adapting Checking Issues

Another important problem IDE developers would face when trying to integrate our catalog in their tools is that they cannot define *checking* issue types. In general, a checking issue type requires the modeler to check something or perform some action manually, and then notify the tool that the check has been performed.

When the action modifies the schema in a way such that the issue instance is no longer raised, current IDEs would have no problem in integrating a checking issue type. However, there are some cases in which issue types have to remain raised and can only be *checked* and, unfortunately, current IDEs do not provide any mechanisms to the modeler to perform this notification.

To overcome this problem, an IDE developer may be tempted to map a *checking* issue type from our formalization to an *ignorable* issue type in (9.1). However, according to the dictionary, to ignore [an issue] means to "refuse to take notice of or acknowledge [the issue]; disregard [the issue] intentionally", so this mapping would be semantically incorrect and, thus, inaccurate.

### 9.3.5 Adapting Precedents

Finally, IDEs do not allow the definition of precedents. Precedents filter the amount of feedback the modeler receives, because only those issues that may be triggered are actually triggered. Currently, IDEs may be able to implement a similar behaviour by duplicating the definition of a precedent inside their issue condition. However, duplicating information is usually a source of errors, and makes the maintenance and comprehension of an issue type catalog more complicated.

Consider, for example, the following issue types (all available in our catalog

[7]):

- $\mathcal{I}_g$: “generalization hierarchies must be directed and acyclical” [93].
- $\mathcal{I}_r$: “a generalization is redundant”.
- $\mathcal{I}_s$: “An abstract class that has a concrete parent class is relevant” [71].

One may consider, for example, that issue types $\mathcal{I}_r$ and $\mathcal{I}_s$ should only be considered if there were no issue instances of type $\mathcal{I}_g$. We believe this assumption is reasonable because we have to make sure that generalization hierarchies are syntactically correct before considering further issues with regard to them.

Assuming the issue conditions of the previous issue types are $\rho_g$, $\rho_r$, and $\rho_s$, our formalization may define these issue types as follows[3]:

```
context CMA::$\mathcal{I}_g$(Generalization): $\rho_g$

context CMA::$\mathcal{I}_r$(Generalization): $\rho_r$

context CMA::$\mathcal{I}_s$(Class): $\rho_s$
```

where $\mathcal{I}_r$ and $\mathcal{I}_s$ would have $\mathcal{I}_g$ as a global precedent.

On the other hand, formalization (9.1) may define these issue types as follows:

```
context IDE::$\mathcal{I}_g$(Generalization):
   $\rho_g$

context IDE::$\mathcal{I}_r$(Generalization):
   if $\rho_g$ then false
   else $\rho_r$ endif

context IDE::$\mathcal{I}_s$(Class):
   if $\rho_g$ then false
   else $\rho_s$ endif
```

where the guard $\rho_g$ is being repeated in both issue types $\mathcal{I}_r$ and $\mathcal{I}_s$.

[3]For the sake of simplicity, we only include those components that are relevant for the understanding of these example.

## 9.4 Summary

The starting point of this chapter has been the view that one of the most effective ways of increasing the quality of conceptual schemas in practice is by using an IDE that enforces all relevant quality issues. With that view, we have analyzed the support provided by twenty-nine IDEs in the enforcement of quality issues. In particular, we were interested in determining (a) which issue types each IDE includes, (b) how IDEs deal with issue types, (c) whether IDEs permit or not the inclusion of new issue types, and (d) whether IDEs provide or not corrective actions to tackle the issues they find. The results are quite discouraging—only two IDEs (*ArgoUML* and *SDMetrics*) provide some significant support.

In Sect. 9.2, we have compared the support provided by current IDEs with the one that could be provided by those IDEs if they enforced all quality issues defined in our catalog. As a result, we have seen that there is still a lot of room for improvement—*SDMetrics* and *ArgoUML* can detect several quality issues but, as it is expected, not as many as our catalog. The benefit of the additional support provided by our catalog in the quality of the conceptual schemas developed by students is significant.

Finally, we have compared the formalization IDEs ideally implement—which includes a context and an issue condition, the possibility to ignore an issue instance, and the set of operations that solve it—to ours. This comparison make it clear that the formalization IDEs are currently using is much less powerful than ours. Only a few IDEs include non-syntactic issue types, set aside additional support like corrective actions. Nonetheless, we have discussed how an issue type from our catalog can be adapted to this other formalization and we have pointed out the problems they may encounter. The results of this chapter were published in [6].

Anyone can love a thing "because", but
to love something "despite"...

P. Rothfuss, *The Wise Man's Fear*

# 10 Conclusions and Further Work

As the role played by conceptual schemas in software development becomes more relevant, assessing their quality becomes crucial. The quality of a conceptual schema can be analyzed in terms of "quality properties". All conceptual schemas should have the fundamental properties of syntactic and semantic correctness, relevance and completeness, as well as any other quality property that has been proposed in the literature and that may be required or recommended in particular projects. The most effective way to achieve quality is to adopt a systematic and organized approach. Throughout the different chapters of this thesis, we have proposed a method that, on the one hand, permits the definition of these quality properties in a unified way and, on the other hand, fosters the improvement of the quality of the conceptual schemas.

This chapter summarizes the results of the thesis and points out some possible directions for further work. First, Section 10.1 focuses on the problem we address in this thesis and its relevance, and reviews our main contributions to solve it. Section 10.1.1 aligns our research with the problem of developing high-quality conceptual schemas and enumerates the key points of our method. Section 10.1.2 focuses on the formalization our method uses to define quality issues. The section also reviews the experiments we conducted to demonstrate the expressiveness and usefulness of our method when it comes to define quality issues and benefit from using them, respectively. Section 10.1.3 discusses the prototype implementation of our method and its integration to Eclipse. Section 10.1.4 shows the necessity of using an incremental approach to evaluate quality issues. Next, in Sect. 10.2 we discuss some new research lines aimed to improve the support offered by our method. Finally, Section 10.3 lists the publications related to this thesis.

## 10.1 Summary of the Results

This thesis presents a method for the unified definition and treatment of conceptual schema quality issues. Ultimately, the method is aimed at improving the quality of conceptual schemas in a systematic way during its development. In this section, we summarize the main contributions of the research presented throughout the previous chapters.

### 10.1.1 The Challenge of Quality in Conceptual Modeling

Nowadays, conceptual schemas are becoming more and more relevant in the software development process. As a result, assessing their quality becomes crucial. *Engineering* is about getting results of the *required quality* within the schedule and budget [114, p. 8]. The most effective way to do so is to adopt a systematic and organized approach, and software engineering is no exception to this rule.

In Chap. 1, we have stated that the main goal of this thesis is to improve the quality of conceptual schemas. We have seen that the quality of a conceptual schema is the degree up to which a set of properties are met, and that this thesis focus on the syntactic correctness of a conceptual schema and the proper application of any guidelines and/or best practices that are required by the company or project in which the schema is defined.

Following the main ideas of the Design Science Research methodology, we have studied the relevance of the problem we try to solve—i.e. creating high-quality conceptual schemas—, and we have analyzed the different approaches and contributions in the literature related to quality properties and their efficient evaluation (Chapters 6, 8, and 9). As a result, we have proposed a method that permits the unified definition and treatment of these quality properties in terms of conceptual schema quality issues. The method is outlined in Chap. 4 and detailed in the subsequent chapters. Its main components are:

- the definition of quality issue types using our formalization,
- the compilation of quality issue types in a catalog, and
- the usage of quality issue types during the development of a conceptual schema.

### 10.1.2 Formalization of Quality Issues and Compilation of a Catalog

In Chap. 5 we have seen that issues are "important quality topics or problems for debate or discussion". In Sect. 5.1 we have formally described the concepts of quality issue type and quality issue instances, focusing on the different elements that constitute an issue. In the thesis, we have focused on conceptual schemas written in UML/OCL. The quality issues included in our catalog are thus targeted at this language. However, the method could be applied to both DSLs and other conceptual modeling languages.

Two of the most relevant parts of our formalization are (a) the classification of issues according to their kind (which is tightly related to the above definition of issues), and (b) the differences between issue types and issue instances. On the one hand, *problem issues* point out a "problem" that has to be solved by the modeler, whereas *checking issues* notify the modeler of "situation" she has to be aware of (because they may lead to defects in the schema). On the other hand, *issue types* describe the issue "in general", whereas issue instances point out that a certain issue type actually occurs in a concrete conceptual schema.

Chapter 5 continues detailing the different components of our formalization.

This mainly include the lifecycle of an issue instance, an algorithm to detect the issues that a conceptual schema contains, and a mechanism to tackle issues—i.e. issue actions. The quality issues we can define with our method improved the empirical and syntactical quality dimensions of a conceptual schema.

One of our main concerns when proposing our method was to provide a highly *expressive* formalization. In Chap. 6 we have presented a catalog of 65 issue types that has been compiled from the literature. The catalog demonstrates that the vast majority of issue types can be defined using our formalization, only excluding those issues that require information that is not defined in the UML metamodel. Moreover, in Chap. 9 we have analyzed how current IDEs implement issues. We have seen that their formalizations are less expressive and, therefore, they lack several issues our catalog is able to include (e.g. the checking ones).

Another important facet of the method is its *usefulness*. In this sense, in Sect. 6.4 we have presented a small experiment were we had selected 13 projects developed by students as part of their final degree's projects. All conceptual schemas, developed using current[1] modeling tools, presented several quality issues. These issues could have been avoided if our method had been used. Furthermore, in Sect. 9.2 we have demonstrated that the issues types our catalog includes, compared to those IDEs include, are also useful, because all conceptual schemas presented issue instances of these types (which were originally undetected).

### 10.1.3 Implementation and Integration of our Method in an Integrated Development Environment

A fundamental principle of the design-research methodology is that knowledge and understanding of a problem is acquired in the building of an artifact that solves that problem. In the previous section, we have reviewed the catalog of quality issues as one output artifact of our work. Another relevant artifact we have built is the Conceptual Modeling Assistant (CMA).

The CMA is a prototype tool built on top of Eclipse. Its architecture is divided in three main components: (i) the I (()TM) Issue Type Manager, (ii) the Issue

[1]At the time they were developed.

Processor (IP), and (iii) the Issue View (IV). The ITM is responsible of downloading the issue type definitions available in the catalog and controlling which ones are enabled (or disabled) for a particular conceptual schema. The IP is responsible of computing issue instances. This computation was originally performed by the algorithm presented in Sect. 5.1, but it was later changed by the incremental version described in Sect. 8.3. Finally, the IV provides a user interface for the conceptual modeler to see which issues her schema contains.

Another interesting feature of our prototype implementation is its relation with the catalog of issue types. The catalog defines issue types using an XML format and has been made publicly available. The XML permits the definition of all the different elements of our formalization, as well as some additional metadata like the *name*, the *description*, the *references* where the issue was originally described, or a *category*, among others, of an issue type. By using XSLT sheets, the catalog can be browsed by conceptual modelers and practitioners using a web browser in a user-friendly manner. Moreover, we have defined applicability and issue conditions as OCL expressions, so that Eclipse's OCL interpreter can directly use the specification of an issue type as its implementation, reducing the amount of work required to integrate new issue types into a tool.

### 10.1.4 Efficient Evaluation of Quality Issue Types using Incremental Methods

We have discussed that one of the most effective ways of using our method is by integrating it into a modeling environment. Thus, the tool can detect the issues that a conceptual schema contains *as it is being developed*. Unfortunately, when we started testing our prototype implementation, we realized that we were facing efficiency problems: as the number of issue types we wanted to track and the size of the conceptual schema grew, the time required to compute the issues within the conceptual schema became too large. In order to overcome this problem, we have analyzed and integrated an incremental approach in evaluating issue types.

In Chap. 8 we have introduced the necessity of using incremental approaches for computing issue instances. Since our online catalog defines issue types in OCL, Chapter 8 reviews some of the most recent and relevant publications on incremental evaluation of OCL expressions. We have adapted the approach presented in [44] to our method and proposed a new algorithm for computing issues that incrementally determines the *Potential* and *Raised* sets of issues.

We have also compared the efficiency improvement between this new algorithm and the original one. The results have been conclusive and demonstrate that providing *instant* feedback to modelers as they are creating conceptual schemas is possible.

## 10.2 Directions for Further Research

There are several research lines that can be further investigated, provided our work. In this section, we sketch the most interesting ones.

### 10.2.1 Extending the catalog

The first and most obvious research line is to continue expanding the number of issue types our catalog includes. So far, we have only focused on *conceptual modeling quality issues*—i.e. issue types that are specifically targeted to conceptual schemas—and, more precisely, those that deal with structural subschemas defined in UML. In order to provide a better catalog for the conceptual modeling stage, the catalog has to contain issues related to the behavioral subschema, as well as examples of quality issues for other modeling languages and/or DSLs. On the other hand, there is plenty of issue types in the literature for design models (many of which are already integrated in tools like *ArgoUML* or *SDMetrics*). It is also interesting to include these issues in our catalog and, thus, provide a quality control throughout the whole process of developing an information system.

Another problem that may require further addressing is managing a large catalog of quality issue types. As we have described in Chap. 4, quality issue types are defined by method engineers and used by conceptual modelers. Since several issue types are best practices or naming guidelines (i.e. they are "recommendations"), it may be the case that different method engineers propose different issues, which may be conflicting (for example, one naming guideline might state that binary association names have to start with a capital letter, whilst another one might state that they have to start with a lower one). This kind of "conflicting issues" has to be controlled somehow.

### 10.2.2 Validation with Real Users and Projects

In this thesis we have claimed that "one of the most effective ways of creating high-quality conceptual schemas is by using a modeling tool that enforces all the relevant quality criteria". Unfortunately, we have seen that conceptual schemas created by students using current modeling tools contain errors, mainly because these tools "do not enforce all the relevant quality criteria". In order to overcome this problem, we have presented a prototype implementation that integrates issues on top of Eclipse and, as a result, it is able to analyze the conceptual schema as it is being developed.

We believe that the feedback our tool provides can be helpful to conceptual modelers when it comes to improve their schemas. Our tool analyzed the conceptual schema in a non-disruptive manner and offers information on which quality issues are present in a schema and how the conceptual modelers can fix them. Current IDEs follow a similar approach for providing feedback (see Chap. 9), but we have not tested with real users whether this interaction between the conceptual modeler and the modeling tool is the most appropriate. In order to evaluate this particular aspect, we plan to work in two different directions. On the one hand, we have released our prototype tool—i.e. the Eclipse plugin—as an open source project. The source code can be found in [8] and is available for anyone to test and modify. The feedback we may get from real users will help us improve and evolve the prototype tool. On the other hand, we plan to conduct some experiments where students are asked to develop conceptual schemas using our CMA. These controlled experiments will let us determine whether the tool can be used as we expect—i.e. without interrupting the regular tasks of the conceptual modeler, and providing useful feedback.

### 10.2.3 Automatic Reparation of Conceptual Schemas

Although this thesis focuses on *detecting* quality issues, it also includes a mechanism to *fix* them—i.e. issue actions. As we have defined issue actions, whenever a conceptual modeler wants to fix a particular issue, she is the responsible of selecting the appropriate issue action and execute it. Since quality issues are, in general, independent between them, fixing one issue may lead to new issues.

Another interesting research area is the automatic generation of reparation plans. A reparation plan would be defined as the set of issue actions the conceptual modeler has to execute so that all (or as many as possible) issue instances are fixed. There is a lot of work to do in here, because there are situations in which more than one issue action can be executed, but their consequences differ[2].

### 10.2.4 Integration with Other Methodologies

The ultimate goal of this work is to improve the quality of a conceptual schema. There are other approaches in the literature that pursue the same goal. One notable example is the creation of conceptual schemas using *testing* [123, 124]. The testing approach presented in [123] is based on the notion of "test cases", which represents "expected user stories that represent the knowledge that is expected and correct, according to the stakeholders of the system". Thus, a test case is defined as a sequence of states of the information base and certain asserts about them. The conceptual modeler is responsible for defining (with the help of the stakeholders) test cases and making sure they can be successfully executed.

The integration of our method with this testing approach can be done from two different perspectives. On the one hand, the creation of test cases is error prone—test cases have to comply with their metamodel and may have to follow certain guidelines and practices. Therefore, issue types for defining proper test cases may be defined, thus simplifying their creation. On the other hand, the failure of a test case can be seen as a conceptual modeling issue, because "all test cases are expected to end successfully".

## 10.3 Impact of the Thesis

An important component of the design science research methodology is the communication of the results. In this section, we enumerate the scientific publica-

[2]Consider, for example, executing the *pull-up property* action in an incomplete generalization set, as described in Fig. 6.16 (a) and (b).

tions that are related to this thesis, as well as a degree's final project I co-directed.

## 10.3.1 Publications

### Formalization of Conceptual Schema Quality Issues

AGUILERA, D., GÓMEZ, C., AND OLIVÉ, A. A method for the definition and treatment of conceptual schema quality issues. In *ER* (2012), vol. 7532 of *LNCS*, Springer, pp. 501–514

**Abstract:** In the literature, there are many proposals of quality properties of conceptual schemas, but only a few of them (mainly those related to syntax) have been integrated into the development environments used by professionals and students. A possible explanation of this unfortunate fact may be that the proposals have been defined in disparate ways, which makes it difficult to integrate them into those environments. In this paper we define quality properties in terms of quality issues, which essentially are conditions that should not happen, and we propose a unified method for their definition and treatment. We show that our method is able to define most of the existing quality properties in a uniform way and makes it possible to integrate quality issues into development environments. The method can be adapted to several languages. We present a prototype implementation of our method as an Eclipse plugin. We have evaluated the potential usefulness of our method by analyzing the presence of a set of quality issues in a set of conceptual schemas developed by students as part of their projects.

## A Catalog of Quality Issues

AGUILERA, D., GÓMEZ, C., AND OLIVÉ, A. Enforcement of conceptual schema quality issues in current Integrated Development Environments. In *CAiSE* (2013), vol. 7908 of *LNCS*, Springer, pp. 626–640

**Abstract:** We believe that one of the most effective ways of increasing the quality of conceptual schemas in practice is by using an Integrated Development Environment (IDE) that enforces all relevant quality criteria. With this view, in this paper we analyze the support provided by current IDEs in the enforcement of quality criteria and we compare it with the one that could be provided given the current state of the art. We show that there is a large room for improvement. We introduce the idea of a unified catalog that would include all known quality criteria. We present an initial version of this catalog. We then evaluate the effectiveness of the additional support that could be provided by the current IDEs if they enforced all the quality criteria defined in the catalog. We focus on conceptual schemas written in UML/OCL, although our approach could be applied to other languages.

## Conceptual Modeling Assistant

AGUILERA, D., GÓMEZ, C., AND OLIVÉ, A. An eclipse plugin for improving the quality of UML conceptual schemas. In *ER Workshops* (2012), vol. 7518 of *LNCS*, Springer, pp. 387–390

**Abstract:** The development of an information system requires its conceptual schema to be of high quality. Classically, this quality comprises properties such as syntactic and semantic correctness, relevance, and completeness, but many other quality properties have been proposed in the literature. In this demonstration we integrate some published quality properties in Eclipse by extending the core functionalities of MDT. These properties include syntactic issues, naming guidelines, and best practices. A quality property is defined using OCL and is specified in an XML file. The set of quality properties included in our tool is available on an online public catalog that can be extended to include new quality properties. We use XSLT to present this catalog in a friendly manner to users that access it using a web browser.

### Naming Guidelines for the UML

Aguilera, D., García-Ranea, R., Gómez, C., and Olivé, A. An eclipse plugin for validating names in UML conceptual schemas. In *ER Workshops* (2011), vol. 6999 of *LNCS*, Springer, pp. 323–327

**Abstract:** Many authors agree on the importance of choosing good names for conceptual schema elements. Several proposals of naming guidelines are available in the literature, but the support offered by current CASE tools is very limited and, in many cases, insufficient. In this demonstration we present an Eclipse plugin that implements a specific proposal of naming guidelines. The implemented proposal provides a guideline for every kind of named element in UML. By using this plugin, the modelers can automatically check whether the names they gave to UML elements are grammatically correct and generate a verbalization that can be analyzed by domain experts.

Aguilera, D., Gómez, C., and Olivé, A. A complete set of guidelines for naming UML conceptual schema elements. *Data Knowl. Eng. 88*, 0 (2013), 60–74

**Abstract:** We focus on the problem of naming conceptual schema elements in UML, which is faced by conceptual modelers every time they define a new element that requires a name. The problem is significant because in general there are many elements that require a name, and the names given have a strong influence on the understandability of that schema. We propose a guideline for every kind of element to which a conceptual modeler may give a name in UML. The guideline comprises the grammar form of the name and a pattern sentence. A name complies with our guideline if it has that form and the sentence generated from the pattern sentence is grammatically well-formed and semantically meaningful. The main novelty of our proposal is that it is (as far as we know) the first that provides a naming guideline for each kind of element of conceptual schemas in UML.

### 10.3.2 Degree's Final Project

GARCÍA-RANEA, R. Desarrollo de un plug-in de Eclipse para validar los nombres de los elementos de un esquema conceptual. In *Facultat d'Informàtica de Barcelona, Universitat Politècnica de Catalunya* (2011), (co-directed by David Aguilera and Cristina Gómez)

**Abstract:** Conceptual schemas are a key component in information system development, since they provide an abstraction of the real-world. Generally, a conceptual schema comprises a *structural schema* and a *behavioural schema*. The names given to conceptual schema elements play an important role in the schema understandability. Several proposals of naming guidelines are available in the literature, but the support offered by current CASE tools is very limited. In this work, Raúl presents an Eclipse plugin that implements a specific proposal of naming guidelines. The implemented proposal provides a guideline for every kind of named element in UML.

# Bibliography

[1] Adobe. *ActionScript 2.0 Best Practices*.

[2] Aguilera, D., García-Ranea, R., Gómez, C., and Olivé, A. An eclipse plugin for validating names in UML conceptual schemas. In *ER Workshops* (2011), vol. 6999 of *LNCS*, Springer, pp. 323–327.

[3] Aguilera, D., Gómez, C., and Olivé, A. An eclipse plugin for improving the quality of UML conceptual schemas. In *ER Workshops* (2012), vol. 7518 of *LNCS*, Springer, pp. 387–390.

[4] Aguilera, D., Gómez, C., and Olivé, A. A method for the definition and treatment of conceptual schema quality issues. In *ER* (2012), vol. 7532 of *LNCS*, Springer, pp. 501–514.

[5] Aguilera, D., Gómez, C., and Olivé, A. A complete set of guidelines for naming UML conceptual schema elements. *Data Knowl. Eng. 88*, 0 (2013), 60–74.

[6] Aguilera, D., Gómez, C., and Olivé, A. Enforcement of conceptual schema quality issues in current Integrated Development Environments. In *CAiSE* (2013), vol. 7908 of *LNCS*, Springer, pp. 626–640.

[7] Aguilera, D., Gómez, C., and Olivé, A. Issue catalog, `http://helios.lsi.upc.edu/phd/catalog/issues.php`.

[8] Aguilera, D., Gómez, C., and Olivé, A. Conceptual Modeling Assistant (CMA) for Eclipse, `http://helios.lsi.upc.edu/phd/downloads`.

[9] Akoka, J., Comyn-Wattiau, I., and Cherfi, S. S.-S. Quality of conceptual schemas an experimental comparison. In *RCIS* (2008), pp. 197–208.

[10] Ambler, S. W. *The Elements of UML 2.0 Style*. Cambridge University, 2005.

[11] ArgoUML. ArgoUML, `http://argouml.tigris.org`.

[12] Atkinson, C., and Kühne, T. Model-driven development: A metamodeling foundation. *IEEE Softw. 20*, 5 (2003), 36–41.

[13] Barker, R. *CASE Method: Entity Relationship Modelling*, 1st ed. Addison-Wesley, 1990.

[14] Becker, J., Delfmann, P., Herwig, S., Lis, L., and Stein, A. Formalizing linguistic conventions for conceptual models. In *ER* (2009), vol. 5829 of *LNCS*, Springer, pp. 70–83.

[15] Blanc, X., Mougenot, A., Mounier, I., and Mens, T. Incremental detection of model inconsistencies based on model operations. In *CAiSE* (2009), vol. 5565 of *LNCS*, Springer, pp. 32–46.

[16] Boehm, B. W., and Basili, V. R. Software defect reduction top 10 list. *IEEE Comput. 34*, 1 (2001), 135–137.

[17] Boger, M., Sturm, T., and Fragemann, P. Refactoring browser for UML. In *NetObjectDays* (2002), vol. 2591 of *LNCS*, Springer, pp. 366–377.

[18] Bolloju, N., and Leung, F. S. Assisting novice analysts in developing quality conceptual models with UML. *Commun. ACM 49*, 7 (2006), 108–112.

[19] Bolloju, N., and Sugumaran, V. A knowledge-based object modeling advisor for developing quality object models. *Expert Syst. Appl. 39*, 3 (2012), 2893–2906.

[20] Booch, G. *Object-Oriented Design with Applications*. Benjamin-Cummings, 1991.

[21] Butler, S., Wermelinger, M., Yu, Y., and Sharp, H. Relating identifier naming flaws and code quality: An empirical study. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering* (2009), WCRE '09, IEEE Comput. Soc., pp. 31–35.

[22] Cabot, J. *Incremental Integrity Checking in UML/OCL Conceptual Schemas*. BarcelonaTech – PhD Thesis, 2006.

[23] Cabot, J., and Teniente, E. Incremental integrity checking of UML/OCL conceptual schemas. *J. of Syst. and Softw. 82*, 9 (2009), 1459–1478.

[24] Ceri, Stefanoand Widom, J. Deriving production rules for constraint maintainance. In *16th International Conference on Very Large Data Bases, Proceedings* (1990), Morgan Kaufmann, pp. 566–577.

[25] Ceri, S., Fraternali, P., Paraboschi, S., and Tanca, L. Automatic generation of production rules for integrity maintenance. *ACM Trans. Database Syst. 19*, 3 (1994), 367–422.

[26] Chen, P. The entity-relationship model – toward a unified view of data. *ACM Trans. Database Syst. 1* (1976), 9–36.

[27] Chen, P. English sentence structure and entity-relationship diagrams. *Inf. Sci.*, 2-3 (1983), 127–149.

[28] Cherfi, S. S.-S., Akoka, J., and Comyn-Wattiau, I. Conceptual modeling quality - from EER to UML schemas evaluation. In *ER* (2002), vol. 2503 of *LNCS*, Springer, pp. 414–428.

[29] Cherfi, S. S.-S., Akoka, J., and Comyn-Wattiau, I. Measuring UML conceptual modeling quality, method and implementation. In *BDA* (2002).

[30] Cherfi, S. S.-S., Akoka, J., and Comyn-Wattiau, I. Perceived vs. measured quality of conceptual schemas: An experimental comparison. In *ER (Tutorials, Posters, Panels & Industrial Contributions)* (2007), vol. 83 of *CRPIT*, ACM, pp. 185–190.

[31] Cherfi, S. S.-S., Comyn-Wattiau, I., and Akoka, J. Quality patterns for conceptual modelling. In *ER* (2008), vol. 5231 of *LNCS*, Springer, pp. 142–153.

[32] Clayberg, E., and Rubel, D. *Eclipse Plug-ins*. Addison-Wesley, 2008.

[33] Costal, D., and Gómez, C. On the use of association redefinition in UML class diagrams. In *ER*, vol. 4215 of *LNCS*. Springer, 2006, pp. 513–527.

[34] Costal, D., Gómez, C., Queralt, A., Raventós, R., and Teniente, E. Facilitating the definition of general constraints in UML. In *Model Driven Engineering Languages and Systems*, vol. 4199 of *LNCS*. Springer, 2006, pp. 260–274.

[35] Curland, M., and Halpin, T. A. Enhanced verbalization of ORM models. In *OTM Workshops* (2012), vol. 7567 of *LNCS*, Springer, pp. 399–408.

[36] Cycorp. *Cyc Ontology*.

[37] Davies, I., Green, P., Rosemann, M., Indulska, M., and Gallo, S. How do practitioners use conceptual modeling in practice? *Data Knowl. Eng. 58*, 3 (2006), 358–380.

[38] Deissenboeck, F., and Pizka, M. Concise and consistent naming. *Softw. Qual. Control 14* (2006), 261–282.

[39] Dennis, A., Wixom, B. H., and Tegarden, D. *System Analysis and Design with UML Version 2.0*. Wiley, 2005.

[40] Dinh-Trong, T. T., Kawane, N., Ghosh, S., France, R. B., and Amschler Andrews, A. A tool-supported approach to testing UML design models. In *ICECCS* (2005), IEEE Comput. Soc., pp. 519–528.

[41] Eclipse Community. MDT-UML2Tools, `http://wiki.eclipse.org/MDT-UML2Tools`.

[42] Eclipse Foundation. Eclipse project, `http://www.eclipse.org`.

[43] Egyed, A. Instant consistency checking for the UML. In *ICSE* (2006), ACM, pp. 381–390.

[44] Egyed, A. Automatically detecting and tracking inconsistencies in software design models. *IEEE Trans. Softw. Eng. 37*, 2 (2011), 188–204.

[45] Egyed, A., and Balzer, R. Integrating cots software into systems through instrumentation and reasoning. *Automated Software Eng. 13*, 1 (2006), 41–64.

[46] Embley, D. W., Kurtz, B., and Woodfield, S. *Object-Oriented Systems Analysis: A Model-Driven Approach*. Yourdon Press, 1992.

[47] Endres, A., and Rombach, D. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*. Addison-Wesley, 2003.

[48] Fagan, M. E. Advances in software inspections. *IEEE Trans. Softw. Eng. 12*, 7 (1986), 744–751.

[49] Finkelstein, A. C. W., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. Inconsistency handling in multiperspective specifications. *IEEE Trans. Softw. Eng. 20*, 8 (1994), 569–578.

[50] Fliedl, G., Kop, C., and Vöhringer, J. Guideline based evaluation and verbalization of OWL class and property labels. *Data Knowl. Eng. 69* (2010), 331–342.

[51] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[52] France, R. B., and Rumpe, B. Model-driven development of complex software: A research roadmap. In *FOSE* (2007), pp. 37–54.

[53] García-Ranea, R. Desarrollo de un plug-in de Eclipse para validar los nombres de los elementos de un esquema conceptual. In *Facultat d'Informàtica de Barcelona, Universitat Politècnica de Catalunya* (2011), (co-directed by David Aguilera and Cristina Gómez).

[54] Garvin, D. A. What does "product quality" really mean? *Sloan Manag. Review 26*, 1 (1984), 25–43.

[55] Genero, M., Fernández-Sáez, A. M., Nelson, H. J., Poels, G., and Piattini, M. Research review: A systematic literature review on the quality of UML models. *J. of Database Manag. 22*, 3 (2011), 46–70.

[56] Gogolla, M. UML and OCL in conceptual modeling. In *Handbook of Conceptual Modeling*, D. W. Embley and B. Thalheim, Eds. Springer, 2011, pp. 85–122.

[57] Griethuysen, J. v. *Concepts and Terminology for the Conceptual Schema and the Information Base*. ISO TC97/SC5/WG3, 1982.

[58] Halpin, T. *Information Modeling and Relational Databases: from Conceptual Analysis to Logical Design*. Morgan Kaufmann, 2001.

[59] Hartmann, S. On the consistency of int-cardinality constraints. In *ER*, vol. 1507 of *LNCS*. Springer, 1998, pp. 150–163.

[60] Hartmann, S. Coping with inconsistent constraint specifications. In *ER* (2001), vol. 2224 of *LNCS*, Springer, pp. 241–255.

[61] HAY, D. C. *Data Model Patterns: Conventions of Thought*, 1st ed. Dorset House, 1996.

[62] HEVNER, A. R., MARCH, S. T., PARK, J., AND RAM, S. Design science in information systems research. *MIS Q. 28*, 1 (2004), 75–105.

[63] IEEE COMPUT. SOC. *IEEE 1320.2: Standard for Conceptual Modeling Language Syntax and Semantics for IDEF1X97 (IDEFobject)*, 1998.

[64] INTERNATIONAL STANDARDS ORGANIZATION (ISO). ISO TC97/SCS/WG3: Concepts and terminology for the conceptual schema and the information base, 1982.

[65] INTERNATIONAL STANDARDS ORGANIZATION (ISO). ISO Standard 9000-2000: Quality management systems: Fundamentals and vocabulary, 2000.

[66] INTERNATIONAL STANDARDS ORGANIZATION (ISO). ISO Standard 9126: Software product quality, 2001.

[67] KALJURAND, K., AND E. FUCHS, N. Verbalizing OWL in Attempto Controlled English. In *Proceedings of Third International Workshop on OWL: Experiences and Directions* (2007), vol. 258.

[68] KLEPPE, A. G., WARMER, J., AND BAST, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[69] KROGSTIE, J. *Model-Based Development and Evolution of Information Systems – A Quality Approach*. Springer, 2012.

[70] KROGSTIE, J., SINDRE, G., AND JØRGENSEN, H. Process models representing knowledge for action: a revised quality framework. *Eur. J. Inf. Syst. 15*, 1 (2006), 91–102.

[71] LANGE, C. F. *Empirical Investigations in Software Architecture Completeness*. Master's Thesis, Department of Mathematics and Computing Science, Technical University Eindhoven, 2003.

[72] LARMAN, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3 ed. Prentice-Hall, 2005.

[73] LAUESEN, S., AND VINTER, O. Preventing requirement defects: An experiment in process improvement. *Requir. Eng. 6*, 1 (2001), 37–50.

[74] Lemaitre, J., and Hainaut, J.-L. Quality evaluation and improvement framework for database schemas – using defect taxonomies. In *CAiSE*, vol. 6741 of *LNCS*. Springer, 2011, pp. 536–550.

[75] Lindland, O. I., Sindre, G., and Sølvberg, A. Understanding quality in conceptual modeling. *IEEE Softw. 11*, 2 (1994), 42–49.

[76] Maes, A., and Poels, G. Evaluating quality of conceptual models based on user perceptions. In *ER*, vol. 4215 of *LNCS*. Springer, 2006, pp. 54–67.

[77] McAllister, A. Complete rules for n-ary relationship cardinality constraints. *Data Knowl. Eng. 27*, 3 (1998), 255–288.

[78] Mellor, S. J., and Balcer, M. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley, 2002.

[79] Meyer, B. *Reusable Software: the Base object-oriented component libraries*. Prentice-Hall, 1994.

[80] Meyer, B. *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.

[81] Meziane, F., Athanasakis, N., and Ananiadou, S. Generating natural language specifications from UML class diagrams. *Requir. Eng. 13*, 1 (2008), 1–18.

[82] Microsoft. *.NET Framework General Reference (v.1.1): Naming Guidelines*.

[83] Microsoft. *.NET Framework General Reference (v.4.0): Capitalization Conventions*.

[84] Moody, D. L. Metrics for evaluating the quality of entity relationship models. In *ER*, vol. 1507 of *LNCS*. Springer, 1998, pp. 211–225.

[85] Moody, D. L. Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. *Data Knowl. Eng. 55*, 3 (2005), 243–276.

[86] Moody, D. L., and Shanks, G. What makes a good data model? evaluating the quality of entity relationship models. In *ER*, vol. 881 of *LNCS*. Springer, 1994, pp. 94–111.

[87] MOODY, D. L., SINDRE, G., BRASETHVIK, T., AND SØLVBERG, A. Evaluating the quality of information models: Empirical testing of a conceptual model quality framework. In *ICSE* (2003), pp. 295–307.

[88] MOZILLA. Open Directory Project (ODP) – List of UML tools, `http://www.dmoz.org`.

[89] NELSON, H. J., POELS, G., GENERO, M., AND PIATTINI, M. A conceptual modeling quality framework. *Softw. Qual. Control 20*, 1 (2012), 201–228.

[90] NIJSSEN, G. M., AND HALPIN, T. A. *Conceptual Schema and Relational Database Design*. Prentice-Hall, 1989.

[91] NUGROHO, A., AND CHAUDRON, M. Evaluating the impact of UML modeling on software quality: An industrial case study. In *MoDELS*, vol. 5795 of *LNCS*. Springer, 2009, pp. 181–195.

[92] OBJECT MANAGEMENT GROUP (OMG). *Object Constraint Language*, 2010.

[93] OBJECT MANAGEMENT GROUP (OMG). *Unified Modeling Language (UML), Superstructure*, 2011.

[94] OLIVÉ, A. Conceptual schema-centric development: A grand challenge for information systems research. In *CAiSE* (2005), vol. 3520 of *LNCS*, Springer, pp. 1–15.

[95] OLIVÉ, A. A method for the definition of integrity constraints in object-oriented conceptual modeling languages. *Data Knowl. Eng. 59* (2006), 559–575.

[96] OLIVÉ, A., AND RAVENTÓS, R. Modeling events as entities in object-oriented conceptual modeling languages. *Data Knowl. Eng. 58* (2006), 243–262.

[97] OLIVÉ, A. *Conceptual Modeling of Information Systems*. Springer, 2007.

[98] OLIVÉ, A., AND CABOT, J. A research agenda for conceptual schema-centric development. *Conceptual Modelling in Information Systems Engineering* (2007), 319–334.

[99] PASTOR, O., AND MOLINA, J. C. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*. Springer, 2007.

[100] Pipino, L., Lee, Y. W., and Y. Wang, R. Data quality assessment. *Commun. ACM 45*, 4 (2002), 211–218.

[101] Posner, R. Charles morris and the behavioral foundations of semiotics. In *Classics of Semiotics* (1987), Plenum, pp. 23–57.

[102] Price, R., and Shanks, G. G. A semiotic information quality framework: development and comparative analysis. *J. of Inf. Tech. 20*, 2 (2005), 88–102.

[103] Queralt, A., Artale, A., Calvanese, D., and Teniente, E. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data Knowl. Eng. 73* (2012), 1–22.

[104] Queralt, A., and Teniente, E. Verification and validation of UML conceptual schemas with OCL constraints. *ACM Trans. Softw. Eng. Methodol. 21*, 2 (2012), 13:1–13:41.

[105] Ramírez, A., Vanpeperstraete, P., Rueckert, A., Odutola, K., Bennett, J., Tolke, L., and van der Wulp, M. ArgoUML user manual: A tutorial and reference description. Tech. rep., 2000–2009.

[106] Relf, P. A. Achieving software quality through identifier names. In *Qualcon 2004* (2004), pp. 33–34.

[107] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-oriented modeling and design*. Prentice-Hall, 1991.

[108] Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language Reference Manual (2nd Edition)*. Addison-Wesley, 2005.

[109] SDMetrics. The software design metrics tool for the UML, `http://sdmetrics.com`.

[110] Selic, B. The pragmatics of model-driven development. *IEEE Softw. 20*, 5 (2003), 19–25.

[111] Shanks, G., Tansley, E., and Weber, R. Using ontology to validate conceptual models. *Commun. ACM 46*, 10 (2003), 85–89.

[112] Silva, M. A. A. d., Mougenot, A., Blanc, X., and Bendraou, R. Towards automated inconsistency handling in design models. In *CAiSE* (2010), vol. 6051 of *LNCS*, Springer, pp. 348–362.

[113] SIMSION, G. C., AND WITT, G. C. *Data Modeling Essentials*, 3rd ed. Morgan Kaufmann, 2005.

[114] SOMMERVILLE, I. *Software Engineering*, 9 ed. Addison-Wesley, 2010.

[115] SPANOUDAKIS, G., AND ZISMAN, A. Inconsistency management in software engineering: Survey and open research issues. In *In Handbook Of Software Engineering and Knowledge Engineering* (2001), World Scientific, pp. 329–380.

[116] STAMELOS, I., ANGELIS, L., OIKONOMOU, A., AND L. BLERIS, G. Code quality analysis in open source software development. *Inf. Syst. J. 12*, 1 (2002), 43–60.

[117] STARR, L. *Executable UML: How to Build Class Models*. Prentice-Hall, 2002.

[118] STEINBERG, D., BUDINSKY, F., PATERNOSTRO, M., AND MERKS, E. *EMF: Eclipse Modeling Framework*, 2 ed. Addison-Wesley, 2008.

[119] STRAETEN, R. V. D., MENS, T., AND BAELEN, S. V. Challenges in model-driven software engineering. In *MoDELS Workshops* (2008), vol. 5421 of *LNCS*, Springer, pp. 35–47.

[120] SUN MICROSYSTEMS. *Code conventions for the Java Programming Language*, 1997.

[121] TAKEDA, H., VEERKAMP, P., TOMIYAMA, T., AND YOSHIKAWA, H. Modeling design processes. *AI Mag. 11*, 4 (1990), 37–48.

[122] TENIENTE, E., AND URPÍ, T. On the abductive or deductive nature of database schema validation and update processing problems. *Theory Pract. Log. Program. 3*, 3 (2003), 287–327.

[123] TORT, A., OLIVÉ, A., AND SANCHO, M.-R. An approach to test-driven development of conceptual schemas. *Data Knowl. Eng. 70*, 12 (2011), 1088–1111.

[124] TORT, A., OLIVÉ, A., AND SANCHO, M.-R. On checking executable conceptual schema validity by testing. In *DEXA (1)* (2012), vol. 7446 of *LNCS*, Springer, pp. 249–264.

[125] Tort, A., Olivé, A., and Sancho, M.-R. The CSTL processor: A tool for automated conceptual schema testing. In *ER Workshops*, vol. 6999 of *LNCS*. Springer, 2011, pp. 349–352.

[126] Urpí, T., and Olivé, A. A method for change computation in deductive databases. In *VLDB* (1992), Morgan Kaufmann, pp. 225–237.

[127] Urpí, T., and Olivé, A. Semantic change computation optimization in active databases. In *RIDE-ADS* (1994), IEEE Comput. Soc., pp. 19–27.

[128] Vaishnavi, V., and Kuechler, W. Desing research in information systems, 2004.

[129] van Vliet, H. *Software Engineering: Principles and Practice*, 2 ed. John Wiley & Sons, 2000.

[130] W3C. *Extensible Markup Language (XML)*.

[131] W3C. *Web Ontology Language (OWL)*.

[132] Wand, Y., and Wang, R. Y. Anchoring data quality dimensions in ontological foundations. *Commun. ACM 39*, 11 (1996), 86–95.

[133] Wand, Y., and Weber, R. Research commentary: Information systems and conceptual modeling – a research agenda. *Inf. Syst. Research 13*, 4 (2002), 363–376.

[134] Wieringa, R. J. *Design methods for reactive systems - Yourdon, Statemate, and the UML*. Morgan Kaufmann, 2003.

[135] Wohed, P. Tool support for reuse of analysis patterns: a case study. In *ER* (2000), vol. 1920 of *LNCS*, Springer, pp. 196–209.

[136] Y. Wang, R., and M. Strong, D. Beyond accuracy: What data quality means to data consumers. *J. of Manag. Inf. Syst. 12*, 4 (1996), 5–33.

# Appendices

A word is nothing but a painting of a fire. A name is the fire itself.

P. Rothfuss, *The Name of the Wind*

# A
# Naming Guidelines for the Unified Modeling Language

The names conceptual modelers give to the elements of a conceptual schema have a strong influence on the understandability of that schema. It is widely recognized that good names make it easier for requirement engineers, conceptual modelers, system developers and users to understand conceptual schemas [1, 38, 79, 82]. However, choosing good names is one of the most difficult aspects of conceptual modeling [107, p. 46].

The conceptual modeling quality framework proposed by Lindland et al. [75] distinguishes among three types of quality: syntactic, semantic and pragmatic. The goal of pragmatic quality is comprehension, meaning that all concerned parties completely understand the statements in the model that are relevant to them [102]. Giving good names to the elements of a conceptual schema makes that schema easier to understand. On the other hand, in the related field of programming, it has been reported statistically significant associations between the number of identifiers that violate naming guidelines (an adaptation of Relf's guidelines [106]) and the program code quality in a set of eight established open source Java application libraries [21].

In this appendix we put forward a proposal of naming guidelines for UML structural and behavioral conceptual schemas [5]. In UML, the structural schema consists of a set of classes, association classes and data types (collectively called here entity types), with their generalization relationships, a set of attributes of those classes, a set of associations, and a set of invariants (integrity constraints). The behavioral schema in UML is defined by either event types or system operations, with their pre/postconditions and state machines [56, 72, 103, 104, 97, 123]. Figure A.1 shows a fragment of the UML metamodel [93] comprising the main *NamedElements* used for defining UML conceptual schemas.

In the literature, there have been several proposals of naming guidelines for conceptual schema elements. Here, we systematically review most of them. Some of the guidelines we propose are built on existing proposals, while others are new. Our main contributions are: (1) a naming guideline for each kind of element of conceptual schemas in UML; (2) the definition of a guideline in terms of a grammatical form of the name, and its pattern sentence; and (3) the pattern sentence of each guideline. There has been similar work for other languages, notably ORM [90, 58, 35], but ours is the first to deal with UML.

The structure of the rest of this appendix is as follows. Sections A.1 to A.6 present the naming guidelines. Each section starts with a review of the relevant literature. Section A.1 deals with entity types and related concepts. Section A.2 describes the naming guidelines for attributes, which take into account their cardinality constraints. Section A.3 describes the naming guidelines for associations, also taking into account their cardinality constraints. Section A.4 presents the naming guideline of invariants, which in UML are the non-graphical static constraints. Section A.5 presents the naming guidelines for event types and related concepts. Section A.6 presents the naming guidelines for UML states, the basic components of the popular state machine or statecharts diagrams.

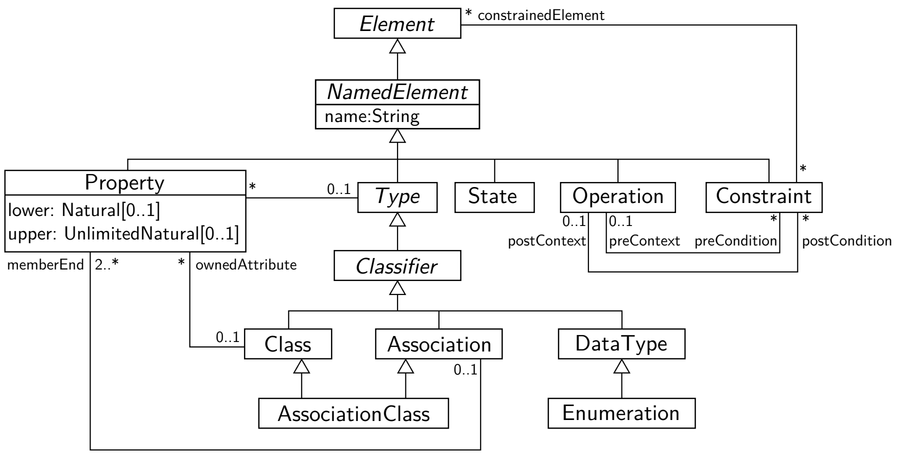


**Figure A.1.** Main *NamedElements* used for defining conceptual schemas in UML.

## A.1 Entity types

In this section we propose a naming guideline for UML constructs related to entity types. Our guideline applies to UML classes, association classes and data types (including enumerations). We first summarize a literature review on similar naming guidelines and then we present our proposal.

### A.1.1 Literature review

[27] was one of the first works that analyzed the correspondence between English sentences and entity types. With respect to entity types, [27] proposed two rules (or guidelines): the first states that “a common noun (such as *person*, *chair*) in English corresponds to an entity type in an ER diagram” [27, p. 130] and the second that “a gerund in English corresponds to a relationship-converted entity type in ER diagrams” [27, p. 135]. The first rule was adopted by most of the later work. [10, 13, 46, 61, 79, 80, 90] suggested that the name of an entity type should be a singular noun (possibly qualified), a recommendation that has been widely followed. The standard IDEF1 observed that the name may be “a noun or a noun phrase” [63, p. 35].

[81] examined 45 class diagrams from the object oriented literature, mainly textbooks, and found that "the names given to classes are either a noun (N), a pair of nouns (NN) or a sequence of three nouns (NNN). These represent 97% of all the names surveyed in this study. An exception to these rules is an adjective followed by a noun (AN), which is found in nearly 3% of the surveyed class names". In the field of programming, [38, p. 6] reports that about 80% of all identifiers in large programs are compounds.

A similar approach has been taken by the proposers of guidelines for naming OWL classes. [67, p. 7] added that "named classes are denoted by singular countable nouns". [50, p. 335] observed the need for delimiters in OWL class names: if a class "consists of more than one term, a definite delimiter between the terms must be used. Here we follow the guideline that an upper case character works as delimiter" and "a class should always start with upper case". Some works call "the Pascal[1] case" to this capitalization form [82].

The above guidelines have been adopted for similar contexts by a few popular professional best practices [1, 82, 120].

In summary, we can say that the literature shows a consensus on how to name entity types. The guideline we propose below is in line with this consensus.

### A.1.2 Naming guideline

The first guideline $\mathcal{G}_1$ deals with the names of UML classes, association classes and data types (collectively called entity types). The guideline consists of the form of the name $\mathcal{G}_{1f}$ and the pattern sentence $\mathcal{G}_{1s}$ (see next page).

It is easy to see that names such as *Person*, *Chair*, *Invoice* or *Category* (classes), *Job* or *Enrollment* (association classes), *Date* or *AmountOfMoney* (data types), and *Sex* or *Color* (enumerations) follow the guideline. An example of a complex name that also follows the guideline, taken from the Cyc ontology [36], is *Path-ForWheeledVehicles*. In this example, the head of the name (*Path*) is a countable

[1]Pascal or Camel case is the practice of writing compound words or phrases in which the elements are joined without spaces, with each element's initial letter capitalized within the compound. In computer programming if the first letter is capitalized, it is called Pascal case; if not, then Camel case (from Wikipedia, CamelCase).

**Entity Types**

$\mathcal{G}_{1f}$ The name of an entity type should be a noun phrase whose head is a countable noun in singular form. The name should be written in the Pascal case.

$\mathcal{G}_{1s}$ If $N$ is the name of an entity type, then the following sentence must be grammatically well-formed and semantically meaningful:

An instance of the entity type named $N$ is [a|an] *lower*$(N)$.

where *lower*$(N)$ is a function that gives $N$ in lower case and using blanks as delimiters.

When the entity type is implicit from the context, the pattern sentence is just:

An instance of this entity type is [a|an] *lower*$(N)$.

noun in singular form, and the sentence generated from this name by using the pattern sentence $\mathcal{G}_{1s}$ is:

An instance of this entity type is a path for wheeled vehicles.

which is grammatically well-formed and semantically meaningful. The UML metamodel [93] also uses many complex names that follow this guideline, such as *DirectedRelationship*, *NamedElement*, *ValueSpecification*, *AssociationClass*, *GeneralizationSet*, and *ProtocolStateMachine*, among others.

Note that guideline $\mathcal{G}_{1f}$ requires the head of the noun phrase to be a "countable noun in singular form". This excludes entity type names such as *Water* or *Gold*, which are uncountable (or mass) nouns and whose instances are portions or amounts of a substance or thing. For these (rare) cases we could suggest a different pattern sentence, such as:

An instance of this entity type is a portion of *lower*$(N)$.

but we refrain from doing it because in general the added complexity of the guideline is not strictly needed. In these cases, we suggest to add a prefix to the name such that it becomes countable. For example: *AmountOfWater* or *PortionOfGold*. This is what we did before in the example of *Money* (*AmountOfMoney*).

Our guidelines adopt the capitalization rules proposed in [83] for acronyms that appear in the noun phrase of $\mathcal{G}_{1f}$. For ease of reference we reproduce the rules below. Examples of the application of the capitalization rules could be *ER-ConferenceEdition*, *CaiseConferenceEdition* and *OmgStandard*, assuming that *ER*, *CAISE* and *OMG*, respectively, are acronyms.

**Capitalization rules for acronyms [83]**

1. Capitalize both characters of a two-character acronym, except if it is the first word of a name in Camel case.
2. Capitalize only the first character of an acronym with three or more characters, except if it is the first word of a name in Camel case.
3. Do not capitalize any of the characters of any acronym, if it is the first word of a name in Camel case.

## A.2 Attributes

In this section we propose a naming guideline for attributes. Our guideline applies to attributes of classes, association classes and data types. We first summarize a literature review on similar naming guidelines and then we present our proposal.

### A.2.1 Literature review

[27] was one of the first works that analyzed the correspondence between parts of English sentences and attributes of entity and relationship types in ER diagrams. [13, pp. 7–20] indicated that "any attribute name must be simple and singular, and must not contain the name of the entity". [79, p. 99] suggested the same guideline, and added a guideline for Boolean valued attributes: "use either an adjective which suggests a true or false property or a name of the form *is_prop*, where *prop* denotes a property". IDEF1 highlighted that "an attribute name is a role name for the value class. An attribute role name is a name used to clarify the sense of the value class in the context of the class for which it is a property" [63, p. 88].

In the analysis of the use of attributes in 45 class diagrams [81, p. 7] found that “nouns are frequently used for attributes and 85% of the strings are single nouns, pair of nouns, triplets of nouns or a noun preceded by an adjective. Verbs are rarely used and if used, they are in the past tense. Only the verb *be* is found to be used in some cases in the present tense and only in the first position of verb-adjective strings and they usually denote attributes of type Boolean”. Concerning the style of attribute names, it has been often suggested to use the Camel case [1, 93, 120, 134].

## A.2.2 Naming guideline

Let $E::A:T\ [min..max]$ be an attribute named $A$ of type $T$ of the entity type $E$ and whose minimum and maximum multiplicities are *min* and *max*, respectively. For presentation purposes, in this appendix we assume that *min* is either 0 or 1, and that *max* is either *1* or * (unlimited); the extension to the general case is straightforward. The naming guideline $\mathcal{G}_2$ we propose for UML attributes is also in line with the previous work summarized above. However, we distinguish between two kinds of attributes (depending on whether $T$ is or is not the *Boolean* data type) and in both cases the pattern sentence takes into account the multiplicities *min* and *max*. We call $\mathcal{G}_2$ the guideline when $T$ is not *Boolean*, and $\mathcal{G}_{2'}$ when it is.

### Guideline for non-boolean attributes

If $T$ is not the *Boolean* data type, then the form of the name $\mathcal{G}_{2f}$ and the pattern sentence $\mathcal{G}_{2s}$ are:

**Non-Boolean Attributes**
Let $E::A:T\ [min..max]$ be a non-boolean attribute.

$\mathcal{G}_{2f}$ The name $A$ should be a noun phrase in singular form, written in the Camel case.

$\mathcal{G}_{2s}$ One of the following sentences must be grammatically well-formed and semantically meaningful:

- If *min*= 0 and *max*= 1:
  [A|An] *lower*($E$) may have [a|an] *lower*($A$).

*(continues in the next page...)*

- If $min = 0$ and $max > 1$:
  [A|An] *lower*($E$) may have zero or more *lower*(*plural*($A$)).
- If $min = max = 1$:
  [A|An] *lower*($E$) has [a|an] *lower*($A$).
- If $min > 0$ and $max > 1$:
  [A|An] *lower*($E$) has one or more *lower*(*plural*($A$)).

Note that *plural*($A$) is a function that gives the plural form of $A$.

In the example of Fig. A.2 there are seven non-boolean attributes, which follow the guideline suggested here. The sentences generated from them are:

A sale offer has a price.
A sale offer may have an expiration date.
An amount of money has a value.
A shop has a name.
A product has zero or more images.
A product has one or more colors.

As we indicated before, in this appendix we adopt the capitalization rules proposed in [83] for acronyms. An example of its application to attribute names is the attribute ISO code of *Currency* (Fig. A.2). According to the rules, the attribute must be written in Camel case as *isoCode* and the generated sentence is:

A currency has an ISO code.

Note that in this case we define *lower*(isoCode) as "ISO code".

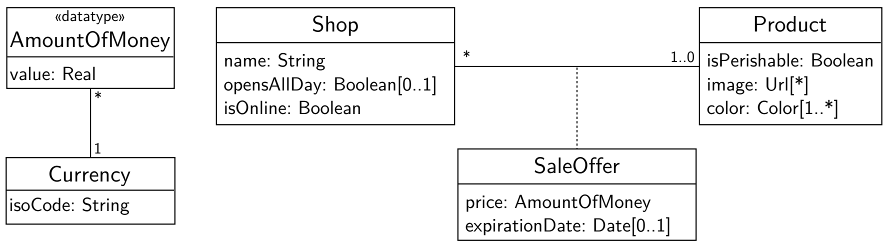


**Figure A.2.** Examples of boolean and non-boolean attributes.

### Guideline for boolean attributes

If $T$ is *Boolean*, we assume that *min* is either 0 or 1, and *max* equals 1. The form of the name $\mathcal{G}_{2'f}$ and the pattern sentence $\mathcal{G}_{2's}$ are:

**Boolean Attributes**
Let $E$::$A$:$Bool$ $[min..max]$ be a boolean attribute.

$\mathcal{G}_{2'f}$ The name $A$ should be a verb phrase in third-person singular form, in the Camel case.

$\mathcal{G}_{2's}$ The following sentence must be grammatically well-formed and semantically meaningful:

[A|An] *lower*($E$) *lower*(*withOrNeg*($A$)) [, or it may be unknown].

where the last optional fragment is included only if *min* is equal to zero.

The function *withOrNeg*($A$) extends $A$ with the insertion of "or *negative*($A$)" after the verb of $A$, where *negative*($A$) is the negative form of the verb of $A$. For example:

*withOrNeg*(isDerived) = isOrIsNotDerived
*withOrNeg*(hasChildren) = hasOrHasNotChildren

Figure A.2 shows three *Boolean* attributes, which follow the guideline suggested here. The sentences generated from them are:

A shop opens or does not open all day, or it may be unknown.
A shop is or is not online.
A product is or is not perishable.

## A.3 Associations

In this section we propose naming guidelines for UML associations. We distinguish between binary and $n$-ary associations because the guidelines are quite different. In each case, we first summarize the existing literature and then we present our guideline.

### A.3.1 Binary associations

#### Literature review

There is an abundant literature on naming binary associations. Existing proposals can be classified along two dimensions: what is named (association, roles, or both) and how it is named. In the simplest case, there is only one name, which is given to the association. An example is OWL [50, 131]. As stated in [81, p. 6], “the names of the associations are in most cases composed of a single verb in third person singular (V) or followed by a preposition (VP), a noun (VN), a preposition and a noun (VPN) or a preposition and a verb (VPV)”. It is also common the guideline “association names should start with a capital letter, since an association represents a classifier” [72, p. 153] [134, p. 81]. A variant is the Object-Relationship Model [46], a successor to ER [26], in which associations may have two names (verbs), one in each direction.

Other languages choose to name only the two roles. Examples are the CASE* Method [13], a variant of ER, and Object-Role Modeling (ORM) [58], a successor to NIAM [90]. There are two approaches on naming roles: noun-based and verb-based. In the noun-based approach, role names are nouns, while in the verb-based approach, role names are verbs [97]. In the CASE*Method, ORM, and in Executable UML [78, 117, 90], role names are verbs, but most methods suggest the use of nouns. In UML, if a role name is missing, it is assumed that it is the name of the corresponding entity type starting with a lowercase character [93, p. 19] [92, p. 17]

Finally, there are languages that allow naming both the association and the two roles. Prominent examples are ER and UML, for which up to three names can be given: the association and two roles. The most complete language is IDEF1X, which allows defining two names for the association and one name for each of the two roles [63].

#### Naming guideline

Let $R(p_1{:}E_1\ [min_1,max_1],\ p_2{:}E_2\ [min_2,max_2])$ be a binary association between entity types $E_1$ and $E_2$, playing roles $p_1$ and $p_2$, with multiplicities $[min_1,max_1]$

and $[min_2,max_2]$, respectively. In UML, the participants of an association are ordered. This order is graphically shown by a solid arrowhead next to the name $R$ [93, p. 41]. When the order is left to right or top to bottom, the arrowhead is usually omitted.

In UML there may be up to three explicit names related with this association (all are optional): the name of the association $R$ and the names of the two roles $p_1$ and $p_2$. The roles $p_1$ and $p_2$ always have an implicit name that is used only when there is no explicit name. The implicit name of a role is that of the corresponding entity type, starting with lower case. When $E_1 = E_2$ then either $p_1$, $p_2$, or both must have an explicit name. There are no implicit names for associations. Two different associations may have the same name. In UML, there are two special associations, *aggregation* and *composition*, that have predefined names and need not be named by the modelers.

The naming guideline $\mathcal{G}_3$ we propose for UML binary associations is in line with the consensus of the work summarized above. The guideline has two parts: one for the name of the association $R$ ($\mathcal{G}_3{}^a$), and one for the names of the roles $p_1$ and/or $p_2$ ($\mathcal{G}_3{}^r$). The guideline for the form of the name of the association $\mathcal{G}_{3f}^a$ and the pattern sentence $\mathcal{G}_{3s}^a$ is:

**Binary Associations**
Let $R(p_1{:}E_1\ [min_1,max_1],\ p_2{:}E_2\ [min_2,max_2])$ be a binary association.

$\mathcal{G}_{3f}^a$ The name of the association $R$ should be a verb phrase in third-person singular form, written in the Pascal case.

$\mathcal{G}_{3s}^a$ The sentences generated by the following pattern sentence must be grammatically well-formed and semantically meaningful:

- If $min_2 = 0$ and $max_2 = 1$:
  [A|An] *lower*$(E_1)$ *lower*$(R)$ at most one *lower*$(E_2)$.
- If $min_2 = 1$ and $max_2 = 1$:
  [A|An] *lower*$(E_1)$ *lower*$(R)$ [a|an] *lower*$(E_2)$.
- If $min_2 = 0$ and $max_2 = *$:
  [A|An] *lower*$(E_1)$ *lower*$(R)$ zero or more *lower*(*plural*$(E_2)$).
- If $min_2 = 1$ and $max_2 = *$:
  [A|An] *lower*$(E_1)$ *lower*$(R)$ one or more *lower*(*plural*$(E_2)$).

As we have seen, role names may be implicit or explicit. We distinguish between two types of explicit role names: *external* and *internal*. External role names must follow the guideline explained below. Internal role names are names needed for technical reasons, and are not intended to be verbalized. Internal role names may be distinguished by using a special prefix, such as an underscore.

The guideline we propose for the external name of roles is based on the view that a role can be seen as a non-boolean attribute of the entity type at the other end [107], and therefore the guideline for roles can be the same as that of attributes. More specifically, if $R(p_1{:}E_1\ [min_1{,}max_1],\ p_2{:}E_2\ [min_2{,}max_2])$ is a binary association, then the roles $p_1$ and $p_2$ are seen as the attributes:

- $E_2{::}p_1{:}E_1[min_1{,}max_1]$
- $E_1{::}p_2{:}E_2[min_2{,}max_2]$

and they must follow the guideline $\mathcal{G}_2$ that we proposed for the attributes. For ease of reference, we reproduce the guideline in the following, adapted to role names:

**External Role Names in Binary Associations**
Let $R(p_1{:}E_1\ [min_1{,}max_1],\ p_2{:}E_2\ [min_2{,}max_2])$ be a binary association.

$\mathcal{G}^r_{3f}$ Assuming $p_i$ $(i = 1, 2)$ is an external role name, the name $p_i$ should be a noun phrase in singular form, written in the Camel case.

$\mathcal{G}^r_{3s}$ One of the following sentences must be grammatically well-formed and semantically meaningful:

- If $min_i = 0$ and $max_i = 1$, $i \neq j$:
  [A|An] *lower*$(E_j)$ may have [a|an] *lower*$(p_i)$.
- If $min_i = 0$ and $max_i > 1$, $i \neq j$:
  [A|An] *lower*$(E_j)$ may have zero or more *lower*(*plural*$(p_i)$).
- If $min_i = max_i = 1$, $i \neq j$:
  [A|An] *lower*$(E_j)$ has [a|an] *lower*$(p_i)$.
- If $min_i > 0$ and $max_i > 1$, $i \neq j$:
  [A|An] *lower*$(E_j)$ has one or more *lower*(*plural*$(p_i)$).

Note that *plural*$(p_i)$ is a function that gives the plural form of $p_i$.

As an example, consider the nine associations shown in Fig. A.3. Four associations have an explicit association name that follow guideline $\mathcal{G}_{3f}^{a}$, and generate the following sentences:

A publisher publishes zero or more journals.
A paper cites zero or more papers.
A person writes zero or more papers.
A person was born in a town.

There are eight roles with an explicit name that follow guideline $\mathcal{G}_{3f}^{r}$ and generate the following sentences:

A paper has zero or more references.
A paper has one or more authors.
A town has zero or more natives.
A person has a place of birth.
A journal has one or more editors in chief.
A journal has one or more members of the editorial board.
An organization has zero or more employees.
A person may have an employer.

Note that in the example of Fig. A.3 the role named *_editedJournal* is internal and it is not intended to be verbalized. In this example, the name is needed to satisfy the UML constraint that an entity type (such as *Person*) cannot participate in two associations such that the other participants have the same role name. Without the internal name *_editedJournal*, *Journal* would have the same role name (the implicit name *journal*) in its two associations with *Person*.

Also note that in the example of Fig. A.3 there are two associations that are compositions (*Journal–Issue* and *Issue–paper*), and therefore they need not be named explicitly.

### A.3.2 n-ary associations

#### Literature review

$n$-ary associations (associations among three or more entity types) are much less common than binary ones, and there are very few published guidelines that deal with them. One exception is the Object-Role Modeling (ORM) language [58,

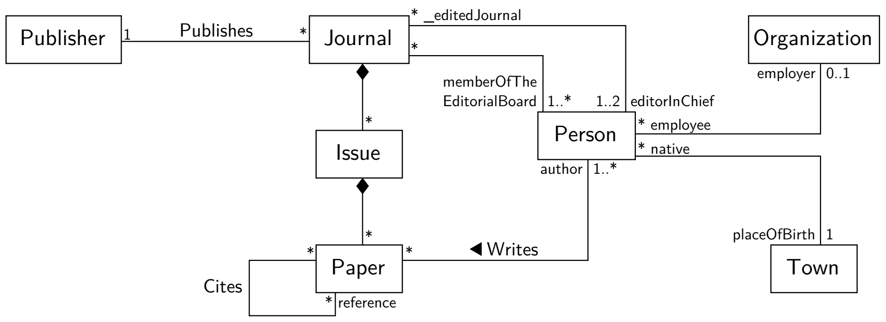


**Figure A.3.** Examples of associations and roles.

p. 84+], which proposed a combination of role names such that a pattern sentence can be generated from it. In UML, [97] observed that it is difficult to generate pattern sentences for $n$-ary associations and suggested to define instead explicit ones. However, these explicit pattern sentences should be defined as stereotyped comments in UML, which is not very practical. The guideline we propose below formalizes the ORM approach for the UML.

## Naming guideline

Let $R(p_1{:}E_1, \ldots, p_n{:}E_n)$ be an $n$-ary association ($n > 2$) between entity types $E_1, \ldots, E_n$, playing roles $p_1, \ldots, p_n$, respectively. In UML there may be up to $n+1$ explicit names related with this association (all are optional): the name of the association $R$ and the names of the roles $p_1, \ldots, p_n$. The roles $p_1, \ldots, p_n$ always have an implicit name that is used only when there is no explicit name. The implicit name of a role is that of the corresponding entity type, starting with lower case. When $E_i = E_j$ then either $p_i$ or $p_j$ or both must have an explicit name. There are no implicit names for $n$-ary associations. Two different associations may have the same name.

The naming guideline $\mathcal{G}_4$ we propose for UML $n$-ary associations builds upon the previous work summarized above. The guideline for the form of the name of the association $\mathcal{G}_{4f}$ and the pattern sentence $\mathcal{G}_{4s}$ are:

***n*-ary Associations**
Let $R(p_1{:}E_1, \ldots, p_n{:}E_n)$ be an $n$-ary association ($n > 2$).

$\mathcal{G}_{4f}$ The name of the association $R$ should be a verb phrase in third-person singular form, written in the Pascal case. The name must include $n-1$ substrings of the form $<p_i>$, one for each of the roles $p_2, \ldots, p_n$, where $p_i$ is the explicit name of the role or (if it has not one) the implicit name. Role names are written in the Camel case.

In UML the participants of an association are ordered, but for $n$-ary associations the order is not graphically shown. Therefore, in the above guideline it may be unknown which is the first participant ($p_1$). The convention we follow is that $p_1$ is the only participant that does not appear in $R$.

$\mathcal{G}_{4s}$ The sentences generated by the following pattern sentence must be grammatically well-formed and semantically meaningful:

[A|An] *lower*($E_1$) *lower*(*expanded*($R$))

In the sentence, *expanded*($R$) is a function that gives the result of substituting "[a|an] ($E_i$)" for each of the $<p_i>$ substrings. Note that the name of the roles are not taken into account in *expanded*($R$), but they may be embedded into the verb phrase of $R$.

The examples of Fig. A.4 illustrate the proposed guideline.

- *Supplies<product>To<project>* generates the pattern sentence:

  A vendor supplies a product to a project.

- *Sells<product>Of<vendor>In<country>* generates:

  A salesperson sells a product of a vendor in a country.

- *CanSubstitute<substitute>In<project>* generates:

  A product can substitute a product in a project.

## A.4 Invariants

In UML, an invariant is a static constraint targeted at an entity type [95]. The constraint may be defined as a boolean OCL expression in the context of that entity type. The expression must be true for each instance of that entity type. Invariants can be named [92]. For example:

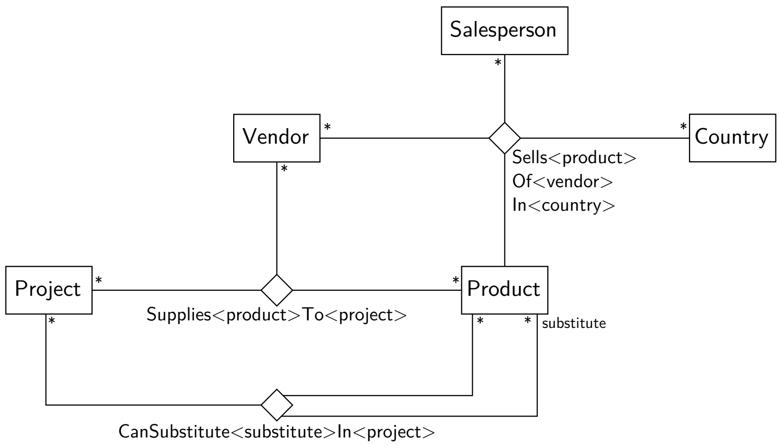


**Figure A.4.** Examples of *n*-ary associations.

```
context Department inv hasEnoughEmployees:
    self.numberOfEmployees > 50
```

is an invariant named *hasEnoughEmployees*.

In UML, untargeted constraints must be defined as invariants whose context is a singleton entity type. For example, in a system that manages a conference facility the untargeted constraint "There must be at least one large room" can be defined by the following invariant:

```
context ConferenceFacility inv hasAtLeastOneLargeRoom:
  Room.allInstances()->select(size = Size::large)
       ->size() > 0
```

where it is assumed that *ConferenceFacility* is a singleton entity type, that *size* is an attribute of *Room*, and that *Size* is an enumeration.

As far as we know, in the literature there are not proposals on naming guidelines for invariants. We have only found a general recommendation in [80, p. 899] stating that "you should label assertion clauses to make the text more readable". The guideline that we propose in the following tries to follow the style of the guidelines for the other elements.

### A.4.1 Naming guideline

The guideline we propose for naming the invariant $I$ has two parts: the form of the name $\mathcal{G}_{5f}$ and the pattern sentence $\mathcal{G}_{5s}$:

**Invariants**

$\mathcal{G}_{5f}$ The name of an invariant should be a verb phrase in third-person singular form. The name should be written either in the Camel or, if no confusion can arise[2], in the ordinary case.

$\mathcal{G}_{5s}$ If $I$ is the name of an invariant whose context is the entity type $E$, then the following sentence must be grammatically well-formed and semantically meaningful:

[A|An] *lower*($E$) *lower*($I$)

The following examples illustrate the application of the guideline (we omit the OCL expression):

```
context Person inv isIdentifiedByHisName

context Marriage inv is a relationship between a woman and a man

context Section inv isIdentifiedByCourseAndName

context Course inv cannotBeASuccessorOfItself

context Course inv consists of sections whose teachers are experts in
    the course's topics
```

which generate the following pattern sentences:

A person is identified by his name.
A marriage is a relationship between a woman and a man.
A section is identified by course and name.
A course cannot be a successor of itself.
A course consists of sections whose teachers are experts in the course's topics.

It can be seen that the names of the invariants tend to be long, because they constitute almost the complete sentence. However, such names convey an

[2]In OCL, the name of the invariant is the string starting after the keyword *inv* and ending in a colon.

accurate meaning of the invariant, which makes it easier to develop (and to understand) the corresponding OCL expressions. In some development contexts, it may even be possible to omit the OCL expression and define the name of the invariant only.

## A.5 Event Types

We now deal with the behavioral part of conceptual schemas. In general, a behavioral schema consists of a set of event types or system operations, and/or a set of state transition diagrams. In this section, we deal with event types or system operations, and in the next section we deal with state transition diagrams.

An event can be modeled as an instance of an event type [96] or as an invocation of a system operation [72]. Both event types and system operations have preconditions and postconditions. Event types have characteristics, which are the set of attributes and associations in which they participate. System operations have parameters. Event attributes and operation parameters are similar to entity attributes discussed in Sect. A.2, and can follow the same guidelines. Event associations are similar to ordinary associations, and can follow the guidelines presented in Sect. A.3. In the following we present our naming guidelines for event types, operations, preconditions and postconditions.

### A.5.1 Event Types

As far as we know, [97, pp. 254, 281] is the only proposal on naming guidelines for event types published in the literature. We adopt here the above mentioned proposal because it follows the style of the guidelines for the other elements and it is highly consistent with the examples of event types published in the literature.

Let *Ev* be an event type. The guideline we propose for naming *Ev* has two parts: the form of the name $\mathcal{G}_{6f}$ and the pattern sentence $\mathcal{G}_{6s}$:

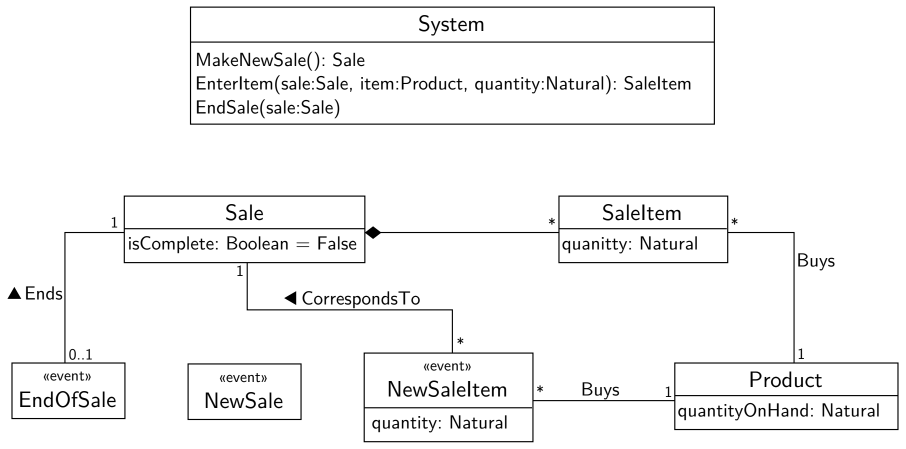


**Figure A.5.** Fragment of a structural schema and its behavioural specification using event types and system operations.

**Event Types**

$\mathcal{G}_{6f}$ The name of an event type should be a noun phrase whose head is a countable noun in singular form. The name should be written in the Pascal case. Note that this guideline is the same as $\mathcal{G}_{1f}$, which we proposed for entity types.

$\mathcal{G}_{6s}$ If *Ev* is the name of an event type, then the following sentence must be grammatically well-formed and semantically meaningful:

An instance of the event type named *Ev* is [a|an] *lower*(*Ev*) event.

When the event type is implicit from the context, the pattern sentence is just:

An instance of this event type is [a|an] *lower*(*Ev*) event.

Figure A.5 shows an example (taken from [72]) with three event types: *NewSale*, *NewSaleItem* and *EndOfSale*, which generate the sentences:

An instance of this event type is a new sale event.
An instance of this event type is a new sale item event.
An instance of this event type is an end of sale event.

### A.5.2 System Operations

There is a wide agreement on naming operations with verbs: "Modifier operations should be named with active verb phrases such as *Draw* or *Move*" [20]; "It improves clarity to start the name [...] with a verb [...] since it emphasizes these are commands or requests" [72, p. 178]; "Use verbs or verb phrases to name methods, in Pascal case" [82]; "Methods should be verbs, in mixed case" [120]; "Name operations with strong verbs" [10]. We adopt here this guideline.

Let *Op* be a system operation. The guideline we propose for naming *Op* has two parts: the form of the name $\mathcal{G}_{7f}$ and the pattern sentence $\mathcal{G}_{7s}$:

**System Operations**

$\mathcal{G}_{7f}$ The name of a system operation should be a verb phrase. The name should be written in the Pascal case.

$\mathcal{G}_{7s}$ If *Op* is the name of system operation, then the following sentence must be grammatically well-formed and semantically meaningful:

> An invocation of the operation named *Op* requests the system to perform the *Op* command.

When the operation is implicit from the context, the pattern sentence is just:

> An invocation of this operation requests the system to perform the *Op* command.

Note that in the pattern sentence we have written *Op*, and not $lower(Op)$ as it could be expected from previous sentences. The reason is that most operation names given in practice are compact descriptions of the command to be performed, and it does not seem possible to give a general rule for its verbalization.

As an example, Figure A.5 shows also the three system operations, named *MakeNewSale*, *EnterItem* and *EndSale*, corresponding to the above mentioned event types. The sentences generated from those names are:

An invocation of this op. requests the system to perform the MakeNewSale command.
An invocation of this op. requests the system to perform the EnterItem command.
An invocation of this op. requests the system to perform the EndSale command.

### A.5.3 Preconditions

A precondition is a condition that must be satisfied by the event characteristics or operation parameters and/or information base when the event occurs or the system operation is invoked [97]. In UML, preconditions can be named and formally specified in OCL.

As far as we know, in the literature there are no proposals on naming guidelines for preconditions. We have found only a general recommendation: "You should label assertion clauses to make the text more readable" [80, p. 899].

The guideline we propose for naming *Pre* tries to follow the style of the guidelines for the other elements. It has two parts, the form of the name $\mathcal{G}_{8f}$ and the pattern sentence $\mathcal{G}_{8s}$:

**Preconditions**

Let *Pre* be a precondition of event type *Ev* or of system operation *Op*.

$\mathcal{G}_{8f}$ The name of a precondition should be a phrase. The name should be written in either the Camel or the ordinary case.

$\mathcal{G}_{8s}$ If *Pre* is the name of a precondition, then the following sentence must be grammatically well-formed and semantically meaningful:

[Before the occurrence of *lower*(*Ev*) | Before the operation *lower*(*Op*) is invoked], *lower*(*Pre*).

When the event type or system operation is implicit from the context, the pattern sentence is just:

[Before the occurrence of this event | Before this operation is invoked], *lower*(*Pre*).

In the example of Fig. A.5, event type *NewSaleItem* (or the equivalent system operation *EnterItem*) has two preconditions with names: "the sale is not complete", and "there is enough stock":

```
context NewSaleItem::effect()
   pre the sale is not complete:
       not sale.isComplete
   pre there is enough stock:
       quantity >= product.quantityOnHand
```

which generate the sentences:

Before the occurrence of this event, the sale is not complete.

Before the occurrence of this event, there is enough stock.

### A.5.4 Postconditions

The effect of events or system operations is described by means of one or more postconditions. A postcondition is a condition that must be satisfied by the information base after the occurrence of the event or the execution of the system operation [72]. In general, a postcondition may also include assertions on outputs produced (queries or communications) [97]. In UML, postconditions can be named and formally specified in OCL.

As far as we know, in the literature there are not proposals on naming guidelines for postconditions. The one that we propose in the following tries to follow the style of the guidelines for the other elements.

The guideline we propose for naming *Post* has two parts: the form of the name $\mathcal{G}_{9f}$ and the pattern sentence $\mathcal{G}_{9s}$:

**Postconditions**

Let *Post* be a postcondition of event type *Ev* or of system operation *Op*.

$\mathcal{G}_{9f}$ The name of a postcondition should be a phrase. The name should be written in either the Camel or the ordinary case.

$\mathcal{G}_{9s}$ If *Post* is the name of a postcondition, then the following sentence must be grammatically well-formed and semantically meaningful:

[After the occurrence of *lower*(*Ev*) occurs | After the execution of *lower*(*Op*)], *lower*(*Post*).

When the event type or system operation is implicit from the context, the pattern sentence is just:

[After the occurrence of this event | After the execution of this system operation], *lower*(*Post*).

In the example of Fig. A.5, event type *NewSale* (or the equivalent system operation *MakeNewSale*) has one postcondition named "a sale has been created":

```
context NewSale::effect()
  post a sale has been created:
    s.oclIsNew() and s.oclIsTypeOf(Sale)
```

which produces the sentence:

> After the occurrence of this event, a sale has been created.

Similarly, the event type *NewSaleItem* (or the equivalent system operation *EnterItem*) has two postconditions with names: “a sale item has been created”, and “the product’s quantity on hand has decreased”, which generate the sentences:

> After the occurrence of this event, a sale item has been created.
> After the occurrence of this event, the product's quantity on hand has decreased.

## A.6 States

In UML, a state is a “condition or situation during the life of an object in which it satisfies some condition, performs some activity or waits for some event” [108, p. 597]. States are the basic components of the popular state machine or statecharts diagrams.

As far as we know, [97, p. 302] is the most detailed proposal on naming guidelines for states published in the literature. [78, p. 152] acknowledges that “it is important to choose good, meaningful state names”, but no specific guideline is suggested. [10, p. 105] indicates that “state names should be simple but descriptive”. [134, p. 112] suggests that “state names start with an initial uppercase letter”. We adopt here the above mentioned proposal because it follows the style of the guidelines for the other elements and it is highly consistent with the examples of states published in the literature.

### A.6.1 Naming guideline

The guideline we propose for naming $S$ has two parts: the form of the name $\mathcal{G}_{10f}$ and the pattern sentence $\mathcal{G}_{10s}$:

**States**

Let $S$ be a state of entity type $E$.

$\mathcal{G}_{10f}$ The name of a state should be an adjective, an adjectival phrase or other noun modifier, written in the Pascal case.

$\mathcal{G}_{10s}$ If $S$ is the name of a state of the entity type $E$, then at least one of the following sentences must be grammatically well-formed and semantically meaningful:

[A|An] *lower*$(E)$ is *lower*$(S)$.
[A|An] *lower*$(E)$ is in the state of *lower*$(S)$.

The following examples, taken from the literature, illustrate the application of the guideline:

- MicrowaveOven: *ReadyToCook*, *DoorOpen*, *CookingInterrupted*, *Cooking*, *CookingComplete*. [78]
- Patient: *Entering, Admitted, UnderObservation, Released.* [39]
- PhoneCall: *Idle, DialTone, Dialing, Connecting, Busy,* ... [108]

A few examples of the sentences they generate are:

A microwave oven is ready to cook.
A microwave oven is in the state of door open.
A patient is entering.
A patient is under observation.
A phone call is busy.